Reinforcement Learning Toolbox™

Reference

MATLAB®

R2023a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Reinforcement Learning Toolbox™ Reference

Trademarks

Patents

Revision History

March 2019	Online only	New for Version 1.0 (Release 2019a)
September 2019	Online only	Revised for Version 1.1 (Release 2019b)
March 2020	Online only	Revised for Version 1.2 (Release 2020a)
September 2020	Online only	Revised for Version 1.3 (Release 2020b)
March 2021	Online only	Revised for Version 2.0 (Release 2021a)
September 2021	Online only	Revised for Version 2.1 (Release 2021b)
March 2022	Online only	Revised for Version 2.2 (Release 2022a)
September 2022	Online only	Revised for Version 2.3 (Release 2022b)
March 2023	Online only	Revised for Version 2.4 (Release 2023a)

Contents

1

Apps

Reinforcement Learning Designer

Design, train, and simulate reinforcement learning agents

Description

The **Reinforcement Learning Designer** app lets you design, train, and simulate agents for existing environments.

Using this app, you can:

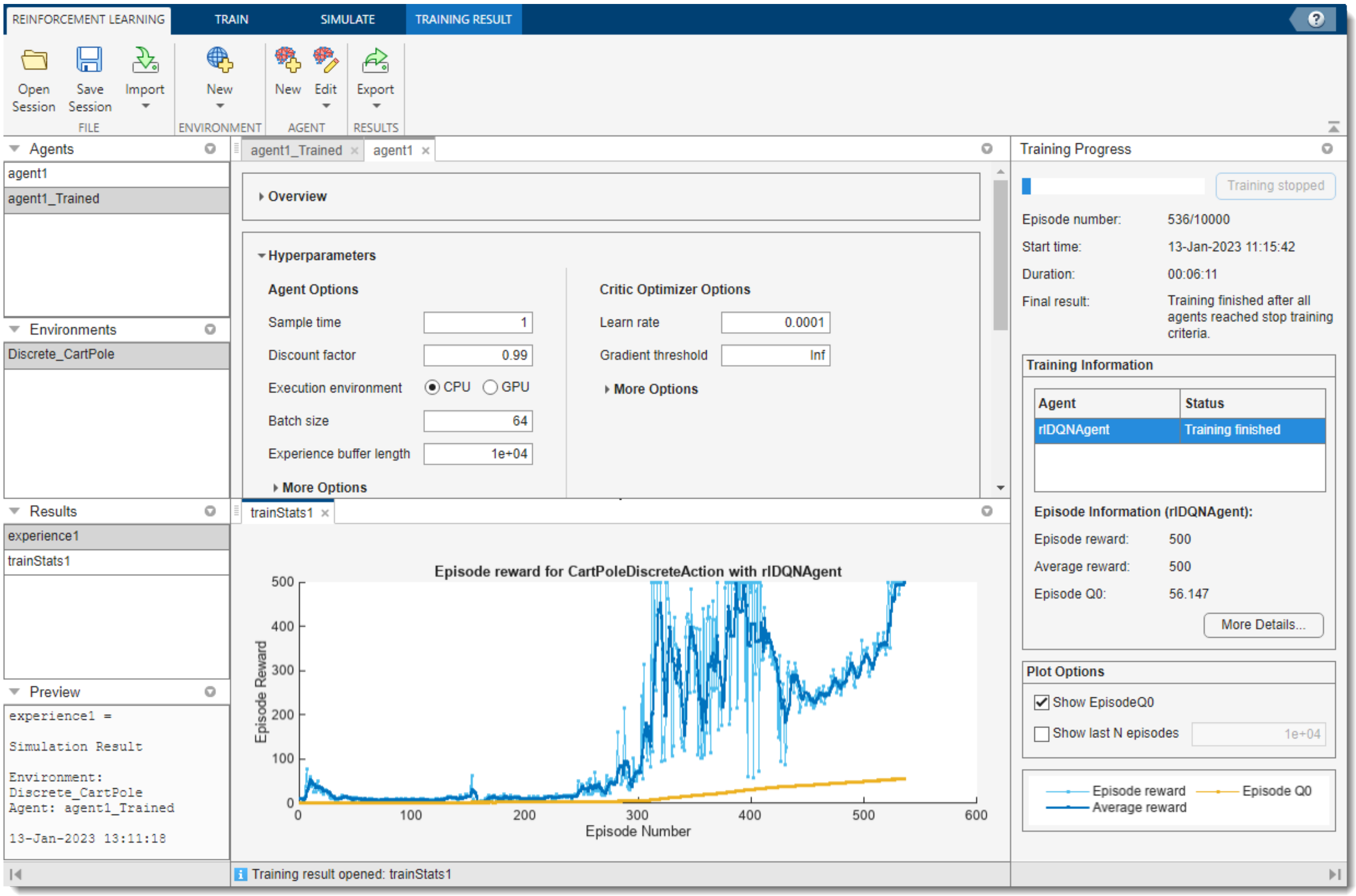
- Import an existing environment from the MATLAB® workspace or create a predefined environment.
- Automatically create or import an agent for your environment (DQN, DDPG, TD3, SAC, and PPO agents are supported).
- Train and simulate the agent against the environment.
- Analyze simulation results and refine your agent parameters.
- Export the final agent to the MATLAB workspace for further use and deployment.

Limitations

The following features are not supported in the **Reinforcement Learning Designer** app.

- Multi-agent systems
- Q, SARSA, PG, AC, and SAC agents
- Custom agents
- Agents relying on table or custom basis function representations

If your application requires any of these features then design, train, and simulate your agent at the command line.



```
experience1 =

Simulation Result

Environment:
Discrete_CartPole
Agent: agent1_Trained

13-Jan-2023 13:11:18
```

Agent Options

Sample time: 1

Discount factor: 0.99

Execution environment: ● CPU ○ GPU

Batch size: 64

Experience buffer length: 1e+04

Critic Optimizer Options

Learn rate: 0.0001

Gradient threshold: Inf

Episode number: 536/10000

Start time: 13-Jan-2023 11:15:42

Duration: 00:06:11

Final result: Training finished after all agents reached stop training criteria.

Agent	Status
rlDQNAgent	Training finished

Episode Information (rlDQNAgent):

Episode reward: 500

Average reward: 500

Episode Q0: 56.147

☑ Show EpisodeQ0

□ Show last N episodes 1e+04

Training result opened: trainStats1

Open the Reinforcement Learning Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.
- MATLAB command prompt: Enter `reinforcementLearningDesigner`.

Examples

- “Design and Train Agent Using Reinforcement Learning Designer”
- “Train Reinforcement Learning Agents”

Programmatic Use

`reinforcementLearningDesigner` opens the **Reinforcement Learning Designer** app. You can then import an environment and start the design process, or open a saved design session.

Version History

Introduced in R2021a

See Also

Apps

Deep Network Designer

Functions

`analyzeNetwork`

Objects

`rlDQNAgent` | `rlDDPGAgent` | `rlTD3Agent` | `rlPPOAgent`

Topics

"Design and Train Agent Using Reinforcement Learning Designer"
"Train Reinforcement Learning Agents"
"Create or Import MATLAB Environments in Reinforcement Learning Designer"
"Create or Import Simulink Environments in Reinforcement Learning Designer"
"Create Agents Using Reinforcement Learning Designer"
"What Is Reinforcement Learning?"
"Reinforcement Learning Agents"

2

Functions

accelerate

Package: `rl.function`

Option to accelerate computation of gradient for approximator object based on neural network

Syntax

```
newAppx = accelerate(oldAppx,useAcceleration)
```

Description

`newAppx = accelerate(oldAppx,useAcceleration)` returns the new neural-network-based function approximator object `newAppx`, which has the same configuration as the original object, `oldAppx`, and the option to accelerate the gradient computation set to the logical value `useAcceleration`.

Examples

Accelerate Gradient Computation for a Q-Value Function

Create observation and action specification objects (or alternatively use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment). For this example, define an observation space with two channels. The first channel carries an observation from a continuous four-dimensional space. The second carries a discrete scalar observation that can be either zero or one. Finally, the action space is a three-dimensional vector in a continuous action space.

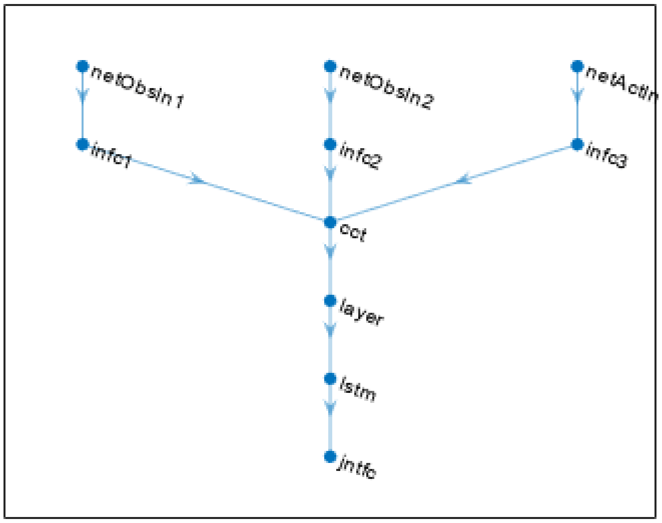
```
obsInfo = [rlNumericSpec([4 1])
           rlFiniteSetSpec([0 1])];

actInfo =  rlNumericSpec([3 1]);
```

To approximate the Q-value function within the critic, create a recurrent deep neural network. The output layer must be a scalar expressing the value of executing the action given the observation.

Define each network path as an array of layer objects. Get the dimensions of the observation and action spaces from the environment specification objects, and specify a name for the input layers, so you can later explicitly associate them with the appropriate environment channel. Since the network is recurrent, use `sequenceInputLayer` as the input layer and include an `lstmLayer` as one of the other network layers.

```
% Define paths
inPath1 = [ sequenceInputLayer( ...
                prod(obsInfo(1).Dimension), ...
                Name="netObsIn1")
            fullyConnectedLayer(5,Name="infc1") ];

inPath2 = [ sequenceInputLayer( ...
                prod(obsInfo(2).Dimension), ...
                Name="netObsIn2")
            fullyConnectedLayer(5,Name="infc2") ];
```

```
inPath3 = [ sequenceInputLayer( ...
                prod(actInfo(1).Dimension), ...
                Name="netActIn")
            fullyConnectedLayer(5,Name="infc3") ];

% Concatenate 3 previous layer outputs along dim 1
jointPath = [ concatenationLayer(1,3,Name="cct")
              tanhLayer
              lstmLayer(8,"OutputMode","sequence")
              fullyConnectedLayer(1,Name="jntfc") ];

% Add layers to network object
net = layerGraph;
net = addLayers(net,inPath1);
net = addLayers(net,inPath2);
net = addLayers(net,inPath3);
net = addLayers(net,jointPath);

% Connect layers
net = connectLayers(net,"infc1","cct/in1");
net = connectLayers(net,"infc2","cct/in2");
net = connectLayers(net,"infc3","cct/in3");

% Plot network
plot(net)
```



```
% Convert to dlnetwork and display number of weights
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 832
   Inputs:
      1   'netObsIn1'   Sequence input with 4 dimensions
      2   'netObsIn2'   Sequence input with 1 dimensions
      3   'netActIn'    Sequence input with 3 dimensions
```

Create the critic with `rlQValueFunction`, using the network, and the observation and action specification objects.

```
critic = rlQValueFunction(net, ...
            obsInfo, ...
            actInfo, ...
            ObservationInputNames=["netObsIn1","netObsIn2"], ...
            ActionInputNames="netActIn");
```

To return the value of the actions as a function of the current observation, use `getValue` or `evaluate`.

```
val = evaluate(critic, ...
                { rand(obsInfo(1).Dimension), ...
                  rand(obsInfo(2).Dimension), ...
                  rand(actInfo(1).Dimension) })
```

```
val = 1x1 cell array
    {[0.1360]}
```

When you use `evaluate`, the result is a single-element cell array containing the value of the action in the input, given the observation.

```
val{1}
```

```
ans = single
    0.1360
```

Calculate the gradients of the sum of the three outputs with respect to the inputs, given a random observation.

```
gro = gradient(critic,"output-input", ...
                { rand(obsInfo(1).Dimension) , ...
                  rand(obsInfo(2).Dimension) , ...
                  rand(actInfo(1).Dimension) } )
```

```
gro=3×1 cell array
    {4x1 single}
    {[  0.0243]}
    {3x1 single}
```

The result is a cell array with as many elements as the number of input channels. Each element contains the derivatives of the sum of the outputs with respect to each component of the input channel. Display the gradient with respect to the element of the second channel.

```
gro{2}
```

```
ans = single
    0.0243
```

Obtain the gradient with respect of five independent sequences, each one made of nine sequential observations.

```
gro_batch = gradient(critic,"output-input", ...
                { rand([obsInfo(1).Dimension 5 9]) , ...
                  rand([obsInfo(2).Dimension 5 9]) , ...
                  rand([actInfo(1).Dimension 5 9]) } )
```

```
gro_batch=3×1 cell array
    {4x5x9 single}
    {1x5x9 single}
    {3x5x9 single}
```

Display the derivative of the sum of the outputs with respect to the third observation element of the first input channel, after the seventh sequential observation in the fourth independent batch.

```
gro_batch{1}(3,4,7)
```

```
ans = single
    0.0108
```

Set the option to accelerate the gradient computations.

```
critic = accelerate(critic,true);
```

Calculate the gradients of the sum of the outputs with respect to the parameters, given a random observation.

```
grp = gradient(critic,"output-parameters", ...
                { rand(obsInfo(1).Dimension) , ...
                  rand(obsInfo(2).Dimension) , ...
                  rand(actInfo(1).Dimension) } )
```

```
grp=11×1 cell array
    { 5x4  single                                                }
    { 5x1  single                                                }
    { 5x1  single                                                }
    { 5x1  single                                                }
    { 5x3  single                                                }
    { 5x1  single                                                }
    {32x15 single                                                }
    {32x8  single                                                }
    {32x1  single                                                }
    {[0.0444 0.1280 -0.1560 0.0193 0.0262 0.0453 -0.0186 -0.0651]}
    {[                                                         1]}
```

Each array within a cell contains the gradient of the sum of the outputs with respect to a group of parameters.

```
grp_batch = gradient(critic,"output-parameters", ...
                { rand([obsInfo(1).Dimension 5 9]) , ...
                  rand([obsInfo(2).Dimension 5 9]) , ...
                  rand([actInfo(1).Dimension 5 9]) } )
```

```
grp_batch=11×1 cell array
    { 5x4  single                                                 }
    { 5x1  single                                                 }
    { 5x1  single                                                 }
    { 5x1  single                                                 }
    { 5x3  single                                                 }
    { 5x1  single                                                 }
    {32x15 single                                                 }
    {32x8  single                                                 }
    {32x1  single                                                 }
    {[2.6325 10.1821 -14.0886 0.4162 2.0677 5.3991 0.3904 -8.9048]}
    {[                                                         45]}
```

If you use a batch of inputs, `gradient` uses the whole input sequence (in this case nine steps), and all the gradients with respect to the independent batch dimensions (in this case five) are added together. Therefore, the returned gradient always has the same size as the output from `getLearnableParameters`.

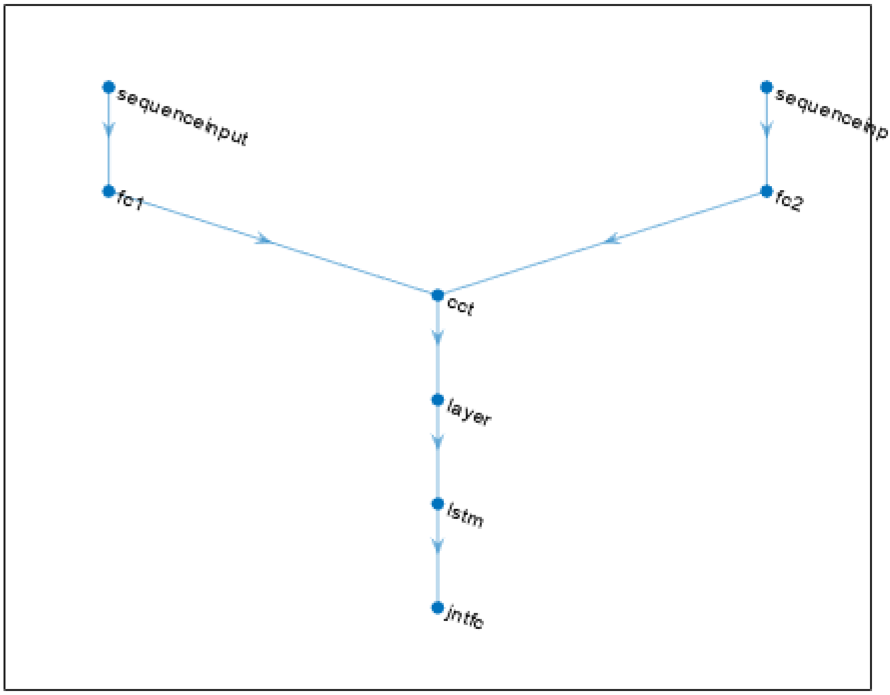
Accelerate Gradient Computation for a Discrete Categorical Actor

Create observation and action specification objects (or alternatively use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment). For this example, define an observation space with two channels. The first channel carries an observation from a continuous four-dimensional space. The second carries a discrete scalar observation that can be either zero or one. Finally, the action space consist of a scalar that can be `-1`, `0`, or `1`.

```
obsInfo = [rlNumericSpec([4 1])
           rlFiniteSetSpec([0 1])];

actInfo =  rlFiniteSetSpec([-1 0 1]);
```

Create a deep neural network to be used as approximation model within the actor. The output layer must have three elements, each one expressing the value of executing the corresponding action, given the observation. To create a recurrent neural network, use `sequenceInputLayer` as the input layer and include an `lstmLayer` as one of the other network layers.

```
% Define paths
inPath1 = [ sequenceInputLayer(prod(obsInfo(1).Dimension))
            fullyConnectedLayer(prod(actInfo.Dimension),Name="fc1") ];

inPath2 = [ sequenceInputLayer(prod(obsInfo(2).Dimension))
            fullyConnectedLayer(prod(actInfo.Dimension),Name="fc2") ];

% Concatenate previous paths outputs along first dimension
jointPath = [ concatenationLayer(1,2,Name="cct")
              tanhLayer;
              lstmLayer(8,OutputMode="sequence");
              fullyConnectedLayer( ...
                prod(numel(actInfo.Elements)), ...
                Name="jntfc"); ];

% Add layers to network object
net = layerGraph;
net = addLayers(net,inPath1);
net = addLayers(net,inPath2);
net = addLayers(net,jointPath);

% Connect layers
net = connectLayers(net,"fc1","cct/in1");
net = connectLayers(net,"fc2","cct/in2");

% Plot network
plot(net)
```



```
% Convert to dlnetwork and display the number of weights
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 386

   Inputs:
      1   'sequenceinput'     Sequence input with 4 dimensions
      2   'sequenceinput_1'   Sequence input with 1 dimensions
```

Since each element of the output layer must represent the probability of executing one of the possible actions the software automatically adds a `softmaxLayer` as a final output layer if you do not specify it explicitly.

Create the actor with `rlDiscreteCategoricalActor`, using the network and the observations and action specification objects. When the network has multiple input layers, they are automatically associated with the environment observation channels according to the dimension specifications in `obsInfo`.

```
actor = rlDiscreteCategoricalActor(net, obsInfo, actInfo);
```

To return a vector of probabilities for each possible action, use `evaluate`.

```
[prob,state] = evaluate(actor, ...
                { rand(obsInfo(1).Dimension) , ...
                  rand(obsInfo(2).Dimension) });
prob{1}
```

```
ans = 3x1 single column vector

    0.3403
    0.3114
    0.3483
```

To return an action sampled from the distribution, use `getAction`.

```
act = getAction(actor, ...
                { rand(obsInfo(1).Dimension) , ...
                  rand(obsInfo(2).Dimension) });
act{1}
```

```
ans = 1
```

Set the option to accelerate the gradient computations.

```
actor = accelerate(actor,true);
```

Each array within a cell contains the gradient of the sum of the outputs with respect to a group of parameters.

```
grp_batch = gradient(actor,"output-parameters", ...
                        { rand([obsInfo(1).Dimension 5 9]) , ...
                          rand([obsInfo(2).Dimension 5 9])} )
```

```
grp_batch=9×1 cell array
    {[-4.0250e-09 -4.5360e-09 -4.1017e-09 -4.2893e-09]}
    {[                                    -1.1114e-08]}
    {[                                    -2.3304e-08]}
    {[                                    -4.5522e-08]}
    {32x2 single                                      }
    {32x8 single                                      }
    {32x1 single                                      }
    { 3x8 single                                      }
    { 3x1 single                                      }
```

If you use a batch of inputs, the `gradient` uses the whole input sequence (in this case nine steps), and all the gradients with respect to the independent batch dimensions (in this case five) are added together. Therefore, the returned gradient always has the same size as the output from `getLearnableParameters`.

Input Arguments

`oldAppx` — Function approximator object

function approximator object

Function approximator object, specified as:

- rlValueFunction,
- rlQValueFunction,
- rlVectorQValueFunction,
- rlDiscreteCategoricalActor,
- rlContinuousDeterministicActor,
- rlContinuousGaussianActor,
- rlContinuousDeterministicTransitionFunction,
- rlContinuousGaussianTransitionFunction,
- rlContinuousDeterministicRewardFunction,
- rlContinuousGaussianRewardFunction,
- rlIsDoneFunction.

useAcceleration — Option to use acceleration for gradient computations
false (default) | true

Option to use acceleration for gradient computations, specified as a logical value. When useAcceleration is true, the gradient computations are accelerated by optimizing and caching some inputs needed by the automatic-differentiation computation graph. For more information, see "Deep Learning Function Acceleration for Custom Training Loops".

Output Arguments

newAppx — Actor or critic
approximator object

New actor or critic, returned as an approximator object with the same type as oldAppx but with the gradient acceleration option set to useAcceleration.

Version History

Introduced in R2022a

See Also

Functions
evaluate | gradient | getLearnableParameters

Objects
rlValueFunction | rlQValueFunction | rlVectorQValueFunction | rlContinuousDeterministicActor | rlDiscreteCategoricalActor | rlContinuousGaussianActor | rlContinuousDeterministicTransitionFunction | rlContinuousGaussianTransitionFunction | rlContinuousDeterministicRewardFunction | rlContinuousGaussianRewardFunction | rlIsDoneFunction

Topics
"Train Reinforcement Learning Policy Using Custom Training Loop"
"Create Custom Reinforcement Learning Agents"

allExperiences

Package: `rl.replay`

Return all experiences in replay memory buffer

Syntax

```
experiences = allExperiences(buffer)
experience = allExperiences(buffer,ConcatenateMode=mode)
```

Description

`experiences = allExperiences(buffer)` returns all experiences stored in experience buffer `buffer` as individual experiences, each with a batch size of 1 and a sequence length of 1.

`experience = allExperiences(buffer,ConcatenateMode=mode)` returns experiences concatenated along the dimension specified by `mode`. You can concatenate experiences along the batch dimension or the sequence dimension.

Examples

Extract All Experiences from Replay Memory Buffer

Define observation specifications for the environment. For this example, assume that the environment has two observation channels: one channel with two continuous observations and one channel with a three-valued discrete observation

```
obsContinuous = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[1;5]);
obsDiscrete = rlFiniteSetSpec([1 2 3]);
obsInfo = [obsContinuous obsDiscrete];
```

Define action specifications for the environment. For this example, assume that the environment has a single action channel with one continuous action in a specified range.

```
actInfo = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[5;10]);
```

Create an experience buffer with a maximum length of 5,000.

```
buffer = rlReplayMemory(obsInfo,actInfo,5000);
```

Append a sequence of 10 random experiences to the buffer.

```
for i = 1:10
    experience(i).Observation = ...
        {obsInfo(1).UpperLimit.*rand(2,1) randi(3)};
    experience(i).Action = {actInfo.UpperLimit.*rand(2,1)};
    experience(i).NextObservation = ...
```

```
        {obsInfo(1).UpperLimit.*rand(2,1) randi(3)};
    experience(i).Reward = 10*rand(1);
    experience(i).IsDone = 0;
end

append(buffer,experience);
```

After appending experiences to the buffer, you extract all of the experiences from the buffer. Extract all of the experiences as individual experiences, each with a batch size of 1 and sequence size of 1.

```
experience = allExperiences(buffer)
```

```
experience=10×1 struct array with fields:
    Observation
    Action
    NextObservation
    Reward
    IsDone
```

Alternatively, you can extract all of the experiences as a single experience batch.

```
expBatch = allExperiences(buffer,ConcatenateMode="batch")
```

```
expBatch = struct with fields:
        Observation: {[2x1x10 double]  [1x1x10 double]}
             Action: {[2x1x10 double]}
             Reward: [9.5751 9.1574 7.4313 8.2346 1.8687 1.6261 5.0596 2.5428 3.5166 5.6782]
    NextObservation: {[2x1x10 double]  [1x1x10 double]}
             IsDone: [0 0 0 0 0 0 0 0 0 0]
```

Input Arguments

buffer — Experience buffer

rlReplayMemory object | rlPrioritizedReplayMemory object | rlHindsightReplayMemory object | rlHindsightPrioritizedReplayMemory object

Experience buffer, specified as one of the following replay memory objects.

- rlReplayMemory
- rlPrioritizedReplayMemory
- rlHindsightReplayMemory
- rlHindsightPrioritizedReplayMemory

mode — Concatenation mode

"none" (default) | "batch" | "sequence"

Concatenation mode specified as a one of the following values.

- "none" — Return experience as *N* individual experiences, each with a batch size of 1 and a sequence length of 1.
- "batch" — Return experience as a single batch with a sequence length of 1.

- `"sequence"` — Return experience as a single sequence with a batch size of 1.

Output Arguments

`experience` — All buffered experiences
structure array | structure

All *N* buffered experiences, returned as a structure array or structure. When `mode` is:

- `"none"`, `experience` is returned as a structure array of length *N*, where each element contains one buffered experience (`batchSize` = `1` and `SequenceLength` = `1`).
- `"batch"`, `experience` is returned as a structure. Each field of `experience` contains all buffered experiences concatenated along the batch dimension (`batchSize` = *N* and `SequenceLength` = `1`).
- `"sequence"`, `experience` is returned as a structure. Each field of `experience` contains all buffered experiences concatenated along the batch dimension (`batchSize` = `1` and `SequenceLength` = *N*).

`experience` contains the following fields.

`Observation` — Observation
cell array

Observation, returned as a cell array with length equal to the number of observation specifications specified when creating the buffer. Each element of `Observation` contains a D_O-by-`batchSize`-by-`SequenceLength` array, where D_O is the dimension of the corresponding observation specification.

`Action` — Agent action
cell array

Agent action, returned as a cell array with length equal to the number of action specifications specified when creating the buffer. Each element of `Action` contains a D_A-by-`batchSize`-by-`SequenceLength` array, where D_A is the dimension of the corresponding action specification.

`Reward` — Reward value
scalar | array

Reward value obtained by taking the specified action from the observation, returned as a 1-by-1-by-`SequenceLength` array.

`NextObservation` — Next observation
cell array

Next observation reached by taking the specified action from the observation, returned as a cell array with the same format as `Observation`.

`IsDone` — Termination signal
integer | array

Termination signal, returned as a 1-by-1-by-`SequenceLength` array of integers. Each element of `IsDone` has one of the following values.

- `0` — This experience is not the end of an episode.

- 1 — The episode terminated because the environment generated a termination signal.
- 2 — The episode terminated by reaching the maximum episode length.

Version History

Introduced in R2022b

See Also

Objects
`rlReplayMemory` | `rlPrioritizedReplayMemory` | `rlHindsightReplayMemory` | `rlHindsightPrioritizedReplayMemory`

append

Package: `rl.replay`

Append experiences to replay memory buffer

Syntax

```
append(buffer,experience)
append(buffer,experience,dataSourceID)
```

Description

`append(buffer,experience)` appends the experiences in `experience` to the replay memory `buffer`.

`append(buffer,experience,dataSourceID)` appends experiences for the specified data source to the replay memory buffer.

Examples

Create Experience Buffer

Define observation specifications for the environment. For this example, assume that the environment has a single observation channel with three continuous signals in specified ranges.

```
obsInfo = rlNumericSpec([3 1],...
    LowerLimit=0,...
    UpperLimit=[1;5;10]);
```

Define action specifications for the environment. For this example, assume that the environment has a single action channel with two continuous signals in specified ranges.

```
actInfo = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[5;10]);
```

Create an experience buffer with a maximum length of 20,000.

```
buffer = rlReplayMemory(obsInfo,actInfo,20000);
```

Append a single experience to the buffer using a structure. Each experience contains the following elements: current observation, action, next observation, reward, and is-done.

For this example, create an experience with random observation, action, and reward values. Indicate that this experience is not a terminal condition by setting the `IsDone` value to 0.

```
exp.Observation = {obsInfo.UpperLimit.*rand(3,1)};
exp.Action = {actInfo.UpperLimit.*rand(2,1)};
exp.Reward = 10*rand(1);
exp.NextObservation = {obsInfo.UpperLimit.*rand(3,1)};
exp.IsDone = 0;
```

Before appending experience to the buffer, you can validate whether the experience is compatible with the buffer. The `validateExperience` function generates an error if the expereince is incompatible with the buffer.

```
validateExperience(buffer,exp)
```

Append the experience to the buffer.

```
append(buffer,exp);
```

You can also append a batch of experiences to the experience buffer using a structure array. For this example, append a sequence of 100 random experiences, with the final experience representing a terminal condition.

```
for i = 1:100
    expBatch(i).Observation = {obsInfo.UpperLimit.*rand(3,1)};
    expBatch(i).Action = {actInfo.UpperLimit.*rand(2,1)};
    expBatch(i).Reward = 10*rand(1);
    expBatch(i).NextObservation = {obsInfo.UpperLimit.*rand(3,1)};
    expBatch(i).IsDone = 0;
end
expBatch(100).IsDone = 1;

validateExperience(buffer,expBatch)

append(buffer,expBatch);
```

After appending experiences to the buffer, you can sample mini-batches of experiences for training of your RL agent. For example, randomly sample a batch of 50 experiences from the buffer.

```
miniBatch = sample(buffer,50);
```

You can sample a horizon of data from the buffer. For example, sample a horizon of 10 consecutive experiences with a discount factor of 0.95.

```
horizonSample = sample(buffer,1,...
    NStepHorizon=10,...
    DiscountFactor=0.95);
```

The returned sample includes the following information.

- `Observation` and `Action` are the observation and action from the first experience in the horizon.
- `NextObservation` and `IsDone` are the next observation and termination signal from the final experience in the horizon.
- `Reward` is the cumulative reward across the horizon using the specified discount factor.

You can also sample a sequence of consecutive experiences. In this case, the structure fields contain arrays with values for all sampled experiences.

```
sequenceSample = sample(buffer,1,...
    SequenceLength=20);
```

Create Experience Buffer with Multiple Observation Channels

Define observation specifications for the environment. For this example, assume that the environment has two observation channels: one channel with two continuous observations and one channel with a three-valued discrete observation.

```
obsContinuous = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[1;5]);
obsDiscrete = rlFiniteSetSpec([1 2 3]);
obsInfo = [obsContinuous obsDiscrete];
```

Define action specifications for the environment. For this example, assume that the environment has a single action channel with one continuous action in a specified range.

```
actInfo = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[5;10]);
```

Create an experience buffer with a maximum length of 5,000.

```
buffer = rlReplayMemory(obsInfo,actInfo,5000);
```

Append a sequence of 50 random experiences to the buffer.

```
for i = 1:50
    exp(i).Observation = ...
        {obsInfo(1).UpperLimit.*rand(2,1) randi(3)};
    exp(i).Action = {actInfo.UpperLimit.*rand(2,1)};
    exp(i).NextObservation = ...
        {obsInfo(1).UpperLimit.*rand(2,1) randi(3)};
    exp(i).Reward = 10*rand(1);
    exp(i).IsDone = 0;
end

append(buffer,exp);
```

After appending experiences to the buffer, you can sample mini-batches of experiences for training of your RL agent. For example, randomly sample a batch of 10 experiences from the buffer.

```
miniBatch = sample(buffer,10);
```

Input Arguments

`buffer` — Experience buffer

`rlReplayMemory` object | `rlPrioritizedReplayMemory` object | `rlHindsightReplayMemory` object | `rlHindsightPrioritizedReplayMemory` object

Experience buffer, specified as one of the following replay memory objects.

- `rlReplayMemory`
- `rlPrioritizedReplayMemory`
- `rlHindsightReplayMemory`
- `rlHindsightPrioritizedReplayMemory`

experience — Experience to append to the buffer

structure | structure array

Experience to append to the buffer, specified as a structure or structure array with the following fields.

Observation — Observation

cell array

Observation, specified as a cell array with length equal to the number of observation specifications specified when creating the buffer. The dimensions of each element in `Observation` must match the dimensions in the corresponding observation specification.

Action — Agent action

cell array

Action taken by the agent, specified as a cell array with length equal to the number of action specifications specified when creating the buffer. The dimensions of each element in `Action` must match the dimensions in the corresponding action specification.

Reward — Reward value

scalar

Reward value obtained by taking the specified action from the starting observation, specified as a scalar.

NextObservation — Next observation

cell array

Next observation reached by taking the specified action from the starting observation, specified as a cell array with the same format as `Observation`.

IsDone — Termination signal

`0` | `1` | `2`

Termination signal, specified as one of the following values.

- `0` — This experience is not the end of an episode.
- `1` — The episode terminated because the environment generated a termination signal.
- `2` — The episode terminated by reaching the maximum episode length.

dataSourceID — Data source index

`0` (default) | nonnegative integer | array of nonnegative integers

Data source index, specified as a nonnegative integer or array of nonnegative integers.

If `experience` is a scalar structure, specify `dataSourceID` as a scalar integer.

If `experience` is a structure array, specify `dataSourceID` as an array with length equal to the length of `experience`. You can specify different data source indices for each element of `experience`. If all elements in `experience` come from the same data source, you can specify `dataSourceID` as a scalar integer.

Version History

Introduced in R2022a

See Also

Functions
`sample`

Objects
`rlReplayMemory` | `rlPrioritizedReplayMemory` | `rlHindsightReplayMemory` | `rlHindsightPrioritizedReplayMemory`

barrierPenalty

Logarithmic barrier penalty value for a point with respect to a bounded region

Syntax

```
p = barrierPenalty(x,xmin,xmax)
p = barrierPenalty( ___ ,maxValue,curvature)
```

Description

p = barrierPenalty(x,xmin,xmax) calculates the nonnegative (logarithmic barrier) penalty vector p for the point x with respect to the region bounded by xmin and xmax. p has the same dimension as x. This syntax uses the default values of 1 and 0.1 for the maxValue and curvature parameters of the barrier function, respectively.

p = barrierPenalty(___ ,maxValue,curvature) specifies both the maxValue and curvature parameters of the barrier function. If maxValue is an empty matrix its default value is used. Likewise if curvature is an empty matrix or it is omitted, its default value is used.

Examples

Calculate Logarithmic Barrier Penalty for a Point

This example shows how to use the logarithmic barrierPenalty function to calculate the barrier penalty for a given point, with respect to a bounded region.

Calculate the penalty value for the point 0.1 within the interval [-2,2] using default values for the maximum value and curvature parameters.

```
barrierPenalty(0.1,-2,2)
```

```
ans = 2.5031e-04
```

Calculate the penalty value for the point 4 outside the interval [-2,2].

```
barrierPenalty(4,-2,2)
```

```
ans = 1
```

Calculate the penalty value for the point 4 outside the interval [-2,2], using a maximum value parameter of 5.

```
barrierPenalty(4,-2,2,5)
```

```
ans = 5
```

Calculate the penalty value for the point 0.1 inside the interval [-2,2], using a curvature parameter of 0.5.

```
barrierPenalty(0.1,-2,2,5,0.5)
```

```
ans = 0.0013
```

Calculate the penalty value for the point [-2,0,4] with respect to the box defined by [0,1], [-1,1], and [-2,2] along the x, y, and z dimensions, respectively, using the default value for maximum value and a curvature parameter of 0.

```
barrierPenalty([-2 0 4],[0 -1 -2],[1 1 2],1,0)
```

```
ans = 3×1

     1
     0
     1
```

Visualize Penalty Values for an Interval

Create a vector of 1001 equidistant points distributed between -5 and 5.

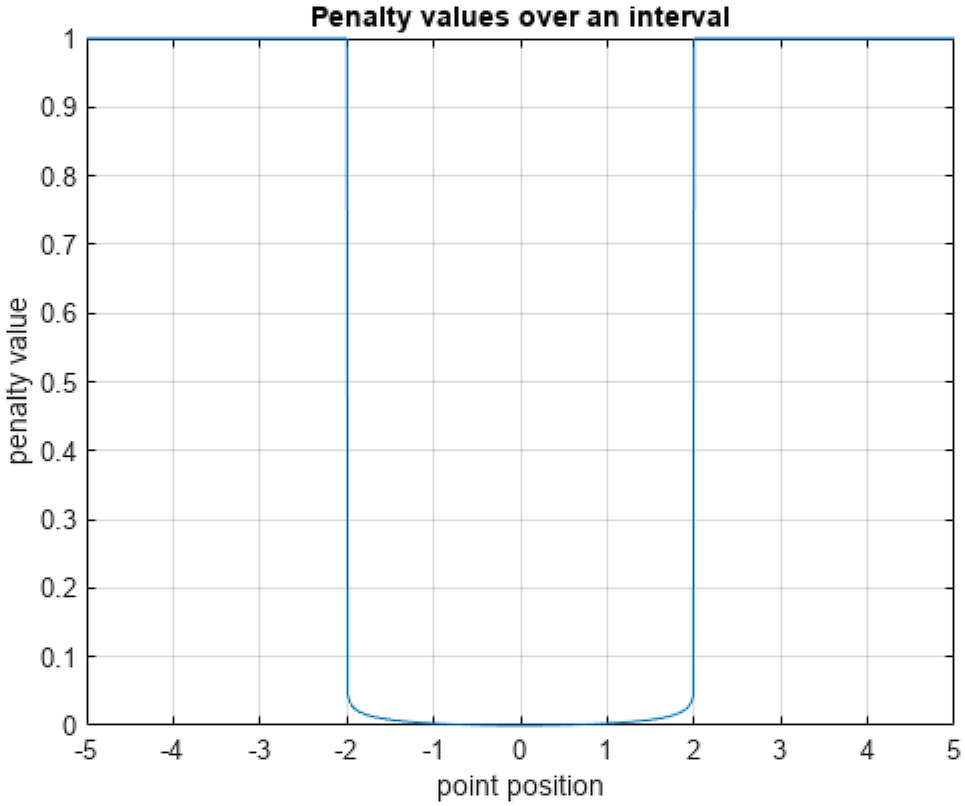
```
x = -5:0.01:5;
```

Calculate penalties for all the points in the vector, using the default value for the maximum value parameter and a value of 0.01 for the curvature parameter.

```
p = barrierPenalty(x,-2,2,1,0.01);
```

Plot the points and add grid, labels and title.

```
plot(x,p)
grid
xlabel("point position");
ylabel("penalty value");
title("Penalty values over an interval");
```



Input Arguments

x — Point for which the penalty is calculated

scalar | vector | matrix

Point for which the penalty is calculated, specified as a numeric scalar, vector or matrix.

Example: `[0.5; 1.6]`

`xmin` — Lower bounds

scalar | vector | matrix

Lower bounds for `x`, specified as a numeric scalar, vector or matrix. To use the same minimum value for all elements in `x`, specify `xmin` as a scalar.

Example: `-1`

`xmax` — Upper bounds

scalar | vector | matrix

Upper bounds for `x`, specified as a numeric scalar, vector or matrix. To use the same maximum value for all elements in `x`, specify `xmax` as a scalar.

Example: 2

`maxValue` — Maximum value parameter of the barrier function

`1` (default) | nonnegative scalar

Maximum value parameter of the barrier function, specified as a scalar.

Example: 2

`curvature` — Curvature parameter of the barrier function
`0.1` (default) | nonnegative scalar

Curvature parameter of the barrier function, specified as a scalar.

Example: 0.2

Output Arguments

`p` — Penalty value
nonnegative vector

Penalty value, returned as a vector of nonnegative elements. Each element `p`$_i$ depends on the position of `x`$_i$ with respect to the interval specified by `xmin`$_i$ and `xmax`$_i$. The barrier penalty function returns the value

$$p(x) = \min\left(p_{\mathrm{max}}, C\left(\log\left(0.25(x_{\mathrm{max}} - x_{\mathrm{min}})^2\right) - \log((x - x_{\mathrm{min}})(x_{\mathrm{max}} - x_{\mathrm{min}}))\right)\right)$$

when $x_{min} < x < x_{max}$, and `maxValue` otherwise. Here, C is the argument `curvature`, and p_{max} is the argument `maxValue`. Note that for positive values of C the returned penalty value is always positive. If C is zero, then the returned penalty is zero inside the interval defined by the bounds, and p_{max} outside this interval. If x is multidimensional, then the calculation is applied independently on each dimension. Penalty functions are typically used to generate negative rewards when constraints are violated, such as in `generateRewardFunction`.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`generateRewardFunction` | `exteriorPenalty` | `hyperbolicPenalty`

Topics

"Generate Reward Function from a Model Predictive Controller for a Servomotor"
"Generate Reward Function from a Model Verification Block for a Water Tank System"
"Define Reward Signals"

bus2RLSpec

Create reinforcement learning data specifications for elements of a Simulink bus

Syntax

```
specs = bus2RLSpec(busName)
specs = bus2RLSpec(busName,Name,Value)
```

Description

`specs = bus2RLSpec(busName)` creates a set of reinforcement learning data specifications from the Simulink® bus object specified by `busName`. One specification element is created for each leaf element in the corresponding Simulink bus. Use these specifications to define actions and observations for a Simulink reinforcement learning environment.

`specs = bus2RLSpec(busName,Name,Value)` specifies options for creating specifications using one or more `Name,Value` pair arguments.

Examples

Create an observation specification object from a bus object

This example shows how to use the function `bus2RLSpec` to create an observation specification object from a Simulink® bus object.

Create a bus object.

```
obsBus = Simulink.Bus();
```

Create three elements in the bus and specify their names.

```
obsBus.Elements(1) = Simulink.BusElement;
obsBus.Elements(1).Name = "sin_theta";
obsBus.Elements(2) = Simulink.BusElement;
obsBus.Elements(2).Name = "cos_theta";
obsBus.Elements(3) = Simulink.BusElement;
obsBus.Elements(3).Name = "dtheta";
```

Create the observation specification objects using the Simulink bus object.

```
obsInfo = bus2RLSpec("obsBus");
```

You can then use `obsInfo`, together with the corresponding Simulink model, to create a reinforcement learning environment. For an example, see “Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal”.

Create an action specification object from a bus object

This example shows how to call the function bus2RLSpec using name and value pairs to create an action specification object from a Simulink® bus object.

Create a bus object.

```
actBus = Simulink.Bus();
```

Create one element in the bus and specify the name.

```
actBus.Elements(1) = Simulink.BusElement;
actBus.Elements(1).Name = "actuator";
```

Create the observation specification objects using the Simulink bus object.

```
actInfo = bus2RLSpec("actBus","DiscreteElements",{"actuator",[-1 1]});
```

This specifies that the "actuator" bus element can carry two possible values, -1, and 1.

You can then use actInfo, together with the corresponding Simulink model, to create a reinforcement learning environment. Specifically the function that creates the environment uses *actInfo* to determine the right bus output of the agent block.

For an example, see “Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal”.

Input Arguments

busName — Name of Simulink bus object
string | character vector

Name of Simulink bus object, specified as a string or character vector.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: 'DiscreteElements',{'force',[-5 0 5]} sets the 'force' bus element to be a discrete data specification with three possible values, -5, 0, and 5

Model — Name of Simulink model
string | character vector

Name of the Simulink model, specified as the comma-separated pair consisting of 'Model' and a string or character vector. Specify the model name when the bus object is defined in the model global workspace (for example, in a data dictionary) instead of the MATLAB workspace.

BusElementNames — Names of bus leaf elements
string array

Names of bus leaf elements for which to create specifications, specified as the comma-separated pair consisting of 'BusElementNames' and a string array. To create observation specifications for a subset of the elements in a Simulink bus object, specify `BusElementNames`. If you do not specify `BusElementNames`, a data specification is created for each leaf element in the bus.

Note Do not specify `BusElementNames` when creating specifications for action signals. The RL Agent block must output the full bus signal.

`DiscreteElements` — Finite values for discrete bus elements

cell array of name-value pairs

Finite values for discrete bus elements, specified as the comma-separated pair consisting of `'DiscreteElements'` and a cell array of name-value pairs. Each name-value pair consists of a bus leaf element name and an array of discrete values. The specified discrete values must be castable to the data type of the specified action signal.

If you do not specify discrete values for an element specification, the element is continuous.

Example: `'ActionDiscretElements',{'force',[-10 0 10],'torque',[-5 0 5]}` specifies discrete values for the `'force'` and `'torque'` leaf elements of a bus action signal.

Output Arguments

`specs` — Data specifications

`rlNumericSpec` object | `rlFiniteSetSpec` object | array of data specification objects

Data specifications for reinforcement learning actions or observations, returned as one of the following:

- `rlNumericSpec` object for a single continuous bus element
- `rlFiniteSetSpec` object for a single discrete bus element
- Array of data specification objects for multiple bus elements

By default, all data specifications for bus elements are `rlNumericSpec` objects. To create a discrete specification for one or more bus elements, specify the element names using the `DiscreteElements` name-value pair.

Version History

Introduced in R2019a

See Also

Functions

`rlSimulinkEnv` | `createIntegratedEnv`

Objects

`rlNumericSpec` | `rlFiniteSetSpec`

Blocks

RL Agent

Topics
“Create Simulink Reinforcement Learning Environments”

cancel

Package: rl.env

Cancel unfinished reinforcement learning environment simulations on workers

Syntax

```
cancel(F)
```

Description

cancel(F) stops all simulations scheduled on workers corresponding to the elements of F that are in the 'pending', 'queued', or 'running' state. For elements in the 'finished' state, no action is taken.

Examples

Defer Outputs of Simulations Running on Workers

This example shows how to use Future objects and their methods fetchNext, fetchOutput, cancel, and wait to defer output retrieval for environment simulations running on workers, monitor the status of ongoing simulations, fetch outputs of completed simulations, cancel ongoing simulations, or wait for ongoing simulations to complete.

Load a predefined environment and a suitable agent. For this example use both the environment and agent described in “Train AC Agent to Balance Cart-Pole System”.

```
env = rlPredefinedEnv("CartPole-Discrete");
load("MATLABCartpoleAC.mat","agent")
```

Start a parallel pool and set up the environment so that it simulates on workers.

```
pp = parpool(2);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 2 workers.
```

```
setup(env,UseParallel=true);
```

To display the simulation completion times, start a timer.

```
tic
```

Schedule six simulation to run on the available workers. At the beginning of the simulation, the reset function of the cart-pole environment sets the initial angle of the pole to a random position in the neighborhood of zero (the upward position). This randomization ensures that each simulation is different.

```
for i=1:6
    ftr(i) = runEpisode(env,agent,CleanupPostSim=false);
end
```

Each element of the Future array ftr represents a scheduled simulation.

```
ftr
```

```
ftr=1×6 object
  1×6 Future array with properties:

    Read
    State
    Diary
    ID
```

Display the state of each simulation.

```
ftr.State
```

```
ans =
'running'

ans =
'running'

ans =
'queued'

ans =
'queued'

ans =
'queued'

ans =
'queued'
```

Two simulations are ongoing while the others are queued.

Use fetchNext with a timeout of 0.1 seconds to retrieve results for simulations that complete within that time (if any).

```
[idx,out] = fetchNext(ftr,0.1)
```

```
idx =

     []


out =

     []
```

Both the outputs are empty, which means that none of the four simulation has completed yet.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   0
```

```
ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

Use `fetchNext` without any timeout to wait until an unretrieved simulation output becomes available and then return the results.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 2

out = struct with fields:
    SimulationInfo: [1×1 struct]
         AgentData: [1×1 struct]
```

Display the state of the simulations.

```
ftr.State
```

```
ans =
'finished'

ans =
'finished'

ans =
'running'

ans =
'running'

ans =
'queued'

ans =
'queued'
```

As expected, the first two simulations, which were running in parallel on the two workers, are finished, while the next two, which were previously queued, are now running, and the final two are still queued.

Display the time taken for the first two simulations to complete.

```
toc
```

```
Elapsed time is 10.451231 seconds.
```

Note that once the results from a simulation has been already retrieved, any attempt to use `fetchNext` to retrieve it again, such as in `fetchNext(ftr(2))`, will result in an error. To retrieve the results from a `Future` object that has already been read, you can use `fetchOuptuts`, such as in `fetchOutputs(ftr(2))`.

Retrieve the next available result, and display the time elapsed since the simulations started.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 1

out = struct with fields:
    SimulationInfo: [1×1 struct]
         AgentData: [1×1 struct]
```

```
toc
```

```
Elapsed time is 11.945070 seconds.
```

As expected, `fetchNext` promptly returns the results from the second simulation, since it was already available.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   1


ans = logical
   1


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

Cancel the last simulation.

```
cancel(ftr(6))
```

Wait for the fourth simulation to complete. The `wait` function blocks the command prompt until the fourth simulation is completed.

```
wait(ftr(4))
```

Display the elapsed time since the simulations started.

```
toc
```

```
Elapsed time is 12.414076 seconds.
```

Display the state of the simulations.

```
ftr.State
```

```
ans = 
'finished'

ans = 
'finished'

ans = 
'finished'

ans = 
'finished'

ans = 
'running'

ans = 
'finished'
```

The status of the last element of the array, for which the simulation has been canceled, is classified as `'finished'`.

Since any attempt to retrieve results from a simulation that has been canceled will result in an error, remove the canceled object from the array.

```
ftr(6)=[]
```

```
ftr=1×5 object
  1×5 Future array with properties:

    Read
    State
    Diary
    ID
```

Use `fetchOutputs` to wait until all remaining simulations are completed and then retrieve all outputs.

```
outs = fetchOutputs(ftr)
```

```
outs=5×1 struct array with fields:
    SimulationInfo
    AgentData
```

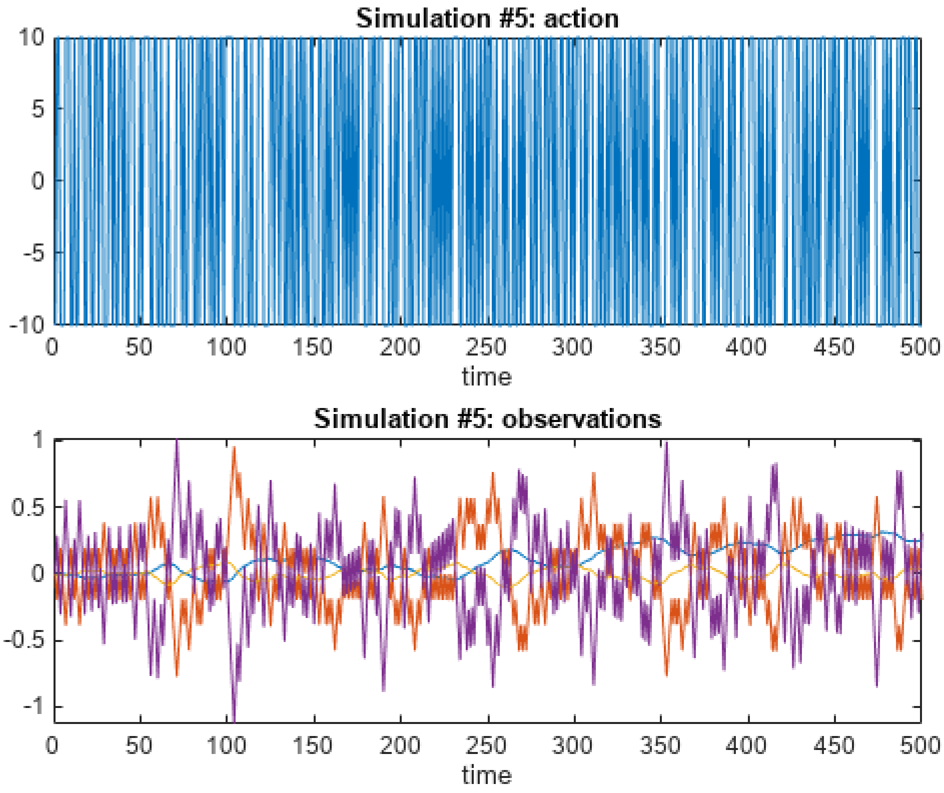
Display the elapsed time.

```
toc
```

```
Elapsed time is 16.265069 seconds.
```

Plot the action and observations from the fifth simulation.

```
figure

subplot(2,1,1);
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Action]))
title('Simulation #5: action');
xlabel('time');

subplot(2,1,2)
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Observation]))
title('Simulation #5: observations')
xlabel('time');
```



Clear the array of `Future` objects, the environment, and delete the parallel pool (this is the reverse order in which they were created).

```
clear ftr
clear env
delete(pp)
```

Input Arguments

F — Future simulation outputs
`Future` object | array of `Future` objects

Future simulation outputs, specified as a `Future` object or as an array of `Future` objects. To create an element of `F`, set the `UseParallel` property of a reinforcement learning environment to `true`, and then use `runEpisode` to simulate an agent or a policy within this environment. Assign the element of `F` to the output of `runEpisode`.

Version History

Introduced in R2022a

See Also

Objects
`Future` | `Future` | `Simulink.Simulation.Future`

Functions
`fetchNext` | `fetchOutputs` | `wait` | `runEpisode` | `setup` | `cleanup` | `reset`

Topics
"Custom Training Loop with Simulink Action Noise"
"Train Reinforcement Learning Policy Using Custom Training Loop"
"Train Agents Using Parallel Computing and GPUs"

cleanup

Package: `rl.env`

Clean up reinforcement learning environment or data logger object

Syntax

`cleanup(env)`

`cleanup(lgr)`

Description

When you define a custom training loop for reinforcement learning, you can simulate an agent or policy against an environment using the `runEpisode` function. Use the `cleanup` function to clean up the environment after running simulations using multiple calls to `runEpisode`. To clean up the environment after each simulation, you can configure `runEpisode` to automatically call the `cleanup` function at the end of each episode.

Also use `cleanup` to perform clean up tasks for a `FileLogger` or `MonitorLogger` object after logging data within a custom training loop.

Environment Objects

`cleanup(env)` cleans up the specified reinforcement learning environment after running multiple simulations using `runEpisode`.

Data Logger Objects

`cleanup(lgr)` cleans up the specified data logger object after logging data within a custom training loop. This task might involve for example transferring any remaining data from `lgr` internal memory to a logging target (either“Log Data to Disk in a Custom Training Loop” on page 2-36 a MAT file or a `trainingProgressMonitor` object).

Examples

Simulate Environment and Agent

Create a reinforcement learning environment and extract its observation and action specifications.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

To approximate the Q-value function withing the critic, use a neural network. Create a network as an array of layer objects.

```
net = [...
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(2)
    softmaxLayer];
```

Convert the network to a `dlnetwork` object and display the number of learnable parameters (weights).

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 770

   Inputs:
      1   'input'   4 features
```

Create a discrete categorical actor using the network.

```
actor = rlDiscreteCategoricalActor(net,obsInfo,actInfo);
```

Check your actor with a random observation.

```
act = getAction(actor,{rand(obsInfo.Dimension)})
```

```
act = 1x1 cell array
    {[-10]}
```

Create a policy object from the actor.

```
policy = rlStochasticActorPolicy(actor);
```

Create an experience buffer.

```
buffer = rlReplayMemory(obsInfo,actInfo);
```

Set up the environment for running multiple simulations. For this example, configure the training to log any errors rather than send them to the command window.

```
setup(env,StopOnError="off")
```

Simulate multiple episodes using the environment and policy. After each episode, append the experiences to the buffer. For this example, run 100 episodes.

```
for i = 1:100
    output = runEpisode(env,policy,MaxSteps=300);
    append(buffer,output.AgentData.Experiences)
end
```

Clean up the environment.

```
cleanup(env)
```

Sample a mini-batch of experiences from the buffer. For this example, sample 10 experiences.

```
batch = sample(buffer,10);
```

You can then learn from the sampled experiences and update the policy and actor.

Log Data to Disk in a Custom Training Loop

This example shows how to log data to disk when training an agent using a custom training loop.

Create a `FileLogger` object using `rlDataLogger`.

```
flgr = rlDataLogger();
```

Set up the logger object. This operation initializes the object performing setup tasks such as, for example, creating the directory to save the data files.

```
setup(flgr);
```

Within a custom training loop, you can now store data to the logger object memory and write data to file.

For this example, store random numbers to the file logger object, grouping them in the variables `Context1` and `Context2`. When you issue a write command, a MAT file corresponding to an iteration and containing both variables is saved with the name specified in `flgr.LoggingOptions.FileNameRule`, in the folder specified by `flgr.LoggingOptions.LoggingDirectory`.

```
for iter = 1:10

    % Store three random numbers in memory
    % as elements of the variable "Context1"
    for ct = 1:3
        store(flgr, "Context1", rand, iter);
    end

    % Store a random number in memory
    % as the variable "Context2"
    store(flgr, "Context2", rand, iter);

    % Write data to file every 4 iterations
    if mod(iter,4)==0
        write(flgr);
    end

end
```

Clean up the logger object. This operation performs clean up tasks like for example writing to file any data still in memory.

```
cleanup(flgr);
```

Input Arguments

env — Reinforcement learning environment
environment object | ...

Reinforcement learning environment, specified as one of the following objects.

- `rlFunctionEnv` — Environment defined using custom functions.
- `SimulinkEnvWithAgent` — Simulink environment created using `rlSimulinkEnv` or `createIntegratedEnv`
- `rlMDPEnv` — Markov decision process environment
- `rlNeuralNetworkEnvironment` — Environment with deep neural network transition models
- Predefined environment created using `rlPredefinedEnv`
- Custom environment created from a template (`rlCreateEnvTemplate`)

If `env` is a `SimulinkEnvWithAgent` object and the associated Simulink model is configured to use fast restart, then `cleanup` terminates the model compilation.

`lgr` — Date logger object
`FileLogger` object | `MonitorLogger` object | ...

Data logger object, specified as either a `FileLogger` or a `MonitorLogger` object.

Version History

Introduced in R2022a

See Also

Functions
`runEpisode` | `setup` | `reset` | `store` | `write`

Objects
`rlFunctionEnv` | `rlMDPEnv` | `SimulinkEnvWithAgent` | `rlNeuralNetworkEnvironment` | `FileLogger` | `MonitorLogger`

Topics
"Custom Training Loop with Simulink Action Noise"

createGridWorld

Create a two-dimensional grid world for reinforcement learning

Syntax

```
GW = createGridWorld(m,n)
GW = createGridWorld(m,n,moves)
```

Description

`GW = createGridWorld(m,n)` creates a grid world `GW` of size `m`-by-`n` with default actions of `['N';'S';'E';'W']`.

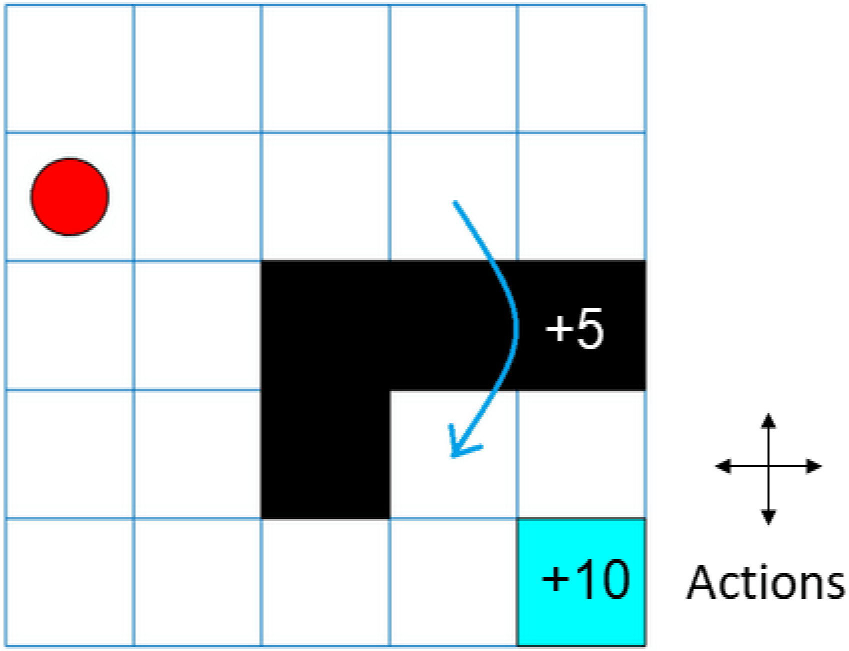
`GW = createGridWorld(m,n,moves)` creates a grid world `GW` of size `m`-by-`n` with actions specified by `moves`.

Examples

Create Grid World Environment

For this example, consider a 5-by-5 grid world with the following rules:

1. A 5-by-5 grid world bounded by borders, with 4 possible actions (North = 1, South = 2, East = 3, West = 4).
2. The agent begins from cell [2,1] (second row, first column).
3. The agent receives reward +10 if it reaches the terminal state at cell [5,5] (blue).
4. The environment contains a special jump from cell [2,4] to cell [4,4] with +5 reward.
5. The agent is blocked by obstacles in cells [3,3], [3,4], [3,5] and [4,3] (black cells).
6. All other actions result in -1 reward.



First, create a `GridWorld` object using the `createGridWorld` function.

```
GW = createGridWorld(5,5)
```

```
GW =
  GridWorld with properties:

                GridSize: [5 5]
            CurrentState: "[1,1]"
                  States: [25x1 string]
                 Actions: [4x1 string]
                       T: [25x25x4 double]
                       R: [25x25x4 double]
          ObstacleStates: [0x1 string]
          TerminalStates: [0x1 string]
    ProbabilityTolerance: 8.8818e-16
```

Now, set the initial, terminal and obstacle states.

```
GW.CurrentState = '[2,1]';
GW.TerminalStates = '[5,5]';
GW.ObstacleStates = ["[3,3]";"[3,4]";"[3,5]";"[4,3]"];
```

Update the state transition matrix for the obstacle states and set the jump rule over the obstacle states.

```
updateStateTranstionForObstacles(GW)
GW.T(state2idx(GW,"[2,4]"),:,:) = 0;
GW.T(state2idx(GW,"[2,4]"),state2idx(GW,"[4,4]"),:) = 1;
```

Next, define the rewards in the reward transition matrix.

```
nS = numel(GW.States);
nA = numel(GW.Actions);
```

```
GW.R = -1*ones(nS,nS,nA);
GW.R(state2idx(GW,"[2,4]"),state2idx(GW,"[4,4]"),:) = 5;
GW.R(:,state2idx(GW,GW.TerminalStates),:) = 10;
```

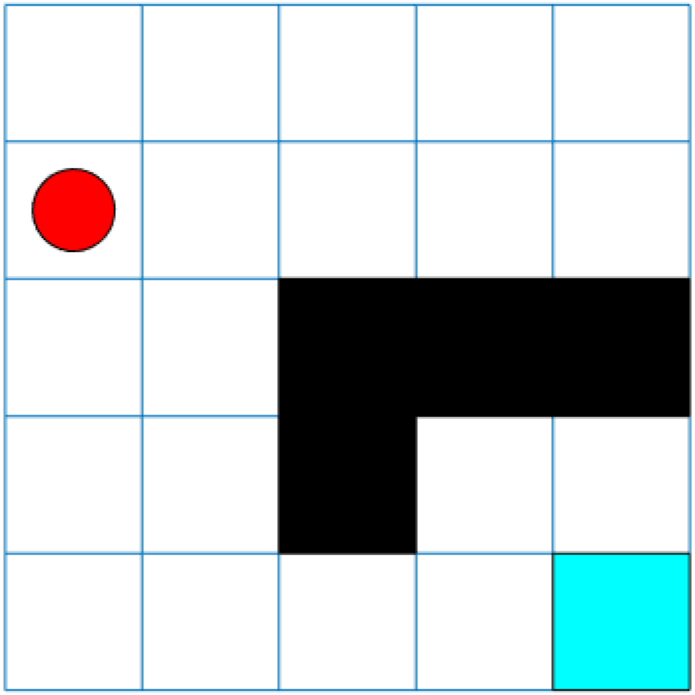
Now, use `rlMDPEnv` to create a grid world environment using the `GridWorld` object `GW`.

```
env = rlMDPEnv(GW)
```

```
env = 
  rlMDPEnv with properties:

       Model: [1x1 rl.env.GridWorld]
    ResetFcn: []
```

You can visualize the grid world environment using the `plot` function.

```
plot(env)
```



Input Arguments

`m` — Number of rows of the grid world

scalar

Number of rows of the grid world, specified as a scalar.

n — Number of columns of the grid world
scalar

Number of columns of the grid world, specified as a scalar.

moves — Action names
'Standard' (default) | 'Kings'

Action names, specified as either 'Standard' or 'Kings'. When moves is set to

- 'Standard', the actions are ['N';'S';'E';'W'].
- 'Kings', the actions are ['N';'S';'E';'W';'NE';'NW';'SE';'SW'].

Output Arguments

GW — Two-dimensional grid world
GridWorld object

Two-dimensional grid world, returned as a GridWorld object with properties listed below. For more information, see “Create Custom Grid World Environments”.

GridSize — Size of the grid world
[m,n] vector

Size of the grid world, specified as a [m,n] vector.

CurrentState — Name of the current state
string

Name of the current state, specified as a string.

States — State names
string vector

State names, specified as a string vector of length m*n.

Actions — Action names
string vector

Action names, specified as a string vector. The length of the Actions vector is determined by the moves argument.

Actions is a string vector of length:

- Four, if moves is specified as 'Standard'.
- Eight, moves is specified as 'Kings'.

T — State transition matrix
3D array

State transition matrix, specified as a 3-D array, which determines the possible movements of the agent in an environment. State transition matrix T is a probability matrix that indicates how likely the agent will move from the current state s to any possible next state s' by performing action a. T is given by,

$$T(s, s', a) \;=\; probability(s'|s, a)\,.$$

T is:

- A K-by-K-by-4 array, if moves is specified as 'Standard'. Here, K = m*n.
- A K-by-K-by-8 array, if moves is specified as 'Kings'.

R — Reward transition matrix
3D array

Reward transition matrix, specified as a 3-D array, determines how much reward the agent receives after performing an action in the environment. R has the same shape and size as state transition matrix T. Reward transition matrix R is given by,

$$r \;=\; R(s, s', a)\,.$$

R is:

- A K-by-K-by-4 array, if moves is specified as 'Standard'. Here, K = m*n.
- A K-by-K-by-8 array, if moves is specified as 'Kings'.

ObstacleStates — State names that cannot be reached in the grid world
string vector

State names that cannot be reached in the grid world, specified as a string vector.

TerminalStates — Terminal state names in the grid world
string vector

Terminal state names in the grid world, specified as a string vector.

Version History

Introduced in R2019a

See Also

Functions
rlPredefinedEnv

Objects
rlMDPEnv

Topics
“Train Reinforcement Learning Agent in Basic Grid World”
“Create Custom Grid World Environments”

createIntegratedEnv

Create Simulink model for reinforcement learning, using reference model as environment

Syntax

```
env = createIntegratedEnv(refModel,newModel)
[env,agentBlock,obsInfo,actInfo] = createIntegratedEnv(___)

[___] = createIntegratedEnv(___,Name,Value)
```

Description

`env = createIntegratedEnv(refModel,newModel)` creates a Simulink model with the name specified by `newModel` and returns a reinforcement learning environment object, `env`, for this model. The new model contains an RL Agent block and uses the reference model `refModel` as a reinforcement learning environment for training the agent specified by this block.

`[env,agentBlock,obsInfo,actInfo] = createIntegratedEnv(___)` returns the block path to the RL Agent block in the new model and the observation and action data specifications for the reference model, `obsInfo` and `actInfo`, respectively.

`[___] = createIntegratedEnv(___,Name,Value)` creates a model and environment interface using port, observation, and action information specified using one or more `Name,Value` pair arguments.

Examples

Create Environment from Simulink Model

This example shows how to use `createIntegratedEnv` to create an environment object starting from a Simulink model that implements the system with which the agent. Such a system is often referred to as *plant, open-loop* system, or *reference* system, while the whole (integrated) system including the agent is often referred to as the *closed-loop* system.

For this example, use the flying robot model described in “Train DDPG Agent to Control Flying Robot” as the reference (open-loop) system.

Open the flying robot model.

```
open_system('rlFlyingRobotEnv')
```

Initialize the state variables and sample time.

```
% initial model state variables
theta0 = 0;
x0 = -15;
y0 = 0;

% sample time
Ts = 0.4;
```

Create the Simulink model `myIntegratedEnv` containing the flying robot model connected in a closed loop to the agent block. The function also returns the reinforcement learning environment object `env` to be used for training.

```
env = createIntegratedEnv('rlFlyingRobotEnv','myIntegratedEnv')
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : myIntegratedEnv
      AgentBlock : myIntegratedEnv/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

The function can also return the block path to the RL Agent block in the new integrated model, as well as the observation and action specifications for the reference model.

```
[~,agentBlk,observationInfo,actionInfo] = createIntegratedEnv( ...
    'rlFlyingRobotEnv','myIntegratedEnv')
```

```
agentBlk = 
'myIntegratedEnv/RL Agent'

observationInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "observation"
    Description: [0x0 string]
      Dimension: [7 1]
       DataType: "double"


actionInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "action"
    Description: [0x0 string]
      Dimension: [2 1]
       DataType: "double"
```

Returning the block path and specifications is useful in cases in which you need to modify descriptions, limits, or names in `observationInfo` and `actionInfo`. After modifying the specifications, you can then create an environment from the integrated model `IntegratedEnv` using the `rlSimulinkEnv` function.

Create Integrated Environment with Specified Port Names

Open the open-loop water tank model.

```
open_system("rlWatertankOpenloop")
```

Set the sample time of the discrete integrator block used to generate the observation, so the simulation can run.

```
Ts = 1;
```

Call `createIntegratedEnv` using name-value pairs to specify port names. The first argument of `createIntegratedEnv` is the name of the *reference* Simulink model that contains the system with which the agent must interact. Such a system is often referred to as *plant*, or *open-loop* system.

For this example, the reference system is the model of a water tank. The input port is called `u` (instead of `action`), and the first and third output ports are called `y` and `stop` (instead of `observation` and `isdone`). Specify the port names using name-value pairs.

```
env = createIntegratedEnv("rlWatertankOpenloop","IntegratedWatertank",...
    ActionPortName="u",ObservationPortName="y",IsDonePortName="stop")
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : IntegratedWatertank
      AgentBlock : IntegratedWatertank/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

The new model `IntegratedWatertank` contains the reference model connected in a closed-loop with the agent block. The function also returns the reinforcement learning environment object to be used for training.

Input Arguments

`refModel` — Reference model name

string | character vector

Reference model name, specified as a string or character vector. This is the Simulink model implementing the system that the agent needs to interact with. Such a system is often referred to as *plant*, *open loop* system or *reference* system, while the whole (integrated) system including the agent is often referred to as the *closed loop* system. The new Simulink model uses this reference model as the dynamic model of the environment for reinforcement learning.

`newModel` — New model name

string | character vector

New model name, specified as a string or character vector. `createIntegratedEnv` creates a Simulink model with this name, but does not save the model.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: 'IsDonePortName',"stopSim" sets the stopSim port of the reference model as the source of the isdone signal.

ObservationPortName — Reference model observation output port name
"observation" (default) | string | character vector

Reference model observation output port name, specified as the comma-separated pair consisting of 'ObservationPortName' and a string or character vector. Specify ObservationPortName when the name of the observation output port of the reference model is not "observation".

ActionPortName — Reference model action input port name
"action" (default) | string | character vector

Reference model action input port name, specified as the comma-separated pair consisting of 'ActionPortName' and a string or character vector. Specify ActionPortName when the name of the action input port of the reference model is not "action".

RewardPortName — Reference model reward output port name
"reward" (default) | string | character vector

Reference model reward output port name, specified as the comma-separated pair consisting of 'RewardPortName' and a string or character vector. Specify RewardPortName when the name of the reward output port of the reference model is not "reward".

IsDonePortName — Reference model done flag output port name
"isdone" (default) | string | character vector

Reference model done flag output port name, specified as the comma-separated pair consisting of 'IsDonePortName' and a string or character vector. Specify IsDonePortName when the name of the done flag output port of the reference model is not "isdone".

ObservationBusElementNames — Names of observation bus leaf elements
string array

Names of observation bus leaf elements for which to create specifications, specified as a string array. To create observation specifications for a subset of the elements in a Simulink bus object, specify BusElementNames. If you do not specify BusElementNames, a data specification is created for each leaf element in the bus.

ObservationBusElementNames is applicable only when the observation output port is a bus signal.

Example: 'ObservationBusElementNames',["sin" "cos"] creates specifications for the observation bus elements with the names "sin" and "cos".

ObservationDiscreteElements — Finite values for observation specifications
cell array of name-value pairs

Finite values for discrete observation specification elements, specified as the comma-separated pair consisting of 'ObservationDiscreteElements' and a cell array of name-value pairs. Each name-value pair consists of an element name and an array of discrete values.

If the observation output port of the reference model is:

- A bus signal, specify the name of one of the leaf elements of the bus specified in by ObservationBusElementNames

- Nonbus signal, specify the name of the observation port, as specified by `ObservationPortName`

The specified discrete values must be castable to the data type of the specified observation signal.

If you do not specify discrete values for an observation specification element, the element is continuous.

Example: `'ObservationDiscretElements',{'observation',[-1 0 1]}` specifies discrete values for a nonbus observation signal with default port name `observation`.

Example: `'ObservationDiscretElements',{'gear',[-1 0 1 2],'direction',[1 2 3 4]}` specifies discrete values for the `'gear'` and `'direction'` leaf elements of a bus action signal.

`ActionDiscreteElements` — Finite values for action specifications
cell array of name-value pairs

Finite values for discrete action specification elements, specified as the comma-separated pair consisting of `'ActionDiscreteElements'` and a cell array of name-value pairs. Each name-value pair consists of an element name and an array of discrete values.

If the action input port of the reference model is:

- A bus signal, specify the name of a leaf element of the bus
- Nonbus signal, specify the name of the action port, as specified by `ActionPortName`

The specified discrete values must be castable to the data type of the specified action signal.

If you do not specify discrete values for an action specification element, the element is continuous.

Example: `'ActionDiscretElements',{'action',[-1 0 1]}` specifies discrete values for a nonbus action signal with default port name `'action'`.

Example: `'ActionDiscretElements',{'force',[-10 0 10],'torque',[-5 0 5]}` specifies discrete values for the `'force'` and `'torque'` leaf elements of a bus action signal.

Output Arguments

`env` — Reinforcement learning environment
`SimulinkEnvWithAgent` object

Reinforcement learning environment interface, returned as an `SimulinkEnvWithAgent` object.

`agentBlock` — Block path to the agent block
character vector

Block path to the agent block in the new model, returned as a character vector. To train an agent in the new Simulink model, you must create an agent and specify the agent name in the RL Agent block indicated by `agentBlock`.

For more information on creating agents, see “Reinforcement Learning Agents”.

`obsInfo` — Observation data specifications
`rlNumericSpec` object | `rlFiniteSetSpec` object | array of data specification objects

Observation data specifications, returned as one of the following:

- `rlNumericSpec` object for a single continuous observation specification
- `rlFiniteSetSpec` object for a single discrete observation specification
- Array of data specification objects for multiple specifications

`actInfo` — Action data specifications
`rlNumericSpec` object | `rlFiniteSetSpec` object | array of data specification objects

Action data specifications, returned as one of the following:

- `rlNumericSpec` object for a single continuous action specification
- `rlFiniteSetSpec` object for a single discrete action specification
- Array of data specification objects for multiple action specifications

Version History

Introduced in R2019a

See Also

Functions
`rlSimulinkEnv` | `bus2RLSpec`

Objects
`rlNumericSpec` | `rlFiniteSetSpec`

Blocks
RL Agent

Topics
“Create Simulink Reinforcement Learning Environments”

createMDP

Create Markov decision process model

Syntax

```
MDP = createMDP(states,actions)
```

Description

`MDP = createMDP(states,actions)` creates a Markov decision process model with the specified states and actions.

Examples

Create MDP Model

Create an MDP model with eight states and two possible actions.

```
MDP = createMDP(8,["up";"down"]);
```

Specify the state transitions and their associated rewards.

```
% State 1 Transition and Reward
MDP.T(1,2,1) = 1;
MDP.R(1,2,1) = 3;
MDP.T(1,3,2) = 1;
MDP.R(1,3,2) = 1;

% State 2 Transition and Reward
MDP.T(2,4,1) = 1;
MDP.R(2,4,1) = 2;
MDP.T(2,5,2) = 1;
MDP.R(2,5,2) = 1;

% State 3 Transition and Reward
MDP.T(3,5,1) = 1;
MDP.R(3,5,1) = 2;
MDP.T(3,6,2) = 1;
MDP.R(3,6,2) = 4;

% State 4 Transition and Reward
MDP.T(4,7,1) = 1;
MDP.R(4,7,1) = 3;
MDP.T(4,8,2) = 1;
MDP.R(4,8,2) = 2;

% State 5 Transition and Reward
MDP.T(5,7,1) = 1;
MDP.R(5,7,1) = 1;
MDP.T(5,8,2) = 1;
MDP.R(5,8,2) = 9;
```

```
% State 6 Transition and Reward
MDP.T(6,7,1) = 1;
MDP.R(6,7,1) = 5;
MDP.T(6,8,2) = 1;
MDP.R(6,8,2) = 1;

% State 7 Transition and Reward
MDP.T(7,7,1) = 1;
MDP.R(7,7,1) = 0;
MDP.T(7,7,2) = 1;
MDP.R(7,7,2) = 0;

% State 8 Transition and Reward
MDP.T(8,8,1) = 1;
MDP.R(8,8,1) = 0;
MDP.T(8,8,2) = 1;
MDP.R(8,8,2) = 0;
```

Specify the terminal states of the model.

```
MDP.TerminalStates = ["s7";"s8"];
```

Input Arguments

states — Model states

positive integer | string vector

Model states, specified as one of the following:

- Positive integer — Specify the number of model states. In this case, each state has a default name, such as "s1" for the first state.
- String vector — Specify the state names. In this case, the total number of states is equal to the length of the vector.

actions — Model actions

positive integer | string vector

Model actions, specified as one of the following:

- Positive integer — Specify the number of model actions. In this case, each action has a default name, such as "a1" for the first action.
- String vector — Specify the action names. In this case, the total number of actions is equal to the length of the vector.

Output Arguments

MDP — MDP model

GenericMDP object

MDP model, returned as a GenericMDP object with the following properties.

CurrentState — Name of the current state

string

Name of the current state, specified as a string.

`States` — State names
string vector

State names, specified as a string vector with length equal to the number of states.

`Actions` — Action names
string vector

Action names, specified as a string vector with length equal to the number of actions.

`T` — State transition matrix
3D array

State transition matrix, specified as a 3-D array, which determines the possible movements of the agent in an environment. State transition matrix `T` is a probability matrix that indicates how likely the agent will move from the current state `s` to any possible next state `s'` by performing action `a`. `T` is an S-by-S-by-A array, where S is the number of states and A is the number of actions. It is given by:

$$T(s, s', a) = probability(s' | s, a) .$$

The sum of the transition probabilities out from a nonterminal state `s` following a given action must sum up to one. Therefore, all stochastic transitions out of a given state must be specified at the same time.

For example, to indicate that in state `1` following action `4` there is an equal probability of moving to states `2` or `3`, use the following:

```
MDP.T(1,[2 3],4) = [0.5 0.5];
```

You can also specify that, following an action, there is some probability of remaining in the same state. For example:

```
MDP.T(1,[1 2 3 4],1) = [0.25 0.25 0.25 0.25];
```

`R` — Reward transition matrix
3D array

Reward transition matrix, specified as a 3-D array, which determines how much reward the agent receives after performing an action in the environment. `R` has the same shape and size as state transition matrix `T`. The reward for moving from state `s` to state `s'` by performing action `a` is given by:

$$r = R(s, s', a) .$$

`TerminalStates` — Terminal state names in the grid world
string vector

Terminal state names in the grid world, specified as a string vector of state names.

Version History

Introduced in R2019a

See Also

Functions
`createGridWorld`

Objects
`rlMDPEnv`

Topics
"Train Reinforcement Learning Agent in MDP Environment"

evaluate

Package: `rl.function`

Evaluate function approximator object given observation (or observation-action) input data

Syntax

```
outData = evaluate(fcnAppx,inData)
[outData,state] = evaluate(fcnAppx,inData)
```

Description

`outData = evaluate(fcnAppx,inData)` evaluates the function approximator object (that is, the actor or critic) `fcnAppx` given the input value `inData`. It returns the output value `outData`.

`[outData,state] = evaluate(fcnAppx,inData)` also returns the updated state of `fcnAppx` when it contains a recurrent neural network.

Examples

Evaluate a Function Approximator Object

This example shows you how to evaluate a function approximator object (that is, an actor or a critic). For this example, the function approximator object is a discrete categorical actor and you evaluate it given some observation data, obtaining in return the action probability distribution and the updated network state.

Load the same environment used in “Train PG Agent to Balance Cart-Pole System”, and obtain the observation and action specifications.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "CartPole States"
    Description: "x, dx, theta, dtheta"
      Dimension: [4 1]
       DataType: "double"
```

```
actInfo = getActionInfo(env)
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [-10 10]
           Name: "CartPole Action"
```

```
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

To approximate the policy within the actor, use a recurrent deep neural network. Define the network as an array of layer objects. Get the dimensions of the observation space and the number of possible actions directly from the environment specification objects.

```
net = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(8)
    reluLayer
    lstmLayer(8,OutputMode="sequence")
    fullyConnectedLayer(numel(actInfo.Elements)) ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 602

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create a stochastic actor representation for the network.

```
actor = rlDiscreteCategoricalActor(net,obsInfo,actInfo);
```

Use `evaluate` to return the probability of each of the two possible actions. Note that the type of the returned numbers is `single`, not `double`.

```
[prob,state] = evaluate(actor,{rand(obsInfo.Dimension)});
prob{1}
```

```
ans = 2x1 single column vector

    0.4847
    0.5153
```

Since a recurrent neural network is used for the actor, the second output argument, representing the updated state of the neural network, is not empty. In this case, it contains the updated (cell and hidden) states for the eight units of the `lstm` layer used in the network.

```
state{:}
```

```
ans = 8x1 single column vector

   -0.0833
    0.0619
   -0.0066
   -0.0651
    0.0714
   -0.0957
    0.0614
   -0.0326


ans = 8x1 single column vector

   -0.1367
    0.1142
   -0.0158
   -0.1820
    0.1305
   -0.1779
    0.0947
   -0.0833
```

You can use `getState` and `setState` to extract and set the current state of the actor.

```
getState(actor)
```

```
ans=2×1 cell array
    {8x1 single}
    {8x1 single}
```

```
actor = setState(actor, ...
                    {-0.01*single(rand(8,1)), ...
                      0.01*single(rand(8,1))});
```

You can obtain action probabilities and updated states for a batch of observations. For example, use a batch of five independent observations.

```
obsBatch = reshape(1:20,4,1,5,1);
[prob,state] = evaluate(actor,{obsBatch})
```

```
prob = 1x1 cell array
    {2x5 single}


state=2×1 cell array
    {8x5 single}
    {8x5 single}
```

The output arguments contain action probabilities and updated states for each observation in the batch.

Note that the actor treats observation data along the batch length dimension independently, not sequentially.

```
prob{1}
```

```
ans = 2x5 single matrix

    0.5187    0.5869    0.6048    0.6124    0.6155
    0.4813    0.4131    0.3952    0.3876    0.3845
```

```
prob = evaluate(actor,{obsBatch(:,:,[5 4 3 1 2])});
prob{1}
```

```
ans = 2x5 single matrix

    0.6155    0.6124    0.6048    0.5187    0.5869
    0.3845    0.3876    0.3952    0.4813    0.4131
```

To evaluate the actor using sequential observations, use the sequence length (time) dimension. For example, obtain action probabilities for five independent sequences, each one made of nine sequential observations.

```
[prob,state] = evaluate(actor, ...
    {rand([obsInfo.Dimension 5 9])})
```

```
prob = 1x1 cell array
    {2x5x9 single}


state=2×1 cell array
    {8x5 single}
    {8x5 single}
```

The first output argument contains a vector of two probabilities (first dimension) for each element of the observation batch (second dimension) and for each time element of the sequence length (third dimension).

The second output argument contains two vectors of final states for each observation batch (that is, the network maintains a separate state history for each observation batch).

Display the probability of the second action, after the seventh sequential observation in the fourth independent batch.

```
prob{1}(2,4,7)
```

```
ans = single
    0.5675
```

For more information on input and output format for recurrent neural networks, see the Algorithms section of `lstmLayer`.

Input Arguments

`fcnAppx` — Function approximator object

function approximator object

Function approximator object, specified as:

- `rlValueFunction`,
- `rlQValueFunction`,
- `rlVectorQValueFunction`,
- `rlDiscreteCategoricalActor`,
- `rlContinuousDeterministicActor`,
- `rlContinuousGaussianActor`,

- `rlContinuousDeterministicTransitionFunction`,
- `rlContinuousGaussianTransitionFunction`,
- `rlContinuousDeterministicRewardFunction`,
- `rlContinuousGaussianRewardFunction`,
- `rlIsDoneFunction` object.

`inData` — Input data for function approximator
cell array

Input data for the function approximator, specified as a cell array with as many elements as the number of input channels of `fcnAppx`. In the following section, the number of observation channels is indicated by N_O.

- If `fcnAppx` is an `rlQValueFunction`, an `rlContinuousDeterministicTransitionFunction` or an `rlContinuousGaussianTransitionFunction` object, then each of the first N_O elements of `inData` must be a matrix representing the current observation from the corresponding observation channel. They must be followed by a final matrix representing the action.
- If `fcnAppx` is a function approximator object representing an actor or critic (but not an `rlQValueFunction` object), `inData` must contain N_O elements, each one a matrix representing the current observation from the corresponding observation channel.
- If `fcnAppx` is an `rlContinuousDeterministicRewardFunction`, an `rlContinuousGaussianRewardFunction`, or an `rlIsDoneFunction` object, then each of the first N_O elements of `inData` must be a matrix representing the current observation from the corresponding observation channel. They must be followed by a matrix representing the action, and finally by N_O elements, each one being a matrix representing the next observation from the corresponding observation channel.

Each element of `inData` must be a matrix of dimension M_C-by-L_B-by-L_S, where:

- M_C corresponds to the dimensions of the associated input channel.
- L_B is the batch size. To specify a single observation, set $L_B = 1$. To specify a batch of (independent) inputs, specify $L_B > 1$. If `inData` has multiple elements, then L_B must be the same for all elements of `inData`.
- L_S specifies the sequence length (length of the sequence of inputs along the time dimension) for recurrent neural network. If `fcnAppx` does not use a recurrent neural network (which is the case for environment function approximators, as they do not support recurrent neural networks), then $L_S = 1$. If `inData` has multiple elements, then L_S must be the same for all elements of `inData`.

For more information on input and output formats for recurrent neural networks, see the Algorithms section of `lstmLayer`.

Example: `{rand(8,3,64,1),rand(4,1,64,1),rand(2,1,64,1)}`

Output Arguments

`outData` — Output data from evaluation of function approximator object
cell array

Output data from the evaluation of the function approximator object, returned as a cell array. The size and contents of outData depend on the type of object you use for fcnAppx, and are shown in the following list. Here, N_O is the number of observation channels.

- rlContinuousDeterministicTransitionFunction - N_O matrices, each one representing the predicted observation from the corresponding observation channel.
- rlContinuousGaussianTransitionFunction - N_O matrices representing the mean value of the predicted observation for the corresponding observation channel, followed by N_O matrices representing the standard deviation of the predicted observation for the corresponding observation channel.
- rlContinuousGaussianActor - Two matrices representing the mean value and standard deviation of the action, respectively.
- rlDiscreteCategoricalActor - A matrix with the probabilities of each action.
- rlContinuousDeterministicActor A matrix with the action.
- rlVectorQValueFunction - A matrix with the values of each possible action.
- rlQValueFunction - A matrix with the value of the action.
- rlValueFunction - A matrix with the value of the current observation.
- rlContinuousDeterministicRewardFunction - A matrix with the predicted reward as a function of current observation, action, and next observation following the action.
- rlContinuousGaussianRewardFunction - Two matrices representing the mean value and standard deviation, respectively, of the predicted reward as a function of current observation, action, and next observation following the action.
- rlIsDoneFunction - A vector with the probabilities of the predicted termination status. Termination probabilities range from 0 (no termination predicted) or 1 (termination predicted), and depend (in the most general case) on the values of observation, action, and next observation following the action.

Each element of outData is a matrix of dimensions D-by-L_B-by-L_S, where:

- D is the vector of dimensions of the corresponding output channel of fcnAppx. Depending on the type of approximator function, this channel can carry a predicted observation (or its mean value or standard deviation), an action (or its mean value or standard deviation), the value (or values) of an observation (or observation-action couple), a predicted reward, or a predicted termination status.
- L_B is the batch size (length of a batch of independent inputs).
- L_S is the sequence length (length of the sequence of inputs along the time dimension) for a recurrent neural network. If fcnAppx does not use a recurrent neural network (which is the case for environment function approximators, as they do not support recurrent neural networks), then $L_S = 1$.

Note If fcnAppx is a critic, then evaluate behaves identically to getValue except that it returns results inside a single-cell array. If fcnAppx is an rlContinuousDeterministicActor actor, then evaluate behaves identically to getAction. If fcnAppx is a stochastic actor such as an rlDiscreteCategoricalActor or rlContinuousGaussianActor, then evaluate returns the action probability distribution, while getAction returns a sample action. Specifically, for an rlDiscreteCategoricalActor actor object, evaluate returns the probability of each possible action. For an rlContinuousGaussianActor actor object, evaluate returns the mean and standard deviation of the Gaussian distribution. For these kinds of actors, see also the note in getAction regarding the enforcement of constraints set by the action specification.

Note If `fcnAppx` is an `rlContinuousDeterministicRewardFunction` object, then `evaluate` behaves identically to `predict` except that it returns results inside a single-cell array. If `fcnAppx` is an `rlContinuousDeterministicTransitionFunction` object, then `evaluate` behaves identically to `predict`. If `fcnAppx` is an `rlContinuousGaussianTransitionFunction` object, then `evaluate` returns the mean value and standard deviation the observation probability distribution, while `predict` returns an observation sampled from this distribution. Similarly, for an `rlContinuousGaussianRewardFunction` object, `evaluate` returns the mean value and standard deviation the reward probability distribution, while `predict` returns a reward sampled from this distribution. Finally, if `fcnAppx` is an `rlIsDoneFunction` object, then `evaluate` returns the probabilities of the termination status being false or true, respectively, while `predict` returns a predicted termination status sampled with these probabilities.

`state` — Updated state of function approximator object

cell array

Next state of the function approximator object, returned as a cell array. If `fcnAppx` does not use a recurrent neural network (which is the case for environment function approximators), then `state` is an empty cell array.

You can set the state of the representation to `state` using the `setState` function. For example:

```
critic = setState(critic,state);
```

Version History

Introduced in R2022a

See Also

Functions
`getValue` | `getAction` | `getMaxQValue` | `accelerate` | `gradient` | `predict`

Objects
`rlValueFunction` | `rlQValueFunction` | `rlVectorQValueFunction` | `rlContinuousDeterministicActor` | `rlDiscreteCategoricalActor` | `rlContinuousGaussianActor` | `rlContinuousDeterministicTransitionFunction` | `rlContinuousGaussianTransitionFunction` | `rlContinuousDeterministicRewardFunction` | `rlContinuousGaussianRewardFunction` | `rlIsDoneFunction`

Topics
"Train Reinforcement Learning Policy Using Custom Training Loop"
"Create Custom Reinforcement Learning Agents"

exteriorPenalty

Exterior penalty value for a point with respect to a bounded region

Syntax

```
p = exteriorPenalty(x,xmin,xmax,method)
```

Description

`p = exteriorPenalty(x,xmin,xmax,method)` uses the specified `method` to calculate the nonnegative (exterior) penalty vector `p` for the point `x` with respect to the region bounded by `xmin` and `xmax`. `p` has the same dimension as `x`.

Examples

Calculate Exterior Penalty for Point

This example shows how to use the `exteriorPenalty` function to calculate the exterior penalty for a given point, with respect to a bounded region.

Calculate the penalty value for the point `0.1` within the interval [`-2`,`2`], using the `step` method.

```
exteriorPenalty(0.1,-2,2,"step")
```

```
ans = 0
```

Calculate the penalty value for the point `4` outside the interval [`-2`,`2`], using the `step` method.

```
exteriorPenalty(4,-2,2,"step")
```

```
ans = 1
```

Calculate the penalty value for the point `4` outside the interval [`-2`,`2`], using the `quadratic` method.

```
exteriorPenalty(4,-2,2,"quadratic")
```

```
ans = 4
```

Calculate the penalty value for the point [`-2`,`0`,`4`] with respect to the box defined by the intervals [`0`,`1`], [`-1`,`1`], and [`-2`,`2`] along the x, y, and z dimensions, respectively, using the `quadratic` method.

```
exteriorPenalty([-2 0 4],[0 -1 -2],[1 1 2],"quadratic")
```

```
ans = 3×1

     4
     0
     4
```

Visualize Penalty Values for an Interval

Create a vector of 1001 equidistant points distributed between -5 and 5.

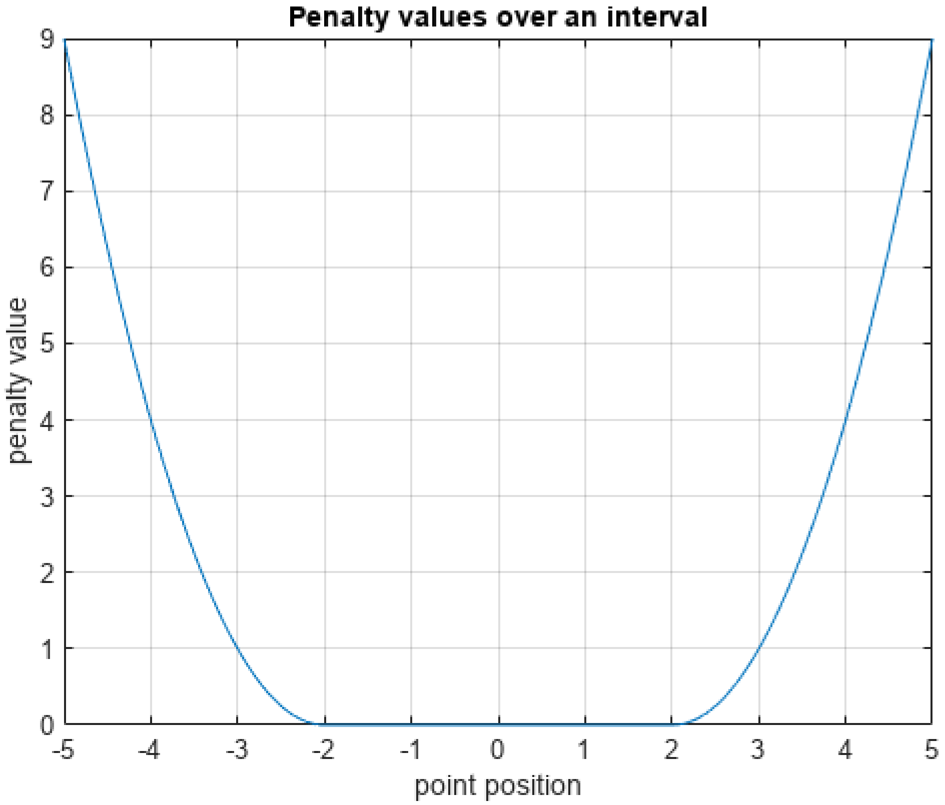
```
x = -5:0.01:5;
```

Calculate penalties for all the points in the vector, using the `quadratic` method.

```
p = exteriorPenalty(x,-2,2,"quadratic");
```

Plot the points and add grid, labels, and title.

```
plot(x,p)
grid
xlabel("point position");
ylabel("penalty value");
title("Penalty values over an interval");
```



Input Arguments

x — Point for which penalty is calculated
scalar | vector | matrix

Point for which the exterior penalty is calculated, specified as a numeric scalar, vector, or matrix.

Example: `[-0.1, 1.3]`

xmin — Lower bounds

scalar | vector | matrix

Lower bounds for x, specified as a numeric scalar, vector, or matrix. To use the same minimum value for all elements in x, specify xmin as a scalar.

Example: -2

xmax — Upper bounds

scalar | vector | matrix

Upper bounds for x, specified as a numeric scalar, vector, or matrix. To use the same maximum value for all elements in x, specify xmax as a scalar.

Example: [5 10]

method — Function used to calculate the penalty

'step' | 'quadratic'

Function used to calculate the penalty, specified either as 'step' or 'quadratic'. You can also use strings instead of character vectors.

Example: "quadratic"

Output Arguments

p — Penalty value

nonnegative vector

Penalty value, returned as a vector of nonnegative elements. With either of the two methods, each element p_i is zero if the corresponding x_i is within the region specified by $xmin_i$ and $xmax_i$, and it is positive otherwise. Penalty functions are typically used to generate negative rewards when constraints are violated, such as in generateRewardFunction.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

generateRewardFunction | hyperbolicPenalty | barrierPenalty

Topics

“Generate Reward Function from a Model Predictive Controller for a Servomotor”
“Define Reward Signals”

fetchNext

Package: `rl.env`

Retrieve next available unread outputs from a reinforcement learning environment simulations running on workers

Syntax

```
[idx,out] = fetchNext(F)
[idx,out] = fetchNext(F,timeout)
```

Description

`[idx,out] = fetchNext(F)` blocks the command prompt and waits for an unread element of `F` (which corresponds to a simulation scheduled on a worker) to reach a finished state. It then returns the index `idx` of the simulation that finished and the corresponding output `out`, which is a scalar structure consistent with the output of `runEpisode`.

`[idx,out] = fetchNext(F,timeout)` waits for a maximum of `timeout` seconds for a result to become available. If no simulation is finished after `timeout` seconds, then both outputs are returned as empty arrays.

Examples

Retrieve Next Unread Output from Environment Simulation

This example shows how to use `fetchNext` to retrieve the next available unread result from reinforcement learning environment simulations running on workers.

Load a predefined environment and a suitable agent. For this example use both the environment and agent described in “Train MBPO Agent to Balance Cart-Pole System”.

```
env = rlPredefinedEnv("CartPole-Continuous");
load("MATLABCartpoleMBPO.mat","agent");
```

Start a parallel pool and set up the environment so that it simulates on workers.

```
pp = parpool(2);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 2 workers.
```

```
setup(env,UseParallel=true);
```

To record the completion time of each simulation, start a timer.

```
tic
```

Initialize two vectors to store the average reward values and the completion time for each simulation, respectively.

```
avr = zeros(6,1);
sct = zeros(6,1);
```

Schedule six simulations to run on the available workers. At the beginning of the simulation, the reset function of the environment sets the initial angle of the pole randomly around zero (the upward position), thereby ensuring that each simulation is unique.

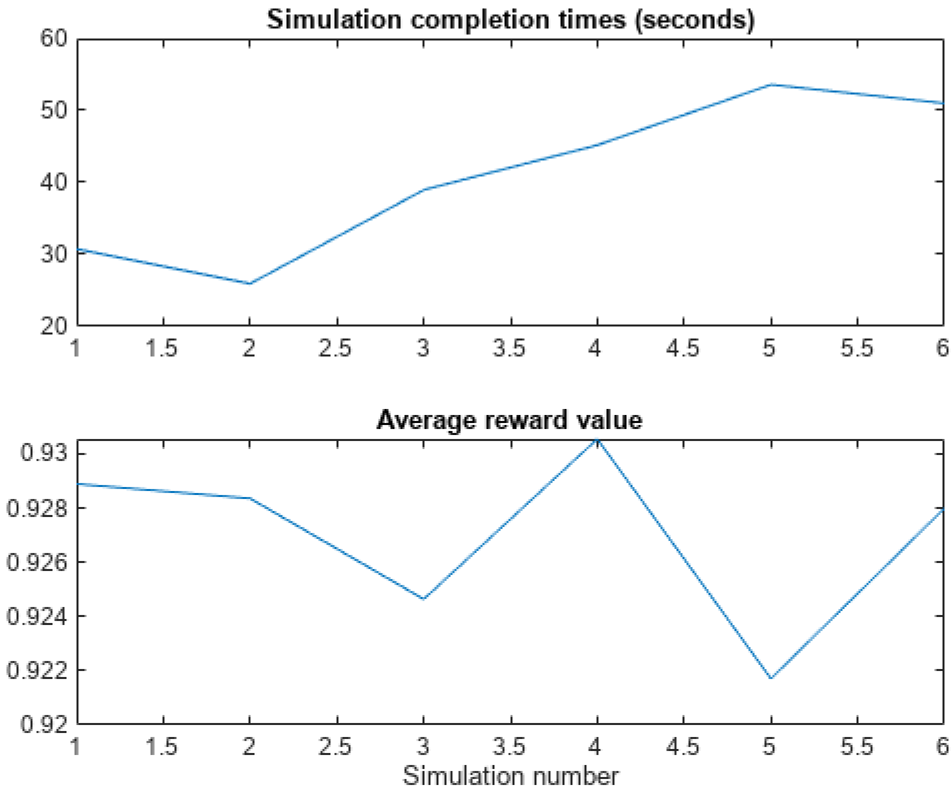
```
for i=1:6
    ftr(i) = runEpisode(env,agent,CleanupPostSim=false);
end
```

Use `fetchNext` in a loop to retrieve results.

```
while ~all([ftr.Read] == true)

    % Wait until an output is available then retrieve it
    [i,out] = fetchNext(ftr);

    % Store the simulation completion time
    sct(i) = toc;

    % Store the average reward value
    avr(i) = mean([out.AgentData.Experiences.Reward]);

end
```

Plot average reward and timing for each simulation.

```
figure

subplot(2,1,1);
plot(sct)
title('Simulation completion times (seconds)');

subplot(2,1,2)
plot(avr)
title('Average reward value');

xlabel('Simulation number');
```



As expected, simulations run in parallel (and therefore terminate at about the same time) in groups of two.

Clear the array of `Future` objects, the environment, and delete the parallel pool (this is the reverse order in which they were created).

```
clear ftr
clear env
delete(pp)
```

Defer Outputs of Simulations Running on Workers

This example shows how to use `Future` objects and their methods `fetchNext`, `fetchOutput`, `cancel`, and `wait` to defer output retrieval for environment simulations running on workers, monitor the status of ongoing simulations, fetch outputs of completed simulations, cancel ongoing simulations, or wait for ongoing simulations to complete.

Load a predefined environment and a suitable agent. For this example use both the environment and agent described in “Train AC Agent to Balance Cart-Pole System”.

```
env = rlPredefinedEnv("CartPole-Discrete");
load("MATLABCartpoleAC.mat","agent")
```

Start a parallel pool and set up the environment so that it simulates on workers.

```
pp = parpool(2);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 2 workers.
```

```
setup(env,UseParallel=true);
```

To display the simulation completion times, start a timer.

```
tic
```

Schedule six simulation to run on the available workers. At the beginning of the simulation, the reset function of the cart-pole environment sets the initial angle of the pole to a random position in the neighborhood of zero (the upward position). This randomization ensures that each simulation is different.

```
for i=1:6
    ftr(i) = runEpisode(env,agent,CleanupPostSim=false);
end
```

Each element of the `Future` array `ftr` represents a scheduled simulation.

```
ftr
```

```
ftr=1×6 object
  1×6 Future array with properties:

    Read
    State
    Diary
    ID
```

Display the state of each simulation.

```
ftr.State
```

```
ans = 
'running'

ans = 
'running'

ans = 
'queued'

ans = 
'queued'

ans = 
'queued'

ans = 
'queued'
```

Two simulations are ongoing while the others are queued.

Use `fetchNext` with a timeout of `0.1` seconds to retrieve results for simulations that complete within that time (if any).

```
[idx,out] = fetchNext(ftr,0.1)
```

```
idx =

     []


out =

     []
```

Both the outputs are empty, which means that none of the four simulation has completed yet.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

Use `fetchNext` without any timeout to wait until an unretrieved simulation output becomes available and then return the results.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 2

out = struct with fields:
    SimulationInfo: [1×1 struct]
         AgentData: [1×1 struct]
```

Display the state of the simulations.

```
ftr.State
```

```
ans =
'finished'
```

```
ans =
'finished'

ans =
'running'

ans =
'running'

ans =
'queued'

ans =
'queued'
```

As expected, the first two simulations, which were running in parallel on the two workers, are finished, while the next two, which were previously queued, are now running, and the final two are still queued.

Display the time taken for the first two simulations to complete.

```
toc
```

```
Elapsed time is 10.451231 seconds.
```

Note that once the results from a simulation has been already retrieved, any attempt to use `fetchNext` to retrieve it again, such as in `fetchNext(ftr(2))`, will result in an error. To retrieve the results from a `Future` object that has already been read, you can use `fetchOuptuts`, such as in `fetchOutputs(ftr(2))`.

Retrieve the next available result, and display the time elapsed since the simulations started.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 1

out = struct with fields:
    SimulationInfo: [1×1 struct]
         AgentData: [1×1 struct]
```

```
toc
```

```
Elapsed time is 11.945070 seconds.
```

As expected, `fetchNext` promptly returns the results from the second simulation, since it was already available.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   1


ans = logical
   1
```

```
ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

Cancel the last simulation.

```
cancel(ftr(6))
```

Wait for the fourth simulation to complete. The `wait` function blocks the command prompt until the fourth simulation is completed.

```
wait(ftr(4))
```

Display the elapsed time since the simulations started.

```
toc
```

```
Elapsed time is 12.414076 seconds.
```

Display the state of the simulations.

```
ftr.State
```

```
ans =
'finished'

ans =
'finished'

ans =
'finished'

ans =
'finished'

ans =
'running'

ans =
'finished'
```

The status of the last element of the array, for which the simulation has been canceled, is classified as `'finished'`.

Since any attempt to retrieve results from a simulation that has been canceled will result in an error, remove the canceled object from the array.

```
ftr(6)=[]
```

```
ftr=1×5 object
  1×5 Future array with properties:

    Read
    State
    Diary
    ID
```

Use `fetchOutputs` to wait until all remaining simulations are completed and then retrieve all outputs.

```
outs = fetchOutputs(ftr)
```

```
outs=5×1 struct array with fields:
    SimulationInfo
    AgentData
```

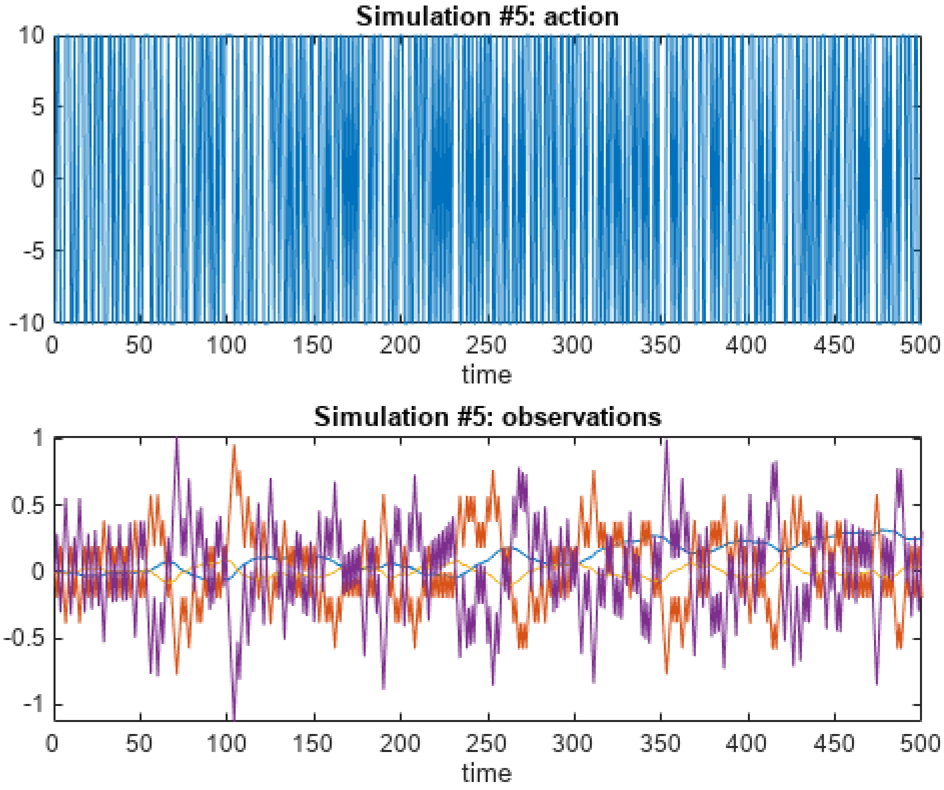
Display the elapsed time.

```
toc
```

```
Elapsed time is 16.265069 seconds.
```

Plot the action and observations from the fifth simulation.

```
figure

subplot(2,1,1);
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Action]))
title('Simulation #5: action');
xlabel('time');

subplot(2,1,2)
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Observation]))
title('Simulation #5: observations')
xlabel('time');
```



Clear the array of `Future` objects, the environment, and delete the parallel pool (this is the reverse order in which they were created).

```
clear ftr
clear env
delete(pp)
```

Input Arguments

F — Future simulation outputs

`Future` object | array of `Future` objects

Future simulation outputs, specified as a `Future` objects or as an array of `Future` objects. To create an element of `F`, set the `UseParallel` property of a reinforcement learning environment to `true`, and then use `runEpisode` to simulate an agent or a policy within this environment. Assign the element of `F` to the output of `runEpisode`.

Note `fetchNext` can only retrieve results from elements of `F` that have not been read before. Any attempt to retrieve results from an element that has its `Read` property set to `true` will result in an error.

`timeout` — Maximum number of seconds to wait for a result

positive scalar

Maximum number of seconds to wait for a result to become available, specified as a positive scalar.

Example: 5

Output Arguments

`idx` — Index

positive integer scalar

Index of the finished simulation that has its output `out` returned by `fetchNext`, returned as a positive integer scalar.

`out` — Simulation output

structure

Output returned from the finished simulation indicated by `idx`, retuned as a structure with the fields `AgentData` and `SimulationInfo`, as described in the outputs section of `runEpisode`.

Version History

Introduced in R2022a

See Also

Objects
`Future` | `Future` | `Simulink.Simulation.Future`

Functions
`fetchOutputs` | `cancel` | `wait` | `runEpisode` | `setup` | `cleanup` | `reset`

Topics
"Custom Training Loop with Simulink Action Noise"
"Train Reinforcement Learning Policy Using Custom Training Loop"
"Train Agents Using Parallel Computing and GPUs"

fetchOutputs

Package: `rl.env`

Retrieve results from all reinforcement learning environment simulations running on workers

Syntax

```
outs = fetchOutputs(F)
```

Description

`outs = fetchOutputs(F)` blocks the command prompt and waits for all elements of `F` (each corresponding to a simulation scheduled on a worker) to reach a finished state. It then returns the output `outs`, which is an array of structures consistent with the output of `runEpisode`. Each elements of `outs` contains results from the simulation specified by the corresponding element of F.

Examples

Defer Outputs of Simulations Running on Workers

This example shows how to use `Future` objects and their methods `fetchNext`, `fetchOutput`, `cancel`, and `wait` to defer output retrieval for environment simulations running on workers, monitor the status of ongoing simulations, fetch outputs of completed simulations, cancel ongoing simulations, or wait for ongoing simulations to complete.

Load a predefined environment and a suitable agent. For this example use both the environment and agent described in “Train AC Agent to Balance Cart-Pole System”.

```
env = rlPredefinedEnv("CartPole-Discrete");
load("MATLABCartpoleAC.mat","agent")
```

Start a parallel pool and set up the environment so that it simulates on workers.

```
pp = parpool(2);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 2 workers.
```

```
setup(env,UseParallel=true);
```

To display the simulation completion times, start a timer.

```
tic
```

Schedule six simulation to run on the available workers. At the beginning of the simulation, the reset function of the cart-pole environment sets the initial angle of the pole to a random position in the neighborhood of zero (the upward position). This randomization ensures that each simulation is different.

```
for i=1:6
    ftr(i) = runEpisode(env,agent,CleanupPostSim=false);
end
```

Each element of the `Future` array `ftr` represents a scheduled simulation.

```
ftr
```

```
ftr=1×6 object
  1×6 Future array with properties:

    Read
    State
    Diary
    ID
```

Display the state of each simulation.

```
ftr.State
```

```
ans =
'running'

ans =
'running'

ans =
'queued'

ans =
'queued'

ans =
'queued'

ans =
'queued'
```

Two simulations are ongoing while the others are queued.

Use `fetchNext` with a timeout of `0.1` seconds to retrieve results for simulations that complete within that time (if any).

```
[idx,out] = fetchNext(ftr,0.1)
```

```
idx =

     []


out =

     []
```

Both the outputs are empty, which means that none of the four simulation has completed yet.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

Use `fetchNext` without any timeout to wait until an unretrieved simulation output becomes available and then return the results.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 2

out = struct with fields:
    SimulationInfo: [1×1 struct]
         AgentData: [1×1 struct]
```

Display the state of the simulations.

```
ftr.State
```

```
ans = 
'finished'

ans = 
'finished'

ans = 
'running'

ans = 
'running'

ans = 
'queued'

ans = 
'queued'
```

As expected, the first two simulations, which were running in parallel on the two workers, are finished, while the next two, which were previously queued, are now running, and the final two are still queued.

Display the time taken for the first two simulations to complete.

```
toc
```

```
Elapsed time is 10.451231 seconds.
```

Note that once the results from a simulation has been already retrieved, any attempt to use `fetchNext` to retrieve it again, such as in `fetchNext(ftr(2))`, will result in an error. To retrieve the results from a `Future` object that has already been read, you can use `fetchOuptuts`, such as in `fetchOutputs(ftr(2))`.

Retrieve the next available result, and display the time elapsed since the simulations started.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 1

out = struct with fields:
    SimulationInfo: [1×1 struct]
         AgentData: [1×1 struct]
```

```
toc
```

```
Elapsed time is 11.945070 seconds.
```

As expected, `fetchNext` promptly returns the results from the second simulation, since it was already available.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   1


ans = logical
   1


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

Cancel the last simulation.

```
cancel(ftr(6))
```

Wait for the fourth simulation to complete. The `wait` function blocks the command prompt until the fourth simulation is completed.

```
wait(ftr(4))
```

Display the elapsed time since the simulations started.

```
toc
```

```
Elapsed time is 12.414076 seconds.
```

Display the state of the simulations.

```
ftr.State
```

```
ans = 
'finished'

ans = 
'finished'

ans = 
'finished'

ans = 
'finished'

ans = 
'running'

ans = 
'finished'
```

The status of the last element of the array, for which the simulation has been canceled, is classified as `'finished'`.

Since any attempt to retrieve results from a simulation that has been canceled will result in an error, remove the canceled object from the array.

```
ftr(6)=[]
```

```
ftr=1×5 object
  1×5 Future array with properties:

    Read
    State
    Diary
    ID
```

Use `fetchOutputs` to wait until all remaining simulations are completed and then retrieve all outputs.

```
outs = fetchOutputs(ftr)
```

```
outs=5×1 struct array with fields:
    SimulationInfo
    AgentData
```

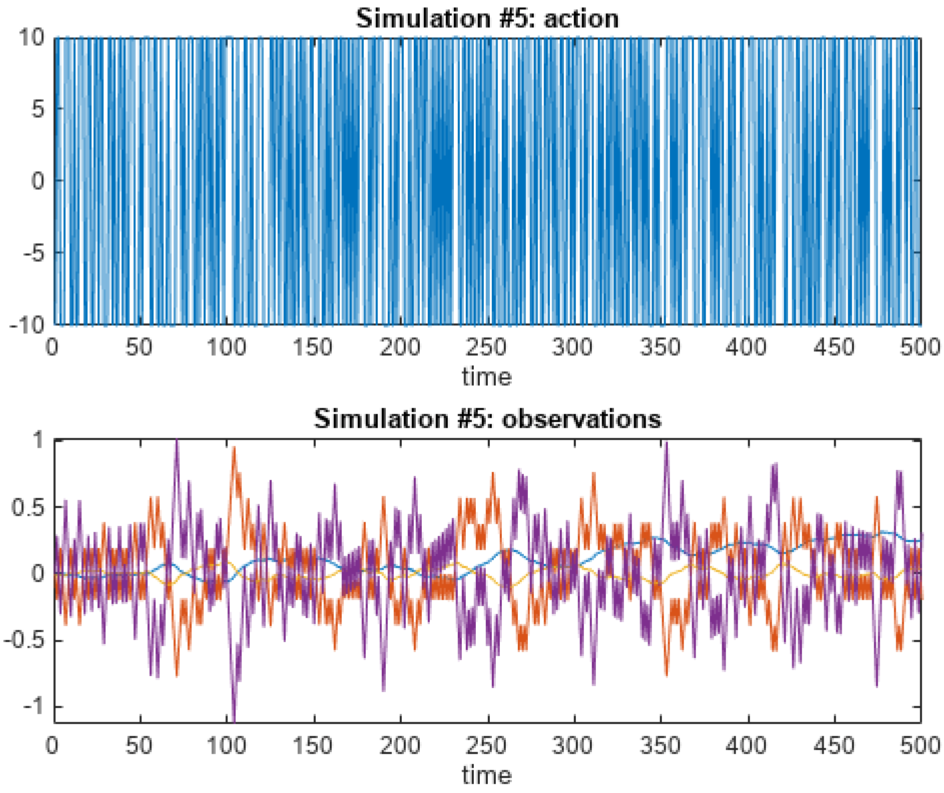
Display the elapsed time.

```
toc
```

```
Elapsed time is 16.265069 seconds.
```

Plot the action and observations from the fifth simulation.

```
figure

subplot(2,1,1);
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Action]))
title('Simulation #5: action');
xlabel('time');

subplot(2,1,2)
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Observation]))
title('Simulation #5: observations')
xlabel('time');
```



Clear the array of `Future` objects, the environment, and delete the parallel pool (this is the reverse order in which they were created).

```
clear ftr
clear env
delete(pp)
```

Input Arguments

F — Future simulation outputs
Future object | array of Future objects

Future simulation outputs, specified as a Future object or as an array of Future objects. To create an element of F, set the UseParallel property of a reinforcement learning environment to true, and then use runEpisode to simulate an agent or a policy within this environment. Assign the element of F to the output of runEpisode.

Output Arguments

outs — Simulation outputs
structure | structure array

Array of structures containing the returned outputs from the simulations specified in F. Each element of the array contains the outputs of the corresponding simulation in the fields AgentData and SimulationInfo. These fields are described in the outputs section of runEpisode.

Version History

Introduced in R2022a

See Also

Objects
Future | Future | Simulink.Simulation.Future

Functions
fetchNext | cancel | wait | runEpisode | setup | cleanup | reset

Topics
“Custom Training Loop with Simulink Action Noise”
“Train Reinforcement Learning Policy Using Custom Training Loop”
“Train Agents Using Parallel Computing and GPUs”

generateHindsightExperiences

Package: rl.replay

Generate hindsight experiences from hindsight experience replay buffer

Syntax

```
experience = generateHindsightExperiences(buffer,trajectoryLength)
```

Description

`experience = generateHindsightExperiences(buffer,trajectoryLength)` generates hindsight experiences from the last trajectory added to the specified hindsight experience replay memory buffer.

Examples

Generate Experiences from Hindsight Replay Memory

When you use a hindsight replay memory buffer within your custom agent training loop, you generate experiences at the end of training episode.

Create an observation specification for an environment with a single a observation channel with six observations. For this example, assume that the observation channel contains the signals [a, x_m, y_m, x_g, y_g, c], where:

- x_g and y_g are the goal observations.
- x_m and y_m are the goal measurements.
- a and c are additional observations.

```
obsInfo = rlNumericSpec([6 1],...
    LowerLimit=0,UpperLimit=[1;5;5;5;5;1]);
```

Create a specification for a single action.

```
actInfo = rlNumericSpec([1 1],...
    LowerLimit=0,UpperLimit=10);
```

To create a hindsight replay memory buffer, first define the goal condition information. Both the goals and goal measurements are in the single observation channel. The goal measurements are in elements 2 and 3 of the observation channel and the goals are in elements 4 and 5 of the observation channel.

```
goalConditionInfo = {{1,[2 3],1,[4 5]}};
```

For this example, use `hindsightRewardFcn1` as the ground-truth reward function and `hindsightIsDoneFcn1` as the termination condition function.

Create the hindsight replay memory buffer.

```
buffer = rlHindsightReplayMemory(obsInfo,actInfo, ...
    @hindsightRewardFcn1,@hindsightIsDoneFcn1,goalConditionInfo);
```

As you train your agent, you add experience trajectories to the experience buffer. For this example, add a random experience trajectory of length 10.

```
for i = 1:10
    exp(i).Observation = {obsInfo.UpperLimit.*rand(6,1)};
    exp(i).Action = {actInfo.UpperLimit.*rand(1)};
    exp(i).NextObservation = {obsInfo.UpperLimit.*rand(6,1)};
    exp(i).Reward = 10*rand(1);
    exp(i).IsDone = 0;
end
exp(10).IsDone = 1;

append(buffer,exp);
```

At the end of the training episode, you generate hindsight experiences from the last trajectory added to the buffer. Generate experiences specifying the length of the last trajectory added to the buffer.

```
newExp = generateHindsightExperiences(buffer,10);
```

For each experience in the final trajectory, the default `"final"` sampling strategy generates a new experience where it replaces the goals in `Observation` and `NextObservation` with the goal measurements from the final experience in the trajectory.

To validate this behavior, first view the final goal measurements from `exp`.

```
exp(10).NextObservation{1}(2:3)
```

```
ans = 2×1

    0.7277
    0.6803
```

Next, view the goal values for one of the generated experiences. This value should match the final goal measurement

```
newExp(6).Observation{1}(4:5)
```

```
ans = 2×1

    0.7277
    0.6803
```

After generating the new experiences, append them to the buffer.

```
append(buffer,newExp);
```

Input Arguments

`buffer` — Hindsight experience buffer

`rlHindsightReplayMemory` object | `rlHindsightPrioritizedReplayMemory` object

Hindsight experience buffer, specified as one of the following replay memory objects.

- `rlHindsightReplayMemory`
- `rlHindsightPrioritizedReplayMemory`

`trajectoryLength` — Length of last trajectory in buffer
positive integer object

Length of last trajectory in buffer, specified as a positive integer.

- `rlHindsightReplayMemory`
- `rlHindsightPrioritizedReplayMemory`

Output Arguments

`experience` — Hindsight experiences generated from the buffer
structure

Experiences sampled from the buffer, returned as a structure with the following fields.

`Observation` — Observation
cell array

Observation, returned as a cell array with length equal to the number of observation specifications specified when creating the buffer. Each element of `Observation` contains a D_O-by-`batchSize`-by-`SequenceLength` array, where D_O is the dimension of the corresponding observation specification.

`Action` — Agent action
cell array

Agent action, returned as a cell array with length equal to the number of action specifications specified when creating the buffer. Each element of `Action` contains a D_A-by-`batchSize`-by-`SequenceLength` array, where D_A is the dimension of the corresponding action specification.

`Reward` — Reward value
scalar | array

Reward value obtained by taking the specified action from the observation, returned as a 1-by-1-by-`SequenceLength` array.

`NextObservation` — Next observation
cell array

Next observation reached by taking the specified action from the observation, returned as a cell array with the same format as `Observation`.

`IsDone` — Termination signal
integer | array

Termination signal, returned as a 1-by-1-by-`SequenceLength` array of integers. Each element of `IsDone` has one of the following values.

- `0` — This experience is not the end of an episode.
- `1` — The episode terminated because the environment generated a termination signal.
- `2` — The episode terminated by reaching the maximum episode length.

`trajectoryLength` — Length of last trajectory in experience buffer
positive integer

Length of last trajectory in experience buffer, specified as a positive integer.

Version History

Introduced in R2023a

See Also

Objects
`rlPrioritizedReplayMemory` | `rlHindsightPrioritizedReplayMemory`

generatePolicyBlock

Generate Simulink block that evaluates policy of an agent or policy object

Syntax

```
generatePolicyBlock(agent)
```

```
generatePolicyBlock(policy)
```

```
generatePolicyBlock( ___ ,MATFileName=dataFileName)
```

Description

This function generates a Simulink Policy evaluation block from an agent or policy object. It also creates a data file which stores policy information. The generated policy block loads this data file to properly initialize itself prior to simulation. You can use the block to simulate the policy and generate code for deployment purposes.

For more information on policies and value functions, see “Create Policies and Value Functions”.

`generatePolicyBlock(agent)` creates a block that evaluates the policy of the specified agent using the default block name, policy name, and data file name.

`generatePolicyBlock(policy)` creates a block that evaluates the learned policy of the specified policy object using the default block name, policy name, and data file name.

`generatePolicyBlock( ___ ,MATFileName=dataFileName)` specifies the file name of the data file.

Examples

Create Policy Evaluation Block from PG Agent

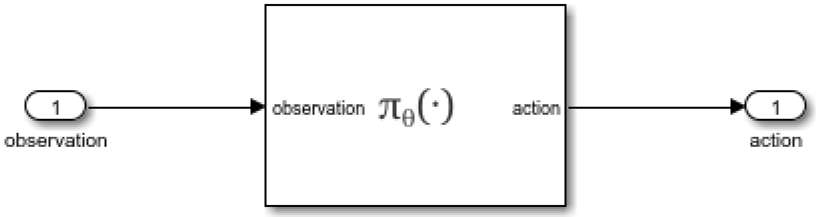
First, create and train a reinforcement learning agent. For this example, load the PG agent trained in “Train PG Agent to Balance Cart-Pole System”.

```
load("MATLABCartpolePG.mat","agent")
```

Then, create a policy evaluation block from this agent using default names.

```
generatePolicyBlock(agent);
```

This command creates an untitled Simulink® model, containing the policy block, and the `blockAgentData.mat` file, containing information needed to create and initialize the policy block, (such as the trained deep neural network used by the actor within the agent). The block loads this data file to properly initialize itself prior to simulation.



You can now drag and drop the block in a Simulink® model and connect it so that it takes the observation from the environment as input and so that the calculated action is returned to the environment. This allows you to simulate the policy in a closed loop. You can then generate code for deployment purposes. For more information, see “Deploy Trained Reinforcement Learning Policies”.

Close the model.

```
bdclose("untitled")
```

Create Policy Block from Deterministic Actor Policy Object

Create observation and action specification objects. For this example, define the observation and action spaces as continuous four- and two-dimensional spaces, respectively.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlNumericSpec([2 1]);
```

Alternatively use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment.

Create a continuous deterministic actor. This actor must accept an observation as input and return an action as output.

To approximate the policy function within the actor, use a recurrent deep neural network model. Define the network as an array of layer objects, and get the dimension of the observation and action spaces from the environment specification objects. To create a recurrent network, use a `sequenceInputLayer` as the input layer (with size equal to the number of dimensions of the observation channel) and include at least one `lstmLayer`.

```
layers = [
    sequenceInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(10)
    reluLayer
    lstmLayer(8,OutputMode="sequence")
    fullyConnectedLayer(20)
    fullyConnectedLayer(actInfo.Dimension(1))
    tanhLayer
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
model = dlnetwork(layers);
summary(model)
```

```
   Initialized: true

   Number of learnables: 880

   Inputs:
      1   "sequenceinput"   Sequence input with 4 dimensions (CTB)
```

Create the actor using `model`, and the observation and action specifications.

```
actor = rlContinuousDeterministicActor(model,obsInfo,actInfo)
```

```
actor = 
  rlContinuousDeterministicActor with properties:

    ObservationInfo: [1×1 rl.util.rlNumericSpec]
         ActionInfo: [1×1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

Check the actor with a random observation input.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = 2×1 single column vector

   -0.0742
    0.0158
```

Create a policy object from `actor`.

```
policy = rlDeterministicActorPolicy(actor)
```

```
policy = 
  rlDeterministicActorPolicy with properties:

              Actor: [1×1 rl.function.rlContinuousDeterministicActor]
    ObservationInfo: [1×1 rl.util.rlNumericSpec]
         ActionInfo: [1×1 rl.util.rlNumericSpec]
         SampleTime: -1
```

You can access the policy options using dot notation. Check the policy with a random observation input.

```
act = getAction(policy,{rand(obsInfo.Dimension)});
act{1}
```

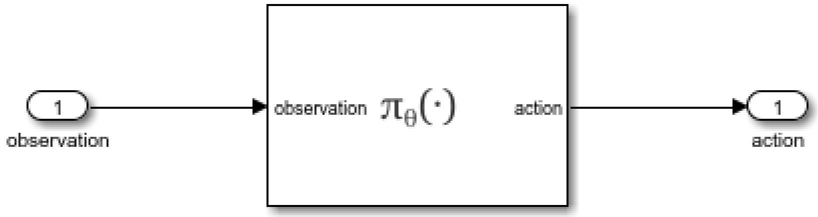
```
ans = 2×1

   -0.0060
   -0.0161
```

You can train the policy with a custom training loop.

Then, create a policy evaluation block from this policy object using the default name for the generated MAT-file.

```
generatePolicyBlock(policy);
```

This command creates an untitled Simulink® model, containing the policy block, and the `blockAgentData.mat` file, containing information needed to create and initialize the policy block, (such as the trained deep neural network used by the actor within the agent). The block loads this data file to properly initialize itself prior to simulation.



You can now drag and drop the block in a Simulink® model and connect it so that it takes the observation from the environment as input and so that the calculated action is returned to the environment. This allows you to simulate the policy in a closed loop. You can then generate code for deployment purposes. For more information, see “Deploy Trained Reinforcement Learning Policies”.

Close the model.

```
bdclose("untitled")
```

Input Arguments

agent — Reinforcement learning agent

reinforcement learning agent object

Trained reinforcement learning agent, specified as one of the following agent objects. To train your agent, use the `train` function.

- `rlQAgent`
- `rlSARSAAgent`
- `rlDQNAgent`
- `rlDDPGAgent`
- `rlTD3Agent`
- `rlACAgent`
- `rlPGAgent`

- rlPPOAgent
- rlTRPOAgent
- rlSACAgent

For agents with a stochastic actor (PG, PPO, SAC, TRPO, AC), the action returned by the generated policy function depends on the value of the UseExplorationPolicy property of the agent. By default, UseExplorationPolicy is false and the generated action is deterministic. If UseExplorationPolicy is true, the generated action is stochastic.

policy — Reinforcement learning policy
rlMaxQPolicy | rlDeterministicActorPolicy | rlStochasticActorPolicy

Reinforcement learning policy, specified as one of the following objects:

- rlMaxQPolicy
- rlDeterministicActorPolicy
- rlStochasticActorPolicy

Note rlAdditiveNoisePolicy and rlEpsilonGreedyPolicy policy objects are not supported.

dataFileName — Name of generated data file
"blockAgentData" (default) | string | character vector

Name of generated data file, specified as a string or character vector. If a file with the specified name already exists in the current MATLAB folder, then an appropriate digit is added to the name so that no existing file is overwritten.

The generated data file contains four structures that store data needed to fully characterize the policy. Prior to simulation, the block (which is generated with the data file name as mask parameter) loads this data file to properly initialize itself.

Version History

Introduced in R2019a

See Also

Functions
generatePolicyFunction | train

Objects
rlMaxQPolicy | rlEpsilonGreedyPolicy | rlAdditiveNoisePolicy | rlDeterministicActorPolicy | rlStochasticActorPolicy | dlnetwork

Blocks
Policy

Topics
“Generate Policy Block for Deployment”
“Train Reinforcement Learning Agents”

“Create Policies and Value Functions”
“Reinforcement Learning Agents”
“Deploy Trained Reinforcement Learning Policies”

generatePolicyFunction

Package: `rl.policy`

Generate MATLAB function that evaluates policy of an agent or policy object

Syntax

```
generatePolicyFunction(agent)
```

```
generatePolicyFunction(policy)
```

```
generatePolicyFunction( ___ ,Name=Value)
```

Description

This function generates a MATLAB policy evaluation function which you can use to:

- Generate code for deployment purposes using MATLAB Coder™ or GPU Coder™. For more information, see “Deploy Trained Reinforcement Learning Policies”.
- Simulate the trained agent in Simulink using a MATLAB Function block.

This function also creates a data file which stores policy information. The evaluation function loads this data file to properly initialize itself the first time it is called.

For more information on policies and value functions, see “Create Policies and Value Functions”.

`generatePolicyFunction(agent)` creates a function that evaluates the learned policy of the specified agent using the default function name, policy name, and data file name.

`generatePolicyFunction(policy)` creates a function that evaluates the learned policy of the specified policy object using the default function name, policy name, and data file name.

`generatePolicyFunction( ___ ,Name=Value)` specifies the function name, policy name, and data file name using one or more name-value pair arguments.

Examples

Create Policy Evaluation Function for PG Agent

This example shows how to create a policy evaluation function for a PG Agent.

First, create and train a reinforcement learning agent. For this example, load the PG agent trained in “Train PG Agent to Balance Cart-Pole System”.

```
load("MATLABCartpolePG.mat","agent")
```

Then, create a policy evaluation function for this agent using default names.

```
generatePolicyFunction(agent);
```

This command creates the `evaluatePolicy.m` file, which contains the policy function, and the `agentData.mat` file, which contains the trained deep neural network actor.

View the generated function.

```
type evaluatePolicy.m
```

```
function action1 = evaluatePolicy(observation1)
%#codegen

% Reinforcement Learning Toolbox
% Generated on: 04-Mar-2023 01:25:12

persistent policy;
if isempty(policy)
    policy = coder.loadRLPolicy("agentData.mat");
end
% evaluate the policy
action1 = getAction(policy,observation1);
```

Evaluate the policy for a random observation.

```
evaluatePolicy(rand(agent.ObservationInfo.Dimension))
```

```
ans = 10
```

You can now generate code for this policy function using MATLAB® Coder™. For more information, see “Deploy Trained Reinforcement Learning Policies”.

Create Policy Evaluation Function from Policy Object

You can create and train a policy object in a custom training loop or extract a trained policy object from a trained agent.

For this example, load the PG agent trained in “Train PG Agent to Balance Cart-Pole System”, and extract its deterministic (and greedy) policy using `getGreedyPolicy`. Alternatively, you can extract a stochastic policy using `getExplorationPolicy`, which would be useful for exploration.

```
load("MATLABCartpolePG.mat","agent")
policy = getGreedyPolicy(agent)
```

```
policy = 
  rlStochasticActorPolicy with properties:

                     Actor: [1x1 rl.function.rlDiscreteCategoricalActor]
    UseMaxLikelihoodAction: 1
           ObservationInfo: [1x1 rl.util.rlNumericSpec]
                ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
                SampleTime: 1
```

Then, create a policy evaluation function for this policy using default names.

```
generatePolicyFunction(policy);
```

This command creates the `evaluatePolicy.m` file, which contains the policy function, and the `agentData.mat` file, which contains the trained deep neural network actor.

View the generated function.

```
type evaluatePolicy.m
```

```
function action1 = evaluatePolicy(observation1)
%#codegen

% Reinforcement Learning Toolbox
% Generated on: 04-Mar-2023 01:25:19

persistent policy;
if isempty(policy)
    policy = coder.loadRLPolicy("agentData.mat");
end
% evaluate the policy
action1 = getAction(policy,observation1);
```

Evaluate the policy for a random observation.

```
evaluatePolicy(rand(policy.ObservationInfo.Dimension))
```

```
ans = 10
```

You can now generate code for this policy function using MATLAB® Coder™. For more information, see “Deploy Trained Reinforcement Learning Policies”.

Create Policy Evaluation Function for Q-Learning Agent

This example shows how to create a policy evaluation function for a Q-Learning Agent.

For this example, load the Q-learning agent trained in “Train Reinforcement Learning Agent in Basic Grid World”

```
load("basicGWQAgent.mat","qAgent")
```

Create a policy evaluation function for this agent and specify the name of the agent data file.

```
generatePolicyFunction(qAgent,"MATFileName","policyFile.mat")
```

This command creates the `evaluatePolicy.m` file, which contains the policy function, and the `policyFile.mat` file, which contains the trained Q table value function.

View the generated function.

```
type evaluatePolicy.m
```

```
function action1 = evaluatePolicy(observation1)
%#codegen

% Reinforcement Learning Toolbox
% Generated on: 05-May-2022 17:43:27

persistent policy;
```

```
if isempty(policy)
    policy = coder.loadRLPolicy("policyFile.mat");
end
% evaluate the policy
action1 = getAction(policy,observation1);
```

Evaluate the policy for a random observation.

```
evaluatePolicy(randi(25))
```

```
ans = 3
```

You can now generate code for this policy function using MATLAB® Coder™. For more information, see “Deploy Trained Reinforcement Learning Policies”.

Input Arguments

`agent` — Reinforcement learning agent

reinforcement learning agent object

Trained reinforcement learning agent, specified as one of the following agent objects. To train your agent, use the `train` function.

- `rlQAgent`
- `rlSARSAAgent`
- `rlDQNAgent`
- `rlDDPGAgent`
- `rlTD3Agent`
- `rlACAgent`
- `rlPGAgent`
- `rlPPOAgent`
- `rlTRPOAgent`
- `rlSACAgent`

For agents with a stochastic actor (PG, PPO, SAC, TRPO, AC), the action returned by the generated policy function depends on the value of the `UseExplorationPolicy` property of the agent. By default, `UseExplorationPolicy` is `false` and the generated action is deterministic. If `UseExplorationPolicy` is `true`, the generated action is stochastic.

`policy` — Reinforcement learning policy

`rlMaxQPolicy` | `rlDeterministicActorPolicy` | `rlStochasticActorPolicy`

Reinforcement learning policy, specified as one of the following objects:

- `rlMaxQPolicy`
- `rlDeterministicActorPolicy`
- `rlStochasticActorPolicy`

Note `rlAdditiveNoisePolicy` and `rlEpsilonGreedyPolicy` policy objects are not supported.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `FunctionName="computeAction"`

`FunctionName` — Name of the generated function

`"evaluatePolicy"` (default) | string | character vector

Name of the generated function, specified as a string or character vector.

`PolicyName` — Name of the policy object within the generated function

`"policy"` (default) | string | character vector

Name of the policy object within the generated function, specified as a string or character vector.

`MATFileName` — Name of generated data file

`"agentData"` (default) | string | character vector

Name of generated data file, specified as a string or character vector. If a file with the specified name already exists in the current MATLAB folder, then an appropriate digit is added to the name so that no existing file is overwritten.

The generated data file contains four structures that store data needed to fully characterize the policy. The evaluation function loads this data file to properly initialize itself the first time it is called.

Version History

Introduced in R2019a

R2022a: Code generated by `generatePolicyFunction` now uses policy objects

Behavior change in future release

The code generated by `generatePolicyFunction` now loads a deployable policy object from a reinforcement learning agent. The results from running the generated policy function remain the same.

See Also

Functions
`generatePolicyBlock` | `train`

Objects
`rlMaxQPolicy` | `rlEpsilonGreedyPolicy` | `rlAdditiveNoisePolicy` | `rlDeterministicActorPolicy` | `rlStochasticActorPolicy` | `dlnetwork`

Blocks
Policy

Topics
"Train Reinforcement Learning Agents"

“Create Policies and Value Functions”
“Reinforcement Learning Agents”
“Deploy Trained Reinforcement Learning Policies”

generateRewardFunction

Generate a reward function from control specifications to train a reinforcement learning agent

Syntax

```
generateRewardFunction(mpcobj)
generateRewardFunction(blks)
generateRewardFunction( ___ ,'FunctionName',myFcnName)
```

Description

`generateRewardFunction(mpcobj)` generates a MATLAB reward function based on the cost and constraints defined in the linear or nonlinear MPC object `mpcobj`. The generated reward function is displayed in a new editor window and you can use it as a starting point for reward design. You can tune the weights, use a different penalty function, and then use the resulting reward function within an environment to train an agent.

This syntax requires Model Predictive Control Toolbox™ software.

`generateRewardFunction(blks)` generates a MATLAB reward function based on performance constraints defined in the model verification blocks specified in the array of block paths `blks`.

This syntax requires Simulink Design Optimization™ software.

`generateRewardFunction( ___ ,'FunctionName',myFcnName)` specifies the name of the generated reward function, and saves it into a file with the same name. It also overwrites any preexisting file with the same name in the current directory. Provide this name after either of the previous input arguments.

Examples

Generate a Reward Function from MPC object

This example shows how to generate a reinforcement learning reward function from an MPC object.

Define Plant and Create MPC Controller

Create a random plant using the `rss` function and set the feedthrough matrix to zero.

```
plant = rss(4,3,2);
plant.d = 0;
```

Specify which of the plant signals are manipulated variables, measured disturbances, measured outputs and unmeasured outputs.

```
plant = setmpcsignals(plant,"MV",1,"MD",2,"MO",[1 2],"UO",3);
```

Create an MPC controller with a sample time of 0.1 and prediction and control horizons of 10 and 3 steps, respectively.

```
mpcobj = mpc(plant,0.1,10,3);
```

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2 y3
```

Set limits and a scale factor for the manipulated variable.

```
mpcobj.ManipulatedVariables.Min = -2;
mpcobj.ManipulatedVariables.Max = 2;
mpcobj.ManipulatedVariables.ScaleFactor = 4;
```

Set weights for the quadratic cost function.

```
mpcobj.Weights.OutputVariables = [10 1 0.1];
mpcobj.Weights.ManipulatedVariablesRate = 0.2;
```

Generate the Reward Function

Generate the reward function code from specifications in the `mpc` object using `generateRewardFunction`. The code is displayed in the MATLAB Editor.

```
generateRewardFunction(mpcobj)
```

For this example, the is saved in the MATLAB function file `myMpcRewardFcn.m`. Display the generated reward function.

```
type myMpcRewardFcn.m
```

```
function reward = myMpcRewardFcn(y,refy,mv,refmv,lastmv)
% MYMPCREWARDFCN generates rewards from MPC specifications.
%
% Description of input arguments:
%
% y : Output variable from plant at step k+1
% refy : Reference output variable at step k+1
% mv : Manipulated variable at step k
% refmv : Reference manipulated variable at step k
% lastmv : Manipulated variable at step k-1
%
% Limitations (MPC and NLMPC):
%     - Reward computed based on first step in prediction horizon.
%       Therefore, signal previewing and control horizon settings are ignored.
%     - Online cost and constraint update is not supported.
%     - Custom cost and constraint specifications are not considered.
%     - Time varying cost weights and constraints are not supported.
%     - Mixed constraint specifications are not considered (for the MPC case).

% Reinforcement Learning Toolbox
% 27-May-2021 14:47:29

%#codegen

%% Specifications from MPC object
% Standard linear bounds as specified in 'States', 'OutputVariables', and
% 'ManipulatedVariables' properties
ymin = [-Inf -Inf -Inf];
ymax = [Inf Inf Inf];
```

```
mvmin = -2;
mvmax = 2;
mvratemin = -Inf;
mvratemax = Inf;

% Scale factors as specified in 'States', 'OutputVariables', and
% 'ManipulatedVariables' properties
Sy  = [1 1 1];
Smv = 4;

% Standard cost weights as specified in 'Weights' property
Qy      = [10 1 0.1];
Qmv     = 0;
Qmvrate = 0.2;

%% Compute cost
dy      = (refy(:)-y(:)) ./ Sy';
dmv     = (refmv(:)-mv(:)) ./ Smv';
dmvrate = (mv(:)-lastmv(:)) ./ Smv';
Jy      = dy'      * diag(Qy.^2)      * dy;
Jmv     = dmv'     * diag(Qmv.^2)     * dmv;
Jmvrate = dmvrate' * diag(Qmvrate.^2) * dmvrate;
Cost    = Jy + Jmv + Jmvrate;

%% Penalty function weight (specify nonnegative)
Wy = [1 1 1];
Wmv = 10;
Wmvrate = 10;

%% Compute penalty
% Penalty is computed for violation of linear bound constraints.
%
% To compute exterior bound penalty, use the exteriorPenalty function and
% specify the penalty method as 'step' or 'quadratic'.
%
% Alternaltively, use the hyperbolicPenalty or barrierPenalty function for
% computing hyperbolic and barrier penalties.
%
% For more information, see help for these functions.
%
% Set Pmv value to 0 if the RL agent action specification has
% appropriate 'LowerLimit' and 'UpperLimit' values.
Py      = Wy      * exteriorPenalty(y,ymin,ymax,'step');
Pmv     = Wmv     * exteriorPenalty(mv,mvmin,mvmax,'step');
Pmvrate = Wmvrate * exteriorPenalty(mv-lastmv,mvratemin,mvratemax,'step');
Penalty = Py + Pmv + Pmvrate;

%% Compute reward
reward = -(Cost + Penalty);
end
```

The calculated reward depends only on the current values of the plant input and output signals and their reference values, and it is composed of two parts.

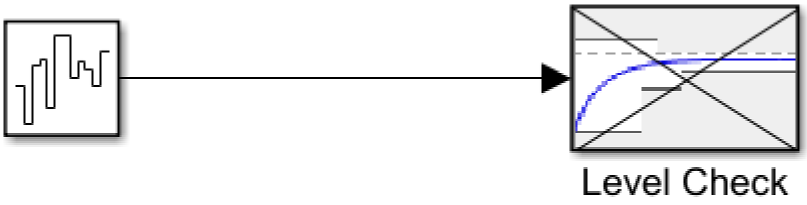
The first is a negative cost that depends on the squared difference between desired and current plant inputs and outputs. This part uses the cost function weights specified in the MPC object. The second part is a penalty that acts as a negative reward whenever the current plant signals violate the constraints.

The generated reward function is a starting point for reward design. You can tune the weights or use a different penalty function to define a more appropriate reward for your reinforcement learning agent.

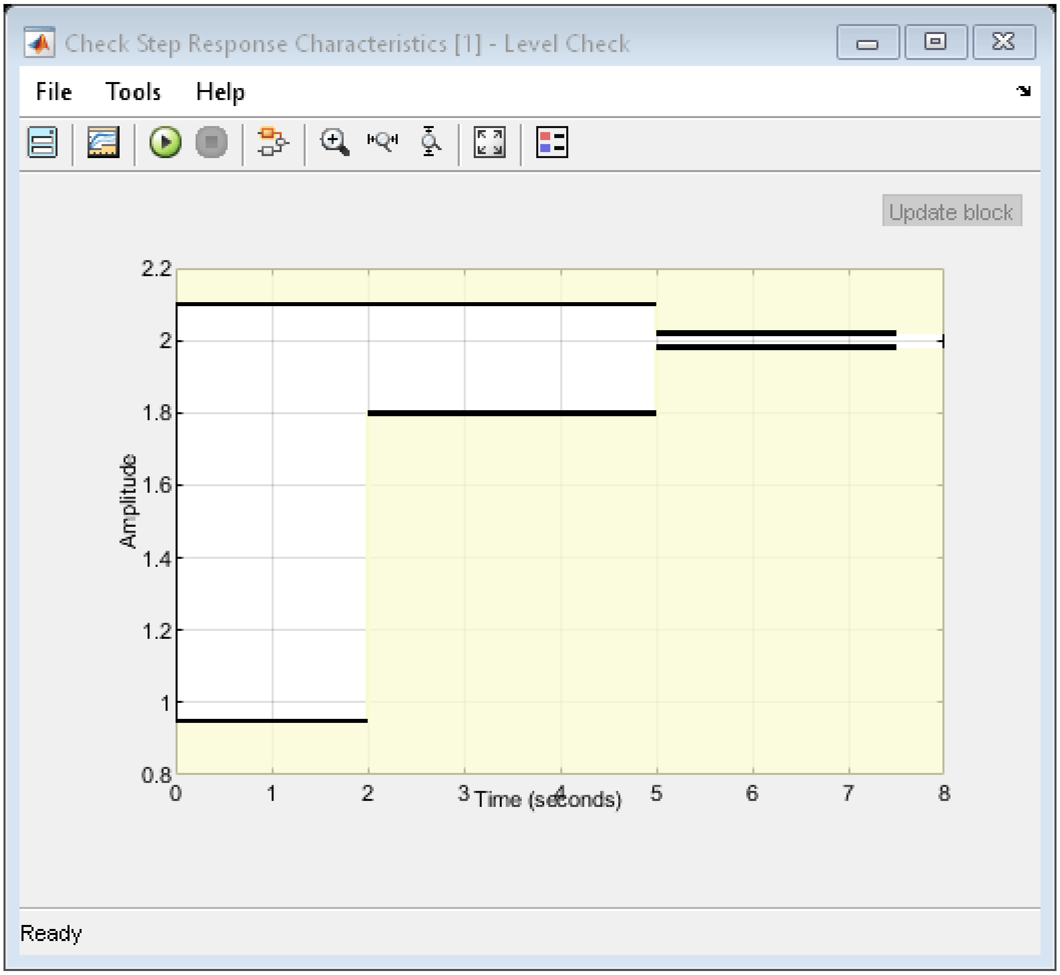
Generate Reward Function from Verification Block

This example shows how to generate a reinforcement learning reward function from a Simulink Design Optimization model verification block.

For this example, open the Simulink model `LevelCheckBlock.slx`, which contains a **Check Step Response Characteristics** block named **Level Check**.



```
open_system("LevelCheckBlock")
```



Generate the reward function code from specifications in the **Level Check** block, using `generateRewardFunction`. The code is displayed in the MATLAB Editor.

```
generateRewardFunction("LevelCheckBlock/Level Check")
```

For this example, the code is saved in the MATLAB function file `myBlockRewardFcn.m`.

Display the generated reward function.

```
type myBlockRewardFcn.m
```

```
function reward = myBlockRewardFcn(x,t)
% MYBLOCKREWARDFCN generates rewards from Simulink block specifications.
%
% x : Input of LevelCheckBlock/Level Check
% t : Simulation time (s)

% Reinforcement Learning Toolbox
% 27-May-2021 16:45:27

%#codegen

%% Specifications from LevelCheckBlock/Level Check
```

```
Block1_InitialValue = 1;
Block1_FinalValue = 2;
Block1_StepTime = 0;
Block1_StepRange = Block1_FinalValue - Block1_InitialValue;
Block1_MinRise = Block1_InitialValue + Block1_StepRange * 80/100;
Block1_MaxSettling = Block1_InitialValue + Block1_StepRange * (1+2/100);
Block1_MinSettling = Block1_InitialValue + Block1_StepRange * (1-2/100);
Block1_MaxOvershoot = Block1_InitialValue + Block1_StepRange * (1+10/100);
Block1_MinUndershoot = Block1_InitialValue - Block1_StepRange * 5/100;

if t >= Block1_StepTime
    if Block1_InitialValue <= Block1_FinalValue
        Block1_UpperBoundTimes = [0,5; 5,max(5+1,t+1)];
        Block1_UpperBoundAmplitudes = [Block1_MaxOvershoot,Block1_MaxOvershoot; Block1_MaxSettlin
        Block1_LowerBoundTimes = [0,2; 2,5; 5,max(5+1,t+1)];
        Block1_LowerBoundAmplitudes = [Block1_MinUndershoot,Block1_MinUndershoot; Block1_MinRise
    else
        Block1_UpperBoundTimes = [0,2; 2,5; 5,max(5+1,t+1)];
        Block1_UpperBoundAmplitudes = [Block1_MinUndershoot,Block1_MinUndershoot; Block1_MinRise
        Block1_LowerBoundTimes = [0,5; 5,max(5+1,t+1)];
        Block1_LowerBoundAmplitudes = [Block1_MaxOvershoot,Block1_MaxOvershoot; Block1_MaxSettlin
    end

    Block1_xmax = zeros(1,size(Block1_UpperBoundTimes,1));
    for idx = 1:numel(Block1_xmax)
        tseg = Block1_UpperBoundTimes(idx,:);
        xseg = Block1_UpperBoundAmplitudes(idx,:);
        Block1_xmax(idx) = interp1(tseg,xseg,t,'linear',NaN);
    end
    if all(isnan(Block1_xmax))
        Block1_xmax = Inf;
    else
        Block1_xmax = max(Block1_xmax,[],'omitnan');
    end

    Block1_xmin = zeros(1,size(Block1_LowerBoundTimes,1));
    for idx = 1:numel(Block1_xmin)
        tseg = Block1_LowerBoundTimes(idx,:);
        xseg = Block1_LowerBoundAmplitudes(idx,:);
        Block1_xmin(idx) = interp1(tseg,xseg,t,'linear',NaN);
    end
    if all(isnan(Block1_xmin))
        Block1_xmin = -Inf;
    else
        Block1_xmin = max(Block1_xmin,[],'omitnan');
    end
else
    Block1_xmin = -Inf;
    Block1_xmax = Inf;
end




%% Penalty function weight (specify nonnegative)
Weight = 1;

%% Compute penalty
```

```
% Penalty is computed for violation of linear bound constraints.
%
% To compute exterior bound penalty, use the exteriorPenalty function and
% specify the penalty method as 'step' or 'quadratic'.
%
% Alternaltively, use the hyperbolicPenalty or barrierPenalty function for
% computing hyperbolic and barrier penalties.
%
% For more information, see help for these functions.
Penalty = sum(exteriorPenalty(x,Block1_xmin,Block1_xmax,'step'));

%% Compute reward
reward = -Weight * Penalty;
end
```

The generated reward function takes as input arguments the current value of the verification block input signals and the simulation time. A negative reward is calculated using a weighted penalty that acts whenever the current block input signals violate the linear bound constraints defined in the verification block.

The generated reward function is a starting point for reward design. You can tune the weights or use a different penalty function to define a more appropriate reward for your reinforcement learning agent.

Close the Simulink model.

```
close_system("LevelCheckBlock")
```

Input Arguments

mpcobj — Linear or nonlinear MPC object

mpc object | nlmpc object

Linear or nonlinear MPC object, specified as an mpc object or an nlmpc object, respectively.

Note that:

- The generated function calculates rewards using signal values at the current time only. Predicted future values, signal previewing, and control horizon settings are not used in the reward calculation.
- Using time-varying cost weights and constraints, or updating them online, is not supported.
- Only the standard quadratic cost function, as described in “Optimization Problem” (Model Predictive Control Toolbox), is supported. Therefore, for mpc objects, using mixed constraint specifications is not supported. Similarly, for nlmpc objects, custom cost and constraint specifications are not supported.

Example: mpc(tf([1 1],[1 2 0]),0.1)

blks — Path to model verification blocks

char array | cell array | string array

Path to model verification blocks, specified as character array, cell array or string array. The supported Simulink Design Optimization model verification blocks are the following ones.

- Check Against Reference (Simulink Design Optimization)
- Check Custom Bounds (Simulink Design Optimization)
- Check Step Response Characteristics (Simulink Design Optimization)
- Any block belonging to the Simulink “Model Verification” (Simulink) library

The generated reward function takes as input arguments the current value of the verification block input signals and the simulation time. A negative reward is calculated using a weighted penalty that acts whenever the current block input signals violate the linear bound constraints defined in the verification block.

Example: `"mySimulinkModel02/Check Against Reference"`

`myFcnName` — Function name
`string` | `char` vector

Function name, specified as a string object or character vector.

Example: `"reward03epf_step"`

Tips

By default, the exterior bound penalty function `exteriorPenalty` is used to calculate the penalty. Alternatively, to calculate hyperbolic and barrier penalties, you can use the `hyperbolicPenalty` or `barrierPenalty` functions.

Version History

Introduced in R2021b

See Also

Functions
`exteriorPenalty` | `hyperbolicPenalty` | `barrierPenalty`

Objects
`mpc` | `nlmpc`

Topics
“Generate Reward Function from a Model Predictive Controller for a Servomotor”
“Generate Reward Function from a Model Verification Block for a Water Tank System”
“Define Reward Signals”
“Create MATLAB Reinforcement Learning Environments”
“Create Simulink Reinforcement Learning Environments”

getActionInfo

Package: `rl.env`

Obtain action data specifications from reinforcement learning environment, agent, or experience buffer

Syntax

```
actInfo = getActionInfo(env)
actInfo = getActionInfo(agent)
actInfo = getActionInfo(buffer)
```

Description

`actInfo = getActionInfo(env)` extracts action information from reinforcement learning environment `env`.

`actInfo = getActionInfo(agent)` extracts action information from reinforcement learning agent `agent`.

`actInfo = getActionInfo(buffer)` extracts action information from experience buffer `buffer`.

Examples

Extract Action and Observation Specifications from Reinforcement Learning Environment

The reinforcement learning environment for this example is a longitudinal dynamics model comprising two cars, a leader and a follower. The vehicle model is also used in the “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox) example.

Open the model.

```
mdl = "rlACCMdl";
open_system(mdl);
```

Specify path to the agent block in the model.

```
agentblk = mdl + "/RL Agent";
```

Create the observation and action specifications.

```
% Observation specifications
obsInfo = rlNumericSpec([3 1],LowerLimit=-inf*ones(3,1),UpperLimit=inf*ones(3,1));
obsInfo.Name = "observations";
obsInfo.Description = "information on velocity error and ego velocity";

% Action specifications
actInfo = rlNumericSpec([1 1],LowerLimit=-3,UpperLimit=2);
actInfo.Name = "acceleration";
```

Define environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlACCMdl
      AgentBlock : rlACCMdl/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

The reinforcement learning environment `env` is a `SimulinkEnvWithAgent` object.

Extract the action and observation specifications from `env`.

```
actInfoExt = getActionInfo(env)
```

```
actInfoExt = 
  rlNumericSpec with properties:

     LowerLimit: -3
     UpperLimit: 2
           Name: "acceleration"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

```
obsInfoExt = getObservationInfo(env)
```

```
obsInfoExt = 
  rlNumericSpec with properties:

     LowerLimit: [3x1 double]
     UpperLimit: [3x1 double]
           Name: "observations"
    Description: "information on velocity error and ego velocity"
      Dimension: [3 1]
       DataType: "double"
```

The action information contains acceleration values while the observation information contains the velocity and velocity error values of the ego vehicle.

Input Arguments

env — Reinforcement learning environment

`rlFunctionEnv` object | `SimulinkEnvWithAgent` object | `rlNeuralNetworkEnvironment` object | predefined MATLAB environment object

Reinforcement learning environment from which to extract the action information, specified as one of the following:

- MATLAB environment represented as one of the following objects.
  - `rlFunctionEnv`

  - `rlNeuralNetworkEnvironment`
  - Predefined MATLAB environment created using `rlPredefinedEnv`
- Simulink environment represented as a `SimulinkEnvWithAgent` object.

For more information on reinforcement learning environments, see “Create MATLAB Reinforcement Learning Environments” and “Create Simulink Reinforcement Learning Environments”.

agent — Reinforcement learning agent
`rlQAgent` object | `rlSARSAAgent` object | `rlDQNAgent` object | `rlPGAgent` object | `rlDDPGAgent` object | `rlTD3Agent` object | `rlACAgent` object | `rlPPOAgent` object | `rlTRPOAgent` object | `rlSACAgent` object | `rlMBPOAgent` object

Reinforcement learning agent from which to extract the action information, specified as one of the following objects.

- `rlQAgent`
- `rlSARSAAgent`
- `rlDQNAgent`
- `rlDDPGAgent`
- `rlTD3Agent`
- `rlPGAgent`
- `rlACAgent`
- `rlPPOAgent`
- `rlTRPOAgent`
- `rlSACAgent`
- `rlMBPOAgent`

For more information on reinforcement learning agents, see “Reinforcement Learning Agents”.

buffer — Experience buffer
`rlReplayMemory` object | `rlPrioritizedReplayMemory` object | `rlHindsightReplayMemory` object | `rlHindsightPrioritizedReplayMemory` object

Experience buffer, specified as one of the following replay memory objects.

- `rlReplayMemory`
- `rlPrioritizedReplayMemory`
- `rlHindsightReplayMemory`
- `rlHindsightPrioritizedReplayMemory`

Output Arguments

actInfo — Action data specifications
array of `rlNumericSpec` objects | array of `rlFiniteSetSpec` objects

Action data specifications extracted from the reinforcement learning environment, returned as an array of one of the following:

- `rlNumericSpec` objects
- `rlFiniteSetSpec` objects
- A mix of `rlNumericSpec` and `rlFiniteSetSpec` objects

Version History

Introduced in R2019a

See Also

Functions
`getObservationInfo`

Objects
`rlNumericSpec` | `rlFiniteSetSpec` | `rlQAgent` | `rlSARSAAgent` | `rlDQNAgent` | `rlPGAgent` | `rlACAgent` | `rlDDPGAgent`

Topics
"Create Simulink Reinforcement Learning Environments"
"Reinforcement Learning Agents"

getAction

Package: `rl.policy`

Obtain action from agent, actor, or policy object given environment observations

Syntax

```
action = getAction(agent,obs)
[action,agent] = getAction(agent,obs)

action = getAction(actor,obs)
[action,nextState] = getAction(actor,obs)

action = getAction(policy,obs)
[action,updatedPolicy] = getAction(policy,obs)
```

Description

Agent

`action = getAction(agent,obs)` returns the action generated from the policy of a reinforcement learning agent, given environment observations. If `agent` contains internal states, they are updated.

`[action,agent] = getAction(agent,obs)` also returns the updated agent as an output argument.

Actor

`action = getAction(actor,obs)` returns the action generated from the policy represented by the actor `actor`, given environment observations `obs`.

`[action,nextState] = getAction(actor,obs)` also returns the updated state of the actor when it uses a recurrent neural network.

Policy

`action = getAction(policy,obs)` returns the action generated from the policy object `policy`, given environment observations.

`[action,updatedPolicy] = getAction(policy,obs)` also returns the updated policy as an output argument (any internal state of the policy, if used, is updated).

Examples

Get Actions from Agent

Create an environment with a discrete action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”.

```
% load predefined environment
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");
```

Obtain the observation and action specifications for this environment.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a TRPO agent from the environment observation and action specifications.

```
agent = rlTRPOAgent(obsInfo,actInfo);
```

Use `getAction` to return the action from a random observation.

```
getAction(agent, ...
    {rand(obsInfo(1).Dimension), ...
     rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[-2]}
```

You can also obtain actions for a batch of observations. For example, obtain actions for a batch of 10 observations.

```
actBatch = getAction(agent, ...
    {rand([obsInfo(1).Dimension 10]), ...
     rand([obsInfo(2).Dimension 10])});
size(actBatch{1})
```

```
ans = 1×3

     1     1    10
```

```
actBatch{1}(1,1,7)
```

```
ans = -2
```

`actBatch` contains one action for each observation in the batch.

Get Action from Deterministic Actor

Create observation and action information. You can also obtain these specifications from an environment.

```
obsinfo = rlNumericSpec([4 1]);
actinfo = rlNumericSpec([2 1]);
```

Create a deep neural network for the actor.

```
net = [featureInputLayer(obsinfo.Dimension(1), ...
           "Normalization","none","Name","state")
       fullyConnectedLayer(10,"Name","fc1")
       reluLayer("Name","relu1")
```

```
        fullyConnectedLayer(20,"Name","CriticStateFC2")
        fullyConnectedLayer(actinfo.Dimension(1),"Name","fc2")
        tanhLayer("Name","tanh1")];
net = dlnetwork(net);
```

Create a deterministic actor representation for the network.

```
actor = rlContinuousDeterministicActor(net, ...
    obsinfo,actinfo,...
    ObservationInputNames={"state"});
```

Obtain an action from this actor for a random batch of 20 observations.

```
act = getAction(actor,{rand(4,1,10)})
```

```
act = 1x1 cell array
    {2x1x10 single}
```

`act` is a single cell array that contains the two computed actions for all 10 observations in the batch.

```
act{1}(:,1,7)
```

```
ans = 2x1 single column vector

    0.2643
   -0.2934
```

Get Actions from Policy Object

Create observation and action specification objects. For this example, define the observation and action spaces as continuous four- and two-dimensional spaces, respectively.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlNumericSpec([2 1]);
```

Alternatively, you can use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment.

Create a continuous deterministic actor. This actor must accept an observation as input and return an action as output.

To approximate the policy function within the actor, use a recurrent deep neural network model. Define the network as an array of layer objects, and get the dimension of the observation and action spaces from the environment specification objects. To create a recurrent network, use a `sequenceInputLayer` as the input layer (with size equal to the number of dimensions of the observation channel) and include at least one `lstmLayer`.

```
layers = [
    sequenceInputLayer(obsInfo.Dimension(1))
    lstmLayer(2)
    reluLayer
    fullyConnectedLayer(actInfo.Dimension(1))
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
model = dlnetwork(layers);
summary(model)
```

```
   Initialized: true

   Number of learnables: 62

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create the actor using model, and the observation and action specifications.

```
actor = rlContinuousDeterministicActor(model,obsInfo,actInfo)
```

```
actor = 
  rlContinuousDeterministicActor with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

Check the actor with a random observation input.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = 2x1 single column vector

    0.0568
    0.0691
```

Create an additive noise policy object from `actor`.

```
policy = rlAdditiveNoisePolicy(actor)
```

```
policy = 
  rlAdditiveNoisePolicy with properties:

               Actor: [1x1 rl.function.rlContinuousDeterministicActor]
           NoiseType: "gaussian"
        NoiseOptions: [1x1 rl.option.GaussianActionNoise]
    EnableNoiseDecay: 1
      UseNoisyAction: 1
     ObservationInfo: [1x1 rl.util.rlNumericSpec]
          ActionInfo: [1x1 rl.util.rlNumericSpec]
          SampleTime: -1
```

Use dot notation to set the standard deviation decay rate.

```
policy.NoiseOptions.StandardDeviationDecayRate = 0.9;
```

Use `getAction` to generate an action from the policy, given a random observation input.

```
act = getAction(policy,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = 2×1

    0.5922
   -0.3745
```

Display the state of the recurrent neural network in the policy object.

```
xNN = getRNNState(policy);
xNN{1}
```

```
ans = 2x1 single column vector

     0
     0
```

Use `getAction` to also return the updated policy as a second argument.

```
[act, updatedPolicy] = getAction(policy,{rand(obsInfo.Dimension)});
```

Display the state of the recurrent neural network in the updated policy object.

```
xpNN = getRNNState(updatedPolicy);
xpNN{1}
```

```
ans = 2x1 single column vector

    0.3327
   -0.2479
```

As expected, the state is updated.

Input Arguments

agent — Reinforcement learning agent

reinforcement learning agent object

Reinforcement learning agent, specified as one of the following objects:

- `rlQAgent`
- `rlSARSAAgent`
- `rlDQNAgent`
- `rlPGAgent`
- `rlDDPGAgent`
- `rlTD3Agent`
- `rlACAgent`
- `rlSACAgent`
- `rlPPOAgent`
- `rlTRPOAgent`
- Custom agent — For more information, see “Create Custom Reinforcement Learning Agents”.

Note `agent` is an handle object, so it is updated whether it is returned as an output argument or not. For more information about handle objects, see "Handle Object Behavior".

`actor` — Actor

`rlContinousDeterministicActor` object | `rlContinousGaussianActor` object | `rlDiscreteCategoricalActor` object

Actor, specified as an `rlContinuousDeterministicActor`, `rlDiscreteCategoricalActor` or `rlContinuousGaussianActor` object.

`policy` — Reinforcement learning policy

`rlMaxQPolicy` | `rlEpsilonGreedyPolicy` | `rlDeterministicActorPolicy` | `rlAdditiveNoisePolicy` | `rlStochasticActorPolicy`

Reinforcement learning policy, specified as one of the following objects:

- `rlMaxQPolicy`
- `rlEpsilonGreedyPolicy`
- `rlDeterministicActorPolicy`
- `rlAdditiveNoisePolicy`
- `rlStochasticActorPolicy`

`obs` — Environment observations

cell array

Environment observations, specified as a cell array with as many elements as there are observation input channels. Each element of `obs` contains an array of observations for a single observation input channel.

The dimensions of each element in `obs` are M_O-by-L_B-by-L_S, where:

- M_O corresponds to the dimensions of the associated observation input channel.
- L_B is the batch size. To specify a single observation, set $L_B = 1$. To specify a batch of observations, specify $L_B > 1$. If `valueRep` or `qValueRep` has multiple observation input channels, then L_B must be the same for all elements of `obs`.
- L_S specifies the sequence length for a recurrent neural network. If `valueRep` or `qValueRep` does not use a recurrent neural network, then $L_S = 1$. If `valueRep` or `qValueRep` has multiple observation input channels, then L_S must be the same for all elements of `obs`.

L_B and L_S must be the same for both `act` and `obs`.

For more information on input and output formats for recurrent neural networks, see the Algorithms section of `lstmLayer`.

Output Arguments

`action` — Action

single-element cell array

Action, returned as single-element cell array containing an array with dimensions M_A-by-L_B-by-L_S, where:

- M_A corresponds to the dimensions of the associated action specification.
- L_B is the batch size.
- L_S is the sequence length for recurrent neural networks. If the agent, actor, or policy calculating `action` do not use recurrent neural networks, then $L_S = 1$.

Note The following continuous action-space actor, policy and agent objects do not enforce the constraints set by the action specification:

- `rlContinuousDeterministicActor`
- `rlStochasticActorPolicy`
- `rlACAgent`
- `rlPGAgent`
- `rlPPOAgent`

In these cases, you must enforce action space constraints within the environment.

`nextState` — Next state of the actor
cell array

Next state of the actor, returned as a cell array. If `actor` does not use a recurrent neural network, then `state` is an empty cell array.

You can set the state of the representation to `state` using the `setState` function. For example:

```
actor = setState(actor,state);
```

`agent` — Updated agent
reinforcement learning agent object

Updated agent, returned as the same agent object as the agent in the input argument. Note that `agent` is an handle object. Therefore, its internal states (if any) are updated whether `agent` is returned as an output argument or not. For more information about handle objects, see “Handle Object Behavior”.

`updatedPolicy` — Updated policy object
reinforcement learning policy object

Updated policy object. It is identical to the policy object supplied as first input argument, except that its internal states (if any) are updated.

Tips

The function `evaluate` behaves, for actor objects, similarly to `getAction` except for the following differences.

- For an `rlDiscreteCategoricalActor` actor object, `evaluate` returns the probability of each possible actions, (instead of a sample action as `getAction`).
- For an `rlContinuousGaussianActor` actor object, `evaluate` returns the mean and standard deviation of the Gaussian distribution, (instead of a sample action as `getAction`).

para

Version History

Introduced in R2020a

See Also

Functions

`evaluate` | `getValue` | `getMaxQValue`

Topics

“Train Reinforcement Learning Policy Using Custom Training Loop”
“Create Policies and Value Functions”
“Reinforcement Learning Agents”
“Create Custom Reinforcement Learning Agents”

getActor

Package: `rl.agent`

Extract actor from reinforcement learning agent

Syntax

```
actor = getActor(agent)
```

Description

`actor = getActor(agent)` returns the actor object from the specified reinforcement learning agent.

Examples

Modify Actor Parameter Values

Assume that you have an existing trained reinforcement learning agent. For this example, load the trained agent from “Train DDPG Agent to Control Double Integrator System”.

```
load("DoubleIntegDDPG.mat","agent")
```

Obtain the actor function approximator from the agent.

```
actor = getActor(agent);
```

Obtain the learnable parameters from the actor.

```
params = getLearnableParameters(actor)
```

```
params=2×1 cell array
    {[-15.4601 -7.2076]}
    {[               0]}
```

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,"UniformOutput",false);
```

Set the parameter values of the actor to the new modified values.

```
actor = setLearnableParameters(actor,modifiedParams);
```

Set the actor in the agent to the new modified actor.

```
setActor(agent,actor);
```

Display the new parameter values.

```
getLearnableParameters(getActor(agent))
```

```
ans=2×1 cell array
    {[-30.9201 -14.4153]}
    {[                0]}
```

Modify Deep Neural Networks in Reinforcement Learning Agent

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”.

Load the predefined environment.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a PPO agent from the environment observation and action specifications. This agent uses default deep neural networks for its actor and critic.

```
agent = rlPPOAgent(obsInfo,actInfo);
```

To modify the deep neural networks within a reinforcement learning agent, you must first extract the actor and critic function approximators.

```
actor = getActor(agent);
critic = getCritic(agent);
```

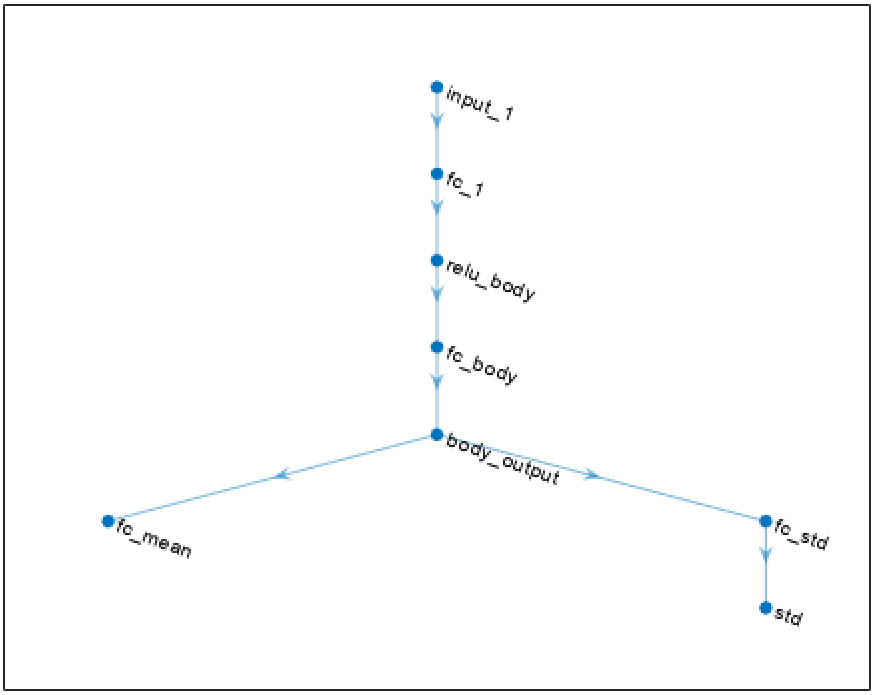
Extract the deep neural networks from both the actor and critic function approximators.

```
actorNet = getModel(actor);
criticNet = getModel(critic);
```

The networks are `dlnetwork` objects. To view them using the `plot` function, you must convert them to `layerGraph` objects.

For example, view the actor network.

```
plot(layerGraph(actorNet))
```



To validate a network, use `analyzeNetwork`. For example, validate the critic network.

```
analyzeNetwork(criticNet)
```

You can modify the actor and critic networks and save them back to the agent. To modify the networks, you can use the Deep Network Designer app. To open the app for each network, use the following commands.

```
deepNetworkDesigner(layerGraph(criticNet))
deepNetworkDesigner(layerGraph(actorNet))
```

In **Deep Network Designer**, modify the networks. For example, you can add additional layers to your network. When you modify the networks, do not change the input and output layers of the networks returned by `getModel`. For more information on building networks, see “Build Networks with Deep Network Designer”.

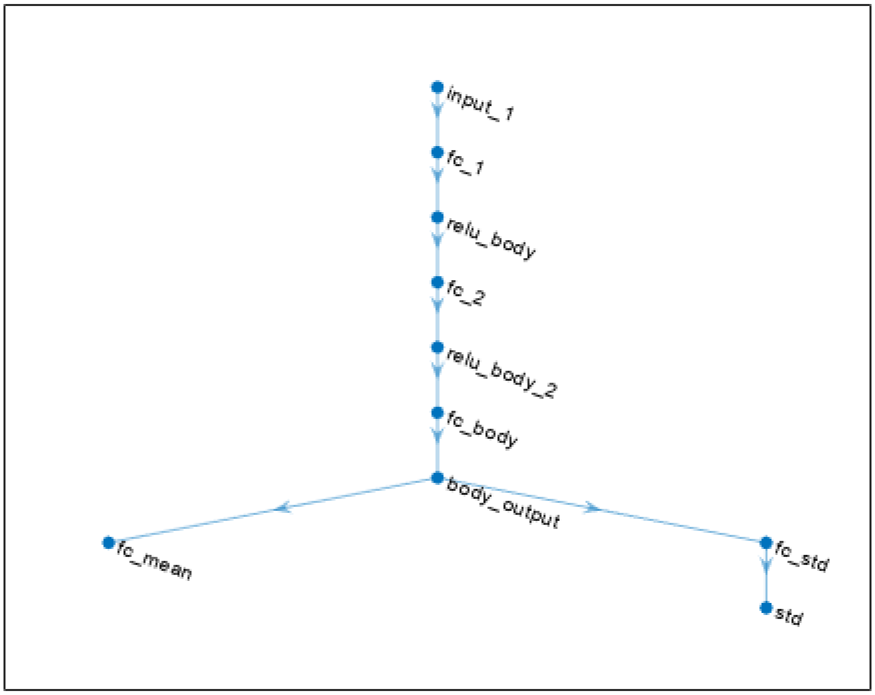
To validate the modified network in **Deep Network Designer**, you must click on **Analyze for dlnetwork**, under the **Analysis** section. To export the modified network structures to the MATLAB® workspace, generate code for creating the new networks and run this code from the command line. Do not use the exporting option in **Deep Network Designer**. For an example that shows how to generate and run code, see “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”.

For this example, the code for creating the modified actor and critic networks is in the `createModifiedNetworks` helper script.

```
createModifiedNetworks
```

Each of the modified networks includes an additional `fullyConnectedLayer` and `reluLayer` in their main common path. View the modified actor network.

```
plot(layerGraph(modifiedActorNet))
```



After exporting the networks, insert the networks into the actor and critic function approximators.

```
actor = setModel(actor,modifiedActorNet);
critic = setModel(critic,modifiedCriticNet);
```

Finally, insert the modified actor and critic function approximators into the actor and critic objects.

```
agent = setActor(agent,actor);
agent = setCritic(agent,critic);
```

Input Arguments

agent — Reinforcement learning agent
`rlDDPGAgent` object | `rlTD3Agent` object | `rlPGAgent` object | `rlACAgent` object | `rlPPOAgent` object | `rlSACAgent` object

Reinforcement learning agent that contains an actor, specified as one of the following:

- `rlPGAgent` object
- `rlDDPGAgent` object

- `rlTD3Agent` object
- `rlACAgent` object
- `rlSACAgent` object
- `rlPPOAgent` object
- `rlTRPOAgent` object

Note if `agent` is an `rlMBPOAgent` object, to extract the actor, use `getActor(agent.BaseAgent)`.

Output Arguments

`actor` — Actor
`rlContinuousDeterministicActor` object | `rlDiscreteCategoricalActor` object | `rlContinuousGaussianActor` object

Actor object, returned as one of the following:

- `rlContinuousDeterministicActor` object — Returned when `agent` is an `rlDDPGAgent` or `rlTD3Agent` object
- `rlDiscreteCategoricalActor` object — Returned when `agent` is an `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, `rlTRPOAgent` or `rlSACAgent` object for an environment with a discrete action space.
- `rlContinuousGaussianActor` object — Returned when `agent` is an `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, `rlTRPOAgent` or `rlSACAgent` object for an environment with a continuous action space.

Version History

Introduced in R2019a

See Also

Functions

`getCritic` | `setActor` | `setCritic` | `getModel` | `setModel` | `getLearnableParameters` | `setLearnableParameters`

Topics

"Create Policies and Value Functions"
"Import Neural Network Models"

getCritic

Package: `rl.agent`

Extract critic from reinforcement learning agent

Syntax

```
critic = getCritic(agent)
```

Description

`critic = getCritic(agent)` returns the critic object from the specified reinforcement learning agent.

Examples

Modify Critic Parameter Values

Assume that you have an existing trained reinforcement learning agent. For this example, load the trained agent from “Train DDPG Agent to Control Double Integrator System”.

```
load("DoubleIntegDDPG.mat","agent")
```

Obtain the critic function approximator from the agent.

```
critic = getCritic(agent);
```

Obtain the learnable parameters from the critic.

```
params = getLearnableParameters(critic)
```

```
params=2×1 cell array
    {[-4.9869 -1.5577 -0.3351 -0.1097 -0.0444 7.5240e-04]}
    {[                                                 0]}
```

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,"UniformOutput",false);
```

Set the parameter values of the critic to the new modified values.

```
critic = setLearnableParameters(critic,modifiedParams);
```

Set the critic in the agent to the new modified critic.

```
setCritic(agent,critic);
```

Display the new parameter values.

```
getLearnableParameters(getCritic(agent))
```

```
ans=2×1 cell array
    {[-9.9737 -3.1153 -0.6702 -0.2194 -0.0888 0.0015]}
    {[                                            0]}
```

Modify Deep Neural Networks in Reinforcement Learning Agent

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”.

Load the predefined environment.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a PPO agent from the environment observation and action specifications. This agent uses default deep neural networks for its actor and critic.

```
agent = rlPPOAgent(obsInfo,actInfo);
```

To modify the deep neural networks within a reinforcement learning agent, you must first extract the actor and critic function approximators.

```
actor = getActor(agent);
critic = getCritic(agent);
```

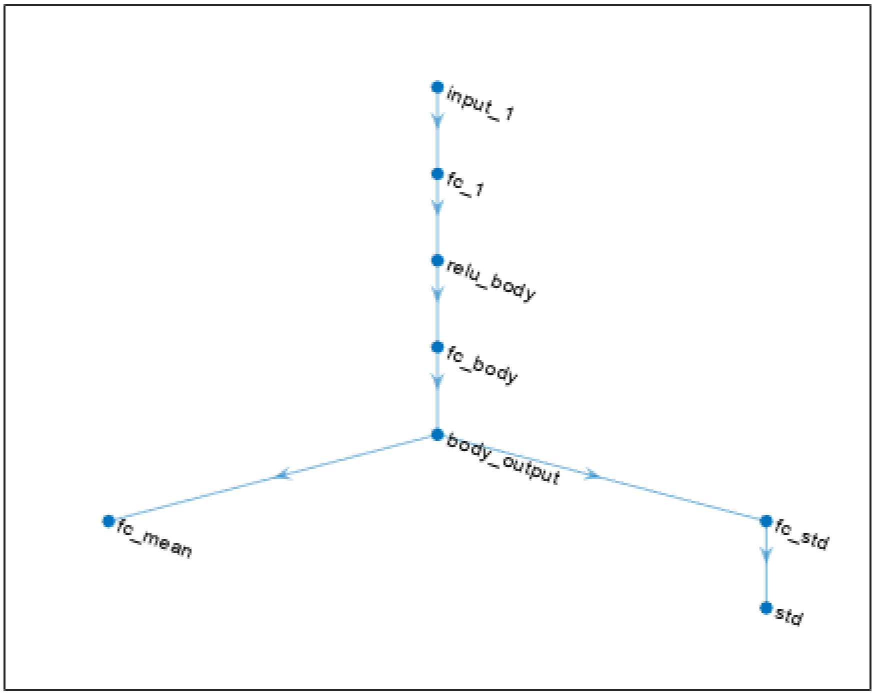
Extract the deep neural networks from both the actor and critic function approximators.

```
actorNet = getModel(actor);
criticNet = getModel(critic);
```

The networks are `dlnetwork` objects. To view them using the `plot` function, you must convert them to `layerGraph` objects.

For example, view the actor network.

```
plot(layerGraph(actorNet))
```



To validate a network, use `analyzeNetwork`. For example, validate the critic network.

```
analyzeNetwork(criticNet)
```

You can modify the actor and critic networks and save them back to the agent. To modify the networks, you can use the Deep Network Designer app. To open the app for each network, use the following commands.

```
deepNetworkDesigner(layerGraph(criticNet))
deepNetworkDesigner(layerGraph(actorNet))
```

In **Deep Network Designer**, modify the networks. For example, you can add additional layers to your network. When you modify the networks, do not change the input and output layers of the networks returned by `getModel`. For more information on building networks, see “Build Networks with Deep Network Designer”.

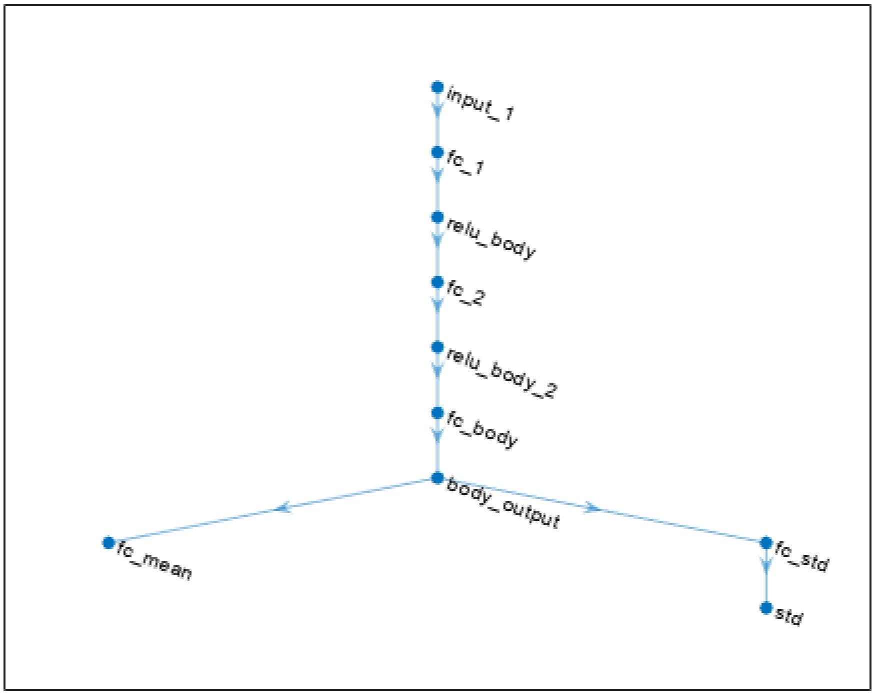
To validate the modified network in **Deep Network Designer**, you must click on **Analyze for dlnetwork**, under the **Analysis** section. To export the modified network structures to the MATLAB® workspace, generate code for creating the new networks and run this code from the command line. Do not use the exporting option in **Deep Network Designer**. For an example that shows how to generate and run code, see “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”.

For this example, the code for creating the modified actor and critic networks is in the `createModifiedNetworks` helper script.

```
createModifiedNetworks
```

Each of the modified networks includes an additional `fullyConnectedLayer` and `reluLayer` in their main common path. View the modified actor network.

```
plot(layerGraph(modifiedActorNet))
```



After exporting the networks, insert the networks into the actor and critic function approximators.

```
actor = setModel(actor,modifiedActorNet);
critic = setModel(critic,modifiedCriticNet);
```

Finally, insert the modified actor and critic function approximators into the actor and critic objects.

```
agent = setActor(agent,actor);
agent = setCritic(agent,critic);
```

Input Arguments

agent — Reinforcement learning agent

reinforcement learning agent object

Reinforcement learning agent that contains a critic, specified as one of the following objects:

- `rlQAgent`
- `rlSARSAAgent`

- rlDQNAgent
- rlPGAgent (when using a critic to estimate a baseline value function)
- rlDDPGAgent
- rlTD3Agent
- rlACAgent
- rlSACAgent
- rlPPOAgent
- rlTRPOAgent
- rlMBPOAgent

Output Arguments

critic — Critic
rlValueFunction object | rlQValueFunction object | rlVectorQValueFunction object | two-element row vector of rlQValueFunction objects

Critic object, returned as one of the following:

- rlValueFunction object — Returned when agent is an rlACAgent, rlPGAgent, or rlPPOAgent object.
- rlQValueFunction object — Returned when agent is an rlQAgent, rlSARSAAgent, rlDQNAgent, rlDDPGAgent, or rlTD3Agent object with a single critic.
- rlVectorQValueFunction object — Returned when agent is an rlQAgent, rlSARSAAgent, rlDQNAgent, object with a discrete action space, vector Q-value function critic.
- Two-element row vector of rlQValueFunction objects — Returned when agent is an rlTD3Agent or rlSACAgent object with two critics.

Version History

Introduced in R2019a

See Also

Functions
getActor | setActor | setCritic | getModel | setModel | getLearnableParameters | setLearnableParameters

Topics
“Create Policies and Value Functions”
“Import Neural Network Models”

getExplorationPolicy

Package: rl.agent

Extract exploratory (stochastic) policy object from agent

Syntax

```
policy = getExplorationPolicy(agent)
```

Description

policy = getExplorationPolicy(agent) returns a stochastic policy object from the specified reinforcement learning agent. Stochastic polices are useful for exploration.

Examples

Extract Policy Object from Agent

For this example, load the PG agent trained in “Train PG Agent to Balance Cart-Pole System”.

```
load("MATLABCartpolePG.mat","agent")
```

Extract the agent greedy policy using getGreedyPolicy.

```
policyDtr = getGreedyPolicy(agent)
```

```
policyDtr = 
  rlStochasticActorPolicy with properties:

                    Actor: [1x1 rl.function.rlDiscreteCategoricalActor]
    UseMaxLikelihoodAction: 1
           ObservationInfo: [1x1 rl.util.rlNumericSpec]
                ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
                SampleTime: 1
```

Note that, in the extracted policy object, the UseMaxLikelihoodAction property is set to true. This means that the policy object always generates the maximum likelihood action in response to a given observation, and is therefore greedy (and deterministic).

Alternatively, you can extract a stochastic policy using getExplorationPolicy.

```
policyXpl = getExplorationPolicy(agent)
```

```
policyXpl = 
  rlStochasticActorPolicy with properties:

                    Actor: [1x1 rl.function.rlDiscreteCategoricalActor]
    UseMaxLikelihoodAction: 0
           ObservationInfo: [1x1 rl.util.rlNumericSpec]
                ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
                SampleTime: 1
```

This time, the extracted policy object has the `UseMaxLikelihoodAction` property is set to `false`. This means that the policy object generates a random action, given an observation. The policy is therefore stochastic and useful for exploration.

Input Arguments

`agent` — Reinforcement learning agent
reinforcement learning agent object

Reinforcement learning agent that contains a critic, specified as one of the following objects:

- `rlQAgent`
- `rlSARSAAgent`
- `rlDQNAgent`
- `rlPGAgent` (when using a critic to estimate a baseline value function)
- `rlDDPGAgent`
- `rlTD3Agent`
- `rlACAgent`
- `rlSACAgent`
- `rlPPOAgent`
- `rlTRPOAgent`
- `rlMBPOAgent`

Note if `agent` is an `rlMBPOAgent` object, to extract the greedy policy, use `getGreedyPolicy(agent.BaseAgent)`.

Output Arguments

`policy` — Reinforcement learning policy object
`rlEpsilonGreedyPolicy` object | `rlAdditiveNoisePolicy` object | `rlStochasticActorPolicy` object

Policy object, returned as one of the following:

- `rlEpsilonGreedyPolicy` object — Returned when `agent` is an `rlQAgent`, `rlSARSAAgent`, or `rlDQNAgent` object.
- `rlAdditiveNoisePolicy` object — Returned when `agent` is an `rlDDPGAgent` or `rlTD3Agent` object.
- `rlStochasticActorPolicy` object, with the `UseMaxLikelihoodAction` set to `false` — Returned when `agent` is an `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, `rlTRPOAgent` or `rlSACAgent` object. Since the returned policy object has the `UseMaxLikelihoodAction` property set to `false`, it always generates a random action (according to the policy probability distribution) as a response to a given observation, and is therefore exploratory (and stochastic).

Version History

Introduced in R2023a

See Also

Functions

`getGreedyPolicy` | `generatePolicyFunction` | `generatePolicyBlock` | `getAction`

Objects

`rlMaxQPolicy` | `rlEpsilonGreedyPolicy` | `rlAdditiveNoisePolicy` | `rlDeterministicActorPolicy` | `rlStochasticActorPolicy`

Blocks

Policy | RL Agent

Topics

"Generate Policy Block for Deployment"
"Create Policy Evaluation Function from Policy Object" on page 2-91
"Create Policies and Value Functions"

getGreedyPolicy

Package: `rl.agent`

Extract greedy (deterministic) policy object from agent

Syntax

```
policy = getGreedyPolicy(agent)
```

Description

`policy = getGreedyPolicy(agent)` returns a deterministic policy object from the specified reinforcement learning agent.

Examples

Extract Policy Object from Agent

For this example, load the PG agent trained in “Train PG Agent to Balance Cart-Pole System”.

```
load("MATLABCartpolePG.mat","agent")
```

Extract the agent greedy policy using `getGreedyPolicy`.

```
policyDtr = getGreedyPolicy(agent)
```

```
policyDtr = 
  rlStochasticActorPolicy with properties:

                     Actor: [1x1 rl.function.rlDiscreteCategoricalActor]
    UseMaxLikelihoodAction: 1
           ObservationInfo: [1x1 rl.util.rlNumericSpec]
                ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
                SampleTime: 1
```

Note that, in the extracted policy object, the `UseMaxLikelihoodAction` property is set to `true`. This means that the policy object always generates the maximum likelihood action in response to a given observation, and is therefore greedy (and deterministic).

Alternatively, you can extract a stochastic policy using `getExplorationPolicy`.

```
policyXpl = getExplorationPolicy(agent)
```

```
policyXpl = 
  rlStochasticActorPolicy with properties:

                     Actor: [1x1 rl.function.rlDiscreteCategoricalActor]
    UseMaxLikelihoodAction: 0
           ObservationInfo: [1x1 rl.util.rlNumericSpec]
                ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
```

```
                  SampleTime: 1
```

This time, the extracted policy object has the `UseMaxLikelihoodAction` property is set to `false`. This means that the policy object generates a random action, given an observation. The policy is therefore stochastic and useful for exploration.

Input Arguments

`agent` — Reinforcement learning agent
reinforcement learning agent object

Reinforcement learning agent that contains a critic, specified as one of the following objects:

- `rlQAgent`
- `rlSARSAAgent`
- `rlDQNAgent`
- `rlPGAgent` (when using a critic to estimate a baseline value function)
- `rlDDPGAgent`
- `rlTD3Agent`
- `rlACAgent`
- `rlSACAgent`
- `rlPPOAgent`
- `rlTRPOAgent`
- `rlMBPOAgent`

Note if `agent` is an `rlMBPOAgent` object, to extract the greedy policy, use `getGreedyPolicy(agent.BaseAgent)`.

Output Arguments

`policy` — Reinforcement learning policy object
`rlMaxQPolicy` object | `rlDeterministicActorPolicy` object | `rlStochasticActorPolicy` object

Policy object, returned as one of the following:

- `rlMaxQPolicy` object — Returned when `agent` is an `rlQAgent`, `rlSARSAAgent`, or `rlDQNAgent` object.
- `rlDeterministicActorPolicy` object — Returned when `agent` is an `rlDDPGAgent` or `rlTD3Agent` object.
- `rlStochasticActorPolicy` object, with the `UseMaxLikelihoodAction` set to `true` — Returned when `agent` is an `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, `rlTRPOAgent` or `rlSACAgent` object. Since the returned policy object has the `UseMaxLikelihoodAction` property set to `true`, it always generates the deterministic maximum likelihood action as a response to given observation.

Version History

Introduced in R2022a

See Also

Functions
`getExplorationPolicy` | `generatePolicyFunction` | `generatePolicyBlock` | `getAction`

Objects
`rlMaxQPolicy` | `rlEpsilonGreedyPolicy` | `rlAdditiveNoisePolicy` | `rlDeterministicActorPolicy` | `rlStochasticActorPolicy`

Blocks
Policy | RL Agent

Topics
"Generate Policy Block for Deployment"
"Create Policy Evaluation Function from Policy Object" on page 2-91
"Create Policies and Value Functions"

getLearnableParameters

Package: `rl.policy`

Obtain learnable parameter values from agent, function approximator, or policy object

Syntax

```
pars = getLearnableParameters(agent)
```

```
pars = getLearnableParameters(fcnAppx)
```

```
pars = getLearnableParameters(policy)
```

Description

Agent

`pars = getLearnableParameters(agent)` returns the learnable parameter values from the agent object `agent`.

Actor or Critic

`pars = getLearnableParameters(fcnAppx)` returns the learnable parameter values from the actor or critic function approximator object `fcnAppx`.

Policy

`pars = getLearnableParameters(policy)` returns the learnable parameters values from the policy object `policy`.

Examples

Modify Critic Parameter Values

Assume that you have an existing trained reinforcement learning agent. For this example, load the trained agent from “Train DDPG Agent to Control Double Integrator System”.

```
load("DoubleIntegDDPG.mat","agent")
```

Obtain the critic function approximator from the agent.

```
critic = getCritic(agent);
```

Obtain the learnable parameters from the critic.

```
params = getLearnableParameters(critic)
```

```
params=2×1 cell array
    {[-4.9869 -1.5577 -0.3351 -0.1097 -0.0444 7.5240e-04]}
    {[                                                 0]}
```

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,"UniformOutput",false);
```

Set the parameter values of the critic to the new modified values.

```
critic = setLearnableParameters(critic,modifiedParams);
```

Set the critic in the agent to the new modified critic.

```
setCritic(agent,critic);
```

Display the new parameter values.

```
getLearnableParameters(getCritic(agent))
```

```
ans=2×1 cell array
    {[-9.9737 -3.1153 -0.6702 -0.2194 -0.0888 0.0015]}
    {[                                             0]}
```

Modify Actor Parameter Values

Assume that you have an existing trained reinforcement learning agent. For this example, load the trained agent from “Train DDPG Agent to Control Double Integrator System”.

```
load("DoubleIntegDDPG.mat","agent")
```

Obtain the actor function approximator from the agent.

```
actor = getActor(agent);
```

Obtain the learnable parameters from the actor.

```
params = getLearnableParameters(actor)
```

```
params=2×1 cell array
    {[-15.4601 -7.2076]}
    {[               0]}
```

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,"UniformOutput",false);
```

Set the parameter values of the actor to the new modified values.

```
actor = setLearnableParameters(actor,modifiedParams);
```

Set the actor in the agent to the new modified actor.

```
setActor(agent,actor);
```

Display the new parameter values.

```
getLearnableParameters(getActor(agent))
```

```
ans=2×1 cell array
    {[-30.9201 -14.4153]}
    {[               0]}
```

Input Arguments

`agent` — Reinforcement learning agent
reinforcement learning agent object

Reinforcement learning agent, specified as one of the following objects:

- `rlQAgent`
- `rlSARSAAgent`
- `rlDQNAgent`
- `rlPGAgent`
- `rlDDPGAgent`
- `rlTD3Agent`
- `rlACAgent`
- `rlSACAgent`
- `rlPPOAgent`
- `rlTRPOAgent`
- Custom agent — For more information, see "Create Custom Reinforcement Learning Agents".

`fcnAppx` — Actor or critic function object
reinforcement learning function approximator object

Function approximator object, specified as one of the following:

- `rlValueFunction` object — Value function critic
- `rlQValueFunction` object — Q-value function critic
- `rlVectorQValueFunction` object — Multi-output Q-value function critic with a discrete action space
- `rlContinuousDeterministicActor` object — Deterministic policy actor with a continuous action space
- `rlDiscreteCategoricalActor` object — Stochastic policy actor with a discrete action space
- `rlContinuousGaussianActor` object — Stochastic policy actor with a continuous action space
- `rlContinuousDeterministicTransitionFunction` object — Continuous deterministic transition function for neural network environments
- `rlContinuousGaussianTransitionFunction` object — Continuous Gaussian transition function for neural network environments
- `rlContinuousDeterministicRewardFunction` object — Continuous deterministic reward function for neural network environments
- `rlContinuousGaussianRewardFunction` object — Continuous Gaussian reward function for neural network environments

- rlIsDoneFunction object — IsDone function for neural network environments

To create an actor or critic function object, use one of the following methods.

- Create a function object directly.
- Obtain the existing critic from an agent using getCritic.
- Obtain the existing actor from an agent using getActor.

policy — Reinforcement learning policy
rlMaxQPolicy | rlEpsilonGreedyPolicy | rlDeterministicActorPolicy | rlAdditiveNoisePolicy | rlStochasticActorPolicy

Reinforcement learning policy, specified as one of the following objects:

- rlMaxQPolicy
- rlEpsilonGreedyPolicy
- rlDeterministicActorPolicy
- rlAdditiveNoisePolicy
- rlStochasticActorPolicy

Output Arguments

pars — Learnable parameters
rlValueFunction object | rlQValueFunction object | rlVectorQValueFunction object | rlContinuousDeterministicActor object | rlDiscreteCategoricalActor object | rlContinuousGaussianActor object

Learnable parameter values for the function object, returned as a cell array. You can modify these parameter values and set them in the original agent or a different agent using the setLearnableParameters function.

Version History

Introduced in R2019a

R2022a: getLearnableParameters now uses approximator objects instead of representation objects

Behavior changed in R2022a

Using representation objects to create actors and critics for reinforcement learning agents is no longer recommended. Therefore, getLearnableParameters now uses function approximator objects instead.

R2020a: getLearnableParameterValues is now getLearnableParameters

Behavior changed in R2020a

getLearnableParameterValues is now getLearnableParameters. To update your code, change the function name from getLearnableParameterValues to getLearnableParameters. The syntaxes are equivalent.

See Also

Functions
`setLearnableParameters` | `getActor` | `getCritic` | `setActor` | `setCritic` | `evaluate`

Objects
`rlValueFunction` | `rlContinuousDeterministicActor` | `rlContinuousDeterministicTransitionFunction` | `rlQAgent` | `rlPGAgent` | `rlACAgent` | `rlMaxQPolicy`

Topics
"Create Policies and Value Functions"
"Reinforcement Learning Agents"
"Import Neural Network Models"

getMaxQValue

Package: rl.function

Obtain maximum estimated value over all possible actions from a Q-value function critic with discrete action space, given environment observations

Syntax

```
[maxQ,maxActionIndex] = getMaxQValue(qValueFcnObj,obs)
[maxQ,maxActionIndex,state] = getMaxQValue( ___ )
```

Description

[maxQ,maxActionIndex] = getMaxQValue(qValueFcnObj,obs) evaluates the discrete-action-space Q-value function critic qValueFcnObj and returns the maximum estimated value over all possible actions maxQ, with the corresponding action index maxActionIndex, given environment observations obs.

[maxQ,maxActionIndex,state] = getMaxQValue(___) also returns the updated state of qValueFcnObj when it contains a recurrent neural network.

Examples

Obtain Maximum Q-Value Function Estimates

Create an observation and action specification objects (or alternatively use getObservationInfo and getActionInfo to extract the specification objects from an environment. For this example, define the observation space as a continuous three-dimensional space, and the action space as a finite set consisting of three possible values (named -1, 0, and 1).

```
obsInfo = rlNumericSpec([3 1]);
actInfo = rlFiniteSetSpec([-1 0 1]);
```

Create a custom basis function to approximate the Q-value function within the critic, and define an initial parameter vector.

```
myBasisFcn = @(myobs,myact) [ ...
    ones(4,1);
    myobs(:); myact;
    myobs(:).^2; myact.^2;
    sin(myobs(:)); sin(myact);
    cos(myobs(:)); cos(myact) ];
W0 = rand(20,1);
```

Create the critic.

```
critic = rlQValueFunction({myBasisFcn,W0}, ...
    obsInfo,actInfo);
```

Use getMaxQValue to return the maximum value, among the possible actions, given a random observation. Also return the index corresponding to the action that maximizes the value.

```
[v,i] = getMaxQValue(critic,{rand(3,1)})
```

```
v = 9.0719
```

```
i = 3
```

Create a batch set of 64 random independent observations. The third dimension is the batch size, while the fourth is the sequence length for any recurrent neural network used by the critic (in this case not used).

```
batchobs = rand(3,1,64,1);
```

Obtain maximum values for all the observations.

```
bv = getMaxQValue(critic,{batchobs});
size(bv)
```

```
ans = 1×2

     1    64
```

Select the maximum value corresponding to the 44th observation.

```
bv(44)
```

```
ans = 10.4138
```

Input Arguments

qValueFcnObj — Q-value function critic
rlQValueFunction object | rlVectorQValueFunction object

Q-value function critic, specified as an `rlQValueFunction` or `rlVectorQValueFunction` object.

obs — Environment observations
cell array

Environment observations, specified as a cell array with as many elements as there are observation input channels. Each element of `obs` contains an array of observations for a single observation input channel.

The dimensions of each element in `obs` are M_O-by-L_B-by-L_S, where:

- M_O corresponds to the dimensions of the associated observation input channel.
- L_B is the batch size. To specify a single observation, set $L_B = 1$. To specify a batch of observations, specify $L_B > 1$. If `qValueFcnObj` has multiple observation input channels, then L_B must be the same for all elements of `obs`.
- L_S specifies the sequence length for a recurrent neural network. If `qValueFcnObj` does not use a recurrent neural network, then $L_S = 1$. If `qValueFcnObj` has multiple observation input channels, then L_S must be the same for all elements of `obs`.

L_B and L_S must be the same for both `act` and `obs`.

For more information on input and output formats for recurrent neural networks, see the Algorithms section of `lstmLayer`.

Output Arguments

`maxQ` — Maximum Q-value estimate

array

Maximum Q-value estimate across all possible discrete actions, returned as a 1-by-L_B-by-L_S array, where:

- L_B is the batch size.
- L_S specifies the sequence length for a recurrent neural network. If `qValueFcnObj` does not use a recurrent neural network, then $L_S = 1$.

`maxActionIndex` — Action index

array

Action index corresponding to the maximum Q value, returned as a 1-by-L_B-by-L_S array, where:

- L_B is the batch size.
- L_S specifies the sequence length for a recurrent neural network. If `qValueFcnObj` does not use a recurrent neural network, then $L_S = 1$.

`state` — Updated state of the critic

cell array

Updated state of `qValueFcnObj`, returned as a cell array. If `qValueFcnObj` does not use a recurrent neural network, then `state` is an empty cell array.

You can set the state of the critic to `state` using the `setState` function. For example:

```
qValueFcnObj = setState(qValueFcnObj,state);
```

Version History

Introduced in R2020a

See Also

Functions

`getValue` | `evaluate` | `getAction` | `getActor` | `getCritic`

Topics

“Train Reinforcement Learning Policy Using Custom Training Loop”
“Create Custom Reinforcement Learning Agents”

getModel

Package: `rl.function`

Get function approximator model from actor or critic

Syntax

```
model = getModel(fcnAppx)
```

Description

`model = getModel(fcnAppx)` returns the function approximation model used by the actor or critic function object `fcnAppx`.

Examples

Modify Deep Neural Networks in Reinforcement Learning Agent

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”.

Load the predefined environment.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a PPO agent from the environment observation and action specifications. This agent uses default deep neural networks for its actor and critic.

```
agent = rlPPOAgent(obsInfo,actInfo);
```

To modify the deep neural networks within a reinforcement learning agent, you must first extract the actor and critic function approximators.

```
actor = getActor(agent);
critic = getCritic(agent);
```

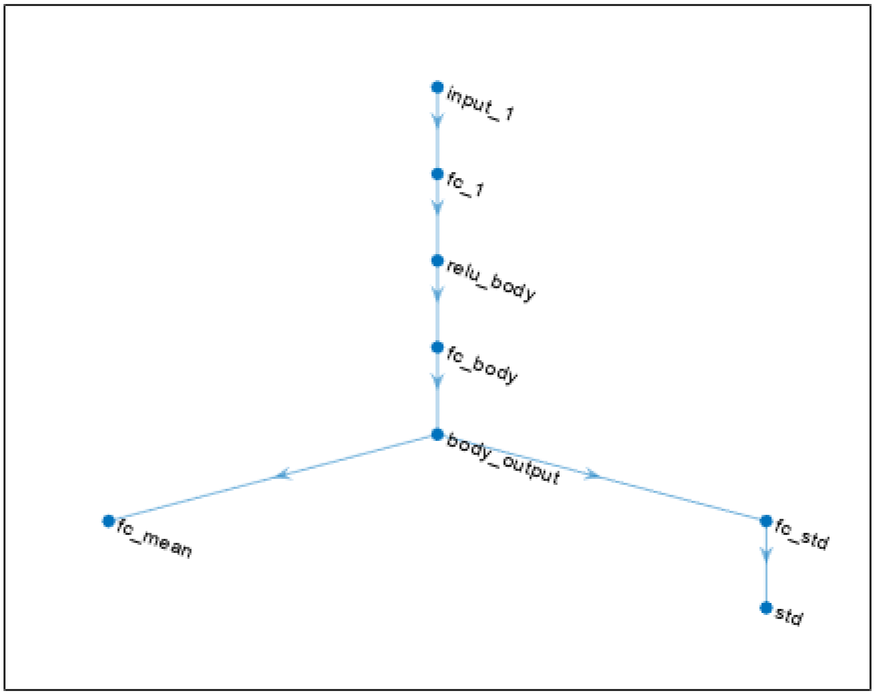
Extract the deep neural networks from both the actor and critic function approximators.

```
actorNet = getModel(actor);
criticNet = getModel(critic);
```

The networks are `dlnetwork` objects. To view them using the `plot` function, you must convert them to `layerGraph` objects.

For example, view the actor network.

```
plot(layerGraph(actorNet))
```



To validate a network, use `analyzeNetwork`. For example, validate the critic network.

```
analyzeNetwork(criticNet)
```

You can modify the actor and critic networks and save them back to the agent. To modify the networks, you can use the Deep Network Designer app. To open the app for each network, use the following commands.

```
deepNetworkDesigner(layerGraph(criticNet))
deepNetworkDesigner(layerGraph(actorNet))
```

In **Deep Network Designer**, modify the networks. For example, you can add additional layers to your network. When you modify the networks, do not change the input and output layers of the networks returned by `getModel`. For more information on building networks, see “Build Networks with Deep Network Designer”.

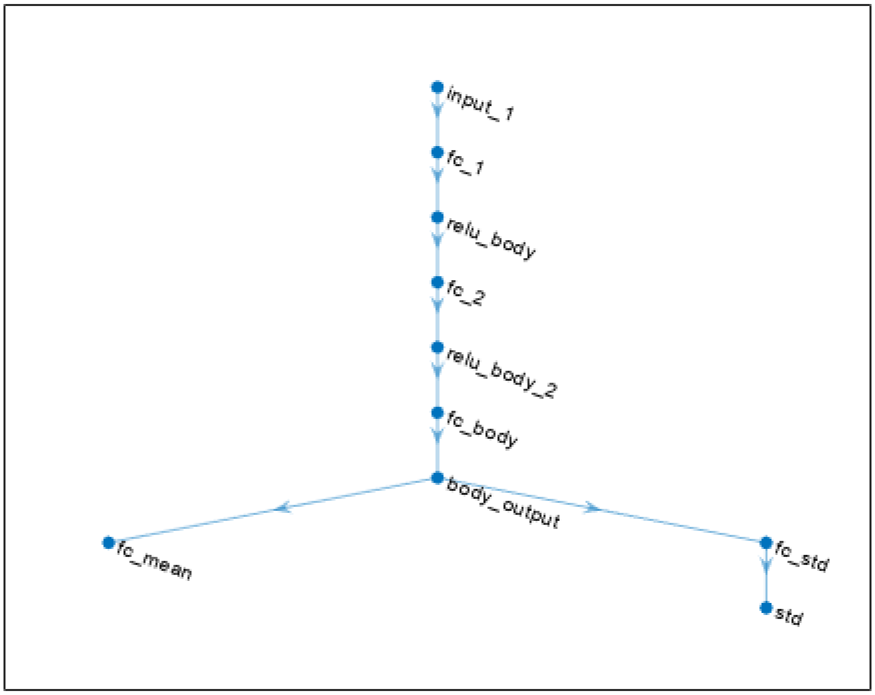
To validate the modified network in **Deep Network Designer**, you must click on **Analyze for dlnetwork**, under the **Analysis** section. To export the modified network structures to the MATLAB® workspace, generate code for creating the new networks and run this code from the command line. Do not use the exporting option in **Deep Network Designer**. For an example that shows how to generate and run code, see “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”.

For this example, the code for creating the modified actor and critic networks is in the `createModifiedNetworks` helper script.

```
createModifiedNetworks
```

Each of the modified networks includes an additional `fullyConnectedLayer` and `reluLayer` in their main common path. View the modified actor network.

```
plot(layerGraph(modifiedActorNet))
```



After exporting the networks, insert the networks into the actor and critic function approximators.

```
actor = setModel(actor,modifiedActorNet);
critic = setModel(critic,modifiedCriticNet);
```

Finally, insert the modified actor and critic function approximators into the actor and critic objects.

```
agent = setActor(agent,actor);
agent = setCritic(agent,critic);
```

Input Arguments

`fcnAppx` — Actor or critic function object
`rlValueFunction` object | `rlQValueFunction` object | `rlVectorQValueFunction` object | `rlContinuousDeterministicActor` object | `rlDiscreteCategoricalActor` object | `rlContinuousGaussianActor` object

Actor or critic function object, specified as one of the following:

- `rlValueFunction` object — Value function critic
- `rlQValueFunction` object — Q-value function critic
- `rlVectorQValueFunction` object — Multi-output Q-value function critic with a discrete action space
- `rlContinuousDeterministicActor` object — Deterministic policy actor with a continuous action space
- `rlDiscreteCategoricalActor` — Stochastic policy actor with a discrete action space
- `rlContinuousGaussianActor` object — Stochastic policy actor with a continuous action space

To create an actor or critic function object, use one of the following methods.

- Create a function object directly.
- Obtain the existing critic from an agent using `getCritic`.
- Obtain the existing actor from an agent using `getActor`.

Note For agents with more than one critic, such as TD3 and SAC agents, you must call `getModel` for each critic representation individually, rather than calling `getModel` for the array returned by `getCritic`.

```
critics = getCritic(myTD3Agent);
criticNet1 = getModel(critics(1));
criticNet2 = getModel(critics(2));
```

Output Arguments

`model` — Function approximation model
`dlnetwork` object | `rlTable` object | 1-by-2 cell array

Function approximation model, returned as one of the following:

- Deep neural network defined as a `dlnetwork` object
- `rlTable` object
- 1-by-2 cell array that contains the function handle for a custom basis function and the basis function parameters

Version History

Introduced in R2020b

R2022a: `getModel` now uses approximator objects instead of representation objects

Behavior changed in R2022a

Using representation objects to create actors and critics for reinforcement learning agents is no longer recommended. Therefore, `getModel` now uses function approximator objects instead.

R2021b: `getModel` returns a `dlnetwork` object

Behavior changed in R2021b

Starting from R2021b, built-in agents use `dlnetwork` objects as actor and critic representations, so `getModel` returns a `dlnetwork` object.

- Due to numerical differences in the network calculations, previously trained agents might behave differently. If this happens, you can retrain your agents.
- To use Deep Learning Toolbox™ functions that do not support `dlnetwork`, you must convert the network to `layerGraph`. For example, to use `deepNetworkDesigner`, replace `deepNetworkDesigner(network)` with `deepNetworkDesigner(layerGraph(network))`.

See Also

Functions
`getActor` | `setActor` | `getCritic` | `setCritic` | `setModel`

Objects
`dlnetwork`

Topics
"Create Policies and Value Functions"

getObservationInfo

Package: `rl.env`

Obtain observation data specifications from reinforcement learning environment, agent, or experience buffer

Syntax

```
obsInfo = getObservationInfo(env)
obsInfo = getObservationInfo(agent)
obsInfo = getObservationInfo(buffer)
```

Description

`obsInfo = getObservationInfo(env)` extracts observation information from reinforcement learning environment `env`.

`obsInfo = getObservationInfo(agent)` extracts observation information from reinforcement learning agent `agent`.

`obsInfo = getObservationInfo(buffer)` extracts observation information from experience buffer `buffer`.

Examples

Extract Action and Observation Specifications from Reinforcement Learning Environment

The reinforcement learning environment for this example is a longitudinal dynamics model comprising two cars, a leader and a follower. The vehicle model is also used in the “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox) example.

Open the model.

```
mdl = "rlACCMdl";
open_system(mdl);
```

Specify path to the agent block in the model.

```
agentblk = mdl + "/RL Agent";
```

Create the observation and action specifications.

```
% Observation specifications
obsInfo = rlNumericSpec([3 1],LowerLimit=-inf*ones(3,1),UpperLimit=inf*ones(3,1));
obsInfo.Name = "observations";
obsInfo.Description = "information on velocity error and ego velocity";

% Action specifications
actInfo = rlNumericSpec([1 1],LowerLimit=-3,UpperLimit=2);
actInfo.Name = "acceleration";
```

Define environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlACCMdl
      AgentBlock : rlACCMdl/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

The reinforcement learning environment `env` is a `SimulinkEnvWithAgent` object.

Extract the action and observation specifications from `env`.

```
actInfoExt = getActionInfo(env)
```

```
actInfoExt = 
  rlNumericSpec with properties:

     LowerLimit: -3
     UpperLimit: 2
           Name: "acceleration"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

```
obsInfoExt = getObservationInfo(env)
```

```
obsInfoExt = 
  rlNumericSpec with properties:

     LowerLimit: [3x1 double]
     UpperLimit: [3x1 double]
           Name: "observations"
    Description: "information on velocity error and ego velocity"
      Dimension: [3 1]
       DataType: "double"
```

The action information contains acceleration values while the observation information contains the velocity and velocity error values of the ego vehicle.

Input Arguments

env — Reinforcement learning environment
`rlFunctionEnv` object | `SimulinkEnvWithAgent` object | `rlNeuralNetworkEnvironment` object | predefined MATLAB environment object

Reinforcement learning environment from which to extract the observation information, specified as one of the following objects.

- MATLAB environment represented as one of the following objects.

  - rlFunctionEnv
  - rlNeuralNetworkEnvironment
  - Predefined MATLAB environment created using rlPredefinedEnv
- Simulink environment represented as a SimulinkEnvWithAgent object.

For more information on reinforcement learning environments, see “Create MATLAB Reinforcement Learning Environments” and “Create Simulink Reinforcement Learning Environments”.

agent — Reinforcement learning agent
rlQAgent object | rlSARSAAgent object | rlDQNAgent object | rlPGAgent object | rlDDPGAgent object | rlTD3Agent object | rlACAgent object | rlPPOAgent object | rlTRPOAgent object | rlSACAgent object | rlMBPOAgent object

Reinforcement learning agent from which to extract the observation information, specified as one of the following objects.

- rlQAgent
- rlSARSAAgent
- rlDQNAgent
- rlDDPGAgent
- rlTD3Agent
- rlPGAgent
- rlACAgent
- rlPPOAgent
- rlTRPOAgent
- rlSACAgent
- rlMBPOAgent

For more information on reinforcement learning agents, see “Reinforcement Learning Agents”.

buffer — Experience buffer
rlReplayMemory object | rlPrioritizedReplayMemory object | rlHindsightReplayMemory object | rlHindsightPrioritizedReplayMemory object

Experience buffer, specified as one of the following replay memory objects.

- rlReplayMemory
- rlPrioritizedReplayMemory
- rlHindsightReplayMemory
- rlHindsightPrioritizedReplayMemory

Output Arguments

obsInfo — Observation data specifications
array of rlNumericSpec objects | array of rlFiniteSetSpec objects

Observation data specifications extracted from the reinforcement learning environment, returned as an array of one of the following:

- `rlNumericSpec` objects
- `rlFiniteSetSpec` objects
- A mix of `rlNumericSpec` and `rlFiniteSetSpec` objects

Version History

Introduced in R2019a

See Also

Functions
`getActionInfo`

Objects
`rlNumericSpec` | `rlFiniteSetSpec` | `rlQAgent` | `rlSARSAAgent` | `rlDQNAgent` | `rlPGAgent` | `rlACAgent` | `rlDDPGAgent`

Topics
"Create Simulink Reinforcement Learning Environments"
"Reinforcement Learning Agents"

getValue

Package: `rl.function`

Obtain estimated value from a critic given environment observations and actions

Syntax

```
value = getValue(valueFcnAppx,obs)
```

```
value = getValue(vqValueFcnAppx,obs)
value = getValue(qValueFcnAppx,obs,act)
```

```
[value,state] = getValue( ___ )
```

Description

Value Function Critic

`value = getValue(valueFcnAppx,obs)` evaluates the value function critic `valueFcnAppx` and returns the value corresponding to the observation `obs`. In this case, `valueFcnAppx` is an `rlValueFunction` approximator object.

Q-Value Function Critics

`value = getValue(vqValueFcnAppx,obs)` evaluates the discrete-action-space Q-value function critic `vqValueFcnAppx` and returns the vector `value`, in which each element represents the estimated value given the state corresponding to the observation `obs` and the action corresponding to the element number of `value`. In this case, `vqValueFcnAppx` is an `rlVectorQValueFunction` approximator object.

`value = getValue(qValueFcnAppx,obs,act)` evaluates the Q-value function critic `qValueFcnAppx` and returns the scalar `value`, representing the value given the observation `obs` and action `act`. In this case, `qValueFcnAppx` is an `rlQValueFunction` approximator object.

Return Recurrent Neural Network State

`[value,state] = getValue( ___ )` also returns the updated state of the critic object when it contains a recurrent neural network.

Examples

Obtain Value Function Estimates

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

To approximate the value function within the critic, create a neural network. Define a single path from the network input (the observation) to its output (the value), as an array of layer objects.

```
net = [ featureInputLayer(4) ...
        fullyConnectedLayer(1)];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
net = dlnetwork(net);
summary(net);
```

```
   Initialized: true

   Number of learnables: 5

   Inputs:
      1   'input'   4 features
```

Create a critic using the network and the observation specification object. When you use this syntax the network input layer is automatically associated with the environment observation according to the dimension specifications in `obsInfo`.

```
critic = rlValueFunction(net,obsInfo);
```

Obtain a value function estimate for a random single observation. Use an observation array with the same dimensions as the observation specification.

```
val = getValue(critic,{rand(4,1)})
```

```
val = single
    0.7904
```

You can also obtain value function estimates for a batch of observations. For example obtain value functions for a batch of 20 observations.

```
batchVal = getValue(critic,{rand(4,1,20)});
size(batchVal)
```

```
ans = 1×2

     1    20
```

`valBatch` contains one value function estimate for each observation in the batch.

Obtain Vector Q-Value Function Estimates

Create observation and action specification objects (or alternatively use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing four doubles, and the action space as a finite set consisting of three possible values (named `7`, `5`, and `3` in this case).

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlFiniteSetSpec([7 5 3]);
```

Create a vector Q-value function approximator to use as a critic. A vector Q-value function takes only the observation as input and returns as output a single vector with as many elements as the number of possible actions. The value of each output element represents the expected discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy afterwards.

To model the parametrized vector Q-value function within the critic. The network must have one input layer that accepts a four-element vector, as defined by `obsInfo`. The output must be a single output layer having as many elements as the number of possible discrete actions (three in this case, as defined by `actInfo`).

Define a single path from the network input to its output as array of layer objects.

```
net = [
    featureInputLayer(4)
    fullyConnectedLayer(3)
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 15

   Inputs:
      1   'input'   4 features
```

Create the critic using the network, as well as the names of the observation and action specification objects. The network input layers are automatically associated with the observation channels, according to the dimension specifications in `obsInfo`.

```
critic = rlVectorQValueFunction(net,obsInfo,actInfo);
```

Use `getValue` to return the values of a random observation, using the current network weights.

```
v = getValue(critic,{rand(obsInfo.Dimension)})
```

```
v = 3x1 single column vector

    0.7232
    0.8177
   -0.2212
```

`v` contains three value function estimates, one for each possible discrete action.

You can also obtain value function estimates for a batch of observations. For example, obtain value function estimates for a batch of 10 observations.

```
batchV = getValue(critic,{rand([obsInfo.Dimension 10])});
size(batchV)
```

```
ans = 1×2
```

```
     3    10
```

batchV contains three value function estimates for each observation in the batch.

Obtain Single-Output Q-Value Function Estimates

Create observation and action specification objects (or alternatively use getObservationInfo and getObservationInfo to extract the specification object from an environment). For this example, define the observation space as having two continuous channels, the first one carrying an 8 by 3 matrix, and the second one a continuous four-dimensional vector. The action specification is a continuous column vector containing 2 doubles.

```
obsInfo = [rlNumericSpec([8 3]), rlNumericSpec([4 1])];
actInfo = rlNumericSpec([2 1]);
```

Create a custom basis function and its initial weight matrix.

```
myBasisFcn = @(obsA,obsB,act) [...
    ones(30,1);
    obsA(:); obsB(:); act(:);
    obsA(:).^2; obsB(:).^2; act(:).^2;
    sin(obsA(:)); sin(obsB(:)); sin(act(:));
    cos(obsA(:)); cos(obsB(:)); cos(act(:))];
W0 = rand(150,1);
```

The output of the critic is the scalar W'*myBasisFcn(obs,act), representing the Q-value function to be approximated.

Create the critic.

```
critic = rlQValueFunction({myBasisFcn,W0}, ...
    obsInfo,actInfo);
```

Use getValue to return the value of a random observation-action pair, using the current parameter matrix.

```
v = getValue(critic,{rand(8,3),(1:4)'},{rand(2,1)})
```

```
v = 68.8628
```

Create a random observation set of batch size 64 for each channel. The third dimension is the batch size, while the fourth is the sequence length for any recurrent neural network used by the critic (in this case not used).

```
batchobs_ch1 = rand(8,3,64,1);
batchobs_ch2 = rand(4,1,64,1);
```

Create a random action set of batch size 64.

```
batchact = rand(2,1,64,1);
```

Obtain the state-action value function estimate for the batch of observations and actions.

```
bv = getValue(critic,{batchobs_ch1,batchobs_ch2},{batchact});
size(bv)
```

```
ans = 1×2

     1    64
```

```
bv(23)
```

```
ans = 46.6310
```

Input Arguments

valueFcnAppx — Value function critic
`rlValueFunction` object

Value function critic, specified as an `rlValueFunction` approximator object.

vqValueFcnAppx — Vector Q-value function critic
`rlVectorQValueFunction` object

Vector Q-value function critic, specified as an `rlVectorQValueFunction` approximator object.

qValueFcnAppx — Q-value function critic
`rlQValueFunction` object

Q-value function critic, specified as an `rlQValueFunction` object.

obs — Observations
cell array

Observations, specified as a cell array with as many elements as there are observation input channels. Each element of `obs` contains an array of observations for a single observation input channel.

The dimensions of each element in `obs` are M_O-by-L_B-by-L_S, where:

- M_O corresponds to the dimensions of the associated observation input channel.
- L_B is the batch size. To specify a single observation, set $L_B = 1$. To specify a batch of observations, specify $L_B > 1$. If the critic object given as first input argument has multiple observation input channels, then L_B must be the same for all elements of `obs`.
- L_S specifies the sequence length for a recurrent neural network. If the critic object given as first input argument does not use a recurrent neural network, then $L_S = 1$. If the critic object has multiple observation input channels, then L_S must be the same for all elements of `obs`.

L_B and L_S must be the same for both `act` and `obs`.

For more information on input and output formats for recurrent neural networks, see the Algorithms section of `lstmLayer`.

act — Action
single-element cell array

Action, specified as a single-element cell array that contains an array of action values.

The dimensions of this array are M_A-by-L_B-by-L_S, where:

- M_A corresponds to the dimensions of the associated action specification.
- L_B is the batch size. To specify a single observation, set $L_B = 1$. To specify a batch of observations, specify $L_B > 1$.
- L_S specifies the sequence length for a recurrent neural network. If the critic object given as a first input argument does not use a recurrent neural network, then $L_S = 1$.

L_B and L_S must be the same for both `act` and `obs`.

For more information on input and output formats for recurrent neural networks, see the Algorithms section of `lstmLayer`.

Output Arguments

`value` — Estimated value function

array

Estimated value function, returned as array with dimensions N-by-L_B-by-L_S, where:

- N is the number of outputs of the critic network.
  - For a state value critics (`valueFcnAppx`), $N = 1$.
  - For a single-output state-action value function critics (`qValueFcnAppx`), $N = 1$.
  - For a multi-output state-action value function critics (`vqValueFcnAppx`), N is the number of discrete actions.
- L_B is the batch size.
- L_S is the sequence length for a recurrent neural network.

`state` — Updated state of critic

cell array

Updated state of the critic, returned as a cell array. If the critic does not use a recurrent neural network, then `state` is an empty cell array.

You can set the state of the critic to `state` using the `setState` function. For example:

```
valueFcnAppx = setState(valueFcnAppx,state);
```

Tips

The more general function `evaluate` behaves, for critic objects, similarly to `getValue` except that `evaluate` returns results inside a single-cell array.

para

Version History

Introduced in R2020a

See Also

Functions
`getMaxQValue` | `evaluate` | `getAction` | `getActor` | `getCritic`

Topics
"Train Reinforcement Learning Policy Using Custom Training Loop"
"Create Custom Reinforcement Learning Agents"

gradient

Package: rl.function

Evaluate gradient of function approximator object given observation and action input data

Syntax

```
grad = gradient(fcnAppx,'output-input',inData)
grad = gradient(fcnAppx,'output-parameters',inData)
grad = gradient(fcnAppx,lossFcn,inData,fcnData)
```

Description

grad = gradient(fcnAppx,'output-input',inData) evaluates the gradient of the sum of the outputs of the function approximator object fcnAppx with respect to its inputs. It returns the value of the gradient grad when the input of fcnAppx is inData.

grad = gradient(fcnAppx,'output-parameters',inData) evaluates the gradient of the sum of the outputs of fcnAppx with respect to its parameters.

grad = gradient(fcnAppx,lossFcn,inData,fcnData) evaluates the gradient of a loss function associated to the function handle lossFcn, with respect to the parameters of fcnAppx. The last optional argument fcnData can contain additional inputs for the loss function.

Examples

Calculate Gradients for Continuous Gaussian Actor

Create observation and action specification objects (or alternatively use getObservationInfo and getActionInfo to extract the specification objects from an environment). For this example, define an observation space with of three channels. The first channel carries an observation from a continuous three-dimensional space, so that a single observation is a column vector containing three doubles. The second channel carries a discrete observation made of a two-dimensional row vector that can take one of five different values. The third channel carries a discrete scalar observation that can be either zero or one. Finally, the action space is a continuous four-dimensional space, so that a single action is a column vector containing four doubles, each between -10 and 10.

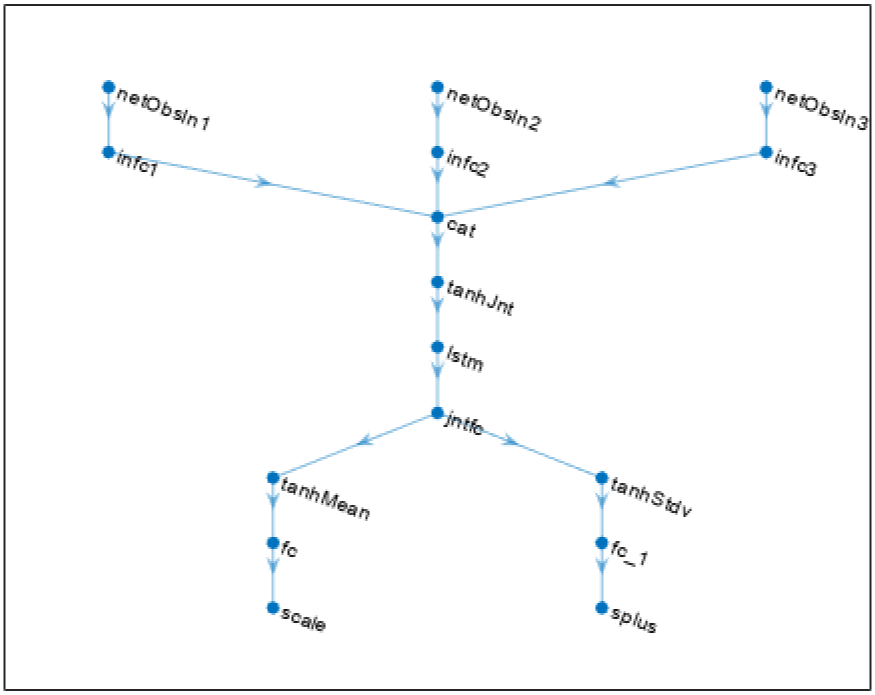
```
obsInfo = [rlNumericSpec([3 1])
           rlFiniteSetSpec({[1 2],[3 4],[5 6],[7 8],[9 10]})
           rlFiniteSetSpec([0 1])];

actInfo = rlNumericSpec([4 1], ...
                UpperLimit= 10*ones(4,1), ...
                LowerLimit=-10*ones(4,1) );
```

To approximate the policy within the actor, use a recurrent deep neural network. For a continuous Gaussian actor, the network must have two output layers (one for the mean values the other for the standard deviation values), each having as many elements as the dimension of the action space.

Create a the network, defining each path as an array of layer objects. Use `sequenceInputLayer` as the input layer and include an `lstmLayer` as one of the other network layers. Also use a softplus layer to enforce nonnegativity of the standard deviations and a ReLU layer to scale the mean values to the desired output range. Get the dimensions of the observation and action spaces from the environment specification objects, and specify a name for the input layers, so you can later explicitly associate them with the appropriate environment channel.

```
inPath1 = [ sequenceInputLayer( ...
                prod(obsInfo(1).Dimension), ...
                Name="netObsIn1")
            fullyConnectedLayer(prod(actInfo.Dimension), ...
                Name="infc1") ];

inPath2 = [ sequenceInputLayer( ...
                prod(obsInfo(2).Dimension), ...
                Name="netObsIn2")
            fullyConnectedLayer( ...
                prod(actInfo.Dimension), ...
                Name="infc2") ];

inPath3 = [ sequenceInputLayer( ...
                prod(obsInfo(3).Dimension), ...
                Name="netObsIn3")
            fullyConnectedLayer( ...
                prod(actInfo.Dimension), ...
                Name="infc3") ];

% Concatenate inputs along the first available dimension
jointPath = [ concatenationLayer(1,3,Name="cat")
              tanhLayer(Name="tanhJnt");
              lstmLayer(8,OutputMode="sequence",Name="lstm")
              fullyConnectedLayer( ...
                prod(actInfo.Dimension), ...
                Name="jntfc");
             ];

% Path layers for mean value
% Using scalingLayer to scale range from (-1,1) to (-10,10)
meanPath = [ tanhLayer(Name="tanhMean");
             fullyConnectedLayer(prod(actInfo.Dimension));
             scalingLayer(Name="scale", ...
                Scale=actInfo.UpperLimit) ];

% Path layers for standard deviations
% Using softplus layer to make them nonnegative
sdevPath = [ tanhLayer(Name="tanhStdv");
             fullyConnectedLayer(prod(actInfo.Dimension));
             softplusLayer(Name="splus") ];

% Add layers to network object
net = layerGraph;
net = addLayers(net,inPath1);
net = addLayers(net,inPath2);
net = addLayers(net,inPath3);
net = addLayers(net,jointPath);
net = addLayers(net,meanPath);
net = addLayers(net,sdevPath);
```

```
% Connect layers
net = connectLayers(net,"infc1","cat/in1");
net = connectLayers(net,"infc2","cat/in2");
net = connectLayers(net,"infc3","cat/in3");
net = connectLayers(net,"jntfc","tanhMean/in");
net = connectLayers(net,"jntfc","tanhStdv/in");

% Plot network
plot(net)
```



```
% Convert to dlnetwork
net = dlnetwork(net);

% Display the number of weights
summary(net)
```

```
   Initialized: true

   Number of learnables: 784

   Inputs:
      1   'netObsIn1'   Sequence input with 3 dimensions
      2   'netObsIn2'   Sequence input with 2 dimensions
      3   'netObsIn3'   Sequence input with 1 dimensions
```

Create the actor with `rlContinuousGaussianActor`, using the network, the observations and action specification objects, as well as the names of the network input layer and the options object.

```
actor = rlContinuousGaussianActor(net, obsInfo, actInfo, ...
    ActionMeanOutputNames="scale",...
    ActionStandardDeviationOutputNames="splus",...
    ObservationInputNames=["netObsIn1","netObsIn2","netObsIn3"]);
```

To return mean value and standard deviations of the Gaussian distribution as a function of the current observation, use `evaluate`.

```
[prob,state] = evaluate(actor, {rand([obsInfo(1).Dimension 1 1]) , ...
                                rand([obsInfo(2).Dimension 1 1]) , ...
                                rand([obsInfo(3).Dimension 1 1]) });
```

The result is a cell array with two elements, the first one containing a vector of mean values, and the second containing a vector of standard deviations.

```
prob{1}
```

```
ans = 4x1 single column vector

   -1.5454
    0.4908
   -0.1697
    0.8081
```

```
prob{2}
```

```
ans = 4x1 single column vector

    0.6913
    0.6141
    0.7291
    0.6475
```

To return an action sampled from the distribution, use `getAction`.

```
act = getAction(actor, {rand(obsInfo(1).Dimension) , ...
                        rand(obsInfo(2).Dimension) , ...
                        rand(obsInfo(3).Dimension) });
act{1}
```

```
ans = 4x1 single column vector

   -3.2003
   -0.0534
   -1.0700
   -0.4032
```

Calculate the gradients of the sum of the outputs (all the mean values plus all the standard deviations) with respect to the inputs, given a random observation.

```
gro = gradient(actor,"output-input", ...
                        {rand(obsInfo(1).Dimension) , ...
```

```
                         rand(obsInfo(2).Dimension) , ...
                         rand(obsInfo(3).Dimension)} )
```

```
gro=3×1 cell array
    {3x1 single}
    {2x1 single}
    {[  0.1311]}
```

The result is a cell array with as many elements as the number of input channels. Each element contains the derivatives of the sum of the outputs with respect to each component of the input channel. Display the gradient with respect to the element of the second channel.

```
gro{2}
```

```
ans = 2x1 single column vector

   -1.3404
    0.6642
```

Obtain the gradient with respect of five independent sequences, each one made of nine sequential observations.

```
gro_batch = gradient(actor,"output-input", ...
                        {rand([obsInfo(1).Dimension 5 9]) , ...
                         rand([obsInfo(2).Dimension 5 9]) , ...
                         rand([obsInfo(3).Dimension 5 9])} )
```

```
gro_batch=3×1 cell array
    {3x5x9 single}
    {2x5x9 single}
    {1x5x9 single}
```

Display the derivative of the sum of the outputs with respect to the third observation element of the first input channel, after the seventh sequential observation in the fourth independent batch.

```
gro_batch{1}(3,4,7)
```

```
ans = single
    0.2020
```

Set the option to accelerate the gradient computations.

```
actor = accelerate(actor,true);
```

Calculate the gradients of the sum of the outputs with respect to the parameters, given a random observation.

```
grp = gradient(actor,"output-parameters", ...
                {rand(obsInfo(1).Dimension) , ...
                 rand(obsInfo(2).Dimension) , ...
                 rand(obsInfo(3).Dimension)} )
```

```
grp=15×1 cell array
    { 4x3  single}
    { 4x1  single}
    { 4x2  single}
    { 4x1  single}
    { 4x1  single}
    { 4x1  single}
    {32x12 single}
    {32x8  single}
    {32x1  single}
    { 4x8  single}
    { 4x1  single}
    { 4x4  single}
    { 4x1  single}
    { 4x4  single}
    { 4x1  single}
```

Each array within a cell contains the gradient of the sum of the outputs with respect to a group of parameters.

```
grp_batch = gradient(actor,"output-parameters", ...
                        {rand([obsInfo(1).Dimension 5 9]) , ...
                         rand([obsInfo(2).Dimension 5 9]) , ...
                         rand([obsInfo(3).Dimension 5 9])} )
```

```
grp_batch=15×1 cell array
    { 4x3  single}
    { 4x1  single}
    { 4x2  single}
    { 4x1  single}
    { 4x1  single}
    { 4x1  single}
    {32x12 single}
    {32x8  single}
    {32x1  single}
    { 4x8  single}
    { 4x1  single}
    { 4x4  single}
    { 4x1  single}
    { 4x4  single}
    { 4x1  single}
```

If you use a batch of inputs, `gradient` uses the whole input sequence (in this case nine steps), and all the gradients with respect to the independent batch dimensions (in this case five) are added together. Therefore, the returned gradient has always the same size as the output from `getLearnableParameters`.

Calculate Gradients for Vector Q-Value Function

Create observation and action specification objects (or alternatively use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment). For this example, define an observation space made of two channels. The first channel carries an observation from a continuous four-dimensional space. The second carries a discrete scalar observation that can be either zero or one. Finally, the action space consist of a scalar that can be `-1`, `0`, or `1`.

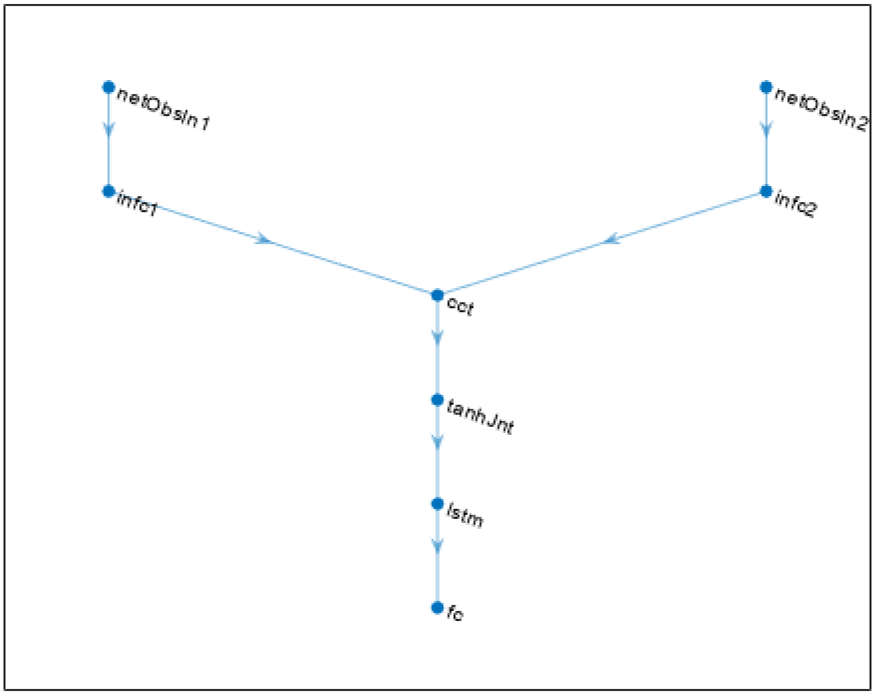
```
obsInfo = [rlNumericSpec([4 1])
           rlFiniteSetSpec([0 1])];
```

```
actInfo =  rlFiniteSetSpec([-1 0 1]);
```

To approximate the vector Q-value function within the critic, use a recurrent deep neural network. The output layer must have three elements, each one expressing the value of executing the corresponding action, given the observation.

Create the neural network, defining each network path as an array of layer objects. Get the dimensions of the observation and action spaces from the environment specification objects, use `sequenceInputLayer` as the input layer, and include an `lstmLayer` as one of the other network layers.

```
inPath1 = [ sequenceInputLayer( ...
                prod(obsInfo(1).Dimension), ...
                Name="netObsIn1")
            fullyConnectedLayer( ...
                prod(actInfo.Dimension), ...
                Name="infc1") ];

inPath2 = [ sequenceInputLayer( ...
                prod(obsInfo(2).Dimension), ...
                Name="netObsIn2")
            fullyConnectedLayer( ...
                prod(actInfo.Dimension), ...
                Name="infc2") ];

% Concatenate inputs along first available dimension
jointPath = [ concatenationLayer(1,2,Name="cct")
              tanhLayer(Name="tanhJnt")
              lstmLayer(8,OutputMode="sequence")
              fullyConnectedLayer(prod(numel(actInfo.Elements))) ];

% Add layers to network object
net = layerGraph;
net = addLayers(net,inPath1);
net = addLayers(net,inPath2);
net = addLayers(net,jointPath);

% Connect layers
net = connectLayers(net,"infc1","cct/in1");
net = connectLayers(net,"infc2","cct/in2");

% Plot network
plot(net)
```



```
% Convert to dlnetwork
net = dlnetwork(net);

% Display the number of weights
summary(net)

   Initialized: true

   Number of learnables: 386

   Inputs:
      1   'netObsIn1'   Sequence input with 4 dimensions
      2   'netObsIn2'   Sequence input with 1 dimensions
```

Create the critic with `rlVectorQValueFunction`, using the network and the observation and action specification objects.

```
critic = rlVectorQValueFunction(net,obsInfo,actInfo);
```

To return the value of the actions as a function of the current observation, use `getValue` or `evaluate`.

```
val = evaluate(critic, ...
                {rand(obsInfo(1).Dimension), ...
                 rand(obsInfo(2).Dimension)})
```

```
val = 1x1 cell array
    {3x1 single}
```

When you use `evaluate`, the result is a single-element cell array, containing a vector with the values of all the possible actions, given the observation.

```
val{1}
```

```
ans = 3x1 single column vector

   -0.0054
   -0.0943
    0.0177
```

Calculate the gradients of the sum of the outputs with respect to the inputs, given a random observation.

```
gro = gradient(critic,"output-input", ...
                {rand(obsInfo(1).Dimension) , ...
                 rand(obsInfo(2).Dimension) } )
```

```
gro=2×1 cell array
    {4x1 single}
    {[ -0.0396]}
```

The result is a cell array with as many elements as the number of input channels. Each element contains the derivative of the sum of the outputs with respect to each component of the input channel. Display the gradient with respect to the element of the second channel.

```
gro{2}
```

```
ans = single
    -0.0396
```

Obtain the gradient with respect of five independent sequences each one made of nine sequential observations.

```
gro_batch = gradient(critic,"output-input", ...
                        {rand([obsInfo(1).Dimension 5 9]) , ...
                         rand([obsInfo(2).Dimension 5 9]) } )
```

```
gro_batch=2×1 cell array
    {4x5x9 single}
    {1x5x9 single}
```

Display the derivative of the sum of the outputs with respect to the third observation element of the first input channel, after the seventh sequential observation in the fourth independent batch.

```
gro_batch{1}(3,4,7)
```

```
ans = single
    0.0443
```

Set the option to accelerate the gradient computations.

```
critic = accelerate(critic,true);
```

Calculate the gradients of the sum of the outputs with respect to the parameters, given a random observation.

```
grp = gradient(critic,"output-parameters", ...
                {rand(obsInfo(1).Dimension) , ...
                 rand(obsInfo(2).Dimension) } )
```

```
grp=9×1 cell array
    {[-0.0101 -0.0097 -0.0039 -0.0065]}
    {[                        -0.0122]}
    {[                        -0.0078]}
    {[                        -0.0863]}
    {32x2 single                       }
    {32x8 single                       }
    {32x1 single                       }
    { 3x8 single                       }
    { 3x1 single                       }
```

Each array within a cell contains the gradient of the sum of the outputs with respect to a group of parameters.

```
grp_batch = gradient(critic,"output-parameters", ...
                        {rand([obsInfo(1).Dimension 5 9]) , ...
                         rand([obsInfo(2).Dimension 5 9]) } )
```

```
grp_batch=9×1 cell array
    {[-1.5801 -1.2618 -2.2168 -1.3463]}
    {[                        -3.7742]}
    {[                        -6.2158]}
    {[                       -12.6841]}
    {32x2 single                       }
    {32x8 single                       }
    {32x1 single                       }
    { 3x8 single                       }
    { 3x1 single                       }
```

If you use a batch of inputs, `gradient` uses the whole input sequence (in this case nine steps), and all the gradients with respect to the independent batch dimensions (in this case five) are added together. Therefore, the returned gradient always has the same size as the output from `getLearnableParameters`.

Input Arguments

`fcnAppx` — Function approximator object

function approximator object

Function approximator object, specified:

- `rlValueFunction`,
- `rlQValueFunction`,
- `rlVectorQValueFunction`,

- `rlDiscreteCategoricalActor`,
- `rlContinuousDeterministicActor`,
- `rlContinuousGaussianActor`,
- `rlContinuousDeterministicTransitionFunction`,
- `rlContinuousGaussianTransitionFunction`,
- `rlContinuousDeterministicRewardFunction`,
- `rlContinuousGaussianRewardFunction`,
- `rlIsDoneFunction`.

`inData` — Input data for the function approximator
cell array

Input data for the function approximator, specified as a cell array with as many elements as the number of input channels of `fcnAppx`. In the following section, the number of observation channels is indicated by N_O.

- If `fcnAppx` is an `rlQValueFunction`, an `rlContinuousDeterministicTransitionFunction` or an `rlContinuousGaussianTransitionFunction` object, then each of the first N_O elements of `inData` must be a matrix representing the current observation from the corresponding observation channel. They must be followed by a final matrix representing the action.
- If `fcnAppx` is a function approximator object representing an actor or critic (but not an `rlQValueFunction` object), `inData` must contain N_O elements, each one being a matrix representing the current observation from the corresponding observation channel.
- If `fcnAppx` is an `rlContinuousDeterministicRewardFunction`, an `rlContinuousGaussianRewardFunction`, or an `rlIsDoneFunction` object, then each of the first N_O elements of `inData` must be a matrix representing the current observation from the corresponding observation channel. They must be followed by a matrix representing the action, and finally by N_O elements, each one being a matrix representing the next observation from the corresponding observation channel.

Each element of `inData` must be a matrix of dimension M_C-by-L_B-by-L_S, where:

- M_C corresponds to the dimensions of the associated input channel.
- L_B is the batch size. To specify a single observation, set $L_B = 1$. To specify a batch of (independent) observations, specify $L_B > 1$. If `inData` has multiple elements, then L_B must be the same for all elements of `inData`.
- L_S specifies the sequence length (along the time dimension) for recurrent neural network. If `fcnAppx` does not use a recurrent neural network, (which is the case of environment function approximators, as they do not support recurrent neural networks) then $L_S = 1$. If `inData` has multiple elements, then L_S must be the same for all elements of `inData`.

For more information on input and output formats for recurrent neural networks, see the Algorithms section of `lstmLayer`.

Example: `{rand(8,3,64,1),rand(4,1,64,1),rand(2,1,64,1)}`

`lossFcn` — Loss function
function handle

Loss function, specified as a function handle to a user-defined function. The user defined function can either be an anonymous function or a function on the MATLAB path. The function first input parameter must be a cell array like the one returned from the evaluation of `fcnAppx`. For more information, see the description of `outData` in `evaluate`. The second, optional, input argument of `lossFcn` contains additional data that might be needed for the gradient calculation, as described below in `fcnData`. For an example of the signature that this function must have, see "Train Reinforcement Learning Policy Using Custom Training Loop".

`fcnData` — Additional input data for loss function
any MATLAB data type

Additional data for the loss function, specified as any MATLAB data type, typically a structure or cell array. For an example see "Train Reinforcement Learning Policy Using Custom Training Loop".

Output Arguments

`grad` — Value of the gradient
cell array

Value of the gradient, returned as a cell array.

When the type of gradient is from the sum of the outputs with respect to the inputs of `fcnAppx`, then `grad` is a cell array in which each element contains the gradient of the sum of all the outputs with respect to the corresponding input channel.

The numerical array in each cell has dimensions D-by-L_B-by-L_S, where:

- D corresponds to the dimensions of the input channel of `fcnAppx`.
- L_B is the batch size (length of a batch of independent inputs).
- L_S is the sequence length (length of the sequence of inputs along the time dimension) for a recurrent neural network. If `fcnAppx` does not use a recurrent neural network (which is the case of environment function approximators, as they do not support recurrent neural networks), then $L_S = 1$.

When the type of gradient is from the output with respect to the parameters of `fcnAppx`, then `grad` is a cell array in which each element contains the gradient of the sum of outputs belonging to an output channel with respect to the corresponding group of parameters. The gradient is calculated using the whole history of L_S inputs, and all the L_B gradients with respect to the independent input sequences are added together in `grad`. Therefore, `grad` has always the same size as the result from `getLearnableParameters`.

For more information on input and output formats for recurrent neural networks, see the Algorithms section of `lstmLayer`.

Version History

Introduced in R2022a

See Also

Functions
`evaluate` | `accelerate` | `getLearnableParameters`

Objects

`rlValueFunction` | `rlQValueFunction` | `rlVectorQValueFunction` | `rlContinuousDeterministicActor` | `rlDiscreteCategoricalActor` | `rlContinuousGaussianActor` | `rlContinuousDeterministicTransitionFunction` | `rlContinuousGaussianTransitionFunction` | `rlContinuousDeterministicRewardFunction` | `rlContinuousGaussianRewardFunction` | `rlIsDoneFunction`

Topics

"Train Reinforcement Learning Policy Using Custom Training Loop"
"Create Custom Reinforcement Learning Agents"

hyperbolicPenalty

Hyperbolic penalty value for a point with respect to a bounded region

Syntax

```
p = hyperbolicPenalty(x,xmin,xmax)
p = hyperbolicPenalty( ___ ,lambda,tau)
```

Description

`p = hyperbolicPenalty(x,xmin,xmax)` calculates the nonnegative (hyperbolic) penalty vector `p` for the point `x` with respect to the region bounded by `xmin` and `xmax`. `p` has the same dimension as `x`. This syntax uses the default values of `1` and `0.1` for the `lambda` and `tau` parameters of the hyperbolic function, respectively.

`p = hyperbolicPenalty( ___ ,lambda,tau)` specifies both the `lambda` and `tau` parameters of the hyperbolic function. If `lambda` is an empty matrix its default value is used. Likewise if `tau` is an empty matrix or it is omitted, its default value is used instead.

Examples

Calculate Hyperbolic Penalty for a Point

This example shows how to use the `hyperbolicPenalty` function to calculate the hyperbolic penalty for a given point with respect to a bounded region.

Calculate the penalty value for the point `0.1` within the interval [`-2`,`2`], using default values for the `lambda` and `tau` parameters.

```
hyperbolicPenalty(0.1,-2,2)
```

```
ans = 0.0050
```

Calculate the penalty value for the point `4` outside the interval [`-2`,`2`].

```
hyperbolicPenalty(4,-2,2)
```

```
ans = 4.0033
```

Calculate the penalty value for the point `0.1` within the interval [`-2`,`2`], using a `lambda` parameter of `5`.

```
hyperbolicPenalty(0.1,-2,2,5)
```

```
ans = 0.0010
```

Calculate the penalty value for the point `4` outside the interval [`-2`,`2`], using a `lambda` parameter of `5`.

```
hyperbolicPenalty(4,-2,2,5)
```

```
ans = 20.0007
```

Calculate the penalty value for the point 4 outside the interval [-2,2], using a tau parameter of 0.5.

```
hyperbolicPenalty(4,-2,2,5,0.5)
```

```
ans = 20.0167
```

Calculate the penalty value for the point [-2,0,4] with respect to the box defined by the intervals [0,1], [-1,1], and [-2,2] along the x, y, and z dimensions, respectively, using the default value for lambda and a tau parameter of 0.

```
hyperbolicPenalty([-2 0 4],[0 -1 -2],[1 1 2],1,0)
```

```
ans = 3×1

     4
     0
     4
```

Visualize Penalty Values for an Interval

Create a vector of 1001 equidistant points distributed between -5 and 5.

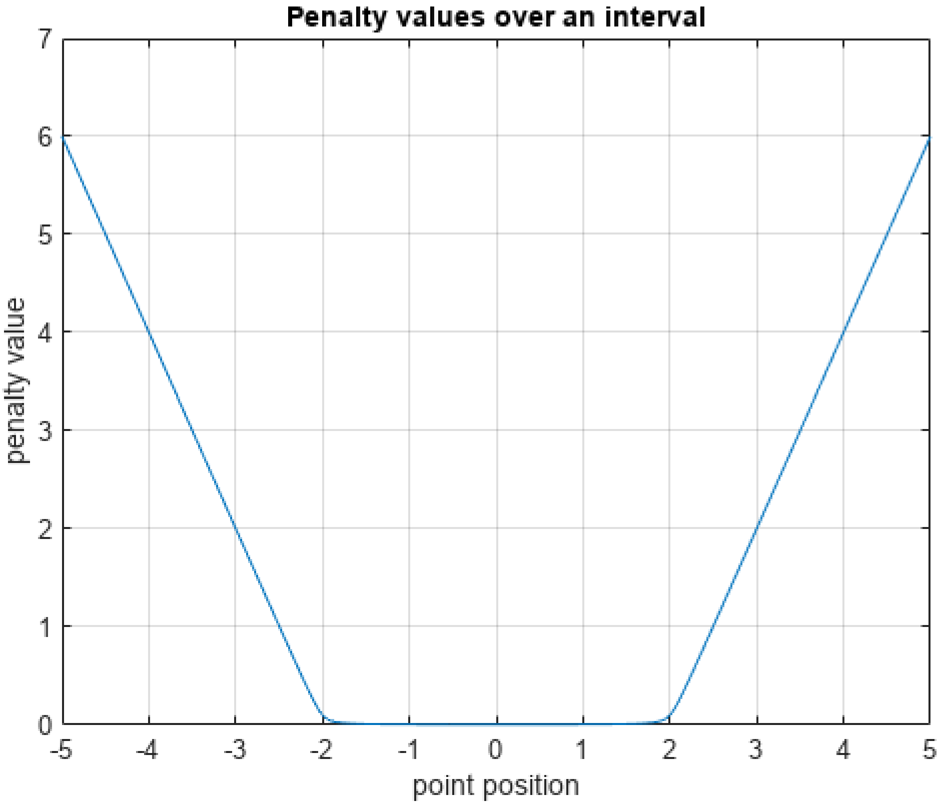
```
x = -5:0.01:5;
```

Calculate penalties for all the points in the vector, using default values for the lambda and tau parameters.

```
p = hyperbolicPenalty(x,-2,2);
```

Plot the points and add grid, labels and title.

```
plot(x,p)
grid
xlabel("point position");
ylabel("penalty value");
title("Penalty values over an interval");
```



Input Arguments

x — Point for which the penalty is calculated

scalar | vector | matrix

Point for which the penalty is calculated, specified as a numeric scalar, vector or matrix.

Example: `[0.5; 1.6]`

xmin — Lower bounds

scalar | vector | matrix

Lower bounds for `x`, specified as a numeric scalar, vector or matrix. To use the same minimum value for all elements in `x` specify `xmin` as a scalar.

Example: `-1`

xmax — Upper bounds

scalar | vector | matrix

Upper bounds for `x`, specified as a numeric scalar, vector or matrix. To use the same maximum value for all elements in `x` specify `xmax` as a scalar.

Example: 2

lambda — Lambda parameter of the hyperbolic function

`1` (default) | nonnegative scalar

Lambda parameter of the hyperbolic function, specified as a scalar.

Example: 3

`tau` — Tau parameter of the hyperbolic function

`0.1` (default) | nonnegative scalar

Tau parameter of the hyperbolic function, specified as a scalar.

Example: 0.3

Output Arguments

p — Penalty value

nonnegative vector

Penalty value, returned as a vector of nonnegative elements. Each element `p`$_i$ depends on the position of `x`$_i$ with respect to the interval specified by `xmin`$_i$ and `xmax`$_i$. The hyperbolic penalty function returns the value:

$$p(x) = \ -\lambda(x - x_{\mathrm{min}}) + \sqrt{\lambda^2(x - x_{\mathrm{min}})^2 + \tau^2} - \lambda(x_{\mathrm{max}} - x) + \sqrt{\lambda^2(x_{\mathrm{max}} - x)^2 + \tau^2}$$

Here, λ is the argument `lambda`, and τ is the argument `tau`. Note that for positive values of τ the returned penalty value is always positive, because on the right side of the equation the magnitude of the second term is always greater than that of the first, and the magnitude of the fourth term is always greater than that of the third. If τ is zero, then the returned penalty is zero inside the interval defined by the bounds, and it grows linearly with x outside this interval. If x is multidimensional, then the calculation is applied independently on each dimension. Penalty functions are typically used to generate negative rewards when constraints are violated, such as in `generateRewardFunction`.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`generateRewardFunction` | `exteriorPenalty` | `barrierPenalty`

Topics

“Generate Reward Function from a Model Predictive Controller for a Servomotor”
“Define Reward Signals”

inspectTrainingResult

Plot training information from a previous training session

Syntax

```
inspectTrainingResult(trainResults)
```

```
inspectTrainingResult(agentResults)
```

Description

By default, the `train` function shows the training progress and results in the Episode Manager during training. If you configure training to not show the Episode Manager or you close the Episode Manager after training, you can view the training results using the `inspectTrainingResult` function, which opens the Episode Manager. You can also use `inspectTrainingResult` to view the training results for agents saved during training.

`inspectTrainingResult(trainResults)` opens the Episode Manager and plots the training results from a previous training session.

`inspectTrainingResult(agentResults)` opens the Episode Manager and plots the training results from a previously saved agent structure.

Examples

View Results From Previous Training Session

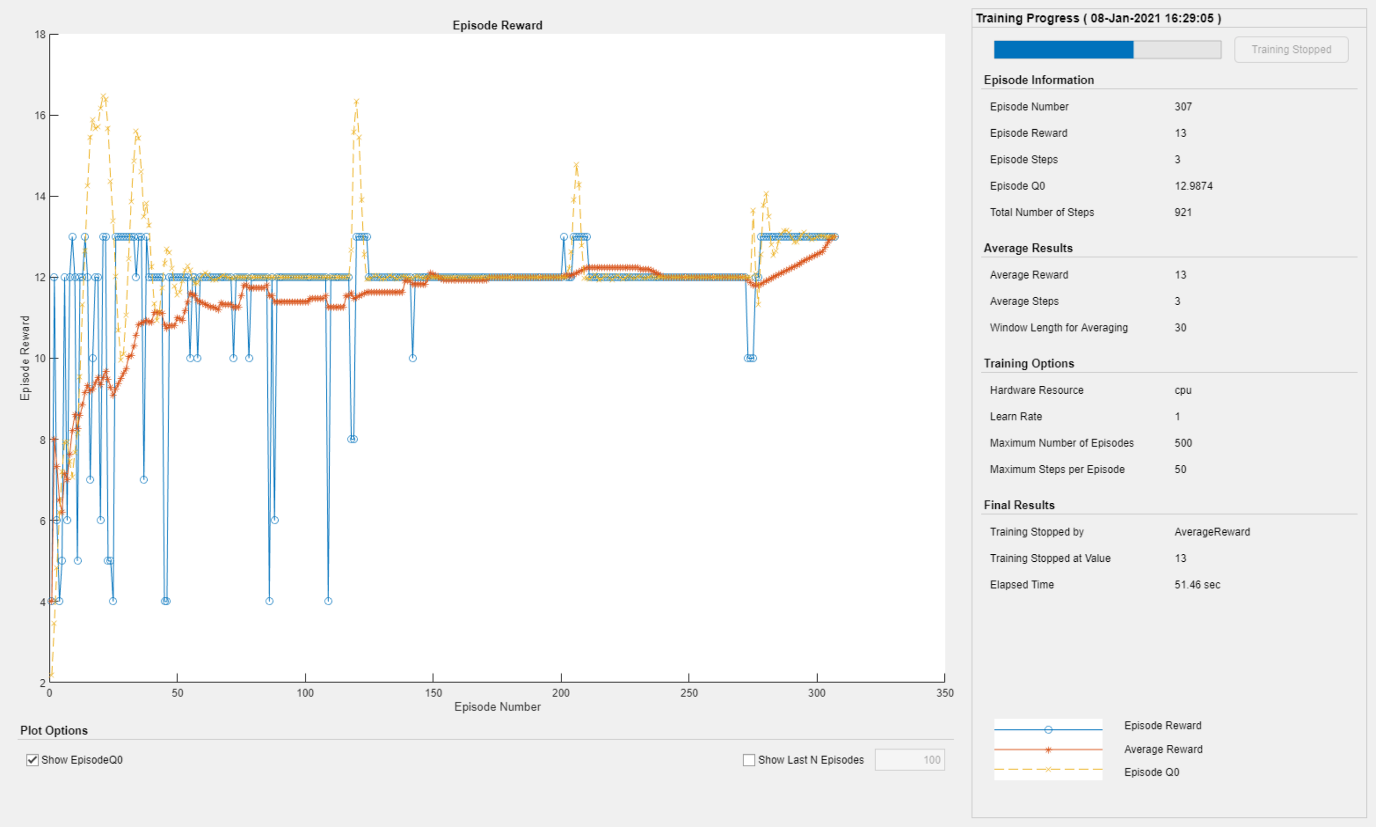
For this example, assume that you have trained the agent in the “Train Reinforcement Learning Agent in MDP Environment” example and subsequently closed the Episode Manager.

Load the training information returned by the `train` function.

```
load mdpTrainingStats trainingStats
```

Reopen the Episode Manager for this training session.

```
inspectTrainingResult(trainingStats)
```



Training Progress (08-Jan-2021 16:29:05)

Episode Information

Episode Number	307
Episode Reward	13
Episode Steps	3
Episode Q0	12.9874
Total Number of Steps	921

Average Results

Average Reward	13
Average Steps	3
Window Length for Averaging	30

Training Options

Hardware Resource	cpu
Learn Rate	1
Maximum Number of Episodes	500
Maximum Steps per Episode	50

Final Results

Training Stopped by	AverageReward
Training Stopped at Value	13
Elapsed Time	51.46 sec

Plot Options

☑ Show EpisodeQ0

□ Show Last N Episodes 100

View Training Results for Saved Agent

For this example, load the environment and agent for the “Train Reinforcement Learning Agent in MDP Environment” example.

```
load mdpAgentAndEnvironment
```

Specify options for training the agent. Configure the `SaveAgentCriteria` and `SaveAgentValue` options to save all agents after episode 30.

```
trainOpts = rlTrainingOptions;
trainOpts.MaxStepsPerEpisode = 50;
trainOpts.MaxEpisodes = 50;
trainOpts.Plots = "none";
trainOpts.SaveAgentCriteria = "EpisodeCount";
trainOpts.SaveAgentValue = 30;
```

Train the agent. During training, when an episode has a reward greater than or equal to 13, a copy of the agent is saved in a `savedAgents` folder.

```
rng("default") % for reproducibility
trainingStats = train(qAgent,env,trainOpts);
```

Load the training results for one of the saved agents. This command loads both the agent and a structure that contains the corresponding training results.

```
load savedAgents/Agent50
```

View the training results from the saved agent result structure.

```
inspectTrainingResult(savedAgentResult)
```

The Episode Manager shows the training progress up to the episode in which the agent was saved.

Input Arguments

`trainResults` — Training episode data

structure | structure array

Training episode data, specified as a structure or structure array returned by the `train` function.

`agentResults` — Saved agent results

structure

Saved agent results, specified as a structure previously saved by the `train` function. The `train` function saves agents when you specify the `SaveAgentCriteria` and `SaveAgentValue` options in the `rlTrainingOptions` object used during training.

When you load a saved agent, the agent and its training results are added to the MATLAB workspace as `saved_agent` and `savedAgentResultStruct`, respectively. To plot the training data for this agent, use the following command.

```
inspectTrainingResult(savedAgentResultStruct)
```

For multi-agent training, `savedAgentResultStruct` contains structure fields with training results for all the trained agents.

Version History

Introduced in R2021a

See Also

Functions

`train`

Topics

"Train Reinforcement Learning Agents"

predict

Package: rl.function

Predict next observation, next reward, or episode termination given observation and action input data

Syntax

```
predNextObs = predict(tsnFcnAppx,obs,act)
predReward = predict(rwdFcnAppx,obs,act,nextObs)
predIsDone = predict(idnFcnAppx,obs,act)
```

Description

predNextObs = predict(tsnFcnAppx,obs,act) evaluates the environment transition function approximator object tsnFcnAppx and returns the predicted next observation nextObs, given the current observation obs and the action act.

predReward = predict(rwdFcnAppx,obs,act,nextObs) evaluates the environment reward function approximator object rwdFcnAppx and returns the predicted reward predReward, given the current observation obs, the action act, and the next observation nextObs.

predIsDone = predict(idnFcnAppx,obs,act) evaluates the environment is-done function approximator object idnFcnAppx and returns the predicted is-done status predIsDone, given the current observation obs, the action act, and the next observation nextObs.

Examples

Predict Next Observation Using Continuous Gaussian Transition Function Approximator

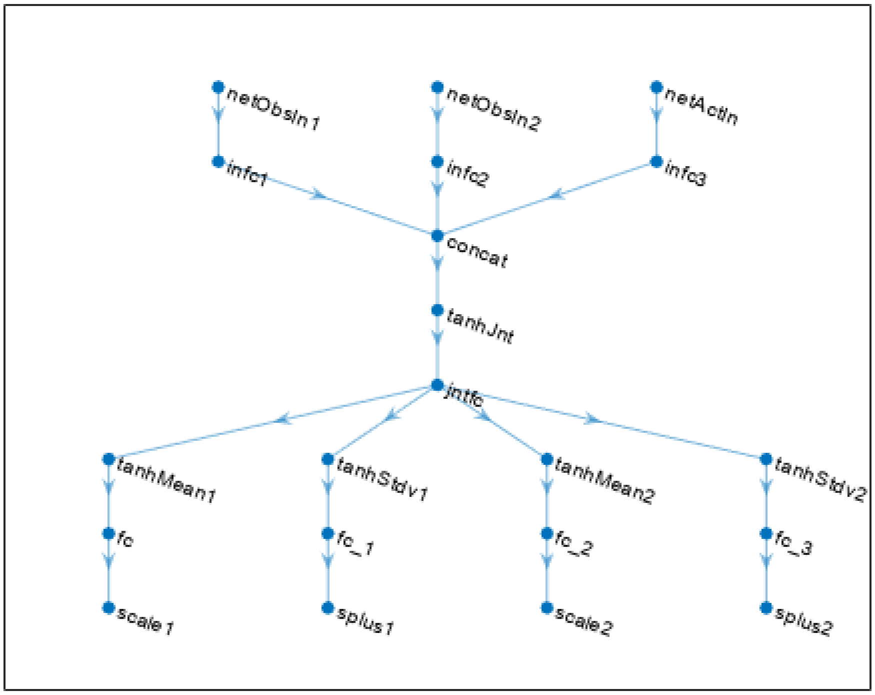
Create observation and action specification objects (or alternatively use getObservationInfo and getActionInfo to extract the specification objects from an environment). For this example, two observation channels carry vectors in a four- and two-dimensional space, respectively. The action is a continuous three-dimensional vector.

```
obsInfo = [rlNumericSpec([4 1],UpperLimit=10*ones(4,1));
           rlNumericSpec([1 2],UpperLimit=20*ones(1,2)) ];

actInfo = rlNumericSpec([3 1]);
```

Create a deep neural network to use as approximation model for the transition function approximator. For a continuous Gaussian transition function approximator, the network must have two output layers for each observation (one for the mean values the other for the standard deviation values).

Define each network path as an array of layer objects. Get the dimensions of the observation and action spaces from the environment specification objects, and specify a name for the input layers, so you can later explicitly associate them with the appropriate environment channel.

```
% Input path layers from first observation channel
inPath1 = [ featureInputLayer( ...
                prod(obsInfo(1).Dimension), ...
                Name="netObsIn1")
            fullyConnectedLayer(5,Name="infc1") ];

% Input path layers from second observation channel
inPath2 = [ featureInputLayer( ...
                prod(obsInfo(2).Dimension), ...
                Name="netObsIn2")
            fullyConnectedLayer(5,Name="infc2") ];

% Input path layers from action channel
inPath3 = [ featureInputLayer( ...
                prod(actInfo(1).Dimension), ...
                Name="netActIn")
            fullyConnectedLayer(5,Name="infc3") ];

% Joint path layers, concatenate 3 inputs along first dimension
jointPath = [ concatenationLayer(1,3,Name="concat")
              tanhLayer(Name="tanhJnt");
              fullyConnectedLayer(10,Name="jntfc") ];

% Path layers for mean values of first predicted obs
% Using scalingLayer to scale range from (-1,1) to (-10,10)
% Note that scale vector must be a column vector
meanPath1 = [ tanhLayer(Name="tanhMean1");
              fullyConnectedLayer(prod(obsInfo(1).Dimension));
              scalingLayer(Name="scale1", ...
                Scale=obsInfo(1).UpperLimit) ];

% Path layers for standard deviations first predicted obs
% Using softplus layer to make them non negative
sdevPath1 = [ tanhLayer(Name="tanhStdv1");
              fullyConnectedLayer(prod(obsInfo(1).Dimension));
              softplusLayer(Name="splus1") ];

% Path layers for mean values of second predicted obs
% Using scalingLayer to scale range from (-1,1) to (-20,20)
% Note that scale vector must be a column vector
meanPath2 = [ tanhLayer(Name="tanhMean2");
              fullyConnectedLayer(prod(obsInfo(2).Dimension));
              scalingLayer(Name="scale2", ...
                Scale=obsInfo(2).UpperLimit(:)) ];

% Path layers for standard deviations second predicted obs
% Using softplus layer to make them non negative
sdevPath2 = [ tanhLayer(Name="tanhStdv2");
              fullyConnectedLayer(prod(obsInfo(2).Dimension));
              softplusLayer(Name="splus2") ];

% Add layers to network object
net = layerGraph;
net = addLayers(net,inPath1);
net = addLayers(net,inPath2);
net = addLayers(net,inPath3);
net = addLayers(net,jointPath);
net = addLayers(net,meanPath1);
net = addLayers(net,sdevPath1);
net = addLayers(net,meanPath2);
net = addLayers(net,sdevPath2);

% Connect layers
net = connectLayers(net,"infc1","concat/in1");
net = connectLayers(net,"infc2","concat/in2");
net = connectLayers(net,"infc3","concat/in3");
net = connectLayers(net,"jntfc","tanhMean1/in");
net = connectLayers(net,"jntfc","tanhStdv1/in");
net = connectLayers(net,"jntfc","tanhMean2/in");
net = connectLayers(net,"jntfc","tanhStdv2/in");

% Plot network
plot(net)
```



```
% Convert to dlnetwork
net=dlnetwork(net);

% Display the number of weights
summary(net)
```

```
   Initialized: true

   Number of learnables: 352

   Inputs:
      1   'netObsIn1'   4 features
      2   'netObsIn2'   2 features
      3   'netActIn'    3 features
```

Create a continuous Gaussian transition function approximator object, specifying the names of all the input and output layers.

```
tsnFcnAppx = rlContinuousGaussianTransitionFunction(...
    net,obsInfo,actInfo,...
    ObservationInputNames=["netObsIn1","netObsIn2"], ...
    ActionInputNames="netActIn", ...
    NextObservationMeanOutputNames=["scale1","scale2"], ...
    NextObservationStandardDeviationOutputNames=["splus1","splus2"] );
```

Predict the next observation for a random observation and action.

```
predObs = predict(tsnFcnAppx, ...
    {rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)}, ...
    {rand(actInfo(1).Dimension)})
```

```
predObs=1×2 cell array
    {4x1 single}    {[-24.9934 0.9501]}
```

Each element of the resulting cell array represents the prediction for the corresponding observation channel.

To display the mean values and standard deviations of the Gaussian probability distribution for the predicted observations, use `evaluate`.

```
predDst = evaluate(tsnFcnAppx, ...
    {rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension), ...
     rand(actInfo(1).Dimension)})
```

```
predDst=1×4 cell array
    {4x1 single}    {2x1 single}    {4x1 single}    {2x1 single}
```

The result is a cell array in which the first and second element represent the mean values for the predicted observations in the first and second channel, respectively. The third and fourth element represent the standard deviations for the predicted observations in the first and second channel, respectively.

Create Deterministic Reward Function and Predict Reward

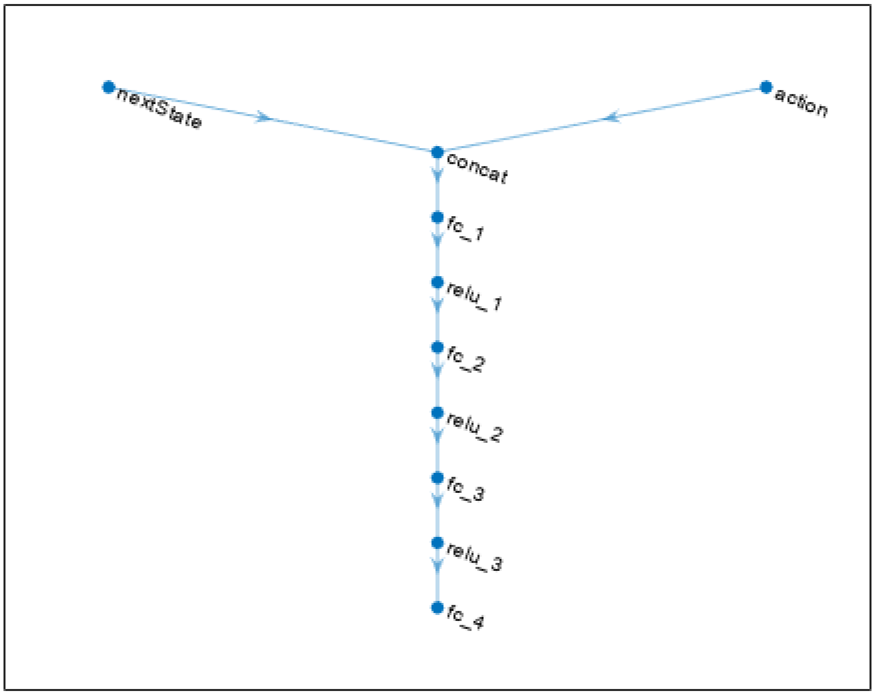
Create an environment interface and extract observation and action specifications. Alternatively, you can create specifications using `rlNumericSpec` and `rlFiniteSetSpec`.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

To approximate the reward function, create a deep neural network. For this example, the network has two input layers, one for the current action and one for the next observations. The single output layer contains a scalar, which represents the value of the predicted reward.

Define each network path as an array of layer objects. Get the dimensions of the observation and action spaces from the environment specifications, and specify a name for the input layers, so you can later explicitly associate them with the appropriate environment channel.

```
actionPath = featureInputLayer( ...
    actInfo.Dimension(1), ...
    Name="action");

nextStatePath = featureInputLayer( ...
    obsInfo.Dimension(1), ...
    Name="nextState");

commonPath = [concatenationLayer(1,2,Name="concat")
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(1)];

net = layerGraph(nextStatePath);
net = addLayers(net,actionPath);
net = addLayers(net,commonPath);

net = connectLayers(net,"nextState","concat/in1");
net = connectLayers(net,"action","concat/in2");

plot(net)
```



Create a `dlnetwork` object and display the number of weights.

```
net = dlnetwork(net);
summary(net);
```

```
   Initialized: true

   Number of learnables: 8.7k

   Inputs:
      1   'nextState'   4 features
      2   'action'      1 features
```

Create a deterministic transition function object.

```
rwdFcnAppx = rlContinuousDeterministicRewardFunction(...
    net,obsInfo,actInfo,...
    ActionInputNames="action", ...
    NextObservationInputNames="nextState");
```

Using this reward function object, you can predict the next reward value based on the current action and next observation. For example, predict the reward for a random action and next observation. Since, for this example, only the action and the next observation influence the reward, use an empty cell array for the current observation.

```
act = rand(actInfo.Dimension);
nxtobs = rand(obsInfo.Dimension);
reward = predict(rwdFcnAppx, {}, {act}, {nxtobs})
```

```
reward = single
    0.1034
```

To predict the reward, you can also use `evaluate`.

```
reward_ev = evaluate(rwdFcnAppx, {act,nxtobs} )
```

```
reward_ev = 1x1 cell array
    {[0.1034]}
```

Create Is-Done Function and Predict Termination

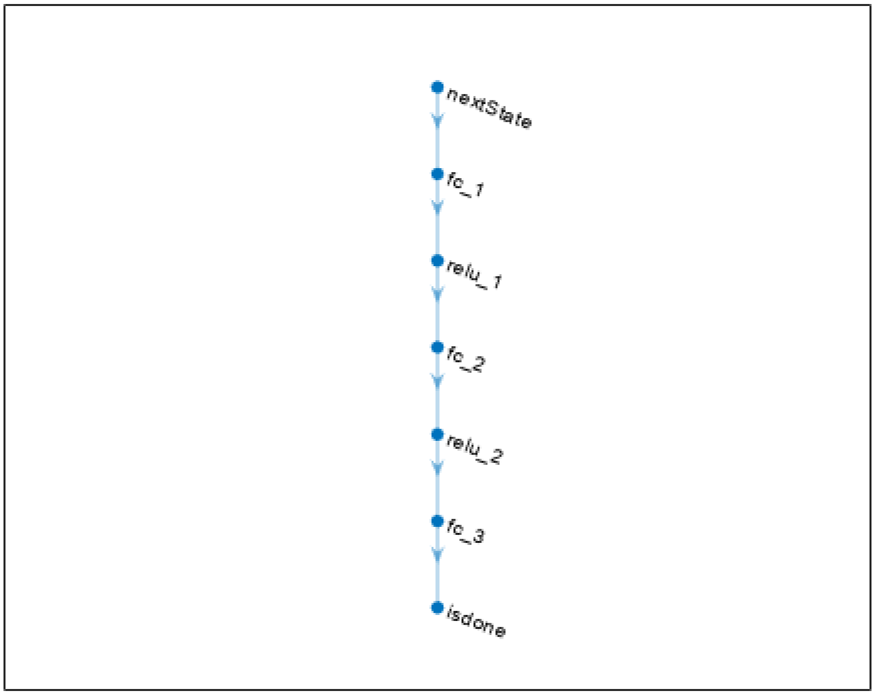
Create an environment interface and extract observation and action specifications. Alternatively, you can create specifications using `rlNumericSpec` and `rlFiniteSetSpec`.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

To approximate the is-done function, use a deep neural network. The network has one input channel for the next observations. The single output channel is for the predicted termination signal.

Create the neural network as a vector of layer objects.

```
commonPath = [
    featureInputLayer( ...
                obsInfo.Dimension(1), ...
                Name="nextState")
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(2)
    softmaxLayer(Name="isdone")];

net = layerGraph(commonPath);

plot(net)
```



Covert the network to a `dlnetwork` object and display the number of weights.

```
net = dlnetwork(net);
summary(net);
```

```
   Initialized: true

   Number of learnables: 4.6k

   Inputs:
      1   'nextState'   4 features
```

Create an is-done function approximator object.

```
isDoneFcnAppx = rlIsDoneFunction(...
    net,obsInfo,actInfo,...
    NextObservationInputNames="nextState");
```

Using this is-done function approximator object, you can predict the termination signal based on the next observation. For example, predict the termination signal for a random next observation. Since for this example the termination signal only depends on the next observation, use empty cell arrays for the current action and observation inputs.

```
nxtobs = rand(obsInfo.Dimension);
predIsDone = predict(isDoneFcnAppx,{},{},{nxtobs})
```

```
predIsDone = 0
```

You can obtain the termination probability using `evaluate`.

```
predIsDoneProb = evaluate(isDoneFcnAppx,{nxtobs})
```

```
predIsDoneProb = 1x1 cell array
    {2x1 single}
```

```
predIsDoneProb{1}
```

```
ans = 2x1 single column vector

    0.5405
    0.4595
```

The first number is the probability of obtaining a `0` (no termination predicted), the second one is the probability of obtaining a `1` (termination predicted).

Input Arguments

`tsnFcnAppx` — Environment transition function approximator object
`rlContinuousDeterministicTransitionFunction` object | `rlContinuousGaussianTransitionFunction` object

Environment transition function approximator object, specified as one of the following:

- `rlContinuousDeterministicTransitionFunction` object
- `rlContinuousGaussianTransitionFunction` object

`rwdFcnAppx` — Environment reward function
`rlContinuousDeterministicRewardFunction` object | `rlContinuousGaussianRewardFunction` object | function handle

Environment reward function approximator object, specified as one of the following:

- `rlContinuousDeterministicRewardFunction` object
- `rlContinuousGaussianRewardFunction` object
- Function handle object. For more information about function handle objects, see “What Is a Function Handle?”.

`idnFcnAppx` — Environment is-done function approximator object
`rlIsDoneFunction` object

Environment is-done function approximator object, specified as an `rlIsDoneFunction` object.

obs — Observations

cell array

Observations, specified as a cell array with as many elements as there are observation input channels. Each element of obs contains an array of observations for a single observation input channel.

The dimensions of each element in obs are M_O-by-L_B, where:

- M_O corresponds to the dimensions of the associated observation input channel.
- L_B is the batch size. To specify a single observation, set $L_B = 1$. To specify a batch of observations, specify $L_B > 1$. If valueRep or qValueRep has multiple observation input channels, then L_B must be the same for all elements of obs.

L_B must be the same for both act and obs.

For more information on input and output formats for recurrent neural networks, see the Algorithms section of lstmLayer.

act — Action

single-element cell array

Action, specified as a single-element cell array that contains an array of action values.

The dimensions of this array are M_A-by-L_B, where:

- M_A corresponds to the dimensions of the associated action specification.
- L_B is the batch size. To specify a single observation, set $L_B = 1$. To specify a batch of observations, specify $L_B > 1$.

L_B must be the same for both act and obs.

For more information on input and output formats for recurrent neural networks, see the Algorithms section of lstmLayer.

nextObs — Next observation

cell array

Next observations, that is the observation following the action act from the observation obs, specified as a cell array of the same dimension as obs.

Output Arguments

predNextObs — Predicted next observation

cell array

Predicted next observation, that is the observation predicted by the transition function approximator tsnFcnAppx given the current observation obs and the action act, retuned as a cell array of the same dimension as obs.

predReward — Predicted next observation

single

Predicted reward, that is the reward predicted by the reward function approximator `rwdFcnAppx` given the current observation `obs`, the action `act`, and the following observation `nextObs`, retuned as a `single`.

`predIsDone` — Predicted next observation
`double`

Predicted is-done episode status, that is the episode termination status predicted by the is-done function approximator `rwdFcnAppx` given the current observation `obs`, the action `act`, and the following observation `nextObs`, returned as a `double`.

Note If `fcnAppx` is an `rlContinuousDeterministicRewardFunction` object, then `evaluate` behaves identically to `predict` except that it returns results inside a single-cell array. If `fcnAppx` is an `rlContinuousDeterministicTransitionFunction` object, then `evaluate` behaves identically to `predict`. If `fcnAppx` is an `rlContinuousGaussianTransitionFunction` object, then `evaluate` returns the mean value and standard deviation the observation probability distribution, while `predict` returns an observation sampled from this distribution. Similarly, for an `rlContinuousGaussianRewardFunction` object, `evaluate` returns the mean value and standard deviation the reward probability distribution, while `predict` returns a reward sampled from this distribution. Finally, if `fcnAppx` is an `rlIsDoneFunction` object, then `evaluate` returns the probabilities of the termination status being false or true, respectively, while `predict` returns a predicted termination status sampled with these probabilities.

Version History

Introduced in R2022a

See Also

Functions
`evaluate` | `accelerate` | `gradient`

Objects
`rlNeuralNetworkEnvironment` | `rlContinuousDeterministicTransitionFunction` | `rlContinuousGaussianTransitionFunction` | `rlContinuousDeterministicRewardFunction` | `rlContinuousGaussianRewardFunction` | `rlIsDoneFunction`

Topics
"Model-Based Policy Optimization (MBPO) Agents"

reset

Package: `rl.policy`

Reset environment, agent, experience buffer, or policy object

Syntax

```
initialObs = reset(env)
```

```
reset(agent)
agent = reset(agent)
```

```
resetPolicy = reset(policy)
```

```
reset(buffer)
```

Description

`initialObs = reset(env)` resets the specified MATLAB environment to an initial state and returns the resulting initial observation value.

Do not use `reset` for Simulink environments, which are implicitly reset when running a new simulation. Instead, customize the reset behavior using the `ResetFcn` property of the environment.

`reset(agent)` resets the specified agent. Resetting a built-in agent performs the following actions, if applicable.

- Empty experience buffer.
- Set recurrent neural network states of actor and critic networks to zero.
- Reset the states of any noise models used by the agent.

`agent = reset(agent)` also returns the reset agent as an output argument.

`resetPolicy = reset(policy)` returns the policy object `resetPolicy` in which any recurrent neural network states are set to zero and any noise model states are set to their initial conditions. This syntax has no effect if the policy object does not use a recurrent neural network and does not have a noise model with state.

`reset(buffer)` resets the specified replay memory buffer by removing all the experiences.

Examples

Reset Environment

Create a reinforcement learning environment. For this example, create a continuous-time cart-pole system.

```
env = rlPredefinedEnv("CartPole-Continuous");
```

Reset the environment and return the initial observation.

```
initialObs = reset(env)
```

```
initialObs = 4×1

         0
         0
    0.0315
         0
```

Reset Agent

Create observation and action specifications.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlNumericSpec([1 1]);
```

Create a default DDPG agent using these specifications.

```
initOptions = rlAgentInitializationOptions(UseRNN=true);
agent = rlDDPGAgent(obsInfo,actInfo,initOptions);
```

Reset the agent.

```
agent = reset(agent);
```

Reset Experience Buffer

Create observation and action specifications.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlNumericSpec([1 1]);
```

Create a replay memory experience buffer.

```
buffer = rlReplayMemory(obsInfo,actInfo,10000);
```

Add experiences to the buffer. For this example, add 20 random experiences.

```
for i = 1:20
    expBatch(i).Observation = {obsInfo.UpperLimit.*rand(4,1)};
    expBatch(i).Action = {actInfo.UpperLimit.*rand(1,1)};
    expBatch(i).NextObservation = {obsInfo.UpperLimit.*rand(4,1)};
    expBatch(i).Reward = 10*rand(1);
    expBatch(i).IsDone = 0;
end
expBatch(20).IsDone = 1;

append(buffer,expBatch);
```

Reset and clear the buffer.

```
reset(buffer)
```

Reset Policy

Create observation and action specifications.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlFiniteSetSpec([-1 0 1]);
```

To approximate the Q-value function within the critic, use a deep neural network. Create each network path as an array of layer objects.

```
% Create Paths
obsPath = [featureInputLayer(4)
           fullyConnectedLayer(1,Name="obsout")];

actPath = [featureInputLayer(1)
           fullyConnectedLayer(1,Name="actout")];

comPath = [additionLayer(2,Name="add")  ...
           fullyConnectedLayer(1)];

% Add Layers
net = layerGraph;
net = addLayers(net,obsPath);
net = addLayers(net,actPath);
net = addLayers(net,comPath);
net = connectLayers(net,"obsout","add/in1");
net = connectLayers(net,"actout","add/in2");

% Convert to dlnetwork object
net = dlnetwork(net);

% Diplay the number of weights
summary(net)
```

```
   Initialized: true

   Number of learnables: 9

   Inputs:
      1   'input'     4 features
      2   'input_1'   1 features
```

Create an epsilon-greedy policy object using a Q-value function approximator.

```
critic = rlQValueFunction(net,obsInfo,actInfo);
policy = rlEpsilonGreedyPolicy(critic)
```

```
policy = 
  rlEpsilonGreedyPolicy with properties:

            QValueFunction: [1x1 rl.function.rlQValueFunction]
        ExplorationOptions: [1x1 rl.option.EpsilonGreedyExploration]
    UseEpsilonGreedyAction: 1
        EnableEpsilonDecay: 1
```

```
          ObservationInfo: [1x1 rl.util.rlNumericSpec]
               ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
               SampleTime: -1
```

Reset the policy.

```
policy = reset(policy);
```

Input Arguments

env — Reinforcement learning environment
environment object | ...

Reinforcement learning environment, specified as one of the following objects.

- `rlFunctionEnv` — Environment defined using custom functions
- `rlMDPEnv` — Markov decision process environment
- `rlNeuralNetworkEnvironment` — Environment with deep neural network transition models
- Predefined environment created using `rlPredefinedEnv`
- Custom environment created from a template (`rlCreateEnvTemplate`)

agent — Reinforcement learning agent
reinforcement learning agent object | ...

Reinforcement learning agent, specified as one of the following objects.

- `rlQAgent`
- `rlSARSAAgent`
- `rlDQNAgent`
- `rlPGAgent`
- `rlDDPGAgent`
- `rlTD3Agent`
- `rlACAgent`
- `rlSACAgent`
- `rlPPOAgent`
- `rlTRPOAgent`
- `rlMBPOAgent`
- Custom agent — For more information, see “Create Custom Reinforcement Learning Agents”.

Note `agent` is a handle object, so it is reset whether it is returned as an output argument or not. For more information about handle objects, see “Handle Object Behavior”.

buffer — Experience buffer
`rlReplayMemory` object | `rlPrioritizedReplayMemory` object | `rlHindsightReplayMemory` object | `rlHindsightPrioritizedReplayMemory` object

Experience buffer, specified as one of the following replay memory objects.

- `rlReplayMemory`
- `rlPrioritizedReplayMemory`
- `rlHindsightReplayMemory`
- `rlHindsightPrioritizedReplayMemory`

`policy` — Reinforcement learning policy
`rlMaxQPolicy` | `rlEpsilonGreedyPolicy` | `rlDeterministicActorPolicy` | `rlAdditiveNoisePolicy` | `rlStochasticActorPolicy`

Reinforcement learning policy, specified as one of the following objects:

- `rlMaxQPolicy`
- `rlEpsilonGreedyPolicy`
- `rlDeterministicActorPolicy`
- `rlAdditiveNoisePolicy`
- `rlStochasticActorPolicy`

Output Arguments

`initialObs` — Initial environment observation
array | cell array

Initial environment observation after reset, returned as one of the following:

- Array with dimensions matching the observation specification for an environment with a single observation channel.
- Cell array with length equal to the number of observation channel for an environment with multiple observation channels. Each element of the cell array contains an array with dimensions matching the corresponding element of the environment observation specifications.

`resetPolicy` — Reset agent
`rlDeterministicActorPolicy` object | `rlAdditiveNoisePolicy` object | `rlEpsilonGreedyPolicy` object | `rlMaxQPolicy` object | `rlStochasticActorPolicy` object

Reset policy, returned as a policy object of the same type as `agent` but with its recurrent neural network states set to zero.

`agent` — Reset agent
reinforcement learning agent object

Reset agent, returned as an agent object. Note that `agent` is a handle object. Therefore, if it contains any recurrent neural network, its state is reset whether `agent` is returned as an output argument or not. For more information about handle objects, see “Handle Object Behavior”.

Version History

Introduced in R2022a

See Also

Functions
`runEpisode` | `setup` | `cleanup`

resize

Package: `rl.replay`

Resize replay memory experience buffer

Syntax

```
resize(buffer,maxLength)
```

Description

`resize(buffer,maxLength)` resizes experience buffer `buffer` to have a maximum length of `maxLength`.

- If `maxLength` is greater than or equal to the number of experiences stored in the buffer, then `buffer` retains its stored experiences.
- If `maxLength` is less than the number of experiences stored in the buffer, then `buffer` retains only the `maxLength` most recent experiences.

Examples

Resize Experience Buffer in Reinforcement Learning Agent

Create an environment for training the agent. For this example, load a predefined environment.

```
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");
```

Extract the observation and action specifications from the agent.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a DQN agent from the environment specifications.

```
agent = rlDQNAgent(obsInfo,actInfo);
```

By default, the agent uses an experience buffer with a maximum size of 10,000.

```
agent.ExperienceBuffer
```

```
ans = 
  rlReplayMemory with properties:

    MaxLength: 10000
       Length: 0
```

Increase the maximum size of the experience buffer to 20,000.

```
resize(agent.ExperienceBuffer,20000)
```

View the updated experience buffer.

```
agent.ExperienceBuffer
```

```
ans = 
  rlReplayMemory with properties:

    MaxLength: 20000
       Length: 0
```

Input Arguments

`buffer` — Experience buffer
`rlReplayMemory` object | `rlPrioritizedReplayMemory` object | `rlHindsightReplayMemory` object | `rlHindsightPrioritizedReplayMemory` object

Experience buffer, specified as one of the following replay memory objects.

- `rlReplayMemory`
- `rlPrioritizedReplayMemory`
- `rlHindsightReplayMemory`
- `rlHindsightPrioritizedReplayMemory`

`maxLength` — Maximum buffer length
nonnegative integer

This property is read-only.

Maximum buffer length, specified as a nonnegative integer.

Version History

Introduced in R2022b

See Also

Objects
`rlReplayMemory` | `rlPrioritizedReplayMemory` | `rlHindsightReplayMemory` | `rlHindsightPrioritizedReplayMemory`

rlCreateEnvTemplate

Create custom reinforcement learning environment template

Syntax

```
rlCreateEnvTemplate(className)
```

Description

`rlCreateEnvTemplate(className)` creates and opens a MATLAB script that contains a template class representing a reinforcement learning environment. The template class contains an implementation of a simple cart-pole balancing environment. To define your custom environment, modify this template class. For more information, see “Create Custom MATLAB Environment from Template”.

Examples

Create Custom Environment Template File

This example shows how to create and open a template file for a reinforcement learning environment.

For this example, name the class `myEnvClass`

```
rlCreateEnvTemplate("myEnvClass")
```

This function opens a MATLAB® script that contains the class. By default, this template class describes a simple cart-pole environment.

Modify this template class, and save the file as `myEnvClass.m`

Input Arguments

`className` — Name of environment class
string | character vector

Name of environment class, specified as a string or character vector. This name defines the name of the class and the name of the MATLAB script.

Version History

Introduced in R2019a

See Also

Topics
“Create MATLAB Reinforcement Learning Environments”

rlDataLogger

Create either a file logger object or a monitor logger object to log training data

Syntax

```
fileLgr = rlDataLogger()
monLgr = rlDataLogger(tpm)
```

Description

`fileLgr = rlDataLogger()` creates the `FileLogger` object `fileLgr` for logging training data to disk.

`monLgr = rlDataLogger(tpm)` creates the `MonitorLogger` object `monLgr` for logging training data to the `TrainingProgressMonitor` object `tpm`, and its associated window.

Examples

Log Data to Disk during Built-in Training

This example shows how to log data to disk when using `train`.

Create a `FileLogger` object using `rlDataLogger`.

```
logger = rlDataLogger();
```

Specify a directory to save logged data.

```
logger.LoggingOptions.LoggingDirectory = "myDataLog";
```

Create callback functions to log the data (for this example, see the helper function section), and specify the appropriate callback functions in the logger object.

```
logger.EpisodeFinishedFcn    = @myEpisodeFinishedFcn;
logger.AgentStepFinishedFcn  = @myAgentStepFinishedFcn;
logger.AgentLearnFinishedFcn = @myAgentLearnFinishedFcn;
```

To train the agent, you can now call `train`, passing `logger` as an argument such as in the following command.

```
trainResult = train(agent, env, trainOpts, Logger=logger);
```

While the training progresses, data will be logged to the specified directory, according to the rule specified in the `FileNameRule` property of `logger.LoggingOptions`.

```
logger.LoggingOptions.FileNameRule
```

```
ans = 
"loggedData<id>"
```

Example Logging Functions

Episode completion logging function. This function is automatically called by the training loop at the end of each episode, and must return a structure containing the episode-related data to log, such as experiences, simulation information, or initial observations.

```
function dataToLog = myEpisodeFinishedFcn(data)
    dataToLog.Experience = data.Experience;
end
```

Agent step completion logging function. This function is automatically called by the training loop at the end of each training step, and must return a structure containing the step-related data to log, such as for example the state of the agent exploration policy.

```
function dataToLog = myAgentStepFinishedFcn(data)
    dataToLog.State = getState(getExplorationPolicy(data.Agent));
end
```

Learn subroutine completion logging function. This function is automatically called by the training loop at the end of each learning subroutine, and must return a structure containing the learning-related data to log, such as the training losses of the actor and critic networks, or, for a model-based agent, the environment model training losses.

```
function dataToLog = myAgentLearnFinishedFcn(data)
    dataToLog.ActorLoss  = data.ActorLoss;
    dataToLog.CriticLoss = data.CriticLoss;
end
```

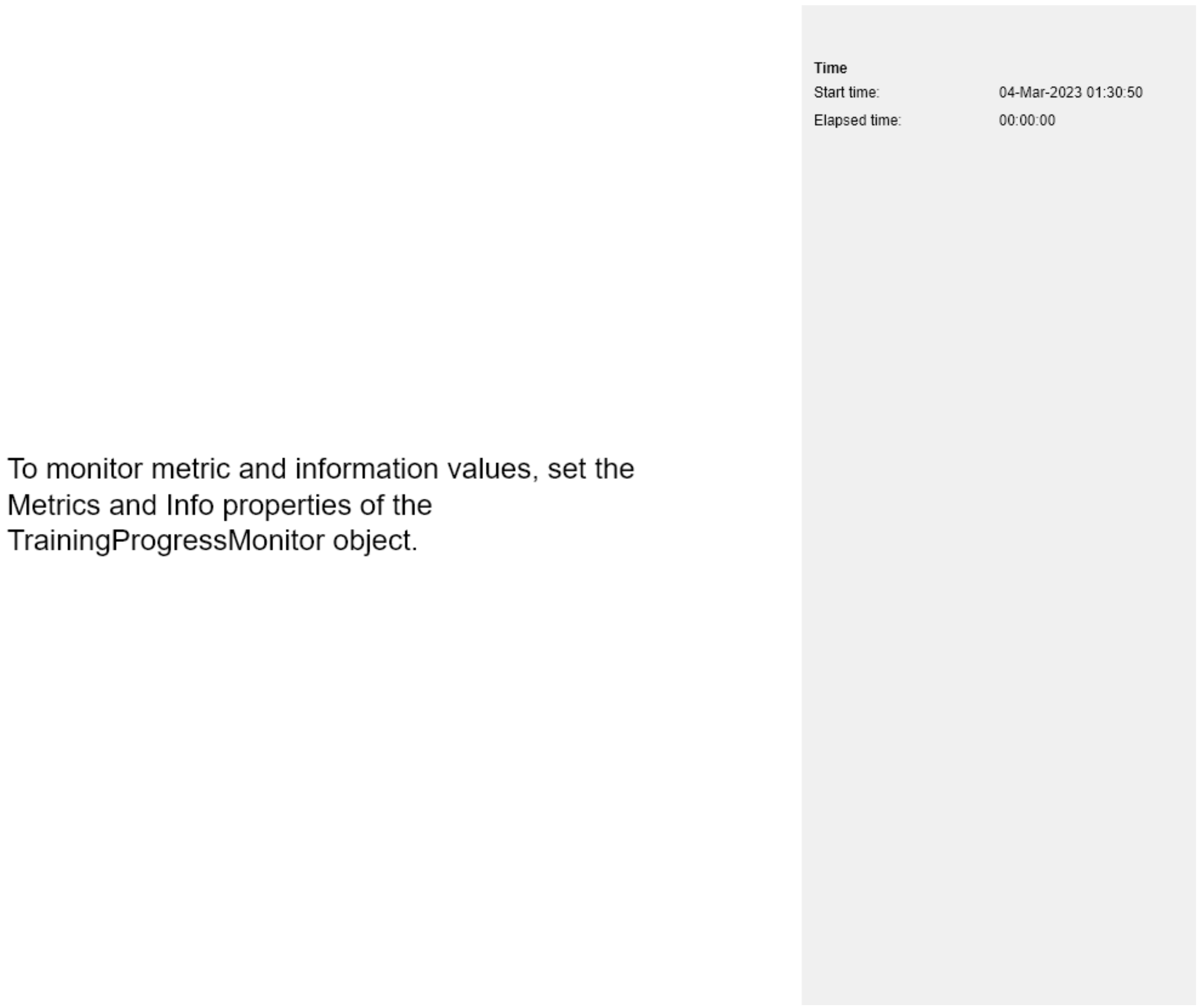
For the specific function signatures and more information on the function input structure, see the corresponding properties of `FileLogger`. For a related example, see “Log Training Data to Disk”.

Log and Visualize Data with a Training Progress Monitor Object

This example shows how to log and visualize data to the window of a `trainingProgressMonitor` object when using `train`.

Create a `trainingProgressMonitor` object. Creating the object also opens a window associated with the object.

```
monitor = trainingProgressMonitor();
```



To monitor metric and information values, set the Metrics and Info properties of the TrainingProgressMonitor object.

Time

Start time: 04-Mar-2023 01:30:50

Elapsed time: 00:00:00

Create a `MonitorLogger` object using `rlDataLogger`.

```
logger = rlDataLogger(monitor);
```

Create callback functions to log the data (for this example, see the helper function section), and specify the appropriate callback functions in the logger object. For the specific function signatures and more information on the function input structure, see the corresponding properties of `MonitorLogger`.

```
logger.AgentLearnFinishedFcn = @myAgentLearnFinishedFcn;
```

To train the agent, you can now call `train`, passing `logger` as an argument such as in the following command.

```
trainResult = train(agent, env, trainOpts, Logger=logger);
```

While the training progresses, data will be logged to the training monitor object, and visualized in the associated window.

Note that only scalar data can be logged with a monitor logger object.

Example Logging Functions

Define a logging function that logs data periodically at the completion of the learning subroutine. This function is automatically called by the training loop at the end of each learning subroutine, and must return a structure containing the learning-related data to log, such as the training losses of the actor and critic networks, or, for a model-based agent, the environment model training losses.

```
function dataToLog = myAgentLearnFinishedFcn(data)

    if mod(data.AgentLearnCount, 2) == 0
        dataToLog.ActorLoss  = data.ActorLoss;
        dataToLog.CriticLoss = data.CriticLoss;
    else
        dataToLog = [];
    end

end
```

Log Data to Disk in a Custom Training Loop

This example shows how to log data to disk when training an agent using a custom training loop.

Create a `FileLogger` object using `rlDataLogger`.

```
flgr = rlDataLogger();
```

Set up the logger object. This operation initializes the object performing setup tasks such as, for example, creating the directory to save the data files.

```
setup(flgr);
```

Within a custom training loop, you can now store data to the logger object memory and write data to file.

For this example, store random numbers to the file logger object, grouping them in the variables `Context1` and `Context2`. When you issue a write command, a MAT file corresponding to an iteration and containing both variables is saved with the name specified in `flgr.LoggingOptions.FileNameRule`, in the folder specified by `flgr.LoggingOptions.LoggingDirectory`.

```
for iter = 1:10

    % Store three random numbers in memory
    % as elements of the variable "Context1"
    for ct = 1:3
        store(flgr, "Context1", rand, iter);
    end

    % Store a random number in memory
    % as the variable "Context2"
    store(flgr, "Context2", rand, iter);

    % Write data to file every 4 iterations
    if mod(iter,4)==0
        write(flgr);
    end
end
```

Clean up the logger object. This operation performs clean up tasks like for example writing to file any data still in memory.

```
cleanup(flgr);
```

Input Arguments

`tpm` — Training progress monitor object

`trainingProgressMonitor` object

Training progress monitor object, specified as a `trainingProgressMonitor` object. This object is used to track the progress of training, update information fields, record metric values, and produce training plots for custom deep learning training loops. For more information, see “Monitor Custom Training Loop Progress”.

Output Arguments

`fileLgr` — File logger object

`FileLogger` object

File logger object, returned as a `FileLogger` object.

`monLgr` — Monitor logger object

`MonitorLogger` object

Monitor logger object, returned as a `MonitorLogger` object.

Limitations

- Only scalar data is supported when logging data with a `MonitorLogger` object. The structure returned by the callback functions must contain fields with scalar data.
- Resuming of training from a previous training result is not supported when logging data with a `MonitorLogger` object.
- Logging data using the `AgentStepFinishedFcn` callback is not supported when training agents in parallel with the train function.

Version History

Introduced in R2022b

See Also

Functions

`rlDataViewer` | `train`

Objects

`FileLogger` | `MonitorLogger` | `trainingProgressMonitor`

Topics
“Log Training Data to Disk”
“Monitor Custom Training Loop Progress”

rlDataViewer

Open Reinforcement Learning Data Viewer tool

Syntax

```
rlDataViewer(fileLgr)
rlDataViewer(folders)
```

Description

`rlDataViewer(fileLgr)` opens the **Reinforcement Learning Data Viewer** tool to visualize data logged using the `FileLogger` object `fileLgr`.

`rlDataViewer(folders)` opens the **Reinforcement Learning Data Viewer** tool to visualize data saved in the `folders` directories.

Examples

Visualize Data Logged In Custom Training Loop

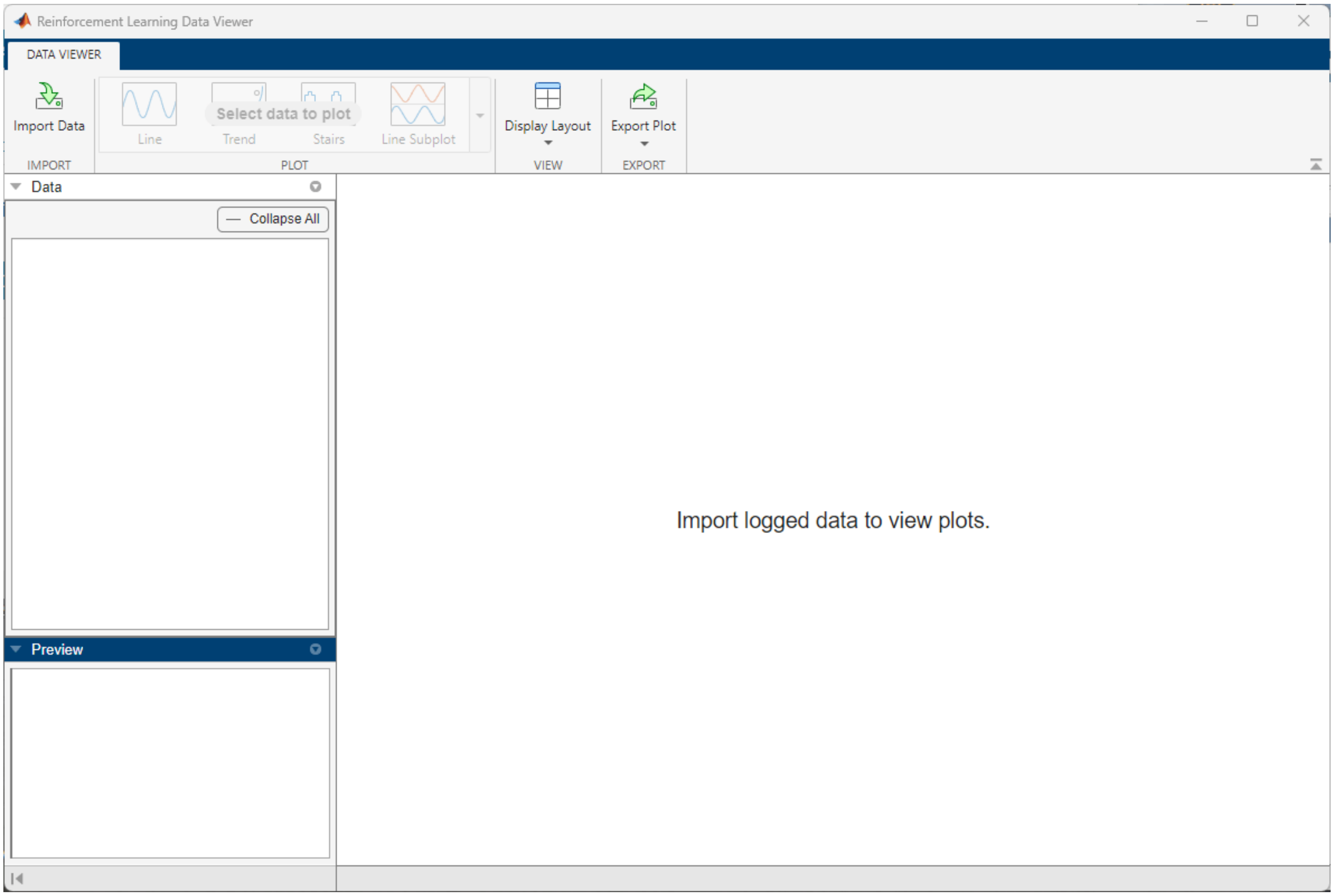
This example shows how to visualize data logged during custom training loop using the **Reinforcement Learning Data Viewer**.

Create a `FileLogger` object using `rlDataLogger`.

```
flgr = rlDataLogger();
```

Open a **Reinforcement Learning Data Viewer** window using `rlDataViewer`.

```
rlDataViewer(flgr);
```



Import logged data to view plots.

Initialize the logger object.

```
setup(flgr);
```

Within a custom training loop, store data to the logger object memory and write data to MAT-files. For this example, store a random scalar, a random 2-by-1 vector, and a random 5-by-5 matrix as variables named `RandomScalarData`, `RandomVectorData`, and `RandomMatrixData`, respectively.

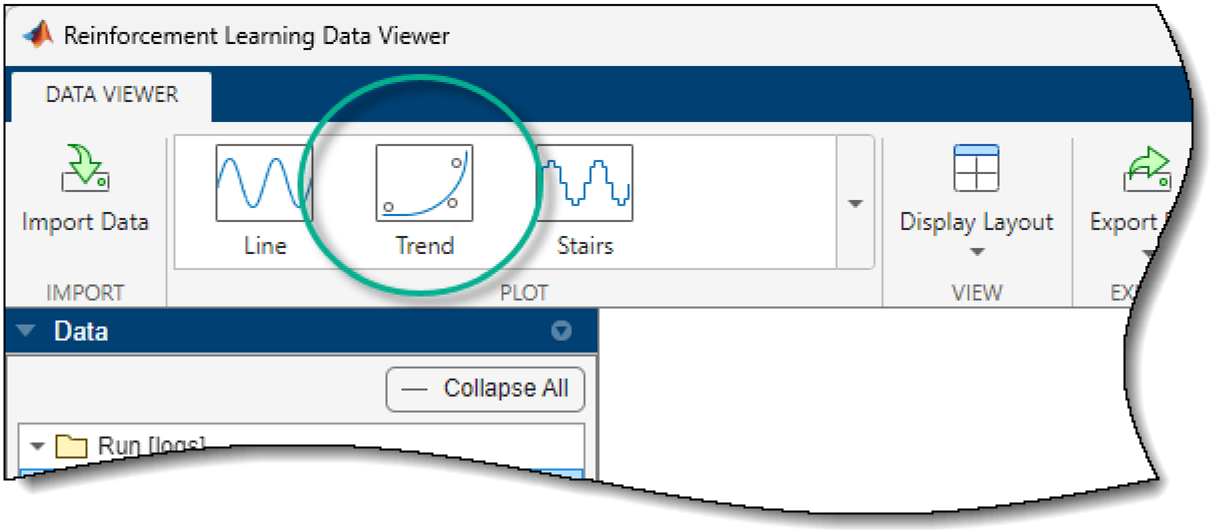
```
for i = 1:100

    % Store a random scalar, vector, and matrix in memory
    store(flgr, "RandomScalarData", rand, i);
    store(flgr, "RandomVectorData", rand(2,1), i);
    store(flgr, "RandomMatrixData", rand(5,5), i);

    % Write data from memory to a MAT-file
    write(flgr);

end
```

Clean up the logger object. This operation performs cleanup tasks like for example writing to file any data still in memory.

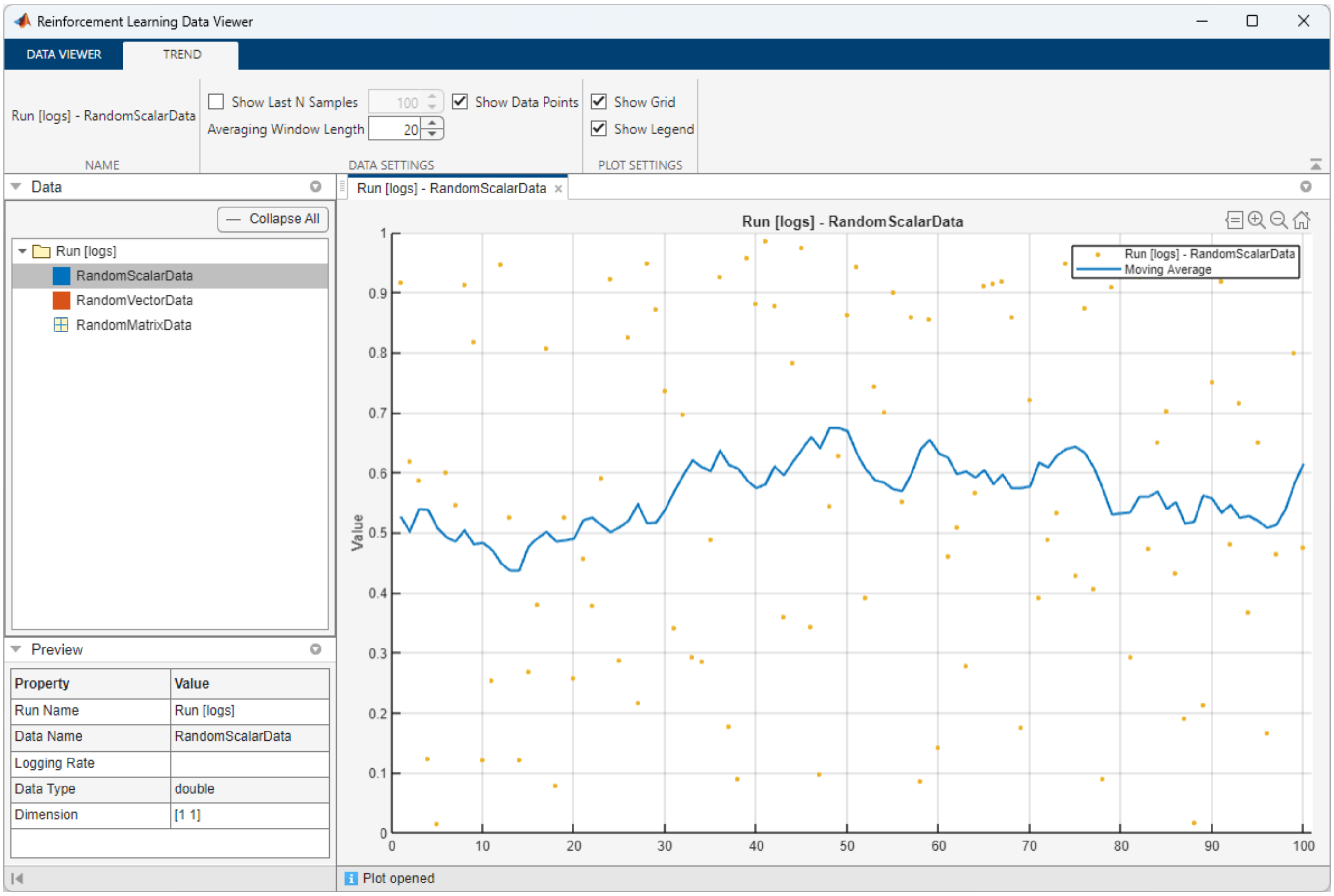
```
cleanup(flgr);
```

The **Data** pane of the **Reinforcement Learning Data Viewer** shows the three variables under the `Run [logs]` folder.

To select the first variable, click on `RandomScalarData`. The **Preview** pane then shows information such as the name, data type, and dimensions for this variable. To visualize a trend plot of the selected variable, on the **Data Viewer** tab, click **Trend**.



This action displays a moving average plot of the variable (using a 20-sample window), along with the data points.



☐ Show Last N Samples 100

☑ Show Data Points

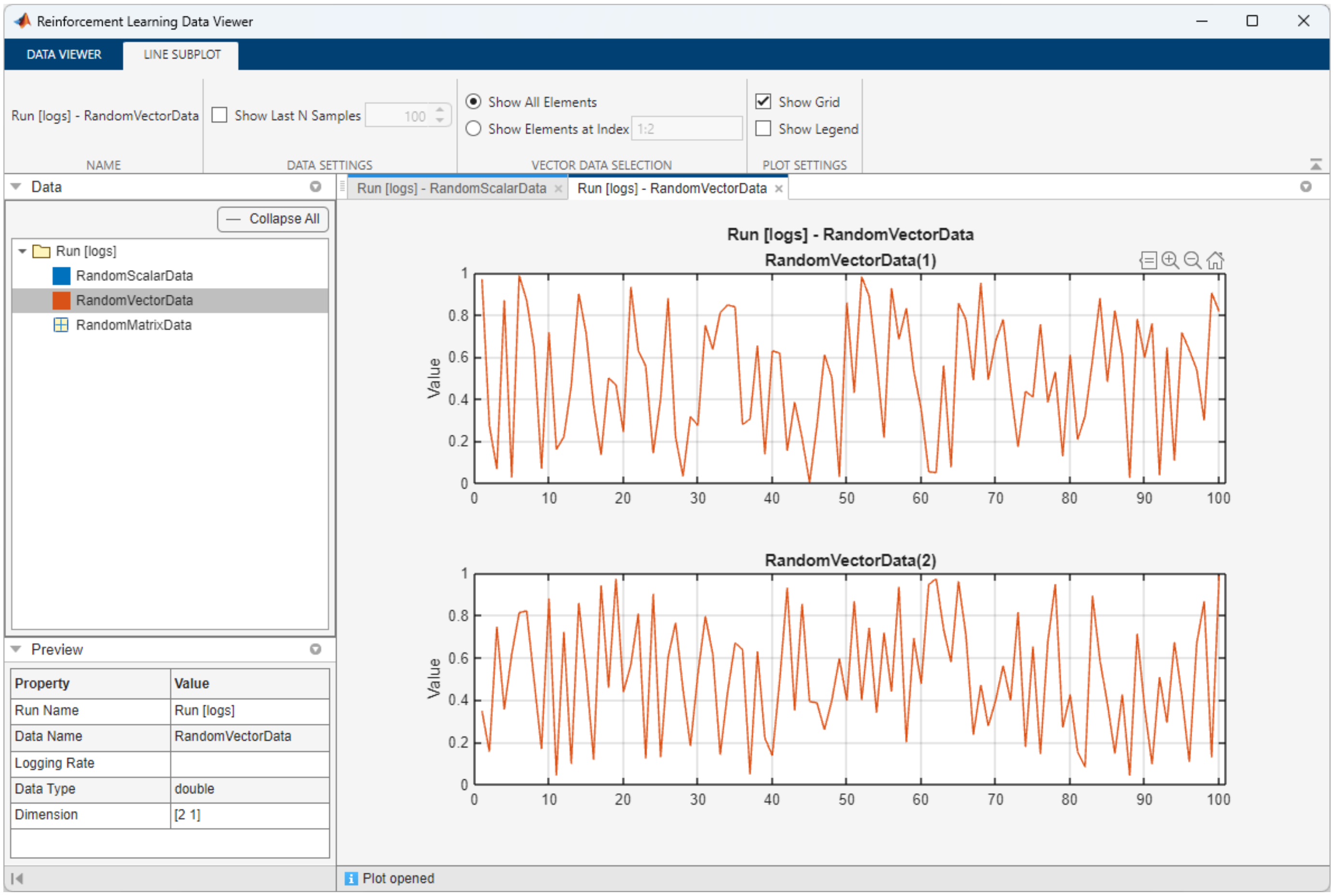
Averaging Window Length 20

☑ Show Grid

☑ Show Legend

Property	Value
Run Name	Run [logs]
Data Name	RandomScalarData
Logging Rate	
Data Type	double
Dimension	[1 1]

To plot both elements of the second variable, select RandomVectorData in the **Data** pane and, in the **Data Viewer** tab, click **Line Subplot**.



Reinforcement Learning Data Viewer

DATA VIEWER | LINE SUBPLOT

NAME: Run [logs] - RandomVectorData

DATA SETTINGS: □ Show Last N Samples 100

VECTOR DATA SELECTION: ● Show All Elements ○ Show Elements at Index 1:2

PLOT SETTINGS: ☑ Show Grid □ Show Legend

Data

— Collapse All

- Run [logs]
  - 🟦 RandomScalarData
  - 🟧 RandomVectorData
  - ⊞ RandomMatrixData

Preview

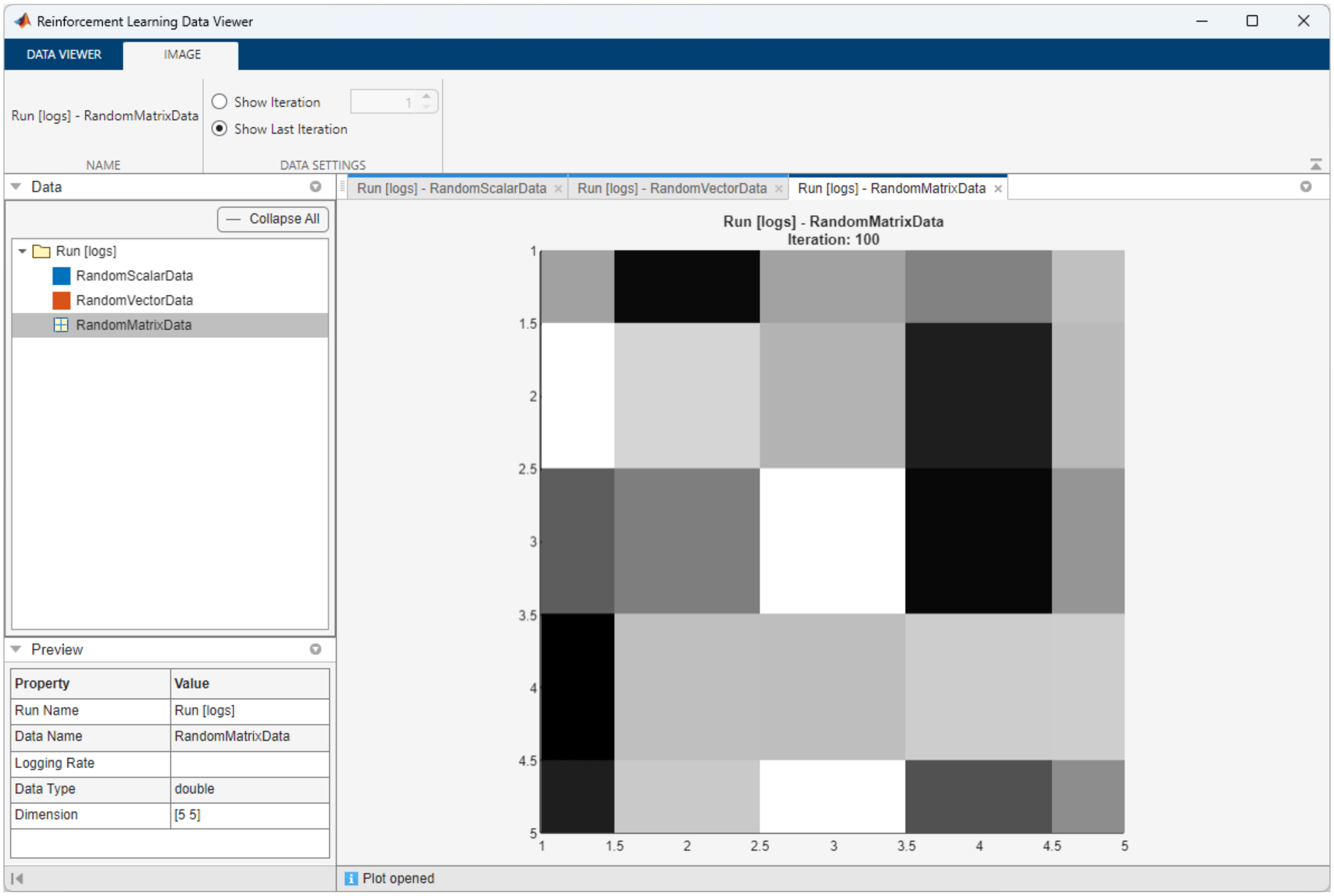
Property	Value
Run Name	Run [logs]
Data Name	RandomVectorData
Logging Rate	
Data Type	double
Dimension	[2 1]

Run [logs] - RandomScalarData × | Run [logs] - RandomVectorData ×

Plot opened

You can select multiple data histories (by holding the CTRL button while clicking their respective squares in the **Data** pane). When these data histories belong to the same run, the **Multiple Lines** button appears in the **Data Viewer** tab. This button allows you to visualize all the selected data in the same plot.

To show the value of `RandomMatrixData` in the last iteration as a heat map, select `RandomMatrixData` in the **Data** pane and, in the **Data Viewer** tab, click **Image**.



Run [logs] - RandomMatrixData

○ Show Iteration 1

● Show Last Iteration

Run [logs] - RandomMatrixData
Iteration: 100

Property	Value
Run Name	Run [logs]
Data Name	RandomMatrixData
Logging Rate	
Data Type	double
Dimension	[5 5]

Plot opened

This image is a heatmap of the matrix, with black corresponding to the maximum value and white the minimum.

Visualize Data Stored in Folder

This example shows how to use the **Reinforcement Learning Data Viewer** to visualize data logged on disk.

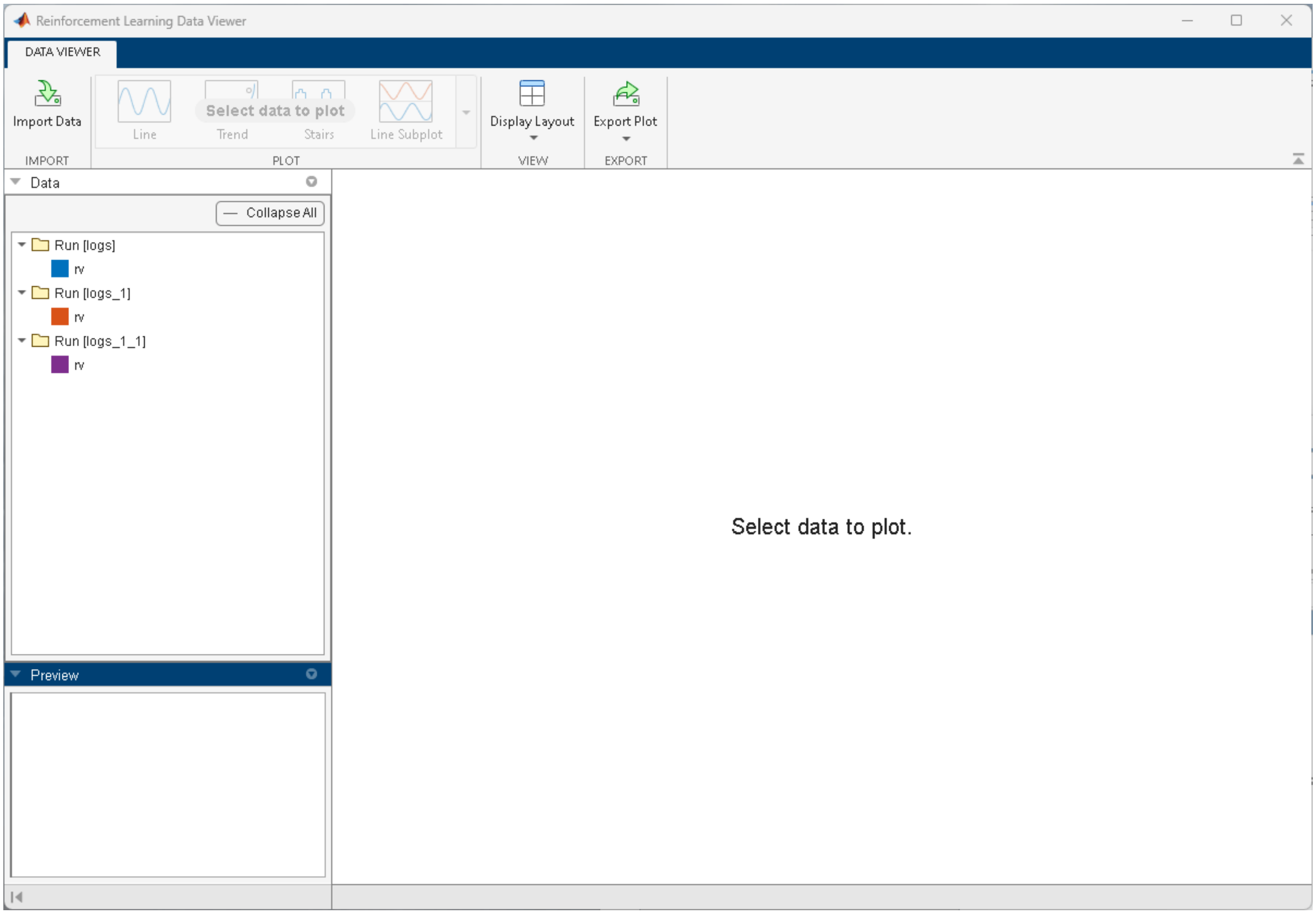
To save data on the disk, use a `FileLogger` object and use it in a custom loop for three times.

```
flgr = rlDataLogger();
for run=1:3

    % Set up logger object
    setup(flgr);

    % Within a custom loop, store a 3-by-1 vector in the
    % variable rv and write it to disk at each iteration.
    for i = 1:20
        store(flgr, "rv", rand(2,1), i);
        write(flgr);
    end
    % Clean up logger object.
    cleanup(flgr);
end
```

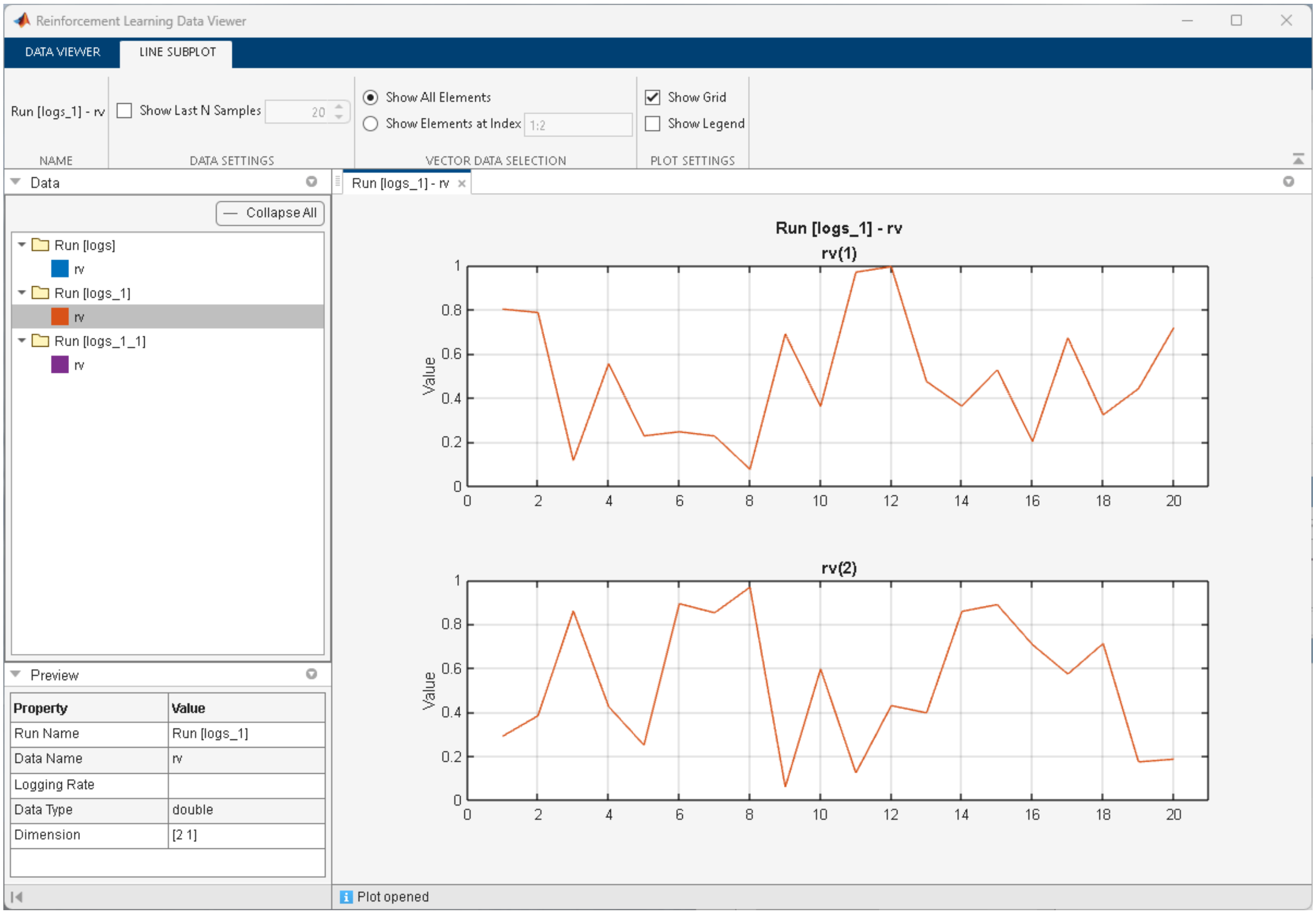
When `setup` is used on an existing `FileLogger` object, a `_1` suffix is added to the name of the destination folder. Therefore, the three loops result in data being saved in the folders `logs`, `logs_1`, and `logs_1_1`.

To visualize the saved data, open a **Reinforcement Learning Data Viewer** window using `rlDataViewer`.

```
rlDataViewer(["logs" "logs_1" "logs_1_1"]);
```



To plot the time history of `rv` in the second run, select the red `rv` square under the **Run [logs_1]** folder in the **Data** pane and, in the **Data Viewer** tab, click **Line Subplot**.



Run [logs_1] - rv

□ Show Last N Samples 20

● Show All Elements

○ Show Elements at Index 1:2

☑ Show Grid

□ Show Legend

Property	Value
Run Name	Run [logs_1]
Data Name	rv
Logging Rate	
Data Type	double
Dimension	[2 1]

Plot opened

You can select multiple data histories (by holding the CTRL button while clicking their respective squares in the **Data** pane). When these data histories belong to the same run, the **Multiple Lines** button appears in the **Data Viewer** tab. This button allows you to visualize all the selected data in the same plot.

Input Arguments

`fileLgr` — File logger object

`FileLogger` object

File logger object, specified as a `FileLogger` object.

`folders` — Names of the folders containing saved data

string | character array | string array | cell array of character vectors

Names, or fully qualified paths, of the folders containing saved data, specified as string array or cell array of character vectors. These are the names of the directories where the MAT-files containing the logged data are saved.

Example: `["C:\Temp\logs_1", "C:\Temp\logs_2"]`

Note When you pass a `FileLogger` object to the `train` function to log training data, the **Reinforcement Learning Episode Manager** displays an **Episode Manager** tab. To open the **Reinforcement Learning Data Viewer**, click **View Logged Data**. For an example, see “Log Training Data to Disk”.

Version History

Introduced in R2023a

See Also

Functions
`rlDataLogger` | `train`

Objects
`FileLogger` | `MonitorLogger` | `trainingProgressMonitor`

Topics
“Log Training Data to Disk”
“Monitor Custom Training Loop Progress”

rlOptimizer

Creates an optimizer object for actors and critics

Syntax

```
algobj = rlOptimizer
algobj = rlOptimizer(algOptions)
```

Description

Create an optimizer object that updates the learnable parameters of an actor or critic in a custom training loop

`algobj = rlOptimizer` returns a default optimizer object. You can modify the object properties using dot notation.

`algobj = rlOptimizer(algOptions)` returns an optimizer object with properties specified by the optimizer options object `algOptions`.

Examples

Create Optimizer Object

Use `rlOprimizer` to create a default optimizer algorithm object to use for the training of an actor or critic in a custom training loop.

```
myAlg = rlOptimizer
```

```
myAlg = 
  rlADAMOptimizer with properties:

           GradientDecayFactor: 0.9000
    SquaredGradientDecayFactor: 0.9990
                       Epsilon: 1.0000e-08
                     LearnRate: 0.0100
        L2RegularizationFactor: 1.0000e-04
             GradientThreshold: Inf
       GradientThresholdMethod: "l2norm"
```

By default, the function returns an `rlADAMOptimizer` object with default options. You can use dot notation to change some parameters.

```
myAlg.LearnRate = 0.1;
```

You can now create a structure and set its `CriticOptimizer` or `ActorOptimizer` field to `myAlg`. When you call `runEpisode`, pass the structure as an input parameter. The `runEpisode` function can then use the update method of `myAlg` to update the learnable parameters of your actor or critic.

Create Optimizer Object Specifying an Option Set

Use `rlOprimizerOptions` to create an optimizer option object. Specify the algorithm as `"rmsprop"` and set the learning rate to `0.2`.

```
myOptions=rlOptimizerOptions( ...
    Algorithm="rmsprop", ...
    LearnRate=0.2);
```

Use `rlOptimizer` to create an optimizer algorithm object to use for the training of an actor or critic in a custom training loop. Specify the optimizer option set `myOptions` as input parameter.

```
myAlg=rlOptimizer(myOptions)
```

```
myAlg = 
  rlRMSPropOptimizer with properties:

    SquaredGradientDecayFactor: 0.9990
                       Epsilon: 1.0000e-08
                     LearnRate: 0.2000
        L2RegularizationFactor: 1.0000e-04
             GradientThreshold: Inf
       GradientThresholdMethod: "l2norm"
```

The function returns an `rlRMSPropOptimizer` object with default options. You can use dot notation to change some parameters.

```
myAlg.GradientThreshold = 2;
```

You can now create a structure and set its `CriticOptimizer` or `ActorOptimizer` field to `myAlg`. When you call `runEpisode`, pass the structure as an input parameter. The `runEpisode` function can then use the update method of `myAlg` to update the learnable parameters of your actor or critic.

Input Arguments

`algOptions` — Algorithm options object

default Adam option set (default) | `rlOptimizerOptions` object

Algorithm options object, specified as an `rlOptimizerOptions` object.

Example: `rlOptimizerOptions(Algorithm="sgdm",LearnRate=0.2)`

Output Arguments

`algobj` — Algorithm optimizer object

`rlADAMOptimizer` object | `rlSGDMOptimizer` object | `rlRMSPropOptimizer` object

Algorithm optimizer object, returned as an `rlADAMOptimizer`, `rlSGDMOptimizer`, or `rlRMSPropOptimizer` object. The `runEpisode` function uses the update method of the returned object to update the learnable parameter of an actor or critic.

Version History

Introduced in R2022a

See Also

Functions

Objects

`rlOptimizerOptions`

Topics

"Create Policies and Value Functions"
"Reinforcement Learning Agents"

rlPredefinedEnv

Create a predefined reinforcement learning environment

Syntax

```
env = rlPredefinedEnv(keyword)
```

Description

`env = rlPredefinedEnv(keyword)` takes a predefined keyword `keyword` representing the environment name to create a MATLAB or Simulink reinforcement learning environment `env`. The environment `env` models the dynamics with which the agent interacts, generating rewards and observations in response to agent actions.

Examples

Basic Grid World Reinforcement Learning Environment

Use the predefined `"BasicGridWorld"` keyword to create a basic grid world reinforcement learning environment.

```
env = rlPredefinedEnv("BasicGridWorld")
```

```
env = 
  rlMDPEnv with properties:

       Model: [1x1 rl.env.GridWorld]
    ResetFcn: []
```

Continuous Double Integrator Reinforcement Learning Environment

Use the predefined `"DoubleIntegrator-Continuous"` keyword to create a continuous double integrator reinforcement learning environment.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous")
```

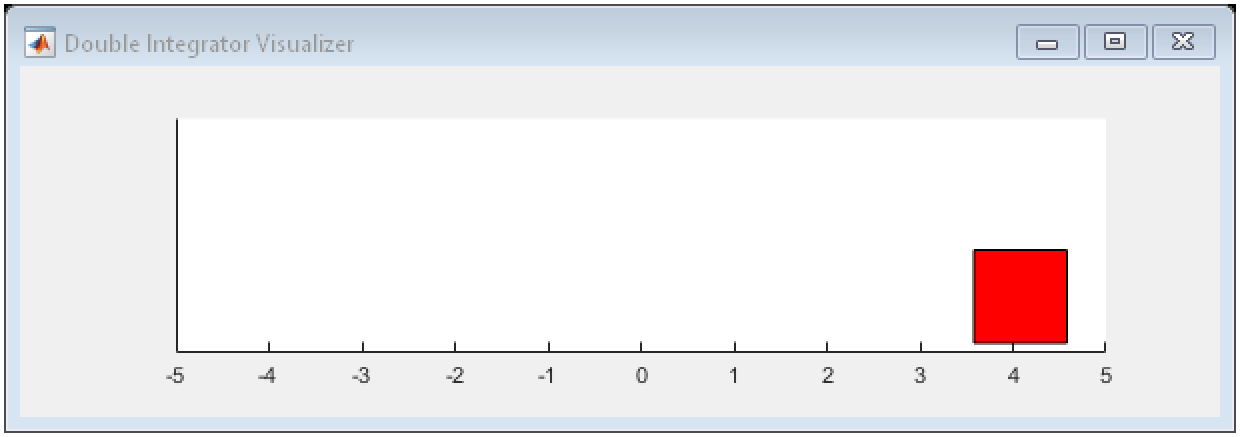
```
env = 
  DoubleIntegratorContinuousAction with properties:

             Gain: 1
               Ts: 0.1000
      MaxDistance: 5
    GoalThreshold: 0.0100
                Q: [2x2 double]
                R: 0.0100
         MaxForce: Inf
            State: [2x1 double]
```

You can visualize the environment using the `plot` function and interact with it using the `reset` and `step` functions.

```
plot(env)
observation = reset(env)
```

```
observation = 2×1

     4
     0
```

```
[observation,reward,isDone] = step(env,16)
```



```
observation = 2×1

    4.0800
    1.6000


reward = -16.5559

isDone = logical
   0
```

Create Continuous Simple Pendulum Model Environment

Use the predefined `"SimplePendulumModel-Continuous"` keyword to create a continuous simple pendulum model reinforcement learning environment.

```
env = rlPredefinedEnv("SimplePendulumModel-Continuous")
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

Input Arguments

keyword — Predefined keyword representing the environment name
'BasicGridWorld' | 'CartPole-Discrete' | 'DoubleIntegrator-Continuous' | 'SimplePendulumWithImage-Discrete' | 'SimplePendulumModel-Discrete' | 'SimplePendulumModel-Continuous' | 'CartPoleSimscapeModel-Continuous' | ...

Predefined keyword representing the environment name, specified as one of the following:

MATLAB Environment

- 'BasicGridWorld'
- 'CartPole-Discrete'
- 'CartPole-Continuous'
- 'DoubleIntegrator-Discrete'
- 'DoubleIntegrator-Continuous'
- 'SimplePendulumWithImage-Discrete'
- 'SimplePendulumWithImage-Continuous'
- 'WaterFallGridWorld-Stochastic'
- 'WaterFallGridWorld-Deterministic'

Simulink Environment

- 'SimplePendulumModel-Discrete'
- 'SimplePendulumModel-Continuous'
- 'CartPoleSimscapeModel-Discrete'
- 'CartPoleSimscapeModel-Continuous'

Output Arguments

env — MATLAB or Simulink environment object
rlMDPEnv object | CartPoleDiscreteAction object | CartPoleContinuousAction object | DoubleIntegratorDiscreteAction object | DoubleIntegratorContinuousAction object | SimplePendlumWithImageDiscreteAction object | SimplePendlumWithImageContinuousAction object | SimulinkEnvWithAgent object

MATLAB or Simulink environment object, returned as one of the following:

- rlMDPEnv object, when you use one of the following keywords.
  - 'BasicGridWorld'
  - 'WaterFallGridWorld-Stochastic'
  - 'WaterFallGridWorld-Deterministic'
- CartPoleDiscreteAction object, when you use the 'CartPole-Discrete' keyword.
- CartPoleContinuousAction object, when you use the 'CartPole-Continuous' keyword.
- DoubleIntegratorDiscreteAction object, when you use the 'DoubleIntegrator-Discrete' keyword.

- `DoubleIntegratorContinuousAction` object, when you use the `'DoubleIntegrator-Continuous'` keyword.
- `SimplePendlumWithImageDiscreteAction` object, when you use the `'SimplePendulumWithImage-Discrete'` keyword.
- `SimplePendlumWithImageContinuousAction` object, when you use the `'SimplePendulumWithImage-Continuous'` keyword.
- `SimulinkEnvWithAgent` object, when you use one of the following keywords.
  - `'SimplePendulumModel-Discrete'`
  - `'SimplePendulumModel-Continuous'`
  - `'CartPoleSimscapeModel-Discrete'`
  - `'CartPoleSimscapeModel-Continuous'`

Version History

Introduced in R2019a

See Also

Topics

"Create MATLAB Reinforcement Learning Environments"
"Create Simulink Reinforcement Learning Environments"
"Load Predefined Control System Environments"
"Load Predefined Simulink Environments"

rlRepresentation

(Not recommended) Model representation for reinforcement learning agents

Note `rlRepresentation` is not recommended. Use `rlValueRepresentation`, `rlQValueRepresentation`, `rlDeterministicActorRepresentation`, or `rlStochasticActorRepresentation` instead. For more information, see "Compatibility Considerations".

Syntax

```
rep = rlRepresentation(net,obsInfo,'Observation',obsNames)
rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action',
actNames)

tableCritic = rlRepresentation(tab)

critic = rlRepresentation(basisFcn,W0,obsInfo)
critic = rlRepresentation(basisFcn,W0,oaInfo)
actor = rlRepresentation(basisFcn,W0,obsInfo,actInfo)

rep = rlRepresentation( ___ ,repOpts)
```

Description

Use `rlRepresentation` to create a function approximator representation for the actor or critic of a reinforcement learning agent. To do so, you specify the observation and action signals for the training environment and options that affect the training of an agent that uses the representation. For more information on creating representations, see "Create Policies and Value Functions".

`rep = rlRepresentation(net,obsInfo,'Observation',obsNames)` creates a representation for the deep neural network `net`. The observation names `obsNames` are the network input layer names. `obsInfo` contains the corresponding observation specifications for the training environment. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `rlACAgent` or `rlPGAgent` agent.

`rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action', actNames)` creates a representation with action signals specified by the names `actNames` and specification `actInfo`. Use this syntax to create a representation for any actor, or for a critic that takes both observation and action as input, such as a critic for an `rlDQNAgent` or `rlDDPGAgent` agent.

`tableCritic = rlRepresentation(tab)` creates a critic representation for the value table or Q table `tab`. When you create a table representation, you specify the observation and action specifications when you create `tab`.

`critic = rlRepresentation(basisFcn,W0,obsInfo)` creates a linear basis function representation using the handle to a custom basis function `basisFcn` and initial weight vector `W0`. `obsInfo` contains the corresponding observation specifications for the training environment. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `rlACAgent` or `rlPGAgent` agent.

`critic = rlRepresentation(basisFcn,W0,oaInfo)` creates a linear basis function representation using the specification cell array `oaInfo`, where `oaInfo` = `{obsInfo,actInfo}`. Use this syntax to create a representation for a critic that takes both observations and actions as inputs, such as a critic for an `rlDQNAgent` or `rlDDPGAgent` agent.

`actor = rlRepresentation(basisFcn,W0,obsInfo,actInfo)` creates a linear basis function representation using the specified observation and action specifications, `obsInfo` and `actInfo`, respectively. Use this syntax to create a representation for an actor that takes observations as inputs and generates actions.

`rep = rlRepresentation( ___ ,repOpts)` creates a representation using additional options that specify learning parameters for the representation when you train an agent. Available options include the optimizer used for training and the learning rate. Use `rlRepresentationOptions` to create the options set `repOpts`. You can use this syntax with any of the previous input-argument combinations.

Examples

Create Actor and Critic Representations

Create an actor representation and a critic representation that you can use to define a reinforcement learning agent such as an Actor Critic (AC) agent.

For this example, create actor and critic representations for an agent that can be trained against the cart-pole environment described in “Train AC Agent to Balance Cart-Pole System”. First, create the environment. Then, extract the observation and action specifications from the environment. You need these specifications to define the agent and critic representations.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

For a state-value-function critic such as those used for AC or PG agents, the inputs are the observations and the output should be a scalar value, the state value. For this example, create the critic representation using a deep neural network with one output, and with observation signals corresponding to `x,xdot,theta,thetadot` as described in “Train AC Agent to Balance Cart-Pole System”. You can obtain the number of observations from the `obsInfo` specification. Name the network layer input `'observation'`.

```
numObservation = obsInfo.Dimension(1);
criticNetwork = [
    imageInputLayer([numObservation 1 1],'Normalization','none','Name','observation')
    fullyConnectedLayer(1,'Name','CriticFC')];
```

Specify options for the critic representation using `rlRepresentationOptions`. These options control parameters of critic network learning, when you train an agent that incorporates the critic representation. For this example, set the learning rate to 0.05 and the gradient threshold to 1.

```
repOpts = rlRepresentationOptions('LearnRate',5e-2,'GradientThreshold',1);
```

Create the critic representation using the specified neural network and options. Also, specify the action and observation information for the critic. Set the observation name to `'observation'`, which is the name you used when you created the network input layer for `criticNetwork`.

```
critic = rlRepresentation(criticNetwork,obsInfo,'Observation',{'observation'},repOpts)
```

```
critic = 
  rlValueRepresentation with properties:

            Options: [1x1 rl.option.rlRepresentationOptions]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: {1x0 cell}
```

Similarly, create a network for the actor. An AC agent decides which action to take given observations using an actor representation. For an actor, the inputs are the observations, and the output depends on whether the action space is discrete or continuous. For the actor of this example, there are two possible discrete actions, –10 or 10. Thus, to create the actor, use a deep neural network with the same observation input as the critic, that can output these two values. You can obtain the number of actions from the `actInfo` specification. Name the output `'action'`.

```
numAction = numel(actInfo.Elements);
actorNetwork = [
    imageInputLayer([4 1 1], 'Normalization','none','Name','observation')
    fullyConnectedLayer(numAction,'Name','action')];
```

Create the actor representation using the observation name and specification and the action name and specification. Use the same representation options.

```
actor = rlRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'action'},repOpts)
```

```
actor = 
  rlStochasticActorRepresentation with properties:

            Options: [1x1 rl.option.rlRepresentationOptions]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
```

You can now use the actor and critic representations to create an AC agent.

```
agentOpts = rlACAgentOptions(...
    'NumStepsToLookAhead',32,...
    'DiscountFactor',0.99);
agent = rlACAgent(actor,critic,agentOpts)
```

```
agent = 
  rlACAgent with properties:

    AgentOptions: [1x1 rl.option.rlACAgentOptions]
```

Create Q Table Representation

This example shows how to create a Q Table representation:

Create an environment interface.

```
env = rlPredefinedEnv("BasicGridWorld");
```

Create a Q table using the action and observation specifications from the environment.

```
qTable = rlTable(getObservationInfo(env),getActionInfo(env));
```

Create a representation for the Q table.

```
tableRep = rlRepresentation(qTable);
```

Create Quadratic Basis Function Critic Representation

This example shows how to create a linear basis function critic representation.

Assume that you have an environment, `env`. For this example, load the environment used in the "Train Custom LQR Agent" example.

```
load myLQREnv.mat
```

Obtain the observation and action specifications from the environment.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a custom basis function. In this case, use the quadratic basis function from "Train Custom LQR Agent".

Set the dimensions and parameters required for your basis function.

```
n = 6;
```

Set an initial weight vector.

```
w0 = 0.1*ones(0.5*(n+1)*n,1);
```

Create a representation using a handle to the custom basis function.

```
critic = rlRepresentation(@(x,u) computeQuadraticBasis(x,u,n),w0,{obsInfo,actInfo});
```

Function to compute the quadratic basis from "Train Custom LQR Agent".

```
function B = computeQuadraticBasis(x,u,n)
z = cat(1,x,u);
idx = 1;
for r = 1:n
    for c = r:n
        if idx == 1
            B = z(r)*z(c);
        else
            B = cat(1,B,z(r)*z(c));
        end
        idx = idx + 1;
    end
end
end
```

Input Arguments

net — Deep neural network for actor or critic

array of `Layer` objects | `layerGraph` object | `DAGNetwork` object | `SeriesNetwork` object

Deep neural network for actor or critic, specified as one of the following:

- Array of `Layer` objects
- `layerGraph` object
- `DAGNetwork` object
- `SeriesNetwork` object
- `dlnetwork` object

For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

`obsNames` — Observation names
cell array of character vectors

Observation names, specified as a cell array of character vectors. The observation names are the network input layer names you specify when you create `net`. The names in `obsNames` must be in the same order as the observation specifications in `obsInfo`.

Example: `{'observation'}`

`obsInfo` — Observation specification
`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

You can extract `obsInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually.

`actNames` — Action name
single-element cell array that contains a character vector

Action name, specified as a single-element cell array that contains a character vector. The action name is the network layer name you specify when you create `net`. For critic networks, this layer is the first layer of the action input path. For actors, this layer is the last layer of the action output path.

Example: `{'action'}`

`actInfo` — Action specification
`rlFiniteSetSpec` object | `rlNumericSpec` object

Action specifications, specified either as an `rlFiniteSetSpec` (for discrete action spaces) or `rlNumericSpec` (for continuous action spaces) object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

You can extract `actInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specifications manually.

`tab` — Value table or Q table for critic
`rlTable` object

Value table or Q table for critic, specified as an `rlTable` object. The learnable parameters of a table representation are the elements of `tab`.

`basisFcn` — Custom basis function

function handle

Custom basis function, specified as a function handle to a user-defined function. For a linear basis function representation, the output of the representation is `f = W'B`, where `W` is a weight array and `B` is the column vector returned by the custom basis function. The learnable parameters of a linear basis function representation are the elements of `W`.

When creating:

- A critic representation with observation inputs only, your basis function must have the following signature.

  ```
  B = myBasisFunction(obs1,obs2,...,obsN)
  ```

  Here `obs1` to `obsN` are observations in the same order and with the same data type and dimensions as the observation specifications in `obsInfo`.
- A critic representation with observation and action inputs, your basis function must have the following signature.

  ```
  B = myBasisFunction(obs1,obs2,...,obsN,act)
  ```

  Here `obs1` to `obsN` are observations in the same order and with the same data type and dimensions as the observation specifications in the first element of `oaInfo`, and `act` has the same data type and dimensions as the action specification in the second element of `oaInfo`.
- An actor representation, your basis function must have the following signature.

  ```
  B = myBasisFunction(obs1,obs2,...,obsN)
  ```

  Here, `obs1` to `obsN` are observations in the same order and with the same data type and dimensions as the observation specifications in `obsInfo`. The data types and dimensions of the action specification in `actInfo` affect the data type and dimensions of `f`.

Example: `@(x,u) myBasisFunction(x,u)`

`W0` — Initial value for linear basis function weight vector

column vector | array

Initial value for linear basis function weight array, `W`, specified as one of the following:

- Column vector — When creating a critic representation or an actor representation with a continuous scalar action signal
- Array — When creating an actor representation with a column vector continuous action signal or a discrete action space.

`oaInfo` — Observation and action specifications

cell array

Observation and action specifications for creating linear basis function critic representations, specified as the cell array `{obsInfo,actInfo}`.

repOpts — Representation options
rlRepresentationOptions object

Representation options, specified as an option set that you create with rlRepresentationOptions. Available options include the optimizer used for training and the learning rate. See rlRepresentationOptions for details.

Output Arguments

rep — Deep neural network representation
rlLayerRepresentation object

Deep neural network representation, returned as an rlLayerRepresentation object. Use this representation to create an agent for reinforcement learning. For more information, see "Reinforcement Learning Agents".

tableCritic — Value or Q table critic representation
rlTableRepresentation object

Value or Q table critic representation, returned as an rlTableRepresentation object. Use this representation to create an agent for reinforcement learning. For more information, see "Reinforcement Learning Agents".

critic — Linear basis function critic representation
rlLinearBasisRepresentation object

Linear basis function critic representation, returned as and rlLinearBasisRepresentation object. Use this representation to create an agent for reinforcement learning. For more information, see "Reinforcement Learning Agents".

actor — Linear basis function actor representation
rlLinearBasisRepresentation object

Linear basis function actor representation, returned as and rlLinearBasisRepresentation object. Use this representation to create an agent for reinforcement learning. For more information, see "Reinforcement Learning Agents".

Version History

Introduced in R2019a

R2020a: rlRepresentation is not recommended

Not recommended starting in R2020a

rlRepresentation is not recommended. Depending on the type of representation being created, use one of the following objects instead:

- rlValueRepresentation — State value critic, computed based on observations from the environment.
- rlQValueRepresentation — State-action value critic, computed based on both actions and observations from the environment.
- rlDeterministicActorRepresentation — Actor with deterministic actions, based on observations from the environment.

- `rlStochasticActorRepresentation` — Actor with stochastic actions, based on observations from the environment.

The following table shows some typical uses of the `rlRepresentation` function to create neural network-based critics and actors, and how to update your code with one of the new objects instead.

Network-Based Representations: Not Recommended	Network-Based Representations: Recommended
`rep = rlRepresentation(net,obsInfo,'Observation',obsName)`, with `net` having only observations as inputs, and a single scalar output.	`rep = rlValueRepresentation(net,obsInfo,'Observation',obsName)`. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `rlACAgent` or `rlPGAgent` agent.
`rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`, with `net` having both observations and action as inputs, and a single scalar output.	`rep = rlQValueRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`. Use this syntax to create a single-output state-action value representation for a critic that takes both observation and action as input, such as a critic for an `rlDQNAgent` or `rlDDPGAgent` agent.
`rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`, with `net` having observations as inputs and actions as outputs, and `actInfo` defining a continuous action space.	`rep = rlDeterministicActorRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`. Use this syntax to create a deterministic actor representation for a continuous action space.
`rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`, with `net` having observations as inputs and actions as outputs, and `actInfo` defining a discrete action space.	`rep = rlStochasticActorRepresentation(net,obsInfo,actInfo,'Observation',obsName)`. Use this syntax to create a stochastic actor representation for a discrete action space.

The following table shows some typical uses of the `rlRepresentation` objects to express table-based critics with discrete observation and action spaces, and how to update your code with one of the new objects instead.

Table-Based Representations: Not Recommended	Table-Based Representations: Recommended
`rep = rlRepresentation(tab)`, with `tab` containing a value table consisting in a column vector as long as the number of possible observations.	`rep = rlValueRepresentation(tab,obsInfo)`. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `rlACAgent` or `rlPGAgent` agent.

Table-Based Representations: Not Recommended	Table-Based Representations: Recommended
`rep = rlRepresentation(tab)`, with `tab` containing a Q-value table with as many rows as the possible observations and as many columns as the possible actions.	`rep = rlQValueRepresentation(tab,obsInfo,actInfo)`. Use this syntax to create a single-output state-action value representation for a critic that takes both observation and action as input, such as a critic for an `rlDQNAgent` or `rlDDPGAgent` agent.

The following table shows some typical uses of the `rlRepresentation` function to create critics and actors which use a custom basis function, and how to update your code with one of the new objects instead. In the recommended function calls, the first input argument is a two-elements cell containing both the handle to the custom basis function and the initial weight vector or matrix.

Custom Basis Function-Based Representations: Not Recommended	Custom Basis Function-Based Representations: Recommended
`rep = rlRepresentation(basisFcn,W0,obsInfo)`, where the basis function has only observations as inputs and `W0` is a column vector.	`rep = rlValueRepresentation({basisFcn,W0},obsInfo)`. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `rlACAgent` or `rlPGAgent` agent.
`rep = rlRepresentation(basisFcn,W0,{obsInfo,actInfo})`, where the basis function has both observations and action as inputs and `W0` is a column vector.	`rep = rlQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a single-output state-action value representation for a critic that takes both observation and action as input, such as a critic for an `rlDQNAgent` or `rlDDPGAgent` agent.
`rep = rlRepresentation(basisFcn,W0,obsInfo,actInfo)`, where the basis function has observations as inputs and actions as outputs, `W0` is a matrix, and `actInfo` defines a continuous action space.	`rep = rlDeterministicActorRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a deterministic actor representation for a continuous action space.
`rep = rlRepresentation(basisFcn,W0,obsInfo,actInfo)`, where the basis function has observations as inputs and actions as outputs, `W0` is a matrix, and `actInfo` defines a discrete action space.	`rep = rlStochasticActorRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a deterministic actor representation for a discrete action space.

See Also

Functions
`getActionInfo` | `getObservationInfo`

Objects
`rlValueRepresentation` | `rlQValueRepresentation` | `rlDeterministicActorRepresentation` | `rlStochasticActorRepresentation` | `rlRepresentationOptions`

Topics
"Create Policies and Value Functions"
"Reinforcement Learning Agents"

rlSimulinkEnv

Create reinforcement learning environment using dynamic model implemented in Simulink

Syntax

```
env = rlSimulinkEnv(mdl,agentBlocks)
env = rlSimulinkEnv(mdl,agentBlocks,obsInfo,actInfo)
env = rlSimulinkEnv( ___ ,'UseFastRestart',fastRestartToggle)
```

Description

The rlSimulinkEnv function creates a reinforcement learning environment object from a Simulink model. The environment object acts an interface so that when you call sim or train, these functions in turn call the Simulink model to generate experiences for the agents.

env = rlSimulinkEnv(mdl,agentBlocks) creates the reinforcement learning environment object env for the Simulink model mdl. agentBlocks contains the paths to one or more reinforcement learning agent blocks in mdl. If you use this syntax, each agent block must reference an agent object already in the MATLAB workspace.

env = rlSimulinkEnv(mdl,agentBlocks,obsInfo,actInfo) creates the reinforcement learning environment object env for the model mdl. The two cell arrays obsInfo and actInfo must contain the observation and action specifications for each agent block in mdl, in the same order as they appear in agentBlocks.

env = rlSimulinkEnv(___ ,'UseFastRestart',fastRestartToggle) creates a reinforcement learning environment object env and additionally enables fast restart. Use this syntax after any of the input arguments in the previous syntaxes.

Examples

Create Simulink Environment Using Agent in Workspace

Create a Simulink environment using the trained agent and corresponding Simulink model from the “Create Simulink Environment and Train Agent” example.

Load the agent in the MATLAB® workspace.

```
load rlWaterTankDDPGAgent
```

Create an environment for the rlwatertank model, which contains an RL Agent block. Since the agent used by the block is already in the workspace, you do not need to pass the observation and action specifications to create the environment.

```
env = rlSimulinkEnv("rlwatertank","rlwatertank/RL Agent")
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlwatertank
```

```
      AgentBlock : rlwatertank/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

Validate the environment by performing a short simulation for two sample times.

```
validateEnvironment(env)
```

You can now train and simulate the agent within the environment by using `train` and `sim`, respectively.

Create Reinforcement Learning Environment for Simulink Model

For this example, consider the `rlSimplePendulumModel` Simulink model. The model is a simple frictionless pendulum that initially hangs in a downward position.

Open the model.

```
mdl = "rlSimplePendulumModel";
open_system(mdl)
```

Create `rlNumericSpec` and `rlFiniteSetSpec` objects for the observation and action information, respectively.

The observation is a vector containing three signals: the sine, cosine, and time derivative of the angle.

```
obsInfo = rlNumericSpec([3 1])
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [3 1]
       DataType: "double"
```

The action is a scalar expressing the torque and can be one of three possible values, -2 Nm, `0` Nm and 2 Nm.

```
actInfo = rlFiniteSetSpec([-2 0 2])
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [3x1 double]
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

You can use dot notation to assign property values for the rlNumericSpec and rlFiniteSetSpec objects.

```
obsInfo.Name = "observations";
actInfo.Name = "torque";
```

Assign the agent block path information, and create the reinforcement learning environment for the Simulink model using the information extracted in the previous steps.

```
agentBlk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,agentBlk,obsInfo,actInfo)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

You can also include a reset function using dot notation. For this example, randomly initialize theta0 in the model workspace.

```
env.ResetFcn = @(in) setVariable(in,"theta0",randn,"Workspace",mdl)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : @(in)setVariable(in,"theta0",randn,"Workspace",mdl)
  UseFastRestart : on
```

Create Simulink Environment for Multiple Agents

Create an environment for the Simulink model from the example “Train Multiple Agents to Perform Collaborative Task”.

Load the agents in the MATLAB workspace.

```
load rlCollaborativeTaskAgents
```

Create an environment for the rlCollaborativeTask model, which has two agent blocks. Since the agents used by the two blocks (agentA and agentB) are already in the workspace, you do not need to pass their observation and action specifications to create the environment.

```
env = rlSimulinkEnv( ...
    "rlCollaborativeTask", ...
    ["rlCollaborativeTask/Agent A","rlCollaborativeTask/Agent B"])
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlCollaborativeTask
```

```
      AgentBlock : [
                      rlCollaborativeTask/Agent A
                      rlCollaborativeTask/Agent B
                    ]
        ResetFcn : []
  UseFastRestart : on
```

You can now simulate or train the agents within the environment using `sim` or `train`, respectively.

Input Arguments

`mdl` — Simulink model name
string | character vector

Simulink model name, specified as a string or character vector. The model must contain at least one RL Agent block.

`agentBlocks` — Agent block paths
string | character vector | string array

Agent block paths, specified as a string, character vector, or string array.

If `mdl` contains a single RL Agent block, specify `agentBlocks` as a string or character vector containing the block path.

If `mdl` contains multiple RL Agent blocks, specify `agentBlocks` as a string array, where each element contains the path of one agent block.

`mdl` can contain RL Agent blocks whose path is not included in `agentBlocks`. Such agent blocks behave as part of the environment, selecting actions based on their current policies. When you call `sim` or `train`, the experiences of these agents are not returned and their policies are not updated.

Multi-agent simulation is not supported for MATLAB environments.

The agent blocks can be inside of a model reference. For more information on configuring an agent block for reinforcement learning, see RL Agent.

`obsInfo` — Observation information
`rlNumericSpec` object | `rlFiniteSetSpec` object | array of `rlNumericSpec` objects | array of `rlFiniteSetSpec` objects | cell array

Observation information, specified as a specification object, an array of specification objects, or a cell array.

If `mdl` contains a single agent block, specify `obsInfo` as an `rlNumericSpec` object, an `rlFiniteSetSpec` object, or an array containing a mix of such objects.

If `mdl` contains multiple agent blocks, specify `obsInfo` as a cell array, where each cell contains a specification object or array of specification objects for the corresponding block in `agentBlocks`.

For more information, see `getObservationInfo`.

`actInfo` — Action information
`rlNumericSpec` object | `rlFiniteSetSpec` object | cell array

Action information, specified as a specification object or a cell array.

If `mdl` contains a single agent block, specify `actInfo` as an `rlNumericSpec` or `rlFiniteSetSpec` object.

If `mdl` contains multiple agent blocks, specify `actInfo` as a cell array, where each cell contains a specification object for the corresponding block in `agentBlocks`.

For more information, see `getActionInfo`.

`fastRestartToggle` — Option to toggle fast restart
`'on'` (default) | `'off'`

Option to toggle fast restart, specified as either `'on'` or `'off'`. Fast restart allows you to perform iterative simulations without compiling a model or terminating the simulation each time.

For more information on fast restart, see “How Fast Restart Improves Iterative Simulations” (Simulink).

Output Arguments

env — Reinforcement learning environment
`SimulinkEnvWithAgent` object

Reinforcement learning environment, returned as a `SimulinkEnvWithAgent` object.

For more information on reinforcement learning environments, see “Create Simulink Reinforcement Learning Environments”.

Version History

Introduced in R2019a

See Also

Functions
`train` | `sim` | `getObservationInfo` | `getActionInfo`

Objects
`rlNumericSpec` | `rlFiniteSetSpec`

Blocks
RL Agent

Topics
“Train DDPG Agent to Control Double Integrator System”
“Train DDPG Agent to Swing Up and Balance Pendulum”
“Train DDPG Agent to Swing Up and Balance Cart-Pole System”
“Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal”
“Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation”
“Train DDPG Agent for Adaptive Cruise Control”
“How Fast Restart Improves Iterative Simulations” (Simulink)

runEpisode

Package: rl.env

Simulate reinforcement learning environment against policy or agent

Syntax

```
output = runEpisode(env,policy)
```

```
output = runEpisode(env,agent)
```

```
output = runEpisode( ___ ,Name=Value)
```

Description

output = runEpisode(env,policy) runs a single simulation of the environment env against the policy policy.

output = runEpisode(env,agent) runs a single simulation of the environment env against the agent agent. During the simulation, the policy of the agent is evaluated to produce actions but learnable parameters are not updated.

output = runEpisode(___ ,Name=Value) specifies nondefault simulation options using one or more name-value arguments.

Examples

Simulate Environment and Agent

Create a reinforcement learning environment and extract its observation and action specifications.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

To approximate the Q-value function withing the critic, use a neural network. Create a network as an array of layer objects.

```
net = [...
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(2)
    softmaxLayer];
```

Convert the network to a dlnetwork object and display the number of learnable parameters (weights).

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 770

   Inputs:
      1   'input'   4 features
```

Create a discrete categorical actor using the network.

```
actor = rlDiscreteCategoricalActor(net,obsInfo,actInfo);
```

Check your actor with a random observation.

```
act = getAction(actor,{rand(obsInfo.Dimension)})
```

```
act = 1x1 cell array
    {[-10]}
```

Create a policy object from the actor.

```
policy = rlStochasticActorPolicy(actor);
```

Create an experience buffer.

```
buffer = rlReplayMemory(obsInfo,actInfo);
```

Set up the environment for running multiple simulations. For this example, configure the training to log any errors rather than send them to the command window.

```
setup(env,StopOnError="off")
```

Simulate multiple episodes using the environment and policy. After each episode, append the experiences to the buffer. For this example, run 100 episodes.

```
for i = 1:100
    output = runEpisode(env,policy,MaxSteps=300);
    append(buffer,output.AgentData.Experiences)
end
```

Clean up the environment.

```
cleanup(env)
```

Sample a mini-batch of experiences from the buffer. For this example, sample 10 experiences.

```
batch = sample(buffer,10);
```

You can then learn from the sampled experiences and update the policy and actor.

Input Arguments

env — Reinforcement learning environment
environment object | ...

Reinforcement learning environment, specified as one of the following objects.

- rlFunctionEnv — Environment defined using custom functions
- SimulinkEnvWithAgent — Simulink environment created using rlSimulinkEnv or createIntegratedEnv
- rlMDPEnv — Markov decision process environment
- rlNeuralNetworkEnvironment — Environment with deep neural network transition models
- Predefined environment created using rlPredefinedEnv
- Custom environment created from a template (rlCreateEnvTemplate)

policy — Policy
policy object | array of policy objects

Policy object, specified as one of the following objects.

- rlDeterministicActorPolicy
- rlAdditiveNoisePolicy
- rlEpsilonGreedyPolicy
- rlMaxQPolicy
- rlStochasticActorPolicy

If env is a Simulink environment configured for multi-agent training, specify policy as an array of policy objects. The order of the policies in the array must match the agent order used to create env.

For more information on a policy object, at the MATLAB command line, type help followed by the policy object name.

agent — Reinforcement learning agent
agent object | array of agent objects

Reinforcement learning agent, specified as one of the following objects.

- rlQAgent
- rlSARSAAgent
- rlDQNAgent
- rlPGAgent
- rlDDPGAgent
- rlTD3Agent
- rlACAgent
- rlSACAgent
- rlPPOAgent
- rlTRPOAgent
- rlMBPOAgent
- Custom agent — For more information, see “Create Custom Reinforcement Learning Agents”.

If env is a Simulink environment configured for multi-agent training, specify agent as an array of agent objects. The order of the agents in the array must match the agent order used to create env.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: MaxSteps=1000

MaxSteps — Maximum simulation steps
500 (default) | positive integer

Maximum simulation steps, specified as a positive integer.

ProcessExperienceFcn — Function for processing experiences
function handle | cell array of function handles

Function for processing experiences and updating the policy or agent based on each experience as it occurs during the simulation, specified as a function handle with the following signature.

```
[updatedPolicy,updatedData] = myFcn(experience,episodeInfo,policy,data)
```

Here:

- experience is a structure that contains a single experience. For more information on the structure fields, see output.Experiences.
- episodeInfo contains data about the current episode and corresponds to output.EpisodeInfo.
- policy is the policy or agent object being simulated.
- data contains experience processing data. For more information, see ProcessExperienceData.
- updatedPolicy is the updated policy or agent.
- updatedData is the updated experience processing data, which is used as the data input when processing the next experience.

If env is a Simulink environment configured for multi-agent training, specify ProcessExperienceFcn as a cell array of function handles. The order of the function handles in the array must match the agent order used to create env.

ProcessExperienceData — Experience processing data
any MATLAB data type | cell array

Experience processing data, specified as any MATLAB data, such as an array or structure. Use this data to pass additional parameters or information to the experience processing function.

You can also update this data within the experience processing function to use different parameters when processing the next experience. The data values that you specify when you call runEpisode are used to process the first experience in the simulation.

If env is a Simulink environment configured for multi-agent training, specify ProcessExperienceData as a cell array. The order of the array elements must match the agent order used to create env.

CleanupPostSim — Option to clean up environment
true (default) | false

Option to clean up the environment after the simulation, specified as true or false. When CleanupPostSim is true, runEpisode calls cleanup(env) when the simulation ends.

To run multiple episodes without cleaning up the environment, set CleanupPostSim to false. You can then call cleanup(env) after running your simulations.

If env is a SimulinkEnvWithAgent object and the associated Simulink model is configured to use fast restart, then the model remains in a compiled state between simulations when CleanUpPostSim is false.

LogExperiences — Option to log experiences

true (default) | false

Option to log experiences for each policy or agent, specified as true or false. When LogExperiences is true, the experiences of the policy or agent are logged in output.Experiences.

Output Arguments

output — Simulation output

structure | Future object

Simulation output, returned as a structure with the fields AgentData and SimulationInfo.

The AgentData field is a structure array containing data for each agent or policy. Each AgentData structure has the following fields.

Field	Description
Experiences	Logged experience of the policy or agent, returned as a structure array. Each experience contains the following fields. • Observation — Observation • Action — Action taken • NextObservation — Resulting next observation • Reward — Corresponding reward • IsDone — Termination signal
Time	Simulation times of experiences, returned as a vector.
EpisodeInfo	Episode information, returned as a structure with the following fields. • CumulativeReward — Total reward for all experiences • StepsTaken — Number of simulation steps taken • InitialObservation — Initial observation at the start of the simulation
ProcessExperienceData	Experience processing data
Agent	Policy or agent used in the simulation

The SimulationInfo field is one of the following:

- For MATLAB environments — Structure containing the field `SimulationError`. This structure contains any errors that occurred during simulation.
- For Simulink environments — `Simulink.SimulationOutput` object containing simulation data. Recorded data includes any signals and states that the model is configured to log, simulation metadata, and any errors that occurred.

If `env` is configured to run simulations on parallel workers, then `output` is a `Future` object, which supports deferred outputs for environment simulations that run on workers.

Tips

- You can speed up episode simulation by using parallel computing. To do so, use the `setup` function and set the `UseParallel` argument to `true`.

  ```
  setup(env,UseParallel=true)
  ```

Version History

Introduced in R2022a

See Also

Objects
`Future`

Functions
`fetchNext` | `fetchOutputs` | `cancel` | `wait` | `setup` | `cleanup` | `reset`

Topics
“Custom Training Loop with Simulink Action Noise”
“Train Reinforcement Learning Policy Using Custom Training Loop”
“Create MATLAB Reinforcement Learning Environments”
“Create Simulink Reinforcement Learning Environments”
“Train Agents Using Parallel Computing and GPUs”

sample

Package: `rl.replay`

Sample experiences from replay memory buffer

Syntax

```
experience = sample(buffer,batchSize)
experience = sample(buffer,batchSize,Name=Value)
[experience,Mask] = sample(buffer,batchSize,Name=Value)
```

Description

`experience = sample(buffer,batchSize)` returns a mini-batch of *N* experiences from the replay memory `buffer`, where *N* is specified using `batchSize`.

`experience = sample(buffer,batchSize,Name=Value)` specifies additional sampling options using one or more name-value pair arguments.

`[experience,Mask] = sample(buffer,batchSize,Name=Value)` returns a sequence padding mask indicating which the padded experiences at the end of a sampled sequence.

Examples

Create Experience Buffer

Define observation specifications for the environment. For this example, assume that the environment has a single observation channel with three continuous signals in specified ranges.

```
obsInfo = rlNumericSpec([3 1],...
    LowerLimit=0,...
    UpperLimit=[1;5;10]);
```

Define action specifications for the environment. For this example, assume that the environment has a single action channel with two continuous signals in specified ranges.

```
actInfo = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[5;10]);
```

Create an experience buffer with a maximum length of 20,000.

```
buffer = rlReplayMemory(obsInfo,actInfo,20000);
```

Append a single experience to the buffer using a structure. Each experience contains the following elements: current observation, action, next observation, reward, and is-done.

For this example, create an experience with random observation, action, and reward values. Indicate that this experience is not a terminal condition by setting the `IsDone` value to 0.

```
exp.Observation = {obsInfo.UpperLimit.*rand(3,1)};
exp.Action = {actInfo.UpperLimit.*rand(2,1)};
exp.Reward = 10*rand(1);
exp.NextObservation = {obsInfo.UpperLimit.*rand(3,1)};
exp.IsDone = 0;
```

Before appending experience to the buffer, you can validate whether the experience is compatible with the buffer. The `validateExperience` function generates an error if the expereince is incompatible with the buffer.

```
validateExperience(buffer,exp)
```

Append the experience to the buffer.

```
append(buffer,exp);
```

You can also append a batch of experiences to the experience buffer using a structure array. For this example, append a sequence of 100 random experiences, with the final experience representing a terminal condition.

```
for i = 1:100
    expBatch(i).Observation = {obsInfo.UpperLimit.*rand(3,1)};
    expBatch(i).Action = {actInfo.UpperLimit.*rand(2,1)};
    expBatch(i).Reward = 10*rand(1);
    expBatch(i).NextObservation = {obsInfo.UpperLimit.*rand(3,1)};
    expBatch(i).IsDone = 0;
end
expBatch(100).IsDone = 1;

validateExperience(buffer,expBatch)

append(buffer,expBatch);
```

After appending experiences to the buffer, you can sample mini-batches of experiences for training of your RL agent. For example, randomly sample a batch of 50 experiences from the buffer.

```
miniBatch = sample(buffer,50);
```

You can sample a horizon of data from the buffer. For example, sample a horizon of 10 consecutive experiences with a discount factor of 0.95.

```
horizonSample = sample(buffer,1,...
    NStepHorizon=10,...
    DiscountFactor=0.95);
```

The returned sample includes the following information.

- `Observation` and `Action` are the observation and action from the first experience in the horizon.
- `NextObservation` and `IsDone` are the next observation and termination signal from the final experience in the horizon.
- `Reward` is the cumulative reward across the horizon using the specified discount factor.

You can also sample a sequence of consecutive experiences. In this case, the structure fields contain arrays with values for all sampled experiences.

```
sequenceSample = sample(buffer,1,...
    SequenceLength=20);
```

Create Experience Buffer with Multiple Observation Channels

Define observation specifications for the environment. For this example, assume that the environment has two observation channels: one channel with two continuous observations and one channel with a three-valued discrete observation.

```
obsContinuous = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[1;5]);
obsDiscrete = rlFiniteSetSpec([1 2 3]);
obsInfo = [obsContinuous obsDiscrete];
```

Define action specifications for the environment. For this example, assume that the environment has a single action channel with one continuous action in a specified range.

```
actInfo = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[5;10]);
```

Create an experience buffer with a maximum length of 5,000.

```
buffer = rlReplayMemory(obsInfo,actInfo,5000);
```

Append a sequence of 50 random experiences to the buffer.

```
for i = 1:50
    exp(i).Observation = ...
        {obsInfo(1).UpperLimit.*rand(2,1) randi(3)};
    exp(i).Action = {actInfo.UpperLimit.*rand(2,1)};
    exp(i).NextObservation = ...
        {obsInfo(1).UpperLimit.*rand(2,1) randi(3)};
    exp(i).Reward = 10*rand(1);
    exp(i).IsDone = 0;
end

append(buffer,exp);
```

After appending experiences to the buffer, you can sample mini-batches of experiences for training of your RL agent. For example, randomly sample a batch of 10 experiences from the buffer.

```
miniBatch = sample(buffer,10);
```

Input Arguments

`buffer` — Experience buffer

`rlReplayMemory` object | `rlPrioritizedReplayMemory` object | `rlHindsightReplayMemory` object | `rlHindsightPrioritizedReplayMemory` object

Experience buffer, specified as one of the following replay memory objects.

- rlReplayMemory
- rlPrioritizedReplayMemory
- rlHindsightReplayMemory
- rlHindsightPrioritizedReplayMemory

batchSize — Batch size

positive integer

Batch size of experiences to sample, specified as a positive integer.

If batchSize is greater than the current length of the buffer, then sample returns no experiences.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: DiscountFactor=0.95

SequenceLength — Sequence length

1 (default) | positive integer

Sequence length, specified as a positive integer. For each batch element, sample up to SequenceLength consecutive experiences. If a sampled experience has a nonzero IsDone value, stop the sequence at that experience.

NStepHorizon — N-step horizon length

1 (default) | positive integer

N-step horizon length, specified as a positive integer. For each batch element, sample up to NStepHorizon consecutive experiences. If a sampled experience has a nonzero IsDone value, stop the horizon at that experience. Return the following experience information based on the sampled horizon.

- Observation and Action values from the first experience in the horizon
- NextObservation and IsDone values from the final experience in the horizon.
- Cumulative reward across the horizon using the specified discount factor, DiscountFactor.

Sampling an n-step horizon is not supported when sampling sequences. Therefore, if SequenceLength > 1, then NStepHorizon must be 1.

DiscountFactor — Discount factor

0.99 (default) | nonnegative scalar less than or equal to one

Discount factor, specified as a nonnegative scalar less than or equal to one. When you sample a horizon of experiences (NStepHorizon > 1), sample returns the cumulative reward R computed as follows.

$$R = \sum_{i=1}^{N} \gamma^i R_i$$

Here:

- γ is the discount factor.
- N is the sampled horizon length, which can be less than `NStepHorizon`.
- R_i is the reward for the ith horizon step.

`DiscountFactor` applies only when `NStepHorizon` is greater than one.

`DataSourceID` — Data source index

`-1` (default) | nonnegative integer

Data source index, specified as one of the following:

- `-1` — Sample from the experiences of all data sources.
- Nonnegative integer — Sample from the experiences of only the data source specified by `DataSourceID`.

Output Arguments

`experience` — Experiences sampled from the buffer

structure

Experiences sampled from the buffer, returned as a structure with the following fields.

`Observation` — Observation

cell array

Observation, returned as a cell array with length equal to the number of observation specifications specified when creating the buffer. Each element of `Observation` contains a D_O-by-`batchSize`-by-`SequenceLength` array, where D_O is the dimension of the corresponding observation specification.

`Action` — Agent action

cell array

Agent action, returned as a cell array with length equal to the number of action specifications specified when creating the buffer. Each element of `Action` contains a D_A-by-`batchSize`-by-`SequenceLength` array, where D_A is the dimension of the corresponding action specification.

`Reward` — Reward value

scalar | array

Reward value obtained by taking the specified action from the observation, returned as a 1-by-1-by-`SequenceLength` array.

`NextObservation` — Next observation

cell array

Next observation reached by taking the specified action from the observation, returned as a cell array with the same format as `Observation`.

`IsDone` — Termination signal

integer | array

Termination signal, returned as a 1-by-1-by-`SequenceLength` array of integers. Each element of `IsDone` has one of the following values.

- `0` — This experience is not the end of an episode.
- `1` — The episode terminated because the environment generated a termination signal.
- `2` — The episode terminated by reaching the maximum episode length.

`Mask` — Sequence padding mask
logical array

Sequence padding mask, returned as a logical array with length equal to `SequenceLength`. When the sampled sequence length is less than `SequenceLength`, the data returned in `experience` is padded. Each element of `Mask` is `true` for a real experience and `false` for a padded experience.

You can ignore `Mask` when `SequenceLength` is 1.

Version History

Introduced in R2022a

See Also

Functions
`append`

Objects
`rlReplayMemory` | `rlPrioritizedReplayMemory` | `rlHindsightReplayMemory` | `rlHindsightPrioritizedReplayMemory`

setActor

Package: `rl.agent`

Set actor of reinforcement learning agent

Syntax

```
agent = setActor(agent,actor)
```

Description

`agent = setActor(agent,actor)` updates the reinforcement learning agent, `agent`, to use the specified actor object, `actor`.

Examples

Modify Actor Parameter Values

Assume that you have an existing trained reinforcement learning agent. For this example, load the trained agent from “Train DDPG Agent to Control Double Integrator System”.

```
load("DoubleIntegDDPG.mat","agent")
```

Obtain the actor function approximator from the agent.

```
actor = getActor(agent);
```

Obtain the learnable parameters from the actor.

```
params = getLearnableParameters(actor)
```

```
params=2×1 cell array
    {[-15.4601 -7.2076]}
    {[               0]}
```

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,"UniformOutput",false);
```

Set the parameter values of the actor to the new modified values.

```
actor = setLearnableParameters(actor,modifiedParams);
```

Set the actor in the agent to the new modified actor.

```
setActor(agent,actor);
```

Display the new parameter values.

```
getLearnableParameters(getActor(agent))
```

```
ans=2×1 cell array
    {[-30.9201 -14.4153]}
    {[                0]}
```

Modify Deep Neural Networks in Reinforcement Learning Agent

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”.

Load the predefined environment.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a PPO agent from the environment observation and action specifications. This agent uses default deep neural networks for its actor and critic.

```
agent = rlPPOAgent(obsInfo,actInfo);
```

To modify the deep neural networks within a reinforcement learning agent, you must first extract the actor and critic function approximators.

```
actor = getActor(agent);
critic = getCritic(agent);
```

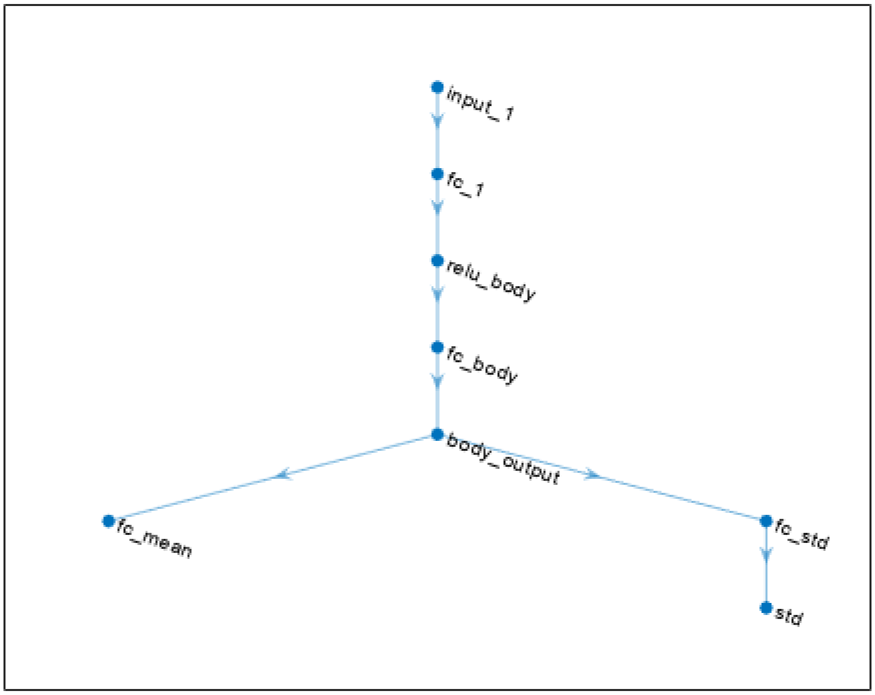
Extract the deep neural networks from both the actor and critic function approximators.

```
actorNet = getModel(actor);
criticNet = getModel(critic);
```

The networks are `dlnetwork` objects. To view them using the `plot` function, you must convert them to `layerGraph` objects.

For example, view the actor network.

```
plot(layerGraph(actorNet))
```



To validate a network, use `analyzeNetwork`. For example, validate the critic network.

```
analyzeNetwork(criticNet)
```

You can modify the actor and critic networks and save them back to the agent. To modify the networks, you can use the Deep Network Designer app. To open the app for each network, use the following commands.

```
deepNetworkDesigner(layerGraph(criticNet))
deepNetworkDesigner(layerGraph(actorNet))
```

In **Deep Network Designer**, modify the networks. For example, you can add additional layers to your network. When you modify the networks, do not change the input and output layers of the networks returned by `getModel`. For more information on building networks, see “Build Networks with Deep Network Designer”.

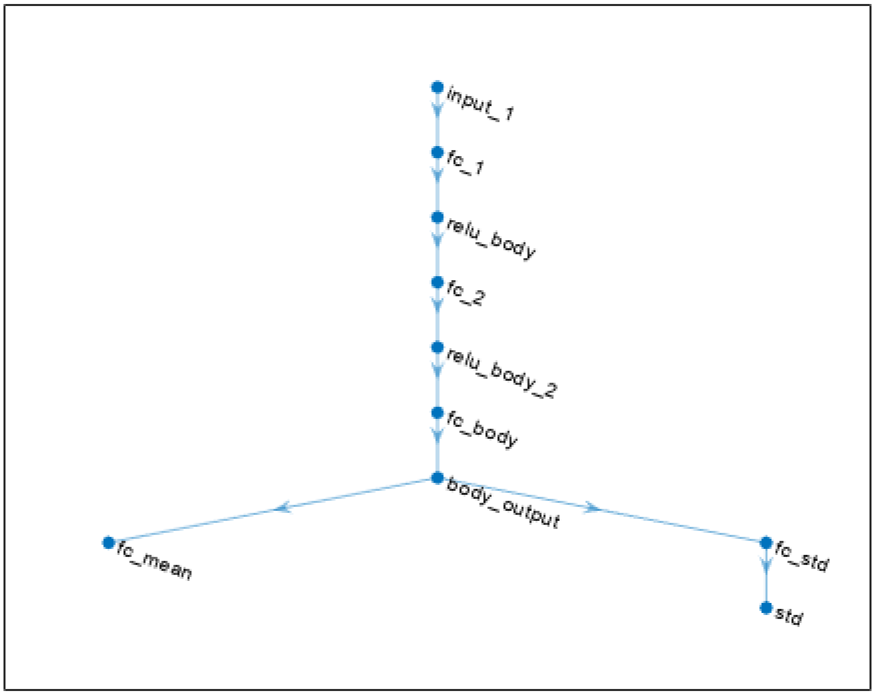
To validate the modified network in **Deep Network Designer**, you must click on **Analyze for dlnetwork**, under the **Analysis** section. To export the modified network structures to the MATLAB® workspace, generate code for creating the new networks and run this code from the command line. Do not use the exporting option in **Deep Network Designer**. For an example that shows how to generate and run code, see “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”.

For this example, the code for creating the modified actor and critic networks is in the `createModifiedNetworks` helper script.

```
createModifiedNetworks
```

Each of the modified networks includes an additional `fullyConnectedLayer` and `reluLayer` in their main common path. View the modified actor network.

```
plot(layerGraph(modifiedActorNet))
```



After exporting the networks, insert the networks into the actor and critic function approximators.

```
actor = setModel(actor,modifiedActorNet);
critic = setModel(critic,modifiedCriticNet);
```

Finally, insert the modified actor and critic function approximators into the actor and critic objects.

```
agent = setActor(agent,actor);
agent = setCritic(agent,critic);
```

Input Arguments

agent — Reinforcement learning agent

`rlPGAgent` | `rlDDPGAgent` | `rlTD3Agent` | `rlACAgent` | `rlSACAgent` | `rlPPOAgent` | `rlTRPOAgent`

Reinforcement learning agent that contains an actor, specified as one of the following:

- `rlPGAgent` object
- `rlDDPGAgent` object

- `rlTD3Agent` object
- `rlACAgent` object
- `rlSACAgent` object
- `rlPPOAgent` object
- `rlTRPOAgent` object

Note `agent` is an handle object. Therefore is updated by `setActor` whether `agent` is returned as an output argument or not. For more information about handle objects, see “Handle Object Behavior”.

Note if `agent` is an `rlMBPOAgent` object, to set the actor, use `setActor(agent.BaseAgent,actor)`.

`actor` — Actor
`rlContinuousDeterministicActor` object | `rlDiscreteCategoricalActor` object | `rlContinuousGaussianActor` object

Actor object, specified as one of the following:

- `rlContinuousDeterministicActor` object — Specify when `agent` is an `rlDDPGAgent` or `rlTD3Agent` object
- `rlDiscreteCategoricalActor` object — Specify when `agent` is an `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, `rlTRPOAgent` or `rlSACAgent` object for an environment with a discrete action space.
- `rlContinuousGaussianActor` object — Specify when `agent` is an `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, `rlTRPOAgent` or `rlSACAgent` object for an environment with a continuous action space.

The input and outputs of the approximation model in the actor (typically, a neural network) must match the observation and action specifications of the original agent.

To create an actor, use one of the following methods:

- Create the actor using the corresponding function approximator object.
- Obtain the existing actor from an agent using `getActor`.

Output Arguments

`agent` — Updated reinforcement learning agent
`rlPGAgent` | `rlDDPGAgent` | `rlTD3Agent` | `rlACAgent` | `rlSACAgent` | `rlPPOAgent` | `rlTRPOAgent`

Updated agent, returned as an agent object. Note that `agent` is an handle object. Therefore its actor is updated by `setActor` whether `agent` is returned as an output argument or not. For more information about handle objects, see “Handle Object Behavior”.

Version History

Introduced in R2019a

See Also

Functions

`getActor` | `getCritic` | `setCritic` | `getModel` | `setModel` | `getLearnableParameters` | `setLearnableParameters`

Topics

"Create Policies and Value Functions"
"Import Neural Network Models"

setCritic

Package: rl.agent

Set critic of reinforcement learning agent

Syntax

```
agent = setCritic(agent,critic)
```

Description

`agent = setCritic(agent,critic)` updates the reinforcement learning agent, `agent`, to use the specified critic object, `critic`.

Examples

Modify Critic Parameter Values

Assume that you have an existing trained reinforcement learning agent. For this example, load the trained agent from “Train DDPG Agent to Control Double Integrator System”.

```
load("DoubleIntegDDPG.mat","agent")
```

Obtain the critic function approximator from the agent.

```
critic = getCritic(agent);
```

Obtain the learnable parameters from the critic.

```
params = getLearnableParameters(critic)
```

```
params=2×1 cell array
    {[-4.9869 -1.5577 -0.3351 -0.1097 -0.0444 7.5240e-04]}
    {[                                                 0]}
```

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,"UniformOutput",false);
```

Set the parameter values of the critic to the new modified values.

```
critic = setLearnableParameters(critic,modifiedParams);
```

Set the critic in the agent to the new modified critic.

```
setCritic(agent,critic);
```

Display the new parameter values.

```
getLearnableParameters(getCritic(agent))
```

```
ans=2×1 cell array
    {[-9.9737 -3.1153 -0.6702 -0.2194 -0.0888 0.0015]}
    {[                                            0]}
```

Modify Deep Neural Networks in Reinforcement Learning Agent

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”.

Load the predefined environment.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a PPO agent from the environment observation and action specifications. This agent uses default deep neural networks for its actor and critic.

```
agent = rlPPOAgent(obsInfo,actInfo);
```

To modify the deep neural networks within a reinforcement learning agent, you must first extract the actor and critic function approximators.

```
actor = getActor(agent);
critic = getCritic(agent);
```

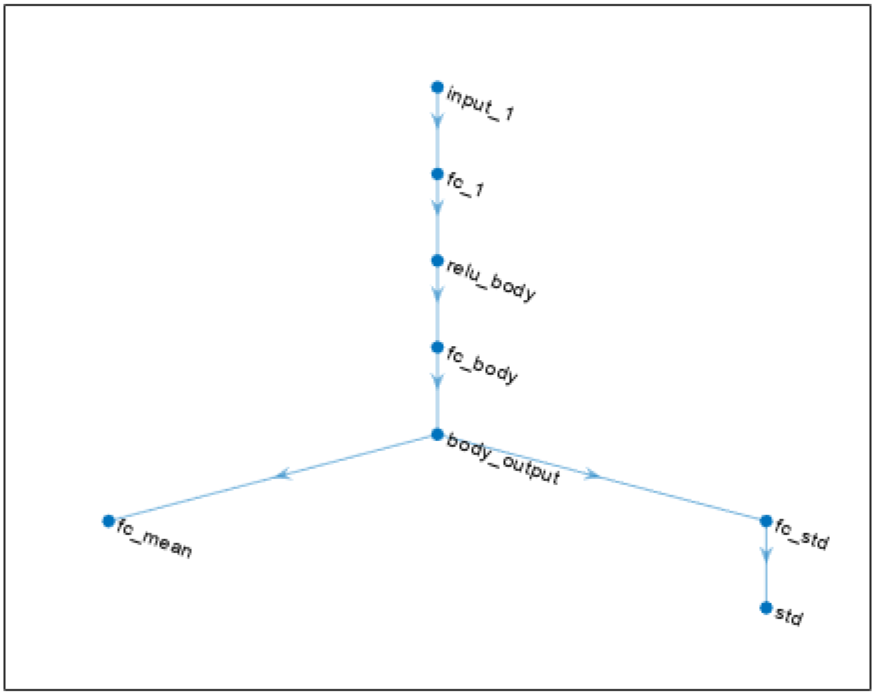
Extract the deep neural networks from both the actor and critic function approximators.

```
actorNet = getModel(actor);
criticNet = getModel(critic);
```

The networks are `dlnetwork` objects. To view them using the `plot` function, you must convert them to `layerGraph` objects.

For example, view the actor network.

```
plot(layerGraph(actorNet))
```



To validate a network, use `analyzeNetwork`. For example, validate the critic network.

```
analyzeNetwork(criticNet)
```

You can modify the actor and critic networks and save them back to the agent. To modify the networks, you can use the Deep Network Designer app. To open the app for each network, use the following commands.

```
deepNetworkDesigner(layerGraph(criticNet))
deepNetworkDesigner(layerGraph(actorNet))
```

In **Deep Network Designer**, modify the networks. For example, you can add additional layers to your network. When you modify the networks, do not change the input and output layers of the networks returned by `getModel`. For more information on building networks, see “Build Networks with Deep Network Designer”.

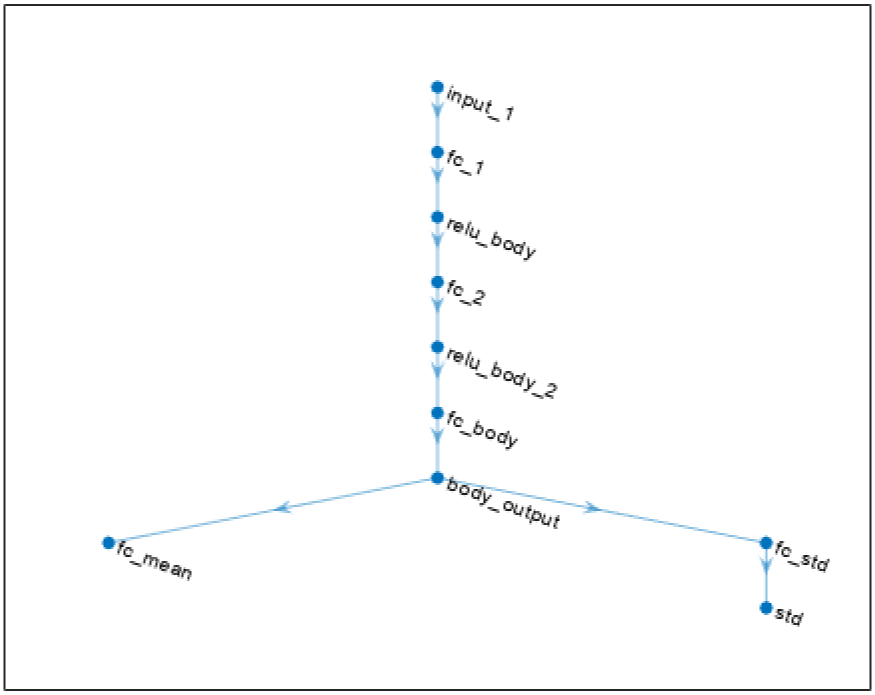
To validate the modified network in **Deep Network Designer**, you must click on **Analyze for dlnetwork**, under the **Analysis** section. To export the modified network structures to the MATLAB® workspace, generate code for creating the new networks and run this code from the command line. Do not use the exporting option in **Deep Network Designer**. For an example that shows how to generate and run code, see “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”.

For this example, the code for creating the modified actor and critic networks is in the `createModifiedNetworks` helper script.

```
createModifiedNetworks
```

Each of the modified networks includes an additional `fullyConnectedLayer` and `reluLayer` in their main common path. View the modified actor network.

```
plot(layerGraph(modifiedActorNet))
```



After exporting the networks, insert the networks into the actor and critic function approximators.

```
actor = setModel(actor,modifiedActorNet);
critic = setModel(critic,modifiedCriticNet);
```

Finally, insert the modified actor and critic function approximators into the actor and critic objects.

```
agent = setActor(agent,actor);
agent = setCritic(agent,critic);
```

Input Arguments

agent — Reinforcement learning agent

reinforcement learning agent object

Reinforcement learning agent that contains a critic, specified as one of the following:

- `rlQAgent`
- `rlSARSAAgent`

- rlDQNAgent
- rlPGAgent (when using a critic to estimate a baseline value function)
- rlDDPGAgent
- rlTD3Agent
- rlACAgent
- rlSACAgent
- rlPPOAgent
- rlTRPOAgent
- rlMBPOAgent

Note agent is an handle object. Therefore is updated by setCritic whether agent is returned as an output argument or not. For more information about handle objects, see "Handle Object Behavior".

critic — Critic
rlValueFunction object | rlQValueFunction object | rlVectorQValueFunction object | two-element row vector of rlQValueFunction objects

Critic object, specified as one of the following:

- rlValueFunction object — Returned when agent is an rlACAgent, rlPGAgent, or rlPPOAgent object.
- rlQValueFunction object — Returned when agent is an rlQAgent, rlSARSAAgent, rlDQNAgent, rlDDPGAgent, or rlTD3Agent object with a single critic.
- rlVectorQValueFunction object — Returned when agent is an rlQAgent, rlSARSAAgent, rlDQNAgent, object with a discrete action space, vector Q-value function critic.
- Two-element row vector of rlQValueFunction objects — Returned when agent is an rlTD3Agent or rlSACAgent object with two critics.

Output Arguments

agent — Updated reinforcement learning agent
rlQAgent | rlSARSAAgent | rlDQNAgent | rlPGAgent | rlDDPGAgent | rlTD3Agent | rlACAgent | rlSACAgent | rlPPOAgent | rlTRPOAgent

Updated agent, returned as an agent object. Note that agent is an handle object. Therefore its actor is updated by setCritic whether agent is returned as an output argument or not. For more information about handle objects, see "Handle Object Behavior".

Version History

Introduced in R2019a

See Also

Functions

`getActor` | `getCritic` | `setActor` | `getModel` | `setModel` | `getLearnableParameters` | `setLearnableParameters`

Topics

"Create Policies and Value Functions"
"Import Neural Network Models"

setLearnableParameters

Package: rl.policy

Set learnable parameter values of agent, function approximator, or policy object

Syntax

```
setLearnableParameters(agent)
agent = setLearnableParameters(agent)
```

```
newFcn = setLearnableParameters(oldFcn,pars)
```

```
newPol = setLearnableParameters(oldPol,pars)
```

Description

Agent

`setLearnableParameters(agent)` sets the learnable parameter values specified in `pars` in the specified agent.

`agent = setLearnableParameters(agent)` also returns the new agent as an output argument.

Actor or Critic

`newFcn = setLearnableParameters(oldFcn,pars)` returns a new actor or critic function approximator object, `newFcn`, with the same structure as the original function object, `oldFcn`, and the learnable parameter values specified in `pars`.

Policy

`newPol = setLearnableParameters(oldPol,pars)` returns a new policy object, `newPol`, with the same structure as the original function object, `oldFcn`, and the learnable parameter values specified in `pars`.

Examples

Modify Critic Parameter Values

Assume that you have an existing trained reinforcement learning agent. For this example, load the trained agent from “Train DDPG Agent to Control Double Integrator System”.

```
load("DoubleIntegDDPG.mat","agent")
```

Obtain the critic function approximator from the agent.

```
critic = getCritic(agent);
```

Obtain the learnable parameters from the critic.

```
params = getLearnableParameters(critic)
```

```
params=2×1 cell array
    {[-4.9869 -1.5577 -0.3351 -0.1097 -0.0444 7.5240e-04]}
    {[                                                 0]}
```

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,"UniformOutput",false);
```

Set the parameter values of the critic to the new modified values.

```
critic = setLearnableParameters(critic,modifiedParams);
```

Set the critic in the agent to the new modified critic.

```
setCritic(agent,critic);
```

Display the new parameter values.

```
getLearnableParameters(getCritic(agent))
```

```
ans=2×1 cell array
    {[-9.9737 -3.1153 -0.6702 -0.2194 -0.0888 0.0015]}
    {[                                             0]}
```

Modify Actor Parameter Values

Assume that you have an existing trained reinforcement learning agent. For this example, load the trained agent from “Train DDPG Agent to Control Double Integrator System”.

```
load("DoubleIntegDDPG.mat","agent")
```

Obtain the actor function approximator from the agent.

```
actor = getActor(agent);
```

Obtain the learnable parameters from the actor.

```
params = getLearnableParameters(actor)
```

```
params=2×1 cell array
    {[-15.4601 -7.2076]}
    {[               0]}
```

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,"UniformOutput",false);
```

Set the parameter values of the actor to the new modified values.

```
actor = setLearnableParameters(actor,modifiedParams);
```

Set the actor in the agent to the new modified actor.

```
setActor(agent,actor);
```

Display the new parameter values.

```
getLearnableParameters(getActor(agent))
```

```
ans=2×1 cell array
    {[-30.9201 -14.4153]}
    {[               0]}
```

Input Arguments

agent — Reinforcement learning agent

reinforcement learning agent object

Reinforcement learning agent, specified as one of the following objects:

- `rlQAgent`
- `rlSARSAAgent`
- `rlDQNAgent`
- `rlPGAgent`
- `rlDDPGAgent`
- `rlTD3Agent`
- `rlACAgent`
- `rlSACAgent`
- `rlPPOAgent`
- `rlTRPOAgent`
- Custom agent — For more information, see “Create Custom Reinforcement Learning Agents”.

Note `agent` is an handle object. Therefore its parameters are updated by `setLearnableParameters` whether `agent` is returned as an output argument or not. For more information about handle objects, see “Handle Object Behavior”.

oldFcn — Original actor or critic function object

reinforcement learning function approximator object

Original function approximator object, specified as one of the following:

- `rlValueFunction` object — Value function critic
- `rlQValueFunction` object — Q-value function critic
- `rlVectorQValueFunction` object — Multi-output Q-value function critic with a discrete action space
- `rlContinuousDeterministicActor` object — Deterministic policy actor with a continuous action space
- `rlDiscreteCategoricalActor` object — Stochastic policy actor with a discrete action space
- `rlContinuousGaussianActor` object — Stochastic policy actor with a continuous action space
- `rlContinuousDeterministicTransitionFunction` object — Continuous deterministic transition function for neural network environments

- rlContinuousGaussianTransitionFunction object — Continuous Gaussian transition function for neural network environments
- rlContinuousDeterministicRewardFunction object — Continuous deterministic reward function for neural network environments
- rlContinuousGaussianRewardFunction object — Continuous Gaussian reward function for neural network environments
- rlIsDoneFunction object — IsDone function for neural network environments

To create an actor or critic function object, use one of the following methods.

- Create a function object directly.
- Obtain the existing critic from an agent using getCritic.
- Obtain the existing actor from an agent using getActor.

oldPol — Reinforcement learning policy
rlMaxQPolicy | rlEpsilonGreedyPolicy | rlDeterministicActorPolicy | rlAdditiveNoisePolicy | rlStochasticActorPolicy

Reinforcement learning policy, specified as one of the following objects:

- rlMaxQPolicy
- rlEpsilonGreedyPolicy
- rlDeterministicActorPolicy
- rlAdditiveNoisePolicy
- rlStochasticActorPolicy

pars — Learnable parameter values
cell array

Learnable parameter values for the representation object, specified as a cell array. The parameters in pars must be compatible with the structure and parameterization of the agent, function approximator, or policy object passed as a first argument.

To obtain a cell array of learnable parameter values from an existing agent, function approximator, or policy object , which you can then modify, use the getLearnableParameters function.

Output Arguments

newFcn — New actor or critic object
rlValueFunction object | rlQValueFunction object | rlVectorQValueFunction object | rlContinuousDeterministicActor object | rlDiscreteCategoricalActor object | rlContinuousGaussianActor object

New actor or critic object, returned as a function object of the same type as oldFcn. Apart from the learnable parameter values, newFcn is the same as oldFcn.

newPol — New reinforcement learning policy
rlMaxQPolicy | rlEpsilonGreedyPolicy | rlDeterministicActorPolicy | rlAdditiveNoisePolicy | rlStochasticActorPolicy

New reinforcement learning policy, returned as a policy object of the same type as `oldPol`. Apart from the learnable parameter values, `newPol` is the same as `oldPol`.

`agent` — Updated agent
reinforcement learning agent object

Updated agent, returned as an agent object. Note that `agent` is an handle object. Therefore its parameters are updated by `setLearnableParameters` whether `agent` is returned as an output argument or not. For more information about handle objects, see “Handle Object Behavior”.

Version History

Introduced in R2019a

R2022a: `setLearnableParameters` now uses approximator objects instead of representation objects

Behavior changed in R2022a

Using representation objects to create actors and critics for reinforcement learning agents is no longer recommended. Therefore, `setLearnableParameters` now uses function approximator objects instead.

R2020a: `setLearnableParameterValues` is now `setLearnableParameters`

Behavior changed in R2020a

`setLearnableParameterValues` is now `setLearnableParameters`. To update your code, change the function name from `setLearnableParameterValues` to `setLearnableParameters`. The syntaxes are equivalent.

See Also

Functions
`getLearnableParameters` | `getActor` | `getCritic` | `setActor` | `setCritic` | `evaluate`

Objects
`rlValueFunction` | `rlContinuousDeterministicActor` | `rlContinuousDeterministicTransitionFunction` | `rlQAgent` | `rlPGAgent` | `rlACAgent` | `rlMaxQPolicy`

Topics
“Create Policies and Value Functions”
“Reinforcement Learning Agents”
“Import Neural Network Models”

setModel

Package: `rl.function`

Set function approximation model for actor or critic

Syntax

```
newFcnAppx = setModel(oldFcnAppx,model)
```

Description

`newFcnAppx = setModel(oldFcnAppx,model)` returns a new actor or critic function object, `newFcnAppx`, with the same configuration as the original function object, `oldFcnAppx`, and the computational model specified in `model`.

Examples

Modify Deep Neural Networks in Reinforcement Learning Agent

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”.

Load the predefined environment.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a PPO agent from the environment observation and action specifications. This agent uses default deep neural networks for its actor and critic.

```
agent = rlPPOAgent(obsInfo,actInfo);
```

To modify the deep neural networks within a reinforcement learning agent, you must first extract the actor and critic function approximators.

```
actor = getActor(agent);
critic = getCritic(agent);
```

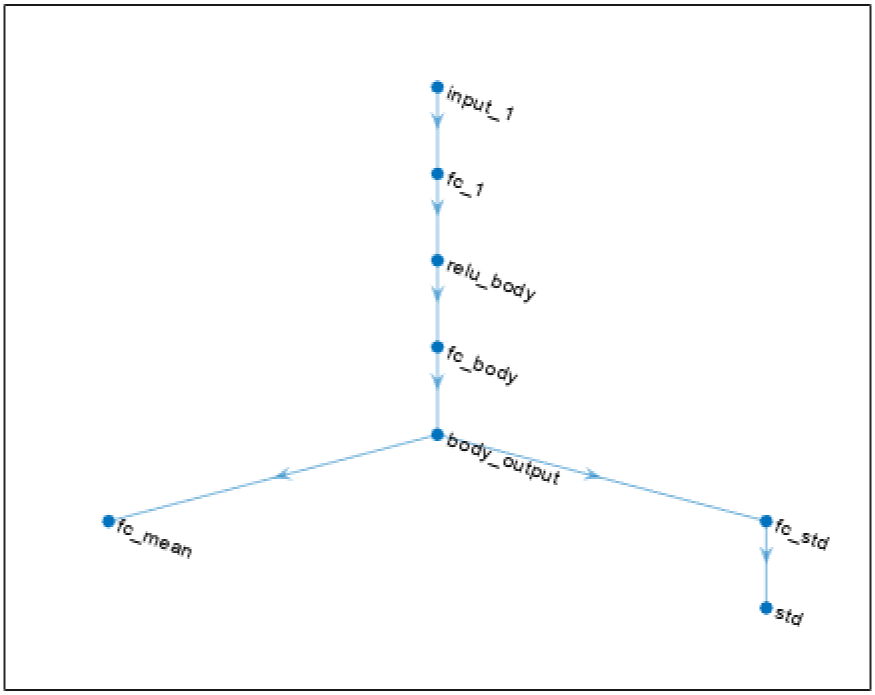
Extract the deep neural networks from both the actor and critic function approximators.

```
actorNet = getModel(actor);
criticNet = getModel(critic);
```

The networks are `dlnetwork` objects. To view them using the `plot` function, you must convert them to `layerGraph` objects.

For example, view the actor network.

```
plot(layerGraph(actorNet))
```



To validate a network, use `analyzeNetwork`. For example, validate the critic network.

```
analyzeNetwork(criticNet)
```

You can modify the actor and critic networks and save them back to the agent. To modify the networks, you can use the Deep Network Designer app. To open the app for each network, use the following commands.

```
deepNetworkDesigner(layerGraph(criticNet))
deepNetworkDesigner(layerGraph(actorNet))
```

In **Deep Network Designer**, modify the networks. For example, you can add additional layers to your network. When you modify the networks, do not change the input and output layers of the networks returned by `getModel`. For more information on building networks, see “Build Networks with Deep Network Designer”.

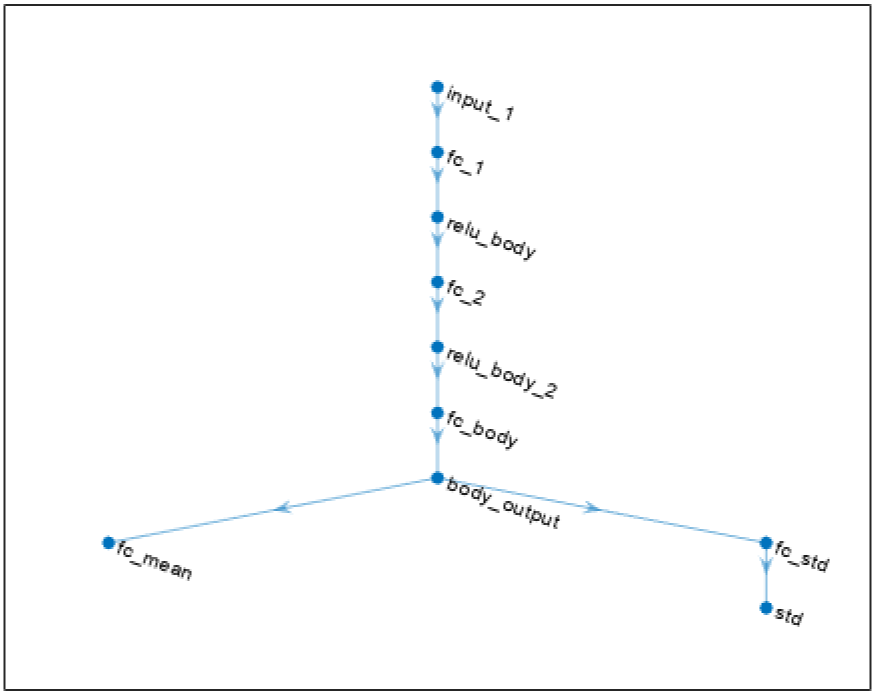
To validate the modified network in **Deep Network Designer**, you must click on **Analyze for dlnetwork**, under the **Analysis** section. To export the modified network structures to the MATLAB® workspace, generate code for creating the new networks and run this code from the command line. Do not use the exporting option in **Deep Network Designer**. For an example that shows how to generate and run code, see “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”.

For this example, the code for creating the modified actor and critic networks is in the `createModifiedNetworks` helper script.

```
createModifiedNetworks
```

Each of the modified networks includes an additional `fullyConnectedLayer` and `reluLayer` in their main common path. View the modified actor network.

```
plot(layerGraph(modifiedActorNet))
```



After exporting the networks, insert the networks into the actor and critic function approximators.

```
actor = setModel(actor,modifiedActorNet);
critic = setModel(critic,modifiedCriticNet);
```

Finally, insert the modified actor and critic function approximators into the actor and critic objects.

```
agent = setActor(agent,actor);
agent = setCritic(agent,critic);
```

Input Arguments

`oldFcnAppx` — Original actor or critic function object

`rlValueFunction` object | `rlQValueFunction` object | `rlVectorQValueFunction` object | `rlContinuousDeterministicActor` object | `rlDiscreteCategoricalActor` object | `rlContinuousGaussianActor` object

Original actor or critic function object, specified as one of the following:

- rlValueFunction object — Value function critic
- rlQValueFunction object — Q-value function critic
- rlVectorQValueFunction object — Multi-output Q-value function critic with a discrete action space
- rlContinuousDeterministicActor object — Deterministic policy actor with a continuous action space
- rlDiscreteCategoricalActor — Stochastic policy actor with a discrete action space
- rlContinuousGaussianActor object — Stochastic policy actor with a continuous action space

To create an actor or critic function object, use one of the following methods.

- Create a function object directly.
- Obtain the existing critic from an agent using getCritic.
- Obtain the existing actor from an agent using getActor.

model — Function approximation model
array of Layer objects | layerGraph object | DAGNetwork object | dlnetwork object | rlTable object | 1-by-2 cell array

Function approximation model, specified as one of the following:

- Deep neural network defined as an array of Layer objects, a layerGraph object, a DAGNetwork object, or a dlnetwork object. The input and output layers of model must have the same names and dimensions as the network returned by getModel for the same function object. Here, the output layer is the layer immediately before the output loss layer.
- rlTable object with the same dimensions as the table model defined in newRep.
- 1-by-2 cell array that contains the function handle for a custom basis function and the basis function parameters.

When specifying a new model, you must use the same type of model as the one already defined in newRep.

Note For agents with more than one critic, such as TD3 and SAC agents, you must call setModel for each critic representation individually, rather than calling setModel for the array of returned by getCritic.

```
critics = getCritic(myTD3Agent);
% Modify critic networks.
critics(1) = setModel(critics(1),criticNet1);
critics(2) = setModel(critics(2),criticNet2);
myTD3Agent = setCritic(myTD3Agent,critics);
```

Output Arguments

newFcnAppx — New actor or critic function object
rlValueFunction object | rlQValueFunction object | rlVectorQValueFunction object | rlContinuousDeterministicActor object | rlDiscreteCategoricalActor object | rlContinuousGaussianActor object

New actor or critic function object, returned as a function object of the same type as `oldFcnAppx`. Apart from the new computational model, `newFcnAppx` is the same as `oldFcnAppx`.

Version History

Introduced in R2020b

R2022a: `setModel` now uses approximator objects instead of representation objects

Behavior changed in R2022a

Using representation objects to create actors and critics for reinforcement learning agents is no longer recommended. Therefore, `setModel` now uses function approximator objects instead.

See Also

Functions

`getActor` | `setActor` | `getCritic` | `setCritic` | `getModel`

Topics

"Create Policies and Value Functions"

setup

Package: rl.env

Set up reinforcement learning environment or initialize data logger object

Syntax

```
setup(env)
setup(env,Name=Value)

setup(lgr)
```

Description

When you define a custom training loop for reinforcement learning, you can simulate an agent or policy against an environment using the runEpisode function. Use the setup function to configure the environment for running simulations using multiple calls to runEpisode.

Also use setup to initialize a FileLogger or MonitorLogger object before logging data within a custom training loop.

Environment Objects

setup(env) sets up the specified reinforcement learning environment for running multiple simulations using runEpisode.

setup(env,Name=Value) specifies nondefault configuration options using one or more name-value pair arguments.

Data Logger Objects

setup(lgr) sets up the specified data logger object. Setup tasks may include setting up a visualization, or creating directories for logging to file.

Examples

Simulate Environment and Agent

Create a reinforcement learning environment and extract its observation and action specifications.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

To approximate the Q-value function withing the critic, use a neural network. Create a network as an array of layer objects.

```
net = [...
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(2)
    softmaxLayer];
```

Convert the network to a `dlnetwork` object and display the number of learnable parameters (weights).

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 770

   Inputs:
      1   'input'   4 features
```

Create a discrete categorical actor using the network.

```
actor = rlDiscreteCategoricalActor(net,obsInfo,actInfo);
```

Check your actor with a random observation.

```
act = getAction(actor,{rand(obsInfo.Dimension)})
```

```
act = 1x1 cell array
    {[-10]}
```

Create a policy object from the actor.

```
policy = rlStochasticActorPolicy(actor);
```

Create an experience buffer.

```
buffer = rlReplayMemory(obsInfo,actInfo);
```

Set up the environment for running multiple simulations. For this example, configure the training to log any errors rather than send them to the command window.

```
setup(env,StopOnError="off")
```

Simulate multiple episodes using the environment and policy. After each episode, append the experiences to the buffer. For this example, run 100 episodes.

```
for i = 1:100
    output = runEpisode(env,policy,MaxSteps=300);
    append(buffer,output.AgentData.Experiences)
end
```

Clean up the environment.

```
cleanup(env)
```

Sample a mini-batch of experiences from the buffer. For this example, sample 10 experiences.

```
batch = sample(buffer,10);
```

You can then learn from the sampled experiences and update the policy and actor.

Log Data to Disk in a Custom Training Loop

This example shows how to log data to disk when training an agent using a custom training loop.

Create a `FileLogger` object using `rlDataLogger`.

```
flgr = rlDataLogger();
```

Set up the logger object. This operation initializes the object performing setup tasks such as, for example, creating the directory to save the data files.

```
setup(flgr);
```

Within a custom training loop, you can now store data to the logger object memory and write data to file.

For this example, store random numbers to the file logger object, grouping them in the variables `Context1` and `Context2`. When you issue a write command, a MAT file corresponding to an iteration and containing both variables is saved with the name specified in `flgr.LoggingOptions.FileNameRule`, in the folder specified by `flgr.LoggingOptions.LoggingDirectory`.

```
for iter = 1:10

    % Store three random numbers in memory
    % as elements of the variable "Context1"
    for ct = 1:3
        store(flgr, "Context1", rand, iter);
    end

    % Store a random number in memory
    % as the variable "Context2"
    store(flgr, "Context2", rand, iter);

    % Write data to file every 4 iterations
    if mod(iter,4)==0
        write(flgr);
    end

end
```

Clean up the logger object. This operation performs clean up tasks like for example writing to file any data still in memory.

```
cleanup(flgr);
```

Input Arguments

env — Reinforcement learning environment
`rlFunctionEnv` object | `SimulinkEnvWithAgent` object | `rlNeuralNetworkEnvironment` object | `rlMDPEnv` object | ...

Reinforcement learning environment, specified as one of the following objects.

- rlFunctionEnv — Environment defined using custom functions.
- SimulinkEnvWithAgent — Simulink environment created using rlSimulinkEnv or createIntegratedEnv
- rlMDPEnv — Markov decision process environment
- rlNeuralNetworkEnvironment — Environment with deep neural network transition models
- Predefined environment created using rlPredefinedEnv
- Custom environment created from a template (rlCreateEnvTemplate)

lgr — Date logger object
FileLogger object | MonitorLogger object | ...

Data logger object, specified as either a FileLogger or a MonitorLogger object.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: StopOnError="on"

StopOnError — Option to stop episode when error occurs
"on" (default) | "off"

Option to stop an episode when an error occurs, specified as one of the following:

- "on" — Stop the episode when an error occurs and generate an error message in the MATLAB command window.
- "off" — Log errors in the SimulationInfo output of runEpisode.

UseParallel — Option for using parallel simulations
false (default) | true

Option for using parallel simulations, specified as a logical value. Using parallel computing allows the usage of multiple cores, processors, computer clusters, or cloud resources to speed up simulation.

When you set UseParallel to true, the output of a subsequent call to runEpisode is an rl.env.Future object, which supports deferred evaluation of the simulation.

SetupFcn — Function to run on each worker before running an episode
[] (default) | function handle

Function to run on the each worker before running an episode, specified as a handle to a function with no input arguments. Use this function to perform any preprocessing required before running an episode.

CleanupFcn — Function to run on each worker when cleaning up environment
[] (default) | function handle

Function to run on each worker when cleaning up the environment, specified as a handle to a function with no input arguments. Use this function to clean up the workspace or perform other processing after calling runEpisode.

`TransferBaseWorkspaceVariables` — Option to send model and workspace variables to parallel workers

`"on"` (default) | `"off"`

Option to send model and workspace variables to parallel workers, specified as `"on"` or `"off"`. When the option is `"on"`, the client sends variables used in models and defined in the base MATLAB workspace to the workers.

`AttachedFiles` — Additional files to attach to parallel pool

string | string array

Additional files to attach to the parallel pool before running an episode, specified as a string or string array.

`WorkerRandomSeeds` — Work random seeds

`-1` (default) | vector

Worker random seeds, specified as one of the following:

- `-1` — Set the random seed of each worker to the worker ID.
- Vector with length equal to the number of workers — Specify the random seed for each worker.

Version History

Introduced in R2022a

See Also

Functions

`runEpisode` | `cleanup` | `reset` | `store` | `write`

Objects

`rlFunctionEnv` | `rlMDPEnv` | `SimulinkEnvWithAgent` | `rlNeuralNetworkEnvironment` | `FileLogger` | `MonitorLogger`

Topics

“Custom Training Loop with Simulink Action Noise”

sim

Package: `rl.env`

Simulate trained reinforcement learning agents within specified environment

Syntax

```
experience = sim(env,agents)
experience = sim(agents,env)

env = sim(___,simOpts)
```

Description

`experience = sim(env,agents)` simulates one or more reinforcement learning agents within an environment, using default simulation options.

`experience = sim(agents,env)` performs the same simulation as the previous syntax.

`env = sim(___,simOpts)` uses the simulation options object `simOpts`. Use simulation options to specify parameters such as the number of steps per simulation or the number of simulations to run. Use this syntax after any of the input arguments in the previous syntaxes.

Examples

Simulate Reinforcement Learning Environment

Simulate a reinforcement learning environment with an agent configured for that environment. For this example, load an environment and agent that are already configured. The environment is a discrete cart-pole environment created with `rlPredefinedEnv`. The agent is a policy gradient (`rlPGAgent`) agent. For more information about the environment and agent used in this example, see “Train PG Agent to Balance Cart-Pole System”.

```
rng(0) % for reproducibility
load RLSimExample.mat
env
```

```
env = 
  CartPoleDiscreteAction with properties:

                  Gravity: 9.8000
                 MassCart: 1
                 MassPole: 0.1000
                   Length: 0.5000
                 MaxForce: 10
                       Ts: 0.0200
    ThetaThresholdRadians: 0.2094
               XThreshold: 2.4000
      RewardForNotFalling: 1
        PenaltyForFalling: -5
```

```
                    State: [4x1 double]
```

```
agent
```

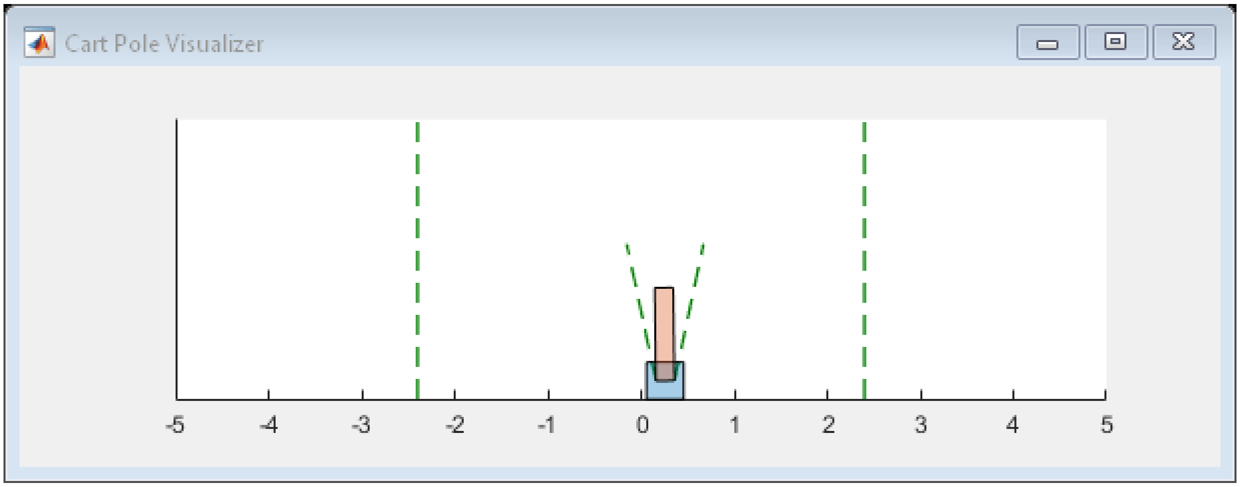
```
agent = 
  rlPGAgent with properties:

            AgentOptions: [1x1 rl.option.rlPGAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 0.1000
```

Typically, you train the agent using `train` and simulate the environment to test the performance of the trained agent. For this example, simulate the environment using the agent you loaded. Configure simulation options, specifying that the simulation run for 100 steps.

```
simOpts = rlSimulationOptions(MaxSteps=100);
```

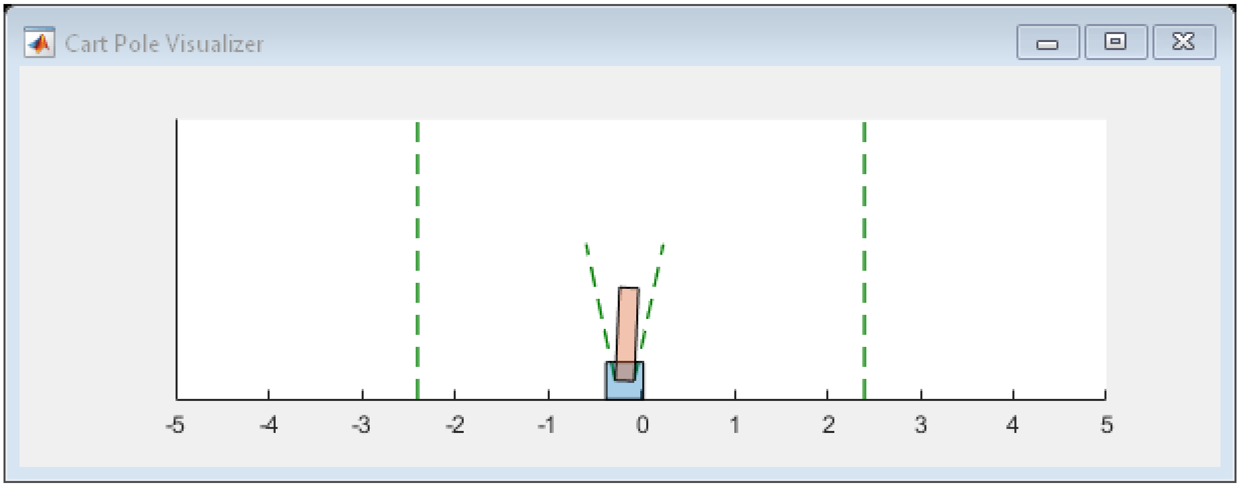
For the predefined cart-pole environment used in this example. you can use `plot` to generate a visualization of the cart-pole system. When you simulate the environment, this plot updates automatically so that you can watch the system evolve during the simulation.

```
plot(env)
```



Simulate the environment.

```
experience = sim(env,agent,simOpts)
```



```
experience = struct with fields:
       Observation: [1x1 struct]
            Action: [1x1 struct]
            Reward: [1x1 timeseries]
            IsDone: [1x1 timeseries]
    SimulationInfo: [1x1 struct]
```

The output structure `experience` records the observations collected from the environment, the action and reward, and other data collected during the simulation. Each field contains a `timeseries` object or a structure of `timeseries` data objects. For instance, `experience.Action` is a `timeseries` containing the action imposed on the cart-pole system by the agent at each step of the simulation.

```
experience.Action
```

```
ans = struct with fields:
    CartPoleAction: [1x1 timeseries]
```

Simulate Simulink Environment with Multiple Agents

Simulate an environment created for the Simulink® model used in the example “Train Multiple Agents to Perform Collaborative Task”, using the agents trained in that example.

Load the agents in the MATLAB® workspace.

```
load rlCollaborativeTaskAgents
```

Create an environment for the `rlCollaborativeTask` Simulink® model, which has two agent blocks. Since the agents used by the two blocks (`agentA` and `agentB`) are already in the workspace, you do not need to pass their observation and action specifications to create the environment.

```
env = rlSimulinkEnv( ...
    "rlCollaborativeTask", ...
    ["rlCollaborativeTask/Agent A","rlCollaborativeTask/Agent B"]);
```

Load the parameters that are needed by the `rlCollaborativeTask` Simulink® model to run.

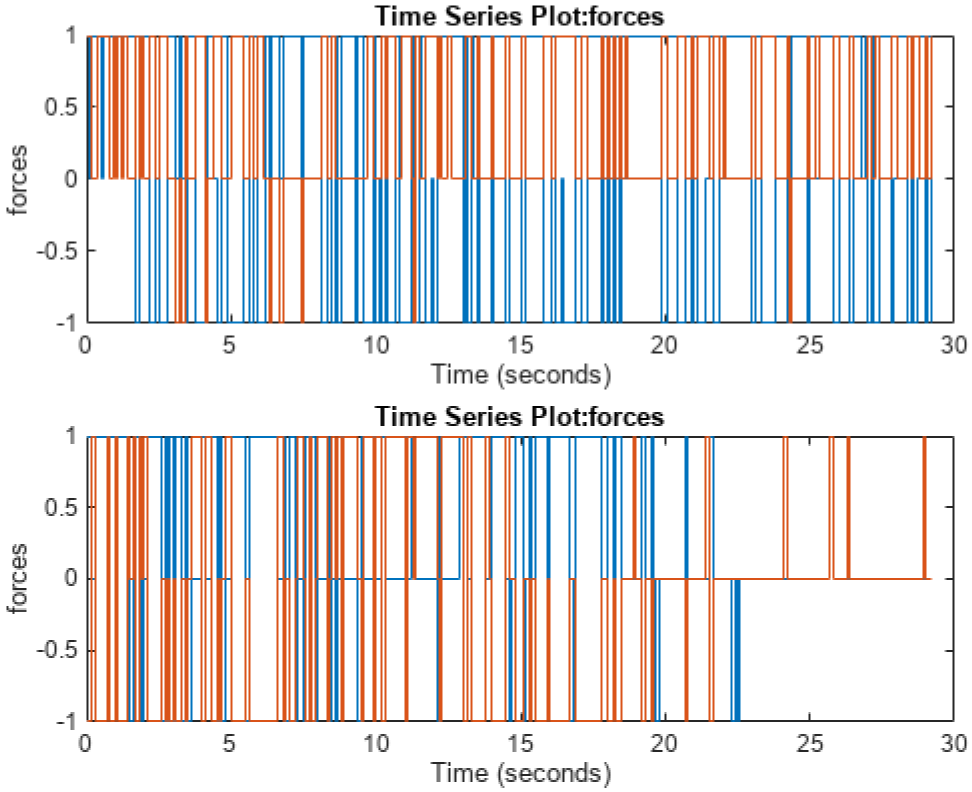
```
rlCollaborativeTaskParams
```

Simulate the agents against the environment, saving the experiences in `xpr`.

```
xpr = sim(env,[agentA agentB]);
```

Plot actions of both agents.

```
subplot(2,1,1); plot(xpr(1).Action.forces)
subplot(2,1,2); plot(xpr(2).Action.forces)
```



Input Arguments

env — Environment
reinforcement learning environment object

Environment in which the agents act, specified as one of the following kinds of reinforcement learning environment object:

- A predefined MATLAB or Simulink environment created using `rlPredefinedEnv`. This kind of environment does not support training multiple agents at the same time.
- A custom MATLAB environment you create with functions such as `rlFunctionEnv` or `rlCreateEnvTemplate`. This kind of environment does not support training multiple agents at the same time.

- A custom Simulink environment you create using `rlSimulinkEnv`. This kind of environment supports training multiple agents at the same time.

For more information about creating and configuring environments, see:

- "Create MATLAB Reinforcement Learning Environments"
- "Create Simulink Reinforcement Learning Environments"

When `env` is a Simulink environment, calling `sim` compiles and simulates the model associated with the environment.

`agents` — Agents
reinforcement learning agent object | array of agent objects

Agents to simulate, specified as a reinforcement learning agent object, such as `rlACAgent` or `rlDDPGAgent`, or as an array of such objects.

If `env` is a multi-agent environment created with `rlSimulinkEnv`, specify agents as an array. The order of the agents in the array must match the agent order used to create `env`. Multi-agent simulation is not supported for MATLAB environments.

For more information about how to create and configure agents for reinforcement learning, see "Reinforcement Learning Agents".

`simOpts` — Simulation options
`rlSimulationOptions` object

Simulation options, specified as an `rlSimulationOptions` object. Use this argument to specify options such as:

- Number of steps per simulation
- Number of simulations to run

For details, see `rlSimulationOptions`.

Output Arguments

`experience` — Simulation results
structure | structure array

Simulation results, returned as a structure or structure array. The number of rows in the array is equal to the number of simulations specified by the `NumSimulations` option of `rlSimulationOptions`. The number of columns in the array is the number of agents. The fields of each `experience` structure are as follows.

`Observation` — Observations
structure

Observations collected from the environment, returned as a structure with fields corresponding to the observations specified in the environment. Each field contains a `timeseries` of length $N + 1$, where N is the number of simulation steps.

To obtain the current observation and the next observation for a given simulation step, use code such as the following, assuming one of the fields of `Observation` is `obs1`.

```
Obs = getSamples(experience.Observation.obs1,1:N);
NextObs = getSamples(experience.Observation.obs1,2:N+1);
```

These values can be useful if you are writing your own training algorithm using `sim` to generate experiences for training.

`Action` — Actions

structure

Actions computed by the agent, returned as a structure with fields corresponding to the action signals specified in the environment. Each field contains a `timeseries` of length *N*, where *N* is the number of simulation steps.

`Reward` — Rewards

`timeseries`

Reward at each step in the simulation, returned as a `timeseries` of length *N*, where *N* is the number of simulation steps.

`IsDone` — Flag indicating termination of episode

`timeseries`

Flag indicating termination of the episode, returned as a `timeseries` of a scalar logical signal. This flag is set at each step by the environment, according to conditions you specify for episode termination when you configure the environment. When the environment sets this flag to 1, simulation terminates.

`SimulationInfo` — Information collected during simulation

structure | vector of `Simulink.SimulationOutput` objects

Information collected during simulation, returned as one of the following:

- For MATLAB environments, a structure containing the field `SimulationError`. This structure contains any errors that occurred during simulation.
- For Simulink environments, a `Simulink.SimulationOutput` object containing simulation data. Recorded data includes any signals and states that the model is configured to log, simulation metadata, and any errors that occurred.

Version History

Introduced in R2019a

See Also

Functions
`train`

Objects
`rlSimulationOptions`

Topics
“Train Reinforcement Learning Agents”
“Reinforcement Learning Agents”

store

Package: rl.logging

Store data in the internal memory of a (file or monitor) logger object

Syntax

store(lgr,context,data,iter)

Description

store(lgr,context,data,iter) stores data into lgr internal memory, grouped in a variable named context and associated with iteration iter.

Examples

Log Data to Disk in a Custom Training Loop

This example shows how to log data to disk when training an agent using a custom training loop.

Create a FileLogger object using rlDataLogger.

```
flgr = rlDataLogger();
```

Set up the logger object. This operation initializes the object performing setup tasks such as, for example, creating the directory to save the data files.

```
setup(flgr);
```

Within a custom training loop, you can now store data to the logger object memory and write data to file.

For this example, store random numbers to the file logger object, grouping them in the variables Context1 and Context2. When you issue a write command, a MAT file corresponding to an iteration and containing both variables is saved with the name specified in flgr.LoggingOptions.FileNameRule, in the folder specified by flgr.LoggingOptions.LoggingDirectory.

```
for iter = 1:10

    % Store three random numbers in memory
    % as elements of the variable "Context1"
    for ct = 1:3
        store(flgr, "Context1", rand, iter);
    end

    % Store a random number in memory
    % as the variable "Context2"
    store(flgr, "Context2", rand, iter);
    % Write data to file every 4 iterations
    if mod(iter,4)==0
        write(flgr);
    end

end
```

Clean up the logger object. This operation performs clean up tasks like for example writing to file any data still in memory.

```
cleanup(flgr);
```

Input Arguments

lgr — Date logger object
FileLogger object | MonitorLogger object | ...

Data logger object, specified as either a `FileLogger` or a `MonitorLogger` object.

context — Name of the saved variable
string | char array | ...

Name of the saved variable, specified as either a string or character array. All data associated with iteration `iter` and the name `context` is vertically concatenated in a single MATLAB variable named `context`. This variable is then written to the logger target (either a MAT file or a `trainingProgressMonitor` object) when `write` is invoked for `lgr`.

data — Data to be saved
any fundamental MATLAB datatype | ...

Data to be saved, specified as any fundamental MATLAB datatype. Data associated with the same iteration and the same context name is grouped in a single variable.

iter — Iteration number
positive integer | ...

Iteration number, specified as a positive integer. When `store` is executed multiple times with the same iteration number, data is appended to the memory entry for that iteration. This memory entry, which can contain several context variables, is then written to the logger target as a single unit (for example as a single MAT file) when `write` is invoked for `lgr`.

Version History

Introduced in R2022b

See Also

Functions
`rlDataLogger` | `train` | `write` | `setup` | `cleanup`

Objects
`FileLogger` | `MonitorLogger`

Topics

“Log Training Data to Disk”
“Function Handles”
“Handle Object Behavior”

train

Package: `rl.agent`

Train reinforcement learning agents within a specified environment

Syntax

```
trainStats = train(env,agents)
trainStats = train(agents,env)
```

```
trainStats = train( ___ ,trainOpts)
```

```
trainStats = train(agents,env,prevTrainStats)
```

```
trainStats = train( ___ ,logger=lgr)
```

Description

`trainStats = train(env,agents)` trains one or more reinforcement learning agents within a specified environment, using default training options. Although `agents` is an input argument, after each training episode, `train` updates the parameters of each agent specified in `agents` to maximize their expected long-term reward from the environment. This is possible because each agent is an handle object. When training terminates, `agents` reflects the state of each agent at the end of the final training episode.

Note To train an off-policy agent offline using existing data, use `trainFromData`.

`trainStats = train(agents,env)` performs the same training as the previous syntax.

`trainStats = train( ___ ,trainOpts)` trains `agents` within `env`, using the training options object `trainOpts`. Use training options to specify training parameters such as the criteria for terminating training, when to save agents, the maximum number of episodes to train, and the maximum number of steps per episode. Use this syntax after any of the input arguments in the previous syntaxes.

`trainStats = train(agents,env,prevTrainStats)` resumes training from the last values of the agent parameters and training results contained in `prevTrainStats` obtained after the previous function call to `train`.

`trainStats = train( ___ ,logger=lgr)` logs training data using the `FileLogger` or `MonitorLogger` `lgr` object.

Examples

Train a Reinforcement Learning Agent

Train the agent configured in the “Train PG Agent to Balance Cart-Pole System” example, within the corresponding environment. The observation from the environment is a vector containing the position and velocity of a cart, as well as the angular position and velocity of the pole. The action is a scalar with two possible elements (a force of either -10 or 10 Newtons applied to a cart).

Load the file containing the environment and a PG agent already configured for it.

```
load RLTrainExample.mat
```

Specify some training parameters using `rlTrainingOptions`. These parameters include the maximum number of episodes to train, the maximum steps per episode, and the conditions for terminating training. For this example, use a maximum of 2000 episodes and 500 steps per episode. Instruct the training to stop when the average reward over the previous five episodes reaches 500. Create a default options set and use dot notation to change some of the parameter values.

```
trainOpts = rlTrainingOptions;

trainOpts.MaxEpisodes = 2000;
trainOpts.MaxStepsPerEpisode = 500;
trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = 500;
trainOpts.ScoreAveragingWindowLength = 5;
```

During training, the `train` command can save candidate agents that give good results. Further configure the training options to save an agent when the episode reward exceeds 500. Save the agent to a folder called `savedAgents`.

```
trainOpts.SaveAgentCriteria = "EpisodeReward";
trainOpts.SaveAgentValue = 500;
trainOpts.SaveAgentDirectory = "savedAgents";
```

Turn off the command-line display. Turn on the Reinforcement Learning Episode Manager so you can observe the training progress visually.

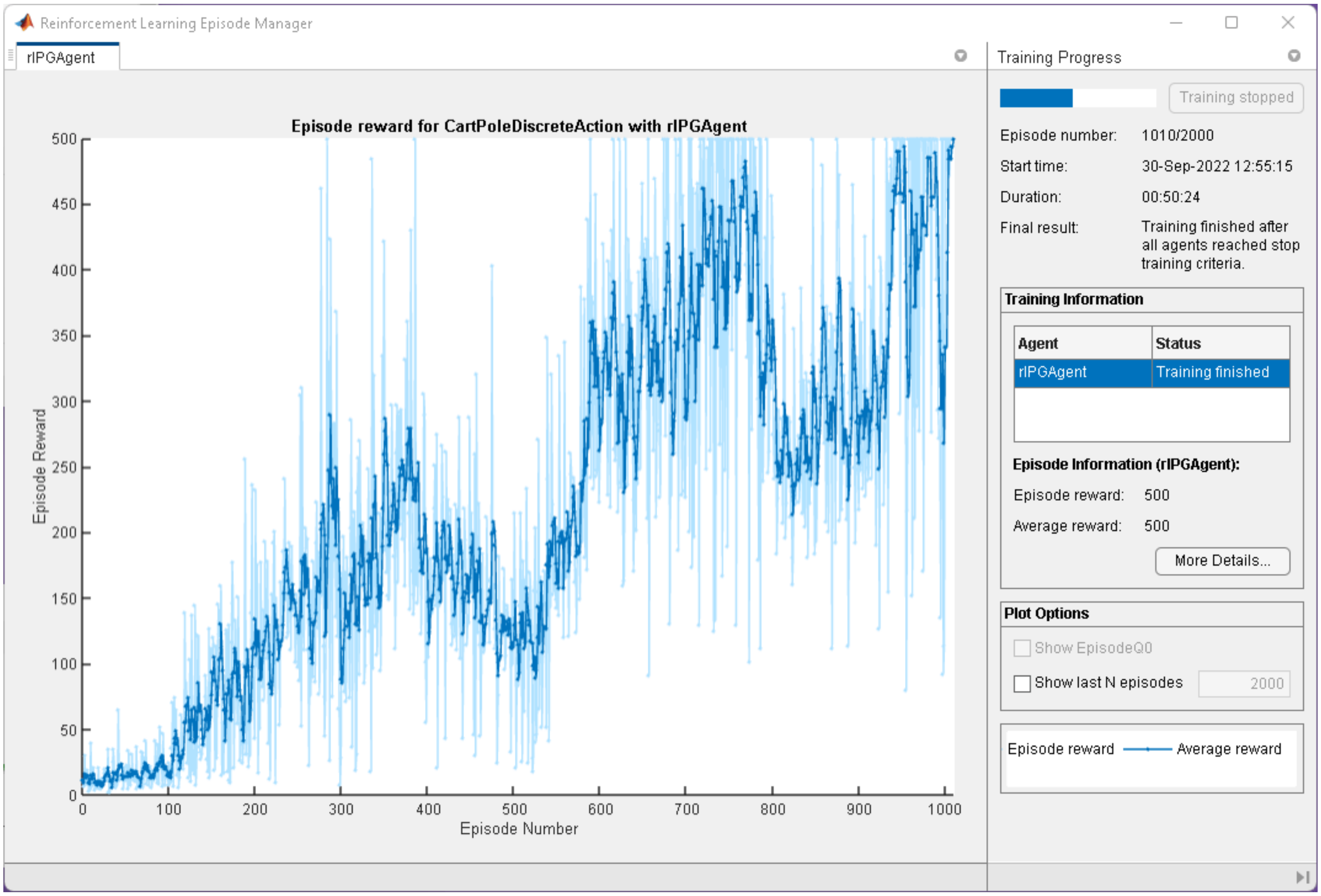
```
trainOpts.Verbose = false;
trainOpts.Plots = "training-progress";
```

You are now ready to train the PG agent. For the predefined cart-pole environment used in this example, you can use `plot` to generate a visualization of the cart-pole system.

```
plot(env)
```

When you run this example, both this visualization and the Reinforcement Learning Episode Manager update with each training episode. Place them side by side on your screen to observe the progress, and train the agent. (This computation can take 20 minutes or more.)

```
trainingInfo = train(agent,env,trainOpts);
```



Episode number: 1010/2000

Start time: 30-Sep-2022 12:55:15

Duration: 00:50:24

Final result: Training finished after all agents reached stop training criteria.

Training Information

Agent	Status
rlPGAgent	Training finished

Episode Information (rlPGAgent):

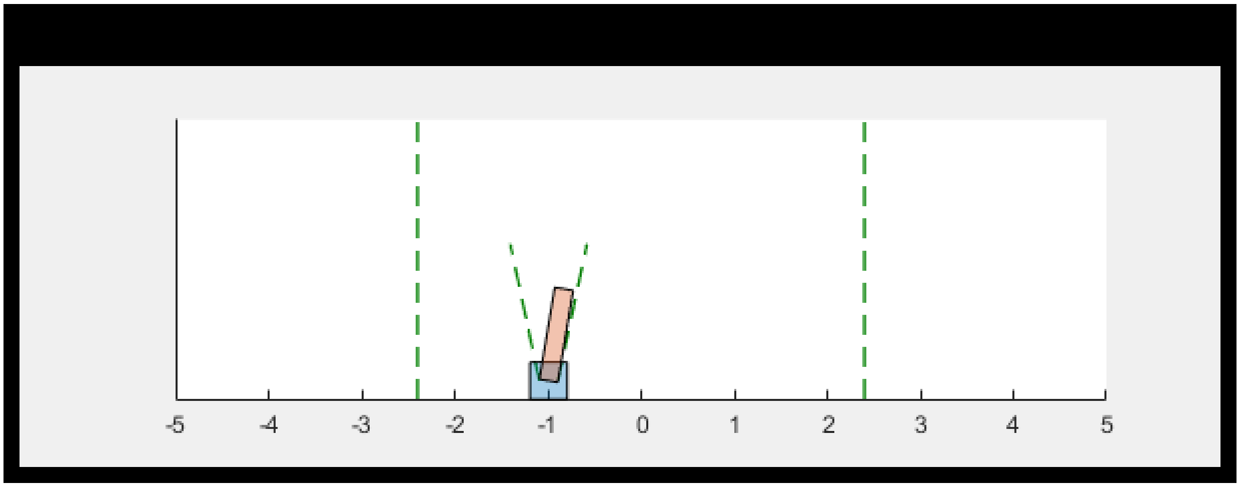
Episode reward: 500

Average reward: 500

Plot Options

Episode Manager shows that the training successfully reaches the termination condition of a reward of 500 averaged over the previous five episodes. At each training episode, `train` updates `agent` with the parameters learned in the previous episode. When training terminates, you can simulate the environment with the trained agent to evaluate its performance. The environment plot updates during simulation as it did during training.

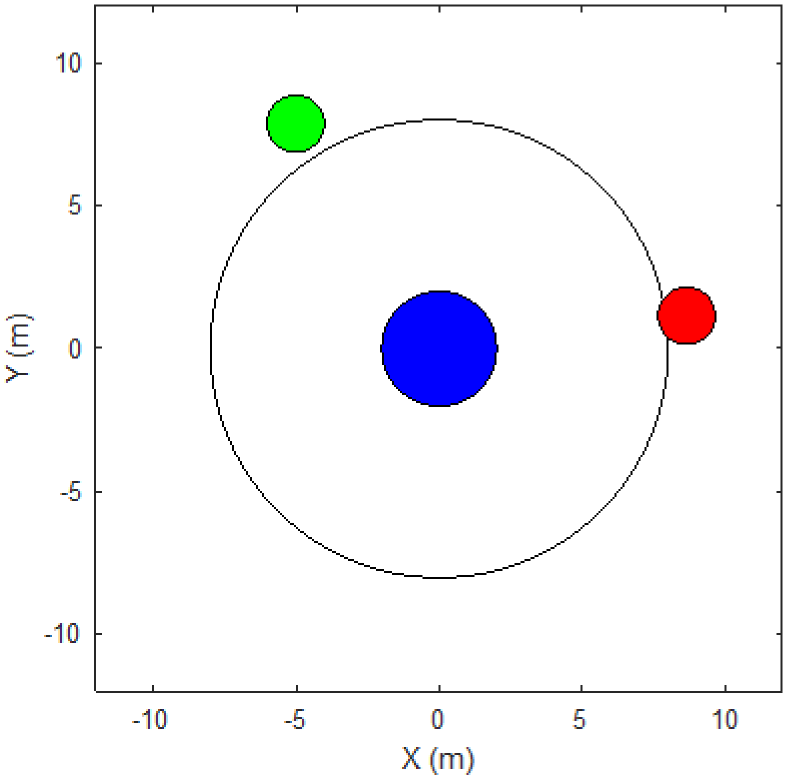
```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```



During training, `train` saves to disk any agents that meet the condition specified with `trainOps.SaveAgentCritera` and `trainOpts.SaveAgentValue`. To test the performance of any of those agents, you can load the data from the data files in the folder you specified using `trainOpts.SaveAgentDirectory`, and simulate the environment with that agent.

Train Multiple Agents to Perform Collaborative Task

This example shows how to set up a multi-agent training session on a Simulink® environment. In the example, you train two agents to collaboratively perform the task of moving an object.



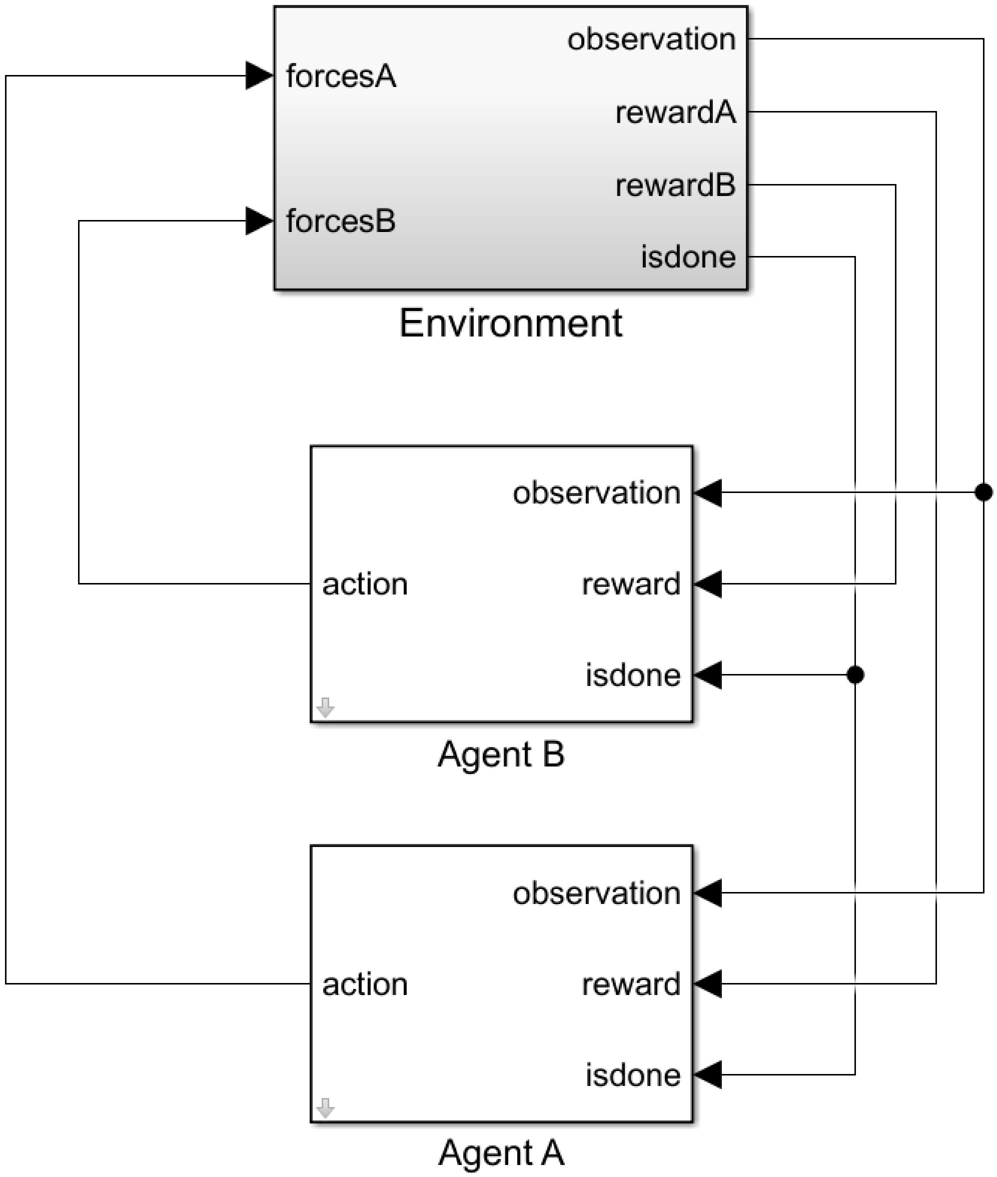
The environment in this example is a frictionless two dimensional surface containing elements represented by circles. A target object C is represented by the blue circle with a radius of 2 m, and robots A (red) and B (green) are represented by smaller circles with radii of 1 m each. The robots attempt to move object C outside a circular ring of a radius 8 m by applying forces through collision. All elements within the environment have mass and obey Newton's laws of motion. In addition, contact forces between the elements and the environment boundaries are modeled as spring and mass damper systems. The elements can move on the surface through the application of externally applied forces in the X and Y directions. There is no motion in the third dimension and the total energy of the system is conserved.

Set the random seed and create the set of parameters required for this example.

```
rng(10)
rlCollaborativeTaskParams
```

Open the Simulink model.

```
mdl = "rlCollaborativeTask";
open_system(mdl)
```



For this environment:

- The 2-dimensional space is bounded from –12 m to 12 m in both the X and Y directions.
- The contact spring stiffness and damping values are 100 N/m and 0.1 N/m/s, respectively.
- The agents share the same observations for positions, velocities of A, B, and C and the action values from the last time step.
- The simulation terminates when object C moves outside the circular ring.
- At each time step, the agents receive the following reward:

$$
\begin{aligned}
r_A &= r_{\text{global}} + r_{\text{local},A} \\
r_B &= r_{\text{global}} + r_{\text{local},B} \\
r_{\text{global}} &= 0.001 d_c \\
r_{\text{local},A} &= -0.005 d_{\text{AC}} - 0.008 u_A^2 \\
r_{\text{local},B} &= -0.005 d_{\text{BC}} - 0.008 u_B^2
\end{aligned}
$$

Here:

- $r_A$and r_B are the rewards received by agents A and B, respectively.
- r_{global} is a team reward that is received by both agents as object C moves closer towards the boundary of the ring.
- $r_{\text{local},A}$ and $r_{\text{local},B}$ are local penalties received by agents A and B based on their distances from object C and the magnitude of the action from the last time step.
- d_C is the distance of object C from the center of the ring.
- d_{AC} and d_{BC} are the distances between agent A and object C and agent B and object C, respectively.
- The agents apply external forces on the robots that result in motion. u_A and u_B are the action values of the two agents A and B from the last time step. The range of action values is between -1 and 1.

Environment

To create a multi-agent environment, specify the block paths of the agents using a string array. Also, specify the observation and action specification objects using cell arrays. The order of the specification objects in the cell array must match the order specified in the block path array. When agents are available in the MATLAB workspace at the time of environment creation, the observation and action specification arrays are optional. For more information on creating multi-agent environments, see `rlSimulinkEnv`.

Create the I/O specifications for the environment. In this example, the agents are homogeneous and have the same I/O specifications.

```
% Number of observations
numObs = 16;

% Number of actions
numAct = 2;

% Maximum value of externally applied force (N)
maxF = 1.0;

% I/O specifications for each agent
oinfo = rlNumericSpec([numObs,1]);
ainfo = rlNumericSpec([numAct,1], ...
    UpperLimit= maxF, ...
    LowerLimit= -maxF);
oinfo.Name = "observations";
ainfo.Name = "forces";
```

Create the Simulink environment interface.

```
blks = ["rlCollaborativeTask/Agent A", "rlCollaborativeTask/Agent B"];
obsInfos = {oinfo,oinfo};
actInfos = {ainfo,ainfo};
env = rlSimulinkEnv(mdl,blks,obsInfos,actInfos);
```

Specify a reset function for the environment. The reset function `resetRobots` ensures that the robots start from random initial positions at the beginning of each episode.

```
env.ResetFcn = @(in) resetRobots(in,RA,RB,RC,boundaryR);
```

Agents

This example uses two Proximal Policy Optimization (PPO) agents with continuous action spaces. The agents apply external forces on the robots that result in motion. To learn more about PPO agents, see “Proximal Policy Optimization (PPO) Agents”.

The agents collect experiences until the experience horizon (600 steps) is reached. After trajectory completion, the agents learn from mini-batches of 300 experiences. An objective function clip factor of 0.2 is used to improve training stability and a discount factor of 0.99 is used to encourage long-term rewards.

Specify the agent options for this example.

```
agentOptions = rlPPOAgentOptions(...
    ExperienceHorizon=600,...
    ClipFactor=0.2,...
    EntropyLossWeight=0.01,...
    MiniBatchSize=300,...
    NumEpoch=4,...
    AdvantageEstimateMethod="gae",...
    GAEFactor=0.95,...
    SampleTime=Ts,...
    DiscountFactor=0.99);
agentOptions.ActorOptimizerOptions.LearnRate  = 1e-4;
agentOptions.CriticOptimizerOptions.LearnRate = 1e-4;
```

Create the agents using the default agent creation syntax. For more information see `rlPPOAgent`.

```
agentA = rlPPOAgent(oinfo, ainfo, ...
    rlAgentInitializationOptions(NumHiddenUnit= 200), agentOptions);
agentB = rlPPOAgent(oinfo, ainfo, ...
    rlAgentInitializationOptions(NumHiddenUnit= 200), agentOptions);
```

Training

To train multiple agents, you can pass an array of agents to the `train` function. The order of agents in the array must match the order of agent block paths specified during environment creation. Doing so ensures that the agent objects are linked to their appropriate I/O interfaces in the environment.

You can train multiple agents in a decentralized or centralized manner. In decentralized training, agents collect their own set of experiences during the episodes and learn independently from those experiences. In centralized training, the agents share the collected experiences and learn from them together. The actor and critic functions are synchronized between the agents after trajectory completion.

To configure a multi-agent training, you can create agent groups and specify a learning strategy for each group through the `rlMultiAgentTrainingOptions` object. Each agent group may contain unique agent indices, and the learning strategy can be "centralized" or "decentralized". For example, you can use the following command to configure training for three agent groups with different learning strategies. The agents with indices [1,2] and [3,4] learn in a centralized manner while agent 4 learns in a decentralized manner.

```
opts = rlMultiAgentTrainingOptions(AgentGroups= {[1,2], 4, [3,5]},
LearningStrategy= ["centralized","decentralized","centralized"])
```

For more information on multi-agent training, type `help rlMultiAgentTrainingOptions` in MATLAB.

You can perform decentralized or centralized training by running one of the following sections using the Run Section button.

1. Decentralized Training

To configure decentralized multi-agent training for this example:

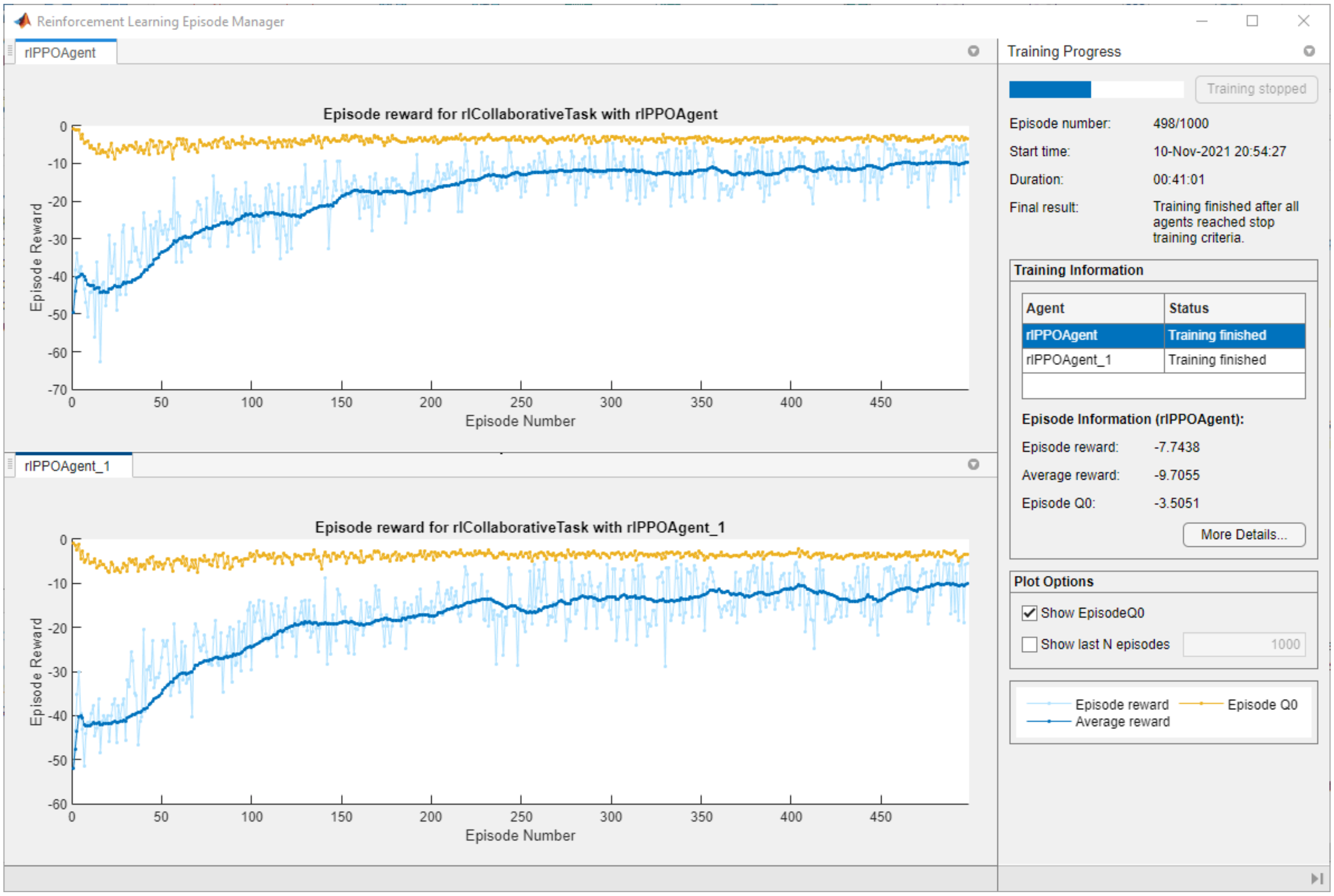
- Automatically assign agent groups using the `AgentGroups=auto` option. This allocates each agent in a separate group.
- Specify the "decentralized" learning strategy.
- Run the training for at most 1000 episodes, with each episode lasting at most 600 time steps.
- Stop the training of an agent when its average reward over 30 consecutive episodes is –10 or more.

```
trainOpts = rlMultiAgentTrainingOptions(...
    AgentGroups="auto",...
    LearningStrategy="decentralized",...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=600,...
    ScoreAveragingWindowLength=30,...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-10);
```

Train the agents using the `train` function. Training can take several hours to complete depending on the available computational power. To save time, load the MAT file `decentralizedAgents.mat` which contains a set of pretrained agents. To train the agents yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    decentralizedTrainResults = train([agentA,agentB],env,trainOpts);
else
    load("decentralizedAgents.mat");
end
```

The following figure shows a snapshot of decentralized training progress. You can expect different results due to randomness in the training process.



Episode number: 498/1000

Start time: 10-Nov-2021 20:54:27

Duration: 00:41:01

Final result: Training finished after all agents reached stop training criteria.

Training Information

Agent	Status
rlPPOAgent	Training finished
rlPPOAgent_1	Training finished

Episode Information (rlPPOAgent):

Episode reward: -7.7438

Average reward: -9.7055

Episode Q0: -3.5051

Plot Options

☑ Show EpisodeQ0

□ Show last N episodes 1000

2. Centralized Training

To configure centralized multi-agent training for this example:

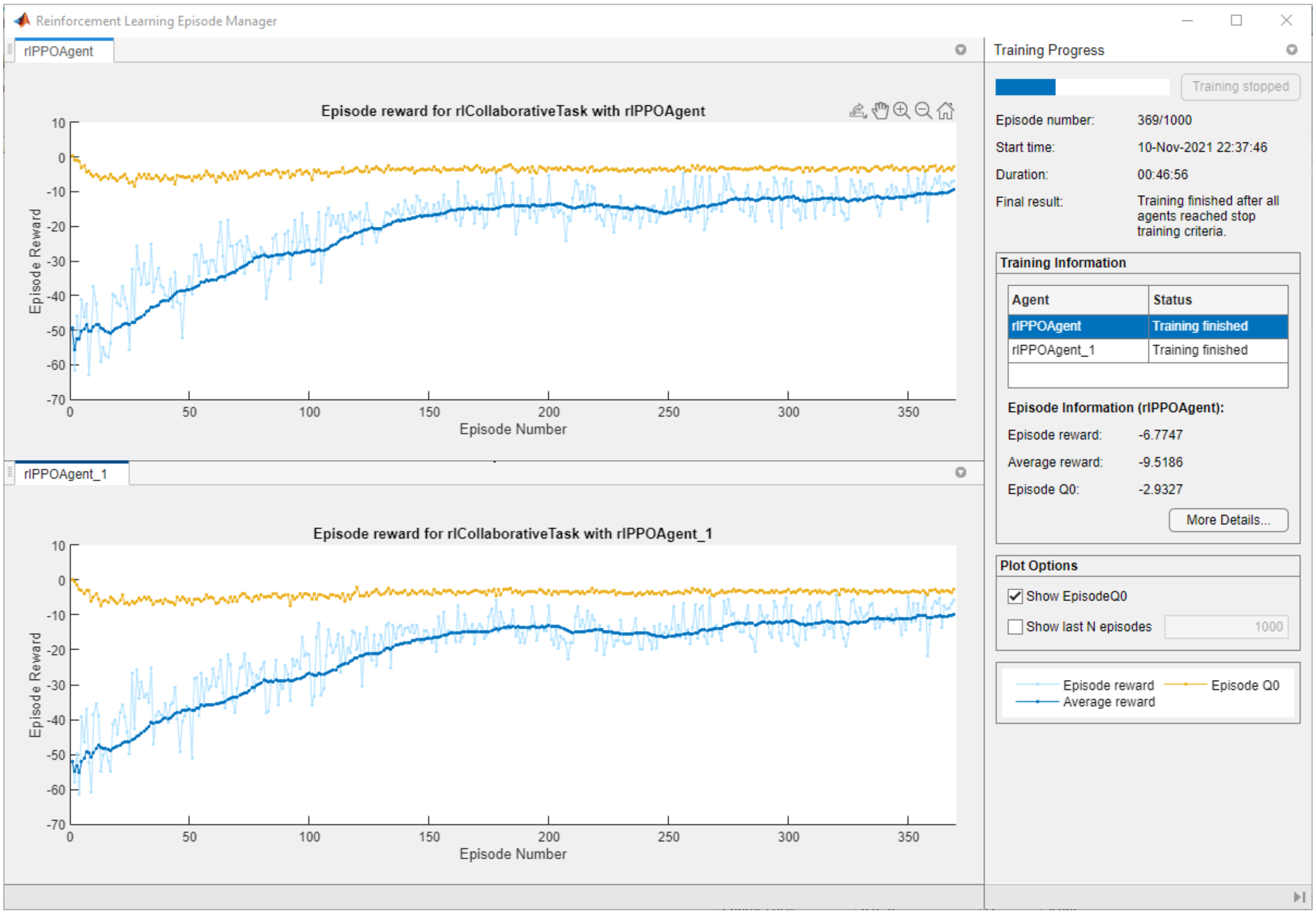
- Allocate both agents (with indices 1 and 2) in a single group. You can do this by specifying the agent indices in the "AgentGroups" option.
- Specify the "centralized" learning strategy.
- Run the training for at most 1000 episodes, with each episode lasting at most 600 time steps.
- Stop the training of an agent when its average reward over 30 consecutive episodes is –10 or more.

```
trainOpts = rlMultiAgentTrainingOptions(...
    AgentGroups={[1,2]},...
    LearningStrategy="centralized",...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=600,...
    ScoreAveragingWindowLength=30,...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-10);
```

Train the agents using the train function. Training can take several hours to complete depending on the available computational power. To save time, load the MAT file centralizedAgents.mat which contains a set of pretrained agents. To train the agents yourself, set doTraining to true.

```
doTraining = false;
if doTraining
    centralizedTrainResults = train([agentA,agentB],env,trainOpts);
else
    load("centralizedAgents.mat");
end
```

The following figure shows a snapshot of centralized training progress. You can expect different results due to randomness in the training process.



Episode number: 369/1000

Start time: 10-Nov-2021 22:37:46

Duration: 00:46:56

Final result: Training finished after all agents reached stop training criteria.

Training Information

Agent	Status
rlPPOAgent	Training finished
rlPPOAgent_1	Training finished

Episode Information (rlPPOAgent):

Episode reward: -6.7747

Average reward: -9.5186

Episode Q0: -2.9327

Plot Options

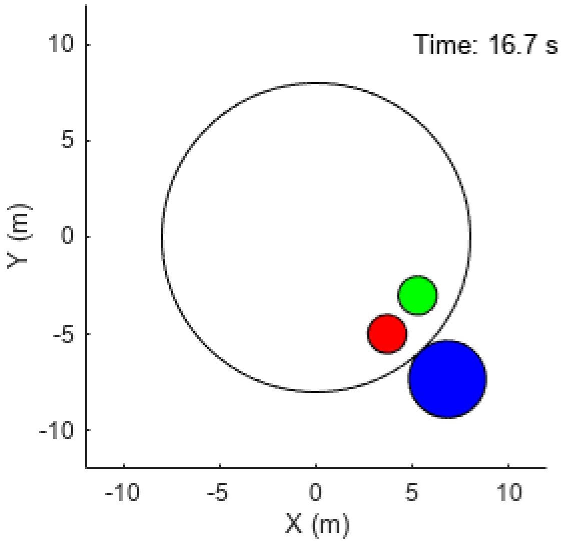
☑ Show EpisodeQ0

□ Show last N episodes 1000

Simulation

Once the training is finished, simulate the trained agents with the environment.

```
simOptions = rlSimulationOptions(MaxSteps=300);
exp = sim(env,[agentA agentB],simOptions);
```



For more information on agent simulation, see `rlSimulationOptions` and `sim`.

Stop and Resume Training Using Existing Training Data

This example shows how to resume training using existing training data for training Q-learning. For more information on these agents, see “Q-Learning Agents” and “SARSA Agents”.

Create Grid World Environment

For this example, create the basic grid world environment.

```
env = rlPredefinedEnv("BasicGridWorld");
```

To specify that the initial state of the agent is always [2,1], create a reset function that returns the state number for the initial agent state.

```
x0 = [1:12 15:17 19:22 24];
env.ResetFcn = @() x0(randi(numel(x0)));
```

Fix the random generator seed for reproducibility.

```
rng(1)
```

Create Q-Learning Agent

To create a Q-learning agent, first create a Q table using the observation and action specifications from the grid world environment. Set the learning rate of the representation to 1.

```
qTable = rlTable(getObservationInfo(env),getActionInfo(env));
qVf = rlQValueFunction(qTable,getObservationInfo(env),getActionInfo(env));
```

Next, create a Q-learning agent using this table representation and configure the epsilon-greedy exploration. For more information on creating Q-learning agents, see `rlQAgent` and `rlQAgentOptions`. Keep the default value of the discount factor to `0.99`.

```
agentOpts = rlQAgentOptions;
agentOpts.EpsilonGreedyExploration.Epsilon = 0.2;
agentOpts.CriticOptimizerOptions.LearnRate = 0.2;
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 1e-3;
agentOpts.EpsilonGreedyExploration.EpsilonMin = 1e-3;
agentOpts.DiscountFactor = 1;
qAgent = rlQAgent(qVf,agentOpts);
```

Train Q-Learning Agent for 100 Episodes

To train the agent, first specify the training options. For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions;
trainOpts.MaxStepsPerEpisode = 200;
trainOpts.MaxEpisodes = 1e6;
trainOpts.Plots = "none";
trainOpts.Verbose = false;

trainOpts.StopTrainingCriteria = "EpisodeCount";
trainOpts.StopTrainingValue = 100;
trainOpts.ScoreAveragingWindowLength = 30;
```

Train the Q-learning agent using the `train` function. Training can take several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
trainingStats = train(qAgent,env,trainOpts);
```

Display index of last episode.

```
trainingStats.EpisodeIndex(end)
```

```
ans = 100
```

Train Q-Learning Agent for 200 More Episodes

Set the training to stop after episode 300.

```
trainingStats.TrainingOptions.StopTrainingValue = 300;
```

Resume the training using the training data that exists in `trainingStats`.

```
trainingStats = train(qAgent,env,trainingStats);
```

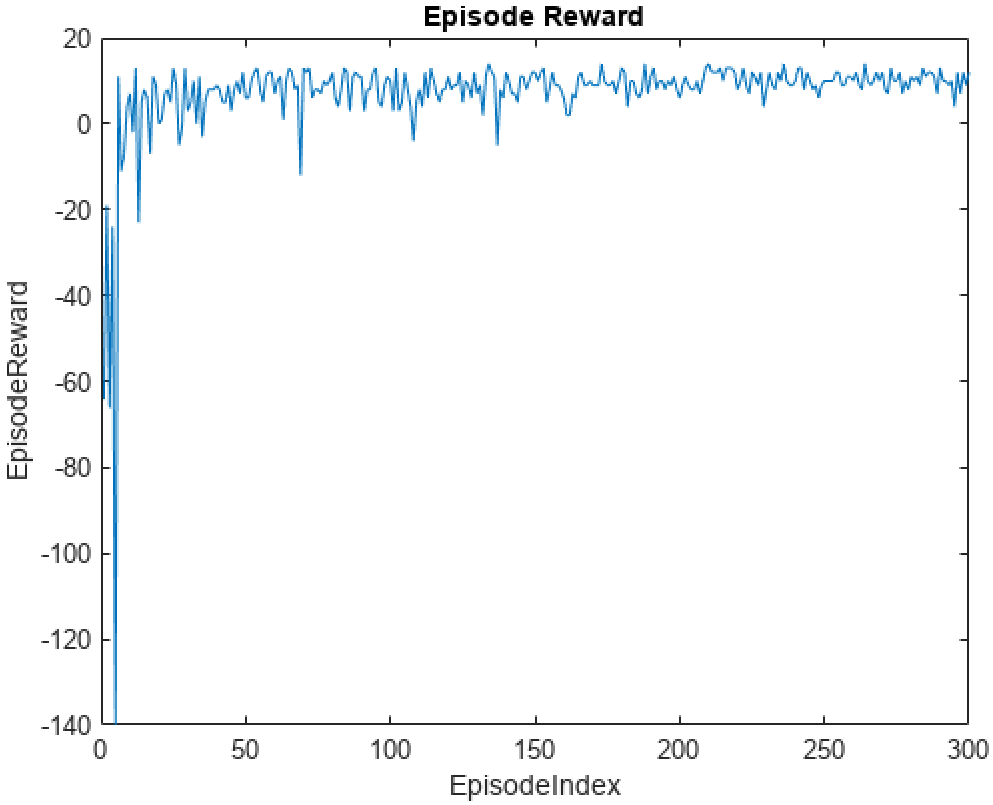
Display index of last episode.

```
trainingStats.EpisodeIndex(end)
```

```
ans = 300
```

Plot episode reward.

```
figure()
plot(trainingStats.EpisodeIndex,trainingStats.EpisodeReward)
title("Episode Reward")
xlabel("EpisodeIndex")
ylabel("EpisodeReward")
```



Display the final Q-value table.

```
qAgentFinalQ = getLearnableParameters(getCritic(qAgent));
qAgentFinalQ{1}
```

```
ans = 25x4 single matrix

    4.8373   10.0000   -1.3036    0.8020
    9.2058    4.3147   11.0000    5.8501
   10.0000    3.3987    4.5830   -6.4751
    6.3569    6.0000    8.9971    5.4393
    5.0433    5.8399    7.0067    4.1439
    5.1031    8.5228   10.9936    0.1200
    9.9616    8.8647   12.0000   10.0026
   11.0000    8.4131    8.6974    6.0001
   10.0000    6.9997    5.8122    8.5523
    7.1164    7.0019    8.0000    5.8196
      ⋮
```

Validate Q-Learning Results

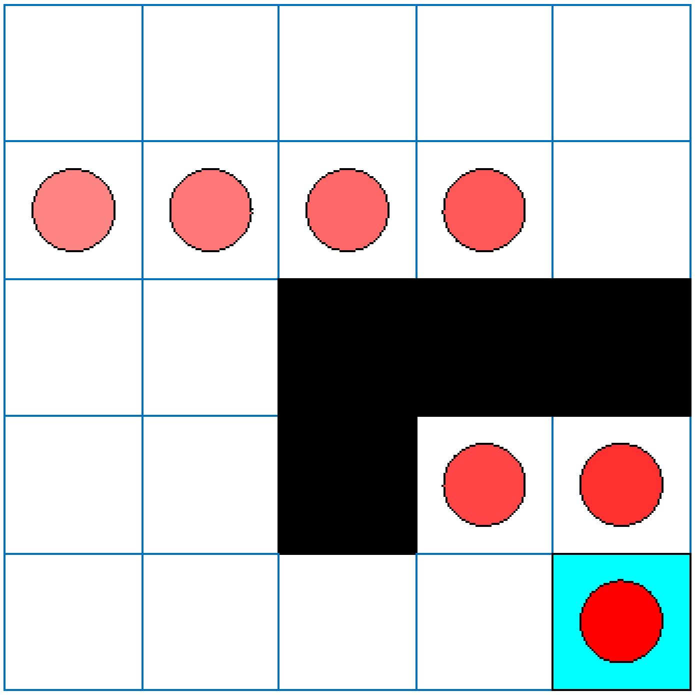
To validate the training results, simulate the agent in the training environment.

Before running the simulation, visualize the environment and configure the visualization to maintain a trace of the agent states.

```
plot(env)
env.ResetFcn = @() 2;
env.Model.Viewer.ShowTrace = true;
env.Model.Viewer.clearTrace;
```

Simulate the agent in the environment using the `sim` function.

```
sim(qAgent,env)
```



Input Arguments

`agents` — Agents
agent object | array of agent objects

Agents to train, specified as a reinforcement learning agent object, such as `rlACAgent` or `rlDDPGAgent`, or as an array of such objects.

If `env` is a multi-agent environment created with `rlSimulinkEnv`, specify agents as an array. The order of the agents in the array must match the agent order used to create `env`. Multi-agent training is not supported for MATLAB environments.

Note `train` updates the agents as training progresses. This is possible because each agent is an handle object. To preserve the original agent parameters for later use, save the agent to a MAT-file (if you copy the agent into a new variable, the new variable will also always point to the most recent agent version with updated parameters). For more information about handle objects, see "Handle Object Behavior".

Note When training terminates, `agents` reflects the state of each agent at the end of the final training episode. The rewards obtained by the final agents are not necessarily the highest achieved during the training process, due to continuous exploration. To save agents during training, create an `rlTrainingOptions` object specifying the SaveAgentCriteria and SaveAgentValue properties and pass it to `train` as a `trainOpts` argument.

For more information about how to create and configure agents for reinforcement learning, see "Reinforcement Learning Agents".

env — Environment

reinforcement learning environment object

Environment in which the agents act, specified as one of the following kinds of reinforcement learning environment object:

- A predefined MATLAB or Simulink environment created using `rlPredefinedEnv`. This kind of environment does not support training multiple agents at the same time.
- A custom MATLAB environment you create with functions such as `rlFunctionEnv` or `rlCreateEnvTemplate`. This kind of environment does not support training multiple agents at the same time.
- A custom Simulink environment you create using `rlSimulinkEnv`. This kind of environment supports training multiple agents at the same time.

For more information about creating and configuring environments, see:

- "Create MATLAB Reinforcement Learning Environments"
- "Create Simulink Reinforcement Learning Environments"

When `env` is a Simulink environment, calling `train` compiles and simulates the model associated with the environment.

trainOpts — Training parameters and options

`rlTrainingOptions` object | `rlMultiAgentTrainingOptions` object

Training parameters and options, specified as either an `rlTrainingOptions` or an `rlMultiAgentTrainingOptions` object. Use this argument to specify parameters and options such as:

- Criteria for ending training
- Criteria for saving candidate agents
- How to display training progress
- Options for parallel computing

For details, see `rlTrainingOptions` and `rlMultiAgentTrainingOptions`.

prevTrainStats — Training episode data

`rlTrainingResult` object | array of `rlTrainingResult` objects

Training episode data, specified as an:

- `rlTrainingResult` object, when training a single agent.
- Array of `rlTrainingResult` objects when training multiple agents.

Use this argument to resume training from the exact point at which it stopped. This starts the training from the last values of the agent parameters and training results object obtained after the previous `train` function call. `prevTrainStats` contains, as one of its properties, the `rlTrainingOptions` object or the `rlMultiAgentTrainingOptions` object specifying the training option set. Therefore, to restart the training with updated training options, first change the training options in `trainResults` using dot notation. If the maximum number of episodes was already reached in the previous training session, you must increase the maximum number of episodes.

For details about the `rlTrainingResult` object properties, see the “trainStats” on page 2-0 output argument.

`lgr` — Logger object
`FileLogger` object | `MonitorLogger` object

Logger object, specified either as a `FileLogger` or as a `MonitorLogger` object. For more information on reinforcement logger objects, see `rlDataLogger`.

Output Arguments

`trainStats` — Training episode data
`rlTrainingResult` object | array of `rlTrainingResult` objects

Training episode data, returned as an:

- `rlTrainingResult` object, when training a single agent.
- Array of `rlTrainingResult` objects when training multiple agents.

The following properties pertain to the `rlTrainingResult` object:

`EpisodeIndex` — Episode numbers
`[1;2;…;N]`

Episode numbers, returned as the column vector `[1;2;…;N]`, where `N` is the number of episodes in the training run. This vector is useful if you want to plot the evolution of other quantities from episode to episode.

`EpisodeReward` — Reward for each episode
column vector

Reward for each episode, returned in a column vector of length `N`. Each entry contains the reward for the corresponding episode.

`EpisodeSteps` — Number of steps in each episode
column vector

Number of steps in each episode, returned in a column vector of length `N`. Each entry contains the number of steps in the corresponding episode.

`AverageReward` — Average reward over the averaging window

column vector

Average reward over the averaging window specified in `trainOpts`, returned as a column vector of length `N`. Each entry contains the average award computed at the end of the corresponding episode.

`TotalAgentSteps` — Total number of steps

column vector

Total number of agent steps in training, returned as a column vector of length `N`. Each entry contains the cumulative sum of the entries in `EpisodeSteps` up to that point.

`EpisodeQ0` — Critic estimate of expected cumulative long-term reward for each episode

column vector

Critic estimate of expected cumulative long-term reward using the current agent and the environment initial conditions, returned as a column vector of length `N`. Each entry is the critic estimate (Q_0) for the agent of the corresponding episode. This field is present only for agents that have critics, such as `rlDDPGAgent` and `rlDQNAgent`.

`SimulationInfo` — Information collected during simulation

structure | vector of `Simulink.SimulationOutput` objects

Information collected during the simulations performed for training, returned as:

- For training in MATLAB environments, a structure containing the field `SimulationError`. This field is a column vector with one entry per episode. When the `StopOnError` option of `rlTrainingOptions` is `"off"`, each entry contains any errors that occurred during the corresponding episode.
- For training in Simulink environments, a vector of `Simulink.SimulationOutput` objects containing simulation data recorded during the corresponding episode. Recorded data for an episode includes any signals and states that the model is configured to log, simulation metadata, and any errors that occurred during the corresponding episode.

`TrainingOptions` — Training options set

`rlTrainingOptions` object | `rlMultiAgentTrainingOptions` object

Training options set, returned as:

- For a single agent, an `rlTrainingOptions` object. For more information, see `rlTrainingOptions`.
- For multiple agents, an `rlMultiAgentTrainingOptions` object. For more information, see `rlMultiAgentTrainingOptions`.

Tips

- `train` updates the agents as training progresses. To preserve the original agent parameters for later use, save the agents to a MAT-file.
- By default, calling `train` opens the Reinforcement Learning Episode Manager, which lets you visualize the progress of the training. The Episode Manager plot shows the reward for each episode, a running average reward value, and the critic estimate Q_0 (for agents that have critics). The Episode Manager also displays various episode and training statistics. To turn off the Reinforcement Learning Episode Manager, set the `Plots` option of `trainOpts` to `"none"`.

- If you use a predefined environment for which there is a visualization, you can use `plot(env)` to visualize the environment. If you call `plot(env)` before training, then the visualization updates during training to allow you to visualize the progress of each episode. (For custom environments, you must implement your own `plot` method.)
- Training terminates when the conditions specified in `trainOpts` are satisfied. To terminate training in progress, in the Reinforcement Learning Episode Manager, click **Stop Training**. Because `train` updates the agent at each episode, you can resume training by calling `train(agent,env,trainOpts)` again, without losing the trained parameters learned during the first call to `train`.
- During training, you can save candidate agents that meet conditions you specify with `trainOpts`. For instance, you can save any agent whose episode reward exceeds a certain value, even if the overall condition for terminating training is not yet satisfied. `train` stores saved agents in a MAT-file in the folder you specify with `trainOpts`. Saved agents can be useful, for instance, to allow you to test candidate agents generated during a long-running training process. For details about saving criteria and saving location, see `rlTrainingOptions`.

Algorithms

In general, `train` performs the following iterative steps:

1. Initialize `agent`.
2. For each episode:
   - a. Reset the environment.
   - b. Get the initial observation s_0 from the environment.
   - c. Compute the initial action $a_0 = \mu(s_0)$.
   - d. Set the current action to the initial action ($a \leftarrow a_0$) and set the current observation to the initial observation ($s \leftarrow s_0$).
   - e. While the episode is not finished or terminated:
     - i. Step the environment with action a to obtain the next observation s' and the reward r.
     - ii. Learn from the experience set (s,a,r,s').
     - iii. Compute the next action $a' = \mu(s')$.
     - iv. Update the current action with the next action ($a \leftarrow a'$) and update the current observation with the next observation ($s \leftarrow s'$).
     - v. Break if the episode termination conditions defined in the environment are met.
3. If the training termination condition defined by `trainOpts` is met, terminate training. Otherwise, begin the next episode.

The specifics of how `train` performs these computations depends on your configuration of the agent and environment. For instance, resetting the environment at the start of each episode can include randomizing initial state values, if you configure your environment to do so.

Version History

Introduced in R2019a

R2022a: `train` now returns an object instead of a structure

Behavior change in future release

Starting in R2022a, `train` returns an object or an array of objects instead of a structure. The properties of the object match the fields of the structure returned in previous versions. Therefore, the code based on dot notation works in the same way.

For example, if you train an agent using the following command:

```
trainStats = train(agent,env,trainOptions);
```

When training terminates, either because a termination condition is reached or because you click **Stop Training** in the Reinforcement Learning Episode Manager, `trainStats` is returned as an `rlTrainingResult` object.

The `rlTrainingResult` object contains the same training statistics previously returned in a structure along with data to correctly recreate the training scenario and update the episode manager.

You can use `trainStats` as third argument for another `train` call, which (when executed with the same agents and environment) will cause training to resume from the exact point at which it stopped.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To train in parallel, set the `UseParallel` and `ParallelizationOptions` options in the option set `trainOpts`. Parallel training is not supported for multi-agent environments. For more information, see `rlTrainingOptions`.

See Also

Functions

`sim` | `rlDataLogger` | `trainFromData`

Objects

`rlTrainingOptions` | `rlMultiAgentTrainingOptions` | `FileLogger` | `MonitorLogger`

Topics

"Train Reinforcement Learning Agents"
"Reinforcement Learning Agents"

trainFromData

Package: rl.agent

Train off-policy reinforcement learning agent using existing data

Syntax

```
tfdStats = trainFromData(agent)
tfdStats = trainFromData(agent,dataStore)
tfdStats = trainFromData( ___ ,tfdOpts)
tfdStats = trainFromData( ___ ,logger=lgr)
```

Description

`tfdStats = trainFromData(agent)` trains the off-policy agent `agent` offline, using data stored in its `ExperienceBuffer` property. Note that `agent` is an handle object and it is updated during training, despite being an input argument.

`tfdStats = trainFromData(agent,dataStore)` trains the off-policy agent `agent` offline, using data stored according to the `FileDataStore` object `dataStore`.

`tfdStats = trainFromData( ___ ,tfdOpts)` also specifies nondefault training options using the `rlTrainingFromDataOptions` object `trainFDOpts`.

`tfdStats = trainFromData( ___ ,logger=lgr)` logs training data using the `FileLogger` object `lgr`.

Examples

Train Agent From Data Collected by Training Another Agent

To collect training data, first, create an environment.

```
env = rlPredefinedEnv("CartPole-Discrete");
```

Create a built-in PPO agent with default networks.

```
agent1 = rlPPOAgent( ...
    getObservationInfo(env), ...
    getActionInfo(env));
```

Create a `FileLogger` object.

```
flgr = rlDataLogger;
```

To log the experiences on disk, assign an appropriate logger function to the logger object. This function is automatically called by the training loop at the end of each episode, and is defined at the end of the example.

```
flgr.EpisodeFinishedFcn = @myEpisodeFinishedFcn;
```

Define a training option object to train agent1 for no more than 100 epochs, without visualizing any training progress.

```
tOpts = rlTrainingOptions(MaxEpisodes=100,Plots="none");
```

Train agent1, logging the experience data.

```
train(agent1,env,tOpts,Logger=flgr);
```

At the end of this training, files containing experience data for each episode are saved in the logs folder.

Note that the only purpose of training agent1 is to collect experience data from the environment. Collecting experiences by simulating the environment in closed loop with a controller (using a for loop), or indeed collecting a series of observations caused by random actions, would also accomplish the same result.

To allow the trainFromData function to read the experience data stored in the logs folder, create a read function that, given a file name, returns the respective experience structure. For this example, the myReadFcn function is defined at the end of the example.

Check that the function can successfully retrieve data from an episode.

```
cd logs
exp = myReadFcn("loggedData002")
```

```
exp=11×1 struct array with fields:
    NextObservation
    Observation
    Action
    Reward
    IsDone
```

```
size(cell2mat([exp.Action]))
```

```
ans = 1×2

     1    11
```

```
cd ..
```

Create a FileDataStore object using fileDatastore. Pass as arguments the name of the folder where files are stored and the read function. The read function is called automatically when the datastore is accessed for reading and is defined at the end of the example.

```
fds = fileDatastore("./logs", "ReadFcn", @myReadFcn);
```

Create a built-in DQN agent with default networks to be trained from the collected dataset.

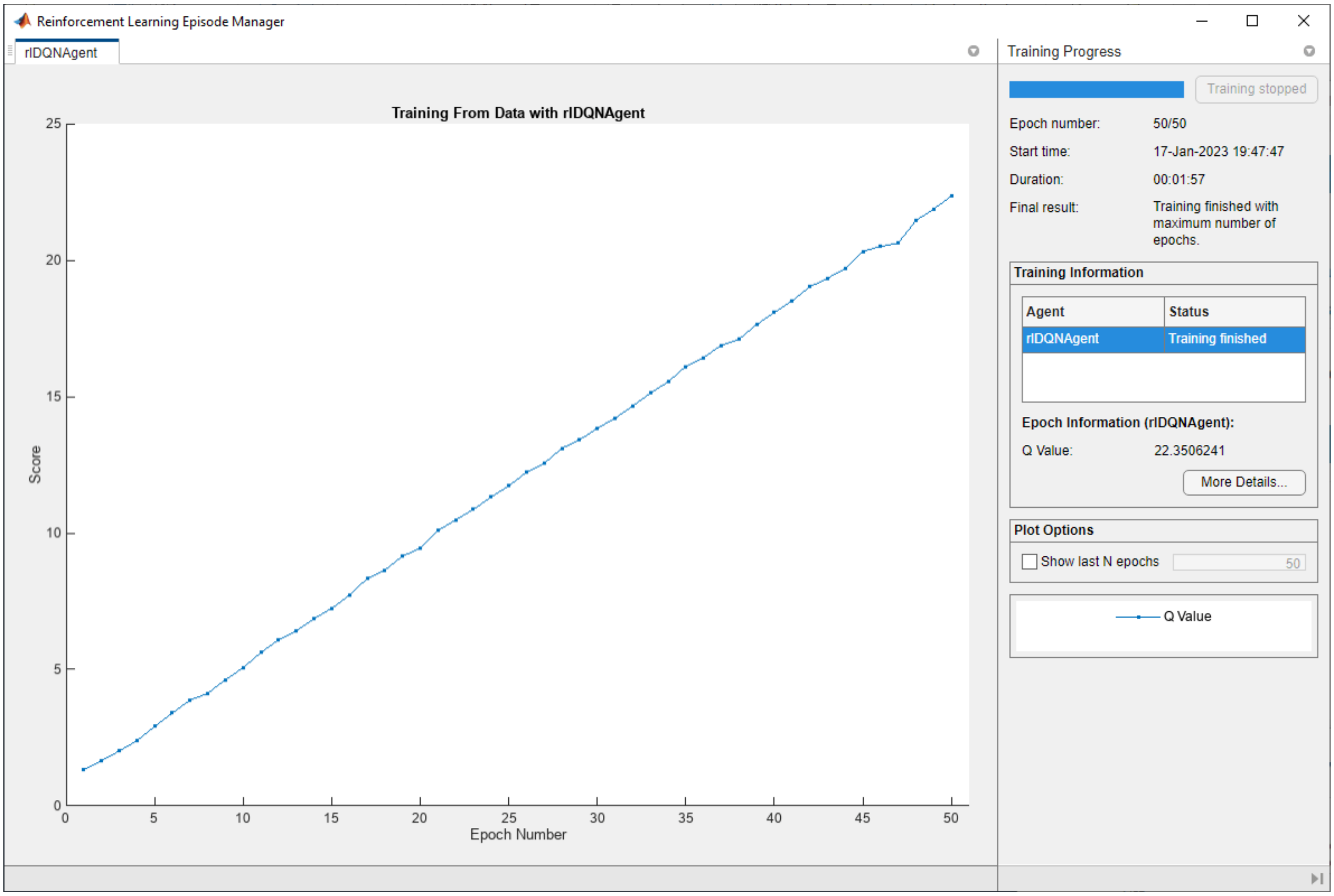
```
agent2 = rlDQNAgent( ...
    getObservationInfo(env), ...
    getActionInfo(env));
```

Define an options object to train agent2 from data for 50 epochs.

```
tfdOpts = rlTrainingFromDataOptions("MaxEpochs",50);
```

To train agent2 from data, use trainFromData. Pass the fileDataStore object fds as second input argument.

```
trainFromData(agent2,fds,tfdOpts);
```



Epoch number: 50/50

Start time: 17-Jan-2023 19:47:47

Duration: 00:01:57

Final result: Training finished with maximum number of epochs.

Training Information

Agent	Status
rlDQNAgent	Training finished

Epoch Information (rlDQNAgent):

Q Value: 22.3506241

Plot Options

□ Show last N epochs 50

Here, the estimated Q-value seems to grow indefinitely over time. This often happens during offline training because the agent updates its estimated Q-value based on the current estimated Q-value, without using any environment feedback. To prevent the Q-value from becoming increasingly large (and inaccurate) over time, stop the training earlier or use data regularizer options such as rlConservativeQLearningOptions (for DQN or SAC agents) or rlBehaviorCloningRegularizerOptions (for DDPG, TD3 or SAC agents).

In general, the Q-value calculated as above for an agent trained offline is not necessarily indicative of the performance of the agent within an environment. Therefore, best practice is to validate the agent within an environment after offline training.

Support functions

The data logging function. This function is automatically called by the training loop at the end of each episode, and must return a structure containing the data to log, such as experiences, simulation information, or initial observations. Here, data is a structure that contains the following fields:

- EpisodeCount — current episode number

- Environment — environment object
- Agent — agent object
- Experience — structure array containing the experiences. Each element of this array corresponds to a step and is a structure containing the fields `NextObservation`, `Observation`, `Action`, `Reward` and `IsDone`.
- Agent — agent object
- EpisodeInfo — structure containing the fields CumulativeReward, StepsTaken and InitialObservation.
- SimulationInfo — contains simulation information from the episode. For MATLAB environments this is a structure with the field SimulationError, and for Simulink fallback for tm_simulink environments it is a Simulink.SimulationOutput object.

```
function dataToLog = myEpisodeFinishedFcn(data)
    dataToLog.Experience = data.Experience;
end
```

For more information on logging data on disk, see `FileLogger`.

The data store read function. This function is automatically called by the training loop when the data store is accessed for reading. It must take a filename and return the experience structure array. Each element of this array corresponds to a step and is a structure containing the fields `NextObservation`, `Observation`, `Action`, `Reward` and `IsDone`.

```
function experiences = myReadFcn(fileName)

if contains(fileName,"loggedData")
    data = load(fileName);
    experiences = data.episodeData.Experience{1};
else
    experiences = [];
end

end
```

Input Arguments

agent — Off-policy agent

`rlDQNAgent` object | `rlDDPGAgent` object | `rlTD3Agent` object | `rlSACAgent` object | `rlMBPOAgent` object

Off-policy agent to train, specified as a reinforcement learning agent object, such as an `rlSACAgent` object.

Note `trainFromData` updates the agent as training progresses. For more information on how to preserve the original agent, how to save an agent during training, and on the state of `agent` after training, see the notes and the tips section in `train`. For more information about handle objects, see “Handle Object Behavior”.

For more information about how to create and configure agents for reinforcement learning, see “Reinforcement Learning Agents”.

dataStore — Data store
FileDataStore object

Data store, specified as a FileDataStore. The function specified in the ReadFcn property of dataStore must return a structure array of experiences with the Observation, Action, Reward, NextObservation, and IsDone fields. The dimensions of the arrays in Observation and NextObservation in each experience must be the same as the dimensions specified in the ObservationInfo of agent. The dimension of the array in Action must be the same as the dimension specified in the ActionInfo of agent. The Reward and IsDone fields must contain scalar values. For more information, see fileDatastore.

tfdOpts — Training from data parameters and options
rlTrainingFromDataOptions object

Training from data parameters and options, specified as an rlTrainingFromDataOptions object. Use this argument to specify parameters and options such as:

- Number of epochs
- Number of steps for each epochs
- Criteria for saving candidate agents
- How to display training progress

Note trainFromData does not support parallel computing.

For details, see rlTrainingFromDataOptions.

lgr — Logger object
FileLogger object | MonitorLogger object

Logger object, specified either as a FileLogger or as a MonitorLogger object. For more information on reinforcement logger objects, see rlDataLogger.

Output Arguments

tfdStats — Training results
rlTrainingFromDataResult object

Training results, returned as an rlTrainingFromDataResult object, which has the following properties:

EpochIndex — Epoch numbers
[1;2;…;N]

Epoch numbers, returned as the column vector [1;2;…;N], where N is the number of epochs in the training run. This vector is useful if you want to plot the evolution of other quantities from epoch to epoch.

EpochSteps — Number of steps in each epoch
column vector

Number of steps in each epoch, returned as a column vector of length N. Each entry contains the number of steps in the corresponding epoch.

TotalSteps — Total number of steps

column vector

Total number of agent steps in training, returned as a column vector of length N. Each entry contains the cumulative sum of the entries in EpochSteps up to that point.

QValue — Q-value estimates for each epoch

column vector

Q-value estimates for each epoch, returned as a column vector of length N. Each element is the average Q-value of the policy, over the observations specified in the QValueObservations property of tfdOpts, evaluated at the end of the epoch, and using the policy parameters at the end of the epoch.

Note During offline training, the agent updates its estimated Q-value based on the current estimated Q-value (without any environment feedback). As a result, the estimated Q-value can become inaccurate (and often increasingly large) over time. To prevent the Q-value from growing indefinitely, stop the training earlier or use data regularizer options. For more information, see rlBehaviorCloningRegularizerOptions and rlConservativeQLearningOptions.

Note The Q-value calculated as above for an agent trained offline is not indicative of the performance of the agent within an environment. Therefore, it is good practice to validate the agent within an environment after offline training.

TrainingOptions — Training options set

rlTrainingFromDataOptions object

Training options set, returned as an rlTrainingFromDataOptions object.

Version History

Introduced in R2023a

See Also

Functions

train | sim | fileDatastore | rlDataLogger

Objects

rlTrainingFromDataOptions | rlBehaviorCloningRegularizerOptions | rlConservativeQLearningOptions | rlReplayMemory | FileLogger

Topics

“Train Reinforcement Learning Agents”
“Reinforcement Learning Agents”
“Getting Started with Datastore”

validateEnvironment

Package: `rl.env`

Validate custom reinforcement learning environment

Syntax

```
validateEnvironment(env)
```

Description

`validateEnvironment(env)` validates a reinforcement learning environment. This function is useful when:

- You are using a custom environment for which you supplied your own step and reset functions, such as an environment created using `rlCreateEnvTemplate`.
- You are using an environment you created from a Simulink model using `rlSimulinkEnv`.

`validateEnvironment` resets the environment, generates an initial observation and action, and simulates the environment for one or two steps (see “Algorithms” on page 2-308). If there are no errors during these operations, validation is successful, and `validateEnvironment` returns no result. If errors occur, these errors appear in the MATLAB command window. Use the errors to determine what to change in your observation specification, action specification, custom functions, or Simulink model.

Examples

Validate Simulink Environment

This example shows how to validate a Simulink environment.

Create and validate and environment for the `rlwatertank` model, which represents a control system containing a reinforcement learning agent (For details about this model, see “Create Simulink Environment and Train Agent”.)

```
open_system("rlwatertank")
```

Create observation and action specifications for the environment.

```
obsInfo = rlNumericSpec([3 1],...
    LowerLimit=[-inf -inf 0  ]',...
    UpperLimit=[ inf  inf inf]');
obsInfo.Name = "observations";
obsInfo.Description = 'integrated error, error, and measured height';
numObservations = obsInfo.Dimension(1);

actInfo = rlNumericSpec([1 1]);
actInfo.Name = "flow";
numActions = numel(actInfo);
```

Create an environment from the model.

```
env = rlSimulinkEnv("rlwatertank","rlwatertank/RL Agent",obsInfo,actInfo);
```

Now you use `validateEnvironment` to check whether the model is configured correctly.

```
validateEnvironment(env)
```

```
Error using rl.env.SimulinkEnvWithAgent/validateEnvironment (line 187)
Simulink environment validation requires an agent in the MATLAB base workspace
or in a data dictionary linked to the model. Specify the agent in the Simulink model.
```

`validateEnvironment` attempts to compile the model, initialize the environment and the agent, and simulate the model. In this case, the RL Agent block is configured to use an agent called `agent`, but no such variable exists in the MATLAB® workspace. Thus, the function returns an error indicating the problem.

Create an appropriate agent for this system using the commands detailed in the “Create Simulink Environment and Train Agent” example. In this case, load the agent from the `rlWaterTankDDPGAgent.mat` file.

```
load rlWaterTankDDPGAgent
```

Now, run `validateEnvironment` again.

```
validateEnvironment(env)
```

Input Arguments

env — Environment to validate
environment object

Environment to validate, specified as a reinforcement learning environment object, such as:

- A custom MATLAB environment you create with `rlCreateEnvTemplate`. In this case, `validateEnvironment` checks that the observations and actions generated during simulation of the environment are consistent in size, data type, and value range with the observation specification and action specification. It also checks that your custom `step` and `reset` functions run without error. (When you create a custom environment using `rlFunctionEnv`, the software runs `validateEnvironment` automatically.)
- A custom Simulink environment you create using `rlSimulinkEnv`. If you use a Simulink environment, you must also have an agent defined and associated with the RL Agent block in the model. For a Simulink model, `validateEnvironment` checks that the model compiles and runs without error. The function does not dirty your model.

For more information about creating and configuring environments, see:

- “Create MATLAB Reinforcement Learning Environments”
- “Create Simulink Reinforcement Learning Environments”

Algorithms

validateEnvironment works by running a brief simulation of the environment and making sure that the generated signals match the observation and action specifications you provided when you created the environment.

MATLAB Environments

For MATLAB environments, validation includes the following steps.

1. Reset the environment using the reset function associated with the environment.
2. Obtain the first observation and check whether it is consistent with the dimension, data type, and range of values in the observation specification.
3. Generate a test action based on the dimension, data type, and range of values in the action specification.
4. Simulate the environment for one step using the generated action and the step function associated with the environment.
5. Obtain the new observation signal and check whether it is consistent with the dimension, data type, and range of values in the observation specification.

If any of these operations generates an error, validateEnvironment returns the error. If validateEnvironment returns no result, then validation is successful.

Simulink Environments

For Simulink environments, validation includes the following steps.

1. Reset the environment.
2. Simulate the model for two time steps.

If any of these operations generates an error, validateEnvironment returns the error. If validateEnvironment returns no result, then validation is successful.

validateEnvironment performs these steps without dirtying the model, and leaves all model parameters in the state they were in when you called the function.

Version History

Introduced in R2019a

See Also

Functions

rlCreateEnvTemplate | rlSimulinkEnv

Objects

rlFunctionEnv

Topics

“Create Simulink Environment and Train Agent”
“Create Custom MATLAB Environment from Template”

validateExperience

Package: `rl.replay`

Validate experiences for replay memory

Syntax

```
validateExperience(buffer,experience)
```

Description

`validateExperience(buffer,experience)` validates whether the experiences in `experience` are compatible with replay memory `buffer`. If the experiences are not compatible with the replay memory, `validateExperience` generates an error message in the MATLAB command window.

Examples

Create Experience Buffer

Define observation specifications for the environment. For this example, assume that the environment has a single observation channel with three continuous signals in specified ranges.

```
obsInfo = rlNumericSpec([3 1],...
    LowerLimit=0,...
    UpperLimit=[1;5;10]);
```

Define action specifications for the environment. For this example, assume that the environment has a single action channel with two continuous signals in specified ranges.

```
actInfo = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[5;10]);
```

Create an experience buffer with a maximum length of 20,000.

```
buffer = rlReplayMemory(obsInfo,actInfo,20000);
```

Append a single experience to the buffer using a structure. Each experience contains the following elements: current observation, action, next observation, reward, and is-done.

For this example, create an experience with random observation, action, and reward values. Indicate that this experience is not a terminal condition by setting the `IsDone` value to 0.

```
exp.Observation = {obsInfo.UpperLimit.*rand(3,1)};
exp.Action = {actInfo.UpperLimit.*rand(2,1)};
exp.Reward = 10*rand(1);
exp.NextObservation = {obsInfo.UpperLimit.*rand(3,1)};
exp.IsDone = 0;
```

Before appending experience to the buffer, you can validate whether the experience is compatible with the buffer. The `validateExperience` function generates an error if the expereince is incompatible with the buffer.

```
validateExperience(buffer,exp)
```

Append the experience to the buffer.

```
append(buffer,exp);
```

You can also append a batch of experiences to the experience buffer using a structure array. For this example, append a sequence of 100 random experiences, with the final experience representing a terminal condition.

```
for i = 1:100
    expBatch(i).Observation = {obsInfo.UpperLimit.*rand(3,1)};
    expBatch(i).Action = {actInfo.UpperLimit.*rand(2,1)};
    expBatch(i).Reward = 10*rand(1);
    expBatch(i).NextObservation = {obsInfo.UpperLimit.*rand(3,1)};
    expBatch(i).IsDone = 0;
end
expBatch(100).IsDone = 1;

validateExperience(buffer,expBatch)

append(buffer,expBatch);
```

After appending experiences to the buffer, you can sample mini-batches of experiences for training of your RL agent. For example, randomly sample a batch of 50 experiences from the buffer.

```
miniBatch = sample(buffer,50);
```

You can sample a horizon of data from the buffer. For example, sample a horizon of 10 consecutive experiences with a discount factor of 0.95.

```
horizonSample = sample(buffer,1,...
    NStepHorizon=10,...
    DiscountFactor=0.95);
```

The returned sample includes the following information.

- `Observation` and `Action` are the observation and action from the first experience in the horizon.
- `NextObservation` and `IsDone` are the next observation and termination signal from the final experience in the horizon.
- `Reward` is the cumulative reward across the horizon using the specified discount factor.

You can also sample a sequence of consecutive experiences. In this case, the structure fields contain arrays with values for all sampled experiences.

```
sequenceSample = sample(buffer,1,...
    SequenceLength=20);
```

Input Arguments

`buffer` — Experience buffer
`rlReplayMemory` object | `rlPrioritizedReplayMemory` object | `rlHindsightReplayMemory` object | `rlHindsightPrioritizedReplayMemory` object

Experience buffer, specified as one of the following replay memory objects.

- `rlReplayMemory`
- `rlPrioritizedReplayMemory`
- `rlHindsightReplayMemory`
- `rlHindsightPrioritizedReplayMemory`

`experience` — Experience to append to the buffer
structure | structure array

Experience to append to the buffer, specified as a structure or structure array with the following fields.

`Observation` — Observation
cell array

Observation, specified as a cell array with length equal to the number of observation specifications specified when creating the buffer. The dimensions of each element in `Observation` must match the dimensions in the corresponding observation specification.

`Action` — Agent action
cell array

Action taken by the agent, specified as a cell array with length equal to the number of action specifications specified when creating the buffer. The dimensions of each element in `Action` must match the dimensions in the corresponding action specification.

`Reward` — Reward value
scalar

Reward value obtained by taking the specified action from the starting observation, specified as a scalar.

`NextObservation` — Next observation
cell array

Next observation reached by taking the specified action from the starting observation, specified as a cell array with the same format as `Observation`.

`IsDone` — Termination signal
`0` | `1` | `2`

Termination signal, specified as one of the following values.

- `0` — This experience is not the end of an episode.
- `1` — The episode terminated because the environment generated a termination signal.
- `2` — The episode terminated by reaching the maximum episode length.

Version History

Introduced in R2023a

See Also

Objects

`rlReplayMemory` | `rlPrioritizedReplayMemory` | `rlHindsightReplayMemory` | `rlHindsightPrioritizedReplayMemory`

wait

Package: rl.env

Wait for reinforcement learning environment simulations running on a workers to finish

Syntax

```
wait(F)
```

Description

wait(F) blocks the command prompt and waits for all elements of F (each corresponding to a simulation scheduled on a worker) to reach a finished state.

Examples

Defer Outputs of Simulations Running on Workers

This example shows how to use Future objects and their methods fetchNext, fetchOutput, cancel, and wait to defer output retrieval for environment simulations running on workers, monitor the status of ongoing simulations, fetch outputs of completed simulations, cancel ongoing simulations, or wait for ongoing simulations to complete.

Load a predefined environment and a suitable agent. For this example use both the environment and agent described in “Train AC Agent to Balance Cart-Pole System”.

```
env = rlPredefinedEnv("CartPole-Discrete");
load("MATLABCartpoleAC.mat","agent")
```

Start a parallel pool and set up the environment so that it simulates on workers.

```
pp = parpool(2);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 2 workers.
```

```
setup(env,UseParallel=true);
```

To display the simulation completion times, start a timer.

```
tic
```

Schedule six simulation to run on the available workers. At the beginning of the simulation, the reset function of the cart-pole environment sets the initial angle of the pole to a random position in the neighborhood of zero (the upward position). This randomization ensures that each simulation is different.

```
for i=1:6
    ftr(i) = runEpisode(env,agent,CleanupPostSim=false);
end
```

Each element of the `Future` array `ftr` represents a scheduled simulation.

```
ftr
```

```
ftr=1×6 object
  1×6 Future array with properties:

    Read
    State
    Diary
    ID
```

Display the state of each simulation.

```
ftr.State
```

```
ans =
'running'

ans =
'running'

ans =
'queued'

ans =
'queued'

ans =
'queued'

ans =
'queued'
```

Two simulations are ongoing while the others are queued.

Use `fetchNext` with a timeout of `0.1` seconds to retrieve results for simulations that complete within that time (if any).

```
[idx,out] = fetchNext(ftr,0.1)
```

```
idx =

     []


out =

     []
```

Both the outputs are empty, which means that none of the four simulation has completed yet.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   0
```

```
ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

Use `fetchNext` without any timeout to wait until an unretrieved simulation output becomes available and then return the results.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 2

out = struct with fields:
    SimulationInfo: [1×1 struct]
         AgentData: [1×1 struct]
```

Display the state of the simulations.

```
ftr.State
```

```
ans =
'finished'

ans =
'finished'

ans =
'running'

ans =
'running'

ans =
'queued'

ans =
'queued'
```

As expected, the first two simulations, which were running in parallel on the two workers, are finished, while the next two, which were previously queued, are now running, and the final two are still queued.

Display the time taken for the first two simulations to complete.

```
toc
```

```
Elapsed time is 10.451231 seconds.
```

Note that once the results from a simulation has been already retrieved, any attempt to use `fetchNext` to retrieve it again, such as in `fetchNext(ftr(2))`, will result in an error. To retrieve the results from a `Future` object that has already been read, you can use `fetchOuptuts`, such as in `fetchOutputs(ftr(2))`.

Retrieve the next available result, and display the time elapsed since the simulations started.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 1

out = struct with fields:
    SimulationInfo: [1×1 struct]
         AgentData: [1×1 struct]
```

```
toc
```

```
Elapsed time is 11.945070 seconds.
```

As expected, `fetchNext` promptly returns the results from the second simulation, since it was already available.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   1


ans = logical
   1


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

Cancel the last simulation.

```
cancel(ftr(6))
```

Wait for the fourth simulation to complete. The `wait` function blocks the command prompt until the fourth simulation is completed.

```
wait(ftr(4))
```

Display the elapsed time since the simulations started.

```
toc
```

```
Elapsed time is 12.414076 seconds.
```

Display the state of the simulations.

```
ftr.State
```

```
ans =
'finished'

ans =
'finished'

ans =
'finished'

ans =
'finished'

ans =
'running'

ans =
'finished'
```

The status of the last element of the array, for which the simulation has been canceled, is classified as `'finished'`.

Since any attempt to retrieve results from a simulation that has been canceled will result in an error, remove the canceled object from the array.

```
ftr(6)=[]
```

```
ftr=1×5 object
  1×5 Future array with properties:

    Read
    State
    Diary
    ID
```

Use `fetchOutputs` to wait until all remaining simulations are completed and then retrieve all outputs.

```
outs = fetchOutputs(ftr)
```

```
outs=5×1 struct array with fields:
    SimulationInfo
    AgentData
```

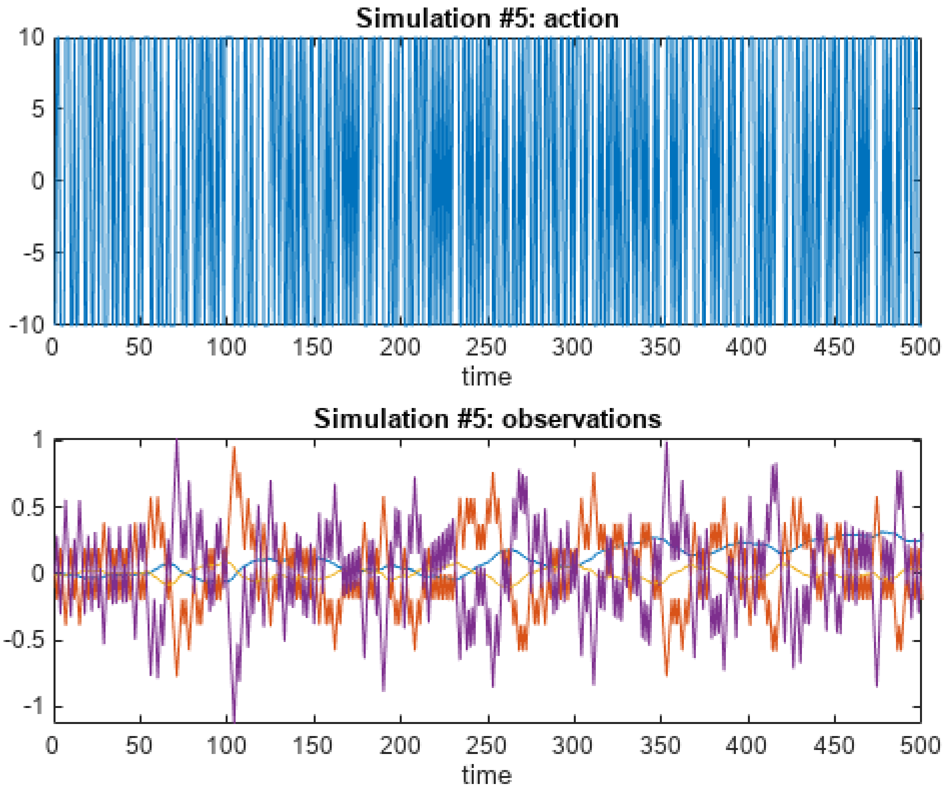
Display the elapsed time.

```
toc
```

```
Elapsed time is 16.265069 seconds.
```

Plot the action and observations from the fifth simulation.

```
figure

subplot(2,1,1);
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Action]))
title('Simulation #5: action');
xlabel('time');

subplot(2,1,2)
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Observation]))
title('Simulation #5: observations')
xlabel('time');
```



Clear the array of `Future` objects, the environment, and delete the parallel pool (this is the reverse order in which they were created).

```
clear ftr
clear env
delete(pp)
```

Input Arguments

F — Future simulation outputs

Future object | array of Future objects

Future simulation outputs, specified as a Future object or as an array of Future objects. To create an element of F, set the UseParallel property of a reinforcement learning environment to true, and then use runEpisode to simulate an agent or a policy within this environment. Assign the element of F to the output of runEpisode.

Version History

Introduced in R2022a

See Also

Objects

Future | Future | Simulink.Simulation.Future

Functions

fetchNext | fetchOutputs | cancel | runEpisode | setup | cleanup | reset

Topics

"Custom Training Loop with Simulink Action Noise"
"Train Reinforcement Learning Policy Using Custom Training Loop"
"Train Agents Using Parallel Computing and GPUs"

write

Package: rl.logging

Transfer stored data from the internal logger memory to the logging target

Syntax

```
write(lgr)
```

Description

write(lgr) transfers all stored logging contexts into the logger's target (either a MAT file or a trainingProgressMonitor object).

Examples

Log Data to Disk in a Custom Training Loop

This example shows how to log data to disk when training an agent using a custom training loop.

Create a FileLogger object using rlDataLogger.

```
flgr = rlDataLogger();
```

Set up the logger object. This operation initializes the object performing setup tasks such as, for example, creating the directory to save the data files.

```
setup(flgr);
```

Within a custom training loop, you can now store data to the logger object memory and write data to file.

For this example, store random numbers to the file logger object, grouping them in the variables Context1 and Context2. When you issue a write command, a MAT file corresponding to an iteration and containing both variables is saved with the name specified in flgr.LoggingOptions.FileNameRule, in the folder specified by flgr.LoggingOptions.LoggingDirectory.

```
for iter = 1:10

    % Store three random numbers in memory
    % as elements of the variable "Context1"
    for ct = 1:3
        store(flgr, "Context1", rand, iter);
    end

    % Store a random number in memory
    % as the variable "Context2"
    store(flgr, "Context2", rand, iter);
    % Write data to file every 4 iterations
    if mod(iter,4)==0
        write(flgr);
    end

end
```

Clean up the logger object. This operation performs clean up tasks like for example writing to file any data still in memory.

```
cleanup(flgr);
```

Input Arguments

lgr — Date logger object

FileLogger object | MonitorLogger object | ...

Data logger object, specified as either a FileLogger or a MonitorLogger object.

Version History

Introduced in R2022b

See Also

Functions

rlDataLogger | train | setup | store | cleanup

Objects

FileLogger | MonitorLogger

Topics

"Log Training Data to Disk"
"Function Handles"
"Handle Object Behavior"

3

Objects

quadraticLayer

Quadratic layer for actor or critic network

Description

A quadratic layer takes an input vector and outputs a vector of quadratic monomials constructed from the input elements. This layer is useful when you need a layer whose output is a quadratic function of its inputs. For example, to recreate the structure of quadratic value functions such as those used in LQR controller design.

For example, consider an input vector `U = [u1 u2 u3]`. For this input, a quadratic layer gives the output `Y = [u1*u1 u1*u2 u2*u2 u1*u3 u2*u3 u3*u3]`. For an example that uses a `QuadraticLayer`, see “Train DDPG Agent to Control Double Integrator System”.

Note The `QuadraticLayer` layer does not support inputs coming directly or indirectly from a `featureInputLayer` or `sequenceInputLayer`.

The parameters of a `QuadraticLayer` object are not learnable.

Creation

Syntax

```
qLayer = quadraticLayer
qLayer = quadraticLayer(Name,Value)
```

Description

`qLayer = quadraticLayer` creates a quadratic layer with default property values.

`qLayer = quadraticLayer(Name,Value)` sets properties on page 3-2 using name-value pairs. For example, `quadraticLayer('Name','quadlayer')` creates a quadratic layer and assigns the name `'quadlayer'`.

Properties

Name — Name of layer
`'quadratic'` (default) | character vector

Name of layer, specified as a character vector. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with this layer and `Name` is set to `''`, then the software automatically assigns a name to the layer at training time.

`Description` — Description of layer
`'quadratic layer'` (default) | character vector

This property is read-only.

Description of layer, specified as a character vector. When you create the quadratic layer, you can use this property to give it a description that helps you identify its purpose.

Examples

Create Quadratic Layer

Create a quadratic layer that converts an input vector `U` into a vector of quadratic monomials constructed from binary combinations of the elements of `U`.

```
qLayer = quadraticLayer
```

```
qLayer = 
  QuadraticLayer with properties:

    Name: 'quadratic'

   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

Confirm that the layer produces the expected output. For instance, for `U = [u1 u2 u3]`, the expected output is `[u1*u1 u1*u2 u2*u2 u1*u3 u2*u3 u3*u3]`.

```
predict(qLayer,[1 2 3])
```

```
ans = 1×3

     1     4     9
```

You can incorporate `qLayer` into an actor network or critic network for reinforcement learning.

Version History

Introduced in R2019a

See Also

Functions
`scalingLayer` | `softplusLayer`

Topics
“Train DDPG Agent to Control Double Integrator System”
“Create Policies and Value Functions”

FileLogger

Log reinforcement learning training data to MAT files

Description

Use a FileLogger object to log data to MAT files, within the train function or inside a custom training loop. To log data when using the train function, specify appropriate callback functions in FileLogger, as shown in the examples. These callbacks are executed at different stages of training, for example, EpisodeFinishedFcn is executed after the completion of an episode. The output of a callback function is a structure containing the data to log at that stage of training.

Note Using a FileLogger object to log data when using the train function does not affect (and is not affected by) any option to save agents during training specified within an rlTrainingOptions object.

Note FileLogger is a handle object. If you assign an existing FileLogger object to a new FileLogger object, both the new object and the original one refer to the same underlying object in memory. To preserve the original object parameters for later use, save the object to a MAT-file. For more information about handle objects, see “Handle Object Behavior”.

Creation

Create a FileLogger object using rlDataLogger without any input arguments.

Properties

LoggingOptions — Object containing a set of logging options
MATFileLoggingOptions object (default)

Object containing a set of logging options, returned as a MATFileLoggingOptions object. A MATFileLoggingOptions object has the following properties, which you can access using dot notation after creating the FileLogger object.

LoggingDirectory — Name of the logging directory
"logs" subdirectory of the current directory (default) | string or character array

Name or fully qualified path of the logging directory, specified as a string or a character array. This is the name of the directory where the MAT files containing the logged data are saved. As a default, a subdirectory called logs is created in the current folder during setup and files are saved there during training.

Example: LoggingDirectory=mylogs

FileNameRule — Rule to name the MAT files
"loggedData<id>" (default) | string | char array

Rule to name the MAT files, specified as a string or a character array. For example, the naming rule `"episode<id>"` results in the file names `episode001.mat`, `episode002.mat` and so on.

Example: `FileNameRule="ThirdRun<id>"`

`Version` — MAT files version

`"-v7"` (default) | `string` | `char` array

MAT file versions, specified as a string or character array. The default is "-v7". For more information, see “MAT-File Versions”.

Example: `Version="-v7.3"`

`UseCompression` — Option to use compression

`true` (default) | `false`

Option to use compression when saving data to a MAT file, specified as a logical variable. The default is `true`. For more information, see “MAT-File Versions”.

Example: `UseCompression=false`

`DataWriteFrequency` — Disk data write period

`1` (default) | positive integer

Disk data write period, specified as a positive integer. It is the number of episodes after which data is saved to disk. For example, if `DataWriteFrequency` is `5` then data from episodes 1 to 5 will be cached in memory and be written to disk at the end of the 5-th episode. This improves performance in some cases. The default is 1.

Example: `DataWriteFrequency=10`

`MaxEpisodes` — Maximum number of episodes

`500` (default) | positive integer

Maximum number of episodes, specified as a positive integer. When using `train`, the value is automatically initialized. Set this value when using the logger object in a custom training loop. The default is `500`.

Example: `MaxEpisodes=1000`

`EpisodeFinishedFcn` — Callback to log data after episode completion

`[]` (default) | function handle

Callback to log data after episode completion, specified as a function handle object. The specified function is automatically called by the training loop at the end of each episode, and must return a structure containing the data to log, such as experiences, simulation information, or initial observations.

Your function must have the following signature.

```
function dataToLog = myEpisodeFinishedFcn(data)
```

Here, `data` is a structure that contains the following fields:

- `EpisodeCount` — current episode number
- `Environment` — environment object

- `Agent` — agent object
- `Experience` — structure array containing the experiences. Each element of this array corresponds to a step and is a structure containing the fields `NextObservation`, `Observation`, `Action`, `Reward` and `IsDone`.
- `EpisodeInfo` — structure containing the fields `CumulativeReward`, `StepsTaken` and `InitialObservation`.
- `SimulationInfo` — contains simulation information from the episode. For MATLAB environments this is a structure with the field `SimulationError`, and for Simulink environments it is a `Simulink.SimulationOutput` object.

The function output `dataToLog` is the structure containing the data to be logged to disk.

Example: `EpisodeFinishedFcn=@myEpLoggingFcn`

`AgentStepFinishedFcn` — Callback to log data after training step completion

[ ] (default) | function handle | cell array of function handles

Callback to log data after training step completion within an episode, specified as a function handle object. The specified function is automatically called by the training loop at the end of each training step, and must return a structure containing the data to log, such as for example the state of the agent's exploration policy.

Your function must have the following signature.

```
function dataToLog = myAgentStepFinishedFcn(data)
```

Here, `data` is a structure that contains the following fields:

- `EpisodeCount` — current episode number
- `AgentStepCount` — cumulative number of steps taken by the agent
- `SimulationTime` — current simulation time in the environment
- `Agent` — agent object

The function output `dataToLog` is the structure containing the data to be logged to disk.

For multi agent training, `AgentStepFinishedFcn` can be a cell array of function handles with as many elements as number of agent groups.

Note Logging data using the `AgentStepFinishedFcn` callback is not supported when training agents in parallel with the train function.

Example: `AgentStepFinishedFcn=@myAgtStepLoggingFcn`

`AgentLearnFinishedFcn` — Callback to log data after completion of the learn subroutine

[ ] (default) | function handle | cell array of function handles

Callback to log data after completion of the learn subroutine, specified as a function handle object. The specified function is automatically called by the training loop at the end of each learning subroutine, and must return a structure containing the data to log, such as the training losses of the actor and critic networks, or, for a model-based agent, the environment model training losses.

Your function must have the following signature.

```
function dataToLog = myAgentLearnFinishedFcn(data)
```

Here, `data` is a structure that contains the following fields:

- `EpisodeCount` — current episode number
- `AgentStepCount` — cumulative number of steps taken by the agent
- `AgentLearnCount` — cumulative number of learning steps taken by the agent
- `EnvModelTrainingInfo` — structure containing model-based agents related fields `TransitionFcnLoss`, `RewardFcnLoss` and `IsDoneFcnLoss`.
- `Agent` — agent object
- `ActorLoss` — loss of the actor
- `CriticLoss` — loss of the critic

The function output `dataToLog` is the structure containing the data to be logged to disk.

For multi agent training, `AgentLearnFinishedFcn` can be a cell array of function handles with as many elements as number of agent groups.

Example: `AgentLearnFinishedFcn=@myAgtLearnLoggingFcn`

Object Functions

setup	Set up reinforcement learning environment or initialize data logger object
cleanup	Clean up reinforcement learning environment or data logger object

Examples

Log Data to Disk during Built-in Training

This example shows how to log data to disk when using `train`.

Create a `FileLogger` object using `rlDataLogger`.

```
logger = rlDataLogger();
```

Specify a directory to save logged data.

```
logger.LoggingOptions.LoggingDirectory = "myDataLog";
```

Create callback functions to log the data (for this example, see the helper function section), and specify the appropriate callback functions in the logger object.

```
logger.EpisodeFinishedFcn    = @myEpisodeFinishedFcn;
logger.AgentStepFinishedFcn  = @myAgentStepFinishedFcn;
logger.AgentLearnFinishedFcn = @myAgentLearnFinishedFcn;
```

To train the agent, you can now call `train`, passing `logger` as an argument such as in the following command.

```
trainResult = train(agent, env, trainOpts, Logger=logger);
```

While the training progresses, data will be logged to the specified directory, according to the rule specified in the `FileNameRule` property of `logger.LoggingOptions`.

```
logger.LoggingOptions.FileNameRule
```

```
ans = 
"loggedData<id>"
```

Example Logging Functions

Episode completion logging function. This function is automatically called by the training loop at the end of each episode, and must return a structure containing the episode-related data to log, such as experiences, simulation information, or initial observations.

```
function dataToLog = myEpisodeFinishedFcn(data)
    dataToLog.Experience = data.Experience;
end
```

Agent step completion logging function. This function is automatically called by the training loop at the end of each training step, and must return a structure containing the step-related data to log, such as for example the state of the agent exploration policy.

```
function dataToLog = myAgentStepFinishedFcn(data)
    dataToLog.State = getState(getExplorationPolicy(data.Agent));
end
```

Learn subroutine completion logging function. This function is automatically called by the training loop at the end of each learning subroutine, and must return a structure containing the learning-related data to log, such as the training losses of the actor and critic networks, or, for a model-based agent, the environment model training losses.

```
function dataToLog = myAgentLearnFinishedFcn(data)
    dataToLog.ActorLoss  = data.ActorLoss;
    dataToLog.CriticLoss = data.CriticLoss;
end
```

For the specific function signatures and more information on the function input structure, see the corresponding properties of `FileLogger`. For a related example, see “Log Training Data to Disk”.

Log Data to Disk in a Custom Training Loop

This example shows how to log data to disk when training an agent using a custom training loop.

Create a `FileLogger` object using `rlDataLogger`.

```
flgr = rlDataLogger();
```

Set up the logger object. This operation initializes the object performing setup tasks such as, for example, creating the directory to save the data files.

```
setup(flgr);
```

Within a custom training loop, you can now store data to the logger object memory and write data to file.

For this example, store random numbers to the file logger object, grouping them in the variables `Context1` and `Context2`. When you issue a write command, a MAT file corresponding to an iteration and containing both variables is saved with the name specified in flgr.LoggingOptions.FileNameRule, in the folder specified by flgr.LoggingOptions.LoggingDirectory.

```
for iter = 1:10

    % Store three random numbers in memory
    % as elements of the variable "Context1"
    for ct = 1:3
        store(flgr, "Context1", rand, iter);
    end

    % Store a random number in memory
    % as the variable "Context2"
    store(flgr, "Context2", rand, iter);

    % Write data to file every 4 iterations
    if mod(iter,4)==0
        write(flgr);
    end

end
```

Clean up the logger object. This operation performs clean up tasks like for example writing to file any data still in memory.

```
cleanup(flgr);
```

Limitations

- Logging data using the AgentStepFinishedFcn callback is not supported when training agents in parallel with the train function.

Version History

Introduced in R2022b

See Also

Functions
rlDataLogger | rlDataViewer | train | setup | store | write | cleanup

Objects
MonitorLogger | rlTrainingOptions

Topics
"Log Training Data to Disk"
"Function Handles"
"Handle Object Behavior"

Future

Object that supports deferred outputs for reinforcement learning environment simulations running on workers

Description

When `runEpisode` runs in the background it returns a `Future` object as a result. Use this object to monitor the status of ongoing simulations, fetch outputs of completed simulations, or cancel ongoing simulations.

Creation

A `Future` object is returned by `runEpisode` when the environment is configured for parallel simulations (such as in `setup(env,UseParallel=true)`).

Note that creating a `Future` object requires Parallel Computing Toolbox™ software.

Properties

Read — Flag indicating whether outputs have been read
`true` | `false`

Flag indicating whether outputs have been read, returned as a logical value. This property is read-only.

`State` — Current state of the future object array
`'pending'` | `'queued'` | `'running'` | `'finished'` | `'failed'` | `'unavailable'`

Current state of the future object array, returned as one of the following character arrays:

- `'pending'` — The simulation result is pending.
- `'queued'` — The simulation is queued.
- `'running'` — The simulation is running.
- `'finished'` — The simulation is finished.
- `'failed'` — The simulation has failed.
- `'unavailable'` — The result is not available.

This property is read-only.

`Diary` — Simulation outputs log
character array

Simulation outputs log, returned as a character array. This property is read-only.

`ID` — Numeric identifier of the object
scalar integer

Numeric identifier of the `Future` object, returned as a scalar integer. This property is read-only.

Object Functions

Examples

Defer Outputs of Simulations Running on Workers

This example shows how to use `Future` objects and their methods `fetchNext`, `fetchOutput`, `cancel`, and `wait` to defer output retrieval for environment simulations running on workers, monitor the status of ongoing simulations, fetch outputs of completed simulations, cancel ongoing simulations, or wait for ongoing simulations to complete.

Load a predefined environment and a suitable agent. For this example use both the environment and agent described in “Train AC Agent to Balance Cart-Pole System”.

```
env = rlPredefinedEnv("CartPole-Discrete");
load("MATLABCartpoleAC.mat","agent")
```

Start a parallel pool and set up the environment so that it simulates on workers.

```
pp = parpool(2);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 2 workers.
```

```
setup(env,UseParallel=true);
```

To display the simulation completion times, start a timer.

```
tic
```

Schedule six simulation to run on the available workers. At the beginning of the simulation, the reset function of the cart-pole environment sets the initial angle of the pole to a random position in the neighborhood of zero (the upward position). This randomization ensures that each simulation is different.

```
for i=1:6
    ftr(i) = runEpisode(env,agent,CleanupPostSim=false);
end
```

Each element of the `Future` array `ftr` represents a scheduled simulation.

```
ftr
```

```
ftr=1×6 object
  1×6 Future array with properties:

    Read
    State
    Diary
    ID
```

Display the state of each simulation.

```
ftr.State
```

```
ans =
'running'

ans =
'running'

ans =
'queued'

ans =
'queued'

ans =
'queued'

ans =
'queued'
```

Two simulations are ongoing while the others are queued.

Use `fetchNext` with a timeout of `0.1` seconds to retrieve results for simulations that complete within that time (if any).

```
[idx,out] = fetchNext(ftr,0.1)
```

```
idx =

     []


out =

     []
```

Both the outputs are empty, which means that none of the four simulation has completed yet.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

```
ans = logical
   0
```

Use `fetchNext` without any timeout to wait until an unretrieved simulation output becomes available and then return the results.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 2
```

```
out = struct with fields:
    SimulationInfo: [1×1 struct]
         AgentData: [1×1 struct]
```

Display the state of the simulations.

```
ftr.State
```

```
ans = 
'finished'

ans = 
'finished'

ans = 
'running'

ans = 
'running'

ans = 
'queued'

ans = 
'queued'
```

As expected, the first two simulations, which were running in parallel on the two workers, are finished, while the next two, which were previously queued, are now running, and the final two are still queued.

Display the time taken for the first two simulations to complete.

```
toc
```

```
Elapsed time is 10.451231 seconds.
```

Note that once the results from a simulation has been already retrieved, any attempt to use `fetchNext` to retrieve it again, such as in `fetchNext(ftr(2))`, will result in an error. To retrieve the results from a `Future` object that has already been read, you can use `fetchOuptuts`, such as in `fetchOutputs(ftr(2))`.

Retrieve the next available result, and display the time elapsed since the simulations started.

```
[idx,out] = fetchNext(ftr)
```

```
idx = 1
```

```
out = struct with fields:
    SimulationInfo: [1×1 struct]
```

```
         AgentData: [1×1 struct]
```

```
toc
```

```
Elapsed time is 11.945070 seconds.
```

As expected, `fetchNext` promptly returns the results from the second simulation, since it was already available.

Display how many output results have been already retrieved.

```
ftr.Read
```

```
ans = logical
   1


ans = logical
   1


ans = logical
   0


ans = logical
   0


ans = logical
   0


ans = logical
   0
```

Cancel the last simulation.

```
cancel(ftr(6))
```

Wait for the fourth simulation to complete. The `wait` function blocks the command prompt until the fourth simulation is completed.

```
wait(ftr(4))
```

Display the elapsed time since the simulations started.

```
toc
```

```
Elapsed time is 12.414076 seconds.
```

Display the state of the simulations.

```
ftr.State
```

```
ans = 
'finished'
```

```
ans = 
'finished'

ans = 
'finished'

ans = 
'finished'

ans = 
'running'

ans = 
'finished'
```

The status of the last element of the array, for which the simulation has been canceled, is classified as `'finished'`.

Since any attempt to retrieve results from a simulation that has been canceled will result in an error, remove the canceled object from the array.

```
ftr(6)=[]
```

```
ftr=1×5 object
  1×5 Future array with properties:

    Read
    State
    Diary
    ID
```

Use `fetchOutputs` to wait until all remaining simulations are completed and then retrieve all outputs.

```
outs = fetchOutputs(ftr)
```

```
outs=5×1 struct array with fields:
    SimulationInfo
    AgentData
```

Display the elapsed time.

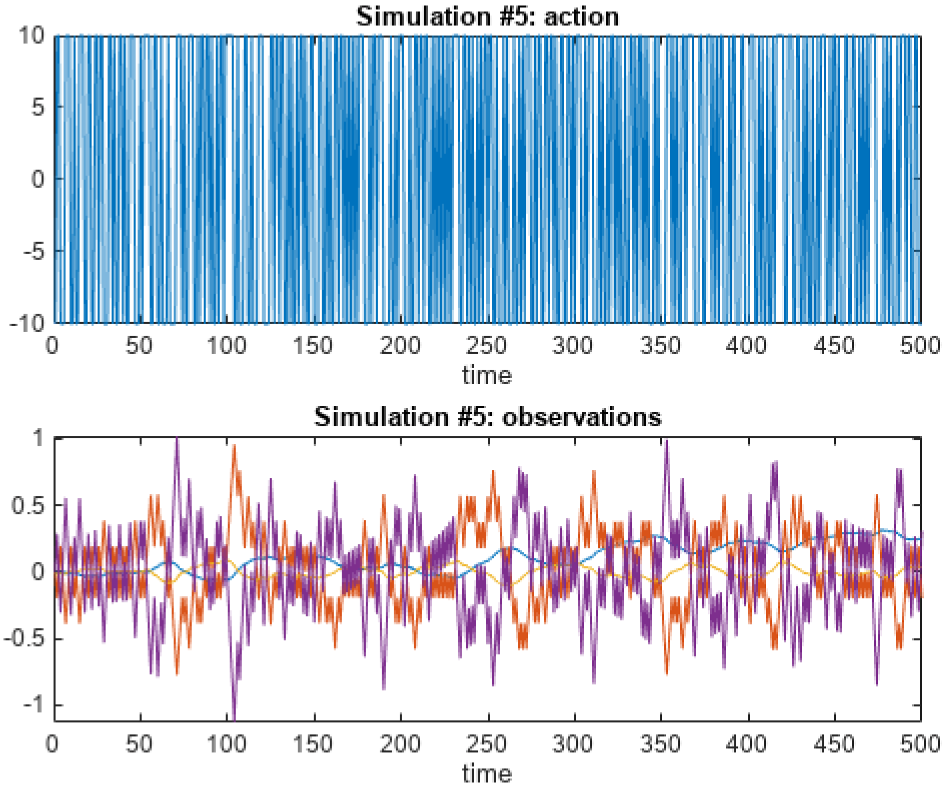
```
toc
```

```
Elapsed time is 16.265069 seconds.
```

Plot the action and observations from the fifth simulation.

```
figure

subplot(2,1,1);
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Action]))
title('Simulation #5: action');
xlabel('time');

subplot(2,1,2)
```

```
plot(outs(5).AgentData.Time(2:end), ...
     cell2mat([outs(5).AgentData.Experiences.Observation]))
title('Simulation #5: observations')
xlabel('time');
```



Clear the array of `Future` objects, the environment, and delete the parallel pool (this is the reverse order in which they were created).

```
clear ftr
clear env
delete(pp)
```

Version History

Introduced in R2022a

See Also

Objects

`Future` | `Simulink.Simulation.Future`

Functions

`fetchNext` | `fetchOutputs` | `cancel` | `wait` | `runEpisode` | `setup` | `cleanup` | `reset`

Topics

“Custom Training Loop with Simulink Action Noise”
“Train Reinforcement Learning Policy Using Custom Training Loop”
“Create MATLAB Reinforcement Learning Environments”
“Create Simulink Reinforcement Learning Environments”
“Train Agents Using Parallel Computing and GPUs”

MonitorLogger

Log reinforcement learning training data to monitor window

Description

Use a MonitorLogger object to log data to a monitor window, within the train function or inside a custom training loop. To log data when using the train function, specify appropriate callback functions in MonitorLogger, as shown in the examples. These callbacks are executed at different stages of training, for example, EpisodeFinishedFcn is executed after the completion of an episode. The output of a callback function is a structure containing the data to log at that stage of training.

Note Using a MonitorLogger object to log data when using the train function does not affect (and is not affected by) any option to save agents during training specified within an rlTrainingOptions object.

Note MonitorLogger is an handle object. If you assign an existing MonitorLogger object to a new MonitorLogger object, both the new object and the original one refer to the same underlying object in memory. To preserve the original object parameters for later use, save the object to a MAT-file. For more information about handle objects, see “Handle Object Behavior”.

Creation

Create a MonitorLogger object using rlDataLogger specifying a trainingProgressMonitor object as input argument.

Properties

LoggingOptions — Object containing a set of logging options
MonitorLoggingOptions object (default)

Object containing a set of logging options, returned as a MonitorLoggingOptions object. A MonitorLoggingOptions object has the following properties, which you can access using dot notation after creating the MonitorLogger object.

DataWriteFrequency — Monitor data write period
1 (default) | positive integer

Monitor data write period, specified as a positive integer. It is the number of episodes after which data is transmitted to the trainingProgressMonitor object. For example, if DataWriteFrequency is 5 then data from episodes 1 to 5 will be cached in memory and be transmitted to the monitor object at the end of the 5-th episode. This improves performance in some cases. The default is 1.

Example: `DataWriteFrequency=10`

MaxEpisodes — Maximum number of episodes

500 (default) | positive integer

Maximum number of episodes, specified as a positive integer. When using train, the value is automatically initialized. Set this value when using the logger object in a custom training loop. The default is 500.

Example: MaxEpisodes=1000

EpisodeFinishedFcn — Callback to log data after episode completion

[] (default) | function handle

Callback to log data after episode completion, specified as a function handle object. The specified function is automatically called by the training loop at the end of each episode, and must return a structure containing the data to log, such as experiences, simulation information, or initial observations.

Your function must have the following signature.

```
function dataToLog = myEpisodeFinishedFcn(data)
```

Here, data is a structure that contains the following fields:

- EpisodeCount — current episode number
- Environment — environment object
- Agent — agent object
- Experience — structure array containing the experiences. Each element of this array corresponds to a step and is a structure containing the fields NextObservation, Observation, Action, Reward and IsDone.
- EpisodeInfo — structure containing the fields CumulativeReward, StepsTaken and InitialObservation.
- SimulationInfo — contains simulation information from the episode. For MATLAB environments this is a structure with the field SimulationError, and for Simulink environments it is a Simulink.SimulationOutput object.

The function output dataToLog is the structure containing the data to be logged to disk.

Example: EpisodeFinishedFcn=@myEpLoggingFcn

AgentStepFinishedFcn — Callback to log data after training step completion

[] (default) | function handle | cell array of function handles

Callback to log data after training step completion within an episode, specified as a function handle object. The specified function is automatically called by the training loop at the end of each training step, and must return a structure containing the data to log, such as for example the state of the agent's exploration policy.

Your function must have the following signature.

```
function dataToLog = myAgentStepFinishedFcn(data)
```

Here, data is a structure that contains the following fields:

- EpisodeCount — current episode number

- AgentStepCount — cumulative number of steps taken by the agent
- SimulationTime — current simulation time in the environment
- Agent — agent object

The function output dataToLog is the structure containing the data to be logged to disk.

For multi agent training, AgentStepFinishedFcn can be a cell array of function handles with as many elements as number of agent groups.

Note Logging data using the AgentStepFinishedFcn callback is not supported when training agents in parallel with the train function.

Example: AgentStepFinishedFcn=@myAgtStepLoggingFcn

AgentLearnFinishedFcn — Callback to log data after completion of the learn subroutine
[] (default) | function handle | cell array of function handles

Callback to log data after completion of the learn subroutine, specified as a function handle object. The specified function is automatically called by the training loop at the end of each learning subroutine, and must return a structure containing the data to log, such as the training losses of the actor and critic networks, or, for a model-based agent, the environment model training losses.

Your function must have the following signature.

```
function dataToLog = myAgentLearnFinishedFcn(data)
```

Here, data is a structure that contains the following fields:

- EpisodeCount — current episode number
- AgentStepCount — cumulative number of steps taken by the agent
- AgentLearnCount — cumulative number of learning steps taken by the agent
- EnvModelTrainingInfo — structure containing model-based agents related fields TransitionFcnLoss, RewardFcnLoss and IsDoneFcnLoss.
- Agent — agent object
- ActorLoss — loss of the actor
- CriticLoss — loss of the critic

The function output dataToLog is the structure containing the data to be logged to disk.

For multi agent training, AgentLearnFinishedFcn can be a cell array of function handles with as many elements as number of agent groups.

Example: AgentLearnFinishedFcn=@myAgtLearnLoggingFcn

Object Functions

setup	Set up reinforcement learning environment or initialize data logger object
cleanup	Clean up reinforcement learning environment or data logger object

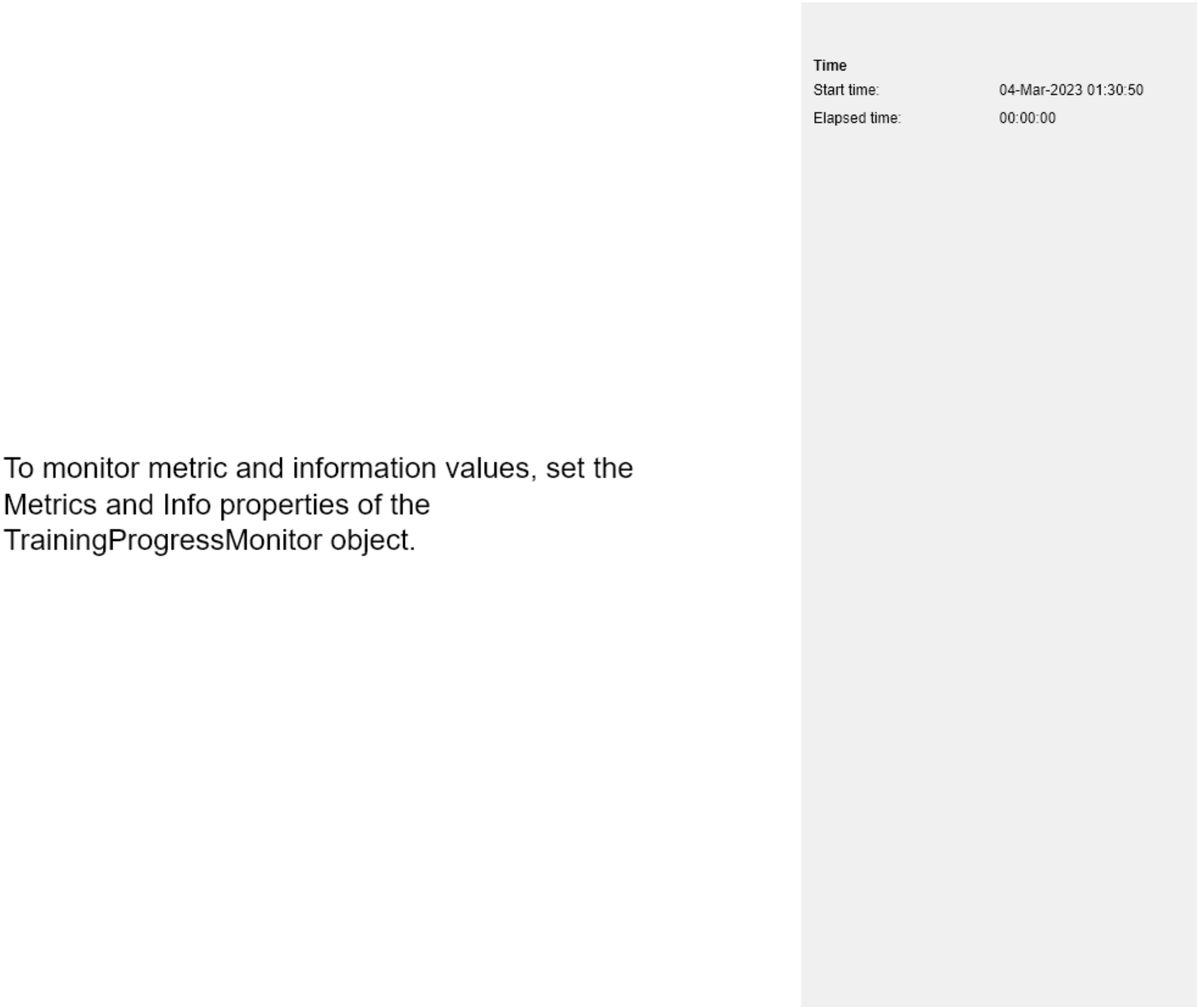
Examples

Log and Visualize Data with a Training Progress Monitor Object

This example shows how to log and visualize data to the window of a `trainingProgressMonitor` object when using `train`.

Create a `trainingProgressMonitor` object. Creating the object also opens a window associated with the object.

```
monitor = trainingProgressMonitor();
```



To monitor metric and information values, set the Metrics and Info properties of the TrainingProgressMonitor object.

Time

Start time: 04-Mar-2023 01:30:50

Elapsed time: 00:00:00

Create a `MonitorLogger` object using `rlDataLogger`.

```
logger = rlDataLogger(monitor);
```

Create callback functions to log the data (for this example, see the helper function section), and specify the appropriate callback functions in the logger object. For the specific function signatures and more information on the function input structure, see the corresponding properties of `MonitorLogger`.

```
logger.AgentLearnFinishedFcn = @myAgentLearnFinishedFcn;
```

To train the agent, you can now call `train`, passing `logger` as an argument such as in the following command.

```
trainResult = train(agent, env, trainOpts, Logger=logger);
```

While the training progresses, data will be logged to the training monitor object, and visualized in the associated window.

Note that only scalar data can be logged with a monitor logger object.

Example Logging Functions

Define a logging function that logs data periodically at the completion of the learning subroutine. This function is automatically called by the training loop at the end of each learning subroutine, and must return a structure containing the learning-related data to log, such as the training losses of the actor and critic networks, or, for a model-based agent, the environment model training losses.

```
function dataToLog = myAgentLearnFinishedFcn(data)

    if mod(data.AgentLearnCount, 2) == 0
        dataToLog.ActorLoss  = data.ActorLoss;
        dataToLog.CriticLoss = data.CriticLoss;
    else
        dataToLog = [];
    end

end
```

Limitations

- Only scalar data is supported when logging data with a `MonitorLogger` object. The structure returned by the callback functions must contain fields with scalar data.
- Resuming of training from a previous training result is not supported when logging data with a `MonitorLogger` object.

Version History

Introduced in R2022b

See Also

Functions

`rlDataLogger` | `train` | `setup` | `store` | `write` | `cleanup`

Objects

`FileLogger` | `trainingProgressMonitor` | `rlTrainingOptions`

Topics

“Log Training Data to Disk”
“Monitor Custom Training Loop Progress”
“Function Handles”
“Handle Object Behavior”

rlACAgent

Actor-critic (AC) reinforcement learning agent

Description

Actor-critic (AC) agents implement actor-critic algorithms such as A2C and A3C, which are model-free, online, on-policy reinforcement learning methods. The actor-critic agent optimizes the policy (actor) directly and uses a critic to estimate the return or future rewards. The action space can be either discrete or continuous. For continuous action spaces, this agent does not enforce constraints set in the action specification; therefore, if you need to enforce action constraints, you must do so within the environment.

For more information, see “Actor-Critic (AC) Agents”. For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlACAgent(observationInfo,actionInfo)
agent = rlACAgent(observationInfo,actionInfo,initOpts)

agent = rlACAgent(actor,critic)

agent = rlACAgent( ___ ,agentOptions)
```

Description

Create Agent from Observation and Action Specifications

`agent = rlACAgent(observationInfo,actionInfo)` creates an actor-critic agent for an environment with the given observation and action specifications, using default initialization options. The actor and critic in the agent use default deep neural networks built from the observation specification `observationInfo` and the action specification `actionInfo`. The `ObservationInfo` and `ActionInfo` properties of `agent` are set to the `observationInfo` and `actionInfo` input arguments, respectively.

`agent = rlACAgent(observationInfo,actionInfo,initOpts)` creates an actor-critic agent for an environment with the given observation and action specifications. The agent uses default networks in which each hidden fully connected layer has the number of units specified in the `initOpts` object. Actor-critic agents do not support recurrent neural networks. For more information on the initialization options, see `rlAgentInitializationOptions`.

Create Agent from Actor and Critic

`agent = rlACAgent(actor,critic)` creates an actor-critic agent with the specified actor and critic, using the default options for the agent.

Specify Agent Options

`agent = rlACAgent( ___ ,agentOptions)` creates an actor-critic agent and sets the AgentOptions property to the `agentOptions` input argument. Use this syntax after any of the input arguments in the previous syntaxes.

Input Arguments

`initOpts` — Agent initialization options
`rlAgentInitializationOptions` object

Agent initialization options, specified as an `rlAgentInitializationOptions` object. Actor-critic agents do not support recurrent neural networks.

`actor` — Actor
`rlDiscreteCategoricalActor` object | `rlContinuousGaussianActor` object

Actor that implements the policy, specified as an `rlDiscreteCategoricalActor` or `rlContinuousGaussianActor` function approximator object. For more information on creating actor approximators, see “Create Policies and Value Functions”.

`critic` — Critic
`rlValueFunction` object

Critic that estimates the discounted long-term reward, specified as an `rlValueFunction` object. For more information on creating critic approximators, see “Create Policies and Value Functions”.

Properties

`ObservationInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

If you create the agent by specifying an actor and critic, the value of `ObservationInfo` matches the value specified in the actor and critic objects.

You can extract `observationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`ActionInfo` — Action specification
`rlFiniteSetSpec` object | `rlNumericSpec` object

Action specifications, specified either as an `rlFiniteSetSpec` (for discrete action spaces) or `rlNumericSpec` (for continuous action spaces) object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

If you create the agent by specifying an actor and critic, the value of `ActionInfo` matches the value specified in the actor and critic objects.

You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`AgentOptions` — Agent options
`rlACAgentOptions` object

Agent options, specified as an `rlACAgentOptions` object.

`UseExplorationPolicy` — Option to use exploration policy
`true` (default) | `false`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- `true` — Use the base agent exploration policy when selecting actions in `sim` and `generatePolicyFunction`. In this case, the agent selects its actions by sampling its probability distribution, the policy is therefore stochastic and the agent explores its observation space.
- `false` — Use the base agent greedy policy (the action with maximum likelihood) when selecting actions in `sim` and `generatePolicyFunction`. In this case, the simulated agent and generated policy behave deterministically.

Note This option affects only simulation and deployment; it does not affect training.

`SampleTime` — Sample time of agent
positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations. The value of `SampleTime` matches the value specified in `AgentOptions`.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent
setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent
setCritic	Set critic of reinforcement learning agent

generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create Discrete Actor-Critic Agent from Observation and Action Specifications

Create an environment with a discrete action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar with five possible elements (a torque of -2, -1, 0, 1, or 2 Nm applied to a swinging pole).

```
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");
```

Obtain observation and action specifications

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create an actor-critic agent from the environment observation and action specifications.

```
agent = rlACAgent(obsInfo,actInfo);
```

To check your agent, use `getAction` to return the action from random observations.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[-2]}
```

You can now test and train the agent within the environment. You can also use `getActor` and `getCritic` to extract the actor and critic, respectively, and `getModel` to extract the approximator model (by default a deep neural network) from the actor or critic.

Create Continuous Actor-Critic Agent Using Initialization Options

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar representing a torque ranging continuously from -2 to 2 Nm.

```
% Load predefined environment
env = rlPredefinedEnv("SimplePendulumWithImage-Continuous");
```

```
% Obtain observation and action specifications
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create an agent initialization option object, specifying that each hidden fully connected layer in the network must have 128 neurons (instead of the default number, 256).

```
initOpts = rlAgentInitializationOptions(NumHiddenUnit=128);
```

The agent creation function initializes the actor and critic networks randomly. You can ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create an actor-critic agent from the environment observation and action specifications.

```
agent = rlACAgent(obsInfo,actInfo,initOpts);
```

Extract the deep neural networks from both the agent actor and critic.

```
actorNet = getModel(getActor(agent));
criticNet = getModel(getCritic(agent));
```

To verify that each hidden fully connected layer has 128 neurons, you can display the layers on the MATLAB® command window,

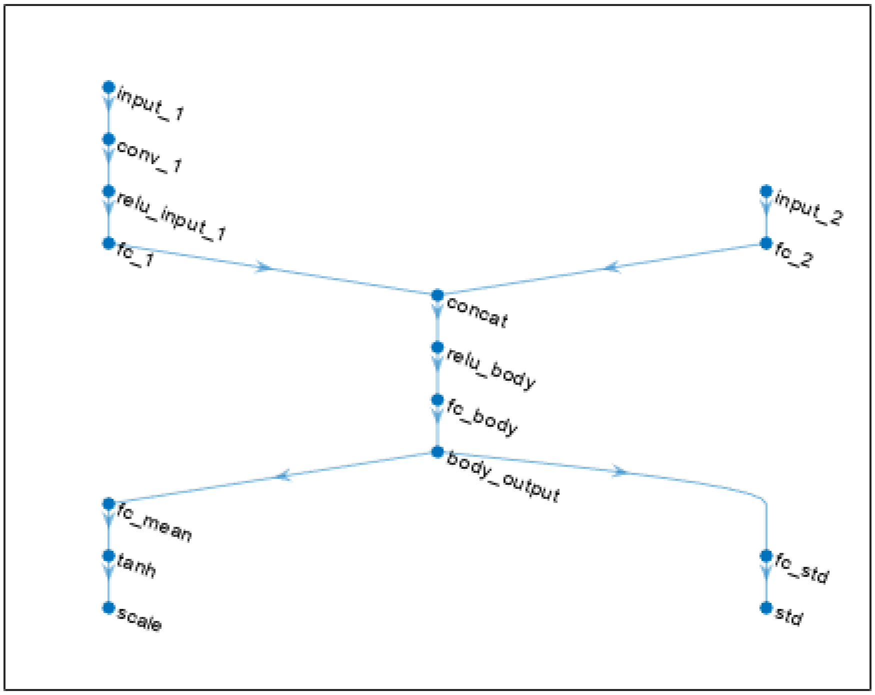
```
criticNet.Layers
```

or visualize the structure interactively using `analyzeNetwork`.

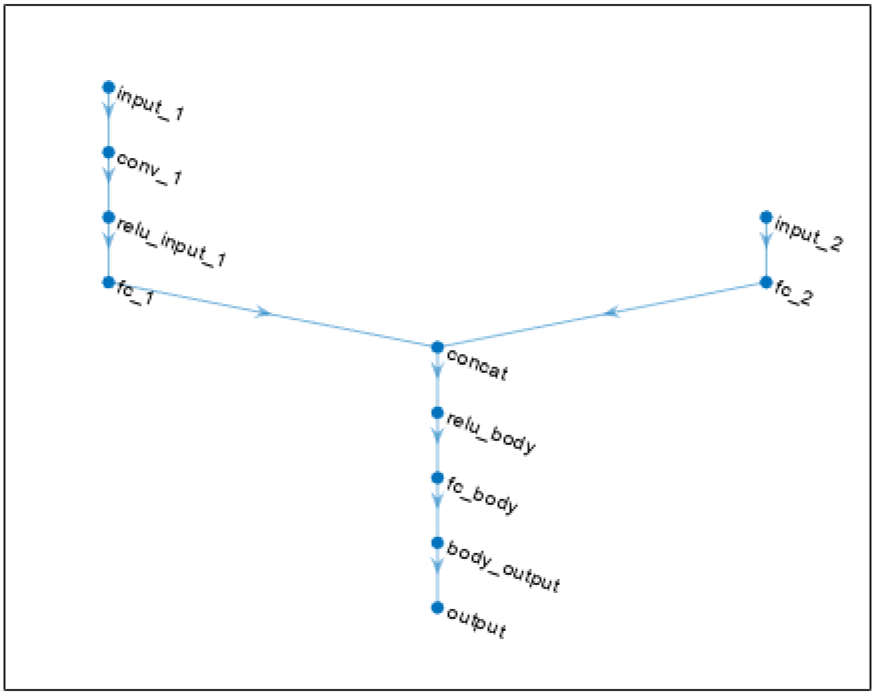
```
analyzeNetwork(criticNet)
```

Plot actor and critic networks

```
plot(layerGraph(actorNet))
```



```
plot(layerGraph(criticNet))
```



To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[0.9228]}
```

You can now test and train the agent within the environment.

Create Discrete Actor-Critic Agent from Actor and Critic

Create an environment with a discrete action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DQN Agent to Balance Cart-Pole System”. This environment has a four-dimensional observation vector (cart position and velocity, pole angle, and pole angle derivative), and a scalar action with two possible elements (a force of either -10 or +10 N applied on the cart).

```
env = rlPredefinedEnv("CartPole-Discrete");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlNumericSpec with properties:
     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "CartPole States"
    Description: "x, dx, theta, dtheta"
      Dimension: [4 1]
       DataType: "double"
```

```
actInfo = getActionInfo(env)
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [-10 10]
           Name: "CartPole Action"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

The agent creation function initializes the actor and critic networks randomly. You can ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Actor-critic agents use a parametrized value function approximator to estimate the value of the policy. A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (which receives the content of the observation channel, as specified by obsInfo) and one output layer (which returns the scalar value). Note that prod(obsInfo.Dimension) returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects, and get the dimension of the observation space from the environment specification objects.

```
criticNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(50)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert the network to a dlnetwork object, and display the number of weights.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 301

   Inputs:
      1   'input'   4 features
```

Create the critic approximator object using `criticNet` and the observation specification. For more information on value function approximators, see `rlValueFunction`.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Check your critic with a random observation input.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.1411
```

Actor-critic agents use a parametrized stochastic policy, which for discrete action spaces is implemented by a discrete categorical actor. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the observation channel, as specified by `obsInfo`) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by `actInfo`. Note that `numel(actInfo.Dimension)` returns the number of elements of the discrete action space.

Define the network as an array of layer objects.

```
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Dimension))
    ];
```

Convert the network to a `dlnetwork` object, and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 386

   Inputs:
      1   'input'   4 features
```

Create the actor using `actorNet` and the observation and action specifications. For more information on discrete categorical actors, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

Check your actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[10]}
```

Create the AC agent using the actor and the critic.

```
agent = rlACAgent(actor,critic)
```

```
agent = 
  rlACAgent with properties:

            AgentOptions: [1x1 rl.option.rlACAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Specify some options for the agent, including training options for the actor and critic.

```
agent.AgentOptions.NumStepsToLookAhead=32;
agent.AgentOptions.DiscountFactor=0.99;
agent.AgentOptions.CriticOptimizerOptions.LearnRate=8e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold=1;
agent.AgentOptions.ActorOptimizerOptions.LearnRate=8e-3;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold=1;
```

Check your agent with a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-10]}
```

You can now test and train the agent within the environment.

Create Continuous Actor-Critic Agent from Actor and Critic

Create an environment with a continuous action space, and obtain its observation and action specifications. For this example, load the double integrator continuous action space environment used in the example “Train DDPG Agent to Control Double Integrator System”.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "states"
    Description: "x, dx"
      Dimension: [2 1]
       DataType: "double"
```

```
actInfo = getActionInfo(env)
actInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "force"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

In this example, the action is a scalar value representing a force ranging from -2 to 2 Newton. To make sure that the output from the agent is in this range, you perform an appropriate scaling operation. Store these limits so you can easily access them later.

```
% Make sure action space upper and lower limits are finite
actInfo.LowerLimit=-2;
actInfo.UpperLimit=2;
```

The actor and critic networks are initialized randomly. You can ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Actor-critic agents use a parametrized value function approximator to estimate the value of the policy. A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (which receives the content of the observation channel, as specified by obs`Info`) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects.

```
criticNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(50)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 201

   Inputs:
      1   'input'   2 features
```

Create the critic approximator object using `criticNet` and the observation specification. For more information on value function approximators, see `rlValueFunction`.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Check your critic with a random input observation.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.0969
```

Actor-critic agents use a parametrized stochastic policy, which for continuous action spaces is implemented by a continuous Gaussian actor. This actor takes an observation as input and returns as output a random action sampled from a Gaussian probability distribution.

To approximate the mean values and standard deviations of the Gaussian distribution, you must use a neural network with two output layers, each having as many elements as the dimension of the action space. One output layer must return a vector containing the mean values for each action dimension. The other must return a vector containing the standard deviation for each action dimension.

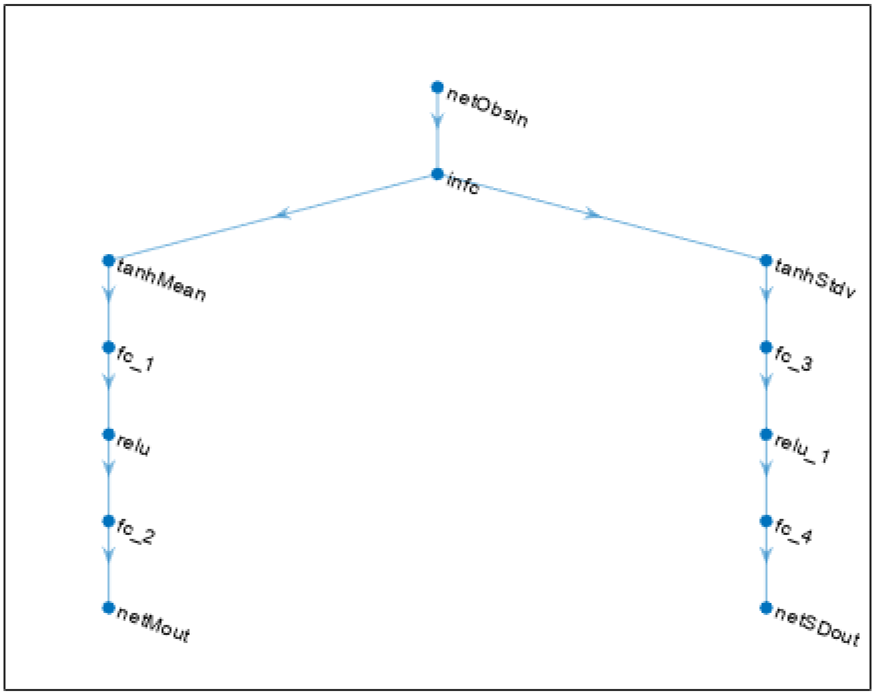
Note that standard deviations must be nonnegative and mean values must fall within the range of the action. Therefore the output layer that returns the standard deviations must be a softplus or ReLU layer, to enforce nonnegativity, while the output layer that returns the mean values must be a scaling layer, to scale the mean values to the output range.

For this example the environment has only one observation channel and therefore the network has only one input layer.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% Input path
inPath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="netObsIn")
    fullyConnectedLayer(prod(actInfo.Dimension),Name="infc")
    ];

% Mean value path
meanPath = [
    tanhLayer(Name="tanhMean");
    fullyConnectedLayer(50)
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension));
    scalingLayer( ...
    Name="netMout", ...
    Scale=actInfo.UpperLimit)  % scale to range
    ];

% Standard deviation path
sdevPath = [
    tanhLayer(Name="tanhStdv");
    fullyConnectedLayer(50)
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension));
    softplusLayer(Name="netSDout")  % nonnegative
    ];

% Add layers to network object
```

```
actorNet = layerGraph;
actorNet = addLayers(actorNet,inPath);
actorNet = addLayers(actorNet,meanPath);
actorNet = addLayers(actorNet,sdevPath);

% Connect layers
actorNet = connectLayers(actorNet,"infc","tanhMean/in");
actorNet = connectLayers(actorNet,"infc","tanhStdv/in");

% Plot network
plot(actorNet)
```



Convert the network to a `dlnetwork` object and display the number of learnable parameters (weights).

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 305

   Inputs:
      1   'netObsIn'   2 features
```

Create the actor approximator object using `actorNet` and the environment specifications. For more information, on continuous Gaussian actors, see `rlContinuousGaussianActor`.

```
actor = rlContinuousGaussianActor(actorNet, obsInfo, actInfo, ...
    ActionMeanOutputNames="netMout",...
    ActionStandardDeviationOutputNames="netSDout",...
    ObservationInputNames="netObsIn");
```

Check your actor with a random input observation.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-1.2332]}
```

Create the AC agent using the actor and the critic.

```
agent = rlACAgent(actor,critic)
```

```
agent = 
  rlACAgent with properties:

            AgentOptions: [1x1 rl.option.rlACAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlNumericSpec]
              SampleTime: 1
```

Specify agent options, including training options for its actor and critic.

```
agent.AgentOptions.NumStepsToLookAhead = 32;
agent.AgentOptions.DiscountFactor=0.99;

agent.AgentOptions.CriticOptimizerOptions.LearnRate=8e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold=1;

agent.AgentOptions.ActorOptimizerOptions.LearnRate=8e-3;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold=1;
```

Check your agent using a random input observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-1.5401]}
```

You can now test and train the agent within the environment.

Create a Discrete Actor-Critic Agent with Recurrent Neural Networks

For this example load the predefined environment used for the “Train DQN Agent to Balance Cart-Pole System” example. This environment has a four-dimensional observation vector (cart position and velocity, pole angle, and pole angle derivative), and a scalar action with two possible elements (a force of either -10 or +10 N applied on the cart).

```
env = rlPredefinedEnv("CartPole-Discrete");
```

Get observation and action information.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

To model the parametrized value function within the critic, use a recurrent neural network, which must have one input layer (which receives the content of the observation channel, as specified by obsInfo) and one output layer (which returns the scalar value).

Define the network as an array of layer objects. To create a recurrent network, use a `sequenceInputLayer` as the input layer and include at least one `lstmLayer`.

```
criticNet = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    lstmLayer(10)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert the network to a `dlnetwork` object and display the number of learnable parameters (weights).

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 611

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create the critic approximator object using `criticNet` and the observation specification. For more information on value function approximators, see `rlValueFunction`.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Check the critic with a random input observation.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.0344
```

Since the critic has a recurrent network, the (discrete categorical) actor must also use a recurrent network. The network must have one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which must return a vector of probabilities for each possible action, as specified by `actInfo`).

Define the recurrent network as an array of layer objects.

```
actorNet = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    lstmLayer(20)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 2k

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create the actor using `actorNet` and the observation and action specifications. For more information on discrete categorical actors, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

Check the actor with a random input observation.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[10]}
```

Set some training options for the critic.

```
criticOpts = rlOptimizerOptions( ...
    LearnRate=8e-3,GradientThreshold=1);
```

Set some training options for the actor.

```
actorOpts = rlOptimizerOptions( ...
    LearnRate=8e-3,GradientThreshold=1);
```

Specify training options for the agent, and include `actorOpts` and `criticOpts`. Since the agent uses recurrent neural networks, `NumStepsToLookAhead` is treated as the training trajectory length.

```
agentOpts = rlACAgentOptions( ...
    NumStepsToLookAhead=32, ...
    DiscountFactor=0.99, ...
    CriticOptimizerOptions=criticOpts, ...
    ActorOptimizerOptions=actorOpts);
```

Create an AC agent using the actor, the critic, and the agent options object.

```
agent = rlACAgent(actor,critic,agentOpts)
```

```
agent = 
  rlACAgent with properties:

            AgentOptions: [1x1 rl.option.rlACAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
```

```
          ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
          SampleTime: 1
```

To check your agent, return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[10]}
```

To evaluate the agent using sequential observations, use the sequence length (time) dimension. For example, obtain actions for a sequence of 9 observations.

```
[action,state] = getAction(agent, ...
    {rand([obsInfo.Dimension 1 9])});
```

Display the action corresponding to the seventh element of the observation.

```
action = action{1};
action(1,1,1,7)
```

```
ans = -10
```

You can now test and train the agent within the environment.

Configure Options for A3C Training

To train an agent using the asynchronous advantage actor-critic (A3C) method, you must set the agent and parallel training options appropriately.

When creating the AC agent, set the NumStepsToLookAhead value to be greater than 1. Common values are 64 and 128.

```
agentOpts = rlACAgentOptions(NumStepsToLookAhead=64);
```

Use agentOpts when creating your agent. Alternatively, create your agent first and then modify its options, including the actor and critic options later using dot notation.

Configure the training algorithm to use asynchronous parallel training.

```
trainOpts = rlTrainingOptions(UseParallel=true);
trainOpts.ParallelizationOptions.Mode = "async";
```

Configure the workers to return gradient data to the host. Also, set the number of steps before the workers send data back to the host to match the number of steps to look ahead.

```
trainOpts.ParallelizationOptions.DataToSendFromWorkers = ...
    "gradients";
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = ...
    agentOpts.NumStepsToLookAhead;
```

Use trainOpts when training your agent.

For an example on asynchronous advantage actor-critic agent training, see “Train AC Agent to Balance Cart-Pole System Using Parallel Computing”.

Tips

- For continuous action spaces, the `rlACAgent` object does not enforce the constraints set by the action specification, so you must enforce action space constraints within the environment.

Version History

Introduced in R2019a

See Also

Apps

Reinforcement Learning Designer | **Deep Network Designer**

Functions

`getAction` | `getActor` | `getCritic` | `generatePolicyFunction` | `generatePolicyBlock` | `getActionInfo` | `getObservationInfo`

Objects

`rlACAgentOptions` | `rlAgentInitializationOptions` | `rlValueFunction` | `rlDiscreteCategoricalActor` | `rlContinuousGaussianActor` | `rlPGAgent` | `rlPPOAgent` | `rlTRPOAgent`

Blocks

RL Agent | Policy

Topics

“Train AC Agent to Balance Cart-Pole System”
“Train Reinforcement Learning Agents”
“Actor-Critic (AC) Agents”
“Reinforcement Learning Agents”

rlACAgentOptions

Options for AC agent

Description

Use an `rlACAgentOptions` object to specify options for creating actor-critic (AC) agents. To create an actor-critic agent, use `rlACAgent`

For more information see “Actor-Critic (AC) Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlACAgentOptions
opt = rlACAgentOptions(Name,Value)
```

Description

`opt = rlACAgentOptions` creates a default option set for an AC agent. You can modify the object properties using dot notation.

`opt = rlACAgentOptions(Name,Value)` sets option properties on page 3-41 using name-value pairs. For example, `rlDQNAgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`NumStepsToLookAhead` — Number of steps ahead
`32` (default) | positive integer

Number of steps the agent interacts with the environment before learning from its experience, specified as a positive integer. When the agent uses a recurrent neural network, `NumStepsToLookAhead` is treated as the training trajectory length. When the agent is trained in parallel, `NumStepsToLookAhead` is ignored, and the whole episode is used to compute the gradients.

`EntropyLossWeight` — Entropy loss weight
`0` (default) | scalar value between `0` and `1`

Entropy loss weight, specified as a scalar value between `0` and `1`. A higher entropy loss weight value promotes agent exploration by applying a penalty for being too certain about which action to take. Doing so can help the agent move out of local optima.

When gradients are computed during training, an additional gradient component is computed for minimizing this loss function.

ActorOptimizerOptions — Actor optimizer options

rlOptimizerOptions object

Actor optimizer options, specified as an rlOptimizerOptions object. It allows you to specify training parameters of the actor approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see rlOptimizerOptions and rlOptimizer.

CriticOptimizerOptions — Critic optimizer options

rlOptimizerOptions object

Critic optimizer options, specified as an rlOptimizerOptions object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see rlOptimizerOptions and rlOptimizer.

SampleTime — Sample time of agent

1 (default) | positive scalar | -1

Sample time of agent, specified as a positive scalar or as -1. Setting this parameter to -1 allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every SampleTime seconds of simulation time. If SampleTime is -1, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, SampleTime is the time interval between consecutive elements in the output experience returned by sim or train. If SampleTime is -1, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

DiscountFactor — Discount factor

0.99 (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlACAgent	Actor-critic (AC) reinforcement learning agent

Examples

Create AC Agent Options Object

Create an AC agent options object, specifying the discount factor.

```
opt = rlACAgentOptions(DiscountFactor=0.95)
```

```
opt = 
  rlACAgentOptions with properties:
      NumStepsToLookAhead: 32
        EntropyLossWeight: 0
    ActorOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
   CriticOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
               SampleTime: 1
           DiscountFactor: 0.9500
               InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

Configure Options for A3C Training

To train an agent using the asynchronous advantage actor-critic (A3C) method, you must set the agent and parallel training options appropriately.

When creating the AC agent, set the `NumStepsToLookAhead` value to be greater than `1`. Common values are `64` and `128`.

```
agentOpts = rlACAgentOptions(NumStepsToLookAhead=64);
```

Use `agentOpts` when creating your agent. Alternatively, create your agent first and then modify its options, including the actor and critic options later using dot notation.

Configure the training algorithm to use asynchronous parallel training.

```
trainOpts = rlTrainingOptions(UseParallel=true);
trainOpts.ParallelizationOptions.Mode = "async";
```

Configure the workers to return gradient data to the host. Also, set the number of steps before the workers send data back to the host to match the number of steps to look ahead.

```
trainOpts.ParallelizationOptions.DataToSendFromWorkers = ...
    "gradients";
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = ...
    agentOpts.NumStepsToLookAhead;
```

Use `trainOpts` when training your agent.

For an example on asynchronous advantage actor-critic agent training, see “Train AC Agent to Balance Cart-Pole System Using Parallel Computing”.

Version History

Introduced in R2019a

R2022a: Simulation and deployment: `UseDeterministicExploitation` will be removed

Warns starting in R2022a

The property `UseDeterministicExploitation` of the `rlACAgentOptions` object will be removed in a future release. Use the `UseExplorationPolicy` property of `rlACAgent` instead.

Previously, you set UseDeterministicExploitation as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.AgentOptions.UseDeterministicExploitation = true;
  ```
- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.AgentOptions.UseDeterministicExploitation = false;
  ```

Starting in R2022a, set UseExplorationPolicy as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.UseExplorationPolicy = false;
  ```
- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.UseExplorationPolicy = true;
  ```

Similarly to UseDeterministicExploitation, UseExplorationPolicy affects only simulation and deployment; it does not affect training.

R2020b: Default value for NumStepsToLookAhead changed to 32
Behavior change in future release

A value of 32 for this property should work better than 1 for most environments. If you nave MATLAB R2020b or a later version and you want to reproduce how rlACAgent behaved on versions prior to R2020b, set this value to 1.

See Also

Objects
rlACAgent | rlAgentInitializationOptions

Topics
“Train AC Agent to Balance Cart-Pole System”
“Train Reinforcement Learning Agents”
“Actor-Critic (AC) Agents”
“Reinforcement Learning Agents”

rlAdditiveNoisePolicy

Policy object to generate continuous noisy actions for custom training loops

Description

This object implements an additive noise policy, which returns continuous deterministic actions with added noise, given an input observation. You can create an `rlAdditiveNoisePolicy` object from an `rlContinuousDeterministicActor` or extract it from an `rlDDPGAgent` or `rlTD3Agent`. You can then train the policy object using a custom training loop. If `UseNoisyAction` is set to `0` the policy does not explore. This object is not compatible with `generatePolicyBlock` and `generatePolicyFunction`. For more information on policies and value functions, see “Create Policies and Value Functions”.

Creation

Syntax

```
policy = rlAdditiveNoisePolicy(actor)
policy = rlAdditiveNoisePolicy(actor,NoiseType=noiseType)
```

Description

`policy = rlAdditiveNoisePolicy(actor)` creates the additive noise policy object `policy` from the continuous deterministic actor `actor`. It also sets the `Actor` property of `policy` to the input argument `actor`.

`policy = rlAdditiveNoisePolicy(actor,NoiseType=noiseType)` specifies the type of noise distribution for the policy. `noiseType` can be either `"gaussian"` (Gaussian noise) or `"ou"` (Ornstein-Uhlenbeck noise). This syntax also sets the `NoiseType` property of `policy` to the input argument `noiseType`.

Properties

`Actor` — Continuous deterministic actor
`rlContinuousDeterministicActor` object

Continuous deterministic actor, specified as an `rlContinuousDeterministicActor` object.

`NoiseType` — Noise type
`"gaussian"` (default) | `"ou"`

Noise type, specified as either `"gaussian"` (default, Gaussian noise) or `"ou"` (Ornstein-Uhlenbeck noise). For more information on noise models, see “Noise Models” on page 3-422.

Example: `"ou"`

`NoiseOptions` — Noise model options
`GaussianActionNoise` object (default) | `OrnsteinUhlenbeckActionNoise` object

Noise model options, specified as a `GaussianActionNoise` object or an `OrnsteinUhlenbeckActionNoise` object. Changing the noise state or any noise option of an `rlAdditiveNoisePolicy` object deployed through code generation is not supported.

For more information on noise models, see “Noise Models” on page 3-422.

`EnableNoiseDecay` — Option to enable noise decay

`true` (default) | `false`

Option to enable noise decay, specified as a logical value: either `true` (default, enabling noise decay) or `false` (disabling noise decay).

Example: `false`

`UseNoisyAction` — Option to enable noisy action

`true` (default) | `false`

Option to enable noisy actions, specified as a logical value: either `true` (default, adding noise to actions, which helps exploration) or `false` (no noise is added to the actions). When noise is not added to the actions the policy is deterministic and therefore it does not explore.

Example: `false`

`ObservationInfo` — Observation specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array of such objects. These objects define properties such as the dimensions, data types, and names of the observation channels.

`ActionInfo` — Action specifications

`rlNumericSpec` object

Action specifications, specified as an `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

`SampleTime` — Sample time of policy

positive scalar | `-1` (default)

Sample time of the policy, specified as a positive scalar or as `-1` (default). Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the policy is specified executes every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the policy is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience. If `SampleTime` is `-1`, the sample time is treated as being equal to `1`.

Example: `0.2`

Object Functions

getAction Obtain action from agent, actor, or policy object given environment observations

getLearnableParameters Obtain learnable parameter values from agent, function approximator, or policy object

reset Reset environment, agent, experience buffer, or policy object

setLearnableParameters Set learnable parameter values of agent, function approximator, or policy object

Examples

Create Additive Noise Policy from Continuous Deterministic Actor

Create observation and action specification objects. For this example, define the observation and action spaces as continuous four- and two-dimensional spaces, respectively.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlNumericSpec([2 1]);
```

Alternatively, use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment.

Create a continuous deterministic actor. This actor must accept an observation as input and return an action as output.

To approximate the policy function within the actor, use a deep neural network model. Define the network as an array of layer objects, and get the dimension of the observation and action spaces from the environment specification objects.

```
layers = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(actInfo.Dimension(1))
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
model = dlnetwork(layers);
summary(model)
```

```
   Initialized: true

   Number of learnables: 114

   Inputs:
      1   'input'   4 features
```

Create the actor using `model`, and the observation and action specifications.

```
actor = rlContinuousDeterministicActor(model,obsInfo,actInfo)
```

```
actor = 
  rlContinuousDeterministicActor with properties:
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

Check the actor with a random observation input.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = 2x1 single column vector

    0.4013
    0.0578
```

Create a policy object from `actor`.

```
policy = rlAdditiveNoisePolicy(actor)
```

```
policy = 
  rlAdditiveNoisePolicy with properties:

               Actor: [1x1 rl.function.rlContinuousDeterministicActor]
           NoiseType: "gaussian"
        NoiseOptions: [1x1 rl.option.GaussianActionNoise]
    EnableNoiseDecay: 1
      UseNoisyAction: 1
     ObservationInfo: [1x1 rl.util.rlNumericSpec]
          ActionInfo: [1x1 rl.util.rlNumericSpec]
          SampleTime: -1
```

You can access the policy options using dot notation. For example, change the upper and lower limits of the distribution.

```
policy.NoiseOptions.LowerLimit = -3;
policy.NoiseOptions.UpperLimit = 3;
```

Check the policy with a random observation input.

```
act = getAction(policy,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = 2×1

    0.1878
   -0.1645
```

You can now train the policy with a custom training loop and then deploy it to your application.

Create Additive Noise Policy Specifying Noise Model

Create observation and action specification objects. For this example, define the observation and action spaces as continuous three- and one-dimensional spaces, respectively.

```
obsInfo = rlNumericSpec([3 1]);
actInfo = rlNumericSpec([1 1]);
```

Alternatively, use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment

Create a continuous deterministic actor. This actor must accept an observation as input and return an action as output.

To approximate the policy function within the actor, use a deep neural network model. Define the network as an array of layer objects, and get the dimension of the observation and action spaces from the environment specification objects.

```
layers = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(9)
    reluLayer
    fullyConnectedLayer(actInfo.Dimension(1))
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
model = dlnetwork(layers);
summary(model)
```

```
   Initialized: true

   Number of learnables: 46

   Inputs:
      1   'input'   3 features
```

Create the actor using `model`, and the observation and action specifications.

```
actor = rlContinuousDeterministicActor(model,obsInfo,actInfo)
```

```
actor = 
  rlContinuousDeterministicActor with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

Check the actor with a random observation input.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = single
    -0.2535
```

Create a policy object from `actor`, specifying an Ornstein-Uhlenbeck probability distribution for the noise.

```
policy = rlAdditiveNoisePolicy(actor,NoiseType="ou")
```

```
policy = 
  rlAdditiveNoisePolicy with properties:
```

```
               Actor: [1x1 rl.function.rlContinuousDeterministicActor]
           NoiseType: "ou"
        NoiseOptions: [1x1 rl.option.OrnsteinUhlenbeckActionNoise]
    EnableNoiseDecay: 1
      UseNoisyAction: 1
     ObservationInfo: [1x1 rl.util.rlNumericSpec]
          ActionInfo: [1x1 rl.util.rlNumericSpec]
          SampleTime: -1
```

You can access the policy options using dot notation. For example, change the standard deviation of the distribution.

```
policy.NoiseOptions.StandardDeviation = 0.6;
```

Check the policy with a random observation input.

```
act = getAction(policy,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = -0.1625
```

You can now train the policy with a custom training loop and then deploy it to your application.

Version History

Introduced in R2022a

See Also

Functions

getGreedyPolicy | getExplorationPolicy | generatePolicyBlock | generatePolicyFunction | getAction | getLearnableParameters | setLearnableParameters

Objects

rlMaxQPolicy | rlEpsilonGreedyPolicy | rlDeterministicActorPolicy | rlStochasticActorPolicy | rlTD3Agent | rlDDPGAgent

Blocks

Policy | RL Agent

Topics

“Model-Based Reinforcement Learning Using Custom Training Loop”
“Train Reinforcement Learning Policy Using Custom Training Loop”
“Create Policies and Value Functions”

rlAgentInitializationOptions

Options for initializing reinforcement learning agents

Description

Use the `rlAgentInitializationOptions` object to specify initialization options for an agent. To create an agent, use the specific agent creation function, such as `rlACAgent`.

Creation

Syntax

```
initOpts = rlAgentInitializationOptions
initOpts = rlAgentInitializationOptions(Name,Value)
```

Description

`initOpts = rlAgentInitializationOptions` returns a default options object for initializing a reinforcement learning agent that supports default networks. Use the initialization options to specify agent initialization parameters, such as the number of units for each hidden layer of the agent networks and whether to use a recurrent neural network.

`initOpts = rlAgentInitializationOptions(Name,Value)` creates an initialization options object and sets its properties on page 3-51 by using one or more name-value pair arguments.

Properties

`NumHiddenUnit` — Number of units in each hidden fully connected layer
`256` (default) | positive integer

Number of units in each hidden fully connected layer of the agent networks, except for the fully connected layer just before the network output, specified as a positive integer. The value you set also applies to any LSTM layers.

Example: `'NumHiddenUnit',64`

UseRNN — Flag to use recurrent neural network
`false` (default) | `true`

Flag to use recurrent neural network, specified as a logical.

If you set `UseRNN` to `true`, during agent creation the software inserts a recurrent LSTM layer with the output mode set to sequence in the output path of the agent networks. Policy gradient and actor-critic agents do not support recurrent neural networks. For more information on LSTM, see “Long Short-Term Memory Neural Networks”.

Note TRPO agents do not support recurrent networks.

Example: 'UseRNN',true

Object Functions

rlACAgent	Actor-critic (AC) reinforcement learning agent
rlPGAgent	Policy gradient (PG) reinforcement learning agent
rlDDPGAgent	Deep deterministic policy gradient (DDPG) reinforcement learning agent
rlDQNAgent	Deep Q-network (DQN) reinforcement learning agent
rlPPOAgent	Proximal policy optimization (PPO) reinforcement learning agent
rlTD3Agent	Twin-delayed deep deterministic (TD3) policy gradient reinforcement learning agent
rlSACAgent	Soft actor-critic (SAC) reinforcement learning agent
rlTRPOAgent	Trust region policy optimization (TRPO) reinforcement learning agent

Examples

Create Agent Initialization Options Object

Create an agent initialization options object, specifying the number of hidden neurons and use of a recurrent neural network.

```
initOpts = rlAgentInitializationOptions(NumHiddenUnit=64,UseRNN=true)
```

```
initOpts = 
  rlAgentInitializationOptions with properties:

    NumHiddenUnit: 64
           UseRNN: 1
```

You can modify the options using dot notation. For example, set the agent sample time to 0.5.

```
initOpts.NumHiddenUnit = 128
```

```
initOpts = 
  rlAgentInitializationOptions with properties:

    NumHiddenUnit: 128
           UseRNN: 1
```

To create your agent, use initOpts as an input argument of an agent constructor function.

Version History

Introduced in R2020b

See Also

Objects

rlACAgent | rlPGAgent | rlDDPGAgent | rlDQNAgent | rlPPOAgent | rlTD3Agent | rlSACAgent | rlTRPOAgent

Topics
“Reinforcement Learning Agents”

rlBehaviorCloningRegularizerOptions

Regularizer options object to train DDPG, TD3 and SAC agents

Description

Use an `rlBehaviorCloningRegularizerOptions` object to specify behavioral cloning regularizer options to train a DDPG, TD3, or SAC agent. The only option you can specify is the regularizer weight, which balances the actor loss with the behavioral cloning penalty and is mostly useful to train agents offline (specifically to deal with possible differences between the probability distribution of the dataset and the one generated by the environment). To enable the behavioral cloning regularizer when training an agent, set the `BatchDataRegularizerOptions` property of the agent options object to an `rlBehaviorCloningRegularizerOptions` object that has your preferred regularizer weight.

Creation

Syntax

```
bcOpts = rlBehaviorCloningRegularizerOptions
bcOpts = rlBehaviorCloningRegularizerOptions(Name=Value)
```

Description

`bcOpts = rlBehaviorCloningRegularizerOptions` returns a default behavioral cloning regularizer options set.

`bcOpts = rlBehaviorCloningRegularizerOptions(Name=Value)` creates the behavioral cloning regularizer option set `bcOpts` and sets its properties using one or more name-value arguments.

Properties

`BehaviorCloningRegularizerWeight` — Behavioral cloning regularizer weight
`2.5` (default) | positive scalar

Behavioral cloning regularizer weight, specified as a positive scalar. This weight controls the trade-off between the actor loss and the behavioral cloning penalty.

Specifically, the behavioral cloning regularizer $k^2(\pi(s_i)\text{-}a_i)^2$ is added to the actor loss L_{actor}, where a_i is an action from the minibatch (which stores N experiences) and $\pi(s_i)$ is an action from the current actor given the observation s_i (also taken from the minibatch). The actor is therefore updated to minimize the loss function, L'_{actor}:

$$L'_{actor} = \frac{1}{N}\sum_{i=1}^{N}\left(\lambda L_{actor}(s_i, \pi(s_i)) + k^2(\pi(s_i) - a_i)^2\right)$$

Here the normalization term λ depends on the behavioral cloning weight W_{bc}, which regulates the importance of the standard L_{actor} factor:

$$\lambda = \frac{W_{bc}}{\frac{1}{N}\sum_{i=1}^{N} |Q(s_i, a_i)|}$$

The scaling factor k scales the regularization term to the appropriate action range:

$$k = \frac{2}{A_{mx} - A_{mn}}$$

Here A_{mx} and A_{mn} are the upper and lower limits of the action range. These limits are taken from the action specifications (or are otherwise estimated if unavailable).

To set W_{bc}, assign a value to the `BehaviorCloningRegularizerWeight` property of the `rlBehaviorCloningRegularizerOptions` object. For more information, see [1].

Example: `BehaviorCloningRegularizerWeight=5`

Object Functions

Examples

Create Behavioral Cloning Regularizer Options Object

Create an `rlBehaviorCloningRegularizerOptions` object specifying the `BehaviorCloningRegularizerWeight`.

```
opt = rlBehaviorCloningRegularizerOptions( ...
    BehaviorCloningRegularizerWeight=5)
```

```
opt = 
  rlBehaviorCloningRegularizerOptions with properties:

    BehaviorCloningRegularizerWeight: 5
```

You can modify options using dot notation. For example, set `BehaviorCloningRegularizerWeight` to `3`.

```
opt.BehaviorCloningRegularizerWeight = 3;
```

To specify this behavioral cloning option set for an agent, first create the agent options object. For this example, create a default `rlTD3AgentOptions` object for a TD3 agent.

```
agentOpts = rlTD3AgentOptions;
```

Then, assign the `rlBehaviorCloningRegularizerOptions` object to the `BatchDataRegularizerOptions` property.

```
agentOpts.BatchDataRegularizerOptions  = opt;
```

When you create the agent, use `agentOpts` as the last input argument for the agent constructor function `rlTD3Agent`.

Version History

Introduced in R2023a

References

[1] Fujimoto, Scott, and Shixiang Shane Gu. "A minimalist approach to offline reinforcement learning." *Advances in Neural Information Processing Systems* 34 (2021): 20132-20145.

See Also

Objects

`rlConservativeQLearningOptions` | `rlDDPGAgentOptions` | `rlTD3AgentOptions` | `rlSACAgentOptions`

Topics

“Soft Actor-Critic (SAC) Agents”
“Deep Deterministic Policy Gradient (DDPG) Agents”
“Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents”

rlConservativeQLearningOptions

Regularizer options object to train DQN and SAC agents

Description

Use an `rlConservativeQLearningOptions` object to specify conservative Q-learning regularizer options to train a DQN or SAC agents. The options you can specify are the minimum weight and the number of random actions used for Q-value compensation, and are mostly useful to train agents offline (specifically to deal with possible differences between the probability distribution of the dataset and the one generated by the environment). To enable the conservative Q-learning regularizer when training an agent, set the `BatchDataRegularizerOptions` property of the agent options object to a `rlConservativeQLearningOptions` object (that has your preferred minimum weight and number of samples).

Creation

Syntax

```
cqOpts = rlConservativeQLearningOptions
cqOpts = rlConservativeQLearningOptions(Name=Value)
```

Description

`cqOpts = rlConservativeQLearningOptions` returns a default conservative Q-learning regularizer options set.

`cqOpts = rlConservativeQLearningOptions(Name=Value)` creates the conservative Q-learning regularizer option set `cqOpts` and sets its properties using one or more name-value arguments.

Properties

`MinQValueWeight` — Weight used for Q-value compensation
`1` (default) | positive scalar

Weight used for Q-value compensation, specified as a positive scalar. For more information, see “Algorithms” on page 3-58.

Example: `MinQValueWeight=0.1`

`NumSampledActions` — Number of sampled actions used for Q-value compensation
`10` (default) | positive integer

Number of sampled actions used for Q-value compensation, specified as a positive integer. This is the number of random actions used to estimate the logarithm of the sum of Q-values for the SAC agent. For more information see “Continuous Actions Regularizer (SAC)” on page 3-59.

Example: `NumSampledActions=30`

Object Functions

Examples

Create Conservative Q-Learning Options Object

Create an `rlConservativeQLearningOptions` object specifying the weight to be used for Q-value compensation.

```
opt = rlConservativeQLearningOptions( ...
    MinQValueWeight=5)
```

```
opt = 
  rlConservativeQLearningOptions with properties:

      MinQValueWeight: 5
    NumSampledActions: 10
```

You can modify options using dot notation. For example, set `NumSampledActions` to `20`.

```
opt.NumSampledActions = 20;
```

To specify this behavioral cloning option set for an agent, first create the agent options object. For this example, create a default `rlDQNAgentOptions` object for a DQN agent.

```
agentOpts = rlDQNAgentOptions;
```

Then, assign the `rlBehaviorCloningRegularizerOptions` object to the `BatchDataRegularizerOptions` property.

```
agentOpts.BatchDataRegularizerOptions  = opt;
```

When you create the agent, use `agentOpts` as the last input argument for the agent constructor function `rlDQNAgent`.

Algorithms

In conservative Q-learning the regularizer that is added to the critic loss relies on the difference between the expected Q-values of the actions from the current policy and the Q-values of the actions from the data set.

Discrete Actions Regularizer (DQN)

For an agent with a discrete action space, the resulting loss function that the agents minimizes is the following:

$$L = W_{cq}\frac{1}{M}\sum_{i=1}^{M}\left(\log\left(\sum_{a \in A}\exp(Q(s_i, a))\right) - Q(s_i, a_i) + \frac{1}{2}(Q(s_i, a_i) - y_i)^2\right)$$

Here, A is the set of all possible actions, M is the number of experiences in the minibatch, s_i is an observation stored in the minibatch, and y_i the target Q-value corresponding to $Q(s_i,a_i)$.

To set W_{cq}, assign a value to the `MinQValueWeight` property of the `rlConservativeQLearningOptions` object. For more information, see [1].

Continuous Actions Regularizer (SAC)

Similarly to the discrete action case, for an agent with a continuous action space, the resulting loss function that the agents minimizes is the following:

$$L = \frac{1}{M}\sum_{i=1}^{M}\left(W_{cq}\left(\log\left(\sum_{A}\exp(Q(s_i,a))\right) - Q(s_i,a_i)\right) + \frac{1}{2}(Q(s_i,a_i) - y_i)^2\right)$$

Here A is the (continuous) action set, M the number of experiences in the minibatch, s_i an observation stored in the minibatch, and y_i the target Q-value corresponding to $Q(s_i,a_i)$. The first, logarithmic, term in the sum is properly defined for a continuous action space in [1] and is approximated as follows:

$$\log\left(\sum_{A}\exp(Q(s_i,a))\right) \approx \log\left(\frac{1}{2N}\sum_{a_k \sim Unif(A_{mn},A_{mx})}^{N}\frac{\exp(Q(s_i,a_k))}{f_{Unif(A_{mn},A_{mx})}(a_k)} + \frac{1}{2N}\sum_{a_h \sim \pi(\cdot|s_i)}^{N}\frac{\exp(Q(s_i,a_h))}{\pi(\cdot|s_i)}\right)$$

In this second equation, $Unif(A_{mn},A_{mx})$ is an uniform distribution of action values from A_{mn} to A_{mx}, which are the lower and upper limits of the action range. These limits are taken from the action specifications (or are otherwise estimated if unavailable). The probability density function of the distribution, evaluated at a_k is:

$$f_{Unif(A_{mn},A_{mx})}(a_k) = \frac{1}{A_{mx} - A_{mn}}$$

Finally, $\pi(\bullet|s_i)$ is the distribution of the current policy given s_i.

To set W_{cq} in the first equation, assign a value to the `MinQValueWeight` property of the `rlConservativeQLearningOptions` object.

To set N (the number of actions to be sampled to estimate the logarithm term in the second equation), assign a value to the `NumSampledActions` property of the `rlConservativeQLearningOptions` object.

For more information, see [1].

Version History

Introduced in R2023a

References

[1] Kumar, Aviral, Aurick Zhou, George Tucker, and Sergey Levine. "Conservative q-learning for offline reinforcement learning." *Advances in Neural Information Processing Systems* 33 (2020): 1179-1191.

See Also

Objects
`rlDQNAgentOptions` | `rlSACAgentOptions` | `rlBehaviorCloningRegularizerOptions`

Topics
“Deep Q-Network (DQN) Agents”
“Soft Actor-Critic (SAC) Agents”

rlContinuousDeterministicActor

Deterministic actor with a continuous action space for reinforcement learning agents

Description

This object implements a function approximator to be used as a deterministic actor within a reinforcement learning agent with a continuous action space. A continuous deterministic actor takes an environment observation as input and returns as output an action that is a parametrized deterministic function of the observation, thereby implementing a parametrized deterministic policy. After you create an `rlContinuousDeterministicActor` object, use it to create a suitable agent, such as `rlDDPGAgent`. For more information on creating representations, see “Create Policies and Value Functions”.

Creation

Syntax

```
actor = rlContinuousDeterministicActor(net,observationInfo,actionInfo)
actor = rlContinuousDeterministicActor(net,observationInfo,
actionInfo,ObservationInputNames=netObsNames)
```

```
actor = rlContinuousDeterministicActor({basisFcn,W0},observationInfo,
actionInfo)
```

```
actor = rlContinuousDeterministicActor( ___ ,UseDevice=useDevice)
```

Description

`actor = rlContinuousDeterministicActor(net,observationInfo,actionInfo)` creates a continuous deterministic actor object using the deep neural network `net` as underlying approximation model. For this actor, `actionInfo` must specify a continuous action space. The network input layers are automatically associated with the environment observation channels according to the dimension specifications in `observationInfo`. The network must have a single output layer with the same data type and dimensions as the action specified in `actionInfo`. This function sets the `ObservationInfo` and `ActionInfo` properties of `actor` to the `observationInfo` and `actionInfo` input arguments, respectively.

`actor = rlContinuousDeterministicActor(net,observationInfo, actionInfo,ObservationInputNames=netObsNames)` specifies the names of the network input layers to be associated with the environment observation channels. The function assigns, in sequential order, each environment observation channel specified in `observationInfo` to the layer specified by the corresponding name in the string array `netObsNames`. Therefore, the network input layers, ordered as the names in `netObsNames`, must have the same data type and dimensions as the observation channels, as ordered in `observationInfo`.

`actor = rlContinuousDeterministicActor({basisFcn,W0},observationInfo, actionInfo)` creates a continuous deterministic actor object using a custom basis function as underlying approximation model. The first input argument is a two-element cell array whose first element is the handle `basisFcn` to a custom basis function and whose second element is the initial weight vector `W0`. This function sets the `ObservationInfo` and `ActionInfo` properties of `actor` to the `observationInfo` and `actionInfo` input arguments, respectively.

`actor = rlContinuousDeterministicActor( ___ ,UseDevice=useDevice)` specifies the device used to perform computational operations on the `actor` object, and sets the `UseDevice` property of `actor` to the `useDevice` input argument. You can use this syntax with any of the previous input-argument combinations.

Input Arguments

net — Deep neural network

array of `Layer` objects | `layerGraph` object | `DAGNetwork` object | `SeriesNetwork` object | `dlNetwork` object (preferred)

Deep neural network used as the underlying approximation model within the actor, specified as one of the following:

- Array of `Layer` objects
- `layerGraph` object
- `DAGNetwork` object
- `SeriesNetwork` object
- `dlnetwork` object

Note Among the different network representation options, `dlnetwork` is preferred, since it has built-in validation checks and supports automatic differentiation. If you pass another network object as an input argument, it is internally converted to a `dlnetwork` object. However, best practice is to convert other representations to `dlnetwork` explicitly *before* using it to create a critic or an actor for a reinforcement learning agent. You can do so using `dlnet=dlnetwork(net)`, where `net` is any Deep Learning Toolbox neural network object. The resulting `dlnet` is the `dlnetwork` object that you use for your critic or actor. This practice allows a greater level of insight and control for cases in which the conversion is not straightforward and might require additional specifications.

The network must have as many input layers as the number of environment observation channels (with each input layer receiving input from an observation channel), and a single output layer returning the action.

`rlContinuousDeterministicActor` objects support recurrent deep neural networks. For an example, see “Create Deterministic Actor from Recurrent Neural Network” on page 3-69.

The learnable parameters of the actor are the weights of the deep neural network. For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

netObsNames — Network input layers names corresponding to the environment observation channels

string array | cell array of character vectors

Network input layers names corresponding to the environment observation channels. When you use the pair value arguments `'ObservationInputNames'` with `netObsNames`, the function assigns, in sequential order, each environment observation channel specified in `observationInfo` to each network input layer specified by the corresponding name in the string array `netObsNames`. Therefore, the network input layers, ordered as the names in `netObsNames`, must have the same data type and dimensions as the observation channels, as ordered in `observationInfo`.

Example: `{"NetInput1_airspeed","NetInput2_altitude"}`

`basisFcn` — Custom basis function
function handle

Custom basis function, specified as a function handle to a user-defined MATLAB function. The user defined function can either be an anonymous function or a function on the MATLAB path. The action to be taken based on the current observation, which is the output of the actor, is the vector `a = W'*B`, where `W` is a weight matrix containing the learnable parameters and `B` is the column vector returned by the custom basis function.

Your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here, `obs1` to `obsN` are inputs in the same order and with the same data type and dimensions as the environment observation channels defined in `observationInfo`.

Example: `@(obs1,obs2,obs3) [obs3(2)*obs1(1)^2; abs(obs2(5)+obs3(1))]`

`W0` — Initial value of basis function weights
matrix

Initial value of the basis function weights `W`, specified as a matrix having as many rows as the length of the vector returned by the basis function and as many columns as the dimension of the action space.

Properties

`ObservationInfo` — Observation specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

`rlContinuousDeterministicActor` sets the `ObservationInfo` property of `actor` to the input `observationInfo`.

You can extract `ObservationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually.

`ActionInfo` — Action specifications
`rlNumericSpec` object

Action specifications, specified as an `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

rlContinuousDeterministicActor sets the ActionInfo property of actor to the input observationInfo.

You can extract ActionInfo from an existing environment or agent using getActionInfo. You can also construct the specification manually.

For custom basis function representations, the action signal must be a scalar, a column vector, or a discrete action.

UseDevice — Computation device used for training and simulation
"cpu" (default) | "gpu"

Computation device used to perform operations such as gradient computation, parameter update and prediction during training and simulation, specified as either "cpu" or "gpu".

The "gpu" option requires both Parallel Computing Toolbox software and a CUDA® enabled NVIDIA® GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use gpuDevice (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating an agent on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

To speed up training by using parallel processing over multiple cores, you do not need to use this argument. Instead, when training your agent, use an rlTrainingOptions object in which the UseParallel option is set to true. For more information about training using multicore processors and GPUs for training, see “Train Agents Using Parallel Computing and GPUs”.

Example: "gpu"

Object Functions

rlDDPGAgent	Deep deterministic policy gradient (DDPG) reinforcement learning agent
rlTD3Agent	Twin-delayed deep deterministic (TD3) policy gradient reinforcement learning agent
getAction	Obtain action from agent, actor, or policy object given environment observations
evaluate	Evaluate function approximator object given observation (or observation-action) input data
gradient	Evaluate gradient of function approximator object given observation and action input data
accelerate	Option to accelerate computation of gradient for approximator object based on neural network
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object
setModel	Set function approximation model for actor or critic
getModel	Get function approximator model from actor or critic

Examples

Create Continuous Deterministic Actor from Deep Neural Network

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that the action channel carries a column vector containing two doubles.

```
actInfo = rlNumericSpec([2 1]);
```

A continuous deterministic actor implements a parametrized deterministic policy for a continuous action space. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects, and get the dimension of the observation and action spaces from the environment specification objects.

```
net = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(actInfo.Dimension(1))
    ];
```

Convert the network to a `dlnetwork` object and display the number of learnable parameters.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 226

   Inputs:
      1   'input'   4 features
```

Create the actor object with `rlContinuousDeterministicActor`, using the network and the observation and action specification objects as input arguments. The network input layer is automatically associated with the environment observation channel according to the dimension specifications in `obsInfo`.

```
actor = rlContinuousDeterministicActor( ...
    net, ...
    obsInfo, ...
    actInfo)
```

```
actor = 
  rlContinuousDeterministicActor with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

To check your actor, use `getAction` to return the action from a random observation, using the current network weights.

```
act = getAction(actor, ...
    {rand(obsInfo.Dimension)});
act{1}
```

```
ans = 2x1 single column vector

   -0.0684
   -0.2538
```

You can now use the actor (along with a critic) to create an agent for the environment described by the given specification objects. Examples of agents that can work with continuous action and observation spaces, and use a continuous deterministic actor, are `rlDDPGAgent` and `rlTD3Agent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Continuous Deterministic Actor Specifying Network Input Layer Name

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that the action channel carries a column vector containing two doubles.

```
actInfo = rlNumericSpec([2 1]);
```

A continuous deterministic actor implements a parametrized deterministic policy for a continuous action space. This actor takes the current observation as input and returns an action as output.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects, and get the dimension of the observation and action spaces from the environment specification objects. Name the network input layer `netObsIn` so you can later explicitly associate it to the observation input channel.

```
net = [
    featureInputLayer(obsInfo.Dimension(1),Name="netObsIn")
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(actInfo.Dimension(1))];
```

Convert the network to a `dlnetwork` object, and display the number of learnable parameters.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 114

   Inputs:
      1   'netObsIn'   4 features
```

Create the actor object with `rlContinuousDeterministicActor`, using the network, the observation and action specification objects, and the name of the network input layer to be associated with the environment observation channel.

```
actor = rlContinuousDeterministicActor(net, ...
            obsInfo,actInfo, ...
            Observation="netObsIn")
```

```
actor = 
  rlContinuousDeterministicActor with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

To check your actor, use `getAction` to return the action from a random observation, using the current network weights.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = 2x1 single column vector

    0.4013
    0.0578
```

You can now use the actor (along with a critic) to create an agent for the environment described by the given specification objects. Examples of agents that can work with continuous action and observation spaces, and use a continuous deterministic actor, are `rlDDPGAgent` and `rlTD3Agent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Continuous Deterministic Actor from Custom Basis Function

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as consisting of two environment channels, the first containing a two-by-two continuous matrix and the second containing a scalar that can be only 0 or 1.

```
obsInfo = [rlNumericSpec([2 2])
           rlFiniteSetSpec([0 1])];
```

Create a continuous action space specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous three-dimensional space, so that the environment action channel carries a column vector containing three doubles.

```
actInfo = rlNumericSpec([3 1]);
```

A continuous deterministic actor implements a parametrized deterministic policy for a continuous action space. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a custom basis function with two input arguments (one for each observation channel).

```
myBasisFcn = @(obsA,obsB) [obsA(1,1)+obsB(1)^2;
                           obsA(2,1)-obsB(1)^2;
                           obsA(1,2)^2+obsB(1);
                           obsA(2,2)^2-obsB(1)];
```

The output of the actor is the vector `W'*myBasisFcn(obsA,obsB)`, which is the action taken as a result of the given observation. The weight matrix `W` contains the learnable parameters and must have as many rows as the length of the basis function output and as many columns as the dimension of the action space.

Define an initial parameter matrix.

```
W0 = rand(4,3);
```

Create the actor. The first argument is a two-element cell containing both the handle to the custom function and the initial weight matrix. The second and third arguments are, respectively, the observation and action specification objects.

```
actor = rlContinuousDeterministicActor({myBasisFcn,W0},obsInfo,actInfo)
```

```
actor = 
  rlContinuousDeterministicActor with properties:

    ObservationInfo: [2x1 rl.util.RLDataSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

To check your actor, use the `getAction` function to return the action from a given observation, using the current parameter matrix.

```
a = getAction(actor,{rand(2,2),0})
```

```
a = 1x1 cell array
    {3x1 double}
```

```
a{1}
```

```
ans = 3×1

    1.3192
    0.8420
    1.5053
```

Note that the actor does not enforce the set constraint for the discrete set elements.

```
a = getAction(actor,{rand(2,2),-1});
a{1}
```

```
ans = 3×1

    2.7890
    1.8375
    3.0855
```

You can now use the actor (along with a critic) to create an agent for the environment described by the given specification objects. Examples of agents that can work with a mixed observation space, a continuous action space, and use a continuous deterministic actor, are `rlDDPGAgent` and `rlTD3Agent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Deterministic Actor from Recurrent Neural Network

Create observation and action information. You can also obtain these specifications from an environment. For this example, define the observation space as a continuous four-dimensional space, so that a single observation channel carries a column vector containing four doubles, and the action space as a continuous two-dimensional space, so that the action channel carries a column vector containing two doubles.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlNumericSpec([2 1]);
```

A continuous deterministic actor implements a parametrized deterministic policy for a continuous action space. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects, and get the dimension of the observation and action spaces from the environment specification objects. Since this network is recurrent, use a `sequenceInputLayer` as the input layer and include at least one `lstmLayer`.

```
net = [
    sequenceInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(10)
    reluLayer
    lstmLayer(8,OutputMode="sequence")
    fullyConnectedLayer(20)
    fullyConnectedLayer(actInfo.Dimension(1))
    tanhLayer
    ];
```

Convert the network to a `dlnetwork` object and display the number of learnable parameters.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 880

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create a deterministic actor representation for the network.

```
actor = rlContinuousDeterministicActor( ...
    net, ...
    obsInfo, ...
    actInfo);
```

To check your actor, use `getAction` to return the action from a random observation, given the current network weights.

```
a = getAction(actor, ...
    {rand(obsInfo.Dimension)});
a{1}
```

```
ans = 2x1 single column vector

   -0.0742
    0.0158
```

You can use `getState` and `setState` to extract and set the current state of the recurrent neural network in the actor.

```
getState(actor)
```

```
ans=2×1 cell array
    {8x1 single}
    {8x1 single}
```

```
actor = setState(actor, ...
    {-0.01*single(rand(8,1)), ...
      0.01*single(rand(8,1))});
```

To evaluate the actor using sequential observations, use the sequence length (time) dimension. For example, obtain actions for 5 independent sequences each one consisting of 9 sequential observations.

```
[action,state] = getAction(actor, ...
    {rand([obsInfo.Dimension 5 9])});
```

Display the action corresponding to the seventh element of the observation sequence in the fourth sequence.

```
action = action{1};
action(1,1,4,7)
```

```
ans = single
    -0.2426
```

Display the updated state of the recurrent neural network.

```
state
```

```
state=2×1 cell array
    {8x5 single}
    {8x5 single}
```

You can now use the actor (along with a critic) to create an agent for the environment described by the given specification objects. Examples of agents that can work with continuous action and observation spaces, and use a continuous deterministic actor, are `rlDDPGAgent` and `rlTD3Agent`.

For more information on input and output format for recurrent neural networks, see the Algorithms section of `lstmLayer`. For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Version History

Introduced in R2022a

See Also

Functions
`getAction` | `getModel` | `evaluate` | `getActionInfo` | `getObservationInfo`

Objects
`rlNumericSpec` | `rlFiniteSetSpec` | `rlContinuousGaussianActor` | `rlDiscreteCategoricalActor` | `rlTD3Agent` | `rlDDPGAgent`

Topics
“Train DDPG Agent to Control Double Integrator System”
“Train TD3 Agent for PMSM Control”
“Load Predefined Control System Environments”
“Create Policies and Value Functions”
“Reinforcement Learning Agents”

rlContinuousDeterministicRewardFunction

Deterministic reward function approximator object for neural network-based environment

Description

When creating a neural network-based environment using `rlNeuralNetworkEnvironment`, you can specify the reward function approximator using an `rlContinuousDeterministicRewardFunction` object. Do so when you do not know a ground-truth reward signal for your environment but you expect the reward signal to be deterministic.

The reward function approximator object uses a deep neural network as internal approximation model to predict the reward signal for the environment given one of the following input combinations.

- Observations, actions, and next observations
- Observations and actions
- Actions and next observations
- Next observations

To specify a stochastic reward function, use an `rlContinuousGaussianRewardFunction` object.

Creation

Syntax

```
rwdFcnAppx = rlContinuousDeterministicRewardFunction(net,observationInfo,
actionInfo,Name=Value)
```

Description

`rwdFcnAppx = rlContinuousDeterministicRewardFunction(net,observationInfo,actionInfo,Name=Value)` creates the deterministic reward function approximator object `rwdFcnAppx` using the deep neural network `net` and sets the `ObservationInfo` and `ActionInfo` properties.

When creating a reward function you must specify the names of the deep neural network inputs using one of the following combinations of name-value pair arguments.

- `ObservationInputNames`, `ActionInputNames`, and `NextObservationInputNames`
- `ObservationInputNames` and `ActionInputNames`
- `ActionInputNames` and `NextObservationInputNames`
- `NextObservationInputNames`

You can also specify the `UseDevice` property using and an optional name-value pair argument. For example, to use a GPU for prediction, specify `UseDevice="gpu"`.

Input Arguments

`net` — Deep neural network

`dlnetwork` object

Deep neural network with a scalar output value, specified as a `dlnetwork` object.

The input layer names for this network must match the input names specified using the `ObservationInputNames`, `ActionInputNames`, and `NextObservationInputNames`. The dimensions of the input layers must match the dimensions of the corresponding observation and action specifications in `ObservationInfo` and `ActionInfo`, respectively.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ObservationInputNames="velocity"`

`ObservationInputNames` — Observation input layer names

string | string array

Observation input layer names, specified as a string or string array. Specify `ObservationInputNames` when you expect the reward signal to depend on the current environment observation.

The number of observation input names must match the length of `ObservationInfo` and the order of the names must match the order of the specifications in `ObservationInfo`.

`ActionInputNames` — Action input layer names

string | string array

Action input layer names, specified as a string or string array. Specify `ActionInputNames` when you expect the reward signal to depend on the current action value.

The number of action input names must match the length of `ActionInfo` and the order of the names must match the order of the specifications in `ActionInfo`.

`NextObservationInputNames` — Next observation input layer names

string | string array

Next observation input layer names, specified as a string or string array. Specify `NextObservationInputNames` when you expect the reward signal to depend on the next environment observation.

The number of next observation input names must match the length of `ObservationInfo` and the order of the names must match the order of the specifications in `ObservationInfo`.

Properties

`ObservationInfo` — Observation specifications

`rlNumericSpec` object | array `rlNumericSpec` objects

This property is read-only.

Observation specifications, specified as an `rlNumericSpec` object or an array of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

You can extract the observation specifications from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlNumericSpec`.

`ActionInfo` — Action specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object

This property is read-only.

Action specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

You can extract the action specifications from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`UseDevice` — Computation device used for training and simulation

`"cpu"` (default) | `"gpu"`

Computation device used to perform operations such as gradient computation, parameter updates, and prediction during training and simulation, specified as either `"cpu"` or `"gpu"`.

The `"gpu"` option requires both Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use `gpuDevice` (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating a network on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

Object Functions

rlNeuralNetworkEnvironment	Environment model with deep neural network transition models

Examples

Create Deterministic Reward Function and Predict Reward

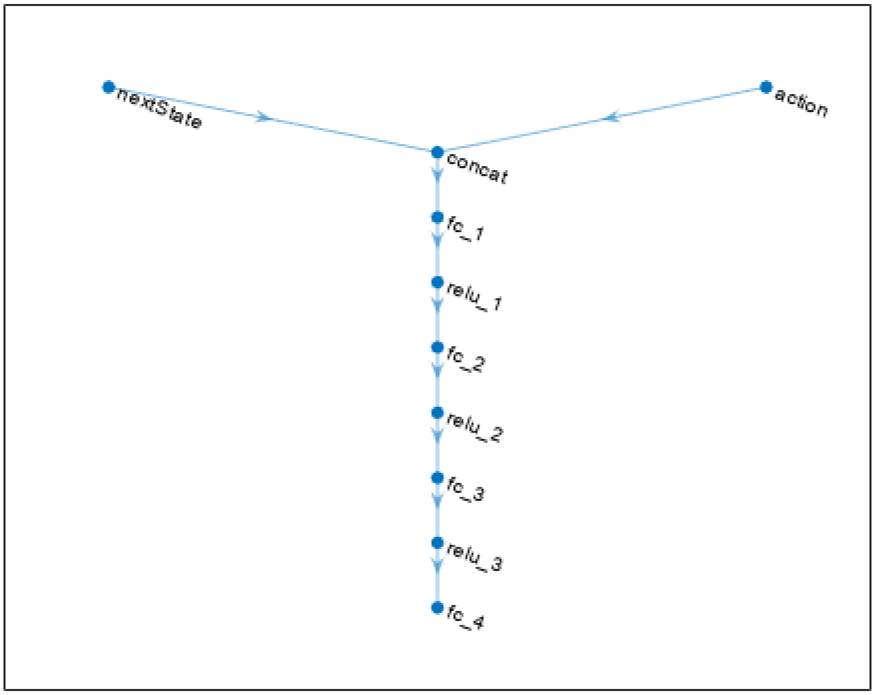
Create an environment interface and extract observation and action specifications. Alternatively, you can create specifications using `rlNumericSpec` and `rlFiniteSetSpec`.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

To approximate the reward function, create a deep neural network. For this example, the network has two input layers, one for the current action and one for the next observations. The single output layer contains a scalar, which represents the value of the predicted reward.

Define each network path as an array of layer objects. Get the dimensions of the observation and action spaces from the environment specifications, and specify a name for the input layers, so you can later explicitly associate them with the appropriate environment channel.

```
actionPath = featureInputLayer( ...
    actInfo.Dimension(1), ...
    Name="action");

nextStatePath = featureInputLayer( ...
    obsInfo.Dimension(1), ...
    Name="nextState");

commonPath = [concatenationLayer(1,2,Name="concat")
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(1)];

net = layerGraph(nextStatePath);
net = addLayers(net,actionPath);
net = addLayers(net,commonPath);

net = connectLayers(net,"nextState","concat/in1");
net = connectLayers(net,"action","concat/in2");

plot(net)
```



Create a `dlnetwork` object and display the number of weights.

```
net = dlnetwork(net);
summary(net);
```

```
   Initialized: true

   Number of learnables: 8.7k

   Inputs:
      1   'nextState'   4 features
      2   'action'      1 features
```

Create a deterministic transition function object.

```
rwdFcnAppx = rlContinuousDeterministicRewardFunction(...
    net,obsInfo,actInfo,...
    ActionInputNames="action", ...
    NextObservationInputNames="nextState");
```

Using this reward function object, you can predict the next reward value based on the current action and next observation. For example, predict the reward for a random action and next observation. Since, for this example, only the action and the next observation influence the reward, use an empty cell array for the current observation.

```
act = rand(actInfo.Dimension);
nxtobs = rand(obsInfo.Dimension);
reward = predict(rwdFcnAppx, {}, {act}, {nxtobs})
reward = single
    0.1034
```

To predict the reward, you can also use `evaluate`.

```
reward_ev = evaluate(rwdFcnAppx, {act,nxtobs} )
```

```
reward_ev = 1x1 cell array
    {[0.1034]}
```

Version History

Introduced in R2022a

See Also

Functions
`evaluate` | `gradient` | `accelerate`

Objects
`rlNeuralNetworkEnvironment` | `rlContinuousDeterministicTransitionFunction` | `rlContinuousGaussianTransitionFunction` | `rlContinuousGaussianRewardFunction` | `rlIsDoneFunction` | `rlMBPOAgent` | `rlMBPOAgentOptions`

Topics
"Train MBPO Agent to Balance Cart-Pole System"
"Model-Based Policy Optimization (MBPO) Agents"

rlContinuousDeterministicTransitionFunction

Deterministic transition function approximator object for neural network-based environment

Description

When creating a neural network-based environment using `rlNeuralNetworkEnvironment`, you can specify deterministic transition function approximators using `rlContinuousDeterministicTransitionFunction` objects.

A transition function approximator object uses a deep neural network to predict the next observations based on the current observations and actions.

To specify stochastic transition function approximators, use `rlContinuousGaussianTransitionFunction` objects.

Creation

Syntax

```
tsnFcnAppx = rlContinuousDeterministicTransitionFunction(net,observationInfo,
actionInfo,Name=Value)
```

Description

`tsnFcnAppx = rlContinuousDeterministicTransitionFunction(net,observationInfo,actionInfo,Name=Value)` creates a deterministic transition function approximator object using the deep neural network `net` and sets the `ObservationInfo` and `ActionInfo` properties.

When creating a deterministic transition function approximator you must specify the names of the deep neural network inputs and outputs using the `ObservationInputNames`, `ActionInputNames`, and `NextObservationOutputNames` name-value pair arguments.

You can also specify the `PredictDiff` and `UseDevice` properties using optional name-value pair arguments. For example, to use a GPU for prediction, specify `UseDevice="gpu"`.

Input Arguments

net — Deep neural network
`dlnetwork` object

Deep neural network, specified as a `dlnetwork` object.

The input layer names for this network must match the input names specified using `ObservationInputNames` and `ActionInputNames`. The dimensions of the input layers must match the dimensions of the corresponding observation and action specifications in `ObservationInfo` and `ActionInfo`, respectively.

The output layer names for this network must match the output names specified using `NextObservationOutputNames`. The dimensions of the input layers must match the dimensions of the corresponding observation specifications in `ObservationInfo`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ObservationInputNames="velocity"`

`ObservationInputNames` — Observation input layer names
string | string array

Observation input layer names, specified as a string or string array.

The number of observation input names must match the length of `ObservationInfo` and the order of the names must match the order of the specifications in `ObservationInfo`.

`ActionInputNames` — Action input layer names
string | string array

Action input layer names, specified as a string or string array.

The number of action input names must match the length of `ActionInfo` and the order of the names must match the order of the specifications in `ActionInfo`.

`NextObservationOutputNames` — Next observation output layer names
string | string array

Next observation output layer names, specified as a string or string array.

The number of next observation output names must match the length of `ObservationInfo` and the order of the names must match the order of the specifications in `ObservationInfo`.

Properties

`ObservationInfo` — Observation specifications
`rlNumericSpec` object | array `rlNumericSpec` objects

This property is read-only.

Observation specifications, specified as an `rlNumericSpec` object or an array of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

You can extract the observation specifications from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlNumericSpec`.

`ActionInfo` — Action specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object

This property is read-only.

Action specifications, specified as an rlFiniteSetSpec or rlNumericSpec object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

You can extract the action specifications from an existing environment or agent using getActionInfo. You can also construct the specification manually using rlFiniteSetSpec or rlNumericSpec.

PredictDiff — Option to predict the difference between the current observation and the next observation

false (default) | true

Option to predict the difference between the current observation and the next observation, specified as one of the following logical values.

- false — Select this option if net outputs the value of the next observation.
- true — Select this option if net outputs the difference between the next observation and the current observation. In this case, the predict function computes the next observation by adding the current observation to the output of net.

UseDevice — Computation device used for training and simulation

"cpu" (default) | "gpu"

Computation device used to perform operations such as gradient computation, parameter updates, and prediction during training and simulation, specified as either "cpu" or "gpu".

The "gpu" option requires both Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use gpuDevice (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating a network on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

Object Functions

rlNeuralNetworkEnvironment	Environment model with deep neural network transition models

Examples

Create Deterministic Transition Function and Predict Next Observation

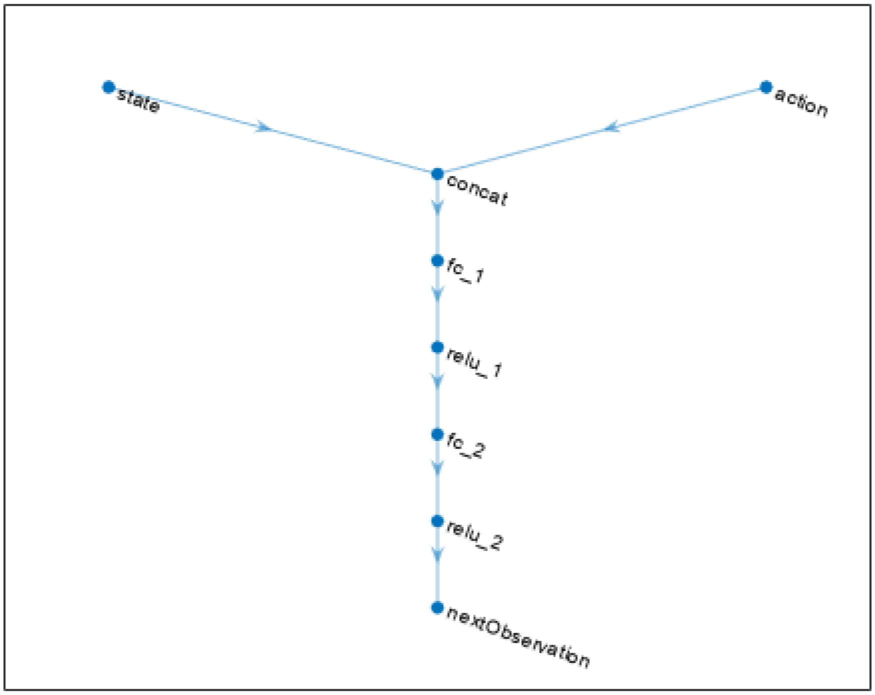
Create an environment interface and extract observation and action specifications. Alternatively, you can create specifications using rlNumericSpec and rlFiniteSetSpec.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

To approximate the transition function, create a deep neural network. The network has two input layers, one for the current observation channel and one for the current action channel. The single output layer is for the predicted next observation.

Define each network path as an array of layer objects. Get the dimensions of the observation and action spaces from the environment specifications, and specify a name for the input and output layers, so you can later explicitly associate them with the appropriate environment channel.

```
statePath = featureInputLayer(obsInfo.Dimension(1),Name="state");
actionPath = featureInputLayer(actInfo.Dimension(1),Name="action");
commonPath = [concatenationLayer(1,2,Name="concat")
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(obsInfo.Dimension(1),Name="nextObservation")];

tsnNet = layerGraph(statePath);
tsnNet = addLayers(tsnNet,actionPath);
tsnNet = addLayers(tsnNet,commonPath);

tsnNet = connectLayers(tsnNet,"state","concat/in1");
tsnNet = connectLayers(tsnNet,"action","concat/in2");

plot(tsnNet)
```



Create a `dlnetwork` object, and display the number of weights.

```
tsnNet = dlnetwork(tsnNet);
summary(tsnNet)
```

```
   Initialized: true

   Number of learnables: 4.8k

   Inputs:
      1   'state'    4 features
      2   'action'   1 features
```

Create a deterministic transition function object.

```
tsnFcnAppx = rlContinuousDeterministicTransitionFunction(...
    tsnNet,obsInfo,actInfo,...
    ObservationInputNames="state", ...
    ActionInputNames="action", ...
    NextObservationOutputNames="nextObservation");
```

Using this transition function object, you can predict the next observation based on the current observation and action. For example, predict the next observation for a random observation and action.

```
obs = rand(obsInfo.Dimension);
act = rand(actInfo.Dimension);
nextObsP = predict(tsnFcnAppx,{obs},{act})
nextObsP = 1x1 cell array
    {4x1 single}
```

```
nextObsP{1}
```

```
ans = 4x1 single column vector

   -0.1172
    0.1168
    0.0493
   -0.0155
```

You can also obtain the same result using `evaluate`.

```
nextObsE = evaluate(tsnFcnAppx,{obs,act})
```

```
nextObsE = 1x1 cell array
    {4x1 single}
```

```
nextObsE{1}
```

```
ans = 4x1 single column vector

   -0.1172
    0.1168
    0.0493
   -0.0155
```

Version History

Introduced in R2022a

See Also

Functions

`evaluate` | `gradient` | `accelerate`

Objects

`rlNeuralNetworkEnvironment` | `rlContinuousGaussianTransitionFunction` | `rlContinuousDeterministicRewardFunction` | `rlContinuousGaussianRewardFunction` | `rlIsDoneFunction` | `rlMBPOAgent` | `rlMBPOAgentOptions`

Topics

"Train MBPO Agent to Balance Cart-Pole System"
"Model-Based Policy Optimization (MBPO) Agents"

rlContinuousGaussianActor

Stochastic Gaussian actor with a continuous action space for reinforcement learning agents

Description

This object implements a function approximator to be used as a stochastic actor within a reinforcement learning agent with a continuous action space. A continuous Gaussian actor takes an environment observation as input and returns as output a random action sampled from a parametrized Gaussian probability distribution, thereby implementing a parametrized stochastic policy. After you create an `rlContinuousGaussianActor` object, use it to create a suitable agent, such as an `rlACAgent` or `rlPGAgent` agent. For more information on creating representations, see “Create Policies and Value Functions”.

Creation

Syntax

```
actor = rlContinuousGaussianActor(net,observationInfo,
actionInfo,ActionMeanOutputNames=
netMeanActName,ActionStandardDeviationOutputNames=netStdvActName)
actor = rlContinuousGaussianActor(net,observationInfo,
actionInfo,ActionMeanOutputNames=
netMeanActName,ActionStandardDeviationOutputNames=
netStdActName,ObservationInputNames=netObsNames)
```

```
actor = rlContinuousGaussianActor( ___ ,UseDevice=useDevice)
```

Description

`actor = rlContinuousGaussianActor(net,observationInfo,actionInfo,ActionMeanOutputNames=netMeanActName,ActionStandardDeviationOutputNames=netStdvActName)` creates a Gaussian stochastic actor with a continuous action space using the deep neural network `net` as approximation model. Here, `net` must have two differently named output layers, each with as many elements as the number of dimensions of the action space, as specified in `actionInfo`. The two output layers must return the mean and standard deviation of each component of the action, respectively. The actor uses the output from these layers, according to the names specified in the strings `netMeanActName` and `netStdActName`, to represent the Gaussian probability distribution from which the action is sampled. This syntax sets the `ObservationInfo` and `ActionInfo` properties of `actor` to the input arguments `observationInfo` and `actionInfo`, respectively.

Note `actor` does not enforce constraints set by the action specification. When using this actor anywhere else than in a SAC agent, you must enforce action space constraints within the environment.

```
actor = rlContinuousGaussianActor(net,observationInfo,
actionInfo,ActionMeanOutputNames=
```

`netMeanActName,ActionStandardDeviationOutputNames=`
`netStdActName,ObservationInputNames=netObsNames)` specifies the names of the network input layers to be associated with the environment observation channels. The function assigns, in sequential order, each environment observation channel specified in `observationInfo` to the layer specified by the corresponding name in the string array `netObsNames`. Therefore, the network input layers, ordered as the names in `netObsNames`, must have the same data type and dimensions as the observation specifications, as ordered in `observationInfo`.

`actor = rlContinuousGaussianActor( ___ ,UseDevice=useDevice)` specifies the device used to perform computational operations on the `actor` object, and sets the `UseDevice` property of `actor` to the `useDevice` input argument. You can use this syntax with any of the previous input-argument combinations.

Input Arguments

net — Deep neural network

array of `Layer` objects | `layerGraph` object | `DAGNetwork` object | `SeriesNetwork` object | `dlNetwork` object (preferred)

Deep neural network used as the underlying approximation model within the actor. It must have as many input layers as the number of environment observation channels (with each input layer receiving input from an observation channel). The network must have two differently named output layers each with as many elements as the number of dimensions of the action space, as specified in `actionInfo`. The two output layers return the mean and standard deviation of each component of the action. The actor uses these layers, according to the names specified in the strings `netMeanActName` and `netStdActName`, to represent the Gaussian probability distribution from which the action is sampled.

Note Since standard deviations must be nonnegative, the output layer that returns the standard deviations must be a softplus or ReLU layer, to enforce nonnegativity. Also, unless the actor is used in a SAC agent, the mean values must fall within the range of the action. In this case, to scale the mean values to the output range, use a scaling layer as the output layer for the mean values, preceded by an hyperbolic tangent layer. SAC agents automatically read the action range from the `UpperLimit` and `LowerLimit` properties of the action specification and then internally scale the distribution and bounds the action. Therefore, if the actor must be used in a SAC agent, do not add any layer that scales or bounds the mean values output.

You can specify the network as one of the following:

- Array of `Layer` objects
- `layerGraph` object
- `DAGNetwork` object
- `SeriesNetwork` object
- `dlnetwork` object

Note Among the different network representation options, `dlnetwork` is preferred, since it has built-in validation checks and supports automatic differentiation. If you pass another network object as an input argument, it is internally converted to a `dlnetwork` object. However, best practice is to convert other representations to `dlnetwork` explicitly *before* using it to create a critic or an actor for a reinforcement learning agent. You can do so using `dlnet=dlnetwork(net)`, where `net` is any neural network object from the Deep Learning Toolbox. The resulting `dlnet` is the `dlnetwork` object that you use for your critic or actor. This practice allows a greater level of insight and control for cases in which the conversion is not straightforward and might require additional specifications.

`rlContinuousGaussianActor` objects support recurrent deep neural networks.

The learnable parameters of the actor are the weights of the deep neural network. For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

`netMeanActName` — Names of the network output layers corresponding to the mean values of the action channel

string | character vector

Names of the network output layers corresponding to the mean values of the action channel, specified as a string or character vector. The actor uses this name to select the network output layer that returns the mean values of each elements of the action channel. Therefore, this network output layer must be named as indicated in `netMeanActName`. Furthermore, it must be a scaling layer that scales the returned mean values to the desired action range.

Example: `"myNetOut_Force_Mean_Values"`

`netStdvActName` — Names of the network output layers corresponding to the standard deviations of the action channel

string | character vector

Names of the network output layers corresponding to the standard deviations of the action channel, specified as a string or character vector. The actor uses this name to select the network output layer that returns the standard deviations of each elements of the action channel. Therefore, this network output layer must be named as indicated in `netStdvActName`. Furthermore, it must be a softplus or ReLU layer, to enforce nonnegativity of the returned standard deviations.

Example: `"myNetOut_Force_Standard_Deviations"`

`netObsNames` — Network input layers names corresponding to the environment observation channels

string array | cell array of character vectors

Network input layers names corresponding to the environment observation channels, specified as a string array or a cell array of character vectors. When you use this argument after `'ObservationInputNames'`, the function assigns, in sequential order, each environment observation channel specified in `observationInfo` to each network input layer specified by the corresponding name in the string array `netObsNames`. Therefore, the network input layers, ordered as the names in `netObsNames`, must have the same data type and dimensions as the observation channels, as ordered in `observationInfo`.

Example: `{"NetInput1_airspeed","NetInput2_altitude"}`

Properties

`ObservationInfo` — Observation specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

`rlContinuousGaussianActor` sets the `ObservationInfo` property of `actor` to the input `observationInfo`.

You can extract `ObservationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually.

`ActionInfo` — Action specifications
`rlNumericSpec` object

Action specifications, specified as an `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

`rlContinuousGaussianActor` sets the `ActionInfo` property of `critic` to the input `actionInfo`.

You can extract `ActionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specifications manually.

`UseDevice` — Computation device used for training and simulation
`"cpu"` (default) | `"gpu"`

Computation device used to perform operations such as gradient computation, parameter update and prediction during training and simulation, specified as either `"cpu"` or `"gpu"`.

The `"gpu"` option requires both Parallel Computing Toolbox software and a CUDA enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use `gpuDevice` (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating an agent on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

To speed up training by using parallel processing over multiple cores, you do not need to use this argument. Instead, when training your agent, use an `rlTrainingOptions` object in which the `UseParallel` option is set to `true`. For more information about training using multicore processors and GPUs for training, see “Train Agents Using Parallel Computing and GPUs”.

Example: `'UseDevice',"gpu"`

Object Functions

rlACAgent	Actor-critic (AC) reinforcement learning agent
rlPGAgent	Policy gradient (PG) reinforcement learning agent
rlPPOAgent	Proximal policy optimization (PPO) reinforcement learning agent

rlSACAgent	Soft actor-critic (SAC) reinforcement learning agent
getAction	Obtain action from agent, actor, or policy object given environment observations
evaluate	Evaluate function approximator object given observation (or observation-action) input data
gradient	Evaluate gradient of function approximator object given observation and action input data
accelerate	Option to accelerate computation of gradient for approximator object based on neural network
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object
setModel	Set function approximation model for actor or critic
getModel	Get function approximator model from actor or critic

Examples

Create Continuous Gaussian Actor

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous six-dimensional space, so that there is a single observation channel that carries a column vector containing five doubles.

```
obsInfo = rlNumericSpec([5 1]);
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous three-dimensional space, so that the action channel carries a column vector containing three doubles, each between -10 and 10.

```
actInfo = rlNumericSpec([3 1], ...
    LowerLimit=-10, ...
    UpperLimit=10);
```

A continuous Gaussian actor implements a parametrized stochastic policy for a continuous action space. This actor takes an observation as input and returns as output a random action sampled from a Gaussian probability distribution.

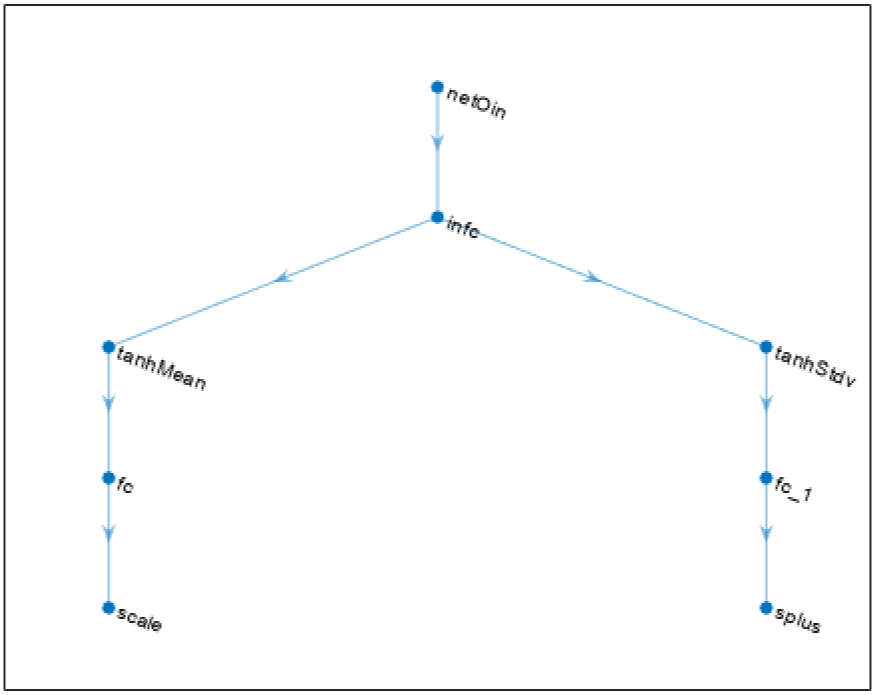
To approximate the mean values and standard deviations of the Gaussian distribution, you must use a neural network with two output layers, each having as many elements as the dimension of the action space. One output layer must return a vector containing the mean values for each action dimension. The other must return a vector containing the standard deviation for each action dimension.

Note that standard deviations must be nonnegative and mean values must fall within the range of the action. Therefore the output layer that returns the standard deviations must be a softplus or ReLU layer, to enforce nonnegativity, while the output layer that returns the mean values must be a scaling layer, to scale the mean values to the output range. However, do not add a `tanhLayer` as the last nonlinear layer in the mean output path if you are going to use the actor within a SAC agent. For more information see “Soft Actor-Critic (SAC) Agents”.

For this example the environment has only one observation channel and therefore the network has only one input layer. Note that `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` return the number of dimensions of the observation and action spaces, respectively, regardless of whether they are arranged as row vectors, column vectors, or matrices.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% Input path layers
inPath = [
    featureInputLayer( ...
        prod(obsInfo.Dimension), ...
        Name="netOin")
    fullyConnectedLayer( ...
        prod(actInfo.Dimension), ...
        Name="infc")
    ];

% Path layers for mean value
% Using scalingLayer to scale range from (-1,1) to (-10,10)
meanPath = [
    tanhLayer(Name="tanhMean");
    fullyConnectedLayer(prod(actInfo.Dimension));
    scalingLayer(Name="scale", ...
    Scale=actInfo.UpperLimit)
    ];

% Path layers for standard deviations
% Using softplus layer to make them non negative
sdevPath = [
    tanhLayer(Name="tanhStdv");
    fullyConnectedLayer(prod(actInfo.Dimension));
    softplusLayer(Name="splus")
    ];

% Add layers to network object
net = layerGraph(inPath);
net = addLayers(net,meanPath);
net = addLayers(net,sdevPath);

% Connect layers
net = connectLayers(net,"infc","tanhMean/in");
net = connectLayers(net,"infc","tanhStdv/in");

% Plot the network
plot(net)
```



Convert the network to a `dlnetwork` object and display the number of learnable parameters (weights).

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 42

   Inputs:
      1   'netOin'   5 features
```

Create the actor with `rlContinuousGaussianActor`, using the network, the observation and action specification objects, and the names of the network input and output layers.

```
actor = rlContinuousGaussianActor(net, obsInfo, actInfo, ...
    ActionMeanOutputNames="scale",...
    ActionStandardDeviationOutputNames="splus",...
    ObservationInputNames="netOin");
```

To check your actor, use `getAction` to return an action from a random observation vector, using the current network weights. Each of the three elements of the action vector is a random sample from the Gaussian distribution with mean and standard deviation calculated, as a function of the current observation, by the neural network.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = 3x1 single column vector

  -12.0285
    1.7628
   10.8733
```

To return the Gaussian distribution of the action, given an observation, use `evaluate`.

```
dist = evaluate(actor,{rand(obsInfo.Dimension)});
```

Display the vector of mean values.

```
dist{1}
```

```
ans = 3x1 single column vector

   -5.6127
    3.9449
    9.6213
```

Display the vector of standard deviations.

```
dist{2}
```

```
ans = 3x1 single column vector

    0.8516
    0.8366
    0.7004
```

You can now use the actor (along with a critic) to create an agent for the environment described by the given specification objects. Examples of agents that can work with continuous action and observation spaces, and use a continuous Gaussian actor, are `rlACAgent`, `rlPGAgent`, `rlSACAgent`, `rlPPOAgent`, and `rlTRPOAgent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Version History

Introduced in R2022a

See Also

Functions

`getAction` | `getModel` | `evaluate` | `getActionInfo` | `getObservationInfo`

Objects

`rlNumericSpec` | `rlFiniteSetSpec` | `rlDiscreteCategoricalActor` | `rlContinuousDeterministicActor` | `rlPGAgent` | `rlACAgent` | `rlSACAgent` | `rlPPOAgent` | `rlTRPOAgent`

Topics
“Train SAC Agent for Ball Balance Control”
“Create Policies and Value Functions”
“Reinforcement Learning Agents”

rlContinuousGaussianRewardFunction

Stochastic Gaussian reward function approximator object for neural network-based environment

Description

When creating a neural network-based environment using `rlNeuralNetworkEnvironment`, you can specify the reward function approximator using an `rlContinuousDeterministicRewardFunction` object. Do so when you do not know a ground-truth reward signal for your environment and you expect the reward signal to be stochastic.

The reward function object uses a deep neural network as internal approximation model to predict the reward signal for the environment given one of the following input combinations.

- Observations, actions, and next observations
- Observations and actions
- Actions and next observations
- Next observations

To specify a deterministic reward function approximator, use an `rlContinuousDeterministicRewardFunction` object.

Creation

Syntax

```
rwdFcnAppx = rlContinuousGaussianRewardFunction(net,observationInfo,
actionInfo,Name=Value)
```

Description

`rwdFcnAppx = rlContinuousGaussianRewardFunction(net,observationInfo,actionInfo,Name=Value)` creates a stochastic reward function using the deep neural network `net` and sets the `ObservationInfo` and `ActionInfo` properties.

When creating a reward function you must specify the names of the deep neural network inputs using one of the following combinations of name-value pair arguments.

- `ObservationInputNames`, `ActionInputNames`, and `NextObservationInputNames`
- `ObservationInputNames` and `ActionInputNames`
- `ActionInputNames` and `NextObservationInputNames`
- `NextObservationInputNames`

You must also specify the names of the deep neural network outputs using the `RewardMeanOutputName` and `RewardStandardDeviationOutputName` name-value pair arguments.

You can also specify the `UseDevice` property using an optional name-value pair argument. For example, to use a GPU for prediction, specify `UseDevice="gpu"`.

Input Arguments

`net` — Deep neural network
`dlnetwork` object

Deep neural network with a scalar output value, specified as a `dlnetwork` object.

The input layer names for this network must match the input names specified using the `ObservationInputNames`, `ActionInputNames`, and `NextObservationInputNames`. The dimensions of the input layers must match the dimensions of the corresponding observation and action specifications in `ObservationInfo` and `ActionInfo`, respectively.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ObservationInputNames="velocity"`

`ObservationInputNames` — Observation input layer names
string | string array

Observation input layer names, specified as a string or string array. Specify `ObservationInputNames` when you expect the reward signal to depend on the current environment observation.

The number of observation input names must match the length of `ObservationInfo` and the order of the names must match the order of the specifications in `ObservationInfo`.

`ActionInputNames` — Action input layer names
string | string array

Action input layer names, specified as a string or string array. Specify `ActionInputNames` when you expect the reward signal to depend on the current action value.

The number of action input names must match the length of `ActionInfo` and the order of the names must match the order of the specifications in `ActionInfo`.

`NextObservationInputNames` — Next observation input layer names
string | string array

Next observation input layer names, specified as a string or string array. Specify `NextObservationInputNames` when you expect the reward signal to depend on the next environment observation.

The number of next observation input names must match the length of `ObservationInfo` and the order of the names must match the order of the specifications in `ObservationInfo`.

`RewardMeanOutputName` — Reward mean output layer name
string

Reward mean output layer name, specified as a string.

`RewardStandardDeviationOutputName` — Reward standard deviation output layer name

string

Reward standard deviation output layer name, specified as a string.

Properties

`ObservationInfo` — Observation specifications

`rlNumericSpec` object | array `rlNumericSpec` objects

This property is read-only.

Observation specifications, specified as an `rlNumericSpec` object or an array of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

You can extract the observation specifications from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlNumericSpec`.

`ActionInfo` — Action specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object

This property is read-only.

Action specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

You can extract the action specifications from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`UseDevice` — Computation device used for training and simulation

`"cpu"` (default) | `"gpu"`

Computation device used to perform operations such as gradient computation, parameter updates, and prediction during training and simulation, specified as either `"cpu"` or `"gpu"`.

The `"gpu"` option requires both Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use `gpuDevice` (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating a network on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

Object Functions

rlNeuralNetworkEnvironment	Environment model with deep neural network transition models

Examples

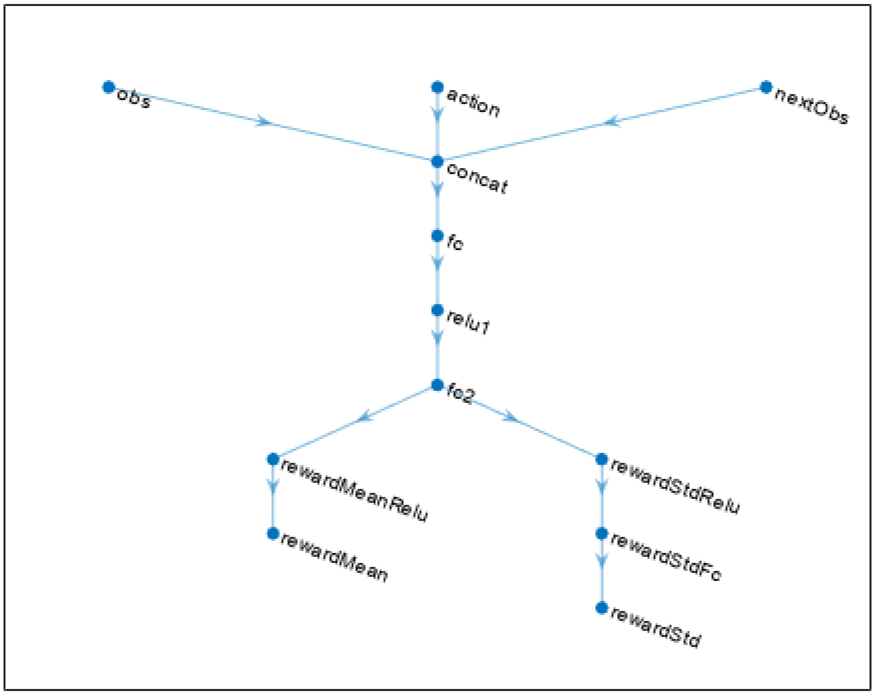
Create Stochastic Reward Function and Predict Reward

Create an environment interface and extract observation and action specifications. Alternatively, you can create specifications using `rlNumericSpec` and `rlFiniteSetSpec`.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a deep neural network. The network has two input channels, one for the current action and one for the next observations. The single output channel is for the predicted reward value.

```
statePath = featureInputLayer(obsInfo.Dimension(1),Name="obs");
actionPath = featureInputLayer(actInfo.Dimension(1),Name="action");
nextStatePath = featureInputLayer(obsInfo.Dimension(1),Name="nextObs");
commonPath = [concatenationLayer(1,3,Name="concat")
    fullyConnectedLayer(32,Name="fc")
    reluLayer(Name="relu1")
    fullyConnectedLayer(32,Name="fc2")];

meanPath = [reluLayer(Name="rewardMeanRelu")
    fullyConnectedLayer(1,Name="rewardMean")];
stdPath = [reluLayer(Name="rewardStdRelu")
    fullyConnectedLayer(1,Name="rewardStdFc")
    softplusLayer(Name="rewardStd")];

rwdNet = layerGraph(statePath);
rwdNet = addLayers(rwdNet,actionPath);
rwdNet = addLayers(rwdNet,nextStatePath);
rwdNet = addLayers(rwdNet,commonPath);
rwdNet = addLayers(rwdNet,meanPath);
rwdNet = addLayers(rwdNet,stdPath);

rwdNet = connectLayers(rwdNet,"nextObs","concat/in1");
rwdNet = connectLayers(rwdNet,"action","concat/in2");
rwdNet = connectLayers(rwdNet,"obs","concat/in3");
rwdNet = connectLayers(rwdNet,"fc2","rewardMeanRelu");
rwdNet = connectLayers(rwdNet,"fc2","rewardStdRelu");

plot(rwdNet)
```



Create a `dlnetwork` object.

```
rwdNet = dlnetwork(rwdNet);
```

Create a stochastic reward function object.

```
rwdFncAppx = rlContinuousGaussianRewardFunction(...
    rwdNet,obsInfo,actInfo,...
    ObservationInputNames="obs",...
    ActionInputNames="action", ...
    NextObservationInputNames="nextObs", ...
    RewardMeanOutputNames="rewardMean", ...
    RewardStandardDeviationOutputNames="rewardStd");
```

Using this reward function object, you can predict the next reward value based on the current action and next observation. For example, predict the reward for a random action and next observation. The reward value is sampled from a Gaussian distribution with the mean and standard deviation output by the reward network.

```
obs = rand(obsInfo.Dimension);
act = rand(actInfo.Dimension);
nextObs = rand(obsInfo.Dimension(1),1);
predRwd = predict(rwdFncAppx,{obs},{act},{nextObs})
```

```
predRwd = single
    -0.1308
```

You can obtain the mean value and standard deviation of the Gaussian distribution for the predicted reward using `evaluate`.

```
predRwdDist = evaluate(rwdFncAppx,{obs,act,nextObs})
```

```
predRwdDist=1×2 cell array
    {[-0.0995]}    {[0.6195]}
```

Version History

Introduced in R2022a

See Also

Functions
`evaluate` | `gradient` | `accelerate`

Objects
`rlNeuralNetworkEnvironment` | `rlContinuousDeterministicTransitionFunction` | `rlContinuousGaussianTransitionFunction` | `rlIsDoneFunction` | `rlMBPOAgent` | `rlMBPOAgentOptions`

Topics
"Train MBPO Agent to Balance Cart-Pole System"
"Model-Based Policy Optimization (MBPO) Agents"

rlContinuousGaussianTransitionFunction

Stochastic Gaussian transition function approximator object for neural network-based environment

Description

When creating a neural network-based environment using `rlNeuralNetworkEnvironment`, you can specify stochastic transition function approximators using `rlContinuousDeterministicTransitionFunction` objects.

A transition function approximator object uses a deep neural network as internal approximation model to predict the next observations based on the current observations and actions.

To specify deterministic transition function approximators, use `rlContinuousGaussianTransitionFunction` objects.

Creation

Syntax

```
tsnFcnAppx = rlContinuousGaussianTransitionFunction(net,observationInfo,
actionInfo,Name=Value)
```

Description

`tsnFcnAppx = rlContinuousGaussianTransitionFunction(net,observationInfo, actionInfo,Name=Value)` creates the stochastic transition function approximator object `tsnFcnAppx` using the deep neural network `net` and sets the `ObservationInfo` and `ActionInfo` properties.

When creating a stochastic transition function approximator, you must specify the names of the deep neural network inputs and outputs using the `ObservationInputNames`, `ActionInputNames`, `NextObservationMeanOutputNames`, and `NextObservationStandardDeviationOutputNames` name-value pair arguments.

You can also specify the `PredictDiff` and `UseDevice` properties using optional name-value pair arguments. For example, to use a GPU for prediction, specify `UseDevice="gpu"`.

Input Arguments

net — Deep neural network
`dlnetwork` object

Deep neural network, specified as a `dlnetwork` object.

The input layer names for this network must match the input names specified using `ObservationInputNames` and `ActionInputNames`. The dimensions of the input layers must match the dimensions of the corresponding observation and action specifications in `ObservationInfo` and `ActionInfo`, respectively.

The output layer names for this network must match the output names specified using NextObservationOutputNames. The dimensions of the input layers must match the dimensions of the corresponding observation specifications in ObservationInfo.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: ObservationInputNames="velocity"

ObservationInputNames — Observation input layer names
string | string array

Observation input layer names, specified as a string or string array.

The number of observation input names must match the length of ObservationInfo and the order of the names must match the order of the specifications in ObservationInfo.

ActionInputNames — Action input layer names
string | string array

Action input layer names, specified as a string or string array.

The number of action input names must match the length of ActionInfo and the order of the names must match the order of the specifications in ActionInfo.

NextObservationMeanOutputNames — Next observation mean output layer names
string | string array

Next observation mean output layer names, specified as a string or string array.

The number of next observation mean output names must match the length of ObservationInfo and the order of the names must match the order of the specifications in ObservationInfo.

NextObservationStandardDeviationOutputNames — Next observation standard deviation output layer names
string | string array

Next observation standard deviation output layer names, specified as a string or string array.

The number of next observation standard deviation output names must match the length of ObservationInfo and the order of the names must match the order of the specifications in ObservationInfo.

Properties

ObservationInfo — Observation specifications
rlNumericSpec object | array rlNumericSpec objects

This property is read-only.

Observation specifications, specified as an rlNumericSpec object or an array of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

You can extract the observation specifications from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlNumericSpec`.

`ActionInfo` — Action specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object

This property is read-only.

Action specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

You can extract the action specifications from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`UseDevice` — Computation device used for training and simulation

`"cpu"` (default) | `"gpu"`

Computation device used to perform operations such as gradient computation, parameter updates, and prediction during training and simulation, specified as either `"cpu"` or `"gpu"`.

The `"gpu"` option requires both Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use `gpuDevice` (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating a network on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

`PredictDiff` — Option to predict the difference between the current observation and the next observation

`false` (default) | `true`

Option to predict the difference between the current observation and the next observation, specified as one of the following logical values.

- `false` — Select this option if `net` outputs the value of the next observation.
- `true` — Select this option if `net` outputs the difference between the next observation and the current observation. In this case, the `predict` function computes the next observation by adding the current observation to the output of `net`.

Object Functions

rlNeuralNetworkEnvironment	Environment model with deep neural network transition models

Examples

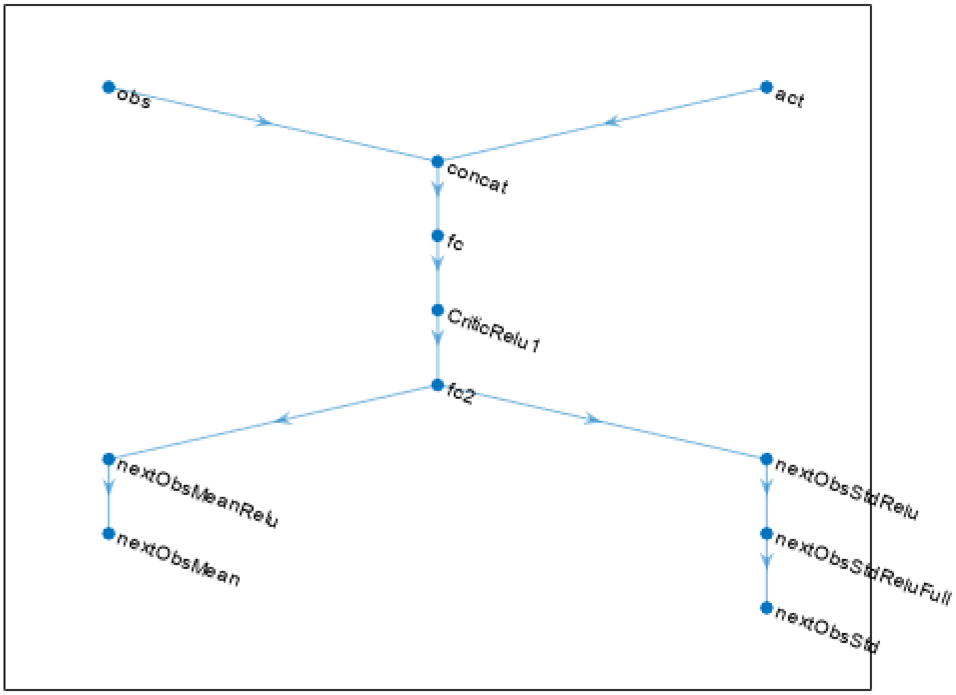
Create Stochastic Transition Function and Predict Next Observation

Create an environment interface and extract observation and action specifications. Alternatively, you can create specifications using `rlNumericSpec` and `rlFiniteSetSpec`.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Define the layers for the deep neural network. The network has two input channels, one for the current observations and one for the current actions. The output of the network is the predicted Gaussian distribution for each next observation. The two output channels correspond to the means and standard deviations of these distribution.

```
statePath = featureInputLayer(obsInfo.Dimension(1),Name="obs");
actionPath = featureInputLayer(actInfo.Dimension(1),Name="act");

commonPath = [concatenationLayer(1,2,Name="concat")
  fullyConnectedLayer(32,Name="fc")
  reluLayer(Name="CriticRelu1")
  fullyConnectedLayer(32,Name="fc2")];

meanPath = [reluLayer(Name="nextObsMeanRelu")
  fullyConnectedLayer(obsInfo.Dimension(1),Name="nextObsMean")];
stdPath = [reluLayer(Name="nextObsStdRelu")
  fullyConnectedLayer(obsInfo.Dimension(1),Name="nextObsStdReluFull")
  softplusLayer(Name="nextObsStd")];

tsnNet = layerGraph(statePath);
tsnNet = addLayers(tsnNet,actionPath);
tsnNet = addLayers(tsnNet,commonPath);
tsnNet = addLayers(tsnNet,meanPath);
tsnNet = addLayers(tsnNet,stdPath);

tsnNet = connectLayers(tsnNet,"obs","concat/in1");
tsnNet = connectLayers(tsnNet,"act","concat/in2");
tsnNet = connectLayers(tsnNet,"fc2","nextObsMeanRelu");
tsnNet = connectLayers(tsnNet,"fc2","nextObsStdRelu");

plot(tsnNet)
```



Create a `dlnetwork` object.

```
tsnNet = dlnetwork(tsnNet);
```

Create a stochastic transition function object.

```
tsnFcnAppx = rlContinuousGaussianTransitionFunction(tsnNet,obsInfo,actInfo, ...
    ObservationInputNames="obs",...
    ActionInputNames="act",...
    NextObservationMeanOutputNames="nextObsMean",...
    NextObservationStandardDeviationOutputNames="nextObsStd");
```

Using this transition function object, you can predict the next observation based on the current observation and action. For example, predict the next observation for a random observation and action. The next observation values are sampled from Gaussian distributions with the means and standard deviations output by the transition network.

```
observation = rand(obsInfo.Dimension);
action = rand(actInfo.Dimension);
nextObs = predict(tsnFcnAppx,{observation},{action})
```

```
nextObs = 1x1 cell array
    {4x1 single}
```

```
nextObs{1}
```

```
ans = 4x1 single column vector
```

```
    1.2414
    0.7307
   -0.5588
   -0.9567
```

You can also obtain the mean value and standard deviation of the Gaussian distribution of the predicted next observation using `evaluate`.

```
nextObsDist = evaluate(tsnFcnAppx,{observation,action})
```

```
nextObsDist=1×2 cell array
    {4x1 single}    {4x1 single}
```

Version History

Introduced in R2022a

See Also

Functions

`evaluate` | `gradient` | `accelerate`

Objects

`rlNeuralNetworkEnvironment` | `rlContinuousDeterministicTransitionFunction` | `rlContinuousDeterministicRewardFunction` | `rlIsDoneFunction` | `rlMBPOAgent` | `rlMBPOAgentOptions`

Topics

“Train MBPO Agent to Balance Cart-Pole System”
“Model-Based Policy Optimization (MBPO) Agents”

rlDDPGAgent

Deep deterministic policy gradient (DDPG) reinforcement learning agent

Description

The deep deterministic policy gradient (DDPG) algorithm is an actor-critic, model-free, online, off-policy reinforcement learning method which computes an optimal policy that maximizes the long-term reward. The action space can only be continuous.

For more information, see “Deep Deterministic Policy Gradient (DDPG) Agents”. For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlDDPGAgent(observationInfo,actionInfo)
agent = rlDDPGAgent(observationInfo,actionInfo,initOpts)
```

```
agent = rlDDPGAgent(actor,critic,agentOptions)
```

```
agent = rlDDPGAgent( ___ ,agentOptions)
```

Description

Create Agent from Observation and Action Specifications

`agent = rlDDPGAgent(observationInfo,actionInfo)` creates a deep deterministic policy gradient agent for an environment with the given observation and action specifications, using default initialization options. The actor and critic in the agent use default deep neural networks built from the observation specification `observationInfo` and the action specification `actionInfo`. The `ObservationInfo` and `ActionInfo` properties of `agent` are set to the `observationInfo` and `actionInfo` input arguments, respectively.

`agent = rlDDPGAgent(observationInfo,actionInfo,initOpts)` creates a deep deterministic policy gradient agent for an environment with the given observation and action specifications. The agent uses default networks configured using options specified in the `initOpts` object. For more information on the initialization options, see `rlAgentInitializationOptions`.

Create Agent from Actor and Critic

`agent = rlDDPGAgent(actor,critic,agentOptions)` creates a DDPG agent with the specified actor and critic, using default DDPG agent options.

Specify Agent Options

`agent = rlDDPGAgent( ___ ,agentOptions)` creates a DDPG agent and sets the `AgentOptions` property to the `agentOptions` input argument. Use this syntax after any of the input arguments in the previous syntaxes.

Input Arguments

`initOpts` — Agent initialization options
`rlAgentInitializationOptions` object

Agent initialization options, specified as an `rlAgentInitializationOptions` object.

`actor` — Actor
`rlContinuousDeterministicActor` object

Actor, specified as an `rlContinuousDeterministicActor`. For more information on creating actors, see “Create Policies and Value Functions”.

`critic` — Critic
`rlQValueFunction` object

Critic, specified as an `rlQValueFunction` object. For more information on creating critics, see “Create Policies and Value Functions”.

Properties

`ObservationInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

If you create the agent by specifying an actor and critic, the value of `ObservationInfo` matches the value specified in the actor and critic objects.

You can extract `observationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`ActionInfo` — Action specification
`rlNumericSpec` object

Action specifications, specified as an `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

If you create the agent by specifying an actor and critic, the value of `ActionInfo` matches the value specified in the actor and critic objects.

You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlNumericSpec`.

`AgentOptions` — Agent options
`rlDDPGAgentOptions` object

Agent options, specified as an `rlDDPGAgentOptions` object.

If you create a DDPG agent with default actor and critic that use recurrent neural networks, the default value of `AgentOptions.SequenceLength` is `32`.

`ExperienceBuffer` — Experience buffer
`rlReplayMemory` object

Experience buffer, specified as an `rlReplayMemory` object. During training the agent stores each of its experiences (*S*,*A*,*R*,*S*',*D*) in a buffer. Here:

- *S* is the current observation of the environment.
- *A* is the action taken by the agent.
- *R* is the reward for taking action *A*.
- *S*' is the next observation after taking action *A*.
- *D* is the is-done signal after taking action *A*.

`UseExplorationPolicy` — Option to use exploration policy
`false` (default) | `true`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- `true` — Use the base agent exploration policy when selecting actions.
- `false` — Use the base agent greedy policy when selecting actions.

`SampleTime` — Sample time of agent
positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations. The value of `SampleTime` matches the value specified in `AgentOptions`.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent
setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent
setCritic	Set critic of reinforcement learning agent
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create DDPG Agent from Observation and Action Specifications

Create an environment with a continuous action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force, applied to the mass, ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a policy gradient agent from the environment observation and action specifications.

```
agent = rlDDPGAgent(obsInfo,actInfo)
```

```
agent = 
  rlDDPGAgent with properties:

        ExperienceBuffer: [1x1 rl.replay.rlReplayMemory]
            AgentOptions: [1x1 rl.option.rlDDPGAgentOptions]
    UseExplorationPolicy: 0
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlNumericSpec]
              SampleTime: 1
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension)})
```

```
ans = 1x1 cell array
    {[0.0182]}
```

You can now test and train the agent within the environment. You can also use `getActor` and `getCritic` to extract the actor and critic, respectively, and `getModel` to extract the approximator model (by default a deep neural network) from the actor or critic.

Create DDPG Agent Using Initialization Options

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation”. This environment has two observations: a

50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar representing a torque ranging continuously from -2 to 2 Nm.

```
env = rlPredefinedEnv("SimplePendulumWithImage-Continuous");
```

Obtain observation and action specifications

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create an agent initialization option object, specifying that each hidden fully connected layer in the network must have `128` neurons (instead of the default number, `256`).

```
initOpts = rlAgentInitializationOptions(NumHiddenUnit=128);
```

The agent creation function initializes the actor and critic networks randomly. You can ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a DDPG agent from the environment observation and action specifications.

```
agent = rlDDPGAgent(obsInfo,actInfo,initOpts);
```

Extract the deep neural networks from both the agent actor and critic.

```
actorNet = getModel(getActor(agent));
criticNet = getModel(getCritic(agent));
```

To verify that each hidden fully connected layer has 128 neurons, you can display the layers on the MATLAB® command window,

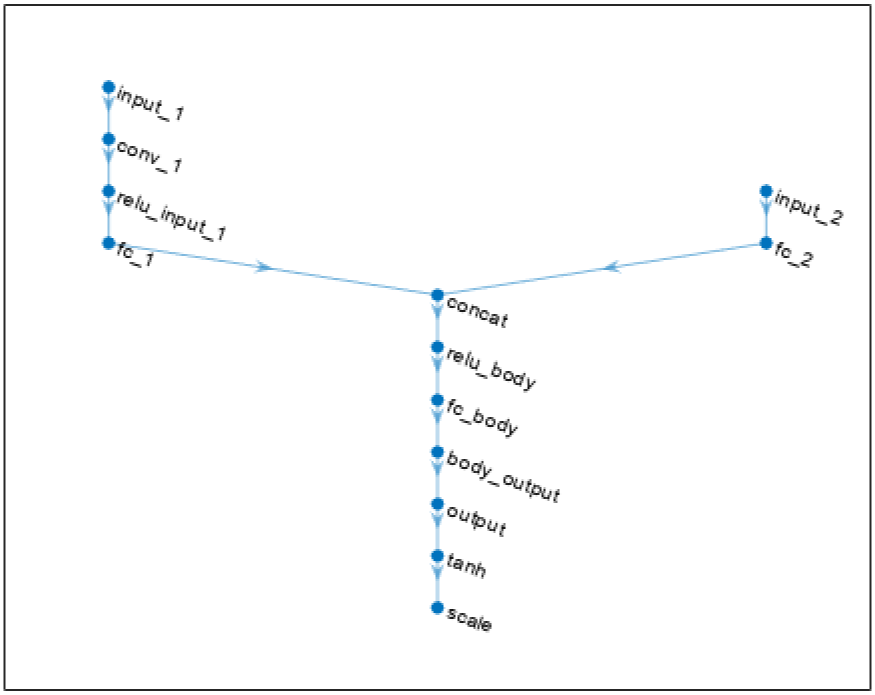
```
criticNet.Layers
```

or visualize the structure interactively using `analyzeNetwork`.

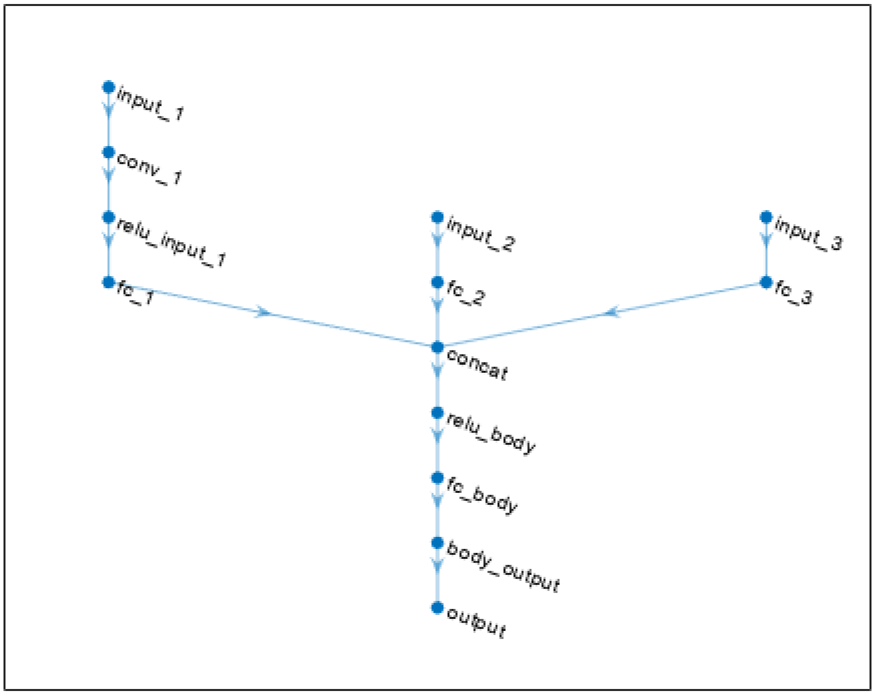
```
analyzeNetwork(criticNet)
```

Plot actor and critic networks

```
plot(layerGraph(actorNet))
```



```
plot(layerGraph(criticNet))
```



To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[-0.0364]}
```

You can now test and train the agent within the environment.

Create DDPG Agent from Actor and Critic

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain the environment observation and action specification objects.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The actor and critic networks are initialized randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

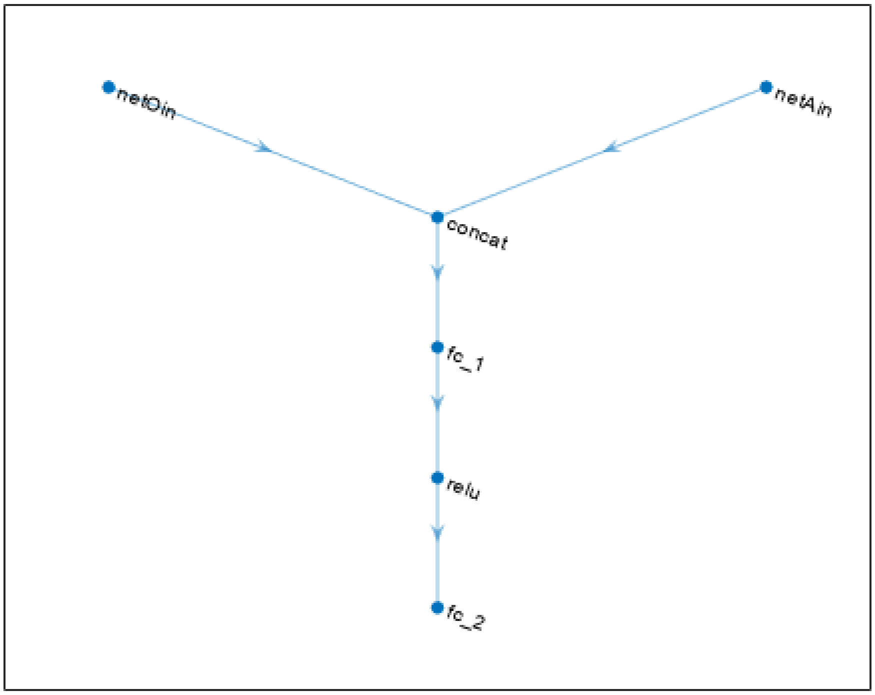
DDPG agents use a parametrized Q-value function approximator to estimate the value of the policy. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value).

Note that `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` return the number of dimensions of the observation and action spaces, respectively, regardless of whether they are arranged as row vectors, column vectors, or matrices.

Define each network path as an array of layer objects and assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% Define observation and action paths
obsPath = featureInputLayer(prod(obsInfo.Dimension),Name="netOin");
actPath = featureInputLayer(prod(actInfo.Dimension),Name="netAin");

% Define common path: concatenate along first dimension
commonPath = [
    concatenationLayer(1,2,Name="concat")
    fullyConnectedLayer(50)
    reluLayer
    fullyConnectedLayer(1)
    ];

% Add paths to layerGraph network
criticNet = layerGraph(obsPath);
criticNet = addLayers(criticNet, actPath);
criticNet = addLayers(criticNet, commonPath);

% Connect paths
criticNet = connectLayers(criticNet,"netOin","concat/in1");
criticNet = connectLayers(criticNet,"netAin","concat/in2");

% Plot the network
plot(criticNet)
```



```
% Convert to dlnetwork object
criticNet = dlnetwork(criticNet);

% Display the number of weights
summary(criticNet)

   Initialized: true

   Number of learnables: 251

   Inputs:
      1   'netOin'   2 features
      2   'netAin'   1 features
```

Create the critic approximator object using `criticNet`, the environment observation and action specifications, and the names of the network input layers to be connected with the environment observation and action channels. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(criticNet,obsInfo,actInfo,...
    ObservationInputNames="netOin", ...
    ActionInputNames="netAin");
```

Check the critic with random observation and action inputs.

```
getValue(critic,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.4260
```

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is implemented by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects.

```
% Create a network to be used as underlying actor approximator
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(16)
    tanhLayer
    fullyConnectedLayer(16)
    tanhLayer
    fullyConnectedLayer(prod(actInfo.Dimension))
    ];

% Convert to dlnetwork object
actorNet = dlnetwork(actorNet);

% Display the number of weights
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 337

   Inputs:
      1   'input'   2 features
```

Create the actor using `actorNet` and the observation and action specifications. For more information on continuous deterministic actors, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNet,obsInfo,actInfo);
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-0.5493]}
```

Create the DDPG agent using the actor and critic.

```
agent = rlDDPGAgent(actor,critic)
```

```
agent = 
  rlDDPGAgent with properties:

        ExperienceBuffer: [1x1 rl.replay.rlReplayMemory]
            AgentOptions: [1x1 rl.option.rlDDPGAgentOptions]
    UseExplorationPolicy: 0
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlNumericSpec]
```

```
             SampleTime: 1
```

Specify agent options, including training options for the actor and critic.

```
agent.AgentOptions.SampleTime=env.Ts;
agent.AgentOptions.TargetSmoothFactor=1e-3;
agent.AgentOptions.ExperienceBufferLength=1e6;
agent.AgentOptions.DiscountFactor=0.99;
agent.AgentOptions.MiniBatchSize=32;

agent.AgentOptions.CriticOptimizerOptions.LearnRate=5e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold=1;

agent.AgentOptions.ActorOptimizerOptions.LearnRate=1e-4;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold=1;
```

Check the agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-0.5947]}
```

You can now train the agent within the environment.

Create DDPG Agent with Recurrent Neural Networks

For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Get the observation and action specification objects.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

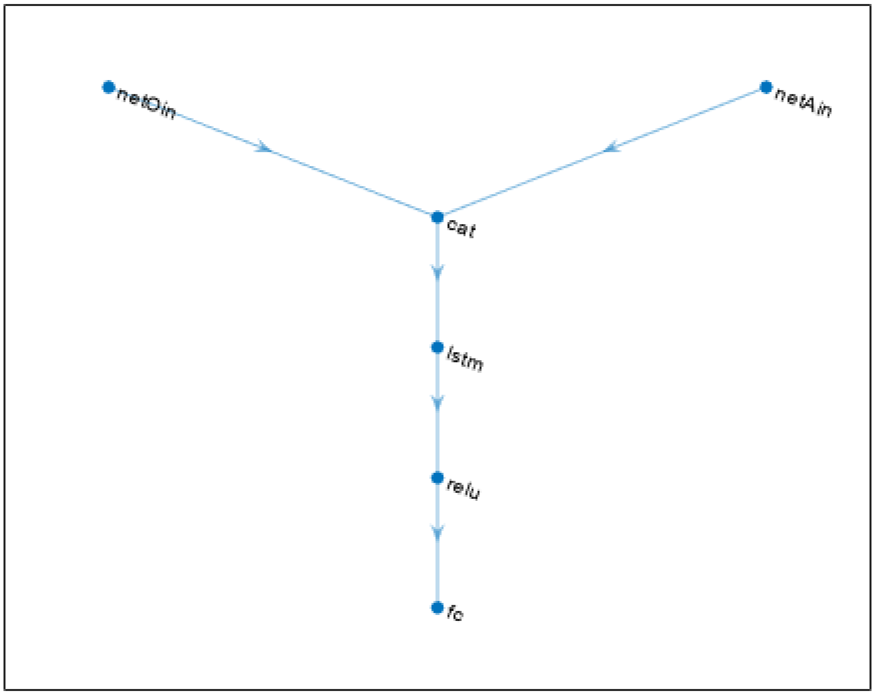
DDPG agents use a parametrized Q-value function approximator to estimate the value of the policy.

To model the parametrized Q-value function within the critic, use a recurrent neural network, which must have two input layers and one output layer (returning the scalar value).

Define each network path as an array of layer objects. To create a recurrent neural network, use `sequenceInputLayer` as the input layer and include an `lstmLayer` as one of the other network layers.

```
% Define observation and action paths
obsPath = sequenceInputLayer(prod(obsInfo.Dimension),Name="netOin");
actPath = sequenceInputLayer(prod(actInfo.Dimension),Name="netAin");

% Define common path: concatenate along first dimension
```

```
commonPath = [
    concatenationLayer(1,2,Name="cat")
    lstmLayer(50)
    reluLayer
    fullyConnectedLayer(1)
    ];

% Add paths to layerGraph network
criticNet = layerGraph(obsPath);
criticNet = addLayers(criticNet, actPath);
criticNet = addLayers(criticNet, commonPath);

% Connect paths
criticNet = connectLayers(criticNet,"netOin","cat/in1");
criticNet = connectLayers(criticNet,"netAin","cat/in2");

% Plot the network
plot(criticNet)
```



```
% Convert to dlnetwork object
criticNet = dlnetwork(criticNet);

% Display the number of weights
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 10.8k
   Inputs:
      1   'netOin'   Sequence input with 2 dimensions
      2   'netAin'   Sequence input with 1 dimensions
```

Create the critic approximator object using `criticNet`, the environment observation and action specifications, and the names of the network input layers to be connected with the environment observation and action channels. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(criticNet,obsInfo,actInfo,...
    ObservationInputNames="netOin",ActionInputNames="netAin");
```

Check the critic with random observation and action inputs.

```
getValue(critic,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.0074
```

DDPG agents use a continuous deterministic actor to approximate the policy. Since the critic has a recurrent network, you must use a recurrent network for the actor too.

Define the network as an array of layer objects.

```
actorNet = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    lstmLayer(10)
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension))
    ];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 531

   Inputs:
      1   'sequenceinput'   Sequence input with 2 dimensions
```

Create the actor using `actorNet` and the observation and action specifications. For more information on continuous deterministic actors, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNet,obsInfo,actInfo);
```

Check the actor with random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.0246]}
```

Specify some training options for the critic.

```
criticOpts = rlOptimizerOptions( ...
    LearnRate=5e-3,GradientThreshold=1);
```

Specify some training options for the actor.

```
actorOpts = rlOptimizerOptions( ...
    LearnRate=1e-4,GradientThreshold=1);
```

Specify agent options. To use a DDPG agent with recurrent neural networks, you must specify a `SequenceLength` greater than 1.

```
agentOpts = rlDDPGAgentOptions(...
    SampleTime=env.Ts,...
    TargetSmoothFactor=1e-3,...
    ExperienceBufferLength=1e6,...
    DiscountFactor=0.99,...
    SequenceLength=20,...
    MiniBatchSize=32, ...
    CriticOptimizerOptions=criticOpts, ...
    ActorOptimizerOptions=actorOpts);
```

Create the DDPG agent using the actor and critic.

```
agent = rlDDPGAgent(actor,critic,agentOpts)
```

```
agent = 
  rlDDPGAgent with properties:

        ExperienceBuffer: [1x1 rl.replay.rlReplayMemory]
            AgentOptions: [1x1 rl.option.rlDDPGAgentOptions]
    UseExplorationPolicy: 0
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlNumericSpec]
              SampleTime: 0.1000
```

To check your agent return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.0158]}
```

To evaluate the agent using sequential observations, use the sequence length (time) dimension. For example, obtain actions for a sequence of 9 observations.

```
[action,state] = getAction(agent, ...
    {rand([obsInfo.Dimension 1 9])});
```

Display the action corresponding to the seventh element of the observation.

```
action = action{1};
action(1,1,1,7)
```

```
ans = 0.0780
```

You can now test and train the agent within the environment.

Version History

Introduced in R2019a

See Also

Apps

Reinforcement Learning Designer | **Deep Network Designer**

Functions

`getAction` | `getActor` | `getCritic` | `generatePolicyFunction` | `generatePolicyBlock` | `getActionInfo` | `getObservationInfo`

Objects

`rlDDPGAgentOptions` | `rlAgentInitializationOptions` | `rlQValueFunction` | `rlContinuousDeterministicActor` | `rlTD3Agent` | `rlSACAgent`

Blocks

RL Agent | Policy

Topics

"Train DDPG Agent to Control Double Integrator System"
"Train Reinforcement Learning Agents"
"Deep Deterministic Policy Gradient (DDPG) Agents"
"Reinforcement Learning Agents"

rlDDPGAgentOptions

Options for DDPG agent

Description

Use an `rlDDPGAgentOptions` object to specify options for deep deterministic policy gradient (DDPG) agents. To create a DDPG agent, use `rlDDPGAgent`.

For more information, see “Deep Deterministic Policy Gradient (DDPG) Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlDDPGAgentOptions
opt = rlDDPGAgentOptions(Name,Value)
```

Description

`opt = rlDDPGAgentOptions` creates an options object for use as an argument when creating a DDPG agent using all default options. You can modify the object properties using dot notation.

`opt = rlDDPGAgentOptions(Name,Value)` sets option properties on page 3-120 using name-value pairs. For example, `rlDDPGAgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`NoiseOptions` — Noise model options
`OrnsteinUhlenbeckActionNoise` object

Noise model options, specified as an `OrnsteinUhlenbeckActionNoise` object. For more information on the noise model, see “Noise Model” on page 3-123.

For an agent with multiple actions, if the actions have different ranges and units, it is likely that each action requires different noise model parameters. If the actions have similar ranges and units, you can set the noise parameters for all actions to the same value.

For example, for an agent with two actions, set the standard deviation of each action to a different value while using the same decay rate for both standard deviations.

```
opt = rlDDPGAgentOptions;
opt.NoiseOptions.StandardDeviation = [0.1 0.2];
opt.NoiseOptions.StandardDeviationDecayRate = 1e-4;
```

ActorOptimizerOptions — Actor optimizer options
rlOptimizerOptions object

Actor optimizer options, specified as an rlOptimizerOptions object. It allows you to specify training parameters of the actor approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see rlOptimizerOptions and rlOptimizer.

CriticOptimizerOptions — Critic optimizer options
rlOptimizerOptions object

Critic optimizer options, specified as an rlOptimizerOptions object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see rlOptimizerOptions and rlOptimizer.

BatchDataRegularizerOptions — Batch data regularizer options
[] (default) | rlBehaviorCloningRegularizerOptions object

Batch data regularizer options, specified as an rlBehaviorCloningRegularizerOptions object. These options are typically used to train the agent offline, from existing data. If you leave this option empty, no regularizer is used.

For more information, see rlBehaviorCloningRegularizerOptions.

Example: opt.BatchDataRegularizerOptions = rlBehaviorCloningRegularizerOptions(BehaviorCloningRegularizerWeight=10)

TargetSmoothFactor — Smoothing factor for target actor and critic updates
1e-3 (default) | positive scalar less than or equal to 1

Smoothing factor for target actor and critic updates, specified as a positive scalar less than or equal to 1. For more information, see “Target Update Methods”.

TargetUpdateFrequency — Number of steps between target actor and critic updates
1 (default) | positive integer

Number of steps between target actor and critic updates, specified as a positive integer. For more information, see “Target Update Methods”.

ResetExperienceBufferBeforeTraining — Option for clearing the experience buffer
false (default) | true

Option for clearing the experience buffer before training, specified as a logical value.

SequenceLength — Maximum batch-training trajectory length when using RNN
1 (default) | positive integer

Maximum batch-training trajectory length when using a recurrent neural network, specified as a positive integer. This value must be greater than 1 when using a recurrent neural network and 1 otherwise.

MiniBatchSize — Size of random experience mini-batch
64 (default) | positive integer

Size of random experience mini-batch, specified as a positive integer. During each training episode, the agent randomly samples experiences from the experience buffer when computing gradients for updating the critic properties. Large mini-batches reduce the variance when computing gradients but increase the computational effort.

`NumStepsToLookAhead` — Number of future rewards used to estimate the value of the policy

`1` (default) | positive integer

Number of future rewards used to estimate the value of the policy, specified as a positive integer. For more information, see [1], Chapter 7.

`ExperienceBufferLength` — Experience buffer size

`10000` (default) | positive integer

Experience buffer size, specified as a positive integer. During training, the agent computes updates using a mini-batch of experiences randomly sampled from the buffer.

`SampleTime` — Sample time of agent

`1` (default) | positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

`DiscountFactor` — Discount factor

`0.99` (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlDDPGAgent	Deep deterministic policy gradient (DDPG) reinforcement learning agent

Examples

Create DDPG Agent Options Object

Create an `rlDDPGAgentOptions` object that specifies the mini-batch size.

```
opt = rlDDPGAgentOptions(MiniBatchSize=48)
```

```
opt = 
  rlDDPGAgentOptions with properties:

                           NoiseOptions: [1x1 rl.option.OrnsteinUhlenbeckActionNoise]
```

```
                ActorOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
               CriticOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
          BatchDataRegularizerOptions: []
                   TargetSmoothFactor: 1.0000e-03
                TargetUpdateFrequency: 1
  ResetExperienceBufferBeforeTraining: 0
                       SequenceLength: 1
                        MiniBatchSize: 48
                  NumStepsToLookAhead: 1
               ExperienceBufferLength: 10000
                           SampleTime: 1
                       DiscountFactor: 0.9900
                           InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

Algorithms

Noise Model

DDPG agents use an Ornstein-Uhlenbeck action noise model for exploration.

Ornstein-Uhlenbeck Action Noise

An `OrnsteinUhlenbeckActionNoise` object has the following numeric value properties.

Property	Description	Default Value
`InitialAction`	Initial value of action	`0`
`Mean`	Noise mean value	`0`
`MeanAttractionConstant`	Constant specifying how quickly the noise model output is attracted to the mean	`0.15`
`StandardDeviationDecayRate`	Decay rate of the standard deviation	`0`
`StandardDeviation`	Initial value of noise standard deviation	`0.3`
`StandardDeviationMin`	Minimum standard deviation	`0`

At each sample time step `k`, the noise value `v(k)` is updated using the following formula, where `Ts` is the agent sample time, and the initial value v(1) is defined by the `InitialAction` parameter.

```
v(k+1) = v(k) + MeanAttractionConstant.*(Mean - v(k)).*Ts
       + StandardDeviation(k).*randn(size(Mean)).*sqrt(Ts)
```

At each sample time step, the standard deviation decays as shown in the following code.

```
decayedStandardDeviation = StandardDeviation(k).*(1 - StandardDeviationDecayRate);
StandardDeviation(k+1) = max(decayedStandardDeviation,StandardDeviationMin);
```

You can calculate how many samples it will take for the standard deviation to be halved using this simple formula.

```
halflife = log(0.5)/log(1-StandardDeviationDecayRate);
```

Note that `StandardDeviation` is conserved between the end of an episode and the start of the next one. Therefore, it keeps on uniformly decreasing over multiple episodes until it reaches `StandardDeviationMin`.

For continuous action signals, it is important to set the noise standard deviation appropriately to encourage exploration. It is common to set `StandardDeviation*sqrt(Ts)` to a value between 1% and 10% of your action range.

If your agent converges on local optima too quickly, promote agent exploration by increasing the amount of noise; that is, by increasing the standard deviation. Also, to increase exploration, you can reduce the `StandardDeviationDecayRate`.

Version History

Introduced in R2019a

R2021a: Property names defining noise probability distribution in the `OrnsteinUhlenbeckActionNoise` object have changed

Behavior changed in R2021a

The properties defining the probability distribution of the Ornstein-Uhlenbeck (OU) noise model have been renamed. DDPG agents use OU noise for exploration.

- The `Variance` property has been renamed `StandardDeviation`.
- The `VarianceDecayRate` property has been renamed `StandardDeviationDecayRate`.
- The `VarianceMin` property has been renamed `StandardDeviationMin`.

The default values of these properties remain the same. When an `OrnsteinUhlenbeckActionNoise` noise object saved from a previous MATLAB release is loaded, the values of `Variance`, `VarianceDecayRate`, and `VarianceMin` are copied in the `StandardDeviation`, `StandardDeviationDecayRate`, and `StandardDeviationMin`, respectively.

The `Variance`, `VarianceDecayRate`, and `VarianceMin` properties still work, but they are not recommended. To define the probability distribution of the OU noise model, use the new property names instead.

Update Code

This table shows how to update your code to use the new property names for `rlDDPGAgentOptions` object `ddpgopt`.

Not Recommended	Recommended
`ddpgopt.NoiseOptions.Variance = 0.5;`	`ddpgopt.NoiseOptions.StandardDeviation = 0.5;`
`ddpgopt.NoiseOptions.VarianceDecayRate = 0.1;`	`ddpgopt.NoiseOptions.StandardDeviationDecayRate = 0.1;`

Not Recommended	Recommended
`ddpgopt.NoiseOptions.VarianceMin = 0;`	`ddpgopt.NoiseOptions.StandardDeviationMin = 0;`

R2020a: Target update method settings for DDPG agents have changed
Behavior changed in R2020a

Target update method settings for DDPG agents have changed. The following changes require updates to your code:

- The `TargetUpdateMethod` option has been removed. Now, DDPG agents determine the target update method based on the `TargetUpdateFrequency` and `TargetSmoothFactor` option values.
- The default value of `TargetUpdateFrequency` has changed from `4` to `1`.

To use one of the following target update methods, set the `TargetUpdateFrequency` and `TargetSmoothFactor` properties as indicated.

Update Method	TargetUpdateFrequency	TargetSmoothFactor
Smoothing	`1`	Less than `1`
Periodic	Greater than `1`	`1`
Periodic smoothing (new method in R2020a)	Greater than `1`	Less than `1`

The default target update configuration, which is a smoothing update with a `TargetSmoothFactor` value of `0.001`, remains the same.

Update Code

This table shows some typical uses of `rlDDPGAgentOptions` and how to update your code to use the new option configuration.

Not Recommended	Recommended
`opt = rlDDPGAgentOptions('TargetUpdateMethod',"smoothing");`	`opt = rlDDPGAgentOptions;`

`opt = rlDDPGAgentOptions('TargetUpdateMethod',"periodic");`	`opt = rlDDPGAgentOptions;`
	`opt.TargetUpdateFrequency = 4;`
	`opt.TargetSmoothFactor = 1;`

`opt = rlDDPGAgentOptions;`
`opt.TargetUpdateMethod = "periodic";`
`opt.TargetUpdateFrequency = 5;`
	`opt = rlDDPGAgentOptions;`
	`opt.TargetUpdateFrequency = 5;`
	`opt.TargetSmoothFactor = 1;`

R2022a: The default value of the `ResetExperienceBufferBeforeTraining` property has changed
Behavior changed in R2022a

The default value of the `ResetExperienceBufferBeforeTraining` has changed from `true` to `false`.

When creating a new DDPG agent, if you want to clear the experience buffer before training, you must specify `ResetExperienceBufferBeforeTraining` as `true`. For example, before training, set the property using dot notation.

```
agent.AgentOptions.ResetExperienceBufferBeforeTraining = true;
```

Alternatively, you can set the property to `true` in an `rlDDPGAgentOptions` object and use this object to create the DDPG agent.

References

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning. Cambridge, Mass: The MIT Press, 2018.

See Also

Objects

`rlDDPGAgent` | `rlAgentInitializationOptions` | `rlBehaviorCloningRegularizerOptions`

Topics

"Train DDPG Agent to Control Double Integrator System"
"Deep Deterministic Policy Gradient (DDPG) Agents"

rlDeterministicActorPolicy

Policy object to generate continuous deterministic actions for custom training loops and application deployment

Description

This object implements a deterministic policy, which returns continuous deterministic actions given an input observation. You can create an `rlDeterministicActorPolicy` object from an `rlContinuousDeterministicActor` or extract it from an `rlDDPGAgent` or `rlTD3Agent`. You can then train the policy object using a custom training loop or deploy it for your application using `generatePolicyBlock` or `generatePolicyFunction`. This policy is always deterministic and does not perform any exploration. For more information on policies and value functions, see “Create Policies and Value Functions”.

Creation

Syntax

```
policy = rlDeterministicActorPolicy(actor)
```

Description

`policy = rlDeterministicActorPolicy(actor)` creates the deterministic actor policy object `policy` from the continuous deterministic actor `actor`. It also sets the `Actor` property of `policy` to the input argument `actor`.

Properties

`Actor` — Continuous deterministic actor
`rlContinuousDeterministicActor` object

Continuous deterministic actor, specified as an `rlContinuousDeterministicActor` object.

`ObservationInfo` — Observation specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array of such objects. These objects define properties such as the dimensions, data types, and names of the observation channels.

`ActionInfo` — Action specifications
`rlNumericSpec` object

Action specifications, specified as an `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

SampleTime — Sample time of policy

positive scalar | -1 (default)

Sample time of the policy, specified as a positive scalar or as `-1` (default). Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the policy is specified executes every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the policy is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience. If `SampleTime` is `-1`, the sample time is treated as being equal to `1`.

Example: `0.2`

Object Functions

generatePolicyBlock	Generate Simulink block that evaluates policy of an agent or policy object
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object
getAction	Obtain action from agent, actor, or policy object given environment observations
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
reset	Reset environment, agent, experience buffer, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object

Examples

Create Deterministic Actor Policy from Continuous Deterministic Actor

Create observation and action specification objects. For this example, define the observation and action spaces as continuous four- and two-dimensional spaces, respectively.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlNumericSpec([2 1]);
```

Alternatively, use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment

Create a continuous deterministic actor. This actor must accept an observation as input and return an action as output.

To approximate the policy function within the actor, use a deep neural network model. Define the network as an array of layer objects, and get the dimension of the observation and action spaces from the environment specification objects.

```
layers = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(actInfo.Dimension(1))
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
model = dlnetwork(layers);
summary(model)
```

```
   Initialized: true

   Number of learnables: 114

   Inputs:
      1   'input'   4 features
```

Create the actor using `model`, and the observation and action specifications.

```
actor = rlContinuousDeterministicActor(model,obsInfo,actInfo)
```

```
actor = 
  rlContinuousDeterministicActor with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

Check the actor with a random observation input.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = 2x1 single column vector

    0.4013
    0.0578
```

Create a policy object from `actor`.

```
policy = rlDeterministicActorPolicy(actor)
```

```
policy = 
  rlDeterministicActorPolicy with properties:

              Actor: [1x1 rl.function.rlContinuousDeterministicActor]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
         SampleTime: -1
```

Check the policy with a random observation input.

```
act = getAction(policy,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = 2×1

    0.4313
   -0.3002
```

You can now train the policy with a custom training loop and then deploy it to your application.

Version History

Introduced in R2022a

See Also

Functions

`getGreedyPolicy` | `getExplorationPolicy` | `generatePolicyBlock` | `generatePolicyFunction` | `getAction` | `getLearnableParameters` | `setLearnableParameters`

Objects

`rlMaxQPolicy` | `rlEpsilonGreedyPolicy` | `rlAdditiveNoisePolicy` | `rlStochasticActorPolicy` | `rlTD3Agent` | `rlDDPGAgent`

Blocks

Policy | RL Agent

Topics

"Model-Based Reinforcement Learning Using Custom Training Loop"
"Train Reinforcement Learning Policy Using Custom Training Loop"
"Create Policies and Value Functions"

rlDeterministicActorRepresentation

(Not recommended) Deterministic actor representation for reinforcement learning agents

Note `rlDeterministicActorRepresentation` is not recommended. Use `rlContinuousDeterministicActor` instead. For more information, see "rlDeterministicActorRepresentation is not recommended".

Description

This object implements a function approximator to be used as a deterministic actor within a reinforcement learning agent with a *continuous* action space. A deterministic actor takes observations as inputs and returns as outputs the action that maximizes the expected cumulative long-term reward, thereby implementing a deterministic policy. After you create an `rlDeterministicActorRepresentation` object, use it to create a suitable agent, such as an `rlDDPGAgent` agent. For more information on creating representations, see "Create Policies and Value Functions".

Creation

Syntax

```
actor = rlDeterministicActorRepresentation(net,observationInfo,
actionInfo,'Observation',obsName,'Action',actName)
actor = rlDeterministicActorRepresentation({basisFcn,W0},observationInfo,
actionInfo)
actor = rlDeterministicActorRepresentation( ___ ,options)
```

Description

`actor = rlDeterministicActorRepresentation(net,observationInfo, actionInfo,'Observation',obsName,'Action',actName)` creates a deterministic actor using the deep neural network `net` as approximator. This syntax sets the ObservationInfo and ActionInfo properties of `actor` to the inputs `observationInfo` and `actionInfo`, containing the specifications for observations and actions, respectively. `actionInfo` must specify a continuous action space, discrete action spaces are not supported. `obsName` must contain the names of the input layers of `net` that are associated with the observation specifications. The action names `actName` must be the names of the output layers of `net` that are associated with the action specifications.

`actor = rlDeterministicActorRepresentation({basisFcn,W0},observationInfo, actionInfo)` creates a deterministic actor using a custom basis function as underlying approximator. The first input argument is a two-elements cell in which the first element contains the handle `basisFcn` to a custom basis function, and the second element contains the initial weight matrix `W0`. This syntax sets the ObservationInfo and ActionInfo properties of `actor` respectively to the inputs `observationInfo` and `actionInfo`.

`actor = rlDeterministicActorRepresentation( ___ ,options)` creates a deterministic actor using the additional options set `options`, which is an `rlRepresentationOptions` object.

This syntax sets the Options property of `actor` to the`options` input argument. You can use this syntax with any of the previous input-argument combinations.

Input Arguments

`net` — Deep neural network

array of `Layer` objects | `layerGraph` object | `DAGNetwork` object | `SeriesNetwork` object | `dlNetwork` object

Deep neural network used as the underlying approximator within the actor, specified as one of the following:

- Array of `Layer` objects
- `layerGraph` object
- `DAGNetwork` object
- `SeriesNetwork` object
- `dlnetwork` object

The network input layers must be in the same order and with the same data type and dimensions as the signals defined in ObservationInfo. Also, the names of these input layers must match the observation names listed in `obsName`.

The network output layer must have the same data type and dimension as the signal defined in ActionInfo. Its name must be the action name specified in `actName`.

`rlDeterministicActorRepresentation` objects support recurrent deep neural networks.

For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

`obsName` — Observation names

string | character vector | cell array of character vectors

Observation names, specified as a cell array of strings or character vectors. The observation names must be the names of the input layers in `net`.

Example: `{'my_obs'}`

`actName` — Action name

string | character vector | single-element cell array containing a character vector

Action name, specified as a single-element cell array that contains a character vector. It must be the name of the output layer of `net`.

Example: `{'my_act'}`

`basisFcn` — Custom basis function

function handle

Custom basis function, specified as a function handle to a user-defined MATLAB function. The user defined function can either be an anonymous function or a function on the MATLAB path. The action to be taken based on the current observation, which is the output of the actor, is the vector `a = W'*B`, where `W` is a weight matrix containing the learnable parameters and `B` is the column vector returned by the custom basis function.

When creating a deterministic actor representation, your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here `obs1` to `obsN` are observations in the same order and with the same data type and dimensions as the signals defined in `observationInfo`

Example: `@(obs1,obs2,obs3) [obs3(2)*obs1(1)^2; abs(obs2(5)+obs3(1))]`

`W0` — Initial value of the basis function weights

column vector

Initial value of the basis function weights, `W`, specified as a matrix having as many rows as the length of the vector returned by the basis function and as many columns as the dimension of the action space.

Properties

`Options` — Representation options

`rlRepresentationOptions` object

Representation options, specified as an `rlRepresentationOptions` object. Available options include the optimizer used for training and the learning rate.

`ObservationInfo` — Observation specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array of such objects. These objects define properties such as the dimensions, data types, and names of the observation signals.

`rlDeterministicActorRepresentation` sets the `ObservationInfo` property of `actor` to the input `observationInfo`.

You can extract `ObservationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually.

`ActionInfo` — Action specifications

`rlNumericSpec` object

Action specifications for a continuous action space, specified as an `rlNumericSpec` object defining properties such as dimensions, data type and name of the action signals. The deterministic actor representation does not support discrete actions.

`rlDeterministicActorRepresentation` sets the `ActionInfo` property of `actor` to the input `observationInfo`.

You can extract `ActionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually.

For custom basis function representations, the action signal must be a scalar, a column vector, or a discrete action.

Object Functions

rlDDPGAgent	Deep deterministic policy gradient (DDPG) reinforcement learning agent
rlTD3Agent	Twin-delayed deep deterministic (TD3) policy gradient reinforcement learning agent
getAction	Obtain action from agent, actor, or policy object given environment observations

Examples

Create Deterministic Actor from Deep Neural Network

Create an observation specification object (or alternatively use getObservationInfo to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create an action specification object (or alternatively use getActionInfo to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that a single action is a column vector containing two doubles.

```
actInfo = rlNumericSpec([2 1]);
```

Create a deep neural network approximator for the actor. The input of the network (here called myobs) must accept a four-element vector (the observation vector just defined by obsInfo), and its output must be the action (here called myact) and be a two-element vector, as defined by actInfo.

```
net = [featureInputLayer(4,'Normalization','none','Name','myobs')
    fullyConnectedLayer(2,'Name','myact')];
```

Create the critic with rlQValueRepresentation, using the network, the observations and action specification objects, as well as the names of the network input and output layers.

```
actor = rlDeterministicActorRepresentation(net,obsInfo,actInfo, ...
    'Observation',{'myobs'},'Action',{'myact'})
```

```
actor = 
  rlDeterministicActorRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlNumericSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your actor, use getAction to return the action from a random observation, using the current network weights.

```
act = getAction(actor,{rand(4,1)}); act{1}
```

```
ans = 2x1 single column vector

   -0.5054
    1.5390
```

You can now use the actor to create a suitable agent (such as an rlACAgent, rlPGAgent, or rlDDPGAgent agent).

Create Deterministic Actor from Custom Basis Function

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing 3 doubles.

```
obsInfo = rlNumericSpec([3 1]);
```

The deterministic actor does not support discrete action spaces. Therefore, create a *continuous action space* specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that a single action is a column vector containing 2 doubles.

```
actInfo = rlNumericSpec([2 1]);
```

Create a custom basis function. Each element is a function of the observations defined by `obsInfo`.

```
myBasisFcn = @(myobs) [myobs(2)^2; myobs(1); 2*myobs(2)+myobs(1); -myobs(3)]
```

```
myBasisFcn = function_handle with value:
    @(myobs)[myobs(2)^2;myobs(1);2*myobs(2)+myobs(1);-myobs(3)]
```

The output of the actor is the vector `W'*myBasisFcn(myobs)`, which is the action taken as a result of the given observation. The weight matrix `W` contains the learnable parameters and must have as many rows as the length of the basis function output and as many columns as the dimension of the action space.

Define an initial parameter matrix.

```
W0 = rand(4,2);
```

Create the actor. The first argument is a two-element cell containing both the handle to the custom function and the initial weight matrix. The second and third arguments are, respectively, the observation and action specification objects.

```
actor = rlDeterministicActorRepresentation({myBasisFcn,W0},obsInfo,actInfo)
```

```
actor = 
  rlDeterministicActorRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlNumericSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your actor, use the `getAction` function to return the action from a given observation, using the current parameter matrix.

```
a = getAction(actor,{[1 2 3]'});
a{1}
```

```
ans = 
  2x1 dlarray
```

```
    2.0595
    2.3788
```

You can now use the actor (along with an critic) to create a suitable continuous action space agent.

Create Deterministic Actor from Recurrent Neural Network

Create observation and action information. You can also obtain these specifications from an environment.

```
obsinfo = rlNumericSpec([4 1]);
actinfo = rlNumericSpec([2 1]);
numObs = obsinfo.Dimension(1);
numAct = actinfo.Dimension(1);
```

Create a recurrent deep neural network for the actor. To create a recurrent neural network, use a `sequenceInputLayer` as the input layer and include at least one `lstmLayer`.

```
net = [sequenceInputLayer(numObs,'Normalization','none','Name','state')
            fullyConnectedLayer(10,'Name','fc1')
            reluLayer('Name','relu1')
            lstmLayer(8,'OutputMode','sequence','Name','ActorLSTM')
            fullyConnectedLayer(20,'Name','CriticStateFC2')
            fullyConnectedLayer(numAct,'Name','action')
            tanhLayer('Name','tanh1')];
```

Create a deterministic actor representation for the network.

```
actorOptions = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1);
actor = rlDeterministicActorRepresentation(net,obsinfo,actinfo,...
    'Observation',{'state'},'Action',{'tanh1'});
```

Version History

Introduced in R2020a

R2022a: `rlDeterministicActorRepresentation` is not recommended

Not recommended starting in R2022a

`rlDeterministicActorRepresentation` is not recommended. Use `rlContinuousDeterministicActor` instead.

The following table shows some typical uses of `rlDeterministicActorRepresentation`, and how to update your code with `rlContinuousDeterministicActor` instead. The first table entry uses a neural network, the second one uses a basis function.

rlDeterministicActorRepresentation: Not Recommended	**rlContinuousDeterministicActor: Recommended**
`myActor = rlDeterministicActorRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action',actNames)` with `actInfo` defining a continuous action space and `net` having observations as inputs and a single output layer with as many elements as the number of dimensions of the continuous action space.	`myActor = rlContinuousDeterministicActor(net,obsInfo,actInfo,'ObservationInputNames',obsNames)`. Use this syntax to create a deterministic actor object with a continuous action space.
`rep = rlDeterministicActorRepresentation({basisFcn,W0},obsInfo,actInfo)`, where the basis function has observations as inputs and actions as outputs, `W0` is a matrix with as many columns as the number of possible actions, and `actInfo` defines a continuous action space.	`rep = rlContinuousDeterministicActor({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a deterministic actor object with a continuous action space.

See Also

Functions
`getActionInfo` | `getObservationInfo`

Objects
`rlContinuousDeterministicActor` | `rlRepresentationOptions`

Topics
“Create Policies and Value Functions”
“Reinforcement Learning Agents”

rlDiscreteCategoricalActor

Stochastic categorical actor with a discrete action space for reinforcement learning agents

Description

This object implements a function approximator to be used as a stochastic actor within a reinforcement learning agent with a discrete action space. A discrete categorical actor takes an environment observation as input and returns as output a random action sampled from a categorical (also known as Multinoulli) probability distribution, thereby implementing a parametrized stochastic policy. After you create an `rlDiscreteCategoricalActor` object, use it to create a suitable agent, such as `rlACAgent` or `rlPGAgent`. For more information on creating representations, see “Create Policies and Value Functions”.

Creation

Syntax

```
actor = rlDiscreteCategoricalActor(net,observationInfo,actionInfo)
actor = rlDiscreteCategoricalActor(net,observationInfo,
actionInfo,ObservationInputNames=netObsNames)
```

```
actor = rlDiscreteCategoricalActor({basisFcn,W0},observationInfo,actionInfo)
```

```
actor = rlDiscreteCategoricalActor( ___ ,UseDevice=useDevice)
```

Description

`actor = rlDiscreteCategoricalActor(net,observationInfo,actionInfo)` creates a stochastic actor with a discrete action space, using the deep neural network `net` as underlying approximation model. For this actor, `actionInfo` must specify a discrete action space. The network input layers are automatically associated with the environment observation channels according to the dimension specifications in `observationInfo`. The network must have a single output layer with as many elements as the number of possible discrete actions, as specified in `actionInfo`. This function sets the `ObservationInfo` and `ActionInfo` properties of `actor` to the inputs `observationInfo` and `actionInfo`, respectively.

`actor = rlDiscreteCategoricalActor(net,observationInfo, actionInfo,ObservationInputNames=netObsNames)` specifies the names of the network input layers to be associated with the environment observation channels. The function assigns, in sequential order, each environment observation channel specified in `observationInfo` to the layer specified by the corresponding name in the string array `netObsNames`. Therefore, the network input layers, ordered as the names in `netObsNames`, must have the same data type and dimensions as the observation channels, as ordered in `observationInfo`.

`actor = rlDiscreteCategoricalActor({basisFcn,W0},observationInfo,actionInfo)` creates a discrete space stochastic actor using a custom basis function as underlying approximation model. The first input argument is a two-element cell array whose first element is the handle basisFcn to a custom basis function and whose second element is the initial weight matrix W0. This function sets the ObservationInfo and ActionInfo properties of actor to the inputs observationInfo and actionInfo, respectively.

actor = rlDiscreteCategoricalActor(___ ,UseDevice=useDevice) specifies the device used to perform computational operations on the actor object, and sets the UseDevice property of actor to the useDevice input argument. You can use this syntax with any of the previous input-argument combinations.

Input Arguments

net — Deep neural network

array of Layer objects | layerGraph object | DAGNetwork object | SeriesNetwork object | dlNetwork object (preferred)

Deep neural network used as the underlying approximator within the actor, specified as one of the following:

- Array of Layer objects
- layerGraph object
- DAGNetwork object
- SeriesNetwork object
- dlnetwork object

Note Among the different network representation options, dlnetwork is preferred, since it has built-in validation checks and supports automatic differentiation. If you pass another network object as an input argument, it is internally converted to a dlnetwork object. However, best practice is to convert other representations to dlnetwork explicitly *before* using it to create a critic or an actor for a reinforcement learning agent. You can do so using dlnet=dlnetwork(net), where net is any Deep Learning Toolbox neural network object. The resulting dlnet is the dlnetwork object that you use for your critic or actor. This practice allows a greater level of insight and control for cases in which the conversion is not straightforward and might require additional specifications.

The network must have as many input layers as the number of environment observation channels (with each input layer receiving input from an observation channel), and a single output layer with as many elements as the number of possible discrete actions. Since the actor must return the probability of executing each possible action, the software automatically adds a softmaxLayer as a final output layer if you do not specify it explicitly. When computing the action, the actor then randomly samples the distribution to return an action.

rlDiscreteCategoricalActor objects support recurrent deep neural networks. For an example, see “Create Discrete Categorical Actor from Deep Recurrent Neural Network” on page 3-147.

The learnable parameters of the actor are the weights of the deep neural network. For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

netObsNames — Network input layers names corresponding to the environment observation channels

string array | cell array of character vectors

Network input layers names corresponding to the environment observation channels, specified as a string array or a cell array of character vectors. When you use this argument after 'ObservationInputNames', the function assigns, in sequential order, each environment observation channel specified in observationInfo to each network input layer specified by the corresponding name in the string array netObsNames. Therefore, the network input layers, ordered as the names in netObsNames, must have the same data type and dimensions as the observation specifications, as ordered in observationInfo.

Example: {"NetInput1_airspeed","NetInput2_altitude"}

basisFcn — Custom basis function

function handle

Custom basis function, specified as a function handle to a user-defined MATLAB function. The user defined function can either be an anonymous function or a function on the MATLAB path. The number of the action to be taken based on the current observation, which is the output of the actor, is randomly sampled from a categorical distribution with probabilities p = softmax(W'*B), where W is a weight matrix containing the learnable parameters and B is the column vector returned by the custom basis function. Each element of p represents the probability of executing the corresponding action from the observed state.

Your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here, obs1 to obsN are inputs in the same order and with the same data type and dimensions as the environment observation channels defined in observationInfo.

Example: @(obs1,obs2,obs3) [obs3(2)*obs1(1)^2; abs(obs2(5)+obs3(1))]

W0 — Initial value of the basis function weights

matrix

Initial value of the basis function weights W, specified as a matrix having as many rows as the length of the vector returned by the basis function and as many columns as the dimension of the action space.

Properties

ObservationInfo — Observation specifications

rlFiniteSetSpec object | rlNumericSpec object | array

Observation specifications, specified as an rlFiniteSetSpec or rlNumericSpec object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

rlDiscreteCategoricalActor sets the ObservationInfo property of actor to the input observationInfo.

You can extract ObservationInfo from an existing environment or agent using getObservationInfo. You can also construct the specifications manually.

ActionInfo — Action specifications

rlFiniteSetSpec object

Action specifications, specified as an `rlFiniteSetSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

`rlDiscreteCategoricalActor` sets the `ActionInfo` property of `critic` to the input `actionInfo`.

You can extract `ActionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specifications manually.

UseDevice — Computation device used for training and simulation
`"cpu"` (default) | `"gpu"`

Computation device used to perform operations such as gradient computation, parameter update and prediction during training and simulation, specified as either `"cpu"` or `"gpu"`.

The `"gpu"` option requires both Parallel Computing Toolbox software and a CUDA enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use `gpuDevice` (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating an agent on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

To speed up training by using parallel processing over multiple cores, you do not need to use this argument. Instead, when training your agent, use an `rlTrainingOptions` object in which the `UseParallel` option is set to `true`. For more information about training using multicore processors and GPUs for training, see “Train Agents Using Parallel Computing and GPUs”.

Example: `"gpu"`

Object Functions

rlACAgent	Actor-critic (AC) reinforcement learning agent
rlPGAgent	Policy gradient (PG) reinforcement learning agent
rlPPOAgent	Proximal policy optimization (PPO) reinforcement learning agent
getAction	Obtain action from agent, actor, or policy object given environment observations
evaluate	Evaluate function approximator object given observation (or observation-action) input data
gradient	Evaluate gradient of function approximator object given observation and action input data
accelerate	Option to accelerate computation of gradient for approximator object based on neural network
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object

setModel	Set function approximation model for actor or critic
getModel	Get function approximator model from actor or critic

Examples

Create Discrete Categorical Actor from Deep Neural Network

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as consisting of three actions, labeled -10, 0, and 10.

```
actInfo = rlFiniteSetSpec([-10 0 10]);
```

A discrete categorical actor implements a parametrized stochastic policy for a discrete action space. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the probability distribution within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer.

The output layer must return a vector of probabilities of taking each possible action, as specified by `actInfo`. Therefore, each element of the output vector must be between 0 and 1. Using softmax as the output layer enforces this requirement (the software automatically adds a `softmaxLayer` as a final output layer if you do not specify it explicitly).

Note that `prod(obsInfo.Dimension)` returns the total number of dimensions of the observation space regardless of whether the observation space is arranged as a column vector, row vector, or matrix, while `numel(actInfo.Dimension)` returns the number of elements of the discrete action space.

Define the network as an array of layer objects.

```
net = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert the network to a `dlnetwork` object and display the number of learnable parameters.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true
   Number of learnables: 403

   Inputs:
      1   'input'   4 features
```

Create the actor with `rlDiscreteCategoricalActor`, using the network, the observations and action specification objects. When the network has multiple input layers, they are automatically associated with the environment observation channels according to the dimension specifications in `obsInfo`.

```
actor = rlDiscreteCategoricalActor(net,obsInfo,actInfo);
```

To check your actor, use `getAction` to return an action from a random observation vector, given the current network weights.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act
```

```
act = 1x1 cell array
    {[-10]}
```

To return the probability distribution of the actions, given an observation, use `evaluate`.

```
prb = evaluate(actor,{rand(obsInfo.Dimension)});
prb{1}
```

```
ans = 3x1 single column vector

    0.3736
    0.1875
    0.4389
```

You can now use the actor (along with a critic) to create an agent for the environment described by the given observation specification object. Examples of agents that can work with a continuous observation space, a discrete action space, and use a discrete categorical actor, are `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, and `rlTRPOAgent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Discrete Categorical Actor from Deep Neural Network Specifying Input Layer Name

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as consisting of three actions, labeled -10, 0, and 10.

```
actInfo = rlFiniteSetSpec([-10 0 10]);
```

A discrete categorical actor implements a parametrized stochastic policy for a discrete action space. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the probability distribution within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by `actInfo`. Therefore, each element of the output vector must be between 0 and 1. Using softmax as the output layer enforces this requirement (the software automatically adds a `softmaxLayer` as a final output layer if you do not specify it explicitly).

Note that `prod(obsInfo.Dimension)` returns the total number of dimensions of the observation space regardless of whether the observation space is arranged as a column vector, row vector, or matrix, while `numel(actInfo.Dimension)` returns the number of elements of the discrete action space.

Define the network as an array of layer objects. Specify a name for the input layer, so you can later explicitly associate it with the observation channel.

```
net = [
    featureInputLayer( ...
        prod(obsInfo.Dimension), ...
        Name="netObsIn")
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    softmaxLayer(Name="actionProb")
    ];
```

Convert the network to a `dlnetwork` object and display the number of learnable parameters (weights).

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 259

   Inputs:
      1   'netObsIn'   4 features
```

Create the actor with `rlDiscreteCategoricalActor`, using the network, the observations and action specification objects, and the name of the network input layer.

```
actor = rlDiscreteCategoricalActor(net, ...
    obsInfo,actInfo,...
    Observation="netObsIn");
```

To validate your actor, use `getAction` to return an action from a random observation, given the current network weights.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = -10
```

To return the probability distribution of the possible actions as a function of a random observation, and given the current network weights, use `evaluate`.

```
prb = evaluate(actor,{rand(obsInfo.Dimension)})
```

```
prb = 1x1 cell array
    {3x1 single}
```

```
prb{1}
```

```
ans = 3x1 single column vector

    0.3038
    0.2658
    0.4304
```

You can now use the actor (along with a critic) to create an agent for the environment described by the given observation specification object. Examples of agents that can work with a continuous observation space, a discrete action space, and use a discrete categorical actor, are `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, and `rlTRPOAgent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Discrete Categorical Actor from Custom Basis Function

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as consisting of two channels, the first carrying a two-dimensional vector in a continuous space, the second carrying a two dimensional vector that can assume only three values, -[1 2], [0 1], and [1 3]. Therefore a single observation consists of two two-dimensional vectors, one continuous, the other discrete.

```
obsInfo = [rlNumericSpec([2 1]) rlFiniteSetSpec({-[1 2],[0 1],[1 3]})];
```

Create a *discrete action space* specification object (or alternatively use `getActionInfo` to extract the specification object from an environment with a discrete action space). For this example, define the action space as a finite set consisting of three possible actions, labeled 7, 5, and 3.

```
actInfo = rlFiniteSetSpec([7 5 3]);
```

A discrete categorical actor implements a parametrized stochastic policy for a discrete action space. To model the parametrized probability distribution within the actor, use a custom basis function with two inputs (which receive the content of the environment observation channels, as specified by `obsInfo`).

Create a function that returns a vector of four elements, depending on a given observation.

```
myBasisFcn = @(obsC,obsD) [obsC(1)^2-obsD(2)^2;
                           obsC(2)^2-obsD(1)^2;
                           exp(obsC(2))+abs(obsD(1));
                           exp(obsC(1))+abs(obsD(2))];
```

The actor samples the action randomly, according to the probability distribution softmax(W'*myBasisFcn(obsC,obsD)). Here, W is a weight matrix, containing the learnable parameters, which must have as many rows as the length of the basis function output (for this example, four), and as many columns as the number of possible actions (for this example, three).

Define an initial parameter matrix.

```
W0 = rand(4,3);
```

Create the actor. The first argument is a two-element cell containing both the handle to the custom function and the initial parameter matrix. The second and third arguments are, respectively, the observation and action specification objects.

```
actor = rlDiscreteCategoricalActor({myBasisFcn,W0},obsInfo,actInfo);
```

To check your actor use the getAction function to return one of the three possible actions, depending on a given random observation and on the current parameter matrix.

```
getAction(actor,{rand(2,1),[1 1]})
```

```
ans = 1x1 cell array
    {[3]}
```

Note that the discrete set constraint is not enforced.

```
getAction(actor,{rand(2,1),[0.5 -0.7]})
```

```
ans = 1x1 cell array
    {[3]}
```

To return the probability of each action as a function of a random observation (and given the current weights), use evaluate.

```
prb = evaluate(actor, ...
    {rand(obsInfo(1).Dimension), ...
     rand(obsInfo(2).Dimension)})
```

```
prb = 1x1 cell array
    {3x1 double}
```

```
prb{1}
```

```
ans = 3×1

    0.3512
    0.1519
    0.4969
```

You can now use the actor (along with a critic) to create an agent for the environment described by the given observation specification object. Examples of agents that can work with a mixed observation space, a discrete action space, and use a discrete categorical actor, are rlACAgent, rlPGAgent, and rlPPOAgent. rlTRPOAgent does not support actors or critics that use custom basis functions.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Discrete Categorical Actor from Deep Recurrent Neural Network

This example shows you how to create a stochastic actor with a discrete action space using a recurrent neural network.

For this example, use the same environment used in “Train PG Agent to Balance Cart-Pole System”. Load the environment and obtain the observation and action specifications.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "CartPole States"
    Description: "x, dx, theta, dtheta"
      Dimension: [4 1]
       DataType: "double"
```

```
actInfo = getActionInfo(env)
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [-10 10]
           Name: "CartPole Action"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

A discrete categorical actor implements a parametrized stochastic policy for a discrete action space. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the probability distribution within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer.

The output layer must return a vector of probabilities of taking each possible action, as specified by `actInfo`. Therefore, each element of the output vector must be between 0 and 1. Using softmax as the output layer enforces this requirement (the software automatically adds a `softmaxLayer` as a final output layer if you do not specify it explicitly).

Note that `prod(obsInfo.Dimension)` returns the total number of dimensions of the observation space regardless of whether the observation space is arranged as a column vector, row vector, or matrix, while `numel(actInfo.Dimension)` returns the number of elements of the discrete action space.

Define the network as an array of layer objects. To create a recurrent network, use a `sequenceInputLayer` as the input layer and include at least one `lstmLayer`.

```
net = [
    sequenceInputLayer( ...
        prod(obsInfo.Dimension), ...
        Name="netObsIn")
    fullyConnectedLayer(8)
    reluLayer
    lstmLayer(8)
    fullyConnectedLayer( ...
    numel(actInfo.Elements))
    ];
```

Convert the network to a `dlnetwork` object and display the number of learnable parameters (weights).

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 602

   Inputs:
      1   'netObsIn'   Sequence input with 4 dimensions
```

Create a discrete categorical actor using the network, the environment specifications, and the name of the network input layer to be associated with the observation channel.

```
actor = rlDiscreteCategoricalActor(net, ...
    obsInfo,actInfo,...
    Observation="netObsIn");
```

To check your actor use `getAction` to return one of the two possible actions, depending on a given random observation and on the current network weights.

```
act = getAction(actor,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = -10
```

To return the probability of each of the two possible action, use `evaluate`. Note that the type of the returned numbers is `single`, not `double`.

```
prb = evaluate(actor,{rand(obsInfo.Dimension)});
prb{1}
```

```
ans = 2x1 single column vector

    0.4704
    0.5296
```

You can use `getState` and `setState` to extract and set the current state of the recurrent neural network in the actor.

```
getState(actor)
```

```
ans=2×1 cell array
    {8x1 single}
    {8x1 single}
```

```
actor = setState(actor, ...
    {-0.01*single(rand(8,1)), ...
      0.01*single(rand(8,1))});
```

To evaluate the actor using sequential observations, use the sequence length (time) dimension. For example, obtain actions for 5 independent sequences each one consisting of 9 sequential observations.

```
[action,state] = getAction(actor, ...
    {rand([obsInfo.Dimension 5 9])});
```

Display the action corresponding to the seventh element of the observation sequence in the fourth sequence.

```
action = action{1};
action(1,1,4,7)
```

```
ans = 10
```

Display the updated state of the recurrent neural network.

```
state
```

```
state=2×1 cell array
    {8x5 single}
    {8x5 single}
```

You can now use the actor (along with a critic) to create an agent for the environment described by the given observation specification object. Examples of agents that can work with a continuous observation space, a discrete action space, and use a discrete categorical actor, are `rlACAgent`, `rlPGAgent`, and `rlPPOAgent`. `rlTRPOAgent` does not support actors or critics with recurrent neural networks.

For more information on input and output format for recurrent neural networks, see the Algorithms section of `lstmLayer`. For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Version History

Introduced in R2022a

See Also

Functions

`getAction` | `getModel` | `evaluate` | `getActionInfo` | `getObservationInfo`

Objects

`rlNumericSpec` | `rlFiniteSetSpec` | `rlContinuousGaussianActor` | `rlContinuousDeterministicActor` | `rlPGAgent` | `rlACAgent` | `rlPPOAgent` | `rlTRPOAgent`

Topics
“Train PG Agent to Balance Cart-Pole System”
“Train AC Agent to Balance Cart-Pole System”
“Train PPO Agent for a Lander Vehicle”
“Load Predefined Control System Environments”
“Create Policies and Value Functions”
“Reinforcement Learning Agents”

rlDQNAgent

Deep Q-network (DQN) reinforcement learning agent

Description

The deep Q-network (DQN) algorithm is a model-free, online, off-policy reinforcement learning method. A DQN agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. DQN is a variant of Q-learning, and it operates only within discrete action spaces.

For more information, “Deep Q-Network (DQN) Agents”. For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlDQNAgent(observationInfo,actionInfo)
agent = rlDQNAgent(observationInfo,actionInfo,initOpts)
```

```
agent = rlDQNAgent(critic)
```

```
agent = rlDQNAgent(critic,agentOptions)
```

Description

Create Agent from Observation and Action Specifications

`agent = rlDQNAgent(observationInfo,actionInfo)` creates a DQN agent for an environment with the given observation and action specifications, using default initialization options. The critic in the agent uses a default vector (that is, multi-output) Q-value deep neural network built from the observation specification `observationInfo` and the action specification `actionInfo`. The `ObservationInfo` and `ActionInfo` properties of `agent` are set to the `observationInfo` and `actionInfo` input arguments, respectively.

`agent = rlDQNAgent(observationInfo,actionInfo,initOpts)` creates a DQN agent for an environment with the given observation and action specifications. The agent uses a default network configured using options specified in the `initOpts` object. For more information on the initialization options, see `rlAgentInitializationOptions`.

Create Agent from Critic

`agent = rlDQNAgent(critic)` creates a DQN agent with the specified critic network using a default option set for a DQN agent.

Specify Agent Options

`agent = rlDQNAgent(critic,agentOptions)` creates a DQN agent with the specified critic network and sets the `AgentOptions` property to the `agentOptions` input argument. Use this syntax after any of the input arguments in the previous syntaxes..

Input Arguments

initOpts — Agent initialization options
rlAgentInitializationOptions object

Agent initialization options, specified as an rlAgentInitializationOptions object.

critic — Critic
rlQValueFunction object | rlVectorQValueFunction object

Critic, specified as an rlQValueFunction or as the generally more efficient rlVectorQValueFunction object. For more information on creating critics, see “Create Policies and Value Functions”.

Your critic can use a recurrent neural network as its function approximator. However, only rlVectorQValueFunction supports recurrent neural networks. For an example, see “Create DQN Agent with Recurrent Neural Network” on page 3-160.

Properties

ObservationInfo — Observation specifications
specification object | array of specification objects

Observation specifications, specified as an rlFiniteSetSpec or rlNumericSpec object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

If you create the agent by specifying a critic object, the value of ObservationInfo matches the value specified in critic.

You can extract observationInfo from an existing environment or agent using getObservationInfo. You can also construct the specifications manually using rlFiniteSetSpec or rlNumericSpec.

ActionInfo — Action specification
rlFiniteSetSpec object

Action specifications, specified as an rlFiniteSetSpec object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

If you create the agent by specifying a critic object, the value of ActionInfo matches the value specified in critic.

You can extract actionInfo from an existing environment or agent using getActionInfo. You can also construct the specification manually using rlFiniteSetSpec.

AgentOptions — Agent options
rlDQNAgentOptions object

Agent options, specified as an rlDQNAgentOptions object.

If you create a DQN agent with a default critic that uses a recurrent neural network, the default value of `AgentOptions.SequenceLength` is `32`.

`ExperienceBuffer` — Experience buffer

`rlReplayMemory` object

Experience buffer, specified as an `rlReplayMemory` object. During training the agent stores each of its experiences (*S*,*A*,*R*,*S*',*D*) in a buffer. Here:

- *S* is the current observation of the environment.
- *A* is the action taken by the agent.
- *R* is the reward for taking action *A*.
- *S*' is the next observation after taking action *A*.
- *D* is the is-done signal after taking action *A*.

`UseExplorationPolicy` — Option to use exploration policy

`false` (default) | `true`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- `true` — Use the base agent exploration policy when selecting actions.
- `false` — Use the base agent greedy policy when selecting actions.

`SampleTime` — Sample time of agent

positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations. The value of `SampleTime` matches the value specified in `AgentOptions`.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent
setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent
setCritic	Set critic of reinforcement learning agent
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create DQN Agent from Observation and Action Specifications

Create an environment with a discrete action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar with five possible elements (a torque of -2, -1, 0, 1, or 2 Nm applied to a swinging pole).

```
% Load predefined environment
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");

% Obtain observation and action specifications
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The agent creation function initializes the actor and critic networks randomly. You can ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a deep Q-network agent from the environment observation and action specifications.

```
agent = rlDQNAgent(obsInfo,actInfo);
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[1]}
```

You can now test and train the agent within the environment.

Create DQN Agent Using Initialization Options

Create an environment with a discrete action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar with five possible elements (a torque of either -2, -1, 0, 1, or 2 Nm applied to a swinging pole).

```
% Load predefined environment
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");

% Obtain observation and action specifications
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create an agent initialization option object, specifying that each hidden fully connected layer in the network must have 128 neurons (instead of the default number, 256).

```
initOpts = rlAgentInitializationOptions(NumHiddenUnit=128);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a policy gradient agent from the environment observation and action specifications.

```
agent = rlDQNAgent(obsInfo,actInfo,initOpts);
```

Extract the deep neural network from both the critic.

```
criticNet = getModel(getCritic(agent));
```

To verify that each hidden fully connected layer has 128 neurons, you can display the layers on the MATLAB® command window,

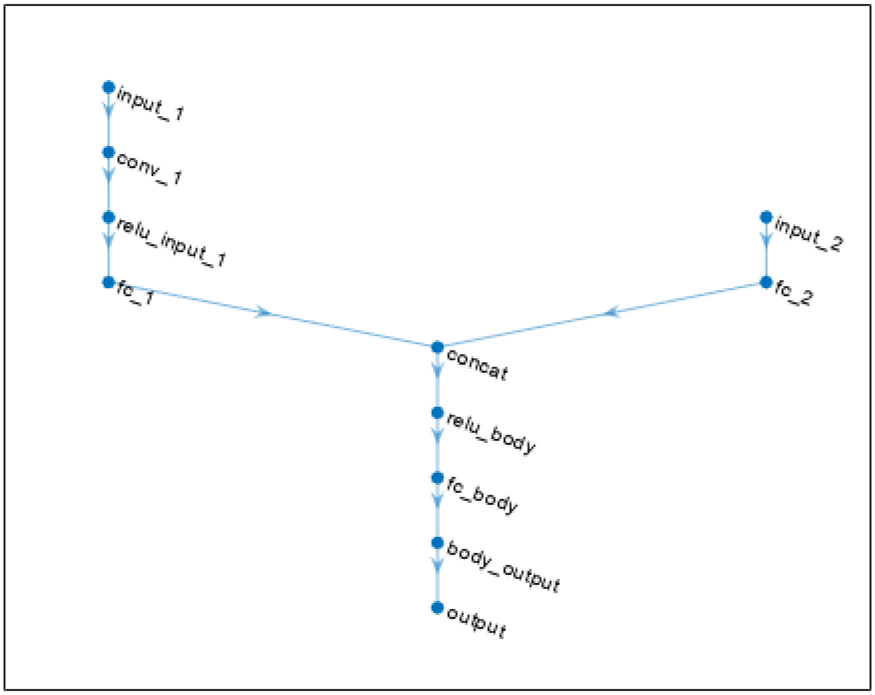
```
criticNet.Layers
```

or visualize the structure interactively using `analyzeNetwork`.

```
analyzeNetwork(criticNet)
```

Plot the critic network

```
plot(layerGraph(criticNet))
```



To check your agent, use `getAction` to return the action from random observations.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[0]}
```

You can now test and train the agent within the environment.

Create a DQN Agent Using a Multi-Output Critic

Create an environment interface and obtain its observation and action specifications. For this example load the predefined environment used for the “Train DQN Agent to Balance Cart-Pole System” example. This environment has a continuous four-dimensional observation space (the positions and velocities of both cart and pole) and a discrete one-dimensional action space consisting on the application of two possible forces, -10N or 10N.

Create the predefined environment.

```
env = rlPredefinedEnv("CartPole-Discrete");
```

Get the observation and action specification objects.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

A DQN agent approximates the long-term reward, given observations and actions, using a parametrized Q-value function critic.

For DQN agents with a discrete action space, you have the option to create a vector (that is a multi-output) Q-value function critic, which is generally more efficient than a comparable single-output critic. A vector Q-value function is a mapping from an environment observation to a vector in which each element represents the expected discounted cumulative long-term reward when an agent starts from the state corresponding to the given observation and executes the action corresponding to the element number (and follows a given policy afterwards).

To model the Q-value function within the critic, use a deep neural network. The network must have one input layer (which receives the content of the observation channel, as specified by `obsInfo`) and one output layer (which returns the vector of values for all the possible actions).

Define the network as an array of layer objects, and get the dimensions of the observation space (that is, `prod(obsInfo.Dimension)`) and the number of possible actions (that is, `numel(actInfo.Elements)`) directly from the environment specification objects.

```
dnn = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))];
```

Convert the network to a `dlnetwork` object ad display the number of weights.

```
dnn = dlnetwork(dnn);
summary(dnn)
```

```
   Initialized: true

   Number of learnables: 770

   Inputs:
      1   'input'   4 features
```

Create the critic using `rlVectorQValueFunction`, the network `dnn` as well as the observation and action specifications.

```
critic = rlVectorQValueFunction(dnn,obsInfo,actInfo);
```

Check that the critic works with a random observation input.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = 2x1 single column vector

   -0.0361
    0.0913
```

Create the DQN agent using the critic.

```
agent = rlDQNAgent(critic)
```

```
agent = 
  rlDQNAgent with properties:

        ExperienceBuffer: [1x1 rl.replay.rlReplayMemory]
            AgentOptions: [1x1 rl.option.rlDQNAgentOptions]
    UseExplorationPolicy: 0
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Specify agent options, including training options for the critic.

```
agent.AgentOptions.UseDoubleDQN=false;
agent.AgentOptions.TargetUpdateMethod="periodic";
agent.AgentOptions.TargetUpdateFrequency=4;
agent.AgentOptions.ExperienceBufferLength=100000;
agent.AgentOptions.DiscountFactor=0.99;
agent.AgentOptions.MiniBatchSize=256;

agent.AgentOptions.CriticOptimizerOptions.LearnRate=1e-2;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold=1;
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[10]}
```

You can now test and train the agent within the environment.

Create a DQN Agent Using a Single-Output Critic

Create an environment interface and obtain its observation and action specifications. For this example load the predefined environment used for the “Train DQN Agent to Balance Cart-Pole System” example. This environment has a continuous four-dimensional observation space (the positions and velocities of both cart and pole) and a discrete one-dimensional action space consisting on the application of two possible forces, -10 N or 10 N.

Create the predefined environment.

```
env = rlPredefinedEnv("CartPole-Discrete");
```

Get the observation and action specification objects.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

For DQN agents, you can use multi-output Q-value function critics, which are generally more efficient than a comparable single-output critics. However, for this example, create a single-output Q-value function critic instead.

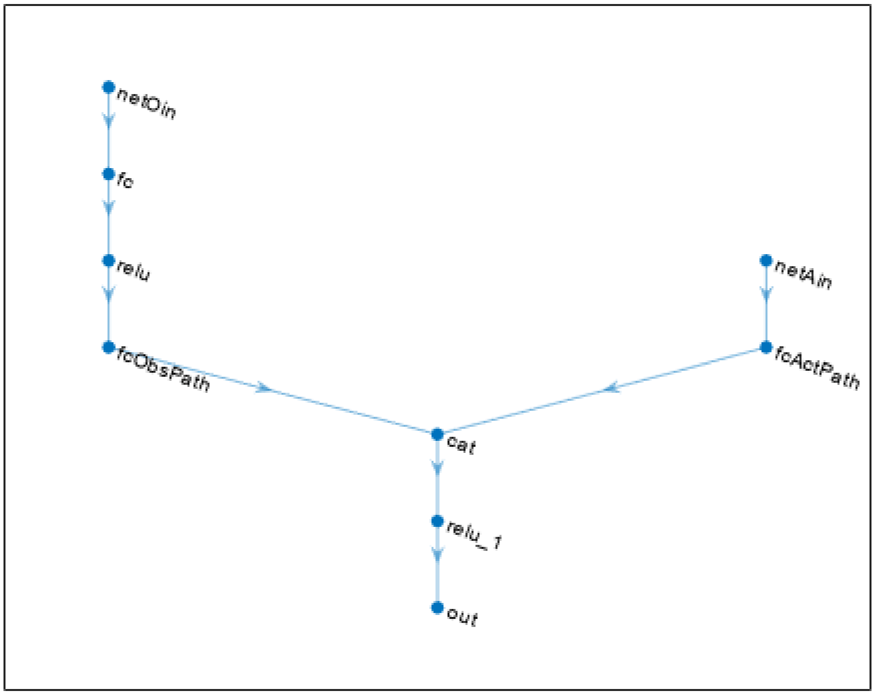
A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value).

Note that `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` return the number of dimensions of the observation and action spaces, respectively, regardless of whether they are arranged as row vectors, column vectors, or matrices.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path, so you can connect the paths.

```
% Observation path
obsPath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="netOin")
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(24,Name="fcObsPath")
    ];

% Action path
actPath = [
    featureInputLayer(prod(actInfo.Dimension),Name="netAin")
    fullyConnectedLayer(24,Name="fcActPath")
    ];

% Common  path (concatenate inputs along dim #1)
commonPath = [
    concatenationLayer(1,2,Name="cat")
    reluLayer
    fullyConnectedLayer(1,Name="out")
    ];

% Add paths to network
net = layerGraph;
net = addLayers(net,obsPath);
net = addLayers(net,actPath);
net = addLayers(net,commonPath);

% Connect layers
net = connectLayers(net,"fcObsPath","cat/in1");
net = connectLayers(net,"fcActPath","cat/in2");

% Plot network
plot(net)
```



```
% Convert to dlnetwork object
net = dlnetwork(net);

% Display the number of weights
summary(net)

   Initialized: true

   Number of learnables: 817
   Inputs:
      1   'netOin'   4 features
      2   'netAin'   1 features
```

Create the critic approximator object using `net`, the environment observation and action specifications, and the names of the network input layers to be connected with the environment observation and action channels. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(net, ...
    obsInfo, ...
    actInfo, ...
    ObservationInputNames="netOin", ...
    ActionInputNames="netAin");
```

Check the critic with a random observation and action input.

```
getValue(critic,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.0232
```

Create the DQN agent using the critic.

```
agent = rlDQNAgent(critic)
```

```
agent = 
  rlDQNAgent with properties:

        ExperienceBuffer: [1x1 rl.replay.rlReplayMemory]
            AgentOptions: [1x1 rl.option.rlDQNAgentOptions]
    UseExplorationPolicy: 0
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Specify agent options, including training options for the critic.

```
agent.AgentOptions.UseDoubleDQN=false;
agent.AgentOptions.TargetUpdateMethod="periodic";
agent.AgentOptions.TargetUpdateFrequency=4;
agent.AgentOptions.ExperienceBufferLength=100000;
agent.AgentOptions.DiscountFactor=0.99;
agent.AgentOptions.MiniBatchSize=256;

agent.AgentOptions.CriticOptimizerOptions.LearnRate=1e-2;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold=1;
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[10]}
```

You can now test and train the agent within the environment.

Create DQN Agent with Recurrent Neural Network

For this example load the predefined environment used for the “Train DQN Agent to Balance Cart-Pole System” example. This environment has a continuous four-dimensional observation space (the positions and velocities of both cart and pole) and a discrete one-dimensional action space consisting on the application of two possible forces, -10N or 10N.

```
env = rlPredefinedEnv("CartPole-Discrete");
```

Get the observation and action specification objects.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

For DQN agents, only the vector function approximator, `rlVectorQValueFunction`, supports recurrent neural networks models. The network must have one input layer (taking the content of the observation channel) and one output layer (returning the vector of values for all the possible actions).

Define the network as an array of layer objects. To create a recurrent neural network, use a `sequenceInputLayer` as the input layer and include at least one `lstmLayer`.

```
net = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(50)
    reluLayer
    lstmLayer(20,OutputMode="sequence");
    fullyConnectedLayer(20)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert to a `dlnetwork` object and display the number of weights.

```
net = dlnetwork(net);
summary(net);
```

```
   Initialized: true

   Number of learnables: 6.3k

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create the critic approximator object using `net` and the environment specifications.

```
critic = rlVectorQValueFunction(net,obsInfo,actInfo);
```

Check your critic with a random input observation.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = 2x1 single column vector

    0.0136
    0.0067
```

Define some training options for the critic.

```
criticOptions = rlOptimizerOptions( ...
    LearnRate=1e-3, ...
    GradientThreshold=1);
```

Specify options for creating the DQN agent. To use a recurrent neural network, you must specify a `SequenceLength` greater than 1.

```
agentOptions = rlDQNAgentOptions(...
    UseDoubleDQN=false, ...
    TargetSmoothFactor=5e-3, ...
    ExperienceBufferLength=1e6, ...
    SequenceLength=32, ...
    CriticOptimizerOptions=criticOptions);

agentOptions.EpsilonGreedyExploration.EpsilonDecay = 1e-4;
```

Create the agent. The actor and critic networks are initialized randomly.

```
agent = rlDQNAgent(critic,agentOptions)
```

```
agent = 
  rlDQNAgent with properties:

        ExperienceBuffer: [1x1 rl.replay.rlReplayMemory]
            AgentOptions: [1x1 rl.option.rlDQNAgentOptions]
    UseExplorationPolicy: 0
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Check your agent using `getAction` to return the action from a random observation.

```
getAction(agent,rand(obsInfo.Dimension))
```

```
ans = 1x1 cell array
    {[-10]}
```

To evaluate the agent using sequential observations, use the sequence length (time) dimension. For example, obtain actions for a sequence of 9 observations.

```
[action,state] = getAction(agent, ...
    {rand([obsInfo.Dimension 1 9])});
```

Display the action corresponding to the seventh element of the observation.

```
action = action{1};
action(1,1,1,7)
```

```
ans = -10
```

You can now test and train the agent within the environment.

Version History

Introduced in R2019a

See Also

Apps

Reinforcement Learning Designer | Deep Network Designer

Functions

`getAction` | `getActor` | `getCritic` | `getModel` | `generatePolicyFunction` | `generatePolicyBlock` | `getActionInfo` | `getObservationInfo`

Objects

`rlDQNAgentOptions` | `rlAgentInitializationOptions` | `rlVectorQValueFunction` | `rlQValueFunction` | `rlQAgent` | `rlSARSAAgent`

Blocks

RL Agent | Policy

Topics

"Train AC Agent to Balance Cart-Pole System"
"Train Reinforcement Learning Agents"
"Deep Q-Network (DQN) Agents"
"Reinforcement Learning Agents"

rlDQNAgentOptions

Options for DQN agent

Description

Use an `rlDQNAgentOptions` object to specify options for deep Q-network (DQN) agents. To create a DQN agent, use `rlDQNAgent`.

For more information, see “Deep Q-Network (DQN) Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlDQNAgentOptions
opt = rlDQNAgentOptions(Name,Value)
```

Description

`opt = rlDQNAgentOptions` creates an options object for use as an argument when creating a DQN agent using all default settings. You can modify the object properties using dot notation.

`opt = rlDQNAgentOptions(Name,Value)` sets option properties on page 3-164 using name-value pairs. For example, `rlDQNAgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`UseDoubleDQN` — Flag for using double DQN
true (default) | false

Flag for using double DQN for value function target updates, specified as a logical value. For most application set `UseDoubleDQN` to `"on"`. For more information, see “Deep Q-Network (DQN) Agents”.

`EpsilonGreedyExploration` — Options for epsilon-greedy exploration
`EpsilonGreedyExploration` object

Options for epsilon-greedy exploration, specified as an `EpsilonGreedyExploration` object with the following properties.

Property	Description	Default Value
Epsilon	Probability threshold to either randomly select an action or select the action that maximizes the state-action value function. A larger value of Epsilon means that the agent randomly explores the action space at a higher rate.	1
EpsilonMin	Minimum value of Epsilon	0.01
EpsilonDecay	Decay rate	0.0050

At the end of each training time step, if Epsilon is greater than EpsilonMin, then it is updated using the following formula.

```
Epsilon = Epsilon*(1-EpsilonDecay)
```

Note that Epsilon is conserved between the end of an episode and the start of the next one. Therefore, it keeps on uniformly decreasing over multiple episodes until it reaches EpsilonMin.

If your agent converges on local optima too quickly, you can promote agent exploration by increasing Epsilon.

To specify exploration options, use dot notation after creating the rlDQNAgentOptions object opt. For example, set the epsilon value to 0.9.

```
opt.EpsilonGreedyExploration.Epsilon = 0.9;
```

CriticOptimizerOptions — Critic optimizer options

rlOptimizerOptions object

Critic optimizer options, specified as an rlOptimizerOptions object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see rlOptimizerOptions and rlOptimizer.

BatchDataRegularizerOptions — Batch data regularizer options

[] (default) | rlConservativeQLearningOptions object

Batch data regularizer options, specified as an rlConservativeQLearningOptions object. These options are used to train the agent offline, from existing data. If you leave this option empty, a default rlConservativeQLearningOptions options object is used (if needed).

For more information, see rlConservativeQLearningOptions.

Example: opt.BatchDataRegularizerOptions = rlConservativeQLearningOptions(MinQValueWeight=5)

TargetSmoothFactor — Smoothing factor for target critic updates

1e-3 (default) | positive scalar less than or equal to 1

Smoothing factor for target critic updates, specified as a positive scalar less than or equal to 1. For more information, see “Target Update Methods”.

`TargetUpdateFrequency` — Number of steps between target critic updates

`1` (default) | positive integer

Number of steps between target critic updates, specified as a positive integer. For more information, see “Target Update Methods”.

`ResetExperienceBufferBeforeTraining` — Option for clearing the experience buffer

`false` (default) | `true`

Option for clearing the experience buffer before training, specified as a logical value.

`SequenceLength` — Maximum batch-training trajectory length when using RNN

`1` (default) | positive integer

Maximum batch-training trajectory length when using a recurrent neural network for the critic, specified as a positive integer. This value must be greater than `1` when using a recurrent neural network for the critic and `1` otherwise.

`MiniBatchSize` — Size of random experience mini-batch

`64` (default) | positive integer

Size of random experience mini-batch, specified as a positive integer. During each training episode, the agent randomly samples experiences from the experience buffer when computing gradients for updating the critic properties. Large mini-batches reduce the variance when computing gradients but increase the computational effort.

When using a recurrent neural network for the critic, `MiniBatchSize` is the number of experience trajectories in a batch, where each trajectory has length equal to `SequenceLength`.

`NumStepsToLookAhead` — Number of future rewards used to estimate the value of the policy

`1` (default) | positive integer

Number of future rewards used to estimate the value of the policy, specified as a positive integer. For more information, see chapter 7 of [1].

N-step Q learning is not supported when using a recurrent neural network for the critic. In this case, `NumStepsToLookAhead` must be `1`.

`ExperienceBufferLength` — Experience buffer size

`10000` (default) | positive integer

Experience buffer size, specified as a positive integer. During training, the agent computes updates using a mini-batch of experiences randomly sampled from the buffer.

`SampleTime` — Sample time of agent

`1` (default) | positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

`DiscountFactor` — Discount factor

`0.99` (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlDQNAgent	Deep Q-network (DQN) reinforcement learning agent

Examples

Create DQN Agent Options Object

Create an `rlDQNAgentOptions` object that specifies the agent mini-batch size.

```
opt = rlDQNAgentOptions(MiniBatchSize=48)
```

```
opt = 
  rlDQNAgentOptions with properties:

                           UseDoubleDQN: 1
               EpsilonGreedyExploration: [1x1 rl.option.EpsilonGreedyExploration]
                 CriticOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
            BatchDataRegularizerOptions: []
                     TargetSmoothFactor: 1.0000e-03
                  TargetUpdateFrequency: 1
    ResetExperienceBufferBeforeTraining: 0
                         SequenceLength: 1
                          MiniBatchSize: 48
                    NumStepsToLookAhead: 1
                 ExperienceBufferLength: 10000
                             SampleTime: 1
                         DiscountFactor: 0.9900
                             InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

Version History

Introduced in R2019a

R2020a: Target update method settings for DQN agents have changed

Behavior changed in R2020a

Target update method settings for DQN agents have changed. The following changes require updates to your code:

- The TargetUpdateMethod option has been removed. Now, DQN agents determine the target update method based on the TargetUpdateFrequency and TargetSmoothFactor option values.
- The default value of TargetUpdateFrequency has changed from 4 to 1.

To use one of the following target update methods, set the TargetUpdateFrequency and TargetSmoothFactor properties as indicated.

Update Method	TargetUpdateFrequency	TargetSmoothFactor
Smoothing	1	Less than 1
Periodic	Greater than 1	1
Periodic smoothing (new method in R2020a)	Greater than 1	Less than 1

The default target update configuration, which is a smoothing update with a TargetSmoothFactor value of 0.001, remains the same.

Update Code

This table shows some typical uses of rlDQNAgentOptions and how to update your code to use the new option configuration.

Not Recommended	Recommended
`opt = rlDQNAgentOptions('TargetUpdateMethod',"smoothing");`	`opt = rlDQNAgentOptions;`

`opt = rlDQNAgentOptions('TargetUpdateMethod',"periodic");`	`opt = rlDQNAgentOptions;`
	`opt.TargetUpdateFrequency = 4;`
	`opt.TargetSmoothFactor = 1;`

`opt = rlDQNAgentOptions;`
`opt.TargetUpdateMethod = "periodic";`
`opt.TargetUpdateFrequency = 5;`
	`opt = rlDQNAgentOptions;`
	`opt.TargetUpdateFrequency = 5;`
	`opt.TargetSmoothFactor = 1;`

R2022a: The default value of the ResetExperienceBufferBeforeTraining property has changed

Behavior changed in R2022a

The default value of the ResetExperienceBufferBeforeTraining has changed from true to false.

When creating a new DQN agent, if you want to clear the experience buffer before training, you must specify ResetExperienceBufferBeforeTraining as true. For example, before training, set the property using dot notation.

```
agent.AgentOptions.ResetExperienceBufferBeforeTraining = true;
```

Alternatively, you can set the property to true in an rlDQNAgentOptions object and use this object to create the DQN agent.

References

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning. Cambridge, Mass: The MIT Press, 2018.

See Also

Objects
`rlDQNAgent` | `rlAgentInitializationOptions` | `rlConservativeQLearningOptions`

Topics
"Train DQN Agent to Balance Cart-Pole System"
"Deep Q-Network (DQN) Agents"

rlEpsilonGreedyPolicy

Policy object to generate discrete epsilon-greedy actions for custom training loops

Description

This object implements an epsilon-greedy policy, which returns either the action that maximizes a discrete action-space Q-value function, with probability `1-Epsilon`, or a random action otherwise, given an input observation. You can create an `rlEpsilonGreedyPolicy` object from an `rlQValueFunction` or `rlVectorQValueFunction` object, or extract it from an `rlQAgent`, `rlDQNAgent` or `rlSARSAAgent`. You can then train the policy object using a custom training loop or deploy it for your application. If `UseEpsilonGreedyAction` is set to `0` the policy is deterministic, therefore in this case it does not explore. This object is not compatible with `generatePolicyBlock` and `generatePolicyFunction`. For more information on policies and value functions, see “Create Policies and Value Functions”.

Creation

Syntax

```
policy = rlEpsilonGreedyPolicy(qValueFunction)
```

Description

`policy = rlEpsilonGreedyPolicy(qValueFunction)` creates the epsilon-greedy policy object `policy` from the discrete action-space Q-value function `qValueFunction`. It also sets the `QValueFunction` property of `policy` to the input argument `qValueFunction`.

Properties

`QValueFunction` — Discrete action-space Q-value function
`rlQValueFunction` object | `rlVectorQValueFunction` object

Discrete action-space Q-value function approximator, specified as an `rlQValueFunction` or `rlVectorQValueFunction` object.

`ExplorationOptions` — Noise model options
`EpsilonGreedyExploration` object

Exploration options, specified as an `EpsilonGreedyExploration` object. Changing the noise state or any exploration option of an `rlEpsilonGreedyPolicy` object deployed through code generation is not supported.

For more information see the `EpsilonGreedyExploration` property in `rlQAgentOptions`.

`UseEpsilonGreedyAction` — Option to enable epsilon-greedy actions
`true` (default) | `false`

Option to enable epsilon-greedy actions, specified as a logical value: either `true` (default, enabling epsilon-greedy actions, which helps exploration) or `false` (epsilon-greedy actions not enabled). When epsilon-greedy actions are disabled the policy is deterministic and therefore it does not explore.

Example: `false`

`EnableEpsilonDecay` — Option to enable epsilon decay

`true` (default) | `false`

Option to enable epsilon decay, specified as a logical value: either `true` (default, enabling epsilon decay) or `false` (disabling epsilon decay).

Example: `false`

`ObservationInfo` — Observation specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array of such objects. These objects define properties such as the dimensions, data types, and names of the observation channels.

`ActionInfo` — Action specifications

`rlFiniteSetSpec` object

Action specifications, specified as an `rlFiniteSetSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

`SampleTime` — Sample time of policy

positive scalar | `-1` (default)

Sample time of the policy, specified as a positive scalar or as `-1` (default). Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the policy is specified executes every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the policy is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience. If `SampleTime` is `-1`, the sample time is treated as being equal to `1`.

Example: `0.2`

Object Functions

getAction	Obtain action from agent, actor, or policy object given environment observations
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
reset	Reset environment, agent, experience buffer, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object

Examples

Create Epsilon-Greedy Policy Object from Vector Q-Value Function

Create observation and action specification objects. For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing four doubles, and the action space as a finite set consisting of two possible row vectors, [1 0] and [0 1].

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlFiniteSetSpec({[1 0],[0 1]});
```

Alternatively, use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment.

Create a vector Q-value function approximator to use as critic. A vector Q-value function must accept an observation as input and return a single vector with as many elements as the number of possible discrete actions.

To model the parametrized vector Q-value function within the critic, use a neural network. Define a single path from the network input to its output as an array of layer objects.

```
layers = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(10)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
model = dlnetwork(layers);
summary(model)
```

```
   Initialized: true

   Number of learnables: 72

   Inputs:
      1   'input'   4 features
```

Create a vector Q-value function using `model`, and the observation and action specifications.

```
qValueFcn = rlVectorQValueFunction(model,obsInfo,actInfo)
```

```
qValueFcn = 
  rlVectorQValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
          UseDevice: "cpu"
```

Check the critic with a random observation input.

```
getValue(qValueFcn,{rand(obsInfo.Dimension)})
```

```
ans = 2x1 single column vector
```

```
    0.6486
   -0.3103
```

Create a policy object from `qValueFcn`.

```
policy = rlEpsilonGreedyPolicy(qValueFcn)
```

```
policy = 
  rlEpsilonGreedyPolicy with properties:

            QValueFunction: [1x1 rl.function.rlVectorQValueFunction]
        ExplorationOptions: [1x1 rl.option.EpsilonGreedyExploration]
    UseEpsilonGreedyAction: 1
        EnableEpsilonDecay: 1
           ObservationInfo: [1x1 rl.util.rlNumericSpec]
                ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
                SampleTime: -1
```

Check the policy with a random observation input.

```
getAction(policy,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[1 0]}
```

You can now train the policy with a custom training loop and then deploy it to your application.

Version History

Introduced in R2022a

See Also

Functions

`getGreedyPolicy` | `getExplorationPolicy` | `generatePolicyBlock` | `generatePolicyFunction` | `getAction` | `getLearnableParameters` | `setLearnableParameters`

Objects

`rlMaxQPolicy` | `rlDeterministicActorPolicy` | `rlAdditiveNoisePolicy` | `rlStochasticActorPolicy` | `rlQValueFunction` | `rlVectorQValueFunction` | `rlSARSAAgent` | `rlQAgent` | `rlDQNAgent`

Blocks

Policy | RL Agent

Topics

“Model-Based Reinforcement Learning Using Custom Training Loop”
“Train Reinforcement Learning Policy Using Custom Training Loop”
“Create Policies and Value Functions”

rlFiniteSetSpec

Create discrete action or observation data specifications for reinforcement learning environments

Description

An `rlFiniteSetSpec` object specifies discrete action or observation data specifications for reinforcement learning environments.

Creation

Syntax

```
spec = rlFiniteSetSpec(elements)
```

Description

`spec = rlFiniteSetSpec(elements)` creates a data specification with a discrete set of actions or observations, setting the Elements property.

Properties

`Elements` — Set of valid actions or observations
vector | cell array

Set of valid actions or observations for the environment, specified as one of the following:

- Vector — Specify valid numeric values for a single action or single observation.
- Cell array — Specify valid numeric value combinations when you have more than one action or observation. Each entry of the cell array must have the same dimensions.

`Name` — Name of the `rlFiniteSetSpec` object
string

Name of the `rlFiniteSetSpec` object, specified as a string. Use this property to set a meaningful name for the signal carried by this data channel. This property is used by the RL Agent block to match the bus signals with their corresponding environment channels.

`Description` — Description of the `rlFiniteSetSpec` object
string

Description of the `rlFiniteSetSpec` object, specified as a string. You can use this property to specify a meaningful description of the signal carried by this environment channel.

`Dimension` — Size of each element
vector (default)

This property is read-only.

Size of each element, specified as a vector.

If you specify `Elements` as a vector, then `Dimension` is `[1 1]`. Otherwise, if you specify a cell array, then `Dimension` indicates the size of the entries in `Elements`. This property is essential to construct agents and function approximators objects that work with a given environment.

`DataType` — Information about the type of data
`"double"` (default) | string

This property is read-only.

Information about the type of data, specified as a string, such as `"double"` or `"single"`. The software uses this property to enforce data type consistency for observations and actions.

Object Functions

rlSimulinkEnv	Create reinforcement learning environment using dynamic model implemented in Simulink
rlFunctionEnv	Specify custom reinforcement learning environment dynamics using functions
rlValueFunction	Value function approximator object for reinforcement learning agents
rlQValueFunction	Q-Value function approximator object for reinforcement learning agents
rlVectorQValueFunction	Vector Q-value function approximator for reinforcement learning agents
rlContinuousDeterministicActor	Deterministic actor with a continuous action space for reinforcement learning agents
rlDiscreteCategoricalActor	Stochastic categorical actor with a discrete action space for reinforcement learning agents
rlContinuousGaussianActor	Stochastic Gaussian actor with a continuous action space for reinforcement learning agents

Examples

Create Reinforcement Learning Environment for Simulink Model

For this example, consider the `rlSimplePendulumModel` Simulink model. The model is a simple frictionless pendulum that initially hangs in a downward position.

Open the model.

```
mdl = "rlSimplePendulumModel";
open_system(mdl)
```

Create `rlNumericSpec` and `rlFiniteSetSpec` objects for the observation and action information, respectively.

The observation is a vector containing three signals: the sine, cosine, and time derivative of the angle.

```
obsInfo = rlNumericSpec([3 1])
```

```
obsInfo = 
  rlNumericSpec with properties:
     LowerLimit: -Inf
     UpperLimit: Inf
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [3 1]
       DataType: "double"
```

The action is a scalar expressing the torque and can be one of three possible values, -2 Nm, 0 Nm and 2 Nm.

```
actInfo = rlFiniteSetSpec([-2 0 2])
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [3x1 double]
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

You can use dot notation to assign property values for the `rlNumericSpec` and `rlFiniteSetSpec` objects.

```
obsInfo.Name = "observations";
actInfo.Name = "torque";
```

Assign the agent block path information, and create the reinforcement learning environment for the Simulink model using the information extracted in the previous steps.

```
agentBlk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,agentBlk,obsInfo,actInfo)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

You can also include a reset function using dot notation. For this example, randomly initialize `theta0` in the model workspace.

```
env.ResetFcn = @(in) setVariable(in,"theta0",randn,"Workspace",mdl)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : @(in)setVariable(in,"theta0",randn,"Workspace",mdl)
  UseFastRestart : on
```

Specify Discrete Value Set for Multiple Actions

If the actor for your reinforcement learning agent has multiple outputs, each with a discrete action space, you can specify the possible discrete actions combinations using an `rlFiniteSetSpec` object.

Suppose that the valid values for a two-output system are `[1 2]` for the first output and `[10 20 30]` for the second output. Create a discrete action space specification for all possible input combinations.

```
actionSpec = rlFiniteSetSpec({[1 10],[1 20],[1 30],...
                              [2 10],[2 20],[2 30]})
```

```
actionSpec = 
  rlFiniteSetSpec with properties:

       Elements: {6x1 cell}
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [1 2]
       DataType: "double"
```

Version History

Introduced in R2019a

See Also

Functions
`rlSimulinkEnv` | `getActionInfo` | `getObservationInfo`

Objects
`rlNumericSpec` | `rlValueRepresentation` | `rlQValueRepresentation` | `rlDeterministicActorRepresentation` | `rlStochasticActorRepresentation` | `rlFunctionEnv`

Blocks
RL Agent

Topics
"Create MATLAB Reinforcement Learning Environments"
"Create Simulink Reinforcement Learning Environments"

rlFunctionEnv

Specify custom reinforcement learning environment dynamics using functions

Description

Use `rlFunctionEnv` to define a custom reinforcement learning environment. You provide MATLAB functions that define the step and reset behavior for the environment. This object is useful when you want to customize your environment beyond the predefined environments available with `rlPredefinedEnv`.

Creation

Syntax

```
env = rlFunctionEnv(observationInfo,actionInfo,stepfcn,resetfcn)
```

Description

`env = rlFunctionEnv(observationInfo,actionInfo,stepfcn,resetfcn)` creates a reinforcement learning environment using the provided observation and action specifications, `observationInfo` and `actionInfo`, respectively. You also set the `StepFcn` and `ResetFcn` properties using MATLAB functions.

Input Arguments

`observationInfo` — Observation specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

You can extract `observationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually.

`actionInfo` — Action specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object

Action specifications, specified either as an `rlFiniteSetSpec` (for discrete action spaces) or `rlNumericSpec` (for continuous action spaces) object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specifications manually.

Properties

StepFcn — Step behavior for the environment

function name | function handle | anonymous function handle

Step behavior for the environment, specified as a function name, function handle, or anonymous function.

StepFcn is a function that you provide which describes how the environment advances to the next state from a given action. When using a function name or function handle, this function must have two inputs and four outputs, as illustrated by the following signature.

```
[Observation,Reward,IsDone,LoggedSignals] = myStepFunction(Action,LoggedSignals)
```

To use additional input arguments beyond the required set, specify StepFcn using an anonymous function handle.

The step function computes the values of the observation and reward for the given action in the environment. The required input and output arguments are as follows.

- Action — Current action, which must match the dimensions and data type specified in actionInfo.
- Observation — Returned observation, which must match the dimensions and data types specified in observationInfo.
- Reward — Reward for the current step, returned as a scalar value.
- IsDone — Logical value indicating whether to end the simulation episode. The step function that you define can include logic to decide whether to end the simulation based on the observation, reward, or any other values.
- LoggedSignals — Any data that you want to pass from one step to the next, specified as a structure.

For an example showing multiple ways to define a step function, see “Create MATLAB Environment Using Custom Functions”.

ResetFcn — Reset behavior for the environment

function name | function handle | anonymous function handle

Reset behavior for the environment, specified as a function, function handle, or anonymous function handle.

The reset function that you provide must have no inputs and two outputs, as illustrated by the following signature.

```
[InitialObservation,LoggedSignals] = myResetFunction
```

To use input arguments with your reset function, specify ResetFcn using an anonymous function handle.

The reset function sets the environment to an initial state and computes the initial values of the observation signals. For example, you can create a reset function that randomizes certain state values, such that each training episode begins from different initial conditions.

The sim function calls the reset function to reset the environment at the start of each simulation, and the train function calls it at the start of each training episode.

The `InitialObservation` output must match the dimensions and data type of `observationInfo`.

To pass information from the reset condition into the first step, specify that information in the reset function as the output structure `LoggedSignals`.

For an example showing multiple ways to define a reset function, see “Create MATLAB Environment Using Custom Functions”.

`LoggedSignals` — Information to pass to next step

structure

Information to pass to the next step, specified as a structure. When you create the environment, whatever you define as the `LoggedSignals` output of `ResetFcn` initializes this property. When a step occurs, the software populates this property with data to pass to the next step, as defined in `StepFcn`.

Object Functions

getActionInfo	Obtain action data specifications from reinforcement learning environment, agent, or experience buffer
getObservationInfo	Obtain observation data specifications from reinforcement learning environment, agent, or experience buffer
train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
validateEnvironment	Validate custom reinforcement learning environment

Examples

Create Custom MATLAB Environment

Create a reinforcement learning environment by supplying custom dynamic functions in MATLAB®. Using `rlFunctionEnv`, you can create a MATLAB reinforcement learning environment from an observation specification, action specification, and `step` and `reset` functions that you define.

For this example, create an environment that represents a system for balancing a cart on a pole. The observations from the environment are the cart position, cart velocity, pendulum angle, and pendulum angle derivative. (For additional details about this environment, see “Create MATLAB Environment Using Custom Functions”.) Create an observation specification for those signals.

```
oinfo = rlNumericSpec([4 1]);
oinfo.Name = "CartPole States";
oinfo.Description = 'x, dx, theta, dtheta';
```

The environment has a discrete action space where the agent can apply one of two possible force values to the cart, –10 N or 10 N. Create the action specification for those actions.

```
ActionInfo = rlFiniteSetSpec([-10 10]);
ActionInfo.Name = "CartPole Action";
```

Next, specify the custom `step` and `reset` functions. For this example, use the supplied functions `myResetFunction.m` and `myStepFunction.m`. For details about these functions and how they are constructed, see “Create MATLAB Environment Using Custom Functions”.

Construct the custom environment using the defined observation specification, action specification, and function names.

```
env = rlFunctionEnv(oinfo,ActionInfo,"myStepFunction","myResetFunction");
```

You can create agents for `env` and train them within the environment as you would for any other reinforcement learning environment.

As an alternative to using function names, you can specify the functions as function handles. For more details and an example, see “Create MATLAB Environment Using Custom Functions”.

Version History

Introduced in R2019a

See Also

Functions

`rlPredefinedEnv` | `rlCreateEnvTemplate`

Topics

“Create MATLAB Environment Using Custom Functions”

rlHindsightPrioritizedReplayMemory

Hindsight replay memory experience buffer with prioritized sampling

Description

An off-policy reinforcement learning agent stores experiences in a circular experience buffer. During training, the agent samples mini-batches of experiences from the buffer and uses these mini-batches to update its actor and critic function approximators.

By default, built-in off-policy agents (DQN, DDPG, TD3, SAC, MBPO) use an `rlReplayMemory` object as their experience buffer. For goal-conditioned tasks, where the observation includes both the goal and a goal measurement, you can use an `rlHindsightPrioritizedReplayMemory` object.

`rlHindsightReplayMemory` objects uniformly sample experiences from the buffer. To use prioritized nonuniform sampling, which can improve sample efficiency, use an `rlHindsightPrioritizedReplayMemory` object.

A hindsight replay memory experience buffer:

- Generates additional experiences by replacing goals with goal measurements
- Improves sample efficiency for tasks with sparse rewards
- Requires a ground-truth reward function and is-done function
- Is not necessary when you have a well-shaped reward function

For more information on hindsight experience replay and prioritized sampling, see “Algorithms” on page 3-187.

Creation

Syntax

```
buffer = rlHindsightPrioritizedReplayMemory(obsInfo,actInfo,rewardFcn,
isDoneFcn,goalConditionInfo)
buffer = rlHindsightReplayMemory(obsInfo,actInfo,rewardFcn,isDoneFcn,
goalConditionInfo,maxLength)
```

Description

`buffer = rlHindsightPrioritizedReplayMemory(obsInfo,actInfo,rewardFcn, isDoneFcn,goalConditionInfo)` creates a hindsight prioritized replay memory experience buffer that is compatible with the observation and action specifications in `obsInfo` and `actInfo`, respectively. This syntax sets the `RewardFcn`, `IsDoneFcn`, and `GoalConditionInfo` properties.

`buffer = rlHindsightReplayMemory(obsInfo,actInfo,rewardFcn,isDoneFcn, goalConditionInfo,maxLength)` sets the maximum length of the buffer by setting the `MaxLength` property.

Input Arguments

`obsInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, specified as a reinforcement learning specification object or an array of specification objects defining properties such as dimensions, data types, and names of the observation signals.

You can extract the observation specifications from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`actInfo` — Action specifications
specification object | array of specification objects

Action specifications, specified as a reinforcement learning specification object defining properties such as dimensions, data types, and names of the action signals.

You can extract the action specifications from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

Properties

`MaxLength` — Maximum buffer length
`10000` (default) | nonnegative integer

This property is read-only.

Maximum buffer length, specified as a nonnegative integer.

To change the maximum buffer length, use the `resize` function.

`Length` — Number of experiences in buffer
`0` (default) | nonnegative integer

This property is read-only.

Number of experiences in buffer, specified as a nonnegative integer.

`GoalConditionInfo` — Goal condition information
1-by-*N* cell array

Goal condition information, specified as a 1-by-*N* cell array, where *N* is the number of goal conditions. For the *i*th goal condition, the corresponding cell of `GoalConditionInfo` contains a 1-by-4 cell array with the following elements.

- `GoalConditionInfo{i}{1}` — Goal measurement channel index.
- `GoalConditionInfo{i}{2}` — Goal measurement element indices.
- `GoalConditionInfo{i}{3}` — Goal channel index.
- `GoalConditionInfo{i}{4}` — Goal element indices.

The goal measurements in GoalConditionInfo{i}{2} correspond to the goals in GoalConditionInfo{i}{4}.

As an example, suppose that obsInfo contains specifications for two observation channels. Further, suppose that there is one goal condition where the goal measurements correspond to elements 2 and 3 of the first observation channel, and the goals correspond to elements 4 and 5 of the second observation channel. In this case, the goal condition information is:

```
GoalConditionInfo = {{1,[1 2],2,[4 5]}};
```

RewardFcn — Reward function

function handle

Reward function, specified as a handle to a function with the following signature.

```
function reward = myRewardFcn(obs,action,nextObs)
```

Here:

- reward is a scalar reward value.
- obs is the current observation.
- act is the action taken from the current observation.
- nextObs is the next observation after taking the specified action.

IsDoneFcn — Is-done function

function handle

Is-done function, specified as a handle to a function with the following signature.

```
function isdone = myIsDoneFcn(obs,action,nextObs)
```

Here:

- isdone is true when the next observation is a terminal condition and false otherwise.
- obs is the current observation.
- act is the action taken from the current observation.
- nextObs is the next observation after taking the specified action.

Strategy — Goal measurement sampling strategy

"final" (default) | "episode" | "future"

Goal measurement sampling strategy, specified as one of the following values.

- "final" — Use the goal measurement from the end of the trajectory.
- "episode" — Randomly sample *M* goal measurements from the trajectory, where *M* is equal to NumGoalSamples.
- "future" — Randomly sample *M* goal measurements from the trajectory, but create hindsight experiences for measurements that were observed at time t+1 or later.

NumGoalSamples — Number of goal measurements

4 (default) | positive integer

Number of goal measurements to sample when generating experiences, specified as a positive integer. This parameter is ignored when Strategy is "final"

`PriorityExponent` — Priority exponent

`0.6` (default) | nonnegative scalar less than or equal to 1

Priority exponent to control the impact of prioritization during probability computation, specified as a nonnegative scalar less than or equal to 1.

If the priority exponent is zero, the agent uses uniform sampling.

`InitialImportanceSamplingExponent` — Initial value of importance sampling exponent

`0.4` (default) | nonnegative scalar less than or equal to 1

Initial value of the importance sampling exponent, specified as a nonnegative scalar less than or equal to 1.

`NumAnnealingSteps` — Number of annealing steps

`1000000` (default) | positive integer

Number of annealing steps for updating the importance sampling exponent, specified as a positive integer.

`ImportanceSamplingExponent` — Current value of importance sampling exponent

`0.4` (default) | nonnegative scalar less than or equal to 1

This property is read-only.

Current value of the importance sampling exponent, specified as a nonnegative scalar less than or equal to 1.

During training, `ImportanceSamplingExponent` is linearly increased from `InitialImportanceSamplingExponent` to 1 over `NumAnnealingSteps` steps.

Object Functions

append	Append experiences to replay memory buffer
sample	Sample experiences from replay memory buffer
resize	Resize replay memory experience buffer
reset	Reset environment, agent, experience buffer, or policy object
allExperiences	Return all experiences in replay memory buffer
validateExperience	Validate experiences for replay memory
generateHindsightExperiences	Generate hindsight experiences from hindsight experience replay buffer
getActionInfo	Obtain action data specifications from reinforcement learning environment, agent, or experience buffer
getObservationInfo	Obtain observation data specifications from reinforcement learning environment, agent, or experience buffer

Examples

Create DDPG Agent With Hindsight Prioritized Replay Memory

For this example, create observation and action specifications directly. You can also extract such specifications from your environment.

Create an observation specification for an environment with a single a observation channel with six observations. For this example, assume that the observation channel contains the signals [a, x_m, y_m, x_g, y_g, c], where:

- x_g and y_g are the goal observations.
- x_m and y_m are the goal measurements.
- a and c are additional observations.

```
obsInfo = rlNumericSpec([5 1]);
```

Create a specification for a single action.

```
actInfo = rlNumericSpec([1 1]);
```

Create a DDPG agent from the environment specifications. By default, the agent uses a replay memory experience buffer with uniform sampling.

```
agent = rlDDPGAgent(obsInfo,actInfo);
```

To create a hindsight replay memory buffer, first define the goal condition information. Both the goals and goal measurements are in the single observation channel. The goal measurements are in elements 2 and 3 of the observation channel and the goals are in elements 4 and 5 of the observation channel.

```
goalConditionInfo = {{1,[2 3],1,[4 5]}};
```

Define an is-done function. For this example, the is-done signal is `true` when the next observation satisfies the goal condition $\sqrt{(x_m - x_g)^2 + (y_m - y_g)^2} < 0.1$.

```
function isdone = hindsightIsDoneFcn1(Observation,Action,NextObservation)
    NextObservation = NextObservation{1};
    xm = NextObservation(2);
    ym = NextObservation(3);
    xg = NextObservation(4);
    yg = NextObservation(5);
    isdone = sqrt((xm-xg)^2 + (ym-yg)^2) < 0.1;
end
```

Define a reward function. For this example, the reward is `1` when the is-done signal is `true` and –`0.01` otherwise.

```
function reward = hindsightRewardFcn1(Observation,Action,NextObservation)
    isdone = hindsightIsDoneFcn1(Observation,Action,NextObservation);
    if isdone
        reward = 1;
    else
        reward = -0.01;
    end
end
```

Create a hindsight prioritized replay memory buffer with a default maximum length.

```
buffer = rlHindsightPrioritizedReplayMemory(obsInfo,actInfo, ...
    @hindsightRewardFcn1,@hindsightIsDoneFcn1,goalConditionInfo);
```

Configure the prioritized replay memory options. For example, set the initial importance sampling exponent to `0.5` and the number of annealing steps for updating the exponent during training to `1e4`.

```
buffer.NumAnnealingSteps = 1e4;
buffer.PriorityExponent = 0.5;
buffer.InitialImportanceSamplingExponent = 0.5;
```

Replace the default experience buffer with the hindsight replay memory buffer.

```
agent.ExperienceBuffer = buffer;
```

Limitations

- Hindsight prioritized experience replay does not support agents that use recurrent neural networks.

Algorithms

Hindsight Experience Replay

Hindsight experience replay is a data augmentation method that you can use for goal-conditioned tasks, where the observation includes both the goal and a goal measurement. In such a goal-conditioned task, the agent reaches the goal when the distance between the goal measurement and the goal is less than a threshold.

However, when the reward is sparse, the agent may not be able to reach the goal during training. In this case, the agent does not get any rewards. Hindsight experience replay generates experience data by replacing the goal with a goal measurement. As a result, the agent can get rewards for improving the goal measurement without actually reaching the goal.

The generated hindsight experiences are created at the end of each training episode by sampling goal measurements from the last trajectory of the episode using one of the following sampling strategies. Specify the sampling strategy using the `Strategy` property.

Sampling Strategy	`Strategy` Value	Description
Final	`"final"`	Sample one goal measurement from the final observation in the trajectory.
Episode	`"episode"`	Randomly sample multiple goal measurements from the entire trajectory.
Future	`"future"`	For a given time t, sample multiple goal measurements from the end of the trajectory starting at time t+1.

Notation

The hindsight experience generation algorithm described here uses the following notation.

- $\tau = (e_1, e_2, \ldots, e_T)$ is an experience trajectory, where T is the trajectory length.
- $e_t = (S_t, A_t, R_t, S_{t+1}, D_t)$ is experience at time t, where:

- S_t is the observation at time t.
- A_t is the action at time t.
- S_{t+1} is the observation at time t + 1.
- R_t is the reward at time t returned by the reward function $F_R(S_t, A_t, S_{t+1})$. To specify f_R, use the `RewardFunction` property.
- D_t is the episode terminal condition at time t returned by the is-done function $F_D(S_t, A_t, S_{t+1})$. To specify f_D, use the `IsDoneFunction` property.

- $S_t = (m_t, g_t, x_t)$ is the observation at time t, where:

  - $g_t = \left(g_{t,1}, \ldots, g_{t,N}\right)$ are goals.
  - $m_t = \left(m_{t,1}, \ldots, m_{t,N}\right)$ are corresponding goals measurements.
  - $x_t = \left(x_{t,1}, \ldots, x_{t,K}\right)$ are observations that are neither measurements nor goals.

Experience Generation Algorithm

At the end of each training episode, hindsight experiences are generated using the following algorithm. When you use a hindsight experience replay buffer in your own custom agent, you can generate experiences using the `generateHindsightExperiences` function.

```
1  Get the last trajectory $\tau$ from the experience buffer.
2  for $t$ = 1:$T$:
   a  Sample $M$ goal measurements $m_1$, ..., $m_M$ based on th sampling strategy.
      • Final strategy:
         • $M$ = 1
         • $m_1 = m_{T+1}$ from $S_{T+1}$ in $e_t$
      • Episode strategy:
         • $M$ = NumGoalSamples
         • Randomly sample goal measurements from $S_1$, ..., $S_{T+1}$ in $\tau M$. Do not sample any
           specific goal measurement more than once.
      • Future strategy:
         • $M$ = min($T$ – $t$ + 1,NumGoalSamples)
         • Randomly sample $M$ goal measurements from $S_{t+1}$, ..., $S_{T+1}$ in $\tau$. Do not sample any
           specific goal measurement more than once.
   b  Create $M$ empty trajectories $\tau_1$, ..., $\tau_M$.
   c  for $i$ = 1:$T$:
      i  In $e_t$, replace the goal $g_t$ in $S_t$ and $g_t$ in $S_{t+1}$ with $m_i$.
```

$$g'_t = m_i$$

$$g'_{t+1} = m_i$$

$$S'_t = (m_t, g'_t, x_t)$$

$$S'_{t+1} = (m_{t+1}, g'_{t+1}, x_{t+1})$$

ii Compute a new reward.

$$R'_t = f_R(S'_t, A_t, S'_{t+1})$$

iii Compute a new is-done signal.

$$D'_t = f_D(S'_t, A_t, S'_{t+1})$$

iv Create a new experience sample.

$$e'_{i,t} = (S'_t, A_t, R'_t, S'_{t+1}, D'_t)$$

v Append the new experience to trajectory τ_i.

d Append τ_1, ..., τ_M to the experience buffer.

Prioritized Sampling

Prioritized replay memory samples experiences according to experience priorities. For a given experience, the priority is defined as the absolute value of the associated temporal difference (TD) error. A larger TD error indicates that the critic network is not well-trained for the corresponding experience. Therefore, sampling such experiences during critic updates can help efficiently improve the critic performance, which often improves the sample efficiency of agent training.

When using prioritized replay memory, agents use the following process when sampling a mini-batch of experiences and updating a critic.

1 Compute the sampling probability P for each experience in the buffer based on the experience priority.

$$P(j) = \frac{p(j)^{\alpha}}{\sum_{i=1}^{N} p(i)^{\alpha}}$$

Here:

- N is the number of experiences in the replay memory buffer.
- p is the experience priority.
- α is a priority exponent. To set α, use the `PriorityExponent` parameter.

2 Sample a mini-batch of experiences according to the computed probabilities.

3 Compute the importance sampling weights (w) for the sampled experiences.

$$w'(j) = (N \cdot P(j))^{-\beta}$$

$$w(j) \leftarrow \frac{w'(j)}{\max\limits_{i \in \text{mini-batch}} w'(i)}$$

Here, β is the importance sampling exponent. The `ImportanceSamplingExponent` parameter contains the current value of β. To control β, set the `ImportanceSamplingExponent` and `NumAnnealingSteps` parameters.

4 Compute the weighted loss using the importance sampling weights w and the TD error δ to update a critic.

5 Update the priorities of the sampled experiences based on the TD error.

$$p(j) = |\delta|$$

6. Update the importance sampling exponent β by linearly annealing the exponent value until it reaches 1.

   $$\beta \leftarrow \beta + \frac{1 - \beta_0}{N_S}$$

   Here:

   - β_0 is the initial importance sampling exponent. To specify β_0, use the `InitialImportanceSamplingExponent` parameter.
   - N_S is the number of annealing steps. To specify N_s, use the `NumAnnealingSteps` parameter.

Version History

Introduced in R2023a

References

[1] Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver. 'Prioritized experience replay'. *arXiv:1511.05952 [Cs]* 25 February 2016. https://arxiv.org/abs/1511.05952.

[2] Andrychowicz, Marcin, Filip Wolski,Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojiech Zaremba. 'Hindsight experience replay'. *31st Conference on Neural Information Processing Systems*. Long Beach, CA, USA: 2017.

See Also

Objects
`rlReplayMemory` | `rlPrioritizedReplayMemory` | `rlHindsightReplayMemory`

rlHindsightReplayMemory

Hindsight replay memory experience buffer

Description

An off-policy reinforcement learning agent stores experiences in a circular experience buffer. During training, the agent samples mini-batches of experiences from the buffer and uses these mini-batches to update its actor and critic function approximators.

By default, built-in off-policy agents (DQN, DDPG, TD3, SAC, MBPO) use an `rlReplayMemory` object as their experience buffer. For goal-conditioned tasks, where the observation includes both the goal and a goal measurement, you can use an `rlHindsightReplayMemory` object.

A hindsight replay memory experience buffer:

- Generates additional experiences by replacing goals with goal measurements
- Improves sample efficiency for tasks with sparse rewards
- Requires a ground-truth reward function and is-done function
- Is not necessary when you have a well-shaped reward function

`rlHindsightReplayMemory` objects uniformly sample experiences from the buffer. To use prioritized nonuniform sampling, which can improve sample efficiency, use an `rlHindsightPrioritizedReplayMemory` object.

For more information on hindsight experience replay, see “Algorithms” on page 3-199.

Creation

Syntax

```
buffer = rlHindsightReplayMemory(obsInfo,actInfo,rewardFcn,isDoneFcn,
goalConditionInfo)
buffer = rlHindsightReplayMemory(obsInfo,actInfo,rewardFcn,isDoneFcn,
goalConditionInfo,maxLength)
```

Description

`buffer = rlHindsightReplayMemory(obsInfo,actInfo,rewardFcn,isDoneFcn,goalConditionInfo)` creates a hindsight replay memory experience buffer that is compatible with the observation and action specifications in `obsInfo` and `actInfo`, respectively. This syntax sets the `RewardFcn`, `IsDoneFcn`, and `GoalConditionInfo` properties.

`buffer = rlHindsightReplayMemory(obsInfo,actInfo,rewardFcn,isDoneFcn,goalConditionInfo,maxLength)` sets the maximum length of the buffer by setting the `MaxLength` property.

Input Arguments

`obsInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, specified as a reinforcement learning specification object or an array of specification objects defining properties such as dimensions, data types, and names of the observation signals.

You can extract the observation specifications from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`actInfo` — Action specifications
specification object | array of specification objects

Action specifications, specified as a reinforcement learning specification object defining properties such as dimensions, data types, and names of the action signals.

You can extract the action specifications from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

Properties

`MaxLength` — Maximum buffer length
`10000` (default) | nonnegative integer

This property is read-only.

Maximum buffer length, specified as a nonnegative integer.

To change the maximum buffer length, use the `resize` function.

`Length` — Number of experiences in buffer
`0` (default) | nonnegative integer

This property is read-only.

Number of experiences in buffer, specified as a nonnegative integer.

`GoalConditionInfo` — Goal condition information
1-by-*N* cell array

Goal condition information, specified as a 1-by-*N* cell array, where *N* is the number of goal conditions. For the *i*th goal condition, the corresponding cell of `GoalConditionInfo` contains a 1-by-4 cell array with the following elements.

- `GoalConditionInfo{i}{1}` — Goal measurement channel index.
- `GoalConditionInfo{i}{2}` — Goal measurement element indices.
- `GoalConditionInfo{i}{3}` — Goal channel index.
- `GoalConditionInfo{i}{4}` — Goal element indices.

The goal measurements in `GoalConditionInfo{i}{2}` correspond to the goals in `GoalConditionInfo{i}{4}`.

As an example, suppose that `obsInfo` contains specifications for two observation channels. Further, suppose that there is one goal condition where the goal measurements correspond to elements 2 and 3 of the first observation channel, and the goals correspond to elements 4 and 5 of the second observation channel. In this case, the goal condition information is:

```
GoalConditionInfo = {{1,[1 2],2,[4 5]}};
```

`RewardFcn` — Reward function

function handle

Reward function, specified as a handle to a function with the following signature.

```
function reward = myRewardFcn(obs,action,nextObs)
```

Here:

- `reward` is a scalar reward value.
- `obs` is the current observation.
- `act` is the action taken from the current observation.
- `nextObs` is the next observation after taking the specified action.

`IsDoneFcn` — Is-done function

function handle

Is-done function, specified as a handle to a function with the following signature.

```
function isdone = myIsDoneFcn(obs,action,nextObs)
```

Here:

- `isdone` is `true` when the next observation is a terminal condition and `false` otherwise.
- `obs` is the current observation.
- `act` is the action taken from the current observation.
- `nextObs` is the next observation after taking the specified action.

`Strategy` — Goal measurement sampling strategy

`"final"` (default) | `"episode"` | `"future"`

Goal measurement sampling strategy, specified as one of the following values.

- `"final"` — Use the goal measurement from the end of the trajectory.
- `"episode"` — Randomly sample M goal measurements from the trajectory, where M is equal to `NumGoalSamples`.
- `"future"` — Randomly sample M goal measurements from the trajectory, but create hindsight experiences for measurements that were observed at time t+1 or later.

`NumGoalSamples` — Number of goal measurements

4 (default) | positive integer

Number of goal measurements to sample when generating experiences, specified as a positive integer. This parameter is ignored when `Strategy` is `"final"`

Object Functions

append	Append experiences to replay memory buffer
sample	Sample experiences from replay memory buffer
resize	Resize replay memory experience buffer
reset	Reset environment, agent, experience buffer, or policy object
allExperiences	Return all experiences in replay memory buffer
validateExperience	Validate experiences for replay memory
generateHindsightExperiences	Generate hindsight experiences from hindsight experience replay buffer
getActionInfo	Obtain action data specifications from reinforcement learning environment, agent, or experience buffer
getObservationInfo	Obtain observation data specifications from reinforcement learning environment, agent, or experience buffer

Examples

Create DDPG Agent With Hindsight Replay Memory

For this example, create observation and action specifications directly. You can also extract such specifications from your environment.

Create an observation specification for an environment with a single a observation channel with six observations. For this example, assume that the observation channel contains the signals [a, x_m, y_m, x_g, y_g, c], where:

- x_g and y_g are the goal observations.
- x_m and y_m are the goal measurements.
- a and c are additional observations.

```
obsInfo = rlNumericSpec([6 1]);
```

Create a specification for a single action.

```
actInfo = rlNumericSpec([1 1]);
```

Create a DDPG agent from the environment specifications. By default, the agent uses a replay memory experience buffer with uniform sampling.

```
agent = rlDDPGAgent(obsInfo,actInfo);
```

To create a hindsight replay memory buffer, first define the goal condition information. Both the goals and goal measurements are in the single observation channel. The goal measurements are in elements 2 and 3 of the observation channel and the goals are in elements 4 and 5 of the observation channel.

```
goalConditionInfo = {{1,[2 3],1,[4 5]}};
```

Define an is-done function. For this example, the is-done signal is `true` when the next observation satisfies the goal condition $\sqrt{(x_m - x_g)^2 + (y_m - y_g)^2} < 0.1$.

```
function isdone = hindsightIsDoneFcn1(Observation,Action,NextObservation)
    NextObservation = NextObservation{1};
    xm = NextObservation(2);
    ym = NextObservation(3);
    xg = NextObservation(4);
    yg = NextObservation(5);
    isdone = sqrt((xm-xg)^2 + (ym-yg)^2) < 0.1;
end
```

Define a reward function. For this example, the reward is `1` when the is-done signal is `true` and –`0.01` otherwise.

```
function reward = hindsightRewardFcn1(Observation,Action,NextObservation)
    isdone = hindsightIsDoneFcn1(Observation,Action,NextObservation);
    if isdone
        reward = 1;
    else
        reward = -0.01;
    end
end
```

Create a hindsight replay memory buffer with a default maximum length.

```
buffer = rlHindsightReplayMemory(obsInfo,actInfo, ...
    @hindsightRewardFcn1,@hindsightIsDoneFcn1,goalConditionInfo);
```

Replace the default experience buffer with the hindsight replay memory buffer.

```
agent.ExperienceBuffer = buffer;
```

Create Hindsight Replay Memory With Two Observation Channels

For this example, create observation and action specifications directly. You can also extract such specifications from your environment.

Create observation specification for an environment with a two observation channels. For this example, assume that the first observation channel contains the signals [a, x_m, y_m] and the second observation channel contains the signals [x_g, y_g, c], where:

- x_g and y_g are the goal observations.
- x_m and y_m are the goal measurements.
- a and c are additional observations.

```
obsInfo = [rlNumericSpec([3 1]), rlNumericSpec([3 1])];
```

Create a specification for a single action.

```
actInfo = rlNumericSpec([1 1]);
```

To create a hindsight replay memory buffer, first define the goal condition information. The goal measurements are in elements 2 and 3 of the first observation channel and the goals are in elements 1 and 2 of the second observation channel.

```
goalConditionInfo = {{1,[2 3],2,[1 2]}};
```

Define an is-done function. For this example, the is-done signal is `true` when the next observation satisfies the goal condition $\sqrt{(x_m - x_g)^2 + (y_m - y_g)^2} < 0.1$.

```
function isdone = hindsightIsDoneFcn2(Observation,Action,NextObservation)
    NextObsCh1 = NextObservation{1};
    NextObsCh2 = NextObservation{2};
    xm = NextObsCh1(2);
    ym = NextObsCh1(3);
    xg = NextObsCh2(1);
    yg = NextObsCh2(2);
    isdone = sqrt((xm-xg)^2 + (ym-yg)^2) < 0.1;
end
```

Define a reward function. For this example, the reward is `1` when the is-done signal is `true` and `0` otherwise.

```
function reward = hindsightRewardFcn2(Observation,Action,NextObservation)
    isdone = hindsightIsDoneFcn1(Observation,Action,NextObservation);
    if isdone
        reward = 1;
    else
        reward = 0;
    end
end
```

Create a hindsight replay memory buffer with a maximum length of `20000`.

```
buffer = rlHindsightReplayMemory(obsInfo,actInfo, ...
    @hindsightRewardFcn2,@hindsightIsDoneFcn2,...
    goalConditionInfo,20000);
```

Create Hindsight Replay Memory With Two Goal Conditions

For this example, create observation and action specifications directly. You can also extract such specifications from your environment.

Create an observation specification for an environment with a single a observation channel with eight observations. For this example, assume that the observation channel contains the signals [a, x_m, y_m, θ, x_g, y_g, θ_m,c], where:

- x_g, y_g, and θ are the goal observations.
- x_m, y_m, and θ_m are the goal measurements.
- a and c are additional observations.

```
obsInfo = rlNumericSpec([8 1]);
```

Create a specification for a single action.

```
actInfo = rlNumericSpec([1 1]);
```

To create a hindsight replay memory buffer, first define the goal condition information. For this example, define two goal conditions.

The first goal condition depends on x_m, y_m, x_g, and y_g as shown in the following equation.

$$\sqrt{\left(x_m - x_g\right)^2 + \left(y_m - y_g\right)^2} < 0.1$$

Specify the information for this goal condition.

```
goalConditionInfo1 = {1,[2 3], 1, [5 6]};
```

The first goal condition depends on θ_m and θ as shown in the following equation.

$$(\theta_m - \theta)^2 < 0.01$$

Specify the information for this goal condition.

```
goalConditionInfo2 = {1,4,1,7};
```

Combine the goal condition information into a cell array.

```
goalConditionInfo = {goalConditionInfo1, goalConditionInfo2};
```

Define an is-done function that returns true when the next observation satisfies both goal conditions.

```
function isdone = hindsightIsDoneFcn3(Observation,Action,NextObservation)
    NextObservation = NextObservation{1};
    xm = NextObservation(2);
    ym = NextObservation(3);
    xg = NextObservation(5);
    yg = NextObservation(6);
    thetam = NextObservation(7);
    theta = NextObservation(4);
    isdone = sqrt((xm-xg)^2 + (ym-yg)^2) < 0.1 ...
      && (thetam-theta)^2 < 0.01;
end
```

Define a reward function. For this example, the reward is `1` when the is-done signal is `true` and –`0.01` otherwise.

```
function reward = hindsightRewardFcn3(Observation,Action,NextObservation)
    isdone = hindsightIsDoneFcn3(Observation,Action,NextObservation);
    if isdone
        reward = 1;
    else
        reward = -0.01;
    end
end
```

Create a hindsight replay memory buffer with a default maximum length.

```
buffer = rlHindsightReplayMemory(obsInfo,actInfo, ...
    @hindsightRewardFcn3,@hindsightIsDoneFcn3, ...
    goalConditionInfo);
```

Generate Experiences from Hindsight Replay Memory

When you use a hindsight replay memory buffer within your custom agent training loop, you generate experiences at the end of training episode.

Create an observation specification for an environment with a single a observation channel with six observations. For this example, assume that the observation channel contains the signals [a, x_m, y_m, x_g, y_g, c], where:

- x_g and y_g are the goal observations.
- x_m and y_m are the goal measurements.
- a and c are additional observations.

```
obsInfo = rlNumericSpec([6 1],...
    LowerLimit=0,UpperLimit=[1;5;5;5;5;1]);
```

Create a specification for a single action.

```
actInfo = rlNumericSpec([1 1],...
    LowerLimit=0,UpperLimit=10);
```

To create a hindsight replay memory buffer, first define the goal condition information. Both the goals and goal measurements are in the single observation channel. The goal measurements are in elements 2 and 3 of the observation channel and the goals are in elements 4 and 5 of the observation channel.

```
goalConditionInfo = {{1,[2 3],1,[4 5]}};
```

For this example, use `hindsightRewardFcn1` as the ground-truth reward function and `hindsightIsDoneFcn1` as the termination condition function.

Create the hindsight replay memory buffer.

```
buffer = rlHindsightReplayMemory(obsInfo,actInfo, ...
    @hindsightRewardFcn1,@hindsightIsDoneFcn1,goalConditionInfo);
```

As you train your agent, you add experience trajectories to the experience buffer. For this example, add a random experience trajectory of length 10.

```
for i = 1:10
    exp(i).Observation = {obsInfo.UpperLimit.*rand(6,1)};
    exp(i).Action = {actInfo.UpperLimit.*rand(1)};
    exp(i).NextObservation = {obsInfo.UpperLimit.*rand(6,1)};
    exp(i).Reward = 10*rand(1);
    exp(i).IsDone = 0;
end
exp(10).IsDone = 1;

append(buffer,exp);
```

At the end of the training episode, you generate hindsight experiences from the last trajectory added to the buffer. Generate experiences specifying the length of the last trajectory added to the buffer.

```
newExp = generateHindsightExperiences(buffer,10);
```

For each experience in the final trajectory, the default "final" sampling strategy generates a new experience where it replaces the goals in Observation and NextObservation with the goal measurements from the final experience in the trajectory.

To validate this behavior, first view the final goal measurements from exp.

```
exp(10).NextObservation{1}(2:3)
```

```
ans = 2×1

    0.7277
    0.6803
```

Next, view the goal values for one of the generated experiences. This value should match the final goal measurement

```
newExp(6).Observation{1}(4:5)
```

```
ans = 2×1

    0.7277
    0.6803
```

After generating the new experiences, append them to the buffer.

```
append(buffer,newExp);
```

Limitations

- Hindsight experience replay does not support agents that use recurrent neural networks.

Algorithms

Hindsight Experience Replay

Hindsight experience replay is a data augmentation method that you can use for goal-conditioned tasks, where the observation includes both the goal and a goal measurement. In such a goal-conditioned task, the agent reaches the goal when the distance between the goal measurement and the goal is less than a threshold.

However, when the reward is sparse, the agent may not be able to reach the goal during training. In this case, the agent does not get any rewards. Hindsight experience replay generates experience data by replacing the goal with a goal measurement. As a result, the agent can get rewards for improving the goal measurement without actually reaching the goal.

The generated hindsight experiences are created at the end of each training episode by sampling goal measurements from the last trajectory of the episode using one of the following sampling strategies. Specify the sampling strategy using the Strategy property.

Sampling Strategy	Strategy Value	Description
Final	`"final"`	Sample one goal measurement from the final observation in the trajectory.
Episode	`"episode"`	Randomly sample multiple goal measurements from the entire trajectory.
Future	`"future"`	For a given time *t*, sample multiple goal measurements from the end of the trajectory starting at time *t*+1.

Notation

The hindsight experience generation algorithm described here uses the following notation.

- $\tau = (e_1, e_2, \ldots, e_T)$ is an experience trajectory, where T is the trajectory length.
- $e_t = (S_t, A_t, R_t, S_{t+1}, D_t)$ is experience at time t, where:
  - S_t is the observation at time t.
  - A_t is the action at time t.
  - S_{t+1} is the observation at time $t + 1$.
  - R_t is the reward at time t returned by the reward function $F_R(S_t, A_t, S_{t+1})$. To specify f_R, use the `RewardFunction` property.
  - D_t is the episode terminal condition at time t returned by the is-done function $F_D(S_t, A_t, S_{t+1})$. To specify f_D, use the `IsDoneFunction` property.
- $S_t = (m_t, g_t, x_t)$ is the observation at time t, where:
  - $g_t = (g_{t,1}, \ldots, g_{t,N})$ are goals.
  - $m_t = (m_{t,1}, \ldots, m_{t,N})$ are corresponding goals measurements.
  - $x_t = (x_{t,1}, \ldots, x_{t,K})$ are observations that are neither measurements nor goals.

Experience Generation Algorithm

At the end of each training episode, hindsight experiences are generated using the following algorithm. When you use a hindsight experience replay buffer in your own custom agent, you can generate experiences using the `generateHindsightExperiences` function.

1. Get the last trajectory τ from the experience buffer.
2. for t = 1:T:
   a. Sample M goal measurements $m_1, \ldots, m_M$ based on th sampling strategy.
      - Final strategy:
        - $M = 1$
        - $m_1 = m_{T+1}$ from S_{T+1} in e_t
      - Episode strategy:
        - M = `NumGoalSamples`

  - Randomly sample goal measurements from S_1, ..., S_{T+1} in τM. Do not sample any specific goal measurement more than once.

- Future strategy:

  - M = min(T – t + 1,`NumGoalSamples`)
  - Randomly sample M goal measurements from S_{t+1}, ..., S_{T+1} in τ. Do not sample any specific goal measurement more than once.

b Create M empty trajectories τ_1, ..., τ_M.

c for i = 1:T:

i In e_t, replace the goal g_t in S_t and g_t in S_{t+1} with m_i.

$$g'_t = m_i$$

$$g'_{t+1} = m_i$$

$$S'_t = (m_t, g'_t, x_t)$$

$$S'_{t+1} = (m_{t+1}, g'_{t+1}, x_{t+1})$$

ii Compute a new reward.

$$R'_t = f_R(S'_t, A_t, S'_{t+1})$$

iii Compute a new is-done signal.

$$D'_t = f_D(S'_t, A_t, S'_{t+1})$$

iv Create a new experience sample.

$$e'_{i,t} = (S'_t, A_t, R'_t, S'_{t+1}, D'_t)$$

v Append the new experience to trajectory τ_i.

d Append τ_1, ..., τ_M to the experience buffer.

Version History

Introduced in R2023a

References

[1] Andrychowicz, Marcin, Filip Wolski,Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojiech Zaremba. 'Hindsight experience replay'. *31st Conference on Neural Information Processing Systems*. Long Beach, CA, USA: 2017.

See Also

Objects
`rlReplayMemory` | `rlPrioritizedReplayMemory` | `rlHindsightPrioritizedReplayMemory`

Topics
“Train DQN Agent Using Hindsight Experience Replay”

rlIsDoneFunction

Is-done function approximator object for neural network-based environment

Description

When creating a neural network-based environment using `rlNeuralNetworkEnvironment`, you can specify the is-done function approximator using an `rlIsDoneFunction` object. Do so when you do not know a ground-truth termination signal for your environment.

The is-done function approximator object uses a deep neural network as internal approximation model to predict the termination signal for the environment given one of the following input combinations.

- Observations, actions, and next observations
- Observations and actions
- Actions and next observations
- Next observations

Creation

Syntax

```
isdFcnAppx = rlIsDoneFunction(net,observationInfo,actionInfo,Name=Value)
```

Description

`isdFcnAppx = rlIsDoneFunction(net,observationInfo,actionInfo,Name=Value)` creates the is-done function approximator object `isdFcnAppx` using the deep neural network `net` and sets the `ObservationInfo` and `ActionInfo` properties.

When creating an is-done function approximator you must specify the names of the deep neural network inputs using one of the following combinations of name-value pair arguments.

- `ObservationInputNames`, `ActionInputNames`, and `NextObservationInputNames`
- `ObservationInputNames` and `ActionInputNames`
- `ActionInputNames` and `NextObservationInputNames`
- `NextObservationInputNames`

You can also specify the `UseDeterministicPredict` and `UseDevice` properties using optional name-value pair arguments. For example, to use a GPU for prediction, specify `UseDevice="gpu"`.

Input Arguments

net — Deep neural network
`dlnetwork` object

Deep neural network with a scalar output value, specified as a `dlnetwork` object.

The input layer names for this network must match the input names specified using the ObservationInputNames, ActionInputNames, and NextObservationInputNames. The dimensions of the input layers must match the dimensions of the corresponding observation and action specifications in ObservationInfo and ActionInfo, respectively.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: ObservationInputNames="velocity"

ObservationInputNames — Observation input layer names

string | string array

Observation input layer names, specified as a string or string array. Specify ObservationInputNames when you expect the termination signal to depend on the current environment observation.

The number of observation input names must match the length of ObservationInfo and the order of the names must match the order of the specifications in ObservationInfo.

ActionInputNames — Action input layer names

string | string array

Action input layer names, specified as a string or string array. Specify ActionInputNames when you expect the termination signal to depend on the current action value.

The number of action input names must match the length of ActionInfo and the order of the names must match the order of the specifications in ActionInfo.

NextObservationInputNames — Next observation input layer names

string | string array

Next observation input layer names, specified as a string or string array. Specify NextObservationInputNames when you expect the termination signal to depend on the next environment observation.

The number of next observation input names must match the length of ObservationInfo and the order of the names must match the order of the specifications in ObservationInfo.

Properties

ObservationInfo — Observation specifications

rlNumericSpec object | array rlNumericSpec objects

This property is read-only.

Observation specifications, specified as an rlNumericSpec object or an array of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

You can extract the observation specifications from an existing environment or agent using getObservationInfo. You can also construct the specifications manually using rlNumericSpec.

ActionInfo — Action specifications
rlFiniteSetSpec object | rlNumericSpec object

This property is read-only.

Action specifications, specified as an rlFiniteSetSpec or rlNumericSpec object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

You can extract the action specifications from an existing environment or agent using getActionInfo. You can also construct the specification manually using rlFiniteSetSpec or rlNumericSpec.

UseDeterministicPredict — Option to predict the terminal signal deterministically
true (default) | false

Option to predict the terminal signal deterministically, specified as one of the following values.

- true — Use deterministic network prediction.
- false — Use stochastic network prediction.

UseDevice — Computation device used for training and simulation
"cpu" (default) | "gpu"

Computation device used to perform operations such as gradient computation, parameter updates, and prediction during training and simulation, specified as either "cpu" or "gpu".

The "gpu" option requires both Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use gpuDevice (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating a network on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

Object Functions

rlNeuralNetworkEnvironment	Environment model with deep neural network transition models

Examples

Create Is-Done Function and Predict Termination

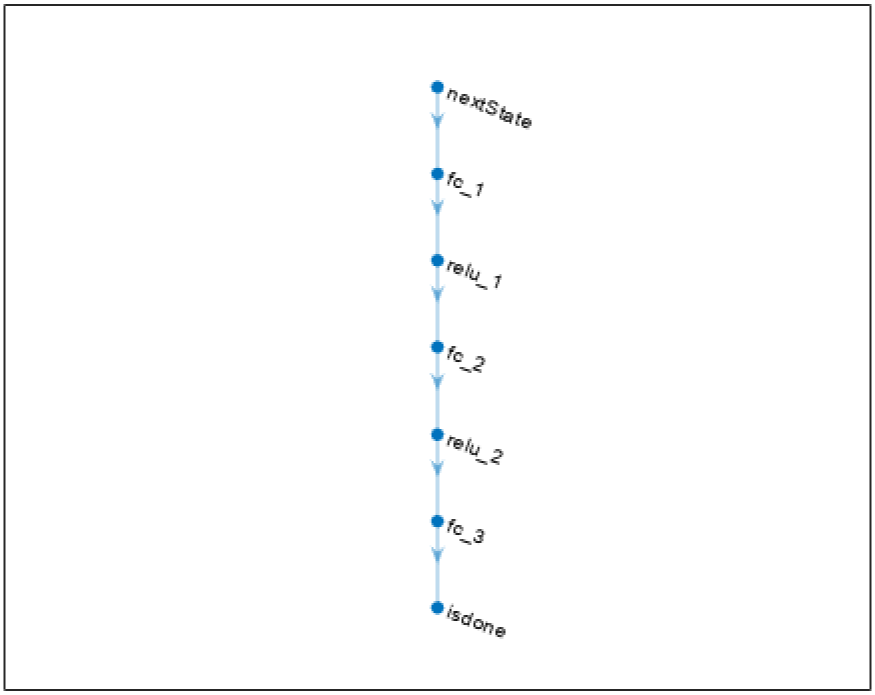
Create an environment interface and extract observation and action specifications. Alternatively, you can create specifications using rlNumericSpec and rlFiniteSetSpec.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

To approximate the is-done function, use a deep neural network. The network has one input channel for the next observations. The single output channel is for the predicted termination signal.

Create the neural network as a vector of layer objects.

```
commonPath = [
    featureInputLayer( ...
                obsInfo.Dimension(1), ...
                Name="nextState")
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(2)
    softmaxLayer(Name="isdone")];

net = layerGraph(commonPath);

plot(net)
```



Covert the network to a `dlnetwork` object and display the number of weights.

```
net = dlnetwork(net);
summary(net);
```

```
   Initialized: true

   Number of learnables: 4.6k

   Inputs:
      1   'nextState'   4 features
```

Create an is-done function approximator object.

```
isDoneFcnAppx = rlIsDoneFunction(...
    net,obsInfo,actInfo,...
    NextObservationInputNames="nextState");
```

Using this is-done function approximator object, you can predict the termination signal based on the next observation. For example, predict the termination signal for a random next observation. Since for this example the termination signal only depends on the next observation, use empty cell arrays for the current action and observation inputs.

```
nxtobs = rand(obsInfo.Dimension);
predIsDone = predict(isDoneFcnAppx,{},{},{nxtobs})
```

```
predIsDone = 0
```

You can obtain the termination probability using `evaluate`.

```
predIsDoneProb = evaluate(isDoneFcnAppx,{nxtobs})
```

```
predIsDoneProb = 1x1 cell array
    {2x1 single}
```

```
predIsDoneProb{1}
```

```
ans = 2x1 single column vector

    0.5405
    0.4595
```

The first number is the probability of obtaining a `0` (no termination predicted), the second one is the probability of obtaining a `1` (termination predicted).

Version History

Introduced in R2022a

See Also

Functions

`evaluate` | `gradient` | `accelerate`

Objects

`rlNeuralNetworkEnvironment` | `rlContinuousDeterministicTransitionFunction` | `rlContinuousGaussianTransitionFunction` | `rlContinuousDeterministicRewardFunction` | `rlContinuousGaussianRewardFunction` | `rlMBPOAgent` | `rlMBPOAgentOptions`

Topics

“Train MBPO Agent to Balance Cart-Pole System”
“Model-Based Policy Optimization (MBPO) Agents”

rlMaxQPolicy

Policy object to generate discrete max-Q actions for custom training loops and application deployment

Description

This object implements a max-Q policy, which returns the action that maximizes a discrete action-space Q-value function, given an input observation. You can create an `rlMaxQPolicy` object from an `rlQValueFunction` or `rlVectorQValueFunction` object, or extract it from an `rlQAgent`, `rlDQNAgent` or `rlSARSAAgent`. You can then train the policy object using a custom training loop or deploy it for your application using `generatePolicyBlock` or `generatePolicyFunction`. This policy is always deterministic and does not perform any exploration. For more information on policies and value functions, see “Create Policies and Value Functions”.

Creation

Syntax

```
policy = rlMaxQPolicy(qValueFunction)
```

Description

`policy = rlMaxQPolicy(qValueFunction)` creates the max-Q policy object `policy` from the discrete action-space Q-value function `qValueFunction`. It also sets the `QValueFunction` property of `policy` to the input argument `qValueFunction`.

Properties

`QValueFunction` — Discrete action-space Q-value function
`rlQValueFunction` object

Discrete action-space Q-value function approximator, specified as an `rlQValueFunction` or `rlVectorQValueFunction` object.

`ObservationInfo` — Observation specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array of such objects. These objects define properties such as the dimensions, data types, and names of the observation channels.

`ActionInfo` — Action specifications
`rlFiniteSetSpec` object

Action specifications, specified as an `rlFiniteSetSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

SampleTime — Sample time of policy

positive scalar | -1 (default)

Sample time of policy, specified as a positive scalar or as -1 (default). Setting this parameter to -1 allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the policy is specified executes every SampleTime seconds of simulation time. If SampleTime is -1, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the policy is executed every time the environment advances. In this case, SampleTime is the time interval between consecutive elements in the output experience. If SampleTime is -1, the sample time is treated as being equal to 1.

Example: 0.2

Object Functions

generatePolicyBlock	Generate Simulink block that evaluates policy of an agent or policy object
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object
getAction	Obtain action from agent, actor, or policy object given environment observations
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
reset	Reset environment, agent, experience buffer, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object

Examples

Create Max-Q Policy Object from Vector Q-Value Function

Create observation and action specification objects. For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing four doubles, and the action space as a finite set consisting of two possible values, -1 and 1.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlFiniteSetSpec([-1 1]);
```

Alternatively, you can use getObservationInfo and getActionInfo to extract the specification objects from an environment.

Create a vector Q-value function approximator to use as critic. A vector Q-value function must accept an observation as input and return a single vector with as many elements as the number of possible discrete actions.

To model the parametrized vector Q-value function within the critic, use a neural network. Define a single path from the network input to its output as an array of layer objects.

```
layers = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(10)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
model = dlnetwork(layers);
summary(model)
```

```
   Initialized: true

   Number of learnables: 72

   Inputs:
      1   'input'   4 features
```

Create a vector Q-value function using `model`, and the observation and action specifications.

```
qValueFcn = rlVectorQValueFunction(model,obsInfo,actInfo)
```

```
qValueFcn = 
  rlVectorQValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
          UseDevice: "cpu"
```

Check the critic with a random observation input.

```
getValue(qValueFcn,{rand(obsInfo.Dimension)})
```

```
ans = 2x1 single column vector

    0.6486
   -0.3103
```

Create a policy object from `qValueFcn`.

```
policy = rlMaxQPolicy(qValueFcn)
```

```
policy = 
  rlMaxQPolicy with properties:

     QValueFunction: [1x1 rl.function.rlVectorQValueFunction]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
         SampleTime: -1
```

Check the policy with a random observation input.

```
getAction(policy,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-1]}
```

You can now train the policy with a custom training loop and then deploy it to your application.

Version History

Introduced in R2022a

See Also

Functions

getGreedyPolicy | getExplorationPolicy | generatePolicyBlock | generatePolicyFunction | getAction | getLearnableParameters | setLearnableParameters

Objects

rlEpsilonGreedyPolicy | rlDeterministicActorPolicy | rlAdditiveNoisePolicy | rlStochasticActorPolicy | rlQValueFunction | rlVectorQValueFunction | rlSARSAAgent | rlQAgent | rlDQNAgent

Blocks

Policy | RL Agent

Topics

“Model-Based Reinforcement Learning Using Custom Training Loop”
“Train Reinforcement Learning Policy Using Custom Training Loop”
“Create Policies and Value Functions”

rlMBPOAgent

Model-based policy optimization (MBPO) reinforcement learning agent

Description

A model-based policy optimization (MBPO) agent is a model-based, online, off-policy, reinforcement learning method. An MBPO agent contains an internal model of the environment, which it uses to generate additional experiences without interacting with the environment.

During training, the MBPO agent generates real experiences by interacting with the environment. These experiences are used to train the internal environment model, which is used to generate additional experiences. The training algorithm then uses both the real and generated experiences to update the agent policy.

Note MBPO agents do not support recurrent networks.

Creation

Syntax

```
agent = rlMBPOAgent(baseAgent,envModel)
agent = rlMBPOAgent( ___ ,agentOptions)
```

Description

`agent = rlMBPOAgent(baseAgent,envModel)` creates a model-based policy optimization agent with default options and sets the `BaseAgent` and `EnvModel` properties.

`agent = rlMBPOAgent( ___ ,agentOptions)` creates a model-based policy optimization agent using specified options and sets the `AgentOptions` property.

Properties

`BaseAgent` — Base reinforcement learning agent
`rlDQNAgent` | `rlDDPGAgent` | `rlTD3Agent` | `rlSACAgent`

Base reinforcement learning agent, specified as an off-policy agent object.

For environments with a discrete action space, specify a DQN agent using an `rlDQNAgent` object.

For environments with a continuous action space, use one of the following agent objects.

- `rlDDPGAgent` — DDPG agent
- `rlTD3Agent` — TD3 agent
- `rlSACAgent` — SAC agent

EnvModel — Environment model
rlNeuralNetworkEnvironment

Environment model, specified as an `rlNeuralNetworkEnvironment` object. This environment contains transition functions, a reward function, and an is-done function.

AgentOptions — Agent options
rlMBPOAgentOptions object

Agent options, specified as an `rlMBPOAgentOptions` object.

RolloutHorizon — Current roll-out horizon value
positive integer

Current roll-out horizon value, specified as a positive integer. For more information on setting the initial horizon value and the horizon update method, see `rlMBPOAgentOptions`.

ModelExperienceBuffer — Model experience buffer
rlReplayMemory object

Model experience buffer, specified as an `rlReplayMemory` object. During training the agent stores each of its generated experiences (S,A,R,S',D) in a buffer. Here:

- S is the current observation of the environment.
- A is the action taken by the agent.
- R is the reward for taking action A.
- S' is the next observation after taking action A.
- D is the is-done signal after taking action A.

UseExplorationPolicy — Option to use exploration policy
true | false

Option to use exploration policy when selecting actions, specified as one of the following logical values.

- `true` — Use the base agent exploration policy when selecting actions.
- `false` — Use the base agent greedy policy when selecting actions.

The initial value of `UseExplorationPolicy` matches the value specified in `BaseAgent`. If you change the value of `UseExplorationPolicy` in either the base agent or the MBPO agent, the same value is used for the other agent.

ObservationInfo — Observation specifications
specification object | array of specification objects

This property is read-only.

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

The value of `ObservationInfo` matches the corresponding value specified in `BaseAgent`.

ActionInfo — Action specification
rlFiniteSetSpec object | rlNumericSpec object

This property is read-only.

Action specifications, specified either as an rlFiniteSetSpec (for discrete action spaces) or rlNumericSpec (for continuous action spaces) object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

The value of ActionInfo matches the corresponding value specified in BaseAgent.

SampleTime — Sample time of agent
positive scalar | -1

Sample time of agent, specified as a positive scalar or as -1. Setting this parameter to -1 allows for event-based simulations.

The initial value of SampleTime matches the value specified in BaseAgent. If you change the value of SampleTime in either the base agent or the MBPO agent, the same value is used for the other agent.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every SampleTime seconds of simulation time. If SampleTime is -1, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, SampleTime is the time interval between consecutive elements in the output experience returned by sim or train. If SampleTime is -1, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment

Examples

Create MBPO Agent

Create an environment interface and extract observation and action specifications.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a base off-policy agent. For this example, use a SAC agent.

```
agentOpts = rlSACAgentOptions;
agentOpts.MiniBatchSize = 256;
initOpts = rlAgentInitializationOptions(NumHiddenUnit=64);
baseagent = rlSACAgent(obsInfo,actInfo,initOpts,agentOpts);
```

Check your agent with a random input observation.

```
getAction(baseagent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-7.2875]}
```

The neural network environment uses a function approximator object to approximate the environment transition function. The function approximator object uses one or more neural networks as approximator model. To account for modeling uncertainty, you can specify multiple transition models. For this example, create a single transition model.

Create a neural network to use as approximation model within the transition function object. Define each network path as an array of layer objects. Specify a name for the input and output layers, so you can later explicitly associate them with the appropriate channel.

```
% Observation and action paths
obsPath = featureInputLayer(obsInfo.Dimension(1),Name="obsIn");
actionPath = featureInputLayer(actInfo.Dimension(1),Name="actIn");

% Common path: concatenate along dimension 1
commonPath = [concatenationLayer(1,2,Name="concat")
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(obsInfo.Dimension(1),Name="nextObsOut")];

% Add layers to layerGraph object
transNet = layerGraph(obsPath);
transNet = addLayers(transNet,actionPath);
transNet = addLayers(transNet,commonPath);

% Connect layers
transNet = connectLayers(transNet,"obsIn","concat/in1");
transNet = connectLayers(transNet,"actIn","concat/in2");

% Convert to dlnetwork object
transNet = dlnetwork(transNet);

% Display number of weights
summary(transNet)
```

```
   Initialized: true

   Number of learnables: 4.8k

   Inputs:
      1   'obsIn'   4 features
      2   'actIn'   1 features
```

Create the transition function approximator object.

```
transitionFcnAppx = rlContinuousDeterministicTransitionFunction( ...
    transNet,obsInfo,actInfo,...
    ObservationInputNames="obsIn",...
    ActionInputNames="actIn",...
    NextObservationOutputNames="nextObsOut");
```

Create a neural network to use as a reward model for the reward function approximator object.

```
% Observation and action paths
actionPath = featureInputLayer(actInfo.Dimension(1),Name="actIn");
nextObsPath = featureInputLayer(obsInfo.Dimension(1),Name="nextObsIn");

% Common path: concatenate along dimension 1
commonPath = [concatenationLayer(1,2,Name="concat")
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(1)];

% Add layers to layerGraph object
rewardNet = layerGraph(nextObsPath);
rewardNet = addLayers(rewardNet,actionPath);
rewardNet = addLayers(rewardNet,commonPath);

% Connect layers
rewardNet = connectLayers(rewardNet,"nextObsIn","concat/in1");
rewardNet = connectLayers(rewardNet,"actIn","concat/in2");

% Convert to dlnetwork object
rewardNet = dlnetwork(rewardNet);

% Display number of weights
summary(transNet)
```

```
   Initialized: true

   Number of learnables: 4.8k

   Inputs:
      1   'obsIn'   4 features
      2   'actIn'   1 features
```

Create the reward function approximator object.

```
rewardFcnAppx = rlContinuousDeterministicRewardFunction( ...
    rewardNet,obsInfo,actInfo, ...
    ActionInputNames="actIn",...
    NextObservationInputNames="nextObsIn");
```

Create an is-done model for the reward function approximator object.

```
% Define main path
net = [featureInputLayer(obsInfo.Dimension(1),Name="nextObsIn");
fullyConnectedLayer(64)
reluLayer
fullyConnectedLayer(64)
reluLayer
fullyConnectedLayer(2)
softmaxLayer(Name="isdoneOut")];

% Convert to layergraph object
```

```
isDoneNet = layerGraph(net);

% Convert to dlnetwork object
isDoneNet = dlnetwork(isDoneNet);

% Display number of weights
summary(transNet)
```

```
   Initialized: true

   Number of learnables: 4.8k

   Inputs:
      1   'obsIn'    4 features
      2   'actIn'    1 features
```

Create the reward function approximator object.

```
isdoneFcnAppx = rlIsDoneFunction(isDoneNet,obsInfo,actInfo, ...
    NextObservationInputNames="nextObsIn");
```

Create the neural network environment using the observation and action specifications and the three function approximator objects.

```
generativeEnv = rlNeuralNetworkEnvironment( ...
    obsInfo,actInfo,...
    transitionFcnAppx,rewardFcnAppx,isdoneFcnAppx);
```

Specify options for creating an MBPO agent. Specify the optimizer options for the transition network and use default values for all other options.

```
MBPOAgentOpts = rlMBPOAgentOptions;
MBPOAgentOpts.TransitionOptimizerOptions = rlOptimizerOptions(...
    LearnRate=1e-4,...
    GradientThreshold=1.0);
```

Create the MBPO agent.

```
agent = rlMBPOAgent(baseagent,generativeEnv,MBPOAgentOpts);
```

Check your agent with a random input observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[7.8658]}
```

Version History

Introduced in R2022a

See Also

Objects

`rlMBPOAgentOptions` | `rlNeuralNetworkEnvironment` | `rlDDPGAgent` | `rlSACAgent` | `rlPPOAgent`

Topics

"Train MBPO Agent to Balance Cart-Pole System"
"Model-Based Policy Optimization (MBPO) Agents"
"Reinforcement Learning Agents"

rlMBPOAgentOptions

Options for MBPO agent

Description

Use an `rlMBPOAgentOptions` object to specify options for model-based policy optimization (MBPO) agents. To create an MBPO agent, use `rlMBPOAgent`.

For more information, see “Model-Based Policy Optimization (MBPO) Agents”.

Creation

Syntax

```
opt = rlMBPOAgentOptions
opt = rlMBPOAgentOptions(Name=Value)
```

Description

`opt = rlMBPOAgentOptions` creates an option object for use as an argument when creating an MBPO agent using all default options. You can modify the object properties using dot notation.

`opt = rlMBPOAgentOptions(Name=Value)` sets option properties on page 3-219 using name-value pair arguments. For example, `rlMBPOAgentOptions(DiscountFactor=0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value pair arguments.

Properties

`NumEpochForTrainingModel` — Number of epochs
`5` (default) | positive integer

Number of epochs for training the environment model, specified as a positive integer.

`NumMiniBatches` — Number of mini-batches
`10` (default) | positive integer | `"all"`

Number of mini-batches used in each environment model training epoch, specified as a positive scalar or `"all"`. When you specify `NumMiniBatches` to `"all"`, the agent selects the number of mini-batches such that all samples in the base agents experience buffer are used to train the model.

`MiniBatchSize` — Size of random experience mini-batch
`128` (default) | positive integer

Size of random experience mini-batch for training environment model, specified as a positive integer. During each model training episode, the agent randomly samples experiences from the experience buffer when computing gradients for updating the environment model properties. Large mini-batches reduce the variance when computing gradients but increase the computational effort.

ModelExperienceBufferLength — Generated experience buffer size

100000 (default) | positive integer

Generated experience buffer size, specified as a positive integer. When the agent generates experiences, they are added to the model experience buffer.

RealSampleRatio — Ratio of real experiences in a mini-batch

0.2 (default) | nonnegative scalar less than or equal to 1

Ratio of real experiences in a mini-batch for agent training, specified as a nonnegative scalar less than or equal to 1.

TransitionOptimizerOptions — Transition function optimizer options

rlOptimizerOptions object | array of rlOptimizerOptions objects

Transition function optimizer options, specified as one of the following:

- rlOptimizerOptions object — When your neural network environment has a single transition function or if you want to use the same options for multiple transition functions, specify a single options object.
- Array of rlOptimizerOptions objects — When your neural network environment agent has multiple transition functions and you want to use different optimizer options for the transition functions, specify an array of options objects with length equal to the number of transition functions.

Using these objects, you can specify training parameters for the transition deep neural network approximators as well as the optimizer algorithms and parameters.

If you have previously trained transition models and do not want the MBPO agent to modify these models during training, set TransitionOptimizerOptions.LearnRate to 0.

RewardOptimizerOptions — Reward function optimizer options

rlOptimizerOptions object

Reward function optimizer options, specified as an rlOptimizerOptions object. Using this object, you can specify training parameters for the reward deep neural network approximator as well as the optimizer algorithm and its parameters.

If you specify a ground-truth reward function using a custom function, the MBPO agent ignores these options.

If you have a previously trained reward model and do not want the MBPO agent to modify the model during training, set RewardOptimizerOptions.LearnRate to 0.

IsDoneOptimizerOptions — Is-done function optimizer options

rlOptimizerOptions object

Is-done function optimizer options, specified as an rlOptimizerOptions object. Using this object, you can specify training parameters for the is-done deep neural network approximator as well as the optimizer algorithm and its parameters.

If you specify a ground-truth is-done function using a custom function, the MBPO agent ignores these options.

If you have a previously trained is-done model and do not want the MBPO agent to modify the model during training, set `IsDoneOptimizerOptions.LearnRate` to `0`.

`ModelRolloutOptions` — Model roll-out options
`rlModelRolloutOptions` object

Model roll-out options for controlling the number and length of generated experience trajectories, specified as an `rlModelRolloutOptions` object with the following fields. At the start of each epoch, the agent generates the roll-out trajectories and adds them to the model experience buffer. To modify the roll-out options, use dot notation.

`NumRollout` — Number of trajectories
`2000` (default) | positive integer

Number of trajectories for generating samples, specified as a positive integer.

`Horizon` — Initial trajectory horizon
`1` (default) | positive integer

Initial trajectory horizon, specified as a positive integer.

`HorizonUpdateSchedule` — Option for increasing horizon length
`"none"` (default) | `"piecewise"`

Option for increasing the horizon length, specified as one of the following values.

- `"none"` — Do not increase the horizon length.
- `"piecewise"` — Increase the horizon length by one after every *N* model training epochs, where *N* is equal to `HorizonUpdateFrequency`.

`RolloutHorizonUpdateFrequency` — Number of epochs after which the horizon increases
`100` (default) | positive integer

Number of epochs after which the horizon increases, specified as a positive integer. When `RolloutHorizonSchedule` is `"none"` this option is ignored.

`HorizonMax` — Maximum horizon length
`20` (default) | positive integer

Maximum horizon length, specified as a positive integer greater than or equal to `RolloutHorizon`. When `RolloutHorizonSchedule` is `"none"` this option is ignored.

`HorizonUpdateStartEpoch` — Training epoch at which to start generating trajectories
`1` (default) | positive integer

Training epoch at which to start generating trajectories, specified as a positive integer.

`NoiseOptions` — Exploration model options
`[]` (default) | `EpsilonGreedyExploration` object | `GaussianActionNoise` object

Exploration model options for generating experiences using the internal environment model, specified as one of the following:

- `[]` — Use the exploration policy of the base agent. You must use this option when training a SAC base agent.

- `EpsilonGreedyExploration` object — You can use this option when training a DQN base agent.
- `GaussianActionNoise` object — You can use this option when training a DDPG or TD3 base agent.

The exploration model uses only the initial noise option values and does not update the values during training.

To specify `NoiseOptions`, create a default model object. Then, specify any nondefault model properties using dot notation.

- Specify epsilon greedy exploration options.

  ```
  opt = rlMBPOAgentOptions;
  opt.ModelRolloutOptions.NoiseOptions = ...
      rl.option.EpsilonGreedyExploration;
  opt.ModelRolloutOptions.NoiseOptions.EpsilonMin = 0.03;
  ```
- Specify Gaussian action noise options.

  ```
  opt = rlMBPOAgentOptions;
  opt.ModelRolloutOptions.NoiseOptions = ...
      rl.option.GaussianActionNoise;
  opt.ModelRolloutOptions.NoiseOptions.StandardDeviation = sqrt(0.15);
  ```

For more information on noise models, see “Noise Models” on page 3-223.

Object Functions

rlMBPOAgent	Model-based policy optimization (MBPO) reinforcement learning agent

Examples

Create MBPO Agent Options Object

Create an MBPO agent options object, specifying the ratio of real experiences to use for training the agent as 30%.

```
opt = rlMBPOAgentOptions(RealSampleRatio=0.3)
```

```
opt = 
  rlMBPOAgentOptions with properties:

       NumEpochForTrainingModel: 1
                 NumMiniBatches: 10
                  MiniBatchSize: 128
     TransitionOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
         RewardOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
         IsDoneOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
    ModelExperienceBufferLength: 100000
            ModelRolloutOptions: [1x1 rl.option.rlModelRolloutOptions]
                RealSampleRatio: 0.3000
                     InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the mini-batch size to 64.

```
opt.MiniBatchSize = 64;
```

Algorithms

Noise Models

Gaussian Action Noise

A GaussianActionNoise object has the following numeric value properties. When generating experiences, MBPO agents do not update their exploration model parameters.

Property	Description	Default Value
Mean	Noise model mean	0
StandardDeviation	Noise model standard deviation	sqrt(0.2)
StandardDeviationDecayRate	Decay rate of the standard deviation (not used for generating samples)	0
StandardDeviationMin	Minimum standard deviation, which must be less than StandardDeviation (not used for generating samples)	0.1
LowerLimit	Noise sample lower limit	-Inf
UpperLimit	Noise sample upper limit	Inf

At each time step k, the Gaussian noise v is sampled as shown in the following code.

```
w = Mean + rand(ActionSize).*StandardDeviation(k);
v(k+1) = min(max(w,LowerLimit),UpperLimit);
```

Epsilon Greedy Exploration

An EpsilonGreedyExploration object has the following numeric value properties. When generating experiences, MBPO agents do not update their exploration model parameters.

Property	Description	Default Value
Epsilon	Probability threshold to either randomly select an action or select the action that maximizes the state-action value function. A larger value of Epsilon means that the agent randomly explores the action space at a higher rate.	1
EpsilonMin	Minimum value of Epsilon (not used for generating samples)	0.01

Property	Description	Default Value
`EpsilonDecay`	Decay rate (not used for generating samples)	`0.005`

Version History

Introduced in R2022a

See Also

Objects

`rlMBPOAgent` | `rlNeuralNetworkEnvironment` | `rlAgentInitializationOptions`

Topics

"Train MBPO Agent to Balance Cart-Pole System"
"Model-Based Policy Optimization (MBPO) Agents"
"Reinforcement Learning Agents"

rlMDPEnv

Create Markov decision process environment for reinforcement learning

Description

A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of the decision maker. MDPs are useful for studying optimization problems solved using reinforcement learning. Use `rlMDPEnv` to create a Markov decision process environment for reinforcement learning in MATLAB.

Creation

Syntax

```
env = rlMDPEnv(MDP)
```

Description

`env = rlMDPEnv(MDP)` creates a reinforcement learning environment `env` with the specified `MDP` model.

Input Arguments

`MDP` — Markov decision process model
`GridWorld` object | `GenericMDP` object

Markov decision process model, specified as one of the following:

- `GridWorld` object created using `createGridWorld`.
- `GenericMDP` object created using `createMDP`.

Properties

`Model` — Markov decision process model
`GridWorld` object | `GenericMDP` object

Markov decision process model, specified as a `GridWorld` object or `GenericMDP` object.

`ResetFcn` — Reset function
function handle

Reset function, specified as a function handle.

Object Functions

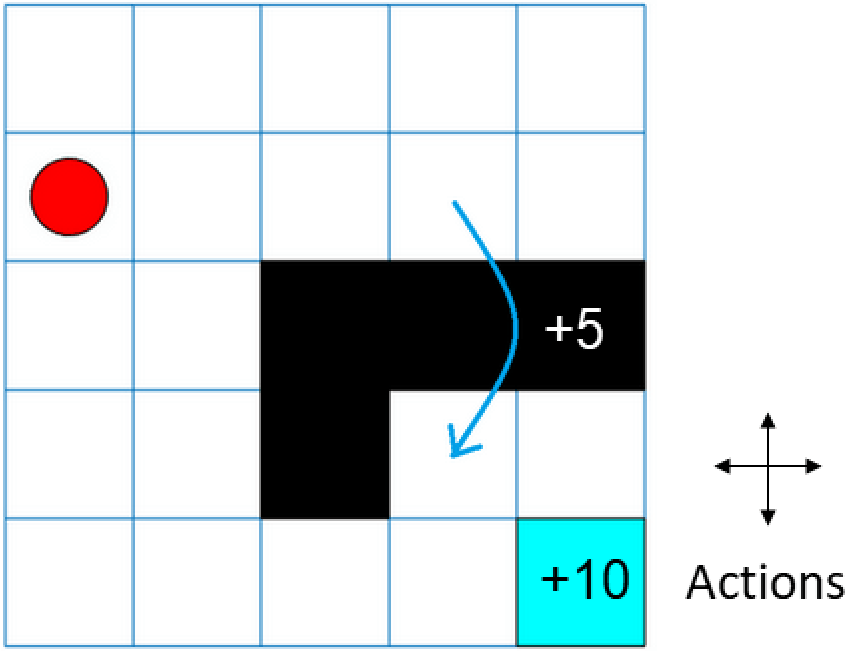
getActionInfo	Obtain action data specifications from reinforcement learning environment, agent, or experience buffer
getObservationInfo	Obtain observation data specifications from reinforcement learning environment, agent, or experience buffer
sim	Simulate trained reinforcement learning agents within specified environment
train	Train reinforcement learning agents within a specified environment
validateEnvironment	Validate custom reinforcement learning environment

Examples

Create Grid World Environment

For this example, consider a 5-by-5 grid world with the following rules:

1. A 5-by-5 grid world bounded by borders, with 4 possible actions (North = 1, South = 2, East = 3, West = 4).
2. The agent begins from cell [2,1] (second row, first column).
3. The agent receives reward +10 if it reaches the terminal state at cell [5,5] (blue).
4. The environment contains a special jump from cell [2,4] to cell [4,4] with +5 reward.
5. The agent is blocked by obstacles in cells [3,3], [3,4], [3,5] and [4,3] (black cells).
6. All other actions result in -1 reward.



First, create a `GridWorld` object using the `createGridWorld` function.

```
GW = createGridWorld(5,5)
```

```
GW = 
  GridWorld with properties:
```

```
                GridSize: [5 5]
            CurrentState: "[1,1]"
                  States: [25x1 string]
                 Actions: [4x1 string]
                       T: [25x25x4 double]
                       R: [25x25x4 double]
          ObstacleStates: [0x1 string]
          TerminalStates: [0x1 string]
    ProbabilityTolerance: 8.8818e-16
```

Now, set the initial, terminal and obstacle states.

```
GW.CurrentState = '[2,1]';
GW.TerminalStates = '[5,5]';
GW.ObstacleStates = ["[3,3]";"[3,4]";"[3,5]";"[4,3]"];
```

Update the state transition matrix for the obstacle states and set the jump rule over the obstacle states.

```
updateStateTranstionForObstacles(GW)
GW.T(state2idx(GW,"[2,4]"),:,:) = 0;
GW.T(state2idx(GW,"[2,4]"),state2idx(GW,"[4,4]"),:) = 1;
```

Next, define the rewards in the reward transition matrix.

```
nS = numel(GW.States);
nA = numel(GW.Actions);
GW.R = -1*ones(nS,nS,nA);
GW.R(state2idx(GW,"[2,4]"),state2idx(GW,"[4,4]"),:) = 5;
GW.R(:,state2idx(GW,GW.TerminalStates),:) = 10;
```

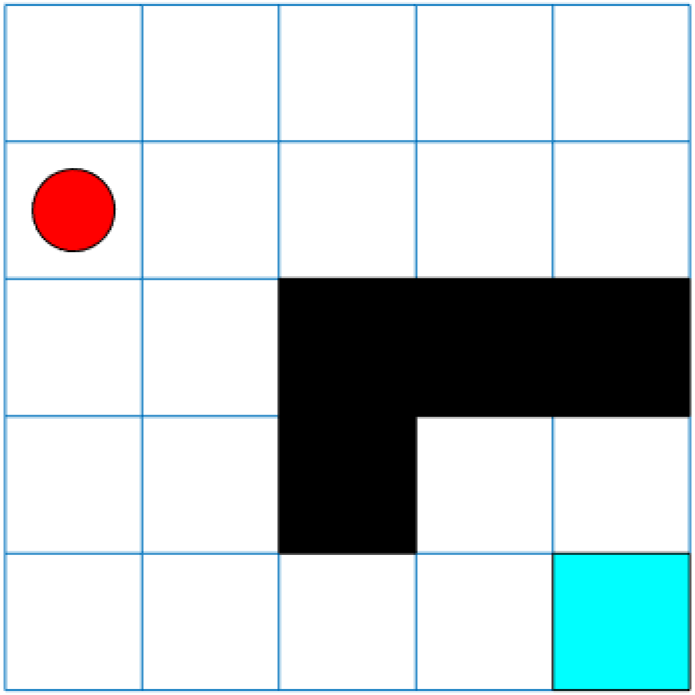
Now, use `rlMDPEnv` to create a grid world environment using the `GridWorld` object `GW`.

```
env = rlMDPEnv(GW)
```

```
env = 
  rlMDPEnv with properties:

       Model: [1x1 rl.env.GridWorld]
    ResetFcn: []
```

You can visualize the grid world environment using the `plot` function.

```
plot(env)
```



Version History

Introduced in R2019a

See Also

Functions

`createGridWorld` | `rlPredefinedEnv`

Topics

"Train Reinforcement Learning Agent in Basic Grid World"
"Train Reinforcement Learning Agent in MDP Environment"
"Create Custom Grid World Environments"

rlMultiAgentTrainingOptions

Options for training multiple reinforcement learning agents

Description

Use an `rlMultiAgentTrainingOptions` object to specify training options for multiple agents. To train the agents, use `train`.

For more information on training agents, see “Train Reinforcement Learning Agents”.

Creation

Syntax

```
trainOpts = rlMultiAgentTrainingOptions
trainOpts = rlMultiAgentTrainingOptions(Name,Value)
```

Description

`trainOpts = rlMultiAgentTrainingOptions` returns the default options for training multiple reinforcement learning agents. Use training options to specify parameters for the training session, such as the maximum number of episodes to train, criteria for stopping training, and criteria for saving agents. After configuring the options, use `trainOpts` as an input argument for `train`.

`trainOpts = rlMultiAgentTrainingOptions(Name,Value)` creates a training option set and sets object “Properties” on page 3-229 using one or more name-value pair arguments.

Properties

`AgentGroups` — Agent grouping indices
`"auto"` (default) | cell array of positive integers | cell array of integer arrays

Agent grouping indices, specified as a cell array of positive integers or a cell array of integer arrays.

For instance, consider a training scenario with 4 agents. You can group the agents in the following ways:

- Allocate each agent in a separate group:

  ```
  trainOpts = rlMultiAgentTrainingOptions("AgentGroups","auto")
  ```
- Specify four agent groups with one agent in each group:

  ```
  trainOpts = rlMultiAgentTrainingOptions("AgentGroups",{1,2,3,4})
  ```
- Specify two agent groups with two agents each:

  ```
  trainOpts = rlMultiAgentTrainingOptions("AgentGroups",{[1,2],[3,4]})
  ```
- Specify three agent groups:

```
trainOpts = rlMultiAgentTrainingOptions("AgentGroups",{[1,4],2,3})
```

AgentGroups and LearningStrategy must be used together to specify whether agent groups learn in a centralized manner or decentralized manner.

Example: AgentGroups={1,2,[3,4]}

LearningStrategy — Learning strategy for each agent group

"decentralized" (default) | "centralized"

Learning strategy for each agent group, specified as either "decentralized" or "centralized". In decentralized training, agents collect their own set of experiences during the episodes and learn independently from those experiences. In centralized training, the agents share the collected experiences and learn from them together.

AgentGroups and LearningStrategy must be used together to specify whether agent groups learn in a centralized manner or decentralized manner. For example, you can use the following command to configure training for three agent groups with different learning strategies. The agents with indices [1,2] and [3,5] learn in a centralized manner, while agent 4 learns in a decentralized manner.

```
trainOpts = rlMultiAgentTrainingOptions(...
     AgentGroups={[1,2],4,[3,5]},...
     LearningStrategy=["centralized","decentralized","centralized"] )
```

Example: LearningStrategy="centralized"

MaxEpisodes — Maximum number of episodes to train the agents

500 (default) | positive integer

Maximum number of episodes to train the agents, specified as a positive integer. Regardless of other criteria for termination, training terminates after MaxEpisodes.

Example: MaxEpisodes=1000

MaxStepsPerEpisode — Maximum number of steps to run per episode

500 (default) | positive integer

Maximum number of steps to run per episode, specified as a positive integer. In general, you define episode termination conditions in the environment. This value is the maximum number of steps to run in the episode if other termination conditions are not met.

Example: MaxStepsPerEpisode=1000

ScoreAveragingWindowLength — Window length for averaging

5 (default) | positive integer scalar | positive integer vector

Window length for averaging the scores, rewards, and number of steps for each agent, specified as a scalar or vector.

Specify a scalar to apply the same window length to all agents. To use a different window length for each agent, specify ScoreAveragingWindowLength as a vector. In this case, the order of the elements in the vector correspond to the order of the agents used during environment creation.

For options expressed in terms of averages, ScoreAveragingWindowLength is the number of episodes included in the average. For instance, if StopTrainingCriteria is "AverageReward" and StopTrainingValue is 500 for a given agent then for that agent, training terminates when the average reward over the number of episodes specified in `ScoreAveragingWindowLength` equals or exceeds `500`. For the other agents, training continues until:

- All agents reach their stop criteria.
- The number of episodes reaches `MaxEpisodes`.
- You stop training by clicking the **Stop Training** button in Episode Manager or pressing **Ctrl-C** at the MATLAB command line.

Example: `ScoreAveragingWindowLength=10`

`StopTrainingCriteria` — Training termination condition
`"AverageSteps"` (default) | `"AverageReward"` | `"EpisodeReward"` | `"GlobalStepCount"` | `"EpisodeCount"` | ...

Training termination condition, specified as one of the following strings:

- `"AverageSteps"` — Stop training when the running average number of steps per episode equals or exceeds the critical value specified by the option `StopTrainingValue`. The average is computed using the window `'ScoreAveragingWindowLength'`.
- `"AverageReward"` — Stop training when the running average reward equals or exceeds the critical value.
- `"EpisodeReward"` — Stop training when the reward in the current episode equals or exceeds the critical value.
- `"GlobalStepCount"` — Stop training when the total number of steps in all episodes (the total number of times the agent is invoked) equals or exceeds the critical value.
- `"EpisodeCount"` — Stop training when the number of training episodes equals or exceeds the critical value.

Example: `StopTrainingCriteria="AverageReward"`

`StopTrainingValue` — Critical value of training termination condition
`500` (default) | scalar | vector

Critical value of the training termination condition, specified as a scalar or a vector.

Specify a scalar to apply the same termination criterion to all agents. To use a different termination criterion for each agent, specify `StopTrainingValue` as a vector. In this case, the order of the elements in the vector corresponds to the order of the agents used during environment creation.

For a given agent, training ends when the termination condition specified by the `StopTrainingCriteria` option equals or exceeds this value. For the other agents, the training continues until:

- All agents reach their stop criteria.
- The number of episodes reaches `maxEpisodes`.
- You stop training by clicking the **Stop Training** button in Episode Manager or pressing **Ctrl-C** at the MATLAB command line.

For instance, if `StopTrainingCriteria` is `"AverageReward"` and `StopTrainingValue` is `100` for a given agent, then for that agent training terminates when the average reward over the number of episodes specified in `ScoreAveragingWindowLength` equals or exceeds `100`.

Example: `StopTrainingValue=100`

SaveAgentCriteria — Condition for saving agents during training

"None" (default) | "EpisodeReward" | "AverageSteps" | "AverageReward" | "GlobalStepCount" | "EpisodeCount" | ...

Condition for saving agents during training, specified as one of the following strings:

- "None" — Do not save any agents during training.
- "EpisodeReward" — Save the agent when the reward in the current episode equals or exceeds the critical value.
- "AverageSteps" — Save the agent when the running average number of steps per episode equals or exceeds the critical value specified by the option StopTrainingValue. The average is computed using the window 'ScoreAveragingWindowLength'.
- "AverageReward" — Save the agent when the running average reward over all episodes equals or exceeds the critical value.
- "GlobalStepCount" — Save the agent when the total number of steps in all episodes (the total number of times the agent is invoked) equals or exceeds the critical value.
- "EpisodeCount" — Save the agent when the number of training episodes equals or exceeds the critical value.

Set this option to store candidate agents that perform well according to the criteria you specify. When you set this option to a value other than "none", the software sets the SaveAgentValue option to 500. You can change that value to specify the condition for saving the agent.

For instance, suppose you want to store for further testing any agent that yields an episode reward that equals or exceeds 100. To do so, set SaveAgentCriteria to "EpisodeReward" and set the SaveAgentValue option to 100. When an episode reward equals or exceeds 100, train saves the corresponding agent in a MAT file in the folder specified by the SaveAgentDirectory option. The MAT file is called AgentK.mat, where K is the number of the corresponding episode. The agent is stored within that MAT file as saved_agent.

Example: SaveAgentCriteria="EpisodeReward"

SaveAgentValue — Critical value of condition for saving agents

"none" (default) | 500 | scalar | vector

Critical value of the condition for saving agents, specified as a scalar or a vector.

Specify a scalar to apply the same saving criterion to each agent. To save the agents when one meets a particular criterion, specify SaveAgentValue as a vector. In this case, the order of the elements in the vector corresponds to the order of the agents used when creating the environment. When a criteria for saving an agent is met, all agents are saved in the same MAT file.

When you specify a condition for saving candidate agents using SaveAgentCriteria, the software sets this value to 500. Change the value to specify the condition for saving the agent. See the SaveAgentCriteria option for more details.

Example: SaveAgentValue=100

SaveAgentDirectory — Folder for saved agents

"savedAgents" (default) | string | character vector

Folder for saved agents, specified as a string or character vector. The folder name can contain a full or relative path. When an episode occurs that satisfies the condition specified by the

SaveAgentCriteria and SaveAgentValue options, the software saves the agents in a MAT file in this folder. If the folder does not exist, train creates it. When SaveAgentCriteria is "none", this option is ignored and train does not create a folder.

Example: SaveAgentDirectory = pwd + "\run1\Agents"

StopOnError — Option to stop training when error occurs

"on" (default) | "off"

Option to stop training when an error occurs during an episode, specified as "on" or "off". When this option is "off", errors are captured and returned in the SimulationInfo output of train, and training continues to the next episode.

Example: StopOnError = "off"

Verbose — Option to display training progress on the command line

false (0) (default) | true (1)

Option to display training progress on the command line, specified as the logical value false (0) or true (1). Set to true to write information from each training episode to the MATLAB command line during training.

Example: Verbose = true

Plots — Option to display training progress with Episode Manager

"training-progress" (default) | "none"

Option to display training progress with Episode Manager, specified as "training-progress" or "none". By default, calling train opens the Reinforcement Learning Episode Manager, which graphically and numerically displays information about the training progress, such as the reward for each episode, average reward, number of episodes, and total number of steps. (For more information, see train.) To turn off this display, set this option to "none".

Example: Plots = "none"

Object Functions

train   Train reinforcement learning agents within a specified environment

Examples

Configure Options for Multi Agent Training

Create an options set for training 5 reinforcement learning agents. Set the maximum number of episodes and the maximum number of steps per episode to 1000. Configure the options to stop training when the average reward equals or exceeds 480, and turn on both the command-line display and Reinforcement Learning Episode Manager for displaying training results. You can set the options using name-value pair arguments when you create the options set. Any options that you do not explicitly set have their default values.

```
trainOpts = rlMultiAgentTrainingOptions(...
    AgentGroups={[1,2],3,[4,5]},...
    LearningStrategy=["centralized","decentralized","centralized"],...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=1000,...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=480,...
    Verbose=true,...
    Plots="training-progress")

trainOpts = 
  rlMultiAgentTrainingOptions with properties:

                   AgentGroups: {[1 2]  [3]  [4 5]}
              LearningStrategy: ["centralized"    "decentralized"    "centralized"]
                   MaxEpisodes: 1000
            MaxStepsPerEpisode: 1000
                   StopOnError: "on"
    ScoreAveragingWindowLength: 5
          StopTrainingCriteria: "AverageReward"
             StopTrainingValue: 480
             SaveAgentCriteria: "none"
                SaveAgentValue: "none"
            SaveAgentDirectory: "savedAgents"
                       Verbose: 1
                         Plots: "training-progress"
```

Alternatively, create a default options set and use dot notation to change some of the values.

```
trainOpts = rlMultiAgentTrainingOptions;

trainOpts.AgentGroups = {[1,2],3,[4,5]};
trainOpts.LearningStrategy = ["centralized","decentralized","centralized"];
trainOpts.MaxEpisodes = 1000;
trainOpts.MaxStepsPerEpisode = 1000;
trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = 480;
trainOpts.Verbose = true;
trainOpts.Plots = "training-progress";

trainOpts

trainOpts = 
  rlMultiAgentTrainingOptions with properties:

                   AgentGroups: {[1 2]  [3]  [4 5]}
              LearningStrategy: ["centralized"    "decentralized"    "centralized"]
                   MaxEpisodes: 1000
            MaxStepsPerEpisode: 1000
                   StopOnError: "on"
    ScoreAveragingWindowLength: 5
          StopTrainingCriteria: "AverageReward"
             StopTrainingValue: 480
             SaveAgentCriteria: "none"
                SaveAgentValue: "none"
            SaveAgentDirectory: "savedAgents"
                       Verbose: 1
                         Plots: "training-progress"
```

You can now use `trainOpts` as an input argument to the `train` command.

Configure Options for Training a Multi-Agent Environment

Create an options object for concurrently training three agents in the same environment.

Set the maximum number of episodes and the maximum steps per episode to 1000. Configure the options to stop training the first agent when its average reward over 5 episodes equals or exceeds 400, the second agent when its average reward over 10 episodes equals or exceeds 500, and the third when its average reward over 15 episodes equals or exceeds 600. The order of agents is the one used during environment creation.

Save the agents when the reward for the first agent in the current episode exceeds 100, or when the reward for the second agent exceeds 120, the reward for the third agent equals or exceeds 140.

Turn on both the command-line display and Reinforcement Learning Episode Manager for displaying training results. You can set the options using name-value pair arguments when you create the options set. Any options that you do not explicitly set have their default values.

```
trainOpts = rlMultiAgentTrainingOptions(...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=1000,...
    ScoreAveragingWindowLength=[5 10 15],...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=[400 500 600],...
    SaveAgentCriteria="EpisodeReward",...
    SaveAgentValue=[100 120 140],...
    Verbose=true,...
    Plots="training-progress")
```

```
trainOpts = 
  rlMultiAgentTrainingOptions with properties:

                   AgentGroups: "auto"
              LearningStrategy: "decentralized"
                   MaxEpisodes: 1000
            MaxStepsPerEpisode: 1000
                   StopOnError: "on"
    ScoreAveragingWindowLength: [5 10 15]
          StopTrainingCriteria: "AverageReward"
             StopTrainingValue: [400 500 600]
             SaveAgentCriteria: "EpisodeReward"
                SaveAgentValue: [100 120 140]
            SaveAgentDirectory: "savedAgents"
                       Verbose: 1
                         Plots: "training-progress"
```

Alternatively, create a default options set and use dot notation to change some of the values.

```
trainOpts = rlMultiAgentTrainingOptions;
trainOpts.MaxEpisodes = 1000;
trainOpts.MaxStepsPerEpisode = 1000;

trainOpts.ScoreAveragingWindowLength = [5 10 15];

trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = [400 500 600];

trainOpts.SaveAgentCriteria = "EpisodeReward";
trainOpts.SaveAgentValue = [100 120 140];

trainOpts.Verbose = true;
trainOpts.Plots = "training-progress";

trainOpts
```

```
trainOpts = 
  rlMultiAgentTrainingOptions with properties:

                   AgentGroups: "auto"
              LearningStrategy: "decentralized"
                   MaxEpisodes: 1000
            MaxStepsPerEpisode: 1000
                   StopOnError: "on"
    ScoreAveragingWindowLength: [5 10 15]
          StopTrainingCriteria: "AverageReward"
             StopTrainingValue: [400 500 600]
             SaveAgentCriteria: "EpisodeReward"
                SaveAgentValue: [100 120 140]
            SaveAgentDirectory: "savedAgents"
                       Verbose: 1
                         Plots: "training-progress"
```

You can specify a scalar to apply the same criterion to all agents. For example, use a window length of `10` for all three agents.

```
trainOpts.ScoreAveragingWindowLength = 10
```

```
trainOpts = 
  rlMultiAgentTrainingOptions with properties:

                   AgentGroups: "auto"
              LearningStrategy: "decentralized"
                   MaxEpisodes: 1000
            MaxStepsPerEpisode: 1000
                   StopOnError: "on"
    ScoreAveragingWindowLength: 10
          StopTrainingCriteria: "AverageReward"
             StopTrainingValue: [400 500 600]
             SaveAgentCriteria: "EpisodeReward"
                SaveAgentValue: [100 120 140]
            SaveAgentDirectory: "savedAgents"
                       Verbose: 1
                         Plots: "training-progress"
```

You can now use `trainOpts` as an input argument to the `train` command.

Version History

Introduced in R2022a

See Also

Functions
`train`

Objects
`rlTrainingOptions`

Topics
"Reinforcement Learning Agents"

rlNeuralNetworkEnvironment

Environment model with deep neural network transition models

Description

Use an rlNeuralNetworkEnvironment object to create a reinforcement learning environment that computes state transitions using deep neural networks.

Using an rlNeuralNetworkEnvironment object you can:

- Create an internal environment model for a model-based policy optimization (MBPO) agent.
- Create an environment for training other types of reinforcement learning agents. You can identify the state-transition network using experimental or simulated data.

Such environments can compute environment rewards and termination conditions using deep neural networks or custom functions.

Creation

Syntax

```
env = rlNeuralNetworkEnvironment(ObservationInfo,ActionInfo,transitionFcn,
rewardFcn,isDoneFcn)
```

Description

env = rlNeuralNetworkEnvironment(ObservationInfo,ActionInfo,transitionFcn,rewardFcn,isDoneFcn) creates a model for an environment with the observation and action specifications specified in ObservationInfo and ActionInfo, respectively. This syntax sets the TransitionFcn, RewardFcn, and IsDoneFcn properties.

Input Arguments

ObservationInfo — Observation specifications
rlNumericSpec object | array rlNumericSpec objects

This property is read-only.

Observation specifications, specified as an rlNumericSpec object or an array of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

You can extract the observation specifications from an existing environment or agent using getObservationInfo. You can also construct the specifications manually using rlNumericSpec.

ActionInfo — Action specifications
rlFiniteSetSpec object | rlNumericSpec object

Action specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

You can extract the action specifications from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

Properties

`TransitionFcn` — Environment transition function

`rlContinuousDeterministicTransitionFunction` object | `rlContinuousGaussianTransitionFunction` object | array of transition objects

Environment transition function, specified as one of the following:

- `rlContinuousDeterministicTransitionFunction` object — Use this option when you expect the environment transitions to be deterministic.
- `rlContinuousGaussianTransitionFunction` object — Use this option when you expect the environment transitions to be stochastic.
- Vector of transition objects — Use multiple transition models for an MBPO agent.

`RewardFcn` — Environment reward function

`rlContinuousDeterministicRewardFunction` object | `rlContinuousGaussianRewardFunction` object | function handle

Environment reward function, specified as one of the following:

- `rlContinuousDeterministicRewardFunction` object — Use this option when you do not know a ground-truth reward signal for your environment and you expect the reward signal to be deterministic.
- `rlContinuousGaussianRewardFunction` object — Use this option when you do not know a ground-truth reward signal for your environment and you expect the reward signal to be stochastic.
- Function handle — Use this option when you know a ground-truth reward signal for your environment. When you use an `rlNeuralNetworkEnvironment` object to create an `rlMBPOAgent` object, the custom reward function must return a batch of rewards given a batch of inputs.

`IsDoneFcn` — Environment is-done function

`rlIsDoneFunction` object | function handle

Environment is-done function, specified as one of the following:

- `rlIsDoneFunction` object — Use this option when you do not know a ground-truth termination signal for your environment.
- Function handle — Use this option when you know a ground-truth termination signal for your environment. When you use an `rlNeuralNetworkEnvironment` object to create an

rlMBPOAgent object, the custom is-done function must return a batch of termination signals given a batch of inputs.

Observation — Observation values
cell array

Observation values, specified as a cell array with length equal to the number of specification objects in ObservationInfo. The order of the observations in Observation must match the order in ObservationInfo. Also, the dimensions of each element of the cell array must match the dimensions of the corresponding observation specification in ObservationInfo.

To evaluate whether the transition models are well-trained, you can manually evaluate the environment for a given observation value using the step function. Specify the observation values before calling step.

When you use this neural network environment object within an MBPO agent, this property is ignored.

TransitionModelNum — Transition model index
1 (default) | positive integer

Transition model index, specified as a positive integer.

To evaluate whether the transition models are well-trained, you can manually evaluate the environment for a given observation value using the step function. To select which transition model in TransitionFcn to evaluate, specify the transition model index before calling step.

When you use this neural network environment object within an MBPO agent, this property is ignored.

Object Functions

rlMBPOAgent	Model-based policy optimization (MBPO) reinforcement learning agent

Examples

Create Neural Network Environment

Create an environment interface and extract observation and action specifications. Alternatively, you can create specifications using rlNumericSpec and rlFiniteSetSpec.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Get the dimension of the observation and action spaces.

```
numObservations = obsInfo.Dimension(1);
numActions = actInfo.Dimension(1);
```

Create a deterministic transition function based on a deep neural network with two input channels (current observations and actions) and one output channel (predicted next observation).

```
% Create network layers.
statePath = featureInputLayer(numObservations, ...
    Normalization="none",Name="state");
actionPath = featureInputLayer(numActions, ...
    Normalization="none",Name="action");
commonPath = [concatenationLayer(1,2,Name="concat")
    fullyConnectedLayer(64,Name="FC1")
    reluLayer(Name="CriticRelu1")
    fullyConnectedLayer(64, Name="FC3")
    reluLayer(Name="CriticCommonRelu2")
    fullyConnectedLayer(numObservations,Name="nextObservation")];

% Combine network layers.
transitionNetwork = layerGraph(statePath);
transitionNetwork = addLayers(transitionNetwork,actionPath);
transitionNetwork = addLayers(transitionNetwork,commonPath);
transitionNetwork = connectLayers( ...
    transitionNetwork,"state","concat/in1");
transitionNetwork = connectLayers( ...
    transitionNetwork,"action","concat/in2");

% Create dlnetwork object.
transitionNetwork = dlnetwork(transitionNetwork);

% Create transition function object.
transitionFcn = rlContinuousDeterministicTransitionFunction(...
    transitionNetwork,obsInfo,actInfo,...
    ObservationInputNames="state", ...
    ActionInputNames="action", ...
    NextObservationOutputNames="nextObservation");
```

Create a deterministic reward function with two input channels (current action and next observations) and one output channel (predicted reward value).

```
% Create network layers.
nextStatePath = featureInputLayer( ...
    numObservations,Name="nextState");
commonPath = [concatenationLayer(1,3,Name="concat")
    fullyConnectedLayer(32,Name="fc")
    reluLayer(Name="relu1")
    fullyConnectedLayer(32,Name="fc2")];
meanPath = [reluLayer(Name="rewardMeanRelu")
    fullyConnectedLayer(1,Name="rewardMean")];
stdPath = [reluLayer(Name="rewardStdRelu")
    fullyConnectedLayer(1,Name="rewardStdFc")
    softplusLayer(Name="rewardStd")];

% Combine network layers.
rewardNetwork = layerGraph(statePath);
rewardNetwork = addLayers(rewardNetwork,actionPath);
rewardNetwork = addLayers(rewardNetwork,nextStatePath);
rewardNetwork = addLayers(rewardNetwork,commonPath);
rewardNetwork = addLayers(rewardNetwork,meanPath);
rewardNetwork = addLayers(rewardNetwork,stdPath);

rewardNetwork = connectLayers( ...
    rewardNetwork,"nextState","concat/in1");
rewardNetwork = connectLayers( ...
    rewardNetwork,"action","concat/in2");
rewardNetwork = connectLayers( ...
    rewardNetwork,"state","concat/in3");
rewardNetwork = connectLayers( ...
    rewardNetwork,"fc2","rewardMeanRelu");
rewardNetwork = connectLayers( ...
    rewardNetwork,"fc2","rewardStdRelu");

% Create dlnetwork object.
rewardNetwork = dlnetwork(rewardNetwork);

% Create reward function object.
rewardFcn = rlContinuousGaussianRewardFunction(...
    rewardNetwork,obsInfo,actInfo,...
    ObservationInputNames="state",...
    ActionInputNames="action", ...
    NextObservationInputNames="nextState", ...
    RewardMeanOutputNames="rewardMean", ...
    RewardStandardDeviationOutputNames="rewardStd");
```

Create an is-done function with one input channel (next observations) and one output channel (predicted termination signal).

```
% Create network layers.
commonPath = [featureInputLayer(numObservations, ...
        Normalization="none",Name="nextState");
    fullyConnectedLayer(64,Name="FC1")
    reluLayer(Name="CriticRelu1")
    fullyConnectedLayer(64,Name="FC3")
    reluLayer(Name="CriticCommonRelu2")
    fullyConnectedLayer(2,Name="isdone0")
    softmaxLayer(Name="isdone")];
isDoneNetwork = layerGraph(commonPath);

% Create dlnetwork object.
isDoneNetwork = dlnetwork(isDoneNetwork);

% Create is-done function object.
isDoneFcn = rlIsDoneFunction(isDoneNetwork, ...
    obsInfo,actInfo, ...
    NextObservationInputNames="nextState");
```

Create a neural network environment using the transition, reward, and is-done functions.

```
env = rlNeuralNetworkEnvironment( ...
    obsInfo,actInfo, ...
    transitionFcn,rewardFcn,isDoneFcn);
```

Create Neural Network Environment Using Custom Functions

Create an environment interface and extract observation and action specifications. Alternatively, you can create specifications using `rlNumericSpec` and `rlFiniteSetSpec`.

```
env = rlPredefinedEnv("CartPole-Continuous");
obsInfo = getObservationInfo(env);
numObservations = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
numActions = actInfo.Dimension(1);
```

Create a deterministic transition function based on a deep neural network with two input channels (current observations and actions) and one output channel (predicted next observation).

```
% Create network layers.
statePath = featureInputLayer(numObservations,...
    Normalization="none",Name="state");
actionPath = featureInputLayer(numActions,...
    Normalization="none",Name="action");
commonPath = [concatenationLayer(1,2,Name="concat")
    fullyConnectedLayer(64,Name="FC1")
    reluLayer(Name="CriticRelu1")
    fullyConnectedLayer(64, Name="FC3")
    reluLayer(Name="CriticCommonRelu2")
    fullyConnectedLayer(numObservations,Name="nextObservation")];

% Combine network layers.
transitionNetwork = layerGraph(statePath);
transitionNetwork = addLayers(transitionNetwork,actionPath);
transitionNetwork = addLayers(transitionNetwork,commonPath);
transitionNetwork = connectLayers(transitionNetwork,"state","concat/in1");
transitionNetwork = connectLayers(transitionNetwork,"action","concat/in2");

% Create dlnetwork object.
transitionNetwork = dlnetwork(transitionNetwork);

% Create transition function object.
transitionFcn = rlContinuousDeterministicTransitionFunction(...
    transitionNetwork,obsInfo,actInfo,...
    ObservationInputNames="state", ...
    ActionInputNames="action", ...
    NextObservationOutputNames="nextObservation");
```

You can define a known reward function for your environment using a custom function. Your custom reward function must take the observations, actions, and next observations as cell-array inputs and return a scalar reward value. For this example, use the following custom reward function, which computes the reward based on the next observation.

```
type cartPoleRewardFunction.m
```

```
function reward = cartPoleRewardFunction(obs,action,nextObs)
% Compute reward value based on the next observation.

    if iscell(nextObs)
        nextObs = nextObs{1};
    end

    % Distance at which to fail the episode
    xThreshold = 2.4;

    % Reward each time step the cart-pole is balanced
    rewardForNotFalling = 1;

    % Penalty when the cart-pole fails to balance
    penaltyForFalling = -5;

    x = nextObs(1,:);
    distReward = 1 - abs(x)/xThreshold;
    isDone = cartPoleIsDoneFunction(obs,action,nextObs);

    reward = zeros(size(isDone));
    reward(logical(isDone)) = penaltyForFalling;
    reward(~logical(isDone)) = ...
        0.5 * rewardForNotFalling + 0.5 * distReward(~logical(isDone));
end
```

You can define a known is-done function for your environment using a custom function. Your custom is-done function must take the observations, actions, and next observations as cell-array inputs and return a logical termination signal. For this example, use the following custom is-done function, which computes the termination signal based on the next observation.

```
type cartPoleIsDoneFunction.m
```

```
function isDone = cartPoleIsDoneFunction(obs,action,nextObs)
% Compute termination signal based on next observation.

    if iscell(nextObs)
        nextObs = nextObs{1};
    end

    % Angle at which to fail the episode
    thetaThresholdRadians = 12 * pi/180;

    % Distance at which to fail the episode
    xThreshold = 2.4;

    x = nextObs(1,:);
    theta = nextObs(3,:);

    isDone = abs(x) > xThreshold | abs(theta) > thetaThresholdRadians;
end
```

Create a neural network environment using the transition function object and the custom reward and is-done functions.

```
env = rlNeuralNetworkEnvironment(obsInfo,actInfo,transitionFcn,...
    @cartPoleRewardFunction,@cartPoleIsDoneFunction);
```

Version History

Introduced in R2022a

See Also

Functions

evaluate | gradient | accelerate

Objects

rlContinuousDeterministicTransitionFunction | rlContinuousGaussianTransitionFunction | rlContinuousDeterministicRewardFunction | rlContinuousGaussianRewardFunction | rlIsDoneFunction | rlMBPOAgent | rlMBPOAgentOptions

Topics

"Train MBPO Agent to Balance Cart-Pole System"
"Model-Based Policy Optimization (MBPO) Agents"

rlNumericSpec

Create continuous action or observation data specifications for reinforcement learning environments

Description

An `rlNumericSpec` object specifies continuous action or observation data specifications for reinforcement learning environments.

Creation

Syntax

```
spec = rlNumericSpec(dimension)
spec = rlNumericSpec(dimension,Name,Value)
```

Description

`spec = rlNumericSpec(dimension)` creates a data specification for continuous actions or observations and sets the Dimension property.

`spec = rlNumericSpec(dimension,Name,Value)` sets "Properties" on page 3-246 using name-value pair arguments.

Properties

`LowerLimit` — Lower limit of the data space
`-Inf` (default) | scalar | matrix

Lower limit of the data space, specified as a scalar or matrix of the same size as the data space. When `LowerLimit` is specified as a scalar, `rlNumericSpec` applies it to all entries in the data space. DDPG, TD3 and SAC agents use this property to enforce lower limits on the action. When using other agents, if you need to enforce constraints on the action, you must do so within the environment.

`UpperLimit` — Upper limit of the data space
`Inf` (default) | scalar | matrix

Upper limit of the data space, specified as a scalar or matrix of the same size as the data space. When `UpperLimit` is specified as a scalar, `rlNumericSpec` applies it to all entries in the data space. DDPG, TD3 and SAC agents use this property to enforce upper limits on the action. When using other agents, if you need to enforce constraints on the action, you must do so within the environment.

Name — Name of the `rlNumericSpec` object
string

Name of the `rlNumericSpec` object, specified as a string. Use this property to set a meaningful name for the signal carried by this data channel. This property is used by the RL Agent block to match the bus signal elements with their corresponding environment channels.

Description — Description of the rlNumericSpec object

string

Description of the rlNumericSpec object, specified as a string. You can use this property to specify a meaningful description of the signal carried by this environment channel.

Dimension — Dimension of the data space

numeric vector

This property is read-only.

Dimension of the data space, specified as a numeric vector. This property is essential to construct agents and function approximators objects that work with a given environment.

DataType — Information about the type of data

"double" (default) | string

This property is read-only.

Information about the type of data, specified as a string, such as "double" or "single". The software uses this property to enforce data type consistency for observations and actions.

Object Functions

rlSimulinkEnv	Create reinforcement learning environment using dynamic model implemented in Simulink
rlFunctionEnv	Specify custom reinforcement learning environment dynamics using functions
rlValueFunction	Value function approximator object for reinforcement learning agents
rlQValueFunction	Q-Value function approximator object for reinforcement learning agents
rlVectorQValueFunction	Vector Q-value function approximator for reinforcement learning agents
rlContinuousDeterministicActor	Deterministic actor with a continuous action space for reinforcement learning agents
rlDiscreteCategoricalActor	Stochastic categorical actor with a discrete action space for reinforcement learning agents
rlContinuousGaussianActor	Stochastic Gaussian actor with a continuous action space for reinforcement learning agents

Examples

Create Reinforcement Learning Environment for Simulink Model

For this example, consider the rlSimplePendulumModel Simulink model. The model is a simple frictionless pendulum that initially hangs in a downward position.

Open the model.

```
mdl = "rlSimplePendulumModel";
open_system(mdl)
```

Create rlNumericSpec and rlFiniteSetSpec objects for the observation and action information, respectively.

The observation is a vector containing three signals: the sine, cosine, and time derivative of the angle.

```
obsInfo = rlNumericSpec([3 1])
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [3 1]
       DataType: "double"
```

The action is a scalar expressing the torque and can be one of three possible values, -2 Nm, 0 Nm and 2 Nm.

```
actInfo = rlFiniteSetSpec([-2 0 2])
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [3x1 double]
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

You can use dot notation to assign property values for the rlNumericSpec and rlFiniteSetSpec objects.

```
obsInfo.Name = "observations";
actInfo.Name = "torque";
```

Assign the agent block path information, and create the reinforcement learning environment for the Simulink model using the information extracted in the previous steps.

```
agentBlk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,agentBlk,obsInfo,actInfo)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

You can also include a reset function using dot notation. For this example, randomly initialize theta0 in the model workspace.

```
env.ResetFcn = @(in) setVariable(in,"theta0",randn,"Workspace",mdl)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : @(in)setVariable(in,"theta0",randn,"Workspace",mdl)
  UseFastRestart : on
```

Version History

Introduced in R2019a

See Also

Functions

`rlSimulinkEnv` | `getActionInfo` | `getObservationInfo`

Objects

`rlFiniteSetSpec` | `rlValueRepresentation` | `rlQValueRepresentation` | `rlDeterministicActorRepresentation` | `rlStochasticActorRepresentation` | `rlFunctionEnv`

Blocks

RL Agent

Topics

“Train DDPG Agent for Adaptive Cruise Control”
“Create MATLAB Reinforcement Learning Environments”
“Create Simulink Reinforcement Learning Environments”

rlOptimizerOptions

Optimization options for actors and critics

Description

Use an `rlOptimizerOptions` object to specify an optimization options set for actors and critics.

Creation

Syntax

```
optOpts = rlOptimizerOptions
optOpts = rlOptimizerOptions(Name=Value)
```

Description

`optOpts = rlOptimizerOptions` creates a default optimizer option set to use as a `CriticOptimizerOptions` or `ActorOptimizerOptions` property of an agent option object, or as a last argument of `rlOptimizer` to create an optimizer object. You can modify the object properties using dot notation.

`optOpts = rlOptimizerOptions(Name=Value)` creates an options set with the specified properties using one or more name-value arguments.

Properties

`LearnRate` — Learning rate used in training the actor or critic function approximator
`0.01` (default) | positive scalar

Learning rate used in training the actor or critic function approximator, specified as a positive scalar. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Example: `LearnRate=0.025`

`GradientThreshold` — Gradient threshold value for the training of the actor or critic function approximator
`Inf` (default) | positive scalar

Gradient threshold value used in training the actor or critic function approximator, specified as `Inf` or a positive scalar. If the gradient exceeds this value, the gradient is clipped as specified by the `GradientThresholdMethod` option. Clipping the gradient limits how much the network parameters can change in a training iteration.

Example: `GradientThreshold=1`

`GradientThresholdMethod` — Gradient threshold method used in training the actor or critic function approximator
`"l2norm"` (default) | `"global-l2norm"` | `"absolute-value"`

Gradient threshold method used in training the actor or critic function approximator. This is the specific method used to clip gradient values that exceed the gradient threshold, and it is specified as one of the following values.

- `"l2norm"` — If the L_2 norm of the vector G_{lyr} containing the gradient components related to the weights or biases of a layer is larger than `GradientThreshold`, then this option scales G_{lyr} by a factor of `GradientThreshold/L`, where L is the L_2 norm of G_{lyr}. When you use this option, the L_2 norm of G_{lyr} in the returned gradient cannot exceed `GradientThreshold`. For example, a fully connected layer has two parameter arrays, `Weights` and `Bias`. The threshold is applied to the L_2 norm of the gradient components related to `Weights` and `Bias` separately.
- `"global-l2norm"` — If the L_2 norm of the gradient G (with respect to all learnable network parameters), is larger than `GradientThreshold`, then this option scales G by a factor of L, where L is the L_2 norm of G. When you use this option, the L_2 norm of the returned gradient cannot exceed `GradientThreshold`.
- `"absolute-value"` — If the absolute value of an individual (scalar) partial derivative in the gradient G (with respect to all learnable network parameters), is larger than `GradientThreshold`, then this option scales the partial derivative so that the corresponding component in the returned gradient has magnitude equal to `GradientThreshold` and the same sign of the original partial derivative. When you use this option, the absolute value of any component of the returned gradient cannot exceed `GradientThreshold`.

For more information, see “Gradient Clipping” in the Algorithms section of `trainingOptions` in Deep Learning Toolbox.

Example: `GradientThresholdMethod="absolute-value"`

`L2RegularizationFactor` — Factor for L_2 regularization used in training the actor or critic function approximator

0.0001 (default) | nonnegative scalar

Factor for L_2 regularization (weight decay) used in training the actor or critic function approximator, specified as a nonnegative scalar. For more information, see “L2 Regularization” in the Algorithms section of `trainingOptions` in Deep Learning Toolbox.

To avoid overfitting when using a representation with many parameters, consider increasing the `L2RegularizationFactor` option.

Example: `L2RegularizationFactor=0.0005`

`Algorithm` — Algorithm used for training actor or critic function approximator

`"adam"` (default) | `"sgdm"` | `"rmsprop"`

Algorithm used for training the actor or critic function approximator, specified as one of the following values.

- `"adam"` — Use the Adam (adaptive movement estimation) algorithm. You can specify the decay rates of the gradient and squared gradient moving averages using the `GradientDecayFactor` and `SquaredGradientDecayFactor` fields of the `OptimizerParameters` option.
- `"sgdm"` — Use the stochastic gradient descent with momentum (SGDM) algorithm. You can specify the momentum value using the `Momentum` field of the `OptimizerParameters` option.
- `"rmsprop"` — Use the RMSProp algorithm. You can specify the decay rate of the squared gradient moving average using the `SquaredGradientDecayFactor` fields of the `OptimizerParameters` option.

For more information about these algorithms, see “Stochastic Gradient Descent” in the Algorithms section of `trainingOptions` in Deep Learning Toolbox.

Example: `Optimizer="sgdm"`

`OptimizerParameters` — Parameters for the training algorithm used for training the actor or critic function approximator

`OptimizerParameters` object

Parameters for the training algorithm used for training the actor or critic function approximator, specified as an `OptimizerParameters` object with the following parameters.

Parameter	Description
`Momentum`	Contribution of previous step, specified as a scalar from 0 to 1. A value of 0 means no contribution from the previous step. A value of 1 means maximal contribution. This parameter applies only when `Optimizer` is `"sgdm"`. In that case, the default value is 0.9. This default value works well for most problems.
`Epsilon`	Denominator offset, specified as a positive scalar. The optimizer adds this offset to the denominator in the network parameter updates to avoid division by zero. This parameter applies only when `Optimizer` is `"adam"` or `"rmsprop"`. In that case, the default value is 10^{-8}. This default value works well for most problems.
`GradientDecayFactor`	Decay rate of gradient moving average, specified as a positive scalar from 0 to 1. This parameter applies only when `Optimizer` is `"adam"`. In that case, the default value is 0.9. This default value works well for most problems.
`SquaredGradientDecayFactor`	Decay rate of squared gradient moving average, specified as a positive scalar from 0 to 1. This parameter applies only when `Optimizer` is `"adam"` or `"rmsprop"`. In that case, the default value is 0.999. This default value works well for most problems.

When a particular property of `OptimizerParameters` is not applicable to the optimizer type specified in `Algorithm`, that property is set to `"Not applicable"`.

To change property values, create an `rlOptimizerOptions` object and use dot notation to access and change the properties of `OptimizerParameters`.

```
repOpts = rlRepresentationOptions;
repOpts.OptimizerParameters.GradientDecayFactor = 0.95;
```

Object Functions

rlQAgentOptions	Options for Q-learning agent
rlSARSAAgentOptions	Options for SARSA agent
rlDQNAgentOptions	Options for DQN agent
rlPGAgentOptions	Options for PG agent
rlDDPGAgentOptions	Options for DDPG agent
rlTD3AgentOptions	Options for TD3 agent
rlACAgentOptions	Options for AC agent
rlPPOAgentOptions	Options for PPO agent
rlTRPOAgentOptions	Options for TRPO agent
rlSACAgentOptions	Options for SAC agent
rlOptimizer	Creates an optimizer object for actors and critics

Examples

Create Optimizer Options Object

Use rlOprimizerOptions to create a default optimizer option object to use for the training of a critic function approximator.

```
myCriticOpts = rlOptimizerOptions
```

```
myCriticOpts = 
  rlOptimizerOptions with properties:

                    LearnRate: 0.0100
            GradientThreshold: Inf
      GradientThresholdMethod: "l2norm"
       L2RegularizationFactor: 1.0000e-04
                    Algorithm: "adam"
          OptimizerParameters: [1x1 rl.option.OptimizerParameters]
```

Using dot notation, change the training algorithm to stochastic gradient descent with momentum and set the value of the momentum parameter to 0.6.

```
myCriticOpts.Algorithm = "sgdm";
myCriticOpts.OptimizerParameters.Momentum = 0.6;
```

Create an AC agent option object, and set its CriticOptimizerOptions property to myCriticOpts.

```
myAgentOpt = rlACAgentOptions;
myAgentOpt.CriticOptimizerOptions = myCriticOpts;
```

You can now use myAgentOpt as last input argument to rlACAgent when creating your AC agent.

Create Optimizer Options Object Specifying Property Values

Use rlOprimizerOptions to create an optimizer option object to use for the training of an actor function approximator. Specify a learning rate of 0.2 and set the GradientThresholdMethod to "absolute-value".

```
myActorOpts=rlOptimizerOptions(LearnRate=0.2, ...
    GradientThresholdMethod="absolute-value")
```

```
myActorOpts = 
  rlOptimizerOptions with properties:

                  LearnRate: 0.2000
          GradientThreshold: Inf
    GradientThresholdMethod: "absolute-value"
     L2RegularizationFactor: 1.0000e-04
                  Algorithm: "adam"
        OptimizerParameters: [1x1 rl.option.OptimizerParameters]
```

Using dot notation, change the a `GradientThreshold` to `10`.

```
myActorOpts.GradientThreshold = 10;
```

Create an AC agent option object and set its `ActorOptimizerOptions` property to `myActorOpts`.

```
myAgentOpt = rlACAgentOptions( ...
    ActorOptimizerOptions=myActorOpts);
```

You can now use `myAgentOpt` as last input argument to `rlACAgent` when creating your AC agent.

Version History

Introduced in R2022a

See Also

Functions

`rlOptimizer`

Topics

"Create Policies and Value Functions"
"Reinforcement Learning Agents"

rlPGAgent

Policy gradient (PG) reinforcement learning agent

Description

The policy gradient (PG) algorithm is a model-free, online, on-policy reinforcement learning method. A PG agent is a policy-based reinforcement learning agent that uses the REINFORCE algorithm to directly compute an optimal policy which maximizes the long-term reward. The action space can be either discrete or continuous. For continuous action spaces, this agent does not enforce constraints set in the action specification; therefore, if you need to enforce action constraints, you must do so within the environment.

For more information on PG agents and the REINFORCE algorithm, see “Policy Gradient (PG) Agents”. For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlPGAgent(observationInfo,actionInfo)
agent = rlPGAgent(observationInfo,actionInfo,initOpts)

agent = rlPGAgent(actor)
agent = rlPGAgent(actor,critic)

agent = rlPGAgent(___,agentOptions)
```

Description

Create Agent from Observation and Action Specifications

`agent = rlPGAgent(observationInfo,actionInfo)` creates a policy gradient agent for an environment with the given observation and action specifications, using default initialization options. The actor and critic in the agent use default deep neural networks built from the observation specification `observationInfo` and the action specification `actionInfo`. The `ObservationInfo` and `ActionInfo` properties of `agent` are set to the `observationInfo` and `actionInfo` input arguments, respectively.

`agent = rlPGAgent(observationInfo,actionInfo,initOpts)` creates a policy gradient agent for an environment with the given observation and action specifications. The agent uses default networks in which each hidden fully connected layer has the number of units specified in the `initOpts` object. Policy gradient agents do not support recurrent neural networks. For more information on the initialization options, see `rlAgentInitializationOptions`.

Create Agent from Actor and Critic

`agent = rlPGAgent(actor)` creates a policy gradient agent with the specified actor network. By default, the `UseBaseline` property of the agent is `false` in this case.

agent = rlPGAgent(actor,critic) creates a policy gradient agent with the specified actor and critic networks. By default, the UseBaseline option is true in this case.

Specify Agent Options

agent = rlPGAgent(___,agentOptions) creates a policy gradient agent and sets the AgentOptions property to the agentOptions input argument. Use this syntax after any of the input arguments in the previous syntaxes.

Input Arguments

initOpts — Agent initialization options

rlAgentInitializationOptions object

Agent initialization options, specified as an rlAgentInitializationOptions object. Policy gradient agents do not support recurrent neural networks.

actor — Actor

rlDiscreteCategoricalActor object | rlContinuousGaussianActor object

Actor that implements the policy, specified as an rlDiscreteCategoricalActor or rlContinuousGaussianActor function approximator object. For more information on creating actor approximators, see “Create Policies and Value Functions”.

critic — Baseline critic

rlValueFunction object

Baseline critic that estimates the discounted long-term reward, specified as an rlValueFunction object. For more information on creating critic approximators, see “Create Policies and Value Functions”.

Properties

ObservationInfo — Observation specifications

specification object | array of specification objects

Observation specifications, specified as an rlFiniteSetSpec or rlNumericSpec object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

If you create the agent by specifying an actor and critic, the value of ObservationInfo matches the value specified in the actor and critic objects.

You can extract observationInfo from an existing environment or agent using getObservationInfo. You can also construct the specifications manually using rlFiniteSetSpec or rlNumericSpec.

ActionInfo — Action specification

rlFiniteSetSpec object | rlNumericSpec object

Action specifications, specified either as an rlFiniteSetSpec (for discrete action spaces) or rlNumericSpec (for continuous action spaces) object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

If you create the agent by specifying an actor and critic, the value of `ActionInfo` matches the value specified in the actor and critic objects.

You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`AgentOptions` — Agent options
`rlPGAgentOptions` object

Agent options, specified as an `rlPGAgentOptions` object.

`UseExplorationPolicy` — Option to use exploration policy
`true` (default) | `false`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- `true` — Use the base agent exploration policy when selecting actions in `sim` and `generatePolicyFunction`. In this case, the agent selects its actions by sampling its probability distribution, the policy is therefore stochastic and the agent explores its observation space.
- `false` — Use the base agent greedy policy (the action with maximum likelihood) when selecting actions in `sim` and `generatePolicyFunction`. In this case, the simulated agent and generated policy behave deterministically.

Note This option affects only simulation and deployment; it does not affect training.

`SampleTime` — Sample time of agent
positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations. The value of `SampleTime` matches the value specified in `AgentOptions`.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent

setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent
setCritic	Set critic of reinforcement learning agent
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create Discrete Policy Gradient Agent from Observation and Action Specifications

Create an environment with a discrete action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar with five possible elements (a torque of -2, -1, 0, 1, or 2 Nm applied to the pole).

```
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");
```

Obtain observation and action specification objects.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a policy gradient agent from the environment observation and action specifications.

```
agent = rlPGAgent(obsInfo,actInfo);
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[-2]}
```

You can now test and train the agent within the environment.

Create Continuous Policy Gradient Agent Using Initialization Options

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar representing a torque ranging continuously from -2 to 2 Nm.

```
env = rlPredefinedEnv("SimplePendulumWithImage-Continuous");
```

Obtain observation and action specifications

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create an agent initialization option object, specifying that each hidden fully connected layer in the network must have 128 neurons (instead of the default number, 256). Policy gradient agents do not support recurrent networks, so setting the `UseRNN` option to `true` generates an error when the agent is created.

```
initOpts = rlAgentInitializationOptions(NumHiddenUnit=128);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a policy gradient agent from the environment observation and action specifications.

```
agent = rlPGAgent(obsInfo,actInfo,initOpts);
```

Extract the deep neural networks from both the agent actor and critic.

```
actorNet = getModel(getActor(agent));
criticNet = getModel(getCritic(agent));
```

To verify that each hidden fully connected layer has 128 neurons, you can display the layers on the MATLAB® command window,

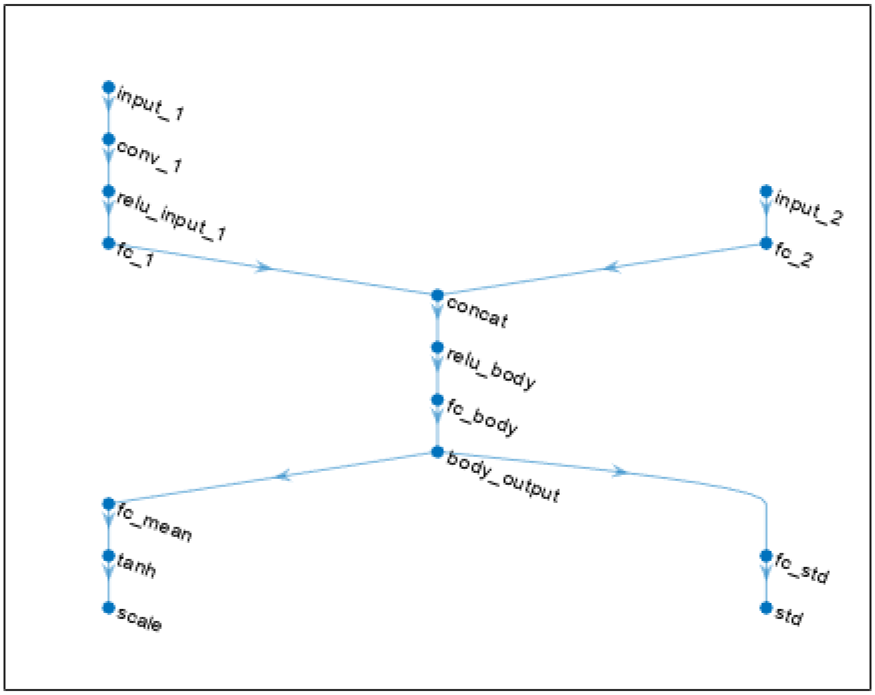
```
criticNet.Layers
```

or visualize the structure interactively using `analyzeNetwork`.

```
analyzeNetwork(criticNet)
```

Plot actor and critic networks, and display their number of weights.

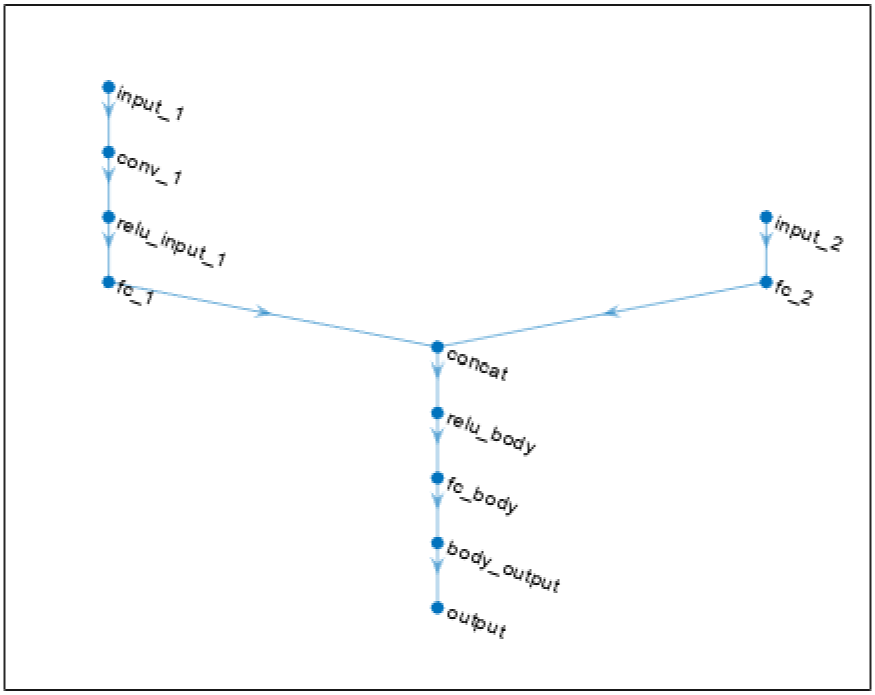
```
plot(layerGraph(actorNet))
```



```
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 18.9M

   Inputs:
      1   'input_1'   50x50x1 images
      2   'input_2'   1 features
```

```
plot(layerGraph(criticNet))
```



```
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 18.9M

   Inputs:
      1   'input_1'   50x50x1 images
      2   'input_2'   1 features
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[0.9228]}
```

You can now test and train the agent within the environment.

Create a Discrete PG Agent from Actor and Baseline Critic

Create an environment with a discrete action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Train PG Agent with Baseline to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force, applied to the mass, having three possible values (-2, 0, or 2 Newton).

```
env = rlPredefinedEnv("DoubleIntegrator-Discrete");
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "states"
    Description: "x, dx"
      Dimension: [2 1]
       DataType: "double"
```

```
actInfo = getActionInfo(env)
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [-2 0 2]
           Name: "force"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

Policy-gradient agents can optionally use a parametrized baseline function to approximate the value of the policy. A value-function approximator takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized baseline function, use a neural network with one input layer (which receives the content of the observation channel, as specified by obsInfo) and one output layer (which returns the scalar value). Note that prod(obsInfo.Dimension) returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects.

```
baselineNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert to a dlnetwork object and display the number of weights.

```
baselineNet = dlnetwork(baselineNet);
summary(baselineNet)
```

```
   Initialized: true

   Number of learnables: 257
   Inputs:
      1   'input'   2 features
```

Create the baseline approximator object using `baselineNet` and the observation specification. For more information on value function approximators, see `rlValueFunction`.

```
baseline = rlValueFunction(baselineNet,obsInfo);
```

Check the critic with a random input observation.

```
getValue(baseline,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.1204
```

Policy gradient agents use a parametrized stochastic policy, which for discrete action spaces is implemented by a discrete categorical actor. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by `actInfo`. Note that `numel(actInfo.Dimension)` returns the number of elements of the discrete action space.

Define the network as an array of layer objects.

```
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert to a `dlnetwork` object and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 387

   Inputs:
      1   'input'   2 features
```

Create the actor using `actorNet` and the observation and action specifications. For more information on discrete categorical actors, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-2]}
```

Create the PG agent using the actor and the baseline critic.

```
agent = rlPGAgent(actor,baseline)
```

```
agent = 
  rlPGAgent with properties:

            AgentOptions: [1x1 rl.option.rlPGAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Specify options for the agent, including training options for the actor and critic.

```
agent.AgentOptions.UseBaseline = true;
agent.AgentOptions.DiscountFactor = 0.99;

agent.AgentOptions.CriticOptimizerOptions.LearnRate = 5e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;

agent.AgentOptions.ActorOptimizerOptions.LearnRate = 5e-3;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold = 1;
```

Check the agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-2]}
```

You can now test and train the agent within the environment.

Create a Continuous PG Agent from Actor and Baseline Critic

Create an environment with a continuous action space, and obtain its observation and action specifications. For this example, load the double integrator continuous action space environment used in the example “Train DDPG Agent to Control Double Integrator System”.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "states"
    Description: "x, dx"
      Dimension: [2 1]
       DataType: "double"
```

```
actInfo = getActionInfo(env)
actInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "force"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

In this example, the action is a scalar value representing a force ranging from -2 to 2 Newton. To make sure that the output from the agent is in this range, you perform an appropriate scaling operation. Store these limits so you can easily access them later.

```
actInfo.LowerLimit = -2;
actInfo.UpperLimit = 2;
```

Policy-gradient agents can optionally use a parametrized baseline function to approximate the value of the policy. A value-function approximator takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized baseline function, use a neural network with one input layer (which receives the content of the observation channel, as specified by `obsInfo`) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects.

```
baselineNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert to a `dlnetwork` object and display the number of weights.

```
baselineNet = dlnetwork(baselineNet);
summary(baselineNet)
```

```
   Initialized: true

   Number of learnables: 257

   Inputs:
      1   'input'   2 features
```

Create the baseline approximator object using `baselineNet` and the observation specification. For more information on value function approximators, see `rlValueFunction`.

```
baseline = rlValueFunction(baselineNet,obsInfo);
```

Check the critic with a random input observation.

```
getValue(baseline,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.1204
```

Policy-gradient agents use a parametrized stochastic policy, which for continuous action spaces is implemented by a continuous Gaussian actor. This actor takes an observation as input and returns as output a random action sampled from a Gaussian probability distribution.

To approximate the mean values and standard deviations of the Gaussian distribution, you must use a neural network with two output layers, each having as many elements as the dimension of the action space. One output layer must return a vector containing the mean values for each action dimension. The other must return a vector containing the standard deviation for each action dimension.

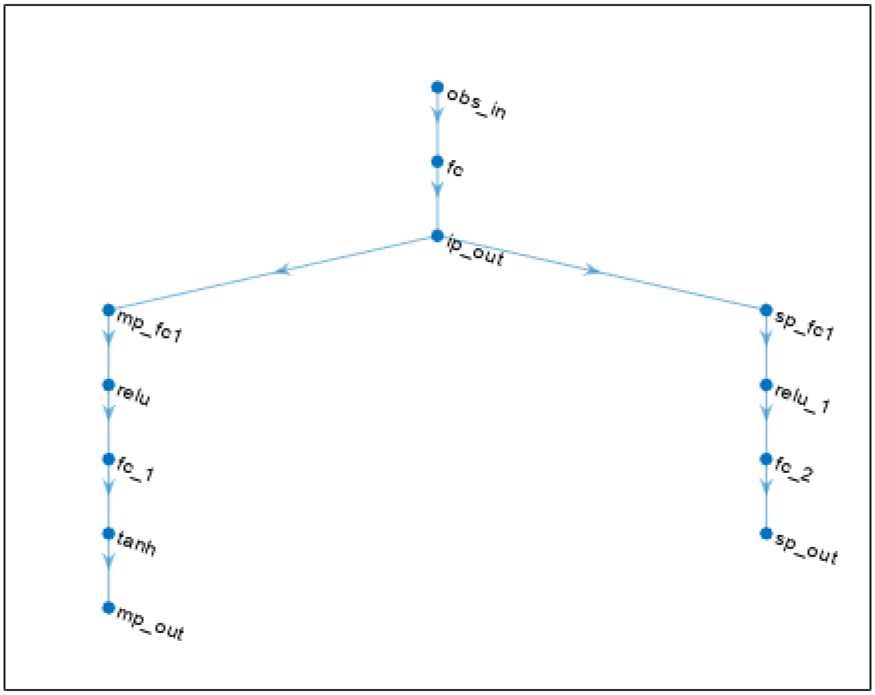
Note that standard deviations must be nonnegative and mean values must fall within the range of the action. Therefore the output layer that returns the standard deviations must be a softplus or ReLU layer, to enforce nonnegativity, while the output layer that returns the mean values must be a scaling layer, to scale the mean values to the output range.

For this example the environment has only one observation channel and therefore the network has only one input layer.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% Input path
inPath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="obs_in")
    fullyConnectedLayer(32)
    reluLayer(Name="ip_out") ];

% Mean path
meanPath = [
    fullyConnectedLayer(16,Name="mp_fc1")
    reluLayer
    fullyConnectedLayer(1)
    tanhLayer(Name="tanh"); % range: -1,1
    scalingLayer(Name="mp_out", ...
        Scale=actInfo.UpperLimit) ]; % range: -2,2

% Standard deviation path
sdevPath = [
    fullyConnectedLayer(16,Name="sp_fc1")
    reluLayer
    fullyConnectedLayer(1);
    softplusLayer(Name="sp_out") ]; % non negative

% Add layers to layerGraph object
actorNet = layerGraph(inPath);
actorNet = addLayers(actorNet,meanPath);
actorNet = addLayers(actorNet,sdevPath);

% Connect output of inPath to meanPath input
actorNet = connectLayers(actorNet,"ip_out","mp_fc1/in");
% Connect output of inPath to variancePath input
actorNet = connectLayers(actorNet,"ip_out","sp_fc1/in");
```

```
% Plot network
plot(actorNet)
```



```
% Convert to dlnetwork object
actorNet = dlnetwork(actorNet);

% Display the number of weights
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 1.1k

   Inputs:
      1   'obs_in'   2 features
```

Create the actor approximator object using `actorNet`, the environment specifications, and the names of the network input and output layers as input arguments. For more information, on continuous Gaussian actors, see `rlContinuousGaussianActor`.

```
actor = rlContinuousGaussianActor(actorNet, ...
    obsInfo,actInfo, ...
    ObservationInputNames="obs_in", ...
    ActionMeanOutputNames="mp_out", ...
    ActionStandardDeviationOutputNames="sp_out");
```

Check the actor with a random input observation.

```
getAction(actor,{rand(obsInfo.Dimension)})

ans = 1x1 cell array
    {[0.0963]}
```

Create the PG agent using the actor and the baseline critic.

```
agent = rlPGAgent(actor,baseline)

agent = 
  rlPGAgent with properties:

            AgentOptions: [1x1 rl.option.rlPGAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlNumericSpec]
              SampleTime: 1
```

Specify options for the agent, including training options for the actor and critic.

```
agent.AgentOptions.UseBaseline = true;
agent.AgentOptions.DiscountFactor = 0.99;

agent.AgentOptions.CriticOptimizerOptions.LearnRate = 5e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;

agent.AgentOptions.ActorOptimizerOptions.LearnRate = 5e-3;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold = 1;
```

Check your agent with a random input observation.

```
getAction(agent,{rand(obsInfo.Dimension)})

ans = 1x1 cell array
    {[1.1197]}
```

You can now test and train the agent within the environment.

Create a Discrete PG Agent with Recurrent Neural Networks

For this example, load the environment used in the example “Train PG Agent with Baseline to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force, applied to the mass, having three possible values (-2, 0, or 2 Newton).

```
env = rlPredefinedEnv("DoubleIntegrator-Discrete");
```

Get observation and specification information.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Policy-gradient agents can optionally use a parametrized baseline function to approximate the value of the policy. To model the parametrized baseline function, use a recurrent neural network.

Define the network as an array of layer objects. To create a recurrent network, use a sequenceInputLayer as the input layer and include at least one lstmLayer.

```
baselineNet = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    lstmLayer(32)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert to a dlnetwork object and display the number of weights.

```
baselineNet = dlnetwork(baselineNet);
summary(baselineNet)
```

```
   Initialized: true

   Number of learnables: 4.5k

   Inputs:
      1   'sequenceinput'   Sequence input with 2 dimensions
```

Create the baseline approximator object using baselineNet and the observation specification.

```
baseline = rlValueFunction(baselineNet,obsInfo);
```

Check the baseline critic with a random input observation.

```
getValue(baseline,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.0065
```

Policy gradient agents use a parametrized stochastic policy, which for discrete action spaces is implemented by a discrete categorical actor.

A discrete categorical actor implements a parametrized stochastic policy for a discrete action space.

This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

Since the critic has a recurrent network, the actor must have a recurrent network too. The network must have with one input layer (which receives the content of the environment observation channel, as specified by obsInfo) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by actInfo.

Define the network as an array of layer objects.

```
actorNet = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    lstmLayer(32)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert to a `dlnetwork` object and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 4.5k

   Inputs:
      1   'sequenceinput'   Sequence input with 2 dimensions
```

Create the actor using `actorNet` and the observation and action specifications.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0]}
```

Create a PG agent using the actor and the baseline network.

```
agent = rlPGAgent(actor,baseline)
```

```
agent = 
  rlPGAgent with properties:

            AgentOptions: [1x1 rl.option.rlPGAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Check the agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0]}
```

To evaluate the agent using sequential observations, use the sequence length (time) dimension. For example, obtain actions for a sequence of 9 observations.

```
[action,state] = getAction(agent, ...
    {rand([obsInfo.Dimension 1 9])});
```

Display the action corresponding to the seventh element of the observation.

```
action = action{1};
action(1,1,1,7)
```

```
ans = 0
```

You can now test and train the agent within the environment.

Tips

- For continuous action spaces, the `rlPGAgent` agent does not enforce the constraints set by the action specification, so you must enforce action space constraints within the environment.

Version History

Introduced in R2019a

See Also

Apps

Reinforcement Learning Designer | **Deep Network Designer**

Functions

`getAction` | `getActor` | `getCritic` | `getModel` | `generatePolicyFunction` | `generatePolicyBlock` | `getActionInfo` | `getObservationInfo`

Objects

`rlPGAgentOptions` | `rlAgentInitializationOptions` | `rlQValueFunction` | `rlDiscreteCategoricalActor` | `rlContinuousGaussianActor` | `rlACAgent` | `rlDDPGAgent`

Blocks

RL Agent | Policy

Topics

"Train PG Agent to Balance Cart-Pole System"
"Train Reinforcement Learning Agents"
"Policy Gradient (PG) Agents"
"Reinforcement Learning Agents"

rlPGAgentOptions

Options for PG agent

Description

Use an `rlPGAgentOptions` object to specify options for policy gradient (PG) agents. To create a PG agent, use `rlPGAgent`

For more information on PG agents, see “Policy Gradient (PG) Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlPGAgentOptions
opt = rlPGAgentOptions(Name,Value)
```

Description

`opt = rlPGAgentOptions` creates an `rlPGAgentOptions` object for use as an argument when creating a PG agent using all default settings. You can modify the object properties using dot notation.

`opt = rlPGAgentOptions(Name,Value)` sets option properties on page 3-272 using name-value pairs. For example, `rlPGAgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`UseBaseline` — Use baseline for learning
`true` (default) | `false`

Option to use baseline for learning, specified as a logical value. When `UseBaseline` is `true`, you must specify a critic network as the baseline function approximator.

In general, for simpler problems with smaller actor networks, PG agents work better without a baseline.

`EntropyLossWeight` — Entropy loss weight
`0` (default) | scalar value between `0` and `1`

Entropy loss weight, specified as a scalar value between `0` and `1`. A higher entropy loss weight value promotes agent exploration by applying a penalty for being too certain about which action to take. Doing so can help the agent move out of local optima.

When gradients are computed during training, an additional gradient component is computed for minimizing this loss function.

`ActorOptimizerOptions` — Actor optimizer options

`rlOptimizerOptions` object

Actor optimizer options, specified as an `rlOptimizerOptions` object. It allows you to specify training parameters of the actor approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see `rlOptimizerOptions` and `rlOptimizer`.

`CriticOptimizerOptions` — Critic optimizer options

`rlOptimizerOptions` object

Critic optimizer options, specified as an `rlOptimizerOptions` object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see `rlOptimizerOptions` and `rlOptimizer`.

`SampleTime` — Sample time of agent

`1` (default) | positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

`DiscountFactor` — Discount factor

`0.99` (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlPGAgent	Policy gradient (PG) reinforcement learning agent

Examples

Create PG Agent Options Object

This example shows how to create and modify a PG agent options object.

Create a PG agent options object, specifying the discount factor.

```
opt = rlPGAgentOptions(DiscountFactor=0.9)
opt = 
  rlPGAgentOptions with properties:

                UseBaseline: 1
          EntropyLossWeight: 0
      ActorOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
     CriticOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
                 SampleTime: 1
             DiscountFactor: 0.9000
                 InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

Version History

Introduced in R2019a

R2022a: Simulation and deployment: `UseDeterministicExploitation` will be removed

Warns starting in R2022a

The property `UseDeterministicExploitation` of the `rlPGAgentOptions` object will be removed in a future release. Use the `UseExplorationPolicy` property of `rlPGAgent` instead.

Previously, you set `UseDeterministicExploitation` as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.AgentOptions.UseDeterministicExploitation = true;
  ```
- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.AgentOptions.UseDeterministicExploitation = false;
  ```

Starting in R2022a, set `UseExplorationPolicy` as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.UseExplorationPolicy = false;
  ```
- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.UseExplorationPolicy = true;
  ```

Similarly to `UseDeterministicExploitation`, `UseExplorationPolicy` affects only simulation and deployment; it does not affect training.

See Also

Objects

`rlPGAgent` | `rlAgentInitializationOptions`

Topics

“Train PG Agent to Balance Cart-Pole System”
“Policy Gradient (PG) Agents”

rlPPOAgent

Proximal policy optimization (PPO) reinforcement learning agent

Description

Proximal policy optimization (PPO) is a model-free, online, on-policy, policy gradient reinforcement learning method. This algorithm alternates between sampling data through environmental interaction and optimizing a clipped surrogate objective function using stochastic gradient descent. The action space can be either discrete or continuous. For continuous action spaces, this agent does not enforce constraints set in the action specification; therefore, if you need to enforce action constraints, you must do so within the environment.

For more information on PPO agents, see “Proximal Policy Optimization (PPO) Agents”. For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlPPOAgent(observationInfo,actionInfo)
agent = rlPPOAgent(observationInfo,actionInfo,initOpts)

agent = rlPPOAgent(actor,critic)

agent = rlPPOAgent(___,agentOptions)
```

Description

Create Agent from Observation and Action Specifications

`agent = rlPPOAgent(observationInfo,actionInfo)` creates a proximal policy optimization (PPO) agent for an environment with the given observation and action specifications, using default initialization options. The actor and critic in the agent use default deep neural networks built from the observation specification `observationInfo` and the action specification `actionInfo`. The `ObservationInfo` and `ActionInfo` properties of `agent` are set to the `observationInfo` and `actionInfo` input arguments, respectively.

`agent = rlPPOAgent(observationInfo,actionInfo,initOpts)` creates a PPO agent for an environment with the given observation and action specifications. The agent uses default networks configured using options specified in the `initOpts` object. Actor-critic agents do not support recurrent neural networks. For more information on the initialization options, see `rlAgentInitializationOptions`.

Create Agent from Actor and Critic

`agent = rlPPOAgent(actor,critic)` creates a PPO agent with the specified actor and critic, using the default options for the agent.

Specify Agent Options

`agent = rlPPOAgent(___,agentOptions)` creates a PPO agent and sets the AgentOptions property to the `agentOptions` input argument. Use this syntax after any of the input arguments in the previous syntaxes.

Input Arguments

`initOpts` — Agent initialization options
`rlAgentInitializationOptions` object

Agent initialization options, specified as an `rlAgentInitializationOptions` object.

`actor` — Actor
`rlDiscreteCategoricalActor` object | `rlContinuousGaussianActor` object

Actor that implements the policy, specified as an `rlDiscreteCategoricalActor` or `rlContinuousGaussianActor` function approximator object. For more information on creating actor approximators, see "Create Policies and Value Functions".

`critic` — Critic
`rlValueFunction` object

Critic that estimates the discounted long-term reward, specified as an `rlValueFunction` object. For more information on creating critic approximators, see "Create Policies and Value Functions".

Your critic can use a recurrent neural network as its function approximator. In this case, your actor must also use a recurrent neural network. For an example, see "Create PPO Agent with Recurrent Neural Networks" on page 3-289.

Properties

`ObservationInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

If you create the agent by specifying an actor and critic, the value of `ObservationInfo` matches the value specified in the actor and critic objects.

You can extract `observationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`ActionInfo` — Action specification
`rlFiniteSetSpec` object | `rlNumericSpec` object

Action specifications, specified either as an `rlFiniteSetSpec` (for discrete action spaces) or `rlNumericSpec` (for continuous action spaces) object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

If you create the agent by specifying an actor and critic, the value of `ActionInfo` matches the value specified in the actor and critic objects.

You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`AgentOptions` — Agent options

`rlPPOAgentOptions` object

Agent options, specified as an `rlPPOAgentOptions` object.

`UseExplorationPolicy` — Option to use exploration policy

`true` (default) | `false`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- `true` — Use the base agent exploration policy when selecting actions in `sim` and `generatePolicyFunction`. In this case, the agent selects its actions by sampling its probability distribution, the policy is therefore stochastic and the agent explores its observation space.
- `false` — Use the base agent greedy policy (the action with maximum likelihood) when selecting actions in `sim` and `generatePolicyFunction`. In this case, the simulated agent and generated policy behave deterministically.

Note This option affects only simulation and deployment; it does not affect training.

`SampleTime` — Sample time of agent

positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations. The value of `SampleTime` matches the value specified in `AgentOptions`.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent
setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent
setCritic	Set critic of reinforcement learning agent

generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create Discrete PPO Agent from Observation and Action Specifications

Create an environment with a discrete action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar with five possible elements (a torque of -2, -1, 0, 1, or 2 Nm applied to a swinging pole).

```
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");
```

Obtain observation and action specifications from the environment.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a PPO agent from the environment observation and action specifications.

```
agent = rlPPOAgent(obsInfo,actInfo);
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[-2]}
```

You can now test and train the agent within the environment. You can also use `getActor` and `getCritic` to extract the actor and critic, respectively, and `getModel` to extract the approximator model (by default a deep neural network) from the actor or critic.

Create Continuous PPO Agent Using Initialization Options

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar representing a torque ranging continuously from -2 to 2 Nm.

```
env = rlPredefinedEnv("SimplePendulumWithImage-Continuous");
```

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create an agent initialization option object, specifying that each hidden fully connected layer in the network must have `128` neurons (instead of the default number, `256`).

```
initOpts = rlAgentInitializationOptions(NumHiddenUnit=128);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a PPO actor-critic agent from the environment observation and action specifications.

```
agent = rlPPOAgent(obsInfo,actInfo,initOpts);
```

Extract the deep neural networks from both the agent actor and critic.

```
actorNet = getModel(getActor(agent));
criticNet = getModel(getCritic(agent));
```

To verify that each hidden fully connected layer has 128 neurons, you can display the layers on the MATLAB® command window,

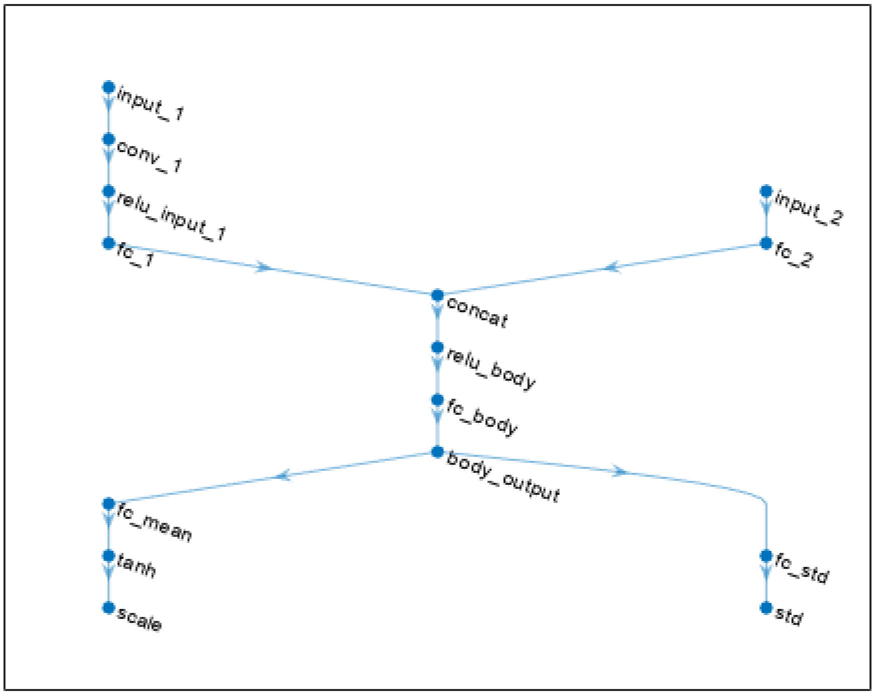
```
criticNet.Layers
```

or visualize the structure interactively using `analyzeNetwork`.

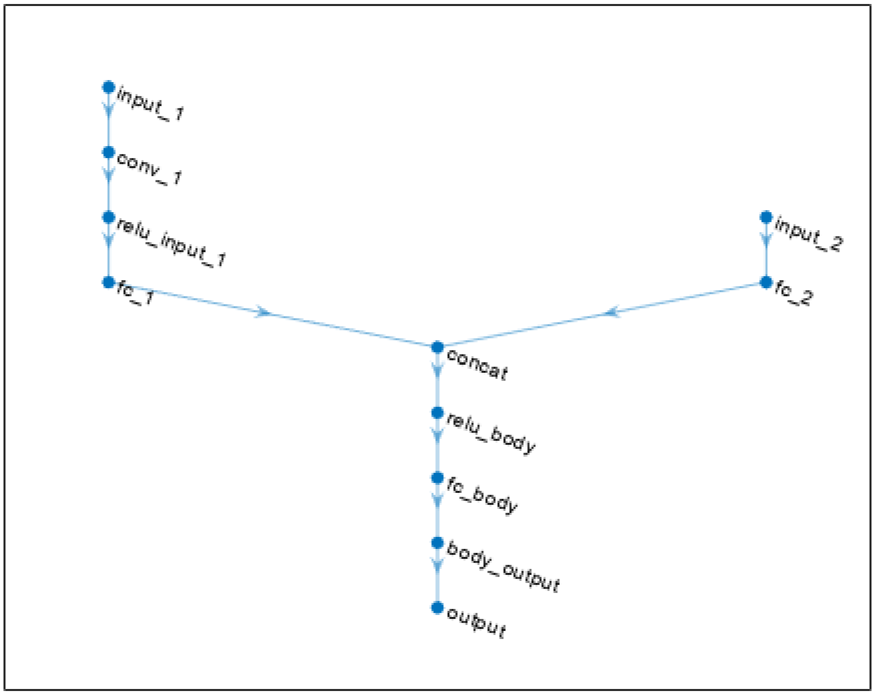
```
analyzeNetwork(criticNet)
```

Plot actor and critic networks

```
plot(layerGraph(actorNet))
```



```
plot(layerGraph(criticNet))
```



To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[0.9228]}
```

You can now test and train the agent within the environment.

Create Proximal Policy Optimization Agent

Create an environment interface, and obtain its observation and action specifications.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

PPO agents use a parametrized value function as a critic. A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (which receives the content of the observation channel, as specified by obs`Info`) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects.

```
criticNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(100)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert to a `dlnetwork` object and display the number of parameters.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 601

   Inputs:
      1   'input'   4 features
```

Create the critic using `criticNet` and the observation specification object. For more information, see `rlValueFunction`.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Check the critic with a random observation input.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.2479
```

Policy gradient agents use a parametrized stochastic policy, which for discrete action spaces is implemented by a discrete categorical actor. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by `actInfo`. Note that `numel(actInfo.Dimension)` returns the number of elements of the discrete action space.

Define the network as an array of layer objects.

```
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(200)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Dimension))
    ];
```

Convert to a `dlnetwork` object and display the number of parameters.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 1.4k

   Inputs:
      1   'input'   4 features
```

Create the actor using `actorNet` and the environment specification objects. For more information, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-10]}
```

Create a PPO agent using the actor and the critic.

```
agent = rlPPOAgent(actor,critic)
```

```
agent = 
  rlPPOAgent with properties:

            AgentOptions: [1x1 rl.option.rlPPOAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Specify agent options, including training options for the actor and the critic.

```
agent.AgentOptions.ExperienceHorizon = 1024;
agent.AgentOptions.DiscountFactor = 0.95;

agent.AgentOptions.CriticOptimizerOptions.LearnRate = 8e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
agent.AgentOptions.ActorOptimizerOptions.LearnRate = 8e-3;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold = 1;
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-10]}
```

You can now test and train the agent against the environment.

Create Continuous PPO Agent

Create an environment with a continuous action space, and obtain its observation and action specifications. For this example, load the double integrator continuous action space environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force, applied to the mass, ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "states"
    Description: "x, dx"
      Dimension: [2 1]
       DataType: "double"
```

```
actInfo = getActionInfo(env)
```

```
actInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "force"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

In this example, the action is a scalar value representing a force ranging from -2 to 2 Newton. To make sure that the output from the agent is in this range, you perform an appropriate scaling operation. Store these limits so you can easily access them later.

```
actInfo.LowerLimit=-2;
actInfo.UpperLimit=2;
```

The actor and critic networks are initialized randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

PPO agents use a parametrized value function as a critic. A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with two input layers (one for each observation channel, as specified by obsInfo) and one output layer (returning the scalar value). Note that `prod(obsInfo.Dimension)` returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects.

```
criticNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(100)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert to a `dlnetwork` object and display the number of parameters.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 401

   Inputs:
      1   'input'   2 features
```

Create the critic approximator object using `criticNet` and the observation specification. For more information, see `rlValueFunction`.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Check the critic with a random observation input.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.0899
```

PPO agents use a parametrized stochastic policy, which for continuous action spaces is implemented by a continuous Gaussian actor. This actor takes an observation as input and returns as output a random action sampled from a Gaussian probability distribution.

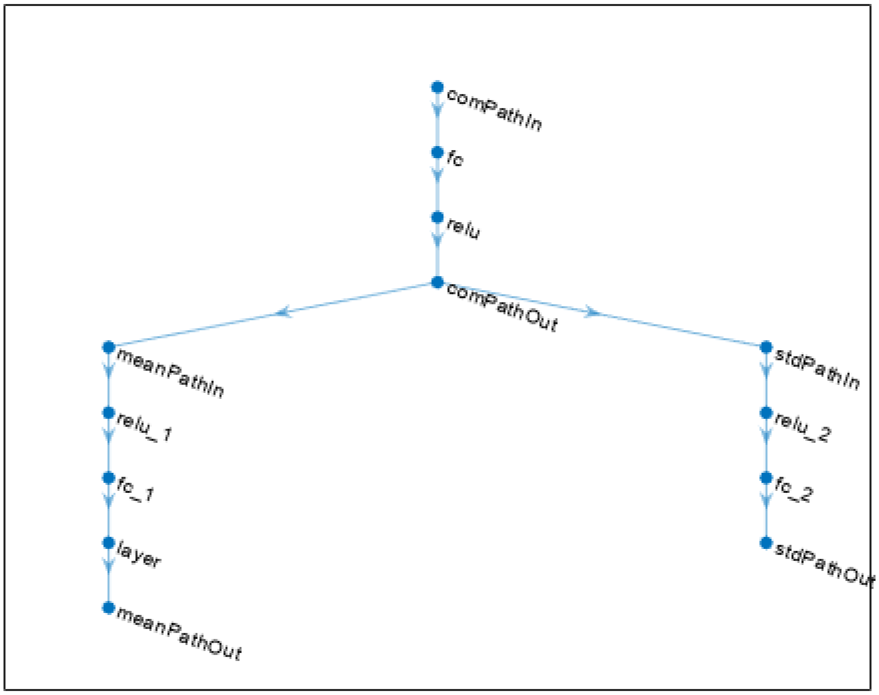
To approximate the mean values and standard deviations of the Gaussian distribution, you must use a neural network with two output layers, each having as many elements as the dimension of the action space. One output layer must return a vector containing the mean values for each action dimension. The other must return a vector containing the standard deviation for each action dimension.

Note that standard deviations must be nonnegative and mean values must fall within the range of the action. Therefore the output layer that returns the standard deviations must be a softplus or ReLU layer, to enforce nonnegativity, while the output layer that returns the mean values must be a scaling layer, to scale the mean values to the output range.

For this example the environment has only one observation channel and therefore the network has only one input layer.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% Define common input path layer
commonPath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="comPathIn")
    fullyConnectedLayer(100)
    reluLayer
    fullyConnectedLayer(1,Name="comPathOut")
    ];

% Define mean value path
meanPath = [
    fullyConnectedLayer(15,Name="meanPathIn")
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension));
    tanhLayer;
    scalingLayer(Name="meanPathOut",Scale=actInfo.UpperLimit)
    ];

% Define standard deviation path
sdevPath = [
    fullyConnectedLayer(15,"Name","stdPathIn")
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension));
    softplusLayer(Name="stdPathOut")
    ];

% Add layers to layerGraph object
actorNet = layerGraph(commonPath);
actorNet = addLayers(actorNet,meanPath);
actorNet = addLayers(actorNet,sdevPath);

% Connect paths
actorNet = connectLayers(actorNet,"comPathOut","meanPathIn/in");
actorNet = connectLayers(actorNet,"comPathOut","stdPathIn/in");

% Plot network
plot(actorNet)
```



```
% Convert to dlnetwork and display number of weights
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 493

   Inputs:
      1   'comPathIn'   2 features
```

Create the actor approximator object using `actorNet`, the environment specifications, the name of the network input layer to be connected with the environment observation channel, and the names of the network output layers that calculate the mean the standard deviation values of the action.

For more information, see `rlContinuousGaussianActor`.

```
actor = rlContinuousGaussianActor(actorNet, obsInfo, actInfo, ...
    "ActionMeanOutputNames","meanPathOut",...
    "ActionStandardDeviationOutputNames","stdPathOut",...
    ObservationInputNames="comPathIn");
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-0.2267]}
```

Create a PPO agent using the actor and the critic.

```
agent = rlPPOAgent(actor,critic)
```

```
agent = 
  rlPPOAgent with properties:

            AgentOptions: [1x1 rl.option.rlPPOAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlNumericSpec]
              SampleTime: 1
```

Specify agent options, including training options for the actor and the critic.

```
agent.AgentOptions.ExperienceHorizon = 1024;
agent.AgentOptions.DiscountFactor = 0.95;

agent.AgentOptions.CriticOptimizerOptions.LearnRate = 8e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
agent.AgentOptions.ActorOptimizerOptions.LearnRate = 8e-3;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold = 1;
```

Specify training options for the critic.

```
criticOpts = rlOptimizerOptions( ...
    LearnRate=8e-3,GradientThreshold=1);
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.2719]}
```

You can now test and train the agent within the environment.

Create PPO Agent with Recurrent Neural Networks

For this example load the predefined environment used for the “Train DQN Agent to Balance Cart-Pole System” example.

```
env = rlPredefinedEnv("CartPole-Discrete");
```

Get observation and action information. This environment has a continuous four-dimensional observation space (the positions and velocities of both cart and pole) and a discrete one-dimensional action space consisting on the application of two possible forces, -10N or 10N.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The actor and critic networks are initialized randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

PPO agents use a parametrized value function as a critic. To model the parametrized value function within the critic, use a recurrent neural network.

Define the network as an array of layer objects. To create a recurrent neural network, use a `sequenceInputLayer` as the input layer and include at least one `lstmLayer`.

```
criticNet = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(8)
    reluLayer
    lstmLayer(8)
    fullyConnectedLayer(1)
    ];
```

Convert to a `dlnetwork` object and display the number of learnable parameters.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 593

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create the critic using `criticNet` and the observation specification object. For more information, see `rlValueFunction`.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Check the critic with a random observation input.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = single
    0.0017
```

Since the critic has a recurrent network, the actor must have a recurrent network too. Define the network as an array of layer objects.

```
actorNet = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(100)
    reluLayer
    lstmLayer(8)
    fullyConnectedLayer(numel(actInfo.Elements))
    softmaxLayer
    ];
```

Convert the network to a `dlnetwork` object and display the number of learnable parameters.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 4k

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create the actor using `actorNet` and the environment specification objects. For more information, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-10]}
```

Set some training option for the critic.

```
criticOptions = rlOptimizerOptions( ...
    LearnRate=1e-2, ...
    GradientThreshold=1);
```

Set some training options for the actor.

```
actorOptions = rlOptimizerOptions( ...
    LearnRate=1e-3, ...
    GradientThreshold=1);
```

Create the agent options object.

```
agentOptions = rlPPOAgentOptions(...
    AdvantageEstimateMethod="finite-horizon", ...
    ClipFactor=0.1, ...
    CriticOptimizerOptions=criticOptions, ...
    ActorOptimizerOptions=actorOptions);
```

When recurrent neural networks are used, the `MiniBatchSize` property is the length of the learning trajectory.

```
agentOptions.MiniBatchSize
```

```
ans = 128
```

Create the agent using the actor and critic, as well as the agent options object.

```
agent = rlPPOAgent(actor,critic,agentOptions)
```

```
agent = 
  rlPPOAgent with properties:

            AgentOptions: [1x1 rl.option.rlPPOAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Check your agent with a random observation input.

```
getAction(agent,rand(obsInfo.Dimension))
```

```
ans = 1x1 cell array
    {[-10]}
```

To evaluate the agent using sequential observations, use the sequence length (time) dimension. For example, obtain actions for a sequence of 9 observations.

```
[action,state] = getAction(agent, ...
    {rand([obsInfo.Dimension 1 9])});
```

Display the action corresponding to the seventh element of the observation.

```
action = action{1};
action(1,1,1,7)
```

```
ans = -10
```

You can now test and train the agent within the environment.

Tips

- For continuous action spaces, this agent does not enforce the constraints set by the action specification. In this case, you must enforce action space constraints within the environment.
- While tuning the learning rate of the actor network is necessary for PPO agents, it is not necessary for TRPO agents.

Version History

Introduced in R2019b

See Also

Apps

Reinforcement Learning Designer | **Deep Network Designer**

Functions

getAction | getActor | getCritic | getModel | generatePolicyFunction | generatePolicyBlock | getActionInfo | getObservationInfo

Objects

rlPPOAgentOptions | rlAgentInitializationOptions | rlValueFunction | rlDiscreteCategoricalActor | rlContinuousGaussianActor | rlPGAgent | rlACAgent | rlTRPOAgent

Blocks

RL Agent | Policy

Topics

"Train PPO Agent for a Lander Vehicle"

“Train Reinforcement Learning Agents”
“Proximal Policy Optimization (PPO) Agents”
“Reinforcement Learning Agents”

rlPPOAgentOptions

Options for PPO agent

Description

Use an `rlPPOAgentOptions` object to specify options for proximal policy optimization (PPO) agents. To create a PPO agent, use `rlPPOAgent`.

For more information on PPO agents, see “Proximal Policy Optimization (PPO) Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlPPOAgentOptions
opt = rlPPOAgentOptions(Name,Value)
```

Description

`opt = rlPPOAgentOptions` creates an `rlPPOAgentOptions` object for use as an argument when creating a PPO agent using all default settings. You can modify the object properties using dot notation.

`opt = rlPPOAgentOptions(Name,Value)` sets option properties on page 3-294 using name-value arguments. For example, `rlPPOAgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value arguments. Enclose each property name in quotes.

Properties

`ExperienceHorizon` — Number of steps the agent interacts with the environment before learning
512 (default) | positive integer

Number of steps the agent interacts with the environment before learning from its experience, specified as a positive integer. When the agent is trained in parallel, `ExperienceHorizon` is ignored, and the whole episode is used to compute the gradients.

The `ExperienceHorizon` value must be greater than or equal to the `MiniBatchSize` value.

`MiniBatchSize` — Mini-batch size
128 (default) | positive integer

Mini-batch size used for each learning epoch, specified as a positive integer. When the agent uses a recurrent neural network, `MiniBatchSize` is treated as the training trajectory length.

The MiniBatchSize value must be less than or equal to the ExperienceHorizon value.

ClipFactor — Clip factor

0.2 (default) | positive scalar less than 1

Clip factor for limiting the change in each policy update step, specified as a positive scalar less than 1.

EntropyLossWeight — Entropy loss weight

0.01 (default) | scalar value between 0 and 1

Entropy loss weight, specified as a scalar value between 0 and 1. A higher entropy loss weight value promotes agent exploration by applying a penalty for being too certain about which action to take. Doing so can help the agent move out of local optima.

When gradients are computed during training, an additional gradient component is computed for minimizing this loss function. For more information, see “Entropy Loss”.

NumEpoch — Number of epochs

3 (default) | positive integer

Number of epochs for which the actor and critic networks learn from the current experience set, specified as a positive integer.

AdvantageEstimateMethod — Method for estimating advantage values

"gae" (default) | "finite-horizon"

Method for estimating advantage values, specified as one of the following:

- "gae" — Generalized advantage estimator
- "finite-horizon" — Finite horizon estimation

For more information on these methods, see the training algorithm information in “Proximal Policy Optimization (PPO) Agents”.

GAEFactor — Smoothing factor for generalized advantage estimator

0.95 (default) | scalar value between 0 and 1

Smoothing factor for generalized advantage estimator, specified as a scalar value between 0 and 1, inclusive. This option applies only when the AdvantageEstimateMethod option is "gae"

NormalizedAdvantageMethod — Method for normalizing advantage function

"none" (default) | "current | "moving"

Method for normalizing advantage function values, specified as one of the following:

- "none" — Do not normalize advantage values
- "current" — Normalize the advantage function using the mean and standard deviation for the current mini-batch of experiences.
- "moving" — Normalize the advantage function using the mean and standard deviation for a moving window of recent experiences. To specify the window size, set the AdvantageNormalizingWindow option.

In some environments, you can improve agent performance by normalizing the advantage function during training. The agent normalizes the advantage function by subtracting the mean advantage value and scaling by the standard deviation.

`AdvantageNormalizingWindow` — Window size for normalizing advantage function
`1e6` (default) | positive integer

Window size for normalizing advantage function values, specified as a positive integer. Use this option when the `NormalizedAdvantageMethod` option is `"moving"`.

`ActorOptimizerOptions` — Actor optimizer options
`rlOptimizerOptions` object

Actor optimizer options, specified as an `rlOptimizerOptions` object. It allows you to specify training parameters of the actor approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see `rlOptimizerOptions` and `rlOptimizer`.

`CriticOptimizerOptions` — Critic optimizer options
`rlOptimizerOptions` object

Critic optimizer options, specified as an `rlOptimizerOptions` object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see `rlOptimizerOptions` and `rlOptimizer`.

`SampleTime` — Sample time of agent
`1` (default) | positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

`DiscountFactor` — Discount factor
`0.99` (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlPPOAgent	Proximal policy optimization (PPO) reinforcement learning agent

Examples

Create PPO Agent Options Object

Create a PPO agent options object, specifying the experience horizon.

```
opt = rlPPOAgentOptions(ExperienceHorizon=256)
```

```
opt = 
  rlPPOAgentOptions with properties:

             ExperienceHorizon: 256
                 MiniBatchSize: 128
                    ClipFactor: 0.2000
             EntropyLossWeight: 0.0100
                      NumEpoch: 3
       AdvantageEstimateMethod: "gae"
                     GAEFactor: 0.9500
     NormalizedAdvantageMethod: "none"
    AdvantageNormalizingWindow: 1000000
         ActorOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
        CriticOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
                    SampleTime: 1
                DiscountFactor: 0.9900
                    InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

Version History

Introduced in R2019b

R2022a: Simulation and deployment: `UseDeterministicExploitation` will be removed

Warns starting in R2022a

The property `UseDeterministicExploitation` of the `rlPPOAgentOptions` object will be removed in a future release. Use the `UseExplorationPolicy` property of `rlPPOAgent` instead.

Previously, you set `UseDeterministicExploitation` as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.AgentOptions.UseDeterministicExploitation = true;
  ```
- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.AgentOptions.UseDeterministicExploitation = false;
  ```

Starting in R2022a, set `UseExplorationPolicy` as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.UseExplorationPolicy = false;
  ```

- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.UseExplorationPolicy = true;
  ```

Similarly to `UseDeterministicExploitation`, `UseExplorationPolicy` affects only simulation and deployment; it does not affect training.

See Also

Objects
`rlPPOAgent` | `rlAgentInitializationOptions`

Topics
“Train PPO Agent for a Lander Vehicle”
“Proximal Policy Optimization (PPO) Agents”

rlPrioritizedReplayMemory

Replay memory experience buffer with prioritized sampling

Description

An off-policy reinforcement learning agent stores experiences in a circular experience buffer. During training, the agent samples mini-batches of experiences from the buffer and uses these mini-batches to update its actor and critic function approximators.

By default, built-in off-policy agents (DQN, DDPG, TD3, SAC, MBPO) use an `rlReplayMemory` object as their experience buffer. Agents uniformly sample data from this buffer. To perform nonuniform prioritized sampling [1], which can improve sample efficiency when training your agent, use an `rlPrioritizedReplayMemory` object. For more information on prioritized sampling, see “Algorithms” on page 3-302.

For goal-conditioned tasks, you can also replace your experience buffer with one of the following hindsight replay memory objects.

- `rlHindsightReplayMemory` — Uniform sampling of experiences and generation of hindsight experiences by replacing goals with goal measurements
- `rlHindsightPrioritizedReplayMemory` — Prioritized nonuniform sampling of experiences and generation of hindsight experiences

Creation

Syntax

```
buffer = rlPrioritizedReplayMemory(obsInfo,actInfo)
buffer = rlPrioritizedReplayMemory(obsInfo,actInfo,maxLength)
```

Description

`buffer = rlPrioritizedReplayMemory(obsInfo,actInfo)` creates a prioritized replay memory experience buffer that is compatible with the observation and action specifications in `obsInfo` and `actInfo`, respectively.

`buffer = rlPrioritizedReplayMemory(obsInfo,actInfo,maxLength)` sets the maximum length of the buffer by setting the `MaxLength` property.

Input Arguments

`obsInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, specified as a reinforcement learning specification object or an array of specification objects defining properties such as dimensions, data types, and names of the observation signals.

You can extract the observation specifications from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`actInfo` — Action specifications
specification object | array of specification objects

Action specifications, specified as a reinforcement learning specification object defining properties such as dimensions, data types, and names of the action signals.

You can extract the action specifications from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

Properties

`MaxLength` — Maximum buffer length
`10000` (default) | nonnegative integer

This property is read-only.

Maximum buffer length, specified as a nonnegative integer.

To change the maximum buffer length, use the `resize` function.

`Length` — Number of experiences in buffer
`0` (default) | nonnegative integer

This property is read-only.

Number of experiences in buffer, specified as a nonnegative integer.

`PriorityExponent` — Priority exponent
`0.6` (default) | nonnegative scalar less than or equal to 1

Priority exponent to control the impact of prioritization during probability computation, specified as a nonnegative scalar less than or equal to 1.

If the priority exponent is zero, the agent uses uniform sampling.

`InitialImportanceSamplingExponent` — Initial value of importance sampling exponent
`0.4` (default) | nonnegative scalar less than or equal to 1

Initial value of the importance sampling exponent, specified as a nonnegative scalar less than or equal to 1.

`NumAnnealingSteps` — Number of annealing steps
`1000000` (default) | positive integer

Number of annealing steps for updating the importance sampling exponent, specified as a positive integer.

`ImportanceSamplingExponent` — Current value of importance sampling exponent
`0.4` (default) | nonnegative scalar less than or equal to 1

This property is read-only.

Current value of the importance sampling exponent, specified as a nonnegative scalar less than or equal to 1.

During training, `ImportanceSamplingExponent` is linearly increased from `InitialImportanceSamplingExponent` to 1 over `NumAnnealingSteps` steps.

Object Functions

append	Append experiences to replay memory buffer
sample	Sample experiences from replay memory buffer
resize	Resize replay memory experience buffer
reset	Reset environment, agent, experience buffer, or policy object
allExperiences	Return all experiences in replay memory buffer
validateExperience	Validate experiences for replay memory
getActionInfo	Obtain action data specifications from reinforcement learning environment, agent, or experience buffer
getObservationInfo	Obtain observation data specifications from reinforcement learning environment, agent, or experience buffer

Examples

Create DQN Agent With Prioritized Replay Memory

Create an environment for training the agent. For this example, load a predefined environment.

```
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");
```

Extract the observation and action specifications from the agent.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a DQN agent from the environment specifications.

```
agent = rlDQNAgent(obsInfo,actInfo);
```

By default, the agent uses a replay memory experience buffer with uniform sampling.

Replace the default experience buffer with a prioritized replay memory buffer.

```
agent.ExperienceBuffer = rlPrioritizedReplayMemory(obsInfo,actInfo);
```

Configure the prioritized replay memory options. For example, set the initial importance sampling exponent to `0.5` and the number of annealing steps for updating the exponent during training to `1e4`.

```
agent.ExperienceBuffer.NumAnnealingSteps = 1e4;
agent.ExperienceBuffer.PriorityExponent = 0.5;
agent.ExperienceBuffer.InitialImportanceSamplingExponent = 0.5;
```

Limitations

- Prioritized experience replay does not support agents that use recurrent neural networks.

Algorithms

Prioritized Sampling

Prioritized replay memory samples experiences according to experience priorities. For a given experience, the priority is defined as the absolute value of the associated temporal difference (TD) error. A larger TD error indicates that the critic network is not well-trained for the corresponding experience. Therefore, sampling such experiences during critic updates can help efficiently improve the critic performance, which often improves the sample efficiency of agent training.

When using prioritized replay memory, agents use the following process when sampling a mini-batch of experiences and updating a critic.

1. Compute the sampling probability P for each experience in the buffer based on the experience priority.

   $$P(j) = \frac{p(j)^{\alpha}}{\sum_{i=1}^{N} p(i)^{\alpha}}$$

   Here:

   - N is the number of experiences in the replay memory buffer.
   - p is the experience priority.
   - α is a priority exponent. To set α, use the `PriorityExponent` parameter.
2. Sample a mini-batch of experiences according to the computed probabilities.
3. Compute the importance sampling weights (w) for the sampled experiences.

   $$w'(j) = (N \cdot P(j))^{-\beta}$$
   $$w(j) \leftarrow \frac{w'(j)}{\max\limits_{i \in \text{mini-batch}} w'(i)}$$

   Here, β is the importance sampling exponent. The `ImportanceSamplingExponent` parameter contains the current value of β. To control β, set the `ImportanceSamplingExponent` and `NumAnnealingSteps` parameters.
4. Compute the weighted loss using the importance sampling weights w and the TD error δ to update a critic.
5. Update the priorities of the sampled experiences based on the TD error.

   $$p(j) = |\delta|$$
6. Update the importance sampling exponent β by linearly annealing the exponent value until it reaches 1.

   $$\beta \leftarrow \beta + \frac{1 - \beta_0}{N_S}$$

   Here:

- β_0 is the initial importance sampling exponent. To specify β_0, use the `InitialImportanceSamplingExponent` parameter.
- N_S is the number of annealing steps. To specify N_s, use the `NumAnnealingSteps` parameter.

Version History

Introduced in R2022b

References

[1] Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver. 'Prioritized experience replay'. *arXiv:1511.05952 [Cs]* 25 February 2016. https://arxiv.org/abs/1511.05952.

See Also

Objects

`rlReplayMemory` | `rlHindsightReplayMemory` | `rlHindsightPrioritizedReplayMemory`

rlQAgent

Q-learning reinforcement learning agent

Description

The Q-learning algorithm is a model-free, online, off-policy reinforcement learning method. A Q-learning agent is a value-based reinforcement learning agent which trains a critic to estimate the return or future rewards.

Note Q-learning agents do not support recurrent networks.

For more information on Q-learning agents, see “Q-Learning Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

`agent = rlQAgent(critic,agentOptions)`

Description

`agent = rlQAgent(critic,agentOptions)` creates a Q-learning agent with the specified critic network and sets the `AgentOptions` property.

Input Arguments

`critic` — Critic
`rlQValueFunction` object

Critic, specified as an `rlQValueFunction` object. For more information on creating critics, see “Create Policies and Value Functions”.

Properties

`AgentOptions` — Agent options
`rlQAgentOptions` object

Agent options, specified as an `rlQAgentOptions` object.

`UseExplorationPolicy` — Option to use exploration policy
`false` (default) | `true`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- `false` — Use the agent greedy policy when selecting actions.
- `true` — Use the agent exploration policy when selecting actions.

`ObservationInfo` — Observation specifications

specification object | array of specification objects

This property is read-only.

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

The value of `ObservationInfo` matches the corresponding value specified in `critic`.

`ActionInfo` — Action specification

`rlFiniteSetSpec` object

This property is read-only.

Action specifications, specified as an `rlFiniteSetSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

The value of `ActionInfo` matches the corresponding value specified in `critic`.

`SampleTime` — Sample time of agent

positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations. The value of `SampleTime` matches the value specified in `AgentOptions`.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent
setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent
setCritic	Set critic of reinforcement learning agent

generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create a Q-Learning Agent

Create an environment interface. For this example, use the same environment as in the example “Train Reinforcement Learning Agent in Basic Grid World”.

```
env = rlPredefinedEnv("BasicGridWorld");
```

Get observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Q-learning agents use a parametrized Q-value function to estimate the value of the policy. A Q-value function takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

Since both observation and action spaces are discrete and low-dimensional, use a table to model the Q-value function within the critic. `rlTable` creates a value table object from the observation and action specifications objects.

Create a table approximation model derived from the environment observation and action specifications.

```
qTable = rlTable(obsInfo,actInfo);
```

Create the Q-value function approximator object using `qTable` and the environment specification objects. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(qTable,obsInfo,actInfo);
```

Create a Q-learning agent using the approximator object.

```
agent = rlQAgent(critic)
```

```
agent = 
  rlQAgent with properties:

            AgentOptions: [1x1 rl.option.rlQAgentOptions]
    UseExplorationPolicy: 0
         ObservationInfo: [1x1 rl.util.rlFiniteSetSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Specify an `Epsilon` value of 0.05.

```
opt = rlSARSAAgentOptions;
agent.AgentOptions.EpsilonGreedyExploration.Epsilon = 0.05;
```

To check your agent, use `getAction` to return the action from a random observation.

```
act = getAction(agent,{randi(numel(obsInfo.Elements))});
act{1}
```

```
ans = 1
```

You can now test and train the agent against the environment.

Version History

Introduced in R2019a

See Also

Apps
Reinforcement Learning Designer | Deep Network Designer

Functions
`getAction` | `getActor` | `getCritic` | `getModel` | `generatePolicyFunction` | `generatePolicyBlock` | `getActionInfo` | `getObservationInfo`

Objects
`rlQAgentOptions` | `rlAgentInitializationOptions` | `rlQValueFunction`

Blocks
RL Agent | Policy

Topics
"Train Reinforcement Learning Agent in Basic Grid World"
"Train Reinforcement Learning Agent in MDP Environment"
"Train Reinforcement Learning Agents"
"Q-Learning Agents"
"Reinforcement Learning Agents"

rlQAgentOptions

Options for Q-learning agent

Description

Use an `rlQAgentOptions` object to specify options for creating Q-learning agents. To create a Q-learning agent, use `rlQAgent`

For more information on Q-learning agents, see “Q-Learning Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlQAgentOptions
opt = rlQAgentOptions(Name,Value)
```

Description

`opt = rlQAgentOptions` creates an `rlQAgentOptions` object for use as an argument when creating a Q-learning agent using all default settings. You can modify the object properties using dot notation.

`opt = rlQAgentOptions(Name,Value)`sets option properties on page 3-308 using name-value pairs. For example, `rlQAgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`EpsilonGreedyExploration` — Options for epsilon-greedy exploration
`EpsilonGreedyExploration` object

Options for epsilon-greedy exploration, specified as an `EpsilonGreedyExploration` object with the following properties.

Property	Description	Default Value
Epsilon	Probability threshold to either randomly select an action or select the action that maximizes the state-action value function. A larger value of Epsilon means that the agent randomly explores the action space at a higher rate.	1
EpsilonMin	Minimum value of Epsilon	0.01
EpsilonDecay	Decay rate	0.0050

At the end of each training time step, if Epsilon is greater than EpsilonMin, then it is updated using the following formula.

```
Epsilon = Epsilon*(1-EpsilonDecay)
```

Note that Epsilon is conserved between the end of an episode and the start of the next one. Therefore, it keeps on uniformly decreasing over multiple episodes until it reaches EpsilonMin.

If your agent converges on local optima too quickly, you can promote agent exploration by increasing Epsilon.

To specify exploration options, use dot notation after creating the rlQAgentOptions object opt. For example, set the epsilon value to 0.9.

```
opt.EpsilonGreedyExploration.Epsilon = 0.9;
```

CriticOptimizerOptions — Critic optimizer options

rlOptimizerOptions object

Critic optimizer options, specified as an rlOptimizerOptions object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see rlOptimizerOptions and rlOptimizer.

SampleTime — Sample time of agent

1 (default) | positive scalar | -1

Sample time of agent, specified as a positive scalar or as -1. Setting this parameter to -1 allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every SampleTime seconds of simulation time. If SampleTime is -1, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, SampleTime is the time interval between consecutive elements in the output experience returned by sim or train. If SampleTime is -1, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

DiscountFactor — Discount factor

0.99 (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlQAgent Q-learning reinforcement learning agent

Examples

Create Q-Learning Agent Options Object

Create a `rlQAgentOptions` object that specifies the agent sample time.

```
opt = rlQAgentOptions(SampleTime=0.5)
```

```
opt = 
  rlQAgentOptions with properties:

    EpsilonGreedyExploration: [1x1 rl.option.EpsilonGreedyExploration]
      CriticOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
                  SampleTime: 0.5000
              DiscountFactor: 0.9900
                  InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent discount factor to `0.95`.

```
opt.DiscountFactor = 0.95;
```

Version History

Introduced in R2019a

See Also

Objects

`rlDQNAgent` | `rlAgentInitializationOptions`

Topics

"Train Reinforcement Learning Agent in MDP Environment"
"Q-Learning Agents"

rlQValueFunction

Q-Value function approximator object for reinforcement learning agents

Description

This object implements a Q-value function approximator that you can use as a critic for a reinforcement learning agent. A Q-value function is a mapping from an environment observation-action pair to the value of a policy. Specifically, its output is a scalar that represents the expected discounted cumulative long-term reward when an agent starts from the state corresponding to the given observation, executes the given action, and keeps on taking actions according to the given policy afterwards. A Q-value function critic therefore needs both the environment state and an action as inputs. After you create an `rlQValueFunction` critic, use it to create an agent such as `rlQAgent`, `rlDQNAgent`, `rlSARSAAgent`, `rlDDPGAgent`, or `rlTD3Agent`. For more information on creating representations, see “Create Policies and Value Functions”.

Creation

Syntax

`critic = rlQValueFunction(net,observationInfo,actionInfo)`

`critic = rlQValueFunction(tab,observationInfo,actionInfo)`

`critic = rlQValueFunction({basisFcn,W0},observationInfo,actionInfo)`

`critic = rlQValueFunction( ___ ,Name=Value)`

Description

`critic = rlQValueFunction(net,observationInfo,actionInfo)` creates the Q-value function object `critic`. Here, `net` is the deep neural network used as an approximation model, and it must have both observation and action as inputs and a single scalar output. The network input layers are automatically associated with the environment observation and action channels according to the dimension specifications in `observationInfo` and `actionInfo`. This function sets the `ObservationInfo` and `ActionInfo` properties of `critic` to the `observationInfo` and `actionInfo` input arguments, respectively.

`critic = rlQValueFunction(tab,observationInfo,actionInfo)` creates the Q-value function object `critic` with *discrete action and observation spaces* from the Q-value table `tab`. `tab` is a `rlTable` object containing a table with as many rows as the number of possible observations and as many columns as the number of possible actions. The function sets the ObservationInfo and ActionInfo properties of `critic` respectively to the `observationInfo` and `actionInfo` input arguments, which in this case must be scalar `rlFiniteSetSpec` objects.

`critic = rlQValueFunction({basisFcn,W0},observationInfo,actionInfo)` creates a Q-value function object `critic` using a custom basis function as underlying approximator. The first input argument is a two-element cell array whose first element is the handle `basisFcn` to a custom basis function and whose second element is the initial weight vector `W0`. Here the basis function must have both observation and action as inputs and W0 must be a column vector. The function sets the ObservationInfo and ActionInfo properties of critic to the observationInfo and actionInfo input arguments, respectively.

critic = rlQValueFunction(___ ,Name=Value) specifies one or more name-value arguments. You can specify the input and output layer names (to mandate their association with the environment observation and action channels) for deep neural network approximators. For all types of approximators, you can specify the computation device, for example UseDevice="gpu".

Input Arguments

net — Deep neural network
array of Layer objects | layerGraph object | DAGNetwork object | SeriesNetwork object | dlNetwork object (preferred)

Deep neural network used as the underlying approximator within the critic, specified as one of the following:

- Array of Layer objects
- layerGraph object
- DAGNetwork object
- SeriesNetwork object
- dlnetwork object

The network must have as many input layers as the number of environment observation channels plus one. Specifically, there must be one input layer for each observation channel, and one additional input layer for the action channel. The network must have a single output layer returning a scalar value

Note Among the different network representation options, dlnetwork is preferred, since it has built-in validation checks and supports automatic differentiation. If you pass another network object as an input argument, it is internally converted to a dlnetwork object. However, best practice is to convert other representations to dlnetwork explicitly *before* using it to create a critic or an actor for a reinforcement learning agent. You can do so using dlnet=dlnetwork(net), where net is any Deep Learning Toolbox neural network object. The resulting dlnet is the dlnetwork object that you use for your critic or actor. This practice allows a greater level of insight and control for cases in which the conversion is not straightforward and might require additional specifications.

rlQValueFunction objects support recurrent deep neural networks.

The learnable parameters of the critic are the weights of the deep neural network. For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

tab — Q-value table
rlTable object

Q-value table, specified as an rlTable object containing an array with as many rows as the possible observations and as many columns as the possible actions. The element (s,a) is the expected cumulative long-term reward for taking action a from observed state s. The elements of this array are the learnable parameters of the critic.

`basisFcn` — Custom basis function

function handle

Custom basis function, specified as a function handle to a user-defined MATLAB function. The user defined function can either be an anonymous function or a function on the MATLAB path. The output of the critic is the scalar `c = W'*B`, where `W` is a weight vector containing the learnable parameters, and `B` is the column vector returned by the custom basis function.

Your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN,act)
```

Here, `obs1` to `obsN` are inputs in the same order and with the same data type and dimensions as the environment observation channels defined in `observationInfo` and `act` is an input with the same data type and dimension as the environment action channel defined in `actionInfo`.

For an example on how to use a basis function to create a Q-value function critic with a mixed continuous and discrete observation space, see “Create Mixed Observation Space Q-Value Function Critic from Custom Basis Function” on page 3-324.

Example: `@(obs1,obs2,act) [act(2)*obs1(1)^2; abs(obs2(5)+act(1))]`

`W0` — Initial value of the basis function weights

column vector

Initial value of the basis function weights `W`, specified as a column vector having the same length as the vector returned by the basis function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `UseDevice="gpu"`

`ActionInputNames` — Network input layers name corresponding to the environment action channel

string array | cell array of character vectors

Network input layers name corresponding to the environment action channel, specified as a string array or a cell array of character vectors. The function assigns the environment action channel specified in `actionInfo` to the specified network input layer. Therefore, the specified network input layer must have the same data type and dimensions of the action channel, as defined in `actionInfo`.

Note The function does not use the name or the description (if any) of the action channel specified in `actionInfo`.

This name-value argument is supported only when the approximation model is a deep neural network.

Example: `ActionInputNames="myNetOutput_Force"`

`ObservationInputNames` — Network input layers names corresponding to the environment observation channels

string array | cell array of character vectors

Network input layers names corresponding to the environment observation channels, specified as a string array or a cell array of character vectors. The function assigns, in sequential order, each environment observation channel specified in `observationInfo` to each specified network input layer. Therefore, the specified network input layers, ordered as indicated in this argument, must have the same data type and dimensions as the observation channels, as ordered in `observationInfo`.

This name-value argument is supported only when the approximation model is a deep neural network.

Example: `ObservationInputNames={"NetInput1_airspeed","NetInput2_altitude"}`

Properties

`ObservationInfo` — Observation specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

`rlQValueFucntion` sets the `ObservationInfo` property of `critic` to the input argument `observationInfo`.

You can extract `ObservationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually.

`ActionInfo` — Action specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object

Action specifications, specified either as an `rlFiniteSetSpec` (for discrete action spaces) or `rlNumericSpec` (for continuous action spaces) object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

`rlQValueRepresentation` sets the `ActionInfo` property of `critic` to the input `actionInfo`.

You can extract `ActionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specifications manually.

`UseDevice` — Computation device used for training and simulation

`"cpu"` (default) | `"gpu"`

Computation device used to perform operations such as gradient computation, parameter update and prediction during training and simulation, specified as either `"cpu"` or `"gpu"`.

The `"gpu"` option requires both Parallel Computing Toolbox software and a CUDA enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use `gpuDevice` (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating an agent on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

To speed up training by using parallel processing over multiple cores, you do not need to use this argument. Instead, when training your agent, use an `rlTrainingOptions` object in which the `UseParallel` option is set to `true`. For more information about training using multicore processors and GPUs for training, see “Train Agents Using Parallel Computing and GPUs”.

Example: `"gpu"`

Object Functions

rlDDPGAgent	Deep deterministic policy gradient (DDPG) reinforcement learning agent
rlTD3Agent	Twin-delayed deep deterministic (TD3) policy gradient reinforcement learning agent
rlDQNAgent	Deep Q-network (DQN) reinforcement learning agent
rlQAgent	Q-learning reinforcement learning agent
rlSARSAAgent	SARSA reinforcement learning agent
rlSACAgent	Soft actor-critic (SAC) reinforcement learning agent
getValue	Obtain estimated value from a critic given environment observations and actions
getMaxQValue	Obtain maximum estimated value over all possible actions from a Q-value function critic with discrete action space, given environment observations
evaluate	Evaluate function approximator object given observation (or observation-action) input data
gradient	Evaluate gradient of function approximator object given observation and action input data
accelerate	Option to accelerate computation of gradient for approximator object based on neural network
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object
setModel	Set function approximation model for actor or critic
getModel	Get function approximator model from actor or critic

Examples

Create Q-Value Function Critic from Deep Neural Network

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that the action channel carries a column vector containing two doubles.

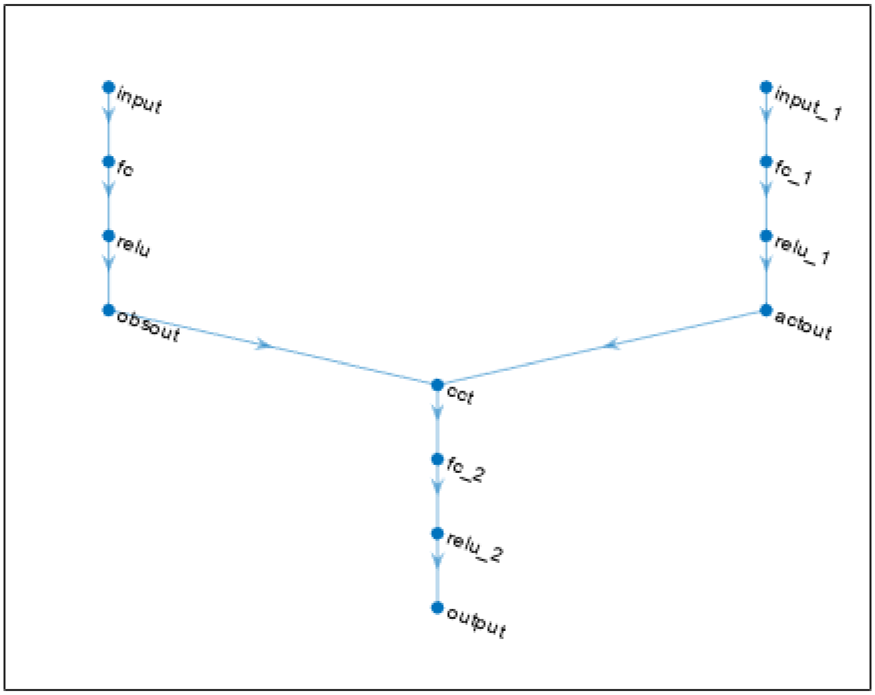
```
actInfo = rlNumericSpec([2 1]);
```

A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with two input layers (one receiving the content of the observation channel, as specified by `obsInfo`, and the other receiving the content of the action channel, as specified by `actInfo`) and one output layer (returning the scalar value).

Define each network path as an array of layer objects. Assign names to the input and output layers of each path, so you can properly connect the paths. Get the dimensions of the observation and action spaces from the environment specification (regardless of whether the observation space is a column vector, row vector, or matrix, `prod(obsInfo.Dimension)` is its total number of dimensions, for this example four, similarly, `prod(actInfo.Dimension)` is the number of dimension of the action space, for this example two).

```
% Observation path layers
obsPath = [featureInputLayer(prod(obsInfo.Dimension))
           fullyConnectedLayer(5)
           reluLayer
           fullyConnectedLayer(5,Name="obsout")];

% Action path layers
actPath = [featureInputLayer(prod(actInfo.Dimension))
           fullyConnectedLayer(5)
           reluLayer
           fullyConnectedLayer(5,Name="actout")];

% Common path to output layers
% Concatenate two layers along dimension one
comPath = [concatenationLayer(1,2,Name="cct")
           fullyConnectedLayer(5)
           reluLayer
           fullyConnectedLayer(1, Name="output")];

% Add layers to network object
net = addLayers(layerGraph(obsPath),actPath);
net = addLayers(net,comPath);

% Connect layers
net = connectLayers(net,"obsout","cct/in1");
net = connectLayers(net,"actout","cct/in2");

% Plot network
plot(net)
```



Convert to `dlnetwork` and display the number of weights.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 161

   Inputs:
      1   'input'     4 features
      2   'input_1'   2 features
```

Create the critic with `rlQValueFunction`, using the network as well as the observations and action specification objects. When using this syntax, the network input layers are automatically associated with the components of the observation and action signals according to the dimension specifications in `obsInfo` and `actInfo`.

```
critic = rlQValueFunction(net,obsInfo,actInfo)
```

```
critic = 
  rlQValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

To check your critic, use the getValue function to return the value of a random observation and action, given the current network weights.

```
v = getValue(critic, ...
    {rand(obsInfo.Dimension)}, ...
    {rand(actInfo.Dimension)})
```

```
v = single
    -1.1006
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given specification objects. Examples of agents that can work with continuous action and observation spaces, and use a Q-value function critic, are rlDDPGAgent, rlTD3Agent, and rlSACAgentOptions.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Q-Value Function Critic from Deep Neural Network Specifying Layer Names

Create an observation specification object (or alternatively use getObservationInfo to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create an action specification object (or alternatively use getActionInfo to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that the action channel carries a column vector containing two doubles.

```
actInfo = rlNumericSpec([2 1]);
```

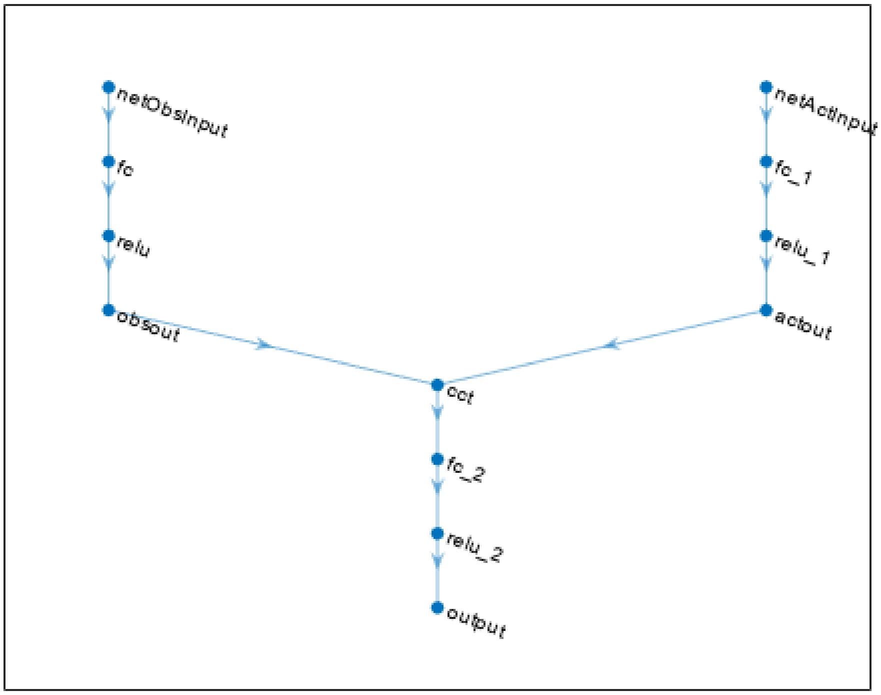
A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with two input layers (one receiving the content of the observation channel, as specified by obsInfo, and the other receiving the content of the action channel, as specified by actInfo) and one output layer (returning the scalar value).

Define each network path as an array of layer objects. Assign names to the input and output layers of each path. This assignment allows you to connect the paths and explicitly associate the network input and output layers with the appropriate environment channel.

Get the dimensions of the observation and action spaces from the environment specification (regardless of whether the observation space is a column vector, row vector, or matrix, prod(obsInfo.Dimension) is its total number of dimensions, for this example four, similarly, prod(actInfo.Dimension) is the number of dimension of the action space, for this example two).

```
% Observation path layers
obsPath = [featureInputLayer( ...
                prod(obsInfo.Dimension), ...
                Name="netObsInput")
           fullyConnectedLayer(16)
           reluLayer
           fullyConnectedLayer(5,Name="obsout")];

% Action path layers
actPath = [featureInputLayer( ...
                prod(actInfo.Dimension), ...
                Name="netActInput")
           fullyConnectedLayer(16)
           reluLayer
           fullyConnectedLayer(5,Name="actout")];

% Common path to output layers
% Concatenate two layers along dimension one
comPath = [concatenationLayer(1,2,Name="cct")
           fullyConnectedLayer(8)
           reluLayer
           fullyConnectedLayer(1, Name="output")];

% Add layers to network object
net = addLayers(layerGraph(obsPath),actPath);
net = addLayers(net,comPath);

% Connect layers
net = connectLayers(net,"obsout","cct/in1");
net = connectLayers(net,"actout","cct/in2");

% Plot network
plot(net)
```



Convert to `dlnetwork` and display the number of weights.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 395

   Inputs:
      1   'netObsInput'   4 features
      2   'netActInput'   2 features
```

Create the critic with `rlQValueFunction`, using the network, the observations and action specification objects, and the names of the network input layers to be associated with the observation and action from the environment.

```
critic = rlQValueFunction(net,...
             obsInfo,actInfo, ...
             ObservationInputNames="netObsInput",...
             ActionInputNames="netActInput")
```

```
critic = 
  rlQValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
```

```
          UseDevice: "cpu"
```

To check your critic, use the getValue function to return the value of a random observation and action, given the current network weights.

```
v = getValue(critic, ...
    {rand(obsInfo.Dimension)}, ...
    {rand(actInfo.Dimension)})
```

```
v = single
    0.2703
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given specification objects. Examples of agents that can work with continuous action and observation spaces, and use a Q-value function critic, are rlDDPGAgent, rlTD3Agent, and rlSACAgentOptions.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Q-Value Function Critic from Table

Create a finite set observation specification object (or alternatively use getObservationInfo to extract the specification object from an environment with a discrete observation space). For this example define the observation space as a finite set with of four possible values.

```
obsInfo = rlFiniteSetSpec([7 5 3 1]);
```

Create a finite set action specification object (or alternatively use getActionInfo to extract the specification object from an environment with a discrete action space). For this example define the action space as a finite set with two possible values.

```
actInfo = rlFiniteSetSpec([4 8]);
```

A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

Since both observation and action spaces are discrete and low-dimensional, use a table to model the Q-value function within the critic. rlTable creates a value table object from the observation and action specifications objects.

```
qTable = rlTable(obsInfo,actInfo);
```

The table stores a value (representing the expected cumulative long term reward) for each possible observation-action pair. Each row corresponds to an observation and each column corresponds to an action. You can access the table using the Table property of the vTable object. The initial value of each element is zero.

```
qTable.Table
```

```
ans = 4×2
```

```
     0     0
     0     0
     0     0
     0     0
```

You can initialize the table to any value, in this case an array containing the integer from 1 through 8.

```
qTable.Table=reshape(1:8,4,2)
```

```
qTable = 
  rlTable with properties:

    Table: [4x2 double]
```

Create the critic using the table as well as the observations and action specification objects.

```
critic = rlQValueFunction(qTable,obsInfo,actInfo)
```

```
critic = 
  rlQValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlFiniteSetSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
          UseDevice: "cpu"
```

To check your critic, use the `getValue` function to return the value of a given observation and action, using the current table entries.

```
v = getValue(critic,{5},{8})
```

```
v = 6
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given specification objects. Examples of agents that can work with discrete action and observation spaces, and use a Q-value function critic, are `rlQAgent`, `rlDQNAgent`, and `rlSARSAAgent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Q-Value Function Critic from Custom Basis Function

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous three-dimensional space, so that there is a single observation channel that carries a column vector containing three doubles.

```
obsInfo = rlNumericSpec([3 1]);
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that the action channel carries a column vector containing two doubles.

```
actInfo = rlNumericSpec([2 1]);
```

A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a custom basis function with two inputs, the first receiving the content of the observation channel (as specified by obsInfo) and the other receiving the content of the action channel, (as specified by actInfo).

Create a custom function that returns a vector of four elements, given an observation and an action as inputs.

```
myBasisFcn = @(myobs,myact) [ ...
    myobs(2)^2; ...
    myobs(1)+exp(myact(1)); ...
    abs(myact(2)); ...
    myobs(3) ]
```

```
myBasisFcn = function_handle with value:
    @(myobs,myact)[myobs(2)^2;myobs(1)+exp(myact(1));abs(myact(2));myobs(3)]
```

The output of the critic is the scalar W'*myBasisFcn(myobs,myact), which represents the estimated value of the observation-action pair, under the given policy. Here W is a weight column vector which must have the same size of the custom function output. The elements of W are the learnable parameters.

Define an initial parameter vector.

```
W0 = [1;4;4;2];
```

Create the critic. The first argument is a two-element cell containing both the handle to the custom function and the initial weight vector. The second and third arguments are, respectively, the observation and action specification objects.

```
critic = rlQValueFunction({myBasisFcn,W0}, ...
    obsInfo,actInfo)
```

```
critic = 
  rlQValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

To check your critic, use getValue to return the value of a given observation-action pair, using the current parameter vector.

```
v = getValue(critic,{[1 2 3]'},{[4 5]'})
```

```
v = 252.3926
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given specification objects. Examples of agents that can work with continuous action and observation spaces, and use a Q-value function critic, are rlDDPGAgent, rlTD3Agent, and rlSACAgentOptions.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Mixed Observation Space Q-Value Function Critic from Custom Basis Function

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as consisting of two channels, the first carrying a vector over a continuous two-dimensional space and the second carrying a vector over a three-dimensional space that can assume only four values.

```
obsInfo = [rlNumericSpec([1 2])
           rlFiniteSetSpec({[1 0 -1], ...
                            [-1 2 1], ...
                            [0.1 0.2 0.3], ...
                            [0 0 0]})];
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as discrete set consisting of three possible actions, labeled 1, 2, and 3.

```
actInfo = rlFiniteSetSpec({1,2,3});
```

A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a custom basis function with three inputs, the first two receiving the content of the two observation channel (as specified by `obsInfo`) and the third receiving the content of the action channel, (as specified by `actInfo`).

Create a custom function that returns a vector of four elements, given an observation and an action as inputs.

```
myBasisFcn = @(obsA,obsB,act) [
    obsA(1)+obsB(2)+obsB(3)+act(1);
    obsA(2)+obsB(1)+obsB(2)-act(1);
    obsA(1)+obsB(2)+obsB(3)+act(1)^2;
    obsA(1)+obsB(1)+obsB(2)-act(1)^2 ];
```

The output of the critic is the scalar `W'*myBasisFcn(myobs,myact)`, which represents the estimated value of the observation-action pair, under the given policy. Here `W` is a weight column vector which must have the same size of the custom function output. The elements of `W` are the learnable parameters.

Define an initial parameter vector.

```
W0 = ones(4,1);
```

Create the critic. The first argument is a two-element cell containing both the handle to the custom function and the initial weight vector. The second and third arguments are, respectively, the observation and action specification objects.

```
critic = rlQValueFunction({myBasisFcn,W0}, ...
    obsInfo,actInfo)
```

```
critic = 
  rlQValueFunction with properties:

    ObservationInfo: [2x1 rl.util.RLDataSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
          UseDevice: "cpu"
```

To check your critic, use the `getValue` function to return the value of a given observation-action pair, using the current parameter vector.

```
v = getValue(critic,{[-0.5 0.6],[1 0 -1]},{3})
```

```
v = -0.9000
```

Note that the critic does not enforce the set constraint for the discrete set elements.

```
v = getValue(critic,{[-0.5 0.6],[10 -10 -0.05]},{33})
```

```
v = -21.0000
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given specification objects. Examples of agents that can work with a discrete action space, a mixed observation space, and use a Q-value function critic, are `rlQAgent`, `rlDQNAgent`, and `rlSARSAAgent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Version History

Introduced in R2022a

See Also

Functions

`getValue` | `getMaxQValue` | `getModel` | `evaluate` | `getObservationInfo` | `getActionInfo`

Objects

`rlNumericSpec` | `rlFiniteSetSpec` | `rlValueFunction` | `rlVectorQValueFunction` | `rlTable` | `rlQAgent` | `rlSARSAAgent` | `rlDQNAgent` | `rlDDPGAgent` | `rlTD3Agent`

Topics

“Train Reinforcement Learning Agent in Basic Grid World”
“Train Reinforcement Learning Agent in MDP Environment”
“Train DDPG Agent to Control Double Integrator System”
“Train TD3 Agent for PMSM Control”
“Create Policies and Value Functions”
“Reinforcement Learning Agents”

rlQValueRepresentation

(Not recommended) Q-Value function critic representation for reinforcement learning agents

Note `rlQValueRepresentation` is not recommended. Use `rlQValueFunction` or `rlVectorQValueFunction` instead. For more information, see “rlQValueRepresentation is not recommended”.

Description

This object implements a Q-value function approximator to be used as a critic within a reinforcement learning agent. A Q-value function is a function that maps an observation-action pair to a scalar value representing the expected total long-term rewards that the agent is expected to accumulate when it starts from the given observation and executes the given action. Q-value function critics therefore need both observations and actions as inputs. After you create an `rlQValueRepresentation` critic, use it to create an agent relying on a Q-value function critic, such as an `rlQAgent`, `rlDQNAgent`, `rlSARSAAgent`, `rlDDPGAgent`, or `rlTD3Agent`. For more information on creating representations, see “Create Policies and Value Functions”.

Creation

Syntax

```
critic = rlQValueRepresentation(net,observationInfo,actionInfo,'Observation',
obsName,'Action',actName)
critic = rlQValueRepresentation(tab,observationInfo,actionInfo)
critic = rlQValueRepresentation({basisFcn,W0},observationInfo,actionInfo)
```

```
critic = rlQValueRepresentation(net,observationInfo,actionInfo,'Observation',
obsName)
critic = rlQValueRepresentation({basisFcn,W0},observationInfo,actionInfo)
```

```
critic = rlQValueRepresentation( ___ ,options)
```

Description

Scalar Output Q-Value Critic

`critic = rlQValueRepresentation(net,observationInfo,actionInfo,'Observation', obsName,'Action',actName)` creates the Q-value function `critic`. `net` is the deep neural network used as an approximator, and must have both observations and action as inputs, and a single scalar output. This syntax sets the ObservationInfo and ActionInfo properties of `critic` respectively to the inputs `observationInfo` and `actionInfo`, containing the observations and action specifications. `obsName` must contain the names of the input layers of `net` that are associated with the observation specifications. The action name `actName` must be the name of the input layer of `net` that is associated with the action specifications.

`critic = rlQValueRepresentation(tab,observationInfo,actionInfo)` creates the Q-value function based `critic` with *discrete action and observation spaces* from the Q-value table `tab`.

`tab` is a `rlTable` object containing a table with as many rows as the possible observations and as many columns as the possible actions. This syntax sets the ObservationInfo and ActionInfo properties of `critic` respectively to the inputs `observationInfo` and `actionInfo`, which must be `rlFiniteSetSpec` objects containing the specifications for the discrete observations and action spaces, respectively.

`critic = rlQValueRepresentation({basisFcn,W0},observationInfo,actionInfo)` creates a Q-value function based `critic` using a custom basis function as underlying approximator. The first input argument is a two-elements cell in which the first element contains the handle `basisFcn` to a custom basis function, and the second element contains the initial weight vector `W0`. Here the basis function must have both observations and action as inputs and `W0` must be a column vector. This syntax sets the ObservationInfo and ActionInfo properties of `critic` respectively to the inputs `observationInfo` and `actionInfo`.

Multi-Output Discrete Action Space Q-Value Critic

`critic = rlQValueRepresentation(net,observationInfo,actionInfo,'Observation',obsName)` creates the *multi-output* Q-value function `critic` *for a discrete action space*. `net` is the deep neural network used as an approximator, and must have only the observations as input and a single output layer having as many elements as the number of possible discrete actions. This syntax sets the ObservationInfo and ActionInfo properties of `critic` respectively to the inputs `observationInfo` and `actionInfo`, containing the observations and action specifications. Here, `actionInfo` must be an `rlFiniteSetSpec` object containing the specifications for the discrete action space. The observation names `obsName` must be the names of the input layers of `net`.

`critic = rlQValueRepresentation({basisFcn,W0},observationInfo,actionInfo)` creates the *multi-output* Q-value function `critic` *for a discrete action space* using a custom basis function as underlying approximator. The first input argument is a two-elements cell in which the first element contains the handle `basisFcn` to a custom basis function, and the second element contains the initial weight matrix `W0`. Here the basis function must have only the observations as inputs, and `W0` must have as many columns as the number of possible actions. This syntax sets the ObservationInfo and ActionInfo properties of `critic` respectively to the inputs `observationInfo` and `actionInfo`.

Options

`critic = rlQValueRepresentation( ___ ,options)` creates the value function based `critic` using the additional option set `options`, which is an `rlRepresentationOptions` object. This syntax sets the Options property of `critic` to the `options` input argument. You can use this syntax with any of the previous input-argument combinations.

Input Arguments

net — Deep neural network

array of `Layer` objects | `layerGraph` object | `DAGNetwork` object | `SeriesNetwork` object | `dlNetwork` object

Deep neural network used as the underlying approximator within the critic, specified as one of the following:

- Array of `Layer` objects
- `layerGraph` object
- `DAGNetwork` object

- SeriesNetwork object
- dlnetwork object

For *single output* critics, net must have both observations and actions as inputs, and a scalar output, representing the expected cumulative long-term reward when the agent starts from the given observation and takes the given action. For *multi-output discrete action space* critics, net must have only the observations as input and a single output layer having as many elements as the number of possible discrete actions. Each output element represents the expected cumulative long-term reward when the agent starts from the given observation and takes the corresponding action. The learnable parameters of the critic are the weights of the deep neural network.

The network input layers must be in the same order and with the same data type and dimensions as the signals defined in ObservationInfo. Also, the names of these input layers must match the observation names listed in obsName.

The network output layer must have the same data type and dimension as the signal defined in ActionInfo. Its name must be the action name specified in actName.

rlQValueRepresentation objects support recurrent deep neural networks for multi-output discrete action space critics.

For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

obsName — Observation names
string | character vector | cell array or character vectors

Observation names, specified as a cell array of strings or character vectors. The observation names must be the names of the observation input layers in net.

Example: {'my_obs'}

actName — Action name
string | character vector | single-element cell array containing a character vector

Action name, specified as a single-element cell array that contains a string or character vector. It must be the name of the action input layer of net.

Example: {'my_act'}

tab — Q-value table
rlTable object

Q-value table, specified as an rlTable object containing an array with as many rows as the possible observations and as many columns as the possible actions. The element (s,a) is the expected cumulative long-term reward for taking action a from observed state s. The elements of this array are the learnable parameters of the critic.

basisFcn — Custom basis function
function handle

Custom basis function, specified as a function handle to a user-defined MATLAB function. The user defined function can either be an anonymous function or a function on the MATLAB path. The output of the critic is c = W'*B, where W is a weight vector or matrix containing the learnable parameters, and B is the column vector returned by the custom basis function.

For a single-output Q-value critic, c is a scalar representing the expected cumulative long term reward when the agent starts from the given observation and takes the given action. In this case, your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN,act)
```

For a multiple-output Q-value critic with a discrete action space, c is a vector in which each element is the expected cumulative long term reward when the agent starts from the given observation and takes the action corresponding to the position of the considered element. In this case, your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here, obs1 to obsN are observations in the same order and with the same data type and dimensions as the signals defined in observationInfo and act has the same data type and dimensions as the action specifications in actionInfo.

Example: @(obs1,obs2,act) [act(2)*obs1(1)^2; abs(obs2(5)+act(1))]

W0 — Initial value of the basis function weights

matrix

Initial value of the basis function weights, W. For a single-output Q-value critic, W is a column vector having the same length as the vector returned by the basis function. For a multiple-output Q-value critic with a discrete action space, W is a matrix which must have as many rows as the length of the basis function output, and as many columns as the number of possible actions.

Properties

Options — Representation options

rlRepresentationOptions object

Representation options, specified as an rlRepresentationOptions object. Available options include the optimizer used for training and the learning rate.

ObservationInfo — Observation specifications

rlFiniteSetSpec object | rlNumericSpec object | array

Observation specifications, specified as an rlFiniteSetSpec or rlNumericSpec object or an array containing a mix of such objects. These objects define properties such as the dimensions, data type, and names of the observation signals.

rlQValueRepresentation sets the ObservationInfo property of critic to the input observationInfo.

You can extract ObservationInfo from an existing environment or agent using getObservationInfo. You can also construct the specifications manually.

ActionInfo — Action specifications

rlFiniteSetSpec object | rlNumericSpec object

Action specifications, specified as an rlFiniteSetSpec or rlNumericSpec object. These objects define properties such as the dimensions, data type and name of the action signals.

rlQValueRepresentation sets the ActionInfo property of critic to the input actionInfo.

You can extract `ActionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specifications manually.

Object Functions

rlDDPGAgent	Deep deterministic policy gradient (DDPG) reinforcement learning agent
rlTD3Agent	Twin-delayed deep deterministic (TD3) policy gradient reinforcement learning agent
rlDQNAgent	Deep Q-network (DQN) reinforcement learning agent
rlQAgent	Q-learning reinforcement learning agent
rlSARSAAgent	SARSA reinforcement learning agent
rlSACAgent	Soft actor-critic (SAC) reinforcement learning agent
getValue	Obtain estimated value from a critic given environment observations and actions
getMaxQValue	Obtain maximum estimated value over all possible actions from a Q-value function critic with discrete action space, given environment observations

Examples

Create Q-Value Function Critic from Deep Neural Network

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that a single action is a column vector containing two doubles.

```
actInfo = rlNumericSpec([2 1]);
```

Create a deep neural network to approximate the Q-value function. The network must have two inputs, one for the observation and one for the action. The observation input (here called `myobs`) must accept a four-element vector (the observation vector defined by `obsInfo`). The action input (here called `myact`) must accept a two-element vector (the action vector defined by `actInfo`). The output of the network must be a scalar, representing the expected cumulative long-term reward when the agent starts from the given observation and takes the given action.

```
% observation path layers
obsPath = [featureInputLayer(4, ...
               'Normalization','none','Name','myobs')
           fullyConnectedLayer(1,'Name','obsout')];

% action path layers
actPath = [featureInputLayer(2, ...
               'Normalization','none','Name','myact')
           fullyConnectedLayer(1,'Name','actout')];

% common path to output layers
comPath = [additionLayer(2,'Name', 'add')  ...
           fullyConnectedLayer(1, 'Name', 'output')];

% add layers to network object
```

```
net = addLayers(layerGraph(obsPath),actPath);
net = addLayers(net,comPath);

% connect layers
net = connectLayers(net,'obsout','add/in1');
net = connectLayers(net,'actout','add/in2');
```

Create the critic with `rlQValueRepresentation`, using the network, the observations and action specification objects, as well as the names of the network input layers.

```
critic = rlQValueRepresentation(net,obsInfo,actInfo, ...
    'Observation',{'myobs'},'Action',{'myact'})
```

```
critic = 
  rlQValueRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlNumericSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your critic, use the `getValue` function to return the value of a random observation and action, using the current network weights.

```
v = getValue(critic,{rand(4,1)},{rand(2,1)})
```

```
v = single
    0.1102
```

You can now use the critic (along with an with an actor) to create an agent relying on a Q-value function critic (such as an `rlQAgent`, `rlDQNAgent`, `rlSARSAAgent`, or `rlDDPGAgent` agent).

Create Multi-Output Q-Value Function Critic from Deep Neural Network

This example shows how to create a multi-output Q-value function critic for a discrete action space using a deep neural network approximator.

This critic takes only the observation as input and produces as output a vector with as many elements as the possible actions. Each element represents the expected cumulative long term reward when the agent starts from the given observation and takes the action corresponding to the position of the element in the output vector.

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create a finite set action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment with a discrete action space). For this example, define the action space as a finite set consisting of three possible values (named `7`, `5`, and `3` in this case).

```
actInfo = rlFiniteSetSpec([7 5 3]);
```

Create a deep neural network approximator to approximate the Q-value function within the critic. The input of the network (here called `myobs`) must accept a four-element vector, as defined by `obsInfo`. The output must be a single output layer having as many elements as the number of possible discrete actions (three in this case, as defined by `actInfo`).

```
net = [featureInputLayer(4,...
           'Normalization','none','Name','myobs')
       fullyConnectedLayer(3,'Name','value')];
```

Create the critic using the network, the observations specification object, and the name of the network input layer.

```
critic = rlQValueRepresentation(net,obsInfo,actInfo,...
             'Observation',{'myobs'})
```

```
critic = 
  rlQValueRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your critic, use the `getValue` function to return the values of a random observation, using the current network weights. There is one value for each of the three possible actions.

```
v = getValue(critic,{rand(4,1)})
```

```
v = 3x1 single column vector

    0.7232
    0.8177
   -0.2212
```

You can now use the critic (along with an actor) to create a discrete action space agent relying on a Q-value function critic (such as an `rlQAgent`, `rlDQNAgent`, or `rlSARSAAgent` agent).

Create Q-Value Function Critic from Table

Create a finite set observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment with a discrete observation space). For this example define the observation space as a finite set with of 4 possible values.

```
obsInfo = rlFiniteSetSpec([7 5 3 1]);
```

Create a finite set action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment with a discrete action space). For this example define the action space as a finite set with 2 possible values.

```
actInfo = rlFiniteSetSpec([4 8]);
```

Create a table to approximate the value function within the critic. `rlTable` creates a value table object from the observation and action specifications objects.

```
qTable = rlTable(obsInfo,actInfo);
```

The table stores a value (representing the expected cumulative long term reward) for each possible observation-action pair. Each row corresponds to an observation and each column corresponds to an action. You can access the table using the `Table` property of the `vTable` object. The initial value of each element is zero.

```
qTable.Table
```

```
ans = 4×2

     0     0
     0     0
     0     0
     0     0
```

You can initialize the table to any value, in this case, an array containing the integer from `1` through `8`.

```
qTable.Table=reshape(1:8,4,2)
```

```
qTable = 
  rlTable with properties:

    Table: [4x2 double]
```

Create the critic using the table as well as the observations and action specification objects.

```
critic = rlQValueRepresentation(qTable,obsInfo,actInfo)
```

```
critic = 
  rlQValueRepresentation with properties:

           ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
      ObservationInfo: [1x1 rl.util.rlFiniteSetSpec]
              Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your critic, use the `getValue` function to return the value of a given observation and action, using the current table entries.

```
v = getValue(critic,{5},{8})
```

```
v = 6
```

You can now use the critic (along with an with an actor) to create a discrete action space agent relying on a Q-value function critic (such as an `rlQAgent`, `rlDQNAgent`, or `rlSARSAAgent` agent).

Create Q-Value Function Critic from Custom Basis Function

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing 3 doubles.

```
obsInfo = rlNumericSpec([3 1]);
```

Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that a single action is a column vector containing 2 doubles.

```
actInfo = rlNumericSpec([2 1]);
```

Create a custom basis function to approximate the value function within the critic. The custom basis function must return a column vector. Each vector element must be a function of the observations and actions respectively defined by `obsInfo` and `actInfo`.

```
myBasisFcn = @(myobs,myact) [...
    myobs(2)^2; ...
    myobs(1)+exp(myact(1)); ...
    abs(myact(2)); ...
    myobs(3)]
```

```
myBasisFcn = function_handle with value:
    @(myobs,myact)[myobs(2)^2;myobs(1)+exp(myact(1));abs(myact(2));myobs(3)]
```

The output of the critic is the scalar `W'*myBasisFcn(myobs,myact)`, where `W` is a weight column vector which must have the same size of the custom basis function output. This output is the expected cumulative long term reward when the agent starts from the given observation and takes the best possible action. The elements of W are the learnable parameters.

Define an initial parameter vector.

```
W0 = [1;4;4;2];
```

Create the critic. The first argument is a two-element cell containing both the handle to the custom function and the initial weight vector. The second and third arguments are, respectively, the observation and action specification objects.

```
critic = rlQValueRepresentation({myBasisFcn,W0},...
         obsInfo,actInfo)
```

```
critic = 
  rlQValueRepresentation with properties:

         ActionInfo: [1×1 rl.util.rlNumericSpec]
    ObservationInfo: [1×1 rl.util.rlNumericSpec]
            Options: [1×1 rl.option.rlRepresentationOptions]
```

To check your critic, use the `getValue` function to return the value of a given observation-action pair, using the current parameter vector.

```
v = getValue(critic,{[1 2 3]'},{[4 5]'})
```

```
v = 
  1×1 dlarray

  252.3926
```

You can now use the critic (along with an with an actor) to create an agent relying on a Q-value function critic (such as an `rlQAgent`, `rlDQNAgent`, `rlSARSAAgent`, or `rlDDPGAgent` agent).

Create Multi-Output Q-Value Function Critic from Custom Basis Function

This example shows how to create a multi-output Q-value function critic for a discrete action space using a custom basis function approximator.

This critic takes only the observation as input and produces as output a vector with as many elements as the possible actions. Each element represents the expected cumulative long term reward when the agent starts from the given observation and takes the action corresponding to the position of the element in the output vector.

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing 2 doubles.

```
obsInfo = rlNumericSpec([2 1]);
```

Create a finite set action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment with a discrete action space). For this example, define the action space as a finite set consisting of 3 possible values (named `7`, `5`, and `3` in this case).

```
actInfo = rlFiniteSetSpec([7 5 3]);
```

Create a custom basis function to approximate the value function within the critic. The custom basis function must return a column vector. Each vector element must be a function of the observations defined by `obsInfo`.

```
myBasisFcn = @(myobs) [myobs(2)^2; ...
                       myobs(1); ...
                       exp(myobs(2)); ...
                       abs(myobs(1))]
```

```
myBasisFcn = function_handle with value:
    @(myobs)[myobs(2)^2;myobs(1);exp(myobs(2));abs(myobs(1))]
```

The output of the critic is the vector c `= W'*myBasisFcn(myobs)`, where `W` is a weight matrix which must have as many rows as the length of the basis function output, and as many columns as the number of possible actions.

Each element of c is the expected cumulative long term reward when the agent starts from the given observation and takes the action corresponding to the position of the considered element. The elements of W are the learnable parameters.

Define an initial parameter matrix.

```
W0 = rand(4,3);
```

Create the critic. The first argument is a two-element cell containing both the handle to the custom function and the initial parameter matrix. The second and third arguments are, respectively, the observation and action specification objects.

```
critic = rlQValueRepresentation({myBasisFcn,W0},...
             obsInfo,actInfo)
```

```
critic = 
  rlQValueRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your critic, use the getValue function to return the values of a random observation, using the current parameter matrix. Note that there is one value for each of the three possible actions.

```
v = getValue(critic,{rand(2,1)})
```

```
v = 
  3x1 dlarray

    2.1395
    1.2183
    2.3342
```

You can now use the critic (along with an actor) to create a discrete action space agent relying on a Q-value function critic (such as an rlQAgent, rlDQNAgent, or rlSARSAAgent agent).

Create Q-Value Function Critic from Recurrent Neural Network

Create an environment and obtain observation and action information.

```
env = rlPredefinedEnv('CartPole-Discrete');
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
numObs = obsInfo.Dimension(1);
numDiscreteAct = numel(actInfo.Elements);
```

Create a recurrent deep neural network for your critic. To create a recurrent neural network, use a sequenceInputLayer as the input layer and include at least one lstmLayer.

Create a recurrent neural network for a multi-output Q-value function representation.

```
criticNetwork = [
    sequenceInputLayer(numObs,...
        'Normalization','none','Name','state')
    fullyConnectedLayer(50, 'Name', 'CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    lstmLayer(20,'OutputMode','sequence',...
        'Name','CriticLSTM');
    fullyConnectedLayer(20,'Name','CriticStateFC2')
    reluLayer('Name','CriticRelu2')
    fullyConnectedLayer(numDiscreteAct,...
        'Name','output')];
```

Create a representation for your critic using the recurrent neural network.

```
criticOptions = rlRepresentationOptions(...
                'LearnRate',1e-3,'GradientThreshold',1);
critic = rlQValueRepresentation(criticNetwork,...
                obsInfo,actInfo,...
                'Observation','state',criticOptions);
```

Version History

Introduced in R2020a

R2022a: `rlQValueRepresentation` is not recommended

Not recommended starting in R2022a

`rlQValueRepresentation` is not recommended. Use either `rlQValueFunction` or `rlVectorQValueFunction` instead.

The following table shows some typical uses of `rlQValueRepresentation` to create *neural network*-based critics, and how to update your code with one of the new Q-value approximator objects instead.

Network-Based Q-Value Representation: Not Recommended	Network-Based Q-Value Approximators: Recommended
`myCritic = rlQValueRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action',actNames)`, with `net` having both observations and actions as inputs and a single scalar output.	`myCritic = rlQValueFunction(net,obsInfo,actInfo,'ObservationInputNames',obsNames,'ActionInputNames',actNames)`. Use this syntax to create a single-output state-action value function object for a critic that takes both observation and action as inputs.
`myCritic = rlQValueRepresentation(net,obsInfo,actInfo,'Observation',obsNames)` with `net` having only the observations as inputs and a single output layer having as many elements as the number of possible discrete actions.	`myCritic = rlVectorQValueFunction(net,obsInfo,actInfo,'ObservationInputNames',obsNames)`. Use this syntax to create a multiple-output state-action value function object for a critic with a discrete action space. This critic takes observations as inputs, and outputs a vector in which each element is the value of one of the possible actions.

The following table shows some typical uses of `rlQValueRepresentation` to create *table*-based critics with discrete observation and action spaces, and how to update your code with one of the new Q-value approximator objects instead.

Table-Based Q-Value Representation: Not Recommended	Table-Based Q-Value Approximators: Recommended
`rep = rlQValueRepresentation(tab,obsInfo,actInfo)`, where the table `tab` contains a vector with as many elements as the number of possible observations plus the number of possible actions.	`rep = rlQValueFunction(tab,obsInfo,actInfo)`. Use this syntax to create a single-output state-action value function object for a critic that takes both observations and actions as input.

Table-Based Q-Value Representation: Not Recommended	Table-Based Q-Value Approximators: Recommended
`rep = rlQValueRepresentation(tab,obsInfo,actInfo)`, where the table `tab` contains a Q-value table with as many rows as the number of possible observations and as many columns as the number of possible actions.	`rep = rlVectorQValueFunction(tab,obsInfo,actInfo)`. Use this syntax to create a multiple-output state-action value function object for a critic with a discrete action space. This critic takes observations as inputs, and outputs a vector in which each element is the value of one of the possible actions. It is good practice to use critics with vector outputs when possible.

The following table shows some typical uses of `rlQValueRepresentation` to create critics which use a (linear in the learnable parameters) *custom basis function*, and how to update your code with one of the new Q-value approximator objects instead. In these function calls, the first input argument is a two-element cell array containing both the handle to the custom basis function and the initial weight vector or matrix.

Custom Basis Function-Based Q-Value Representation: Not Recommended	Custom Basis Function-Based Q-Value Approximators: Recommended
`rep = rlQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`, where the basis function has both observations and action as inputs and `W0` is a column vector.	`rep = rlQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a single-output state-action value function object for a critic that takes both observation and action as inputs.
`rep = rlQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`, where the basis function has both observations and action as inputs and `W0` is a matrix with as many columns as the number of possible actions.	`rep = rlVectorQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a multiple-output state-action value function object for a critic with a discrete action space. This critic takes observations as inputs, and outputs a vector in which each element is the value of one of the possible actions. It is good practice to use critics with vector outputs when possible.

See Also

Functions
`getActionInfo` | `getObservationInfo`

Objects
`rlQValueFunction` | `rlRepresentationOptions`

Topics
“Create Policies and Value Functions”
“Reinforcement Learning Agents”

rlReplayMemory

Replay memory experience buffer

Description

An off-policy reinforcement learning agent stores experiences in a circular experience buffer. During training, the agent samples mini-batches of experiences from the buffer and uses these mini-batches to update its actor and critic function approximators.

By default, built-in off-policy agents (DQN, DDPG, TD3, SAC, MBPO) use an `rlReplayMemory` object as their experience buffer. Agents uniformly sample data from this buffer.

You can replace the default experience buffer using one of the following alternative buffer objects.

- `rlPrioritizedReplayMemory` — Prioritized nonuniform sampling of experiences
- `rlHindsightReplayMemory` — Uniform sampling of experiences and generation of hindsight experiences by replacing goals with goal measurements
- `rlHindsightPrioritizedReplayMemory` — Prioritized nonuniform sampling of experiences and generation of hindsight experiences

When you create a custom off-policy reinforcement learning agent, you can create an experience buffer using an `rlReplayMemory` object.

Creation

Syntax

```
buffer = rlReplayMemory(obsInfo,actInfo)
buffer = rlReplayMemory(obsInfo,actInfo,maxLength)
```

Description

`buffer = rlReplayMemory(obsInfo,actInfo)` creates a replay memory experience buffer that is compatible with the observation and action specifications in `obsInfo` and `actInfo`, respectively.

`buffer = rlReplayMemory(obsInfo,actInfo,maxLength)` sets the maximum length of the buffer by setting the `MaxLength` property.

Input Arguments

`obsInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, specified as a reinforcement learning specification object or an array of specification objects defining properties such as dimensions, data types, and names of the observation signals.

You can extract the observation specifications from an existing environment or agent using getObservationInfo. You can also construct the specifications manually using rlFiniteSetSpec or rlNumericSpec.

actInfo — Action specifications
specification object | array of specification objects

Action specifications, specified as a reinforcement learning specification object defining properties such as dimensions, data types, and names of the action signals.

You can extract the action specifications from an existing environment or agent using getActionInfo. You can also construct the specification manually using rlFiniteSetSpec or rlNumericSpec.

Properties

MaxLength — Maximum buffer length
10000 (default) | nonnegative integer

This property is read-only.

Maximum buffer length, specified as a nonnegative integer.

To change the maximum buffer length, use the resize function.

Length — Number of experiences in buffer
0 (default) | nonnegative integer

This property is read-only.

Number of experiences in buffer, specified as a nonnegative integer.

Object Functions

append	Append experiences to replay memory buffer
sample	Sample experiences from replay memory buffer
resize	Resize replay memory experience buffer
reset	Reset environment, agent, experience buffer, or policy object
allExperiences	Return all experiences in replay memory buffer
validateExperience	Validate experiences for replay memory
getActionInfo	Obtain action data specifications from reinforcement learning environment, agent, or experience buffer
getObservationInfo	Obtain observation data specifications from reinforcement learning environment, agent, or experience buffer

Examples

Create Experience Buffer

Define observation specifications for the environment. For this example, assume that the environment has a single observation channel with three continuous signals in specified ranges.

```
obsInfo = rlNumericSpec([3 1],...
    LowerLimit=0,...
    UpperLimit=[1;5;10]);
```

Define action specifications for the environment. For this example, assume that the environment has a single action channel with two continuous signals in specified ranges.

```
actInfo = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[5;10]);
```

Create an experience buffer with a maximum length of 20,000.

```
buffer = rlReplayMemory(obsInfo,actInfo,20000);
```

Append a single experience to the buffer using a structure. Each experience contains the following elements: current observation, action, next observation, reward, and is-done.

For this example, create an experience with random observation, action, and reward values. Indicate that this experience is not a terminal condition by setting the `IsDone` value to 0.

```
exp.Observation = {obsInfo.UpperLimit.*rand(3,1)};
exp.Action = {actInfo.UpperLimit.*rand(2,1)};
exp.Reward = 10*rand(1);
exp.NextObservation = {obsInfo.UpperLimit.*rand(3,1)};
exp.IsDone = 0;
```

Before appending experience to the buffer, you can validate whether the experience is compatible with the buffer. The `validateExperience` function generates an error if the expereince is incompatible with the buffer.

```
validateExperience(buffer,exp)
```

Append the experience to the buffer.

```
append(buffer,exp);
```

You can also append a batch of experiences to the experience buffer using a structure array. For this example, append a sequence of 100 random experiences, with the final experience representing a terminal condition.

```
for i = 1:100
    expBatch(i).Observation = {obsInfo.UpperLimit.*rand(3,1)};
    expBatch(i).Action = {actInfo.UpperLimit.*rand(2,1)};
    expBatch(i).Reward = 10*rand(1);
    expBatch(i).NextObservation = {obsInfo.UpperLimit.*rand(3,1)};
    expBatch(i).IsDone = 0;
end
expBatch(100).IsDone = 1;

validateExperience(buffer,expBatch)

append(buffer,expBatch);
```

After appending experiences to the buffer, you can sample mini-batches of experiences for training of your RL agent. For example, randomly sample a batch of 50 experiences from the buffer.

```
miniBatch = sample(buffer,50);
```

You can sample a horizon of data from the buffer. For example, sample a horizon of 10 consecutive experiences with a discount factor of 0.95.

```
horizonSample = sample(buffer,1,...
    NStepHorizon=10,...
    DiscountFactor=0.95);
```

The returned sample includes the following information.

- `Observation` and `Action` are the observation and action from the first experience in the horizon.
- `NextObservation` and `IsDone` are the next observation and termination signal from the final experience in the horizon.
- `Reward` is the cumulative reward across the horizon using the specified discount factor.

You can also sample a sequence of consecutive experiences. In this case, the structure fields contain arrays with values for all sampled experiences.

```
sequenceSample = sample(buffer,1,...
    SequenceLength=20);
```

Create Experience Buffer with Multiple Observation Channels

Define observation specifications for the environment. For this example, assume that the environment has two observation channels: one channel with two continuous observations and one channel with a three-valued discrete observation.

```
obsContinuous = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[1;5]);
obsDiscrete = rlFiniteSetSpec([1 2 3]);
obsInfo = [obsContinuous obsDiscrete];
```

Define action specifications for the environment. For this example, assume that the environment has a single action channel with one continuous action in a specified range.

```
actInfo = rlNumericSpec([2 1],...
    LowerLimit=0,...
    UpperLimit=[5;10]);
```

Create an experience buffer with a maximum length of 5,000.

```
buffer = rlReplayMemory(obsInfo,actInfo,5000);
```

Append a sequence of 50 random experiences to the buffer.

```
for i = 1:50
    exp(i).Observation = ...
        {obsInfo(1).UpperLimit.*rand(2,1) randi(3)};
    exp(i).Action = {actInfo.UpperLimit.*rand(2,1)};
    exp(i).NextObservation = ...
        {obsInfo(1).UpperLimit.*rand(2,1) randi(3)};
    exp(i).Reward = 10*rand(1);
    exp(i).IsDone = 0;
```

```
end

append(buffer,exp);
```

After appending experiences to the buffer, you can sample mini-batches of experiences for training of your RL agent. For example, randomly sample a batch of 10 experiences from the buffer.

```
miniBatch = sample(buffer,10);
```

Resize Experience Buffer in Reinforcement Learning Agent

Create an environment for training the agent. For this example, load a predefined environment.

```
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");
```

Extract the observation and action specifications from the agent.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a DQN agent from the environment specifications.

```
agent = rlDQNAgent(obsInfo,actInfo);
```

By default, the agent uses an experience buffer with a maximum size of 10,000.

```
agent.ExperienceBuffer
```

```
ans =
  rlReplayMemory with properties:

    MaxLength: 10000
       Length: 0
```

Increase the maximum size of the experience buffer to 20,000.

```
resize(agent.ExperienceBuffer,20000)
```

View the updated experience buffer.

```
agent.ExperienceBuffer
```

```
ans =
  rlReplayMemory with properties:

    MaxLength: 20000
       Length: 0
```

Version History

Introduced in R2022a

See Also

Objects
`rlPrioritizedReplayMemory` | `rlHindsightReplayMemory` | `rlHindsightPrioritizedReplayMemory`

rlRepresentationOptions

(Not recommended) Options set for reinforcement learning agent representations (critics and actors)

Note `rlRepresentationOptions` is not recommended. Use an `rlOptimizerOptions` object within an agent options object instead. For more information, see “rlRepresentationOptions is not recommended”.

Description

Use an `rlRepresentationOptions` object to specify an options set for critics (`rlValueRepresentation`, `rlQValueRepresentation`) and actors (`rlDeterministicActorRepresentation`, `rlStochasticActorRepresentation`).

Creation

Syntax

```
repOpts = rlRepresentationOptions
repOpts = rlRepresentationOptions(Name,Value)
```

Description

`repOpts = rlRepresentationOptions` creates a default option set to use as a last argument when creating a reinforcement learning actor or critic. You can modify the object properties using dot notation.

`repOpts = rlRepresentationOptions(Name,Value)` creates an options set with the specified “Properties” on page 3-345 using one or more name-value pair arguments.

Properties

`LearnRate` — Learning rate for the representation
`0.01` (default) | positive scalar

Learning rate for the representation, specified as a positive scalar. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Example: `'LearnRate',0.025`

`Optimizer` — Optimizer for representation
`"adam"` (default) | `"sgdm"` | `"rmsprop"`

Optimizer for training the network of the representation, specified as one of the following values.

- `"adam"` — Use the Adam optimizer. You can specify the decay rates of the gradient and squared gradient moving averages using the `GradientDecayFactor` and `SquaredGradientDecayFactor` fields of the `OptimizerParameters` option.
- `"sgdm"` — Use the stochastic gradient descent with momentum (SGDM) optimizer. You can specify the momentum value using the `Momentum` field of the `OptimizerParameters` option.
- `"rmsprop"` — Use the RMSProp optimizer. You can specify the decay rate of the squared gradient moving average using the `SquaredGradientDecayFactor` fields of the `OptimizerParameters` option.

For more information about these optimizers, see “Stochastic Gradient Descent” in the Algorithms section of `trainingOptions` in Deep Learning Toolbox.

Example: `'Optimizer',"sgdm"`

`OptimizerParameters` — Applicable parameters for optimizer

`OptimizerParameters` object

Applicable parameters for the optimizer, specified as an `OptimizerParameters` object with the following parameters.

Parameter	Description
`Momentum`	Contribution of previous step, specified as a scalar from 0 to 1. A value of 0 means no contribution from the previous step. A value of 1 means maximal contribution. This parameter applies only when `Optimizer` is `"sgdm"`. In that case, the default value is 0.9. This default value works well for most problems.
`Epsilon`	Denominator offset, specified as a positive scalar. The optimizer adds this offset to the denominator in the network parameter updates to avoid division by zero. This parameter applies only when `Optimizer` is `"adam"` or `"rmsprop"`. In that case, the default value is 10^{-8}. This default value works well for most problems.
`GradientDecayFactor`	Decay rate of gradient moving average, specified as a positive scalar from 0 to 1. This parameter applies only when `Optimizer` is `"adam"`. In that case, the default value is 0.9. This default value works well for most problems.
`SquaredGradientDecayFactor`	Decay rate of squared gradient moving average, specified as a positive scalar from 0 to 1. This parameter applies only when `Optimizer` is `"adam"` or `"rmsprop"`. In that case, the default value is 0.999. This default value works well for most problems.

When a particular property of OptimizerParameters is not applicable to the optimizer type specified in the Optimizer option, that property is set to "Not applicable".

To change the default values, create an rlRepresentationOptions set and use dot notation to access and change the properties of OptimizerParameters.

```
repOpts = rlRepresentationOptions;
repOpts.OptimizerParameters.GradientDecayFactor = 0.95;
```

GradientThreshold — Threshold value for gradient

Inf (default) | positive scalar

Threshold value for the representation gradient, specified as Inf or a positive scalar. If the gradient exceeds this value, the gradient is clipped as specified by the GradientThresholdMethod option. Clipping the gradient limits how much the network parameters change in a training iteration.

Example: 'GradientThreshold',1

GradientThresholdMethod — Gradient threshold method

"l2norm" (default) | "global-l2norm" | "absolute-value"

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as one of the following values.

- "l2norm" — If the L_2 norm of the gradient of a learnable parameter is larger than GradientThreshold, then scale the gradient so that the L_2 norm equals GradientThreshold.
- "global-l2norm" — If the global L_2 norm, L, is larger than GradientThreshold, then scale all gradients by a factor of GradientThreshold/L. The global L_2 norm considers all learnable parameters.
- "absolute-value" — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than GradientThreshold, then scale the partial derivative to have magnitude equal to GradientThreshold and retain the sign of the partial derivative.

For more information, see “Gradient Clipping” in the Algorithms section of trainingOptions in Deep Learning Toolbox.

Example: 'GradientThresholdMethod',"absolute-value"

L2RegularizationFactor — Factor for L_2 regularization

0.0001 (default) | nonnegative scalar

Factor for L_2 regularization (weight decay), specified as a nonnegative scalar. For more information, see “L2 Regularization” in the Algorithms section of trainingOptions in Deep Learning Toolbox.

To avoid overfitting when using a representation with many parameters, consider increasing the L2RegularizationFactor option.

Example: 'L2RegularizationFactor',0.0005

UseDevice — Computation device for training

"cpu" (default) | "gpu"

Computation device used to perform deep neural network operations such as gradient computation, parameter update and prediction during training. It is specified as either "cpu" or "gpu".

The "gpu" option requires both Parallel Computing Toolbox software and a CUDA enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use gpuDevice (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating an agent on a GPU involves device-specific numerical round off errors. These errors can produce different results compared to performing the same operations a CPU.

Note that if you want to use parallel processing to speed up training, you do not need to set UseDevice. Instead, when training your agent, use an rlTrainingOptions object in which the UseParallel option is set to true. For more information about training using multicore processors and GPUs for training, see “Train Agents Using Parallel Computing and GPUs”.

Example: 'UseDevice',"gpu"

Object Functions

rlValueRepresentation	(Not recommended) Value function critic representation for reinforcement learning agents
rlQValueRepresentation	(Not recommended) Q-Value function critic representation for reinforcement learning agents
rlDeterministicActorRepresentation	(Not recommended) Deterministic actor representation for reinforcement learning agents
rlStochasticActorRepresentation	(Not recommended) Stochastic actor representation for reinforcement learning agents

Examples

Configure Options for Creating Representation

Create an options set for creating a critic or actor representation for a reinforcement learning agent. Set the learning rate for the representation to 0.05, and set the gradient threshold to 1. You can set the options using Name,Value pairs when you create the options set. Any options that you do not explicitly set have their default values.

```
repOpts = rlRepresentationOptions(LearnRate=5e-2,...
                                  GradientThreshold=1)
```

```
repOpts = 
  rlRepresentationOptions with properties:

                    LearnRate: 0.0500
            GradientThreshold: 1
    GradientThresholdMethod: "l2norm"
     L2RegularizationFactor: 1.0000e-04
                    UseDevice: "cpu"
                    Optimizer: "adam"
          OptimizerParameters: [1x1 rl.option.OptimizerParameters]
```

Alternatively, create a default options set and use dot notation to change some of the values.

```
repOpts = rlRepresentationOptions;
repOpts.LearnRate = 5e-2;
repOpts.GradientThreshold = 1
```

```
repOpts = 
  rlRepresentationOptions with properties:

                  LearnRate: 0.0500
          GradientThreshold: 1
    GradientThresholdMethod: "l2norm"
     L2RegularizationFactor: 1.0000e-04
                  UseDevice: "cpu"
                  Optimizer: "adam"
        OptimizerParameters: [1x1 rl.option.OptimizerParameters]
```

If you want to change the properties of the `OptimizerParameters` option, use dot notation to access them.

```
repOpts.OptimizerParameters.Epsilon = 1e-7;
repOpts.OptimizerParameters
```

```
ans = 
  OptimizerParameters with properties:

                      Momentum: "Not applicable"
                       Epsilon: 1.0000e-07
           GradientDecayFactor: 0.9000
    SquaredGradientDecayFactor: 0.9990
```

Version History

Introduced in R2019a

R2022a: `rlRepresentationOptions` is not recommended

Not recommended starting in R2022a

`rlRepresentationOptions` objects are no longer recommended. To specify optimization options for actors and critics, use `rlOptimizerOptions` objects instead.

Specifically, you can create an agent options object and set its `CriticOptimizerOptions` and `ActorOptimizerOptions` properties to suitable `rlOptimizerOptions` objects. Then you pass the agent options object to the function that creates the agent. This workflow is shown in the following table.

`rlRepresentationOptions`: Not Recommended	`rlOptimizerOptions`: Recommended

`crtOpts = rlRepresentationOptions(...`
`'GradientThreshold',1);`
 	
`critic = rlValueRepresentation(...`
`net,obsInfo,'Observation',{'obs'},crtOpts);`
	`criticOpts = rlOptimizerOptions(...`
	`'GradientThreshold',1);`
	
	`agentOpts = rlACAgentOptions(...`
	`'CriticOptimizerOptions',crtOpts);`
	
	`agent = rlACAgent(actor,critic,agentOpts)`

Alternatively, you can create the agent and then use dot notation to access the optimization options for the agent actor and critic, for example: `agent.AgentOptions.ActorOptimizerOptions.GradientThreshold = 1;`.

See Also

Topics
“Create Policies and Value Functions”
“Reinforcement Learning Agents”

rlSACAgent

Soft actor-critic (SAC) reinforcement learning agent

Description

The soft actor-critic (SAC) algorithm is a model-free, online, off-policy, actor-critic reinforcement learning method. The SAC algorithm computes an optimal policy that maximizes both the long-term expected reward and the entropy of the policy. The policy entropy is a measure of policy uncertainty given the state. A higher entropy value promotes more exploration. Maximizing both the reward and the entropy balances exploration and exploitation of the environment. The action space can only be continuous.

For more information, see “Soft Actor-Critic (SAC) Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlSACAgent(observationInfo,actionInfo)
agent = rlSACAgent(observationInfo,actionInfo,initOptions)

agent = rlSACAgent(actor,critics)

agent = rlSACAgent( ___ ,agentOptions)
```

Description

Create Agent from Observation and Action Specifications

`agent = rlSACAgent(observationInfo,actionInfo)` creates a SAC agent for an environment with the given observation and action specifications, using default initialization options. The actor and critics in the agent use default deep neural networks built using the observation specification `observationInfo` and action specification `actionInfo`. The `ObservationInfo` and `ActionInfo` properties of `agent` are set to the `observationInfo` and `actionInfo` input arguments, respectively.

`agent = rlSACAgent(observationInfo,actionInfo,initOptions)` creates a SAC agent with deep neural networks configured using the specified initialization options (`initOptions`).

Create Agent from Actor and Critic

`agent = rlSACAgent(actor,critics)` creates a SAC agent with the specified actor and critic networks and default agent options.

Specify Agent Options

`agent = rlSACAgent( ___ ,agentOptions)` sets the AgentOptions property for any of the previous syntaxes.

Input Arguments

`initOptions` — Agent initialization options

`rlAgentInitializationOptions` object

Agent initialization options, specified as an `rlAgentInitializationOptions` object.

`actor` — Actor

`rlContinuousGaussianActor` object

Actor that implements the policy, specified as an `rlContinuousGaussianActor` function approximator object. For more information on creating actor approximators, see “Create Policies and Value Functions”.

Note A SAC agent automatically reads the action range from the `UpperLimit` and `LowerLimit` properties of the action specification (which is used to create the actor), and then internally scales the distribution and bounds the action. Therefore, do not add a `tanhLayer` as the last nonlinear layer in the mean output path. If you bound the mean value output directly (for example by adding a `tanhLayer` right before the output), the agent does not calculate the entropy of the probability density distribution correctly. Note that you must still add a softplus or ReLU layer to the standard deviations path to enforce nonnegativity. For more information, see “Soft Actor-Critic (SAC) Agents”.

`critics` — Critic

`rlQValueFunction` object | two-element row vector of `rlQValueFunction` objects

Critic, specified as one of the following:

- `rlQValueFunction` object — Create a SAC agent with a single Q-value function.
- Two-element row vector of `rlQValueFunction` objects — Create a SAC agent with two critic value functions. The two critic must be unique `rlQValueFunction` objects with the same observation and action specifications. The critics can either have different structures or the same structure but with different initial parameters.

For a SAC agent, each critic must be a single-output `rlQValueFunction` object that takes both the action and observations as inputs.

For more information on creating critics, see “Create Policies and Value Functions”.

Properties

`ObservationInfo` — Observation specifications

specification object | array of specification objects

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

If you create the agent by specifying an actor and critic, the value of `ObservationInfo` matches the value specified in the actor and critic objects.

You can extract `observationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`ActionInfo` — Action specification

`rlNumericSpec` object

Action specifications, specified as an `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

If you create the agent by specifying an actor and critic, the value of `ActionInfo` matches the value specified in the actor and critic objects.

You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`AgentOptions` — Agent options

`rlSACAgentOptions` object

Agent options, specified as an `rlSACAgentOptions` object.

If you create a SAC agent with default actor and critic that use recurrent neural networks, the default value of `AgentOptions.SequenceLength` is `32`.

`ExperienceBuffer` — Experience buffer

`rlReplayMemory` object

Experience buffer, specified as an `rlReplayMemory` object. During training the agent stores each of its experiences (*S*,*A*,*R*,*S*',*D*) in a buffer. Here:

- *S* is the current observation of the environment.
- *A* is the action taken by the agent.
- *R* is the reward for taking action *A*.
- *S*' is the next observation after taking action *A*.
- *D* is the is-done signal after taking action *A*.

`UseExplorationPolicy` — Option to use exploration policy

`true` (default) | `false`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- `true` — Use the base agent exploration policy when selecting actions in `sim` and `generatePolicyFunction`. In this case, the agent selects its actions by sampling its probability distribution, the policy is therefore stochastic and the agent explores its observation space.

- `false` — Use the base agent greedy policy (the action with maximum likelihood) when selecting actions in `sim` and `generatePolicyFunction`. In this case, the simulated agent and generated policy behave deterministically.

Note This option affects only simulation and deployment; it does not affect training.

`SampleTime` — Sample time of agent

positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations. The value of `SampleTime` matches the value specified in `AgentOptions`.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent
setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent
setCritic	Set critic of reinforcement learning agent
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create SAC Agent from Observation and Action Specifications

Create environment and obtain observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force, applied to the mass, ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a SAC agent from the environment observation and action specifications.

```
agent = rlSACAgent(obsInfo,actInfo);
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension)})
```

```
ans = 1x1 cell array
    {[0.0546]}
```

You can now test and train the agent within the environment. You can also use `getActor` and `getCritic` to extract the actor and critic, respectively, and `getModel` to extract the approximator model (by default a deep neural network) from the actor or critic.

Create SAC Agent Using Initialization Options

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force, applied to the mass, ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create an agent initialization option object, specifying that each hidden fully connected layer in the network must have 128 neurons.

```
initOpts = rlAgentInitializationOptions(NumHiddenUnit=128);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a SAC agent from the environment observation and action specifications using the initialization options.

```
agent = rlSACAgent(obsInfo,actInfo,initOpts);
```

Extract the deep neural network from the actor.

```
actorNet = getModel(getActor(agent));
```

Extract the deep neural networks from the two critics. Note that `getModel(critics)` only returns the first critic network.

```
critics = getCritic(agent);
criticNet1 = getModel(critics(1));
criticNet2 = getModel(critics(2));
```

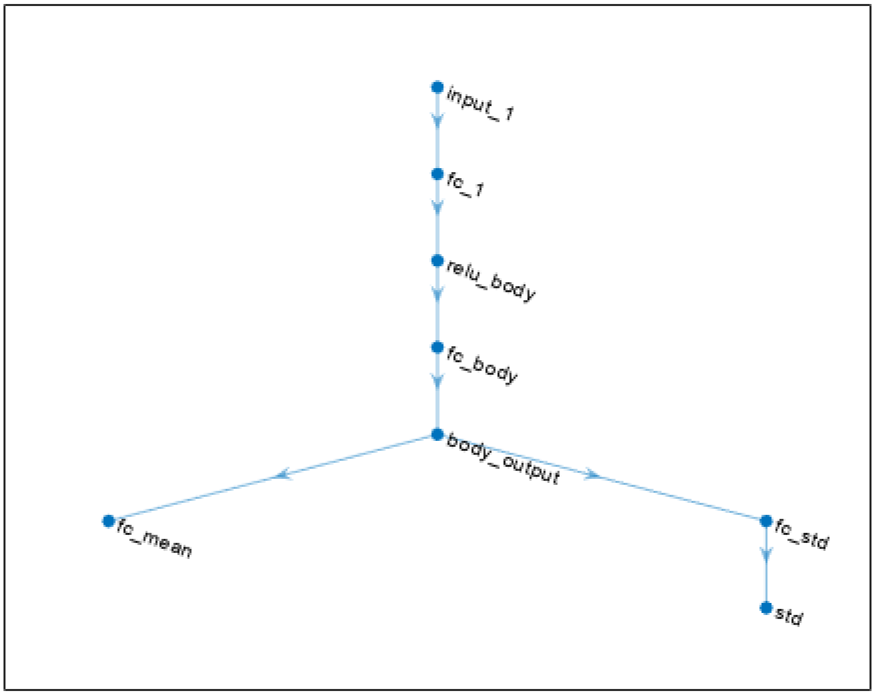
Display the layers of the first critic network, and verify that each hidden fully connected layer has 128 neurons.

```
criticNet1.Layers
```

```
ans = 
  9x1 Layer array with layers:

     1   'concat'         Concatenation     Concatenation of 2 inputs along dimension 1
     2   'relu_body'      ReLU              ReLU
     3   'fc_body'        Fully Connected   128 fully connected layer
     4   'body_output'    ReLU              ReLU
     5   'input_1'        Feature Input     2 features
     6   'fc_1'           Fully Connected   128 fully connected layer
     7   'input_2'        Feature Input     1 features
     8   'fc_2'           Fully Connected   128 fully connected layer
     9   'output'         Fully Connected   1 fully connected layer
```

Plot the networks of the actor and of the second critic, and display the number of weights.

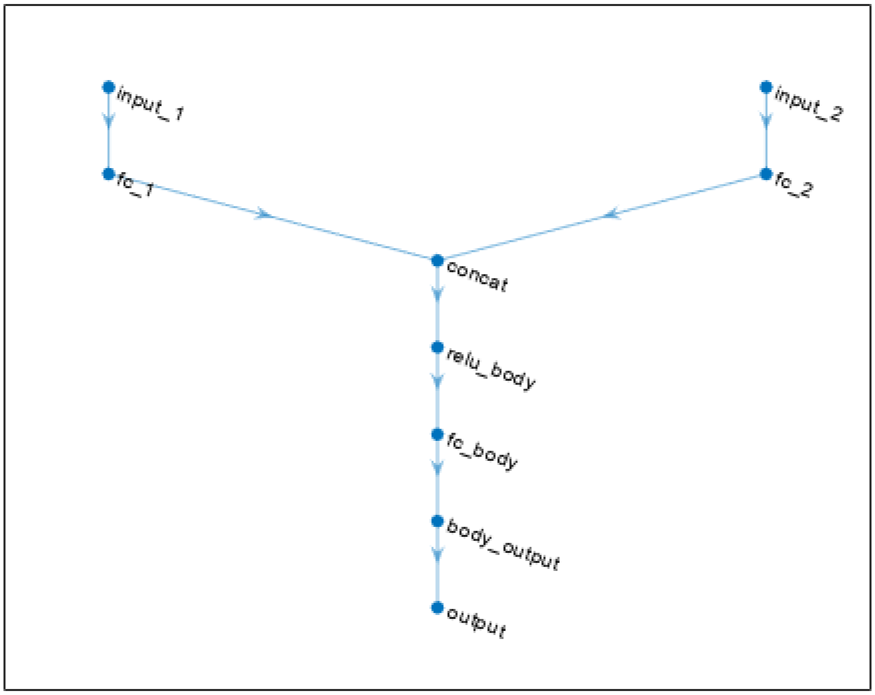
```
plot(layerGraph(actorNet))
```



```
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 17.1k
   Inputs:
      1   'input_1'   2 features
```

```
plot(layerGraph(criticNet2))
```



```
summary(criticNet2)
```

```
   Initialized: true

   Number of learnables: 33.6k

   Inputs:
      1   'input_1'   2 features
      2   'input_2'   1 features
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension)})
```

```
ans = 1x1 cell array
    {[-0.9867]}
```

You can now test and train the agent within the environment.

Create SAC Agent from Actor and Critics

Create an environment and obtain observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observations from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force, applied to the mass, ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Define bounds on the action. The SAC agent automatically uses these values to internally scale the distribution and bound the action properly.

```
actInfo.LowerLimit=-2;
actInfo.UpperLimit=2;
```

SAC agents use two Q-value function critics. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critics, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` return the number of dimensions of the observation and action spaces, respectively, regardless of whether they are arranged as row vectors, column vectors, or matrices.

Define each network path as an array of layer objects. Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input layers with the appropriate environment channel.

```
% Observation path
obsPath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="obsPathIn")
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(16,Name="obsPathOut")
    ];

% Action path
actPath = [
    featureInputLayer(prod(actInfo.Dimension),Name="actPathIn")
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(16,Name="actPathOut")
    ];

% Common path
commonPath = [
    concatenationLayer(1,2,Name="concat")
    reluLayer
    fullyConnectedLayer(1)
    ];
```

```
% Add layers to layergraph object
criticNet = layerGraph;
criticNet = addLayers(criticNet,obsPath);
criticNet = addLayers(criticNet,actPath);
criticNet = addLayers(criticNet,commonPath);

% Connect layers
criticNet = connectLayers(criticNet,"obsPathOut","concat/in1");
criticNet = connectLayers(criticNet,"actPathOut","concat/in2");
```

To initialize the network weights differently for the two critics, create two different `dlnetwork` objects. You must do this because the agent constructor function does not accept two identical critics.

```
criticNet1 = dlnetwork(criticNet);
criticNet2 = dlnetwork(criticNet);
```

Display the number of weights.

```
summary(criticNet1)
```

```
   Initialized: true

   Number of learnables: 1.2k

   Inputs:
      1   'obsPathIn'   2 features
      2   'actPathIn'   1 features
```

Create the two critics using the two networks with different weights and the names of the input layers. Alternatively, if you use exactly the same network with the same weights, you must explicitly initialize the network each time (to make sure weights are initialized differently) before passing it to `rlQValueFunction`. To do so, use `initialize`.

```
critic1 = rlQValueFunction(criticNet1,obsInfo,actInfo, ...
    ActionInputNames="actPathIn",ObservationInputNames="obsPathIn");

critic2 = rlQValueFunction(criticNet2,obsInfo,actInfo, ...
    ActionInputNames="actPathIn",ObservationInputNames="obsPathIn");
```

For more information about value function approximators, see `rlQValueFunction`.

Check the critics with a random observation and action input.

```
getValue(critic1,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.1330
```

```
getValue(critic2,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.1526
```

SAC agents use a parametrized stochastic policy, which for continuous action spaces is implemented by a continuous Gaussian actor. This actor takes an observation as input and returns as output a random action sampled from a Gaussian probability distribution.

To approximate the mean values and standard deviations of the Gaussian distribution, you must use a neural network with two output layers, each having as many elements as the dimension of the action space. One output layer must return a vector containing the mean values for each action dimension. The other must return a vector containing the standard deviation for each action dimension.

The SAC agent automatically reads the action range from the `UpperLimit` and `LowerLimit` properties of `actInfo` (which is used to create the actor), and then internally scales the distribution and bounds the action.

Therefore, do not add a `tanhLayer` as the last nonlinear layer in the mean output path. If you bound the mean value output directly (for example by adding a `tanhLayer` right before the output), the agent does not calculate the entropy of the probability density distribution correctly. Note that you must still add a softplus or ReLU layer to the standard deviations path to enforce nonnegativity. For more information, see “Soft Actor-Critic (SAC) Agents”.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path.

```
% Define common input path
commonPath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="netObsIn")
    fullyConnectedLayer(32)
    reluLayer(Name="CommonRelu")
    ];

% Define path for mean value
meanPath = [
    fullyConnectedLayer(32,Name="meanIn")
    reluLayer
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension),Name="MeanOut")
    ];

% Define path for standard deviation
stdPath = [
    fullyConnectedLayer(32,Name="stdIn")
    reluLayer
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension))
    softplusLayer(Name="StandardDeviationOut")
    ];

% Add layers to layerGraph object
actorNet = layerGraph(commonPath);
actorNet = addLayers(actorNet,meanPath);
actorNet = addLayers(actorNet,stdPath);

% Connect layers
actorNet = connectLayers(actorNet,"CommonRelu","meanIn/in");
actorNet = connectLayers(actorNet,"CommonRelu","stdIn/in");

% Convert to dlnetwork and display the number of weights.
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true
   Number of learnables: 3.2k

   Inputs:
      1   'netObsIn'   2 features
```

Create the actor using `actorNet`, the observation and action specification objects, and the names of the input and output layers.

```
actor = rlContinuousGaussianActor(actorNet, obsInfo, actInfo, ...
    ActionMeanOutputNames="MeanOut",...
    ActionStandardDeviationOutputNames="StandardDeviationOut",...
    ObservationInputNames="netObsIn");
```

For more information about continuous Gaussian actors approximators, see `rlContinuousGaussianActor`.

Check your actor with a random input observation.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-0.8205]}
```

Specify training options for the critics.

```
criticOptions = rlOptimizerOptions( ...
    Optimizer="adam", ...
    LearnRate=1e-3,...
    GradientThreshold=1, ...
    L2RegularizationFactor=2e-4);
```

Specify training options for the actor.

```
actorOptions = rlOptimizerOptions( ...
    Optimizer="adam", ...
    LearnRate=1e-3,...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-5);
```

Specify agent options, including training options for actor and critics.

```
agentOptions = rlSACAgentOptions;
agentOptions.SampleTime = env.Ts;
agentOptions.DiscountFactor = 0.99;
agentOptions.TargetSmoothFactor = 1e-3;
agentOptions.ExperienceBufferLength = 1e6;
agentOptions.MiniBatchSize = 32;

agentOptions.CriticOptimizerOptions = criticOptions;
agentOptions.ActorOptimizerOptions = actorOptions;
```

Create the SAC agent using actor, critics, and options.

```
agent = rlSACAgent(actor,[critic1 critic2],agentOptions)
```

```
agent = 
  rlSACAgent with properties:
        ExperienceBuffer: [1x1 rl.replay.rlReplayMemory]
            AgentOptions: [1x1 rl.option.rlSACAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlNumericSpec]
              SampleTime: 0.1000
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension)})
```

```
ans = 1x1 cell array
    {[0.4490]}
```

You can now test and train the agent within the environment.

Create SAC Agent using Recurrent Neural Networks

For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observations from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force, applied to the mass, ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

SAC agents use two Q-value function critics. To model the parametrized Q-value function within the critics, use a recurrent neural network, which must have two input layers one output layer.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path. To create a recurrent neural network, use `sequenceInputLayer` as the input layer and include an `lstmLayer` as one of the other network layers.

```
% Define observation path
obsPath = [
    sequenceInputLayer(prod(obsInfo.Dimension),Name="obsIn")
    fullyConnectedLayer(40)
    reluLayer
    fullyConnectedLayer(30,Name = "obsOut")
    ];

% Define action path
actPath = [
    sequenceInputLayer(prod(actInfo.Dimension),Name="actIn")
    fullyConnectedLayer(30,Name="actOut")
    ];

% Define common path
commonPath = [
    concatenationLayer(1,2,Name="cat")
    lstmLayer(16)
    reluLayer
    fullyConnectedLayer(1)
    ];

% Add layers to layergraph object
criticNet = layerGraph(obsPath);
criticNet = addLayers(criticNet,actPath);
criticNet = addLayers(criticNet,commonPath);

% Connect paths
criticNet = connectLayers(criticNet,"obsOut","cat/in1");
criticNet = connectLayers(criticNet,"actOut","cat/in2");
```

To initialize the network weights differently for the two critics, create two different `dlnetwork` objects. You must do this because if the agent constructor function does not accept two identical critics.

```
criticNet1 = dlnetwork(criticNet);
criticNet2 = dlnetwork(criticNet);
```

Display the number of weights.

```
summary(criticNet1)
```

```
   Initialized: true

   Number of learnables: 6.3k

   Inputs:
      1   'obsIn'   Sequence input with 2 dimensions
      2   'actIn'   Sequence input with 1 dimensions
```

Create the two critics using the two networks with different weights. Use the same network structure for both critics. The SAC agent initializes the two networks using different default parameters.

```
critic1 = rlQValueFunction(criticNet1,obsInfo,actInfo);
critic2 = rlQValueFunction(criticNet2,obsInfo,actInfo);
```

Check the critics with a random observation and action input.

```
getValue(critic1,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.0508
```

```
getValue(critic2,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    0.0762
```

Since the critic has a recurrent network, the actor must have a recurrent network too.

Do not add a `tanhLayer` or `scalingLayer` in the mean output path. The SAC agent internally transforms the unbounded Gaussian distribution to the bounded distribution to compute the probability density function and entropy properly. However, add a softplus or ReLU layer to the standard deviations path to enforce nonnegativity,

Define each network path as an array of layer objects and specify a name for the input and output layers, so you can later explicitly associate them with the appropriate channel.

```
% Define common path
commonPath = [
    sequenceInputLayer(prod(obsInfo.Dimension),Name="obsIn")
    fullyConnectedLayer(400)
    lstmLayer(8)
    reluLayer(Name="CommonOut")];

meanPath = [
    fullyConnectedLayer(300,Name="MeanIn")
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension),Name="Mean")
    ];

stdPath = [
    fullyConnectedLayer(300,Name="StdIn")
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension))
    softplusLayer(Name="StandardDeviation")
    ];

actorNet = layerGraph(commonPath);
actorNet = addLayers(actorNet,meanPath);
actorNet = addLayers(actorNet,stdPath);

actorNet = connectLayers(actorNet,"CommonOut","MeanIn/in");
actorNet = connectLayers(actorNet,"CommonOut","StdIn/in");

% Convert to dlnetwork and display the number of weights.
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 20.2k

   Inputs:
      1   'obsIn'   Sequence input with 2 dimensions
```

Create the actor using `actorNet`, the observation and action specification objects, and the names of the input and output layers.

```
actor = rlContinuousGaussianActor(actorNet, obsInfo, actInfo, ...
    ActionMeanOutputNames="Mean",...
    ActionStandardDeviationOutputNames="StandardDeviation",...
    ObservationInputNames="obsIn");
```

Check your actor with a random input observation.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-0.6304]}
```

Specify training options for the critics.

```
criticOptions = rlOptimizerOptions( ...
    Optimizer = "adam", LearnRate = 1e-3,...
    GradientThreshold = 1, L2RegularizationFactor = 2e-4);
```

Specify training options for the actor.

```
actorOptions = rlOptimizerOptions( ...
    Optimizer = "adam", LearnRate = 1e-3,...
    GradientThreshold = 1, L2RegularizationFactor = 1e-5);
```

Specify agent options. To use a recurrent neural network, you must specify a `SequenceLength` greater than 1.

```
agentOptions = rlSACAgentOptions;
agentOptions.SampleTime = env.Ts;
agentOptions.DiscountFactor = 0.99;
agentOptions.TargetSmoothFactor = 1e-3;
agentOptions.ExperienceBufferLength = 1e6;
agentOptions.SequenceLength = 32;
agentOptions.MiniBatchSize = 32;
```

Create SAC agent using actor, critics, and options.

```
agent = rlSACAgent(actor,[critic1 critic2],agentOptions);
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-0.8774]}
```

To evaluate the agent using sequential observations, use the sequence length (time) dimension. For example, obtain actions for a sequence of 9 observations.

```
[action,state] = getAction(agent, ...
    {rand([obsInfo.Dimension 1 9])});
```

Display the action corresponding to the seventh element of the observation.

```
action = action{1};
action(1,1,1,7)
```

```
ans = 0.5114
```

You can now test and train the agent within the environment.

Version History

Introduced in R2020b

See Also

Apps

Reinforcement Learning Designer | Deep Network Designer

Functions

`getAction` | `getActor` | `getCritic` | `getModel` | `generatePolicyFunction` | `generatePolicyBlock` | `getActionInfo` | `getObservationInfo`

Objects
rlSACAgentOptions | rlAgentInitializationOptions | rlQValueFunction | rlContinuousGaussianActor | rlDDPGAgent | rlTD3Agent | rlACAgent | rlPPOAgent

Blocks
RL Agent | Policy

Topics
"Train SAC Agent for Ball Balance Control"
"Train Reinforcement Learning Agents"
"Soft Actor-Critic (SAC) Agents"
"Reinforcement Learning Agents"

rlSACAgentOptions

Options for SAC agent

Description

Use an `rlSACAgentOptions` object to specify options for soft actor-critic (SAC) agents. To create a SAC agent, use `rlSACAgent`.

For more information, see “Soft Actor-Critic (SAC) Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlSACAgentOptions
opt = rlSACAgentOptions(Name,Value)
```

Description

`opt = rlSACAgentOptions` creates an options object for use as an argument when creating a SAC agent using all default options. You can modify the object properties using dot notation.

`opt = rlSACAgentOptions(Name,Value)` sets option properties on page 3-120 using name-value pairs. For example, `rlSACAgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`EntropyWeightOptions` — Entropy tuning options
`EntropyWeightOptions` object

Entropy tuning options, specified as an `EntropyWeightOptions` object with the following properties.

`EntropyWeight` — Initial entropy component weight
`1` (default) | positive scalar

Initial entropy component weight, specified as a positive scalar.

`LearnRate` — Optimizer learning rate
`3e-4` (default) | nonnegative scalar

Optimizer learning rate, specified as a nonnegative scalar. If `LearnRate` is zero, the `EntropyWeight` value is fixed during training and the `TargetEntropy` value is ignored.

TargetEntropy — Target entropy value

[ ] (default) | scalar

Target entropy value for tuning entropy weight, specified as a scalar. A higher target entropy value encourages more exploration.

If you do not specify TargetEntropy, the agent uses –*A* as the target value, where *A* is the number of actions.

Algorithm — Algorithm to tune entropy

"adam" (default) | "sgdm" | "rmsprop"

Algorithm to tune entropy, specified as one of the following strings.

- "adam" — Use the Adam optimizer. You can specify the decay rates of the gradient and squared gradient moving averages using the GradientDecayFactor and SquaredGradientDecayFactor fields of the OptimizerParameters option.
- "sgdm" — Use the stochastic gradient descent with momentum (SGDM) optimizer. You can specify the momentum value using the Momentum field of the OptimizerParameters option.
- "rmsprop" — Use the RMSProp optimizer. You can specify the decay rate of the squared gradient moving average using the SquaredGradientDecayFactor fields of the OptimizerParameters option.

For more information about these optimizers, see “Stochastic Gradient Descent” in Deep Learning Toolbox.

GradientThreshold — Threshold value for gradient

Inf (default) | positive scalar

Threshold value for the entropy gradient, specified as Inf or a positive scalar. If the gradient exceeds this value, the gradient is clipped.

OptimizerParameters — Applicable parameters for optimizer

OptimizerParameters object

Applicable parameters for the optimizer, specified as an OptimizerParameters object with the following parameters. The default parameter values work well for most problems.

Parameter	Description	Default
Momentum	Contribution of previous step, specified as a scalar from 0 to 1. A value of 0 means no contribution from the previous step. A value of 1 means maximal contribution. This parameter applies only when Optimizer is "sgdm".	0.9

Parameter	Description	Default
`Epsilon`	Denominator offset, specified as a positive scalar. The optimizer adds this offset to the denominator in the network parameter updates to avoid division by zero. This parameter applies only when `Optimizer` is `"adam"` or `"rmsprop"`.	`1e-8`
`GradientDecayFactor`	Decay rate of gradient moving average, specified as a positive scalar from 0 to 1. This parameter applies only when `Optimizer` is `"adam"`.	`0.9`
`SquaredGradientDecayFactor`	Decay rate of squared gradient moving average, specified as a positive scalar from 0 to 1. This parameter applies only when `Optimizer` is `"adam"` or `"rmsprop"`.	`0.999`

When a particular property of `OptimizerParameters` is not applicable to the optimizer type specified in the `Algorithm` option, that property is set to `"Not applicable"`.

To change the default values, access the properties of `OptimizerParameters` using dot notation.

```
opt = rlSACAgentOptions;
opt.EntropyWeightOptions.OptimizerParameters.GradientDecayFactor = 0.95;
```

`UseCriticTargetEntropy` — Option to use entropy in the critic targets

`true` (default) | `false`

Option to use entropy in the critic targets, specified as a either `true` (default, entropy is used) or `false` (entropy is not used). Note that this option does not affect entropy usage in the actor.

Example: `UseCriticTargetEntropy=false`

`PolicyUpdateFrequency` — Number of steps between actor policy updates

`1` (default) | positive integer

Number of steps between actor policy updates, specified as a positive integer. For more information, see “Training Algorithm”.

`CriticUpdateFrequency` — Number of steps between critic updates

`1` (default) | positive integer

Number of steps between critic updates, specified as a positive integer. For more information, see “Training Algorithm”.

`NumWarmStartSteps` — Number of actions to take before updating actor and critic

positive integer

Number of actions to take before updating actor and critics, specified as a positive integer. By default, the NumWarmStartSteps value is equal to the MiniBatchSize value.

NumGradientStepsPerUpdate — Number of gradient steps when updating actor and critics
1 (default) | positive integer

Number of gradient steps to take when updating actor and critics, specified as a positive integer.

ActorOptimizerOptions — Actor optimizer options
rlOptimizerOptions object

Actor optimizer options, specified as an rlOptimizerOptions object. It allows you to specify training parameters of the actor approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see rlOptimizerOptions and rlOptimizer.

CriticOptimizerOptions — Critic optimizer options
rlOptimizerOptions object

Critic optimizer options, specified as an rlOptimizerOptions object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see rlOptimizerOptions and rlOptimizer.

BatchDataRegularizerOptions — Batch data regularizer options
[] (default) | rlConservativeQLearningOptions object | rlBehaviorCloningRegularizerOptions object

Batch data regularizer options, specified as an rlConservativeQLearningOptions or as an rlBehaviorCloningRegularizerOptions object. These options are typically used to train the agent offline, from existing data. If you leave this option empty, no regularizer is used.

For more information, see rlConservativeQLearningOptions and rlBehaviorCloningRegularizerOptions.

Example: opt.BatchDataRegularizerOptions = rlConservativeQLearningOptions(NumSampledActions=20)

TargetSmoothFactor — Smoothing factor for target critic updates
1e-3 (default) | positive scalar less than or equal to 1

Smoothing factor for target critic updates, specified as a positive scalar less than or equal to 1. For more information, see “Target Update Methods”.

TargetUpdateFrequency — Number of steps between target critic updates
1 (default) | positive integer

Number of steps between target critic updates, specified as a positive integer. For more information, see “Target Update Methods”.

ResetExperienceBufferBeforeTraining — Option for clearing the experience buffer
false (default) | true

Option for clearing the experience buffer before training, specified as a logical value.

`SequenceLength` — Maximum batch-training trajectory length when using RNN

`1` (default) | positive integer

Maximum batch-training trajectory length when using a recurrent neural network, specified as a positive integer. This value must be greater than `1` when using a recurrent neural network and `1` otherwise.

`MiniBatchSize` — Size of random experience mini-batch

`64` (default) | positive integer

Size of random experience mini-batch, specified as a positive integer. During each training episode, the agent randomly samples experiences from the experience buffer when computing gradients for updating the actor and critics. Large mini-batches reduce the variance when computing gradients but increase the computational effort.

`NumStepsToLookAhead` — Number of future rewards used to estimate the value of the policy

`1` (default) | positive integer

Number of future rewards used to estimate the value of the policy, specified as a positive integer. For more information, see [1], Chapter 7.

`ExperienceBufferLength` — Experience buffer size

`10000` (default) | positive integer

Experience buffer size, specified as a positive integer. During training, the agent computes updates using a mini-batch of experiences randomly sampled from the buffer.

`SampleTime` — Sample time of agent

`1` (default) | positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

`DiscountFactor` — Discount factor

`0.99` (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlSACAgent	Soft actor-critic (SAC) reinforcement learning agent

Examples

Create SAC Agent Options Object

Create a SAC agent options object, specifying the discount factor.

```
opt = rlSACAgentOptions(DiscountFactor=0.95)
```

```
opt = 
  rlSACAgentOptions with properties:

                   EntropyWeightOptions: [1x1 rl.option.EntropyWeightOptions]
                 UseCriticTargetEntropy: 1
                  PolicyUpdateFrequency: 1
                  CriticUpdateFrequency: 1
                      NumWarmStartSteps: 64
              NumGradientStepsPerUpdate: 1
                  ActorOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
                 CriticOptimizerOptions: [1x2 rl.option.rlOptimizerOptions]
            BatchDataRegularizerOptions: []
                     TargetSmoothFactor: 1.0000e-03
                  TargetUpdateFrequency: 1
    ResetExperienceBufferBeforeTraining: 0
                         SequenceLength: 1
                          MiniBatchSize: 64
                    NumStepsToLookAhead: 1
                 ExperienceBufferLength: 10000
                             SampleTime: 1
                         DiscountFactor: 0.9500
                             InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

For SAC agents, configure the entropy weight optimizer using the options in `EntropyWeightOptions`. For example, set the target entropy value to –5.

```
opt.EntropyWeightOptions.TargetEntropy = -5;
```

Version History

Introduced in R2020b

R2022a: Simulation and deployment: `UseDeterministicExploitation` will be removed

Warns starting in R2022a

The property `UseDeterministicExploitation` of the `rlSACAgentOptions` object will be removed in a future release. Use the `UseExplorationPolicy` property of `rlSACAgent` instead.

Previously, you set `UseDeterministicExploitation` as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.AgentOptions.UseDeterministicExploitation = true;
  ```

- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.AgentOptions.UseDeterministicExploitation = false;
  ```

Starting in R2022a, set `UseExplorationPolicy` as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.UseExplorationPolicy = false;
  ```
- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.UseExplorationPolicy = true;
  ```

Similarly to `UseDeterministicExploitation`, `UseExplorationPolicy` affects only simulation and deployment; it does not affect training.

R2022a: The default value of the `ResetExperienceBufferBeforeTraining` property has changed

Behavior changed in R2022a

The default value of the `ResetExperienceBufferBeforeTraining` has changed from `true` to `false`.

When creating a new SAC agent, if you want to clear the experience buffer before training, you must specify `ResetExperienceBufferBeforeTraining` as `true`. For example, before training, set the property using dot notation.

```
agent.AgentOptions.ResetExperienceBufferBeforeTraining = true;
```

Alternatively, you can set the property to `true` in an `rlSACAgentOptions` object and use this object to create the SAC agent.

References

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning. Cambridge, Mass: The MIT Press, 2018.

See Also

Objects

`rlSACAgent` | `rlAgentInitializationOptions` | `rlConservativeQLearningOptions` | `rlBehaviorCloningRegularizerOptions`

Topics

"Train SAC Agent for Ball Balance Control"
"Soft Actor-Critic (SAC) Agents"

rlSARSAAgent

SARSA reinforcement learning agent

Description

The SARSA algorithm is a model-free, online, on-policy reinforcement learning method. A SARSA agent is a value-based reinforcement learning agent which trains a critic to estimate the return or future rewards.

Note SARSA agents do not support recurrent networks.

For more information on SARSA agents, see “SARSA Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

`agent = rlSARSAAgent(critic,agentOptions)`

Description

`agent = rlSARSAAgent(critic,agentOptions)` creates a SARSA agent with the specified critic network and sets the `AgentOptions` property.

Input Arguments

`critic` — Critic
`rlQValueFunction` object

Critic, specified as an `rlQValueFunction` object. For more information on creating critics, see “Create Policies and Value Functions”.

Properties

`AgentOptions` — Agent options
`rlSARSAAgentOptions` object

Agent options, specified as an `rlSARSAAgentOptions` object.

`UseExplorationPolicy` — Option to use exploration policy
`false` (default) | `true`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- false — Use the agent greedy policy when selecting actions.
- true — Use the agent exploration policy when selecting actions.

ObservationInfo — Observation specifications

specification object | array of specification objects

This property is read-only.

Observation specifications, specified as an rlFiniteSetSpec or rlNumericSpec object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

The value of ObservationInfo matches the corresponding value specified in critic.

ActionInfo — Action specification

rlFiniteSetSpec object

This property is read-only.

Action specifications, specified as an rlFiniteSetSpec object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

The value of ActionInfo matches the corresponding value specified in critic.

SampleTime — Sample time of agent

positive scalar | -1

Sample time of agent, specified as a positive scalar or as -1. Setting this parameter to -1 allows for event-based simulations. The initial value of SampleTime matches the value specified in AgentOptions.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every SampleTime seconds of simulation time. If SampleTime is -1, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, SampleTime is the time interval between consecutive elements in the output experience returned by sim or train. If SampleTime is -1, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent
setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent

setCritic	Set critic of reinforcement learning agent
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create a SARSA Agent

Create or load an environment interface. For this example load the Basic Grid World environment interface also used in the example “Train Reinforcement Learning Agent in Basic Grid World”.

```
env = rlPredefinedEnv("BasicGridWorld");
```

Get observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

A SARSA agent uses a parametrized Q-value function to estimate the value of the policy. A Q-value function takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

Since both observation and action spaces are discrete and low-dimensional, use a table to model the Q-value function within the critic. `rlTable` creates a value table object from the observation and action specifications objects.

Create a table approximation model derived from the environment observation and action specifications.

```
qTable = rlTable(obsInfo,actInfo);
```

Create the Q-value function approximator object using `qTable` and the environment specification objects. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(qTable,obsInfo,actInfo);
```

Create a SARSA agent using the approximator object.

```
agent = rlSARSAAgent(critic)
```

```
agent = 
  rlSARSAAgent with properties:

            AgentOptions: [1x1 rl.option.rlSARSAAgentOptions]
    UseExplorationPolicy: 0
         ObservationInfo: [1x1 rl.util.rlFiniteSetSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Specify an `Epsilon` value of `0.05`.

```
opt = rlSARSAAgentOptions;
agent.AgentOptions.EpsilonGreedyExploration.Epsilon = 0.05;
```

To check your agent, use `getAction` to return the action from a random observation.

```
act = getAction(agent,{randi(numel(obsInfo.Elements))});
act{1}
```

```
ans = 1
```

You can now test and train the agent against the environment.

Version History

Introduced in R2019a

See Also

Apps

Reinforcement Learning Designer | Deep Network Designer

Functions

`getAction` | `getActor` | `getCritic` | `getModel` | `generatePolicyFunction` | `generatePolicyBlock` | `getActionInfo` | `getObservationInfo`

Objects

`rlSARSAAgentOptions` | `rlAgentInitializationOptions` | `rlQValueFunction` | `rlQAgent` | `rlDQNAgent`

Blocks

RL Agent | Policy

Topics

"Train Reinforcement Learning Agent in MDP Environment"
"Train Reinforcement Learning Agents"
"SARSA Agents"
"Reinforcement Learning Agents"

rlSARSAAgentOptions

Options for SARSA agent

Description

Use an `rlSARSAAgentOptions` object to specify options for creating SARSA agents. To create a SARSA agent, use `rlSARSAAgent`

For more information on SARSA agents, see “SARSA Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlSARSAAgentOptions
opt = rlSARSAAgentOptions(Name,Value)
```

Description

`opt = rlSARSAAgentOptions` creates an `rlSARSAAgentOptions` object for use as an argument when creating a SARSA agent using all default settings. You can modify the object properties using dot notation.

`opt = rlSARSAAgentOptions(Name,Value)` sets option properties on page 3-378 using name-value pairs. For example, `rlSARSAAgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`EpsilonGreedyExploration` — Options for epsilon-greedy exploration
`EpsilonGreedyExploration` object

Options for epsilon-greedy exploration, specified as an `EpsilonGreedyExploration` object with the following properties.

Property	Description	Default Value
Epsilon	Probability threshold to either randomly select an action or select the action that maximizes the state-action value function. A larger value of Epsilon means that the agent randomly explores the action space at a higher rate.	1
EpsilonMin	Minimum value of Epsilon	0.01
EpsilonDecay	Decay rate	0.0050

At the end of each training time step, if Epsilon is greater than EpsilonMin, then it is updated using the following formula.

```
Epsilon = Epsilon*(1-EpsilonDecay)
```

Note that Epsilon is conserved between the end of an episode and the start of the next one. Therefore, it keeps on uniformly decreasing over multiple episodes until it reaches EpsilonMin.

If your agent converges on local optima too quickly, you can promote agent exploration by increasing Epsilon.

To specify exploration options, use dot notation after creating the rlSARSAAgentOptions object opt. For example, set the epsilon value to 0.9.

```
opt.EpsilonGreedyExploration.Epsilon = 0.9;
```

CriticOptimizerOptions — Critic optimizer options

rlOptimizerOptions object

Critic optimizer options, specified as an rlOptimizerOptions object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see rlOptimizerOptions and rlOptimizer.

SampleTime — Sample time of agent

1 (default) | positive scalar | -1

Sample time of agent, specified as a positive scalar or as -1. Setting this parameter to -1 allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every SampleTime seconds of simulation time. If SampleTime is -1, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, SampleTime is the time interval between consecutive elements in the output experience returned by sim or train. If SampleTime is -1, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

DiscountFactor — Discount factor

0.99 (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlSARSAAgent	SARSA reinforcement learning agent

Examples

Create a SARSA Agent Options Object

Create an `rlSARSAAgentOptions` object that specifies the agent sample time.

```
opt = rlSARSAAgentOptions(SampleTime=0.5)
```

```
opt = 
  rlSARSAAgentOptions with properties:

    EpsilonGreedyExploration: [1x1 rl.option.EpsilonGreedyExploration]
      CriticOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
                  SampleTime: 0.5000
              DiscountFactor: 0.9900
                  InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent discount factor to `0.95`.

```
opt.DiscountFactor = 0.95;
```

Version History

Introduced in R2019a

See Also

Objects

`rlSARSAAgent` | `rlAgentInitializationOptions`

Topics

"Train Reinforcement Learning Agent in Basic Grid World"
"SARSA Agents"

rlSimulationOptions

Options for simulating a reinforcement learning agent within an environment

Description

Use an `rlSimulationOptions` object to specify simulation options for simulating a reinforcement learning agent within an environment. To perform the simulation, use `sim`.

For more information on agents training and simulation, see “Train Reinforcement Learning Agents”.

Creation

Syntax

```
simOpts = rlSimulationOptions
opt = rlSimulationOptions(Name,Value)
```

Description

`simOpts = rlSimulationOptions` returns the default options for simulating a reinforcement learning environment against an agent. Use simulation options to specify parameters about the simulation such as the maximum number of steps to run per simulation and the number of simulations to run. After configuring the options, use `simOpts` as an input argument for `sim`.

`opt = rlSimulationOptions(Name,Value)` creates a simulation options set with the specified “Properties” on page 3-381 using one or more name-value pair arguments.

Properties

`MaxSteps` — Number of steps to run the simulation
500 (default) | positive integer

Number of steps to run the simulation, specified as the comma-separated pair consisting of `'MaxSteps'` and a positive integer. In general, you define episode termination conditions in the environment. This value is the maximum number of steps to run in the simulation if those termination conditions are not met.

Example: `'MaxSteps',1000`

`NumSimulations` — Number of simulations
1 (default) | positive integer

Number of simulations to run, specified as the comma-separated pair consisting of `'NumSimulations'` and a positive integer. At the start of each simulation, `sim` resets the environment. You specify what happens on environment reset when you create the environment. For instance, resetting the environment at the start of each episode can include randomizing initial state values, if you configure your environment to do so. In that case, running multiple simulations allows you to validate performance of a trained agent over a range of initial conditions.

Example: 'NumSimulations',10

StopOnError — Stop simulation when error occurs

"on" (default) | "off"

Stop simulation when an error occurs, specified as "off" or "on". When this option is "off", errors are captured and returned in the SimulationInfo output of sim, and simulation continues.

UseParallel — Flag for using parallel simulation

false (default) | true

Flag for using parallel simulation, specified as a logical. Setting this option to true configures the simulation to use parallel processing to simulate the environment, thereby enabling usage of multiple cores, processors, computer clusters or cloud resources to speed up simulation. To specify options for parallel simulation, use the ParallelizationOptions property.

Note that if you want to speed up deep neural network calculations (such as gradient computation, parameter update and prediction) using a local GPU you do not need to set UseParallel to true. Instead, when creating your actor or critic representation, use an rlRepresentationOptions object in which the UseDevice option is set to "gpu".

Using parallel computing or the GPU requires Parallel Computing Toolbox software. Using computer clusters or cloud resources additionally requires MATLAB Parallel Server™.

For more information about training using multicore processors and GPUs, see “Train Agents Using Parallel Computing and GPUs”.

Example: 'UseParallel',true

ParallelizationOptions — Options to control parallel simulation

ParallelTraining object

Parallelization options to control parallel simulation, specified as a ParallelTraining object. For more information about training using parallel computing, see “Train Reinforcement Learning Agents”.

The ParallelTraining object has the following properties, which you can modify using dot notation after creating the rlTrainingOptions object.

WorkerRandomSeeds — Randomizer initialization for workers

–1 (default) | –2 | vector

Randomizer initialization for workers, specified as one the following:

- –1 — Assign a unique random seed to each worker. The value of the seed is the worker ID.
- –2 — Do not assign a random seed to the workers.
- Vector — Manually specify the random seed for each work. The number of elements in the vector must match the number of workers.

TransferBaseWorkspaceVariables — Send model and workspace variables to parallel workers

"on" (default) | "off"

Send model and workspace variables to parallel workers, specified as `"on"` or `"off"`. When the option is `"on"`, the host sends variables used in models and defined in the base MATLAB workspace to the workers.

`AttachedFiles` — Additional files to attach to the parallel pool
[ ] (default) | string | string array

Additional files to attach to the parallel pool, specified as a string or string array.

`SetupFcn` — Function to run before simulation starts
[ ] (default) | function handle

Function to run before simulation starts, specified as a handle to a function having no input arguments. This function is run once per worker before simulation begins. Write this function to perform any processing that you need prior to simulation.

`CleanupFcn` — Function to run after simulation ends
[ ] (default) | function handle

Function to run after simulation ends, specified as a handle to a function having no input arguments. You can write this function to clean up the workspace or perform other processing after simulation terminates.

Object Functions

sim	Simulate trained reinforcement learning agents within specified environment

Examples

Configure Options for Simulation

Create an options set for simulating a reinforcement learning environment. Set the number of steps to simulate to 1000, and configure the options to run three simulations.

You can set the options using Name,Value pairs when you create the options set. Any options that you do not explicitly set have their default values.

```
simOpts = rlSimulationOptions(...
    MaxSteps=1000,...
    NumSimulations=3)
```

```
simOpts = 
  rlSimulationOptions with properties:

                  MaxSteps: 1000
            NumSimulations: 3
               StopOnError: "on"
               UseParallel: 0
    ParallelizationOptions: [1x1 rl.option.ParallelSimulation]
```

Alternatively, create a default options set and use dot notation to change some of the values.

```
simOpts = rlSimulationOptions;
simOpts.MaxSteps = 1000;
simOpts.NumSimulations = 3;

simOpts
```

```
simOpts = 
  rlSimulationOptions with properties:

                  MaxSteps: 1000
            NumSimulations: 3
               StopOnError: "on"
               UseParallel: 0
    ParallelizationOptions: [1x1 rl.option.ParallelSimulation]
```

Version History

Introduced in R2019a

See Also

Functions

`sim`

Objects

`rlTrainingOptions` | `rlMultiAgentTrainingOptions`

Topics

“Train Reinforcement Learning Agents”
“Reinforcement Learning Agents”

rlStochasticActorPolicy

Policy object to generate stochastic actions for custom training loops and application deployment

Description

This object implements a stochastic policy, which returns stochastic actions given an input observation, according to a probability distribution. You can create an `rlStochasticActorPolicy` object from an `rlDiscreteCategoricalActor` or `rlContinuousGaussianActor`, or extract it from an `rlPGAgent`, `rlACAgent`, `rlPPOAgent`, `rlTRPOAgent`, or `rlSACAgent`. You can then train the policy object using a custom training loop or deploy it for your application using `generatePolicyBlock` or `generatePolicyFunction`. If `UseMaxLikelihoodAction` is set to `1` the policy is deterministic, therefore in this case it does not explore. For more information on policies and value functions, see “Create Policies and Value Functions”.

Creation

Syntax

```
policy = rlStochasticActorPolicy(actor)
```

Description

`policy = rlStochasticActorPolicy(actor)` creates the stochastic policy object `policy` from the continuous Gaussian or discrete categorical actor `actor`. It also sets the `Actor` property of `policy` to the input argument `actor`.

Properties

`Actor` — Actor
`rlDiscreteCategoricalActor` object | `rlContinuousGaussianActor` object

Actor, specified as an `rlContinuousGaussianActor` or `rlDiscreteCategoricalActor` object.

`UseMaxLikelihoodAction` — Option to enable maximum likelihood action
`false` (default) | `true`

Option to enable maximum likelihood action, specified as a logical value: either `false` (default, the action is sampled from the probability distribution, this helps exploration) or `true` (always using maximum likelihood action). When the option to always use the maximum likelihood action enabled the policy is deterministic and therefore it does not explore.

Example: `false`

`ObservationInfo` — Observation specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array of such objects. These objects define properties such as the dimensions, data types, and names of the observation channels.

`ActionInfo` — Action specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object

Action specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

`SampleTime` — Sample time of policy

positive scalar | `-1` (default)

Sample time of the policy, specified as a positive scalar or as `-1` (default). Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the policy is specified executes every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the policy is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience. If `SampleTime` is `-1`, the sample time is treated as being equal to `1`.

Example: `0.2`

Object Functions

generatePolicyBlock	Generate Simulink block that evaluates policy of an agent or policy object
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object
getAction	Obtain action from agent, actor, or policy object given environment observations
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
reset	Reset environment, agent, experience buffer, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object

Examples

Create Stochastic Actor Policy from Discrete Categorical Actor

Create observation and action specification objects. For this example, define a continuous four-dimensional observation space and a discrete action space having two possible actions.

```
obsInfo = rlNumericSpec([4 1]);
actInfo = rlFiniteSetSpec([-1 1]);
```

Alternatively, use `getObservationInfo` and `getActionInfo` to extract the specification objects from an environment

Create a discrete categorical actor. This actor must accept an observation as input and return an output vector in which each element represents the probability of taking the corresponding action.

To approximate the policy function within the actor, use a deep neural network model. Define the network as an array of layer objects, and get the dimension of the observation space and the number of possible actions from the environment specification objects.

```
layers = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert the network to a `dlnetwork` object and display the number of weights.

```
model = dlnetwork(layers);
summary(model)
```

```
   Initialized: true

   Number of learnables: 114

   Inputs:
      1   'input'   4 features
```

Create the actor using `model`, and the observation and action specifications.

```
actor = rlDiscreteCategoricalActor(model,obsInfo,actInfo)
```

```
actor = 
  rlDiscreteCategoricalActor with properties:

       Distribution: [1x1 rl.distribution.rlDiscreteGaussianDistribution]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
          UseDevice: "cpu"
```

To return the probability distribution of the possible actions as a function of a random observation, and given the current network weights, use `evaluate`.

```
prb = evaluate(actor,{rand(obsInfo.Dimension)});
prb{1}
```

```
ans = 2x1 single column vector

    0.5850
    0.4150
```

Create a policy object from `actor`.

```
policy = rlStochasticActorPolicy(actor)
```

```
policy = 
  rlStochasticActorPolicy with properties:
```

```
                    Actor: [1x1 rl.function.rlDiscreteCategoricalActor]
    UseMaxLikelihoodAction: 0
           ObservationInfo: [1x1 rl.util.rlNumericSpec]
                ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
                SampleTime: -1
```

You can access the policy options using dot notation. For example, set the option to always use the maximum likelihood action, thereby making the policy deterministic.

```
policy.UseMaxLikelihoodAction = true
```

```
policy = 
  rlStochasticActorPolicy with properties:

                    Actor: [1x1 rl.function.rlDiscreteCategoricalActor]
    UseMaxLikelihoodAction: 1
           ObservationInfo: [1x1 rl.util.rlNumericSpec]
                ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
                SampleTime: -1
```

Check the policy with a random observation input.

```
act = getAction(policy,{rand(obsInfo.Dimension)});
act{1}
```

```
ans = -1
```

You can now train the policy with a custom training loop and then deploy it to your application.

Version History

Introduced in R2022a

See Also

Functions

getGreedyPolicy | getExplorationPolicy | generatePolicyBlock | generatePolicyFunction | getAction | getLearnableParameters | setLearnableParameters

Objects

rlMaxQPolicy | rlEpsilonGreedyPolicy | rlDeterministicActorPolicy | rlAdditiveNoisePolicy | rlDiscreteCategoricalActor | rlContinuousGaussianActor | rlPGAgent | rlACAgent | rlSACAgent | rlPPOAgent | rlTRPOAgent

Blocks

Policy | RL Agent

Topics

“Model-Based Reinforcement Learning Using Custom Training Loop”
“Train Reinforcement Learning Policy Using Custom Training Loop”

"Create Policies and Value Functions"

rlStochasticActorRepresentation

(Not recommended) Stochastic actor representation for reinforcement learning agents

Note `rlStochasticActorRepresentation` is not recommended. Use either `rlDiscreteCategoricalActor` or `rlContinuousGaussianActor` instead. For more information, see “rlStochasticActorRepresentation is not recommended”.

Description

This object implements a function approximator to be used as a stochastic actor within a reinforcement learning agent. A stochastic actor takes the observations as inputs and returns a random action, thereby implementing a stochastic policy with a specific probability distribution. After you create an `rlStochasticActorRepresentation` object, use it to create a suitable agent, such as an `rlACAgent` or `rlPGAgent` agent. For more information on creating representations, see “Create Policies and Value Functions”.

Creation

Syntax

```
discActor = rlStochasticActorRepresentation(net,observationInfo,
discActionInfo,'Observation',obsName)
discActor = rlStochasticActorRepresentation({basisFcn,W0},observationInfo,
actionInfo)
discActor = rlStochasticActorRepresentation( ___ ,options)
```

```
contActor = rlStochasticActorRepresentation(net,observationInfo,
contActionInfo,'Observation',obsName)
contActor = rlStochasticActorRepresentation( ___ ,options)
```

Description

Discrete Action Space Stochastic Actor

`discActor = rlStochasticActorRepresentation(net,observationInfo, discActionInfo,'Observation',obsName)` creates a stochastic actor with a discrete action space, using the deep neural network `net` as function approximator. Here, the output layer of `net` must have as many elements as the number of possible discrete actions. This syntax sets the ObservationInfo and ActionInfo properties of `discActor` to the inputs `observationInfo` and `discActionInfo`, respectively. `obsName` must contain the names of the input layers of `net`.

`discActor = rlStochasticActorRepresentation({basisFcn,W0},observationInfo, actionInfo)` creates a discrete space stochastic actor using a custom basis function as underlying approximator. The first input argument is a two-elements cell in which the first element contains the handle `basisFcn` to a custom basis function, and the second element contains the initial weight matrix `W0`. This syntax sets the ObservationInfo and ActionInfo properties of `discActor` to the inputs `observationInfo` and `actionInfo`, respectively.

`discActor = rlStochasticActorRepresentation( ___ ,options)` creates the discrete action space, stochastic actor `discActor` using the additional options set `options`, which is an `rlRepresentationOptions` object. This syntax sets the Options property of `discActor` to the `options` input argument. You can use this syntax with any of the previous input-argument combinations.

Continuous Action Space Gaussian Actor

`contActor = rlStochasticActorRepresentation(net,observationInfo, contActionInfo,'Observation',obsName)` creates a Gaussian stochastic actor with a continuous action space using the deep neural network `net` as function approximator. Here, the output layer of `net` must have twice as many elements as the number of dimensions of the continuous action space. This syntax sets the ObservationInfo and ActionInfo properties of `contActor` to the inputs `observationInfo` and `contActionInfo` respectively. `obsName` must contain the names of the input layers of `net`.

Note `contActor` does not enforce constraints set by the action specification, therefore, when using this actor, you must enforce action space constraints within the environment.

`contActor = rlStochasticActorRepresentation( ___ ,options)` creates the continuous action space, Gaussian actor `contActor` using the additional `options` option set, which is an `rlRepresentationOptions` object. This syntax sets the Options property of `contActor` to the `options` input argument. You can use this syntax with any of the previous input-argument combinations.

Input Arguments

`net` — Deep neural network

array of `Layer` objects | `layerGraph` object | `DAGNetwork` object | `SeriesNetwork` object | `dlNetwork` object

Deep neural network used as the underlying approximator within the actor, specified as one of the following:

- Array of `Layer` objects
- `layerGraph` object
- `DAGNetwork` object
- `SeriesNetwork` object
- `dlnetwork` object

For a discrete action space stochastic actor, `net` must have the observations as input and a single output layer having as many elements as the number of possible discrete actions. Each element represents the probability (which must be nonnegative) of executing the corresponding action.

For a continuous action space stochastic actor, `net` must have the observations as input and a single output layer having twice as many elements as the number of dimensions of the continuous action space. The elements of the output vector represent all the mean values followed by all the standard deviations (which must be nonnegative) of the Gaussian distributions for the dimensions of the action space.

Note The fact that standard deviations must be nonnegative while mean values must fall within the output range means that the network must have two separate paths. The first path must produce an estimation for the mean values, so any output nonlinearity must be scaled so that its output falls in the desired range. The second path must produce an estimation for the standard deviations, so you must use a softplus or ReLU layer to enforce nonnegativity.

The network input layers must be in the same order and with the same data type and dimensions as the signals defined in ObservationInfo. Also, the names of these input layers must match the observation names specified in `obsName`. The network output layer must have the same data type and dimension as the signal defined in ActionInfo.

`rlStochasticActorRepresentation` objects support recurrent deep neural networks.

For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

`obsName` — Observation names
string | character vector | cell array of character vectors

Observation names, specified as a cell array of strings or character vectors. The observation names must be the names of the input layers in `net`.

Example: `{'my_obs'}`

`basisFcn` — Custom basis function
function handle

Custom basis function, specified as a function handle to a user-defined MATLAB function. The user defined function can either be an anonymous function or a function on the MATLAB path. The output of the actor is the vector `a = softmax(W'*B)`, where `W` is a weight matrix and `B` is the column vector returned by the custom basis function. Each element of `a` represents the probability of taking the corresponding action. The learnable parameters of the actor are the elements of `W`.

When creating a stochastic actor representation, your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here `obs1` to `obsN` are observations in the same order and with the same data type and dimensions as the signals defined in `observationInfo`

Example: `@(obs1,obs2,obs3) [obs3(2)*obs1(1)^2; abs(obs2(5)+obs3(1))]`

`W0` — Initial value of the basis function weights
column vector

Initial value of the basis function weights, `W`, specified as a matrix. It must have as many rows as the length of the basis function output, and as many columns as the number of possible actions.

Properties

`Options` — Representation options
`rlRepresentationOptions` object

Representation options, specified as an `rlRepresentationOptions` object. Available options include the optimizer used for training and the learning rate.

Note TRPO agents use only the `Options.UseDevice` representation options and ignore the other training and learning rate options.

`ObservationInfo` — Observation specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. These objects define properties such as dimensions, data type, and names of the observation signals.

`rlStochasticActorRepresentation` sets the ObservationInfo property of `contActor` or `discActor` to the input `observationInfo`.

You can extract `observationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`ActionInfo` — Action specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object

Action specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object. These objects define properties such as the dimensions, data type and name of the action signals.

For a discrete action space actor, `rlStochasticActorRepresentation` sets `ActionInfo` to the input `discActionInfo`, which must be an `rlFiniteSetSpec` object.

For a continuous action space actor, `rlStochasticActorRepresentation` sets `ActionInfo` to the input `contActionInfo`, which must be an `rlNumericSpec` object.

You can extract `ActionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually.

For custom basis function representations, the action signal must be a scalar, a column vector, or a discrete action.

Object Functions

rlACAgent	Actor-critic (AC) reinforcement learning agent
rlPGAgent	Policy gradient (PG) reinforcement learning agent
rlPPOAgent	Proximal policy optimization (PPO) reinforcement learning agent
rlSACAgent	Soft actor-critic (SAC) reinforcement learning agent
getAction	Obtain action from agent, actor, or policy object given environment observations

Examples

Create Discrete Stochastic Actor from Deep Neural Network

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create an action specification object (or alternatively use getActionInfo to extract the specification object from an environment). For this example, define the action space as consisting of three values, -10, 0, and 10.

```
actInfo = rlFiniteSetSpec([-10 0 10]);
```

Create a deep neural network approximator for the actor. The input of the network (here called state) must accept a four-element vector (the observation vector just defined by obsInfo), and its output (here called actionProb) must be a three-element vector. Each element of the output vector must be between 0 and 1 since it represents the probability of executing each of the three possible actions (as defined by actInfo). Using softmax as the output layer enforces this requirement.

```
net = [  featureInputLayer(4,'Normalization','none',...
             'Name','state')
         fullyConnectedLayer(3,'Name','fc')
         softmaxLayer('Name','actionProb')  ];
```

Create the actor with rlStochasticActorRepresentation, using the network, the observations and action specification objects, as well as the names of the network input layer.

```
actor = rlStochasticActorRepresentation(net,obsInfo,actInfo,...
           'Observation','state')
```

```
actor = 
  rlStochasticActorRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To validate your actor, use getAction to return a random action from the observation vector [1 1 1 1], using the current network weights.

```
act = getAction(actor,{[1 1 1 1]}); 
act{1}
```

```
ans = 10
```

You can now use the actor to create a suitable agent, such as an rlACAgent, or rlPGAgent agent.

Create Continuous Stochastic Actor from Deep Neural Network

Create an observation specification object (or alternatively use getObservationInfo to extract the specification object from an environment). For this example, define the observation space as a continuous six-dimensional space, so that a single observation is a column vector containing 6 doubles.

```
obsInfo = rlNumericSpec([6 1]);
```

Create an action specification object (or alternatively use getActionInfo to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that a single action is a column vector containing 2 doubles both between -10 and 10.

```
actInfo = rlNumericSpec([2 1],'LowerLimit',-10,'UpperLimit',10);
```

Create a deep neural network approximator for the actor. The observation input (here called `myobs`) must accept a six-dimensional vector (the observation vector just defined by `obsInfo`). The output (here called `myact`) must be a four-dimensional vector (twice the number of dimensions defined by `actInfo`). The elements of the output vector represent, in sequence, all the means and all the standard deviations of every action.

The fact that standard deviations must be non-negative while mean values must fall within the output range means that the network must have two separate paths. The first path is for the mean values, and any output nonlinearity must be scaled so that it can produce values in the output range. The second path is for the standard deviations, and you can use a softplus or ReLU layer to enforce non-negativity.

```
% input path layers (6 by 1 input and a 2 by 1 output)
inPath = [ imageInputLayer([6 1 1], ...
               'Normalization','none',...
               'Name','myobs')
           fullyConnectedLayer(2,'Name','infc') ];

% path layers for mean value
% (2 by 1 input and 2 by 1 output)
% using scalingLayer to scale the range
meanPath = [ tanhLayer('Name','tanh'); % range: (-1,1)
             scalingLayer('Name','scale',...
             'Scale',actInfo.UpperLimit) ]; % range: (-10,10)

% path layers for standard deviations
% (2 by 1 input and output)
% using softplus layer to make it non negative
sdevPath =  softplusLayer('Name', 'splus');

% conctatenate two inputs (along dimension #3)
% to form a single (4 by 1) output layer
outLayer = concatenationLayer(3,2,'Name','mean&sdev');

% add layers to network object
net = layerGraph(inPath);
net = addLayers(net,meanPath);
net = addLayers(net,sdevPath);
net = addLayers(net,outLayer);

% connect layers: the mean value path output MUST
% be connected to the FIRST input of the concatenationLayer

% connect output of inPath to meanPath input
net = connectLayers(net,'infc','tanh/in');
% connect output of inPath to sdevPath input
net = connectLayers(net,'infc','splus/in');
% connect output of meanPath to gaussPars input #1
net = connectLayers(net,'scale','mean&sdev/in1');
% connect output of sdevPath to gaussPars input #2
net = connectLayers(net,'splus','mean&sdev/in2');
```

Set some training options for the actor.

```
actorOpts = rlRepresentationOptions('LearnRate',8e-3,'GradientThreshold',1);
```

Create the actor with `rlStochasticActorRepresentation`, using the network, the observations and action specification objects, the names of the network input layer and the options object.

```
actor = rlStochasticActorRepresentation(net, obsInfo, actInfo, 'Observation','myobs',actorOpts)
```

```
actor = 
  rlStochasticActorRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlNumericSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your actor, use `getAction` to return a random action from the observation vector `ones(6,1)`, using the current network weights.

```
act = getAction(actor,{ones(6,1)});
act{1}
```

```
ans = 2x1 single column vector

   -0.0763
    9.6860
```

You can now use the actor to create a suitable agent (such as an `rlACAgent`, `rlPGAgent`, or `rlPPOAgent` agent).

Create Stochastic Actor from Custom Basis Function

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing 2 doubles.

```
obsInfo = rlNumericSpec([2 1]);
```

The stochastic actor based on a custom basis function does not support continuous action spaces. Therefore, create a *discrete action space* specification object (or alternatively use `getActionInfo` to extract the specification object from an environment with a discrete action space). For this example, define the action space as a finite set consisting of 3 possible values (named `7`, `5`, and `3` in this case).

```
actInfo = rlFiniteSetSpec([7 5 3]);
```

Create a custom basis function. Each element is a function of the observations defined by `obsInfo`.

```
myBasisFcn = @(myobs) [myobs(2)^2;
                       myobs(1);
                       exp(myobs(2));
                       abs(myobs(1))]
```

```
myBasisFcn = function_handle with value:
    @(myobs)[myobs(2)^2;myobs(1);exp(myobs(2));abs(myobs(1))]
```

The output of the actor is the action, among the ones defined in `actInfo`, corresponding to the element of `softmax(W'*myBasisFcn(myobs))` which has the highest value. `W` is a weight matrix, containing the learnable parameters, which must have as many rows as the length of the basis function output, and as many columns as the number of possible actions.

Define an initial parameter matrix.

```
W0 = rand(4,3);
```

Create the actor. The first argument is a two-element cell containing both the handle to the custom function and the initial parameter matrix. The second and third arguments are, respectively, the observation and action specification objects.

```
actor = rlStochasticActorRepresentation({myBasisFcn,W0},obsInfo,actInfo)
```

```
actor = 
  rlStochasticActorRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your actor use the `getAction` function to return one of the three possible actions, depending on a given random observation and on the current parameter matrix.

```
v = getAction(actor,{rand(2,1)})
```

```
v = 1x1 cell array
    {[3]}
```

You can now use the actor (along with an critic) to create a suitable discrete action space agent.

Create Stochastic Actor with Recurrent Neural Network

For this example, you create a stochastic actor with a discrete action space using a recurrent neural network. You can also use a recurrent neural network for a continuous stochastic actor using the same method.

Create an environment and obtain observation and action information.

```
env = rlPredefinedEnv('CartPole-Discrete');
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
numObs = obsInfo.Dimension(1);
numDiscreteAct = numel(actInfo.Elements);
```

Create a recurrent deep neural network for the actor. To create a recurrent neural network, use a `sequenceInputLayer` as the input layer and include at least one `lstmLayer`.

```
actorNetwork = [
    sequenceInputLayer(numObs,'Normalization','none','Name','state')
    fullyConnectedLayer(8,'Name','fc')
    reluLayer('Name','relu')
    lstmLayer(8,'OutputMode','sequence','Name','lstm')
    fullyConnectedLayer(numDiscreteAct,'Name','output')
    softmaxLayer('Name','actionProb')];
```

Create a stochastic actor representation for the network.

```
actorOptions = rlRepresentationOptions('LearnRate',1e-3,...
                   'GradientThreshold',1);
actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,actInfo,...
                   'Observation','state', actorOptions);
```

Version History

Introduced in R2020a

R2022a: `rlStochasticActorRepresentation` is not recommended

Not recommended starting in R2022a

`rlStochasticActorRepresentation` is not recommended. Use either `rlDiscreteCategoricalActor` or `rlContinuousGaussianActor` instead.

The following table shows some typical uses of `rlStochasticActorRepresentation` to create *neural network*-based actors, and how to update your code with one of the new stochastic actor approximator objects instead. The first table entry builds an actor with a discrete action space, the second one builds an actor with a continuous action space.

Network-Based Stochastic Actor Representation: Not Recommended	Network-Based Stochastic Actor Approximator: Recommended
`myActor = rlStochasticActorRepresentation(net,obsInfo,actInfo,'Observation',obsNames)`, with `actInfo` defining a discrete action space and `net` having observations as inputs and a single output layer with as many elements as the number of possible discrete actions.	`myActor = rlDiscreteCategoricalActor(net,obsInfo,actInfo,'ObservationInputNames',obsNames)`. Use this syntax to create a stochastic actor object with a discrete action space. This actor samples its action from a categorical (also known as Multinoulli) distribution.
`myActor = rlStochasticActorRepresentation(net,obsInfo,actInfo,'Observation',obsNames)`, with `actInfo` defining a continuous action space and `net` having observations as inputs and a single output layer with twice as many elements as the number of dimensions of the continuous action space (representing, in sequence, all the means and all the standard deviations of every action dimension).	`myActor = rlContinuousGaussianActor(net,obsInfo,actInfo,'ObservationInputNames',obsNames,'ActionMeanOutputNames',actMeanNames,'ActionStandardDeviationOutputNames',actStdNames)`. Use this syntax to create a stochastic actor object with a continuous action space. This actor samples its action from a Gaussian distribution, and you must provide the names of the network outputs representing the mean and standard deviations for the action.

The following table shows a typical use of `rlStochasticActorRepresentation` to create a (discrete action space) actor which use a (linear in the learnable parameters) *custom basis function*, and how to update your code with `rlDiscreteCategoricalActor` instead. In these function calls, the first input argument is a two-element cell array containing both the handle to the custom basis function and the initial weight vector or matrix.

Custom Basis Stochastic Actor Representation: Not Recommended	Custom Basis Function-Based Stochastic Actor Approximator: Recommended
`rep = rlStochasticActorRepresentation({basisFcn,W0},obsInfo,actInfo)`, where the basis function has observations as inputs and actions as outputs, `W0` is a matrix with as many columns as the number of possible actions, and `actInfo` defines a discrete action space.	`rep = rlDiscreteCategoricalActor({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a stochastic actor object with a discrete action space which returns an action sampled from a categorical (also known as Multinoulli) distribution.

See Also

Functions

`getActionInfo` | `getObservationInfo`

Objects

`rlDiscreteCategoricalActor` | `rlContinuousGaussianActor` | `rlRepresentationOptions`

Topics

"Create Policies and Value Functions"
"Reinforcement Learning Agents"

rlTable

Value table or Q table

Description

Value tables and Q-tables are one of the approximation models that can be used within value functions and Q-value functions, respectively. Value tables store rewards for a finite set of observations. Q tables store rewards for corresponding finite observation-action pairs.

To create a value function approximator using an `rlTable` object, use an `rlValueFunction`, `rlQValueFunction`, or `rlVectorQValueFunction` object.

Creation

Syntax

```
T = rlTable(obsinfo)
T = rlTable(obsinfo,actinfo)
```

Description

`T = rlTable(obsinfo)` creates a value table for the given discrete observations.

`T = rlTable(obsinfo,actinfo)` creates a Q table for the given discrete observations and actions.

Input Arguments

`obsinfo` — Observation specification
`rlFiniteSetSpec` object

Observation specification, specified as an `rlFiniteSetSpec` object.

`actinfo` — Action specification
`rlFiniteSetSpec` object

Action specification, specified as an `rlFiniteSetSpec` object.

Properties

`Table` — Reward table
array

Reward table, returned as an array. When `Table` is a:

- Value table, it contains N_O rows, where N_O is the number of finite observation values.
- Q table, it contains N_O rows and N_A columns, where N_A is the number of possible finite actions.

Object Functions

rlValueFunction	Value function approximator object for reinforcement learning agents
rlQValueFunction	Q-Value function approximator object for reinforcement learning agents
rlVectorQValueFunction	Vector Q-value function approximator for reinforcement learning agents

Examples

Create a Value Table

Create an environment interface, and obtain its observation specifications.

```
env = rlPredefinedEnv("BasicGridWorld");
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlFiniteSetSpec with properties:

       Elements: [25x1 double]
           Name: "MDP Observations"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

Create the value table using the observation specification.

```
vTable = rlTable(obsInfo)
```

```
vTable = 
  rlTable with properties:

    Table: [25x1 double]
```

You can now use your table as an approximation model for a value function for an agent with a discrete observation space. For more information, see `rlValueFunction`.

Create a Q Table

This example shows how to use `rlTable` to create a Q table. Such a table could be used to represent the actor or critic of an agent with finite observation and action spaces.

Create an environment interface, and obtain its observation and action specifications.

```
env=rlMDPEnv(createMDP(8,["up";"down"]));
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlFiniteSetSpec with properties:

       Elements: [8x1 double]
           Name: "MDP Observations"
    Description: [0x0 string]
```

```
      Dimension: [1 1]
       DataType: "double"
```

```
actInfo = getActionInfo(env)
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [2x1 double]
           Name: "MDP Actions"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

Create the Q table using the observation and action specifications.

```
qTable = rlTable(obsInfo,actInfo)
```

```
qTable = 
  rlTable with properties:

    Table: [8x2 double]
```

You can now use your table as an approximation model for a Q-value function for an agent with discrete action and observation spaces. For more information, see `rlQValueFunction`.

Version History

Introduced in R2019a

See Also

Functions
`getObservationInfo` | `getActionInfo`

Objects
`rlFiniteSetSpec` | `rlValueFunction` | `rlQValueFunction` | `rlQAgent` | `rlSARSAAgent` | `rlDQNAgent`

Topics
"Train Reinforcement Learning Agent in Basic Grid World"
"Train Reinforcement Learning Agent in MDP Environment"
"Load Predefined Control System Environments"
"Reinforcement Learning Agents"
"Create Policies and Value Functions"

rlTD3Agent

Twin-delayed deep deterministic (TD3) policy gradient reinforcement learning agent

Description

The twin-delayed deep deterministic (TD3) policy gradient algorithm is an actor-critic, model-free, online, off-policy reinforcement learning method which computes an optimal policy that maximizes the long-term reward. The action space can only be continuous.

Use `rlTD3Agent` to create one of the following types of agents.

- Twin-delayed deep deterministic policy gradient (TD3) agent with two Q-value functions. This agent prevents overestimation of the value function by learning two Q value functions and using the minimum values for policy updates.
- Delayed deep deterministic policy gradient (delayed DDPG) agent with a single Q value function. This agent is a DDPG agent with target policy smoothing and delayed policy and target updates.

For more information, see “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents”. For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlTD3Agent(observationInfo,actionInfo)
agent = rlTD3Agent(observationInfo,actionInfo,initOpts)

agent = rlTD3Agent(actor,critics,agentOptions)

agent = rlTD3Agent( ___ ,agentOptions)
```

Description

Create Agent from Observation and Action Specifications

`agent = rlTD3Agent(observationInfo,actionInfo)` creates a TD3 agent for an environment with the given observation and action specifications, using default initialization options. The actor and critics in the agent use default deep neural networks built from the observation specification `observationInfo` and the action specification `actionInfo`. The `ObservationInfo` and `ActionInfo` properties of `agent` are set to the `observationInfo` and `actionInfo` input arguments, respectively.

`agent = rlTD3Agent(observationInfo,actionInfo,initOpts)` creates a deep deterministic policy gradient agent for an environment with the given observation and action specifications. The agent uses default networks configured using options specified in the `initOpts` object. For more information on the initialization options, see `rlAgentInitializationOptions`.

Create Agent from Actor and Critic

`agent = rlTD3Agent(actor,critics,agentOptions)` creates an agent with the specified actor and critic. To create a:

- TD3 agent, specify a two-element row vector of critic.
- Delayed DDPG agent, specify a single critic.

Specify Agent Options

`agent = rlTD3Agent( ___ ,agentOptions)` creates a TD3 agent and sets the `AgentOptions` property to the `agentOptions` input argument. Use this syntax after any of the input arguments in the previous syntaxes.

Input Arguments

`initOpts` — Agent initialization options
`rlAgentInitializationOptions` object

Agent initialization options, specified as an `rlAgentInitializationOptions` object.

`actor` — Actor
`rlContinuousDeterministicActor` object

Actor, specified as an `rlContinuousDeterministicActor`. For more information on creating actors, see “Create Policies and Value Functions”.

`critics` — Critic network
`rlQValueFunction` object | two-element row vector of `rlQValueFunction` objects

Critic, specified as one of the following:

- `rlQValueFunction` object — Create a delayed DDPG agent with a single Q value function. This agent is a DDPG agent with target policy smoothing and delayed policy and target updates.
- Two-element row vector of `rlQValueFunction` objects — Create a TD3 agent with two critic value functions. The two critic networks must be unique `rlQValueFunction` objects with the same observation and action specifications. The critics can either have different structures or the same structure but with different initial parameters.

For more information on creating critics, see “Create Policies and Value Functions”.

Properties

`ObservationInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

If you create the agent by specifying an actor and critic, the value of `ObservationInfo` matches the value specified in the actor and critic objects.

You can extract `observationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`ActionInfo` — Action specification

`rlNumericSpec` object

Action specifications, specified as an `rlNumericSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

If you create the agent by specifying an actor and critic, the value of `ActionInfo` matches the value specified in the actor and critic objects.

You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlNumericSpec`.

`AgentOptions` — Agent options

`rlTD3AgentOptions` object

Agent options, specified as an `rlTD3AgentOptions` object.

If you create a TD3 agent with default actor and critic that use recurrent neural networks, the default value of `AgentOptions.SequenceLength` is `32`.

`UseExplorationPolicy` — Option to use exploration policy

`false` (default) | `true`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- `true` — Use the base agent exploration policy when selecting actions.
- `false` — Use the base agent greedy policy when selecting actions.

`ExperienceBuffer` — Experience buffer

`rlReplayMemory` object

Experience buffer, specified as an `rlReplayMemory` object. During training the agent stores each of its experiences (S,A,R,S',D) in a buffer. Here:

- S is the current observation of the environment.
- A is the action taken by the agent.
- R is the reward for taking action A.
- S' is the next observation after taking action A.
- D is the is-done signal after taking action A.

`SampleTime` — Sample time of agent

positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations. The value of `SampleTime` matches the value specified in `AgentOptions`.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent
setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent
setCritic	Set critic of reinforcement learning agent
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create TD3 Agent from Observation and Action Specifications

Create an environment with a continuous action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force, applied to the mass, ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");

obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a TD3 agent from the environment observation and action specifications.

```
agent = rlTD3Agent(obsInfo,actInfo);
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.0087]}
```

You can now test and train the agent within the environment. You can also use `getActor` and `getCritic` to extract the actor and critic, respectively, and `getModel` to extract the approximator model (by default a deep neural network) from the actor or critic.

Create TD3 Agent Using Initialization Options

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar representing a torque ranging continuously from -2 to 2 Nm.

```
% Load predefined environment
env = rlPredefinedEnv("SimplePendulumWithImage-Continuous");

% Obtain observation and action specifications
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create an agent initialization option object, specifying that each hidden fully connected layer in the network must have 128 neurons (instead of the default number, 256).

```
initOpts = rlAgentInitializationOptions(NumHiddenUnit=128);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a DDPG agent from the environment observation and action specifications.

```
agent = rlTD3Agent(obsInfo,actInfo,initOpts);
```

Extract the deep neural networks from the actor.

```
actorNet = getModel(getActor(agent));
```

Extract the deep neural networks from the two critics. Note that `getModel(critics)` only returns the first critic network.

```
critics = getCritic(agent);
criticNet1 = getModel(critics(1));
criticNet2 = getModel(critics(2));
```

To verify that each hidden fully connected layer has 128 neurons, you can display the layers on the MATLAB® command window,

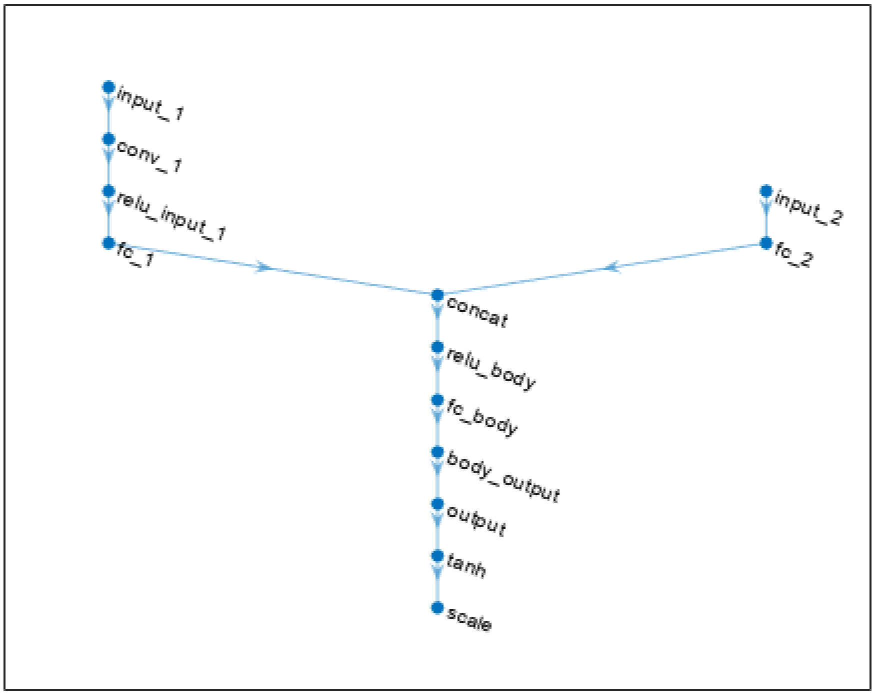
```
criticNet1.Layers
```

or visualize the structure interactively using `analyzeNetwork`.

```
analyzeNetwork(criticNet1)
```

Plot the networks of the actor and of the second critic, and display the number of weights.

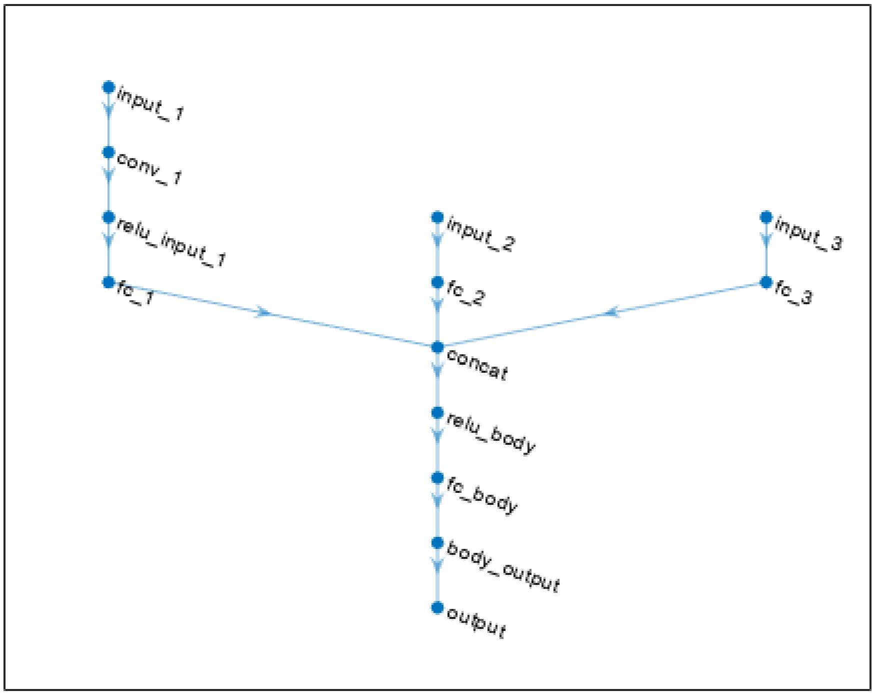
```
plot(layerGraph(actorNet))
```



```
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 18.9M

   Inputs:
      1   'input_1'   50x50x1 images
      2   'input_2'   1 features
```

```
plot(layerGraph(criticNet2))
```



```
summary(criticNet2)
```

```
   Initialized: true

   Number of learnables: 18.9M

   Inputs:
      1   'input_1'   50x50x1 images
      2   'input_2'   1 features
      3   'input_3'   1 features
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[0.0675]}
```

You can now test and train the agent within the environment.

Create TD3 Agent from Actor and Critic

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to

Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

TD3 agents use two Q-value function critics. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critics, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` return the number of dimensions of the observation and action spaces, respectively, regardless of whether they are arranged as row vectors, column vectors, or matrices.

Define each network path as an array of layer objects. Assign names to the input and output layers of each path, so you can connect them.

```
% Observation path
obsPath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="obsPathIn")
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(16,Name="obsPathOut")
    ];

% Action path
actPath = [
    featureInputLayer(prod(actInfo.Dimension),Name="actPathIn")
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(16,Name="actPathOut")
    ];

% Common path
commonPath = [
    concatenationLayer(1,2,Name="concat")
    reluLayer
    fullyConnectedLayer(1)
    ];

% Add layers to layergraph object
criticNet = layerGraph;
criticNet = addLayers(criticNet,obsPath);
criticNet = addLayers(criticNet,actPath);
criticNet = addLayers(criticNet,commonPath);

% Connect layers
criticNet = connectLayers(criticNet,"obsPathOut","concat/in1");
criticNet = connectLayers(criticNet,"actPathOut","concat/in2");
```

To initialize the network weights differently for the two critics, create two different `dlnetwork` objects. You must do this because if the agent constructor function does not accept two identical critics.

```
criticNet1 = dlnetwork(criticNet);
criticNet2 = dlnetwork(criticNet);
```

Display the number of weights.

```
summary(criticNet1)
```

```
   Initialized: true

   Number of learnables: 1.2k

   Inputs:
      1   'obsPathIn'   2 features
      2   'actPathIn'   1 features
```

Create the two critics using the two networks with different weights. Alternatively, if you use exactly the same network with the same weights, you must explicitly initialize the network each time (to make sure weights are initialized differently) before passing it to `rlQValueFunction`. To do so, use `initialize`.

```
critic1 = rlQValueFunction(criticNet1,obsInfo,actInfo);
critic2 = rlQValueFunction(criticNet2,obsInfo,actInfo);
```

For more information about Q-value function approximators, using `rlQValueFunction`.

Check the critics with a random observation and action input.

```
getValue(critic1,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.1330
```

```
getValue(critic2,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.1526
```

TD3 agents use a parametrized deterministic policy over continuous action spaces, which is implemented by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects.

```
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(400)
    reluLayer
    fullyConnectedLayer(300)
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension))
    tanhLayer
];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 121.8k

   Inputs:
      1   'input'   2 features
```

Create the actor using `actorNet`, and the environment specification objects.

```
actor  = rlContinuousDeterministicActor(actorNet,obsInfo,actInfo);
```

For more information about continuous deterministic actor approximators, see `rlContinuousDeterministicActor`.

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.0221]}
```

Specify training options for the critics.

```
criticOptions = rlOptimizerOptions( ...
    Optimizer="adam", ...
    LearnRate=1e-3,...
    GradientThreshold=1, ...
    L2RegularizationFactor=2e-4);
```

Specify training options for the actor.

```
actorOptions = rlOptimizerOptions( ...
    Optimizer="adam", ...
    LearnRate=1e-3,...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-5);
```

Specify agent options, including training options for actor and critics.

```
agentOptions = rlTD3AgentOptions;
agentOptions.DiscountFactor = 0.99;
agentOptions.TargetSmoothFactor = 5e-3;
agentOptions.TargetPolicySmoothModel.Variance = 0.2;
agentOptions.TargetPolicySmoothModel.LowerLimit = -0.5;
agentOptions.TargetPolicySmoothModel.UpperLimit = 0.5;
agentOptions.CriticOptimizerOptions = criticOptions;
agentOptions.ActorOptimizerOptions = actorOptions;
```

Create TD3 agent using actor, critics, and options.

```
agent = rlTD3Agent(actor,[critic1 critic2],agentOptions);
```

You can also create an rlTD3Agent object with a single critic. In this case, the object represents a DDPG agent with target policy smoothing and delayed policy and target updates.

```
delayedDDPGAgent = rlTD3Agent(actor,critic1,agentOptions);
```

To check your agents, use getAction to return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.0351]}
```

```
getAction(delayedDDPGAgent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.0296]}
```

You can now test and train either agent within the environment.

Create TD3 Agent with Recurrent Neural Networks

For this example, load the environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
```

Obtain observation and action specifications.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

TD3 agents use two Q-value function critics. To model the parametrized Q-value function within the critics, use a recurrent neural network with two input layers (one for the observation channel, as specified by obsInfo, and the other for the action channel, as specified by actInfo) and one output layer (which returns the scalar value).

Define each network path as an array of layer objects. To create a recurrent network, use a sequenceInputLayer as the input layer and include at least one lstmLayer.

```
% Define observation path
obsPath = [
    sequenceInputLayer(prod(obsInfo.Dimension),Name="obsIn")
    fullyConnectedLayer(400)
    reluLayer
    fullyConnectedLayer(300,Name="obsOut")
    ];

% Define action path
actPath = [
    sequenceInputLayer(prod(actInfo.Dimension),Name="actIn")
    fullyConnectedLayer(300,Name="actOut")
    ];

% Define common path
commonPath = [
    concatenationLayer(1,2,Name="cat")
    reluLayer
    lstmLayer(16);
    fullyConnectedLayer(1)
    ];

% Add layers to layerGraph object
criticNet = layerGraph;
criticNet = addLayers(criticNet,obsPath);
criticNet = addLayers(criticNet,actPath);
criticNet = addLayers(criticNet,commonPath);

% Connect paths
criticNet = connectLayers(criticNet,"obsOut","cat/in1");
criticNet = connectLayers(criticNet,"actOut","cat/in2");
```

To initialize the network weights differently for the two critics, create two different `dlnetwork` objects. You must do this because the agent constructor function does not accept two identical critics.

```
criticNet1 = dlnetwork(criticNet);
criticNet2 = dlnetwork(criticNet);
```

Display the number of weights.

```
summary(criticNet1)
```

```
   Initialized: true

   Number of learnables: 161.6k

   Inputs:
      1   'obsIn'   Sequence input with 2 dimensions
      2   'actIn'   Sequence input with 1 dimensions
```

Create the critic using the same network structure for both critics. The TD3 agent initializes the two networks using different default parameters.

```
critic1 = rlQValueFunction(criticNet1,obsInfo,actInfo);
critic2 = rlQValueFunction(criticNet2,obsInfo,actInfo);
```

Check the critics with a random observation and action input.

```
getValue(critic1,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.0060
```

```
getValue(critic2,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    0.0481
```

Since the critics has a recurrent network, the (continuous deterministic) actor must have a recurrent network as approximation model too. The network that models the parametrized policy, must take the observation and return the action.

Define the network as an array of layer objects.

```
actorNet = [
    sequenceInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(400)
    lstmLayer(8)
    reluLayer
    fullyConnectedLayer(300,"Name","ActorFC2")
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension))
    tanhLayer
    ];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 17.2k

   Inputs:
      1   'sequenceinput'   Sequence input with 2 dimensions
```

Create the actor using `actorNet`, and the environment specification objects.

```
actor  = rlContinuousDeterministicActor(actorNet,obsInfo,actInfo);
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.0014]}
```

Specify training options for the critics.

```
criticOptions = rlOptimizerOptions( ...
    Optimizer="adam", ...
    LearnRate=1e-3,...
    GradientThreshold=1, ...
    L2RegularizationFactor=2e-4);
```

Specify training options for the actor.

```
actorOptions = rlOptimizerOptions( ...
    Optimizer="adam", ...
    LearnRate=1e-3,...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-5);
```

Specify agent options, including training options for actor and critics. To use a TD3 agent with recurrent neural networks, you must specify a `SequenceLength` greater than 1.

```
agentOptions = rlTD3AgentOptions;
agentOptions.DiscountFactor = 0.99;
agentOptions.SequenceLength = 32;
agentOptions.TargetSmoothFactor = 5e-3;
agentOptions.TargetPolicySmoothModel.Variance = 0.2;
agentOptions.TargetPolicySmoothModel.LowerLimit = -0.5;
agentOptions.TargetPolicySmoothModel.UpperLimit = 0.5;
agentOptions.CriticOptimizerOptions = criticOptions;
agentOptions.ActorOptimizerOptions = actorOptions;
```

Create TD3 agent using actor, critics, and options.

```
agent = rlTD3Agent(actor,[critic1 critic2],agentOptions);
```

You can also create an `rlTD3Agent` object with a single critic. In this case, the object represents a DDPG agent with target policy smoothing and delayed policy and target updates.

```
delayedDDPGAgent = rlTD3Agent(actor,critic1,agentOptions);
```

To check your agents, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.0018]}
```

```
getAction(delayedDDPGAgent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.0015]}
```

To evaluate the agent using sequential observations, use the sequence length (time) dimension. For example, obtain actions for a sequence of 9 observations.

```
[action,state] = getAction(agent, ...
    {rand([obsInfo.Dimension 1 9])});
```

Display the action corresponding to the seventh element of the observation.

```
action = action{1};
action(1,1,1,7)
```

```
ans = -0.0034
```

You can now test and train the agent within the environment.

Version History

Introduced in R2020a

See Also

Apps

Reinforcement Learning Designer | Deep Network Designer

Functions

`getAction` | `getActor` | `getCritic` | `getModel` | `generatePolicyFunction` | `generatePolicyBlock` | `getActionInfo` | `getObservationInfo`

Objects
`rlTD3AgentOptions` | `rlAgentInitializationOptions` | `rlQValueFunction` | `rlContinuousDeterministicActor` | `rlContinuousGaussianActor` | `rlDDPGAgent` | `rlSACAgent` | `rlPPOAgent`

Blocks
RL Agent | Policy

Topics
"Train TD3 Agent for PMSM Control"
"Train Biped Robot to Walk Using Reinforcement Learning Agents"
"Train Reinforcement Learning Agents"
"Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents"
"Reinforcement Learning Agents"

rlTD3AgentOptions

Options for TD3 agent

Description

Use an `rlTD3AgentOptions` object to specify options for twin-delayed deep deterministic policy gradient (TD3) agents. To create a TD3 agent, use `rlTD3Agent`.

For more information see “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlTD3AgentOptions
opt = rlTD3AgentOptions(Name,Value)
```

Description

`opt = rlTD3AgentOptions` creates an options object for use as an argument when creating a TD3 agent using all default options. You can modify the object properties using dot notation.

`opt = rlTD3AgentOptions(Name,Value)` sets option properties on page 3-418 using name-value pairs. For example, `rlTD3AgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`ExplorationModel` — Exploration noise model options
`GaussianActionNoise` object (default) | `OrnsteinUhlenbeckActionNoise` object

Noise model options, specified as a `GaussianActionNoise` object or an `OrnsteinUhlenbeckActionNoise` object. For more information on noise models, see “Noise Models” on page 3-422.

For an agent with multiple actions, if the actions have different ranges and units, it is likely that each action requires different noise model parameters. If the actions have similar ranges and units, you can set the noise parameters for all actions to the same value.

For example, for an agent with two actions, set the standard deviation of each action to a different value while using the same decay rate for both standard deviations.

```
opt = rlTD3AgentOptions;
opt.ExplorationModel.StandardDeviation = [0.1 0.2];
opt.ExplorationModel.StandardDeviationDecayRate = 1e-4;
```

To use Ornstein-Uhlenbeck action noise, first create a default `OrnsteinUhlenbeckActionNoise` object. Then, specify any nondefault model properties using dot notation.

```
opt = rlTD3AgentOptions;
opt.ExplorationModel = rl.option.OrnsteinUhlenbeckActionNoise;
opt.ExplorationModel.StandardDeviation = 0.05;
```

`TargetPolicySmoothModel` — Target smoothing noise model options
`GaussianActionNoise` object

Target smoothing noise model options, specified as a `GaussianActionNoise` object. This model helps the policy exploit actions with high Q-value estimates. For more information on noise models, see “Noise Models” on page 3-422.

For an agent with multiple actions, if the actions have different ranges and units, it is likely that each action requires different smoothing noise model parameters. If the actions have similar ranges and units, you can set the noise parameters for all actions to the same value.

For example, for an agent with two actions, set the standard deviation of each action to a different value while using the same decay rate for both standard deviations.

```
opt = rlTD3AgentOptions;
opt.TargetPolicySmoothModel.StandardDeviation = [0.1 0.2];
opt.TargetPolicySmoothModel.StandardDeviationDecayRate = 1e-4;
```

`PolicyUpdateFrequency` — Number of steps between policy updates
2 (default) | positive integer

Number of steps between policy updates, specified as a positive integer.

`ActorOptimizerOptions` — Actor optimizer options
`rlOptimizerOptions` object

Actor optimizer options, specified as an `rlOptimizerOptions` object. It allows you to specify training parameters of the actor approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see `rlOptimizerOptions` and `rlOptimizer`.

`CriticOptimizerOptions` — Critic optimizer options
`rlOptimizerOptions` object

Critic optimizer options, specified as an `rlOptimizerOptions` object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see `rlOptimizerOptions` and `rlOptimizer`.

`BatchDataRegularizerOptions` — Batch data regularizer options
`[]` (default) | `rlBehaviorCloningRegularizerOptions` object

Batch data regularizer options, specified as an `rlBehaviorCloningRegularizerOptions` object. These options are typically used to train the agent offline, from existing data. If you leave this option empty, no regularizer is used.

For more information, see `rlBehaviorCloningRegularizerOptions`.

Example: `opt.BatchDataRegularizerOptions = rlBehaviorCloningRegularizerOptions(BehaviorCloningRegularizerWeight=10)`

`TargetSmoothFactor` — Smoothing factor for target actor and critic updates
`0.005` (default) | positive scalar less than or equal to 1

Smoothing factor for target actor and critic updates, specified as a positive scalar less than or equal to 1. For more information, see “Target Update Methods”.

`TargetUpdateFrequency` — Number of steps between target actor and critic updates
`2` (default) | positive integer

Number of steps between target actor and critic updates, specified as a positive integer. For more information, see “Target Update Methods”.

`ResetExperienceBufferBeforeTraining` — Option for clearing the experience buffer
`false` (default) | `true`

Option for clearing the experience buffer before training, specified as a logical value.

`SequenceLength` — Maximum batch-training trajectory length when using RNN
`1` (default) | positive integer

Maximum batch-training trajectory length when using a recurrent neural network, specified as a positive integer. This value must be greater than `1` when using a recurrent neural network and `1` otherwise.

`MiniBatchSize` — Size of random experience mini-batch
`64` (default) | positive integer

Size of random experience mini-batch, specified as a positive integer. During each training episode, the agent randomly samples experiences from the experience buffer when computing gradients for updating the critic properties. Large mini-batches reduce the variance when computing gradients but increase the computational effort.

`NumStepsToLookAhead` — Number of future rewards used to estimate the value of the policy
`1` (default) | positive integer

Number of future rewards used to estimate the value of the policy, specified as a positive integer. For more information, see [1], Chapter 7.

`ExperienceBufferLength` — Experience buffer size
`10000` (default) | positive integer

Experience buffer size, specified as a positive integer. During training, the agent computes updates using a mini-batch of experiences randomly sampled from the buffer.

`SampleTime` — Sample time of agent
`1` (default) | positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every SampleTime seconds of simulation time. If SampleTime is -1, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, SampleTime is the time interval between consecutive elements in the output experience returned by sim or train. If SampleTime is -1, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

DiscountFactor — Discount factor

0.99 (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlTD3Agent	Twin-delayed deep deterministic (TD3) policy gradient reinforcement learning agent

Examples

Create TD3 Agent Options Object

Create an rlTD3AgentOptions object that specifies the mini-batch size.

```
opt = rlTD3AgentOptions(MiniBatchSize=48)
```

```
opt = 
  rlTD3AgentOptions with properties:

                       ExplorationModel: [1x1 rl.option.GaussianActionNoise]
                TargetPolicySmoothModel: [1x1 rl.option.GaussianActionNoise]
                  PolicyUpdateFrequency: 2
                  ActorOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
                 CriticOptimizerOptions: [1x2 rl.option.rlOptimizerOptions]
            BatchDataRegularizerOptions: []
                     TargetSmoothFactor: 0.0050
                  TargetUpdateFrequency: 2
    ResetExperienceBufferBeforeTraining: 0
                         SequenceLength: 1
                          MiniBatchSize: 48
                    NumStepsToLookAhead: 1
                 ExperienceBufferLength: 10000
                             SampleTime: 1
                         DiscountFactor: 0.9900
                             InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent sample time to 0.5.

```
opt.SampleTime = 0.5;
```

Algorithms

Noise Models

Gaussian Action Noise

A GaussianActionNoise object has the following numeric value properties.

Property	Description	Default Value (ExplorationModel)	Default Value (TargetPolicySmoothModel)
Mean	Noise mean value	0	0
StandardDeviationDecayRate	Decay rate of the standard deviation	0	0
StandardDeviation	Initial value of noise standard deviation	sqrt(0.1)	sqrt(0.2)
StandardDeviationMin	Minimum standard deviation, which must be less than StandardDeviation	0.01	0.01
LowerLimit	Noise sample lower limit	-Inf	-0.5
UpperLimit	Noise sample upper limit	Inf	0.5

At each time step k, the Gaussian noise v is sampled as shown in the following code.

```
w = Mean + randn(ActionSize).*StandardDeviation(k);
v(k+1) = min(max(w,LowerLimit),UpperLimit);
```

Where the initial value v(1) is defined by the InitialAction parameter. At each sample time step, the standard deviation decays as shown in the following code.

```
decayedStandardDeviation = StandardDeviation(k).*(1 - StandardDeviationDecayRate);
StandardDeviation(k+1) = max(decayedStandardDeviation,StandardDeviationMin);
```

Note that StandardDeviation is conserved between the end of an episode and the start of the next one. Therefore, it keeps on uniformly decreasing over multiple episodes until it reaches StandardDeviationMin.

Ornstein-Uhlenbeck Action Noise

An OrnsteinUhlenbeckActionNoise object has the following numeric value properties.

Property	Description	Default Value
InitialAction	Initial value of action	0
Mean	Noise mean value	0
MeanAttractionConstant	Constant specifying how quickly the noise model output is attracted to the mean	0.15

Property	Description	Default Value
StandardDeviationDecayRate	Decay rate of the standard deviation	0
StandardDeviation	Initial value of noise standard deviation	0.3
StandardDeviationMin	Minimum standard deviation	0

At each sample time step k, the noise value v(k) is updated using the following formula, where Ts is the agent sample time, and the initial value v(1) is defined by the InitialAction parameter.

```
v(k+1) = v(k) + MeanAttractionConstant.*(Mean - v(k)).*Ts
       + StandardDeviation(k).*randn(size(Mean)).*sqrt(Ts)
```

At each sample time step, the standard deviation decays as shown in the following code.

```
decayedStandardDeviation = StandardDeviation(k).*(1 - StandardDeviationDecayRate);
StandardDeviation(k+1) = max(decayedStandardDeviation,StandardDeviationMin);
```

You can calculate how many samples it will take for the standard deviation to be halved using this simple formula.

```
halflife = log(0.5)/log(1-StandardDeviationDecayRate);
```

Note that StandardDeviation is conserved between the end of an episode and the start of the next one. Therefore, it keeps on uniformly decreasing over multiple episodes until it reaches StandardDeviationMin.

For continuous action signals, it is important to set the noise standard deviation appropriately to encourage exploration. It is common to set StandardDeviation*sqrt(Ts) to a value between 1% and 10% of your action range.

If your agent converges on local optima too quickly, promote agent exploration by increasing the amount of noise; that is, by increasing the standard deviation. Also, to increase exploration, you can reduce the StandardDeviationDecayRate.

Version History

Introduced in R2020a

R2021a: Properties defining noise probability distribution in the GaussianActionNoise object have changed

Behavior changed in R2021a

The properties defining the probability distribution of the Gaussian action noise model have changed. This noise model is used by TD3 agents for exploration and target policy smoothing.

- The Variance property has been replaced by the StandardDeviation property.
- The VarianceDecayRate property has been replaced by the StandardDeviationDecayRate property.
- The VarianceMin property has been replaced by the StandardDeviationMin property.

When a GaussianActionNoise noise object saved from a previous MATLAB release is loaded, the value of VarianceDecayRate is copied to StandardDeviationDecayRate, while the square root of the values of Variance and VarianceMin are copied to StandardDeviation and StandardDeviationMin, respectively.

The Variance, VarianceDecayRate, and VarianceMin properties still work, but they are not recommended. To define the probability distribution of the Gaussian action noise model, use the new property names instead.

Update Code

This table shows how to update your code to use the new property names for rlTD3AgentOptions object td3opt.

Not Recommended	Recommended
td3opt.ExplorationModel.Variance = 0.5;	td3opt.ExplorationModel.StandardDeviation = sqrt(0.5);
td3opt.ExplorationModel.VarianceDecayRate = 0.1;	td3opt.ExplorationModel.StandardDeviationDecayRate = 0.1;
td3opt.ExplorationModel.VarianceMin = 0.1;	td3opt.ExplorationModel.StandardDeviationMin = sqrt(0.1);

R2021a: Property names defining noise probability distribution in the OrnsteinUhlenbeckActionNoise object have changed

Behavior changed in R2021a

The properties defining the probability distribution of the Ornstein-Uhlenbeck (OU) noise model have been renamed. TD3 agents use OU noise for exploration.

- The Variance property has been renamed StandardDeviation.
- The VarianceDecayRate property has been renamed StandardDeviationDecayRate.
- The VarianceMin property has been renamed StandardDeviationMin.

The default values of these properties remain the same. When an OrnsteinUhlenbeckActionNoise noise object saved from a previous MATLAB release is loaded, the values of Variance, VarianceDecayRate, and VarianceMin are copied in the StandardDeviation, StandardDeviationDecayRate, and StandardDeviationMin, respectively.

The Variance, VarianceDecayRate, and VarianceMin properties still work, but they are not recommended. To define the probability distribution of the OU noise model, use the new property names instead.

Update Code

This table shows how to update your code to use the new property names for rlTD3AgentOptions object td3opt.

Not Recommended	Recommended
td3opt.ExplorationModel.Variance = 0.5;	td3opt.ExplorationModel.StandardDeviation = sqrt(0.5);
td3opt.ExplorationModel.VarianceDecayRate = 0.1;	td3opt.ExplorationModel.StandardDeviationDecayRate = 0.1;

Not Recommended	Recommended
`td3opt.ExplorationModel.VarianceMin = 0.1;`	`td3opt.ExplorationModel.StandardDeviationMin = sqrt(0.1);`
`td3opt.TargetPolicySmoothModel.Variance = 0.5;`	`td3opt.TargetPolicySmoothModel.StandardDeviation = sqrt(0.5);`
`td3opt.TargetPolicySmoothModel.VarianceDecayRate = 0.1;`	`td3opt.TargetPolicySmoothModel.StandardDeviationDecayRate = 0.1;`
`td3opt.TargetPolicySmoothModel.VarianceMin = 0.1;`	`td3opt.TargetPolicySmoothModel.StandardDeviationMin = sqrt(0.1);`

R2022a: The default value of the `ResetExperienceBufferBeforeTraining` property has changed

Behavior changed in R2022a

The default value of the `ResetExperienceBufferBeforeTraining` has changed from `true` to `false`.

When creating a new TD3 agent, if you want to clear the experience buffer before training, you must specify `ResetExperienceBufferBeforeTraining` as `true`. For example, before training, set the property using dot notation.

```
agent.AgentOptions.ResetExperienceBufferBeforeTraining = true;
```

Alternatively, you can set the property to `true` in an `rlTD3AgentOptions` object and use this object to create the TD3 agent.

References

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning. Cambridge, Mass: The MIT Press, 2018.

See Also

Objects

`rlTD3Agent` | `rlAgentInitializationOptions` | `rlBehaviorCloningRegularizerOptions`

Topics

“Train TD3 Agent for PMSM Control”
“Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents”

rlTrainingFromDataOptions

Options to train reinforcement learning agents using existing data

Description

Use an `rlTrainingFromDataOptions` object to specify options to train an off-policy agent from existing data. Training options include the maximum number of epochs to train, criteria for stopping training and criteria for saving agents. After setting its options, use this object as an input argument for `trainFromData`.

For more information on training agents, see “Train Reinforcement Learning Agents”.

Creation

Syntax

```
tfdOpts = rlTrainingFromDataOptions
tfdOpts = rlTrainingOptions(Name=Value)
```

Description

`tfdOpts = rlTrainingFromDataOptions` returns a default options set to train an off-policy agent offline, from existing data.

`tfdOpts = rlTrainingOptions(Name=Value)` creates the training option set `tfdOpts` and sets its properties using one or more name-value arguments.

Properties

`MaxEpochs` — Maximum number of epochs to train the agent
`1000` (default) | positive integer

Maximum number of epochs to train the agent, specified as a positive integer. Each epoch has a fixed number of learning steps specified by `NumStepsPerEpoch`. Regardless of other criteria for termination, training terminates after `MaxEpochs`.

Example: `MaxEpochs=500`

`NumStepsPerEpoch` — Number of steps to run per epoch
`500` (default) | positive integer

Number of steps to run per epoch, specified as a positive integer.

Example: `NumStepsPerEpoch=1000`

`ExperienceBufferUpdateFrequency` — Buffer update period
`1` (default) | positive integer

Buffer update period, specified as a positive integer. For example, if the value of this option is `1` (default), then the buffer updates every epoch, if it is `2` the buffer updates every other epoch, and so on. Note that the experience buffer is not updated if it already contains all the available data.

Example: `ExperienceBufferUpdateFrequency=2`

`NumExperiencesPerExperienceBufferUpdate` — Number of experiences appended per buffer update

`[]` (default) | positive integer

Number of experiences appended per buffer update, specified as a positive integer or empty matrix. If the value of this option is left empty (default) then, at training time, it is automatically set to half the length of the experience buffer used by the agent.

Example: `NumExperiencesPerExperienceBufferUpdate=5e5`

`QValueObservations` — Batch of observations used to compute Q values

`[]` (default) | cell array

Batch of observations used to compute Q values, specified as an 1-by-N cell array, where N is the number of observation channels. Each cell must contain a batch of observations, along the batch dimension, for the corresponding observation channel. For example, if you have two observation channels carrying a 3-by-1 vector and a scalar, a batch of 10 random observations is `{rand(3,1,10),rand(1,1,10)}`.

If the value of this option is left empty (default) then, at training time, it is automatically set to a cell array in which each element corresponding to an observation channel is an array of zeros having the same dimensions of the observation, without any batch dimension.

Example: `QValueObservations={rand(3,1,10),rand(1,1,10)}`

`ScoreAveragingWindowLength` — Window length for averaging Q-values

`5` (default) | positive integer scalar

Window length for averaging Q-values, specified as a scalar. One termination and one saving options are expressed in terms of average Q-values. For these options, the average is calculated over the last `ScoreAveragingWindowLength` epochs.

Example: `ScoreAveragingWindowLength=10`

`StopTrainingCriteria` — Training termination condition

`"none"` (default) | `"QValue"` | ...

Training termination condition, specified as one of the following strings:

- `"none"` — Stop training after the agent is trained for the number of epochs specified in `MaxEpochs`.
- `"QValue"` — Stop training when the average Q-value (computed using the current critic and the observations specified in `QValueObservations`) over the last `ScoreAveragingWindowLength` epochs equals or exceeds the value specified in the `StopTrainingValue` option.

Example: `StopTrainingCriteria="QValue"`

`StopTrainingValue` — Critical value of training termination condition

`"none"` (default) | scalar

Critical value of the training termination condition, specified as a scalar. Training ends when the termination condition specified by the `StopTrainingCriteria` option equals or exceeds this value.

For instance, if `StopTrainingCriteria` is `"QValue"` and `StopTrainingValue` is `50`, then training terminates when the moving average Q-value (computed using the current critic and the observations specified in `QValueObservations`) over the number of epochs specified in `ScoreAveragingWindowLength` equals or exceeds `50`.

Example: `StopTrainingValue=50`

`SaveAgentCriteria` — Condition for saving agent during training
`"none"` (default) | `"EpochFrequency"` | `"QValue"` | ...

Condition for saving the agent during training, specified as one of the following strings:

- `"none"` — Do not save any agents during training.
- `"EpochFrequency"` — Save the agent when the number of epochs is an integer multiple of the value specified in the `SaveAgentValue` option.
- `"QValue"` — Save the agent when the when the average Q-value (computed using the current critic and the observations specified in `QValueObservations`) over the last `ScoreAveragingWindowLength` epochs equals or exceeds the value specified in `SaveAgentValue`.

Set this option to store candidate agents that perform in term of Q-value, or just to save agent at a fixed rate. For instance, if `SaveAgentCriteria` is `"EpochFrequency"` and `SaveAgentValue` is `5`, then the agent is saved every five epochs.

Example: `SaveAgentCriteria="EpochFrequency"`

`SaveAgentValue` — Critical value of condition for saving agent
`"none"` (default) | scalar

Critical value of the condition for saving the agent, specified as a scalar.

Example: `SaveAgentValue=10`

`SaveAgentDirectory` — Folder name for saved agents
`"savedAgents"` (default) | string | character vector

Folder name for saved agents, specified as a string or character vector. The folder name can contain a full or relative path. When an epoch occurs in which the condition specified by the `SaveAgentCriteria` and `SaveAgentValue` options are satisfied, the software saves the agents in a MAT file in this folder. If the folder does not exist, `train` creates it. When `SaveAgentCriteria` is `"none"`, this option is ignored and `train` does not create a folder.

Example: `SaveAgentDirectory = pwd + "\run1\Agents"`

`Verbose` — Option to display training progress at the command line
`false` or `0` (default) | `true` or `1`

Option to display training progress at the command line, specified as a numerical or logical `0` (`false`) or `1` (`true`). Set to `true` to write information from each training epoch to the MATLAB command line during training.

Example: `Verbose=false`

`Plots` — Option to display training progress with Episode Manager

`"training-progress"` (default) | `"none"`

Option to display training progress with Episode Manager, specified as `"training-progress"` or `"none"`. By default, calling `trainFromData` opens the Reinforcement Learning Episode Manager, which graphically and numerically displays information about the training progress, such as the reward for each epoch, average reward, number of epochs, and total number of steps. To turn off this display, set this option to `"none"`. For more information, see `train`.

Example: `Plots="none"`

Object Functions

trainFromData   Train off-policy reinforcement learning agent using existing data

Examples

Configure Options to Train Agent From Data

Create an options set to train a reinforcement learning agent offline, from an existing dataset.

Set the maximum number of epochs to 2000 and the maximum number of steps per epoch to 1000. Do not set any criteria to stop the training before 1000 epochs. Also, display training progress on the command line instead of using the Episode Manager.

```
tfdOpts = rlTrainingFromDataOptions(...
    MaxEpochs=2000,...
    NumStepsPerEpoch=1000,...
    Verbose=true,...
    Plots="none")
```

```
tfdOpts = 
  rlTrainingFromDataOptions with properties:

                                  MaxEpochs: 2000
                           NumStepsPerEpoch: 1000
            ExperienceBufferUpdateFrequency: 1
    NumExperiencesPerExperienceBufferUpdate: []
                         QValueObservations: []
                 ScoreAveragingWindowLength: 5
                       StopTrainingCriteria: "none"
                          StopTrainingValue: "none"
                          SaveAgentCriteria: "none"
                             SaveAgentValue: "none"
                         SaveAgentDirectory: "savedAgents"
                                    Verbose: 1
                                      Plots: "none"
                                StopOnError: "on"
```

Alternatively, create a default options set and use dot notation to change some of the values.

```
trainOpts = rlTrainingFromDataOptions;
trainOpts.MaxEpochs = 2000;
trainOpts.NumStepsPerEpoch = 1000;
trainOpts.Verbose = true;
trainOpts.Plots = "training-progress";

trainOpts

trainOpts = 
  rlTrainingFromDataOptions with properties:

                                MaxEpochs: 2000
                         NumStepsPerEpoch: 1000
            ExperienceBufferUpdateFrequency: 1
    NumExperiencesPerExperienceBufferUpdate: []
                       QValueObservations: []
                ScoreAveragingWindowLength: 5
                      StopTrainingCriteria: "none"
                         StopTrainingValue: "none"
                         SaveAgentCriteria: "none"
                            SaveAgentValue: "none"
                        SaveAgentDirectory: "savedAgents"
                                   Verbose: 1
                                     Plots: "training-progress"
                               StopOnError: "on"
```

You can now use `trainOpts` as an input argument to the `trainFromData` command.

Version History

Introduced in R2023a

See Also

Functions
`trainFromData`

Objects
`rlTrainingOptions` | `rlBehaviorCloningRegularizerOptions` | `rlConservativeQLearningOptions`

Topics
"Reinforcement Learning Agents"
"Train Reinforcement Learning Agents"

rlTrainingOptions

Options for training reinforcement learning agents

Description

Use an `rlTrainingOptions` object to specify options to train an agent within an environment. Training options include the maximum number of episodes to train, criteria for stopping training, criteria for saving agents, and options for using parallel computing. After setting its options, use this object as an input argument for `train`.

For more information on training agents, see “Train Reinforcement Learning Agents”.

Creation

Syntax

```
trainOpts = rlTrainingOptions
trainOpts = rlTrainingOptions(Name=Value)
```

Description

`trainOpts = rlTrainingOptions` returns the default options for training a reinforcement learning agent.

`trainOpts = rlTrainingOptions(Name=Value)` creates the training option set `trainOpts` and sets its “Properties” on page 3-431 using one or more name-value arguments.

Properties

`MaxEpisodes` — Maximum number of episodes to train the agent
`500` (default) | positive integer

Maximum number of episodes to train the agent, specified as a positive integer. Regardless of other criteria for termination, training terminates after `MaxEpisodes`.

Example: `MaxEpisodes=1000`

`MaxStepsPerEpisode` — Maximum number of steps to run per episode
`500` (default) | positive integer

Maximum number of steps to run per episode, specified as a positive integer. In general, you define episode termination conditions in the environment. This value is the maximum number of steps to run in the episode if other termination conditions are not met.

Example: `MaxStepsPerEpisode=1000`

`ScoreAveragingWindowLength` — Window length for averaging
`5` (default) | positive integer scalar | positive integer vector

Window length for averaging the scores, rewards, and number of steps for each agent, specified as a scalar or vector.

If the training environment contains a single agent, specify `ScoreAveragingWindowLength` as a scalar.

If the training environment is a multi-agent Simulink environment, specify a scalar to apply the same window length to all agents.

To use a different window length for each agent, specify `ScoreAveragingWindowLength` as a vector. In this case, the order of the elements in the vector correspond to the order of the agents used during environment creation.

For options expressed in terms of averages, `ScoreAveragingWindowLength` is the number of episodes included in the average. For instance, if `StopTrainingCriteria` is `"AverageReward"`, and `StopTrainingValue` is `500` for a given agent, then for that agent, training terminates when the average reward over the number of episodes specified in `ScoreAveragingWindowLength` equals or exceeds `500`. For the other agents, training continues until:

- All agents reach their stop criteria.
- The number of episodes reaches `MaxEpisodes`.
- You stop training by clicking the **Stop Training** button in Episode Manager or pressing **Ctrl-C** at the MATLAB command line.

Example: `ScoreAveragingWindowLength=10`

`StopTrainingCriteria` — Training termination condition

`"AverageSteps"` (default) | `"AverageReward"` | `"EpisodeCount"` | ...

Training termination condition, specified as one of the following strings:

- `"AverageSteps"` — Stop training when the running average number of steps per episode equals or exceeds the critical value specified by the option `StopTrainingValue`. The average is computed using the window `'ScoreAveragingWindowLength'`.
- `"AverageReward"` — Stop training when the running average reward equals or exceeds the critical value.
- `"EpisodeReward"` — Stop training when the reward in the current episode equals or exceeds the critical value.
- `"GlobalStepCount"` — Stop training when the total number of steps in all episodes (the total number of times the agent is invoked) equals or exceeds the critical value.
- `"EpisodeCount"` — Stop training when the number of training episodes equals or exceeds the critical value.

Example: `StopTrainingCriteria="AverageReward"`

`StopTrainingValue` — Critical value of training termination condition

`500` (default) | scalar | vector

Critical value of the training termination condition, specified as a scalar or a vector.

If the training environment contains a single agent, specify `StopTrainingValue` as a scalar.

If the training environment is a multi-agent Simulink environment, specify a scalar to apply the same termination criterion to all agents. To use a different termination criterion for each agent, specify

StopTrainingValue as a vector. In this case, the order of the elements in the vector corresponds to the order of the agents used during environment creation.

For a given agent, training ends when the termination condition specified by the StopTrainingCriteria option equals or exceeds this value. For the other agents, the training continues until:

- All agents reach their stop criteria.
- The number of episodes reaches maxEpisodes.
- You stop training by clicking the **Stop Training** button in Episode Manager or pressing **Ctrl-C** at the MATLAB command line.

For instance, if StopTrainingCriteria is "AverageReward", and StopTrainingValue is 100 for a given agent, then for that agent, training terminates when the average reward over the number of episodes specified in ScoreAveragingWindowLength equals or exceeds 100.

Example: StopTrainingValue=100

SaveAgentCriteria — Condition for saving agents during training

"none" (default) | "EpisodeReward" | "AverageReward" | "EpisodeCount" | ...

Condition for saving agents during training, specified as one of the following strings:

- "none" — Do not save any agents during training.
- "EpisodeReward" — Save the agent when the reward in the current episode equals or exceeds the critical value.
- "AverageSteps" — Save the agent when the running average number of steps per episode equals or exceeds the critical value specified by the option StopTrainingValue. The average is computed using the window 'ScoreAveragingWindowLength'.
- "AverageReward" — Save the agent when the running average reward over all episodes equals or exceeds the critical value.
- "GlobalStepCount" — Save the agent when the total number of steps in all episodes (the total number of times the agent is invoked) equals or exceeds the critical value.
- "EpisodeCount" — Save the agent when the number of training episodes equals or exceeds the critical value.

Set this option to store candidate agents that perform well according to the criteria you specify. When you set this option to a value other than "none", the software sets the SaveAgentValue option to 500. You can change that value to specify the condition for saving the agent.

For instance, suppose you want to store for further testing any agent that yields an episode reward that equals or exceeds 100. To do so, set SaveAgentCriteria to "EpisodeReward" and set the SaveAgentValue option to 100. When an episode reward equals or exceeds 100, train saves the corresponding agent in a MAT file in the folder specified by the SaveAgentDirectory option. The MAT file is called AgentK.mat, where K is the number of the corresponding episode. The agent is stored within that MAT file as saved_agent.

Example: SaveAgentCriteria="EpisodeReward"

SaveAgentValue — Critical value of condition for saving agents

"none" (default) | 500 | scalar | vector

Critical value of the condition for saving agents, specified as a scalar or a vector.

If the training environment contains a single agent, specify SaveAgentValue as a scalar.

If the training environment is a multi-agent Simulink environment, specify a scalar to apply the same saving criterion to each agent. To save the agents when one meets a particular criterion, specify SaveAgentValue as a vector. In this case, the order of the elements in the vector corresponds to the order of the agents used when creating the environment. When a criteria for saving an agent is met, all agents are saved in the same MAT file.

When you specify a condition for saving candidate agents using SaveAgentCriteria, the software sets this value to 500. Change the value to specify the condition for saving the agent. See the SaveAgentCriteria option for more details.

Example: SaveAgentValue=100

SaveAgentDirectory — Folder name for saved agents
"savedAgents" (default) | string | character vector

Folder name for saved agents, specified as a string or character vector. The folder name can contain a full or relative path. When an episode occurs in which the conditions specified by the SaveAgentCriteria and SaveAgentValue options are satisfied, the software saves the agents in a MAT file in this folder. If the folder does not exist, train creates it. When SaveAgentCriteria is "none", this option is ignored and train does not create a folder.

Example: SaveAgentDirectory = pwd + "\run1\Agents"

UseParallel — Flag for using parallel training
false (default) | true

Flag for using parallel training, specified as a logical. Setting this option to true configures training to use parallel processing to simulate the environment, thereby enabling usage of multiple cores, processors, computer clusters or cloud resources to speed up training. To specify options for parallel training, use the ParallelizationOptions property.

When UseParallel is true then for DQN, DDPG, TD3, and SAC the NumStepsToLookAhead property or the corresponding agent option object must be set to 1, otherwise an error is generated. This guarantees that experiences are stored contiguously. When AC agents are trained in parallel, a warning is generated if the StepsUntilDataIsSent property of the ParallelizationOptions object is set to a different value than the NumStepToLookAhead property of the AC agent option object.

Note that if you want to speed up deep neural network calculations (such as gradient computation, parameter update and prediction) using a local GPU, you do not need to set UseParallel to true. Instead, when creating your actor or critic representation, use an rlRepresentationOptions object in which the UseDevice option is set to "gpu". Using parallel computing or the GPU requires Parallel Computing Toolbox software. Using computer clusters or cloud resources additionally requires MATLAB Parallel Server. For more information about training using multicore processors and GPUs, see “Train Agents Using Parallel Computing and GPUs”.

Example: UseParallel=true

ParallelizationOptions — Options to control parallel training
ParallelTraining object

Parallelization options to control parallel training, specified as a ParallelTraining object. For more information about training using parallel computing, see “Train Reinforcement Learning Agents”.

The ParallelTraining object has the following properties, which you can modify using dot notation after creating the rlTrainingOptions object.

Mode — Parallel computing mode

"sync" (default) | "async"

Parallel computing mode, specified as one of the following:

- "sync" — Use parpool to run synchronous training on the available workers. In this case, workers pause execution until all workers are finished. The host updates the actor and critic parameters based on the results from all the workers and sends the updated parameters to all workers. Note that synchronous training is required for gradient-based parallelization, that is when DataToSendFromWorkers is set to "gradients" then Mode must be set to "sync".
- "async" — Use parpool to run asynchronous training on the available workers. In this case, workers send their data back to the host as soon as they finish and receive updated parameters from the host. The workers then continue with their task.

Example: Mode="async"

WorkerRandomSeeds — Randomizer initialization for workers

–1 (default) | –2 | vector

Randomizer initialization for workers, specified as one of the following:

- –1 — Assign a unique random seed to each worker. The value of the seed is the worker ID.
- –2 — Do not assign a random seed to the workers.
- Vector — Manually specify the random seed for each worker. The number of elements in the vector must match the number of workers.

Example: WorkerRandomSeeds=[1 2 3 4]

TransferBaseWorkspaceVariables — Option to send model and workspace variables to parallel workers

"on" (default) | "off"

Option to send model and workspace variables to parallel workers, specified as "on" or "off". When the option is "on", the client sends to the workers the variables defined in the base MATLAB workspace and used in the approximation models.

Example: TransferBaseWorkspaceVariables="off"

AttachedFiles — Additional files to attach to the parallel pool

[] (default) | string | string array

Additional files to attach to the parallel pool, specified as a string or string array.

Example: AttachedFiles="myInitFile.m"

SetupFcn — Function to run before training starts

[] (default) | function handle

Function to run before training starts, specified as a handle to a function having no input arguments. This function is run once per worker before training begins. Write this function to perform any processing that you need prior to training.

Example: AttachedFiles=@mySetupFcn

CleanupFcn — Function to run after training ends

[] (default) | function handle

Function to run after training ends, specified as a handle to a function having no input arguments. You can write this function to clean up the workspace or perform other processing after training terminates.

Example: AttachedFiles=@myCleanupFcn

Verbose — Option to display training progress at the command line

false (0) (default) | true (1)

Option to display training progress at the command line, specified as the logical values false (0) or true (1). Set to true to write information from each training episode to the MATLAB command line during training.

Example: Verbose=false

StopOnError — Option to stop training when error occurs

"on" (default) | "off"

Option to stop training when an error occurs during an episode, specified as "on" or "off". When this option is "off", errors are captured and returned in the SimulationInfo output of train, and training continues to the next episode.

Example: StopOnError="off"

Plots — Option to display training progress with Episode Manager

"training-progress" (default) | "none"

Option to display training progress with Episode Manager, specified as "training-progress" or "none". By default, calling train opens the Reinforcement Learning Episode Manager, which graphically and numerically displays information about the training progress, such as the reward for each episode, average reward, number of episodes, and total number of steps. For more information, see train. To turn off this display, set this option to "none".

Example: Plots="none"

Object Functions

train	Train reinforcement learning agents within a specified environment

Examples

Configure Options for Training

Create an options set for training a reinforcement learning agent. Set the maximum number of episodes and the maximum number of steps per episode to 1000. Configure the options to stop training when the average reward equals or exceeds 480, and turn on both the command-line display and Reinforcement Learning Episode Manager for displaying training results. You can set the options using name-value pair arguments when you create the options set. Any options that you do not explicitly set have their default values.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=1000,...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=480,...
    Verbose=true,...
    Plots="training-progress")

trainOpts = 
  rlTrainingOptions with properties:

                   MaxEpisodes: 1000
            MaxStepsPerEpisode: 1000
                   StopOnError: "on"
    ScoreAveragingWindowLength: 5
          StopTrainingCriteria: "AverageReward"
             StopTrainingValue: 480
             SaveAgentCriteria: "none"
                SaveAgentValue: "none"
            SaveAgentDirectory: "savedAgents"
                       Verbose: 1
                         Plots: "training-progress"
                   UseParallel: 0
        ParallelizationOptions: [1x1 rl.option.ParallelTraining]
```

Alternatively, create a default options set and use dot notation to change some of the values.

```
trainOpts = rlTrainingOptions;
trainOpts.MaxEpisodes = 1000;
trainOpts.MaxStepsPerEpisode = 1000;
trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = 480;
trainOpts.Verbose = true;
trainOpts.Plots = "training-progress";

trainOpts

trainOpts = 
  rlTrainingOptions with properties:

                   MaxEpisodes: 1000
            MaxStepsPerEpisode: 1000
                   StopOnError: "on"
    ScoreAveragingWindowLength: 5
          StopTrainingCriteria: "AverageReward"
             StopTrainingValue: 480
             SaveAgentCriteria: "none"
                SaveAgentValue: "none"
            SaveAgentDirectory: "savedAgents"
                       Verbose: 1
                         Plots: "training-progress"
                   UseParallel: 0
        ParallelizationOptions: [1x1 rl.option.ParallelTraining]
```

You can now use `trainOpts` as an input argument to the `train` command.

Configure Parallel Computing Options for Training

To turn on parallel computing for training a reinforcement learning agent, set the UseParallel training option to true.

```
trainOpts = rlTrainingOptions(UseParallel=true);
```

To configure your parallel training, configure the fields of the trainOpts.ParallelizationOptions. For example, specify the asynchronous training mode:

```
trainOpts.ParallelizationOptions.Mode = "async";
trainOpts.ParallelizationOptions
```

```
ans = 
  ParallelTraining with properties:

                             Mode: "async"
                WorkerRandomSeeds: -1
    TransferBaseWorkspaceVariables: "on"
                    AttachedFiles: []
                         SetupFcn: []
                       CleanupFcn: []
```

You can now use trainOpts as an input argument to the train command to perform training with parallel computing.

Configure Options for A3C Training

To train an agent using the asynchronous advantage actor-critic (A3C) method, you must set the agent and parallel training options appropriately.

When creating the AC agent, set the NumStepsToLookAhead value to be greater than 1. Common values are 64 and 128.

```
agentOpts = rlACAgentOptions(NumStepsToLookAhead=64);
```

Use agentOpts when creating your agent. Alternatively, create your agent first and then modify its options, including the actor and critic options later using dot notation.

Configure the training algorithm to use asynchronous parallel training.

```
trainOpts = rlTrainingOptions(UseParallel=true);
trainOpts.ParallelizationOptions.Mode = "async";
```

Configure the workers to return gradient data to the host. Also, set the number of steps before the workers send data back to the host to match the number of steps to look ahead.

```
trainOpts.ParallelizationOptions.DataToSendFromWorkers = ...
    "gradients";
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = ...
    agentOpts.NumStepsToLookAhead;
```

Use trainOpts when training your agent.

For an example on asynchronous advantage actor-critic agent training, see “Train AC Agent to Balance Cart-Pole System Using Parallel Computing”.

Version History

Introduced in R2019a

R2022a: Training Parallelization Options: `DataToSendFromWorkers` and `StepsUntilDataIsSent` properties are no longer active

Warns starting in R2022a

The property `DataToSendFromWorkers` of the `ParallelizationOptions` object is no longer active and will be removed in a future release. The data sent from the workers to the learner is now automatically determined based on agent type.

The property `StepsUntilDataIsSent` of the `ParallelizationOptions` object is no longer active and will be removed in a future release. Data is now sent from the workers to the learner at the end each episode.

R2022a: `rlTrainingOptions` is not recommended for multi agent training

Not recommended starting in R2022a

`rlTrainingOptions` is not recommended to concurrently train agents in a multi-agent environments. Use `rlMultiAgentTrainingOptions` instead.

`rlMultiAgentTrainingOptions` is specifically built for multi-agent reinforcement learning, and allows you to group agents according to a common learning strategy and specify whether their learning is centralized (that is all agents in a group share experiences) or decentralized (agents do not share experiences), whereas `rlTrainingOptions` only allows for decentralized learning.

See Also

Functions
`train`

Objects
`rlMultiAgentTrainingOptions` | `rlTrainingFromDataOptions`

Topics
“Train Reinforcement Learning Agents”
“Reinforcement Learning Agents”

rlTRPOAgent

Trust region policy optimization (TRPO) reinforcement learning agent

Description

Trust region policy optimization (TRPO) is a model-free, online, on-policy, policy gradient reinforcement learning method. This algorithm prevents significant performance drops compared to standard policy gradient methods by keeping the updated policy within a trust region close to the current policy. The action space can be either discrete or continuous. For continuous action spaces, this agent does not enforce constraints set in the action specification; therefore, if you need to enforce action constraints, you must do so within the environment.

Note TRPO agents do not support recurrent networks.

For more information on TRPO agents, see “Trust Region Policy Optimization (TRPO) Agents”. For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlTRPOAgent(observationInfo,actionInfo)
agent = rlTRPOAgent(observationInfo,actionInfo,initOpts)

agent = rlTRPOAgent(actor,critic)

agent = rlTRPOAgent(___,agentOptions)
```

Description

Create Agent from Observation and Action Specifications

`agent = rlTRPOAgent(observationInfo,actionInfo)` creates a trust region policy optimization (TRPO) agent for an environment with the given observation and action specifications, using default initialization options. The actor and critic in the agent use default deep neural networks built from the observation specification `observationInfo` and the action specification `actionInfo`. The `ObservationInfo` and `ActionInfo` properties of `agent` are set to the `observationInfo` and `actionInfo` input arguments, respectively.

`agent = rlTRPOAgent(observationInfo,actionInfo,initOpts)` creates a TRPO agent for an environment with the given observation and action specifications. The agent uses default networks configured using options specified in the `initOpts` object. TRPO agents do not support recurrent neural networks. For more information on the initialization options, see `rlAgentInitializationOptions`.

Create Agent from Actor and Critic

`agent = rlTRPOAgent(actor,critic)` creates a TRPO agent with the specified actor and critic, using the default options for the agent.

Specify Agent Options

`agent = rlTRPOAgent( ___ ,agentOptions)` creates a TRPO agent and sets the AgentOptions property to the `agentOptions` input argument. Use this syntax after any of the input arguments in the previous syntaxes.

Input Arguments

`initOpts` — Agent initialization options
`rlAgentInitializationOptions` object

Agent initialization options, specified as an `rlAgentInitializationOptions` object.

TRPO agents do not support recurrent neural networks. Therefore `initOpts.UseRNN`must be `false`.

`actor` — Actor
`rlDiscreteCategoricalActor` object | `rlContinuousGaussianActor` object

Actor that implements the policy, specified as an `rlDiscreteCategoricalActor` or `rlContinuousGaussianActor` function approximator object. For more information on creating actor approximators, see “Create Policies and Value Functions”.

`critic` — Critic
`rlValueFunction` object

Critic that estimates the discounted long-term reward, specified as an `rlValueFunction` object. For more information on creating critic approximators, see “Create Policies and Value Functions”.

Properties

`ObservationInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

If you create the agent by specifying an actor and critic, the value of `ObservationInfo` matches the value specified in the actor and critic objects.

You can extract `observationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`ActionInfo` — Action specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object

Action specifications, specified either as an `rlFiniteSetSpec` (for discrete action spaces) or `rlNumericSpec` (for continuous action spaces) object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

If you create the agent by specifying an actor and critic, the value of `ActionInfo` matches the value specified in the actor and critic objects.

You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

`AgentOptions` — Agent options

`rlTRPOAgentOptions` object

Agent options, specified as an `rlTRPOAgentOptions` object.

`UseExplorationPolicy` — Option to use exploration policy

`true` (default) | `false`

Option to use exploration policy when selecting actions, specified as a one of the following logical values.

- `true` — Use the base agent exploration policy when selecting actions in `sim` and `generatePolicyFunction`. In this case, the agent selects its actions by sampling its probability distribution, the policy is therefore stochastic and the agent explores its observation space.
- `false` — Use the base agent greedy policy (the action with maximum likelihood) when selecting actions in `sim` and `generatePolicyFunction`. In this case, the simulated agent and generated policy behave deterministically.

Note This option affects only simulation and deployment; it does not affect training.

`SampleTime` — Sample time of agent

positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations. The value of `SampleTime` matches the value specified in `AgentOptions`.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

Object Functions

train	Train reinforcement learning agents within a specified environment

sim	Simulate trained reinforcement learning agents within specified environment
getAction	Obtain action from agent, actor, or policy object given environment observations
getActor	Extract actor from reinforcement learning agent
setActor	Set actor of reinforcement learning agent
getCritic	Extract critic from reinforcement learning agent
setCritic	Set critic of reinforcement learning agent
generatePolicyFunction	Generate MATLAB function that evaluates policy of an agent or policy object

Examples

Create Discrete TRPO Agent from Observation and Action Specifications

Create an environment with a discrete action space, and obtain its observation and action specifications. For this example, load the environment used in the example “Create DQN Agent Using Deep Network Designer and Train Using Image Observations”. This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar with five possible elements (a torque of -2, -1, 0, 1, or 2 Nm applied to a swinging pole).

```
% Load predefined environment
env = rlPredefinedEnv("SimplePendulumWithImage-Discrete");
```

Obtain the observation and action specifications for this environment.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a TRPO agent from the environment observation and action specifications.

```
agent = rlTRPOAgent(obsInfo,actInfo);
```

To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[-2]}
```

You can now test and train the agent within the environment.

Create Continuous TRPO Agent Using Initialization Options

Create an environment with a continuous action space and obtain its observation and action specifications. For this example, load the environment used in the example “Train DDPG Agent to

Swing Up and Balance Pendulum with Image Observation". This environment has two observations: a 50-by-50 grayscale image and a scalar (the angular velocity of the pendulum). The action is a scalar representing a torque ranging continuously from -2 to 2 Nm.

```
env = rlPredefinedEnv("SimplePendulumWithImage-Continuous");
```

Obtain observation and action specifications for this environment.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create an agent initialization options object, specifying that each hidden fully connected layer in the network must have 128 neurons. TRPO agents do not support recurrent networks, so setting the `UseRNN` option to `true` generates an error when the agent is created.

```
initOpts = rlAgentInitializationOptions(NumHiddenUnit=128);
```

The agent creation function initializes the actor and critic networks randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create a TRPO agent from the environment observation and action specifications using the specified initialization options.

```
agent = rlTRPOAgent(obsInfo,actInfo,initOpts);
```

Extract the deep neural networks from both the agent actor and critic.

```
actorNet = getModel(getActor(agent));
criticNet = getModel(getCritic(agent));
```

To verify that each hidden fully connected layer has 128 neurons, you can display the layers on the MATLAB® command window,

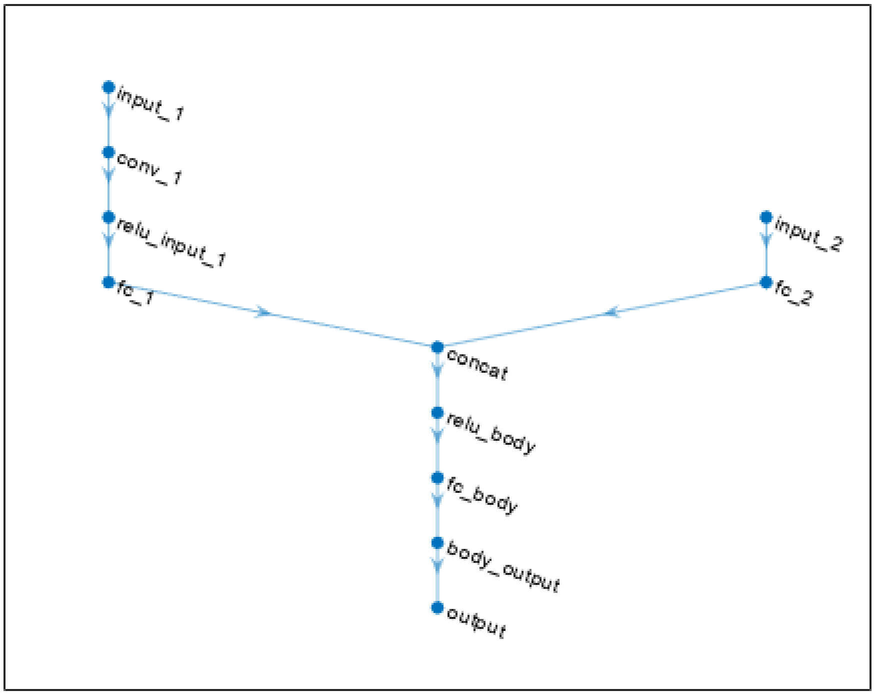
```
criticNet.Layers
```

or visualize the structure interactively using `analyzeNetwork`.

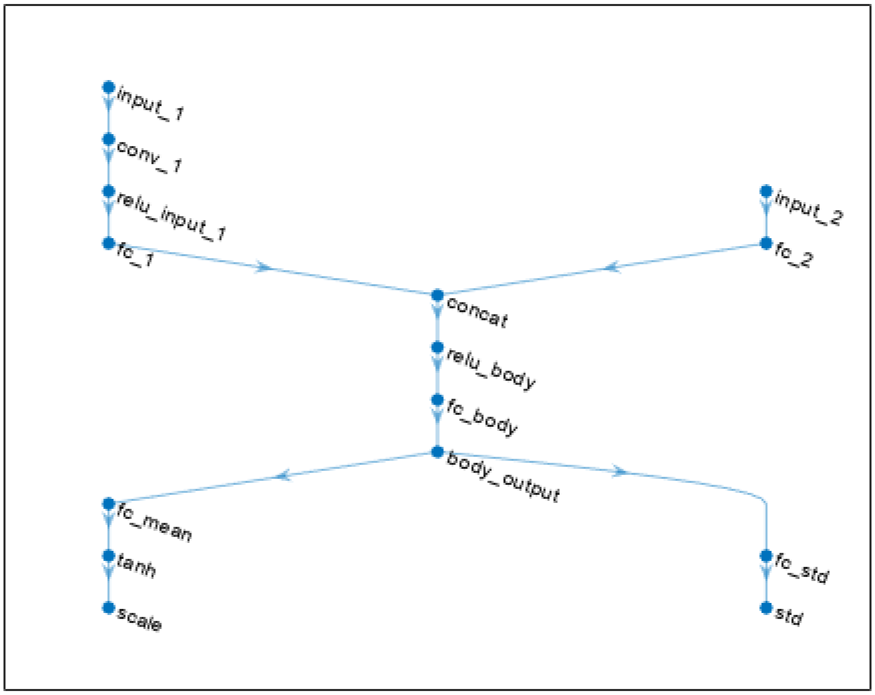
```
analyzeNetwork(criticNet)
```

Plot the critic and actor networks.

```
plot(criticNet)
```



```
plot(actorNet)
```



To check your agent, use `getAction` to return the action from a random observation.

```
getAction(agent,{rand(obsInfo(1).Dimension),rand(obsInfo(2).Dimension)})
```

```
ans = 1x1 cell array
    {[0.9228]}
```

You can now test and train the agent within the environment.

Create Trust Region Policy Optimization Agent

Create an environment interface, and obtain its observation and action specifications.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

TRPO agents use a parametrized value function approximator to estimate the value of the policy. A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (which receives the content of the observation channel, as specified by obs`Info`) and one output layer (which returns the scalar value). Note that prod(obsInfo.Dimension) returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects.

```
criticNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(100)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert to a dlnetwork object and display the number of parameters.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 601

   Inputs:
      1   'input'   4 features
```

Create the critic approximator object using criticNet and the observation specification. For more information on value function approximators, see rlValueFunction.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Check the critic with a random observation input.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.2479
```

TRPO agents use a parametrized stochastic policy, which for discrete action spaces is implemented by a discrete categorical actor. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by obsInfo) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by actInfo. Note that numel(actInfo.Dimension) returns the number of possible actions.

Define the network as an array of layer objects.

```
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(200)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Dimension))
    ];
```

Convert to a dlnetwork object and display the number of parameters.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)

   Initialized: true

   Number of learnables: 1.4k

   Inputs:
      1   'input'   4 features
```

Create the actor using `actorNet` and the observation and action specifications. For more information on discrete categorical actors, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})

ans = 1x1 cell array
    {[-10]}
```

Create a TRPO agent using the actor and the critic.

```
agent = rlTRPOAgent(actor,critic)

agent = 
  rlTRPOAgent with properties:

            AgentOptions: [1x1 rl.option.rlTRPOAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Specify agent options, including training options for the actor and the critic.

```
agent.AgentOptions.ExperienceHorizon = 1024;
agent.AgentOptions.DiscountFactor = 0.95;

agent.AgentOptions.CriticOptimizerOptions.LearnRate = 8e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
```

Check your agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})

ans = 1x1 cell array
    {[-10]}
```

You can now test and train the agent against the environment.

Create Continuous TRPO Agent

Create an environment with a continuous action space, and obtain its observation and action specifications. For this example, load the double integrator continuous action space environment used in the example “Train DDPG Agent to Control Double Integrator System”. The observation from the environment is a vector containing the position and velocity of a mass. The action is a scalar representing a force applied to the mass, ranging continuously from -2 to 2 Newton.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
obsInfo = getObservationInfo(env)
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "states"
    Description: "x, dx"
      Dimension: [2 1]
       DataType: "double"
```

```
actInfo = getActionInfo(env)
```

```
actInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "force"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

In this example, the action is a scalar value representing a force ranging from -2 to 2 Newton. To make sure that the output from the agent is in this range, you perform an appropriate scaling operation. Store these limits so you can easily access them later.

```
actInfo.LowerLimit=-2;
actInfo.UpperLimit=2;
```

The actor and critic networks are initialized randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

TRPO agents use a parametrized value function approximator to estimate the value of the policy. A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (which receives the content of the observation channel, as specified by obsInfo) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects.

```
criticNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(100)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert to a `dlnetwork` object and display the number of parameters.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 401

   Inputs:
      1   'input'   2 features
```

Create the critic approximator object using `criticNet` and the observation specification. For more information on value function approximators, see `rlValueFunction`.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Check the critic with a random observation input.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.0899
```

TRPO agents use a parametrized stochastic policy, which for continuous action spaces is implemented by a continuous Gaussian actor. This actor takes an observation as input and returns as output a random action sampled from a Gaussian probability distribution.

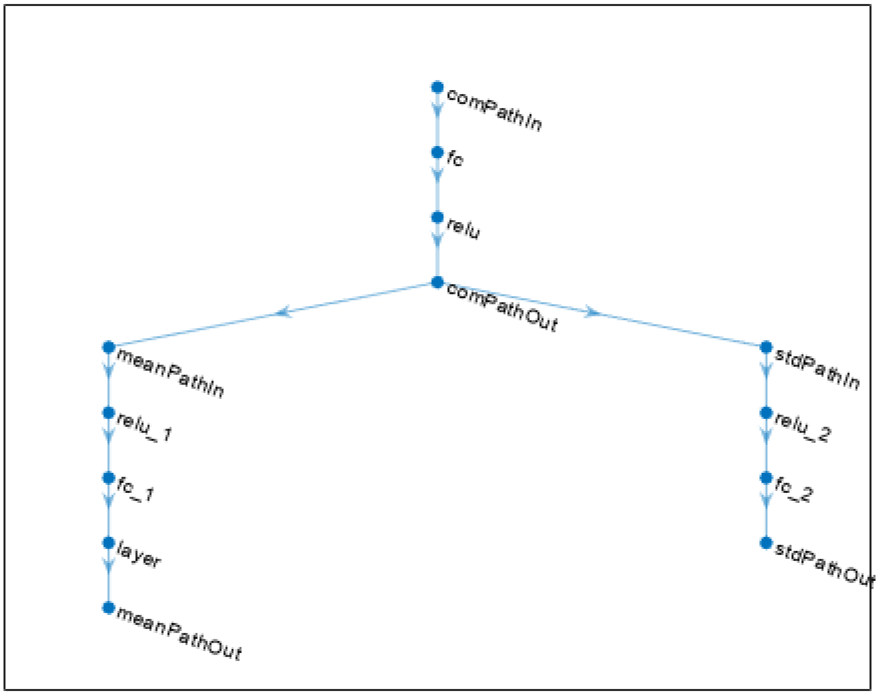
To approximate the mean values and standard deviations of the Gaussian distribution, you must use a neural network with two output layers, each having as many elements as the dimension of the action space. One output layer must return a vector containing the mean values for each action dimension. The other must return a vector containing the standard deviation for each action dimension.

Note that standard deviations must be nonnegative and mean values must fall within the range of the action. Therefore the output layer that returns the standard deviations must be a softplus or ReLU layer, to enforce nonnegativity, while the output layer that returns the mean values must be a scaling layer, to scale the mean values to the output range.

For this example the environment has only one observation channel and therefore the network has only one input layer.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% Define common input path layer
commonPath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="comPathIn")
    fullyConnectedLayer(100)
    reluLayer
    fullyConnectedLayer(1,Name="comPathOut")
    ];

% Define mean value path
meanPath = [
    fullyConnectedLayer(32,Name="meanPathIn")
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension));
    tanhLayer;
    scalingLayer(Name="meanPathOut",Scale=actInfo.UpperLimit)
    ];

% Define standard deviation path
sdevPath = [
    fullyConnectedLayer(32,"Name","stdPathIn")
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension));
    softplusLayer(Name="stdPathOut")
    ];

% Add layers to layerGraph object
actorNet = layerGraph(commonPath);
actorNet = addLayers(actorNet,meanPath);
actorNet = addLayers(actorNet,sdevPath);

% Connect paths
actorNet = connectLayers(actorNet,"comPathOut","meanPathIn/in");
actorNet = connectLayers(actorNet,"comPathOut","stdPathIn/in");

% Plot network
plot(actorNet)
```



```
% Convert to dlnetwork and display number of weights
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 595

   Inputs:
      1   'comPathIn'   2 features
```

Create the actor approximator object using `actorNet` and the environment specifications. For more information, on continuous Gaussian actors, see `rlContinuousGaussianActor`.

```
actor = rlContinuousGaussianActor(actorNet, obsInfo, actInfo, ...
    "ActionMeanOutputNames","meanPathOut",...
    "ActionStandardDeviationOutputNames","stdPathOut",...
    ObservationInputNames="comPathIn");
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.2732]}
```

Create a TRPO agent using the actor and the critic.

```
agent = rlTRPOAgent(actor,critic)
```

```
agent = 
  rlTRPOAgent with properties:

            AgentOptions: [1x1 rl.option.rlTRPOAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlNumericSpec]
              SampleTime: 1
```

Specify agent options, including training options for the actor and the critic.

```
agent.AgentOptions.ExperienceHorizon = 1024;
agent.AgentOptions.DiscountFactor = 0.95;

agent.AgentOptions.CriticOptimizerOptions.LearnRate = 8e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
```

Check your agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-0.5005]}
```

You can now test and train the agent within the environment.

Tips

- For continuous action spaces, this agent does not enforce the constraints set by the action specification. In this case, you must enforce action space constraints within the environment.
- While tuning the learning rate of the actor network is necessary for PPO agents, it is not necessary for TRPO agents.
- For high-dimensional observations, such as for images, it is recommended to use PPO, SAC, or TD3 agents.

Version History

Introduced in R2021b

See Also

Apps

Reinforcement Learning Designer | **Deep Network Designer**

Functions

`getAction` | `getActor` | `getCritic` | `getModel` | `generatePolicyFunction` | `generatePolicyBlock` | `getActionInfo` | `getObservationInfo`

Objects

`rlTRPOAgentOptions` | `rlAgentInitializationOptions` | `rlValueFunction` | `rlDiscreteCategoricalActor` | `rlContinuousGaussianActor` | `rlACAgent` | `rlPGAgent` | `rlPPOAgent`

Blocks

RL Agent | Policy

Topics

"Train Reinforcement Learning Agents"
"Trust Region Policy Optimization (TRPO) Agents"
"Reinforcement Learning Agents"

rlTRPOAgentOptions

Options for TRPO agent

Description

Use an `rlTRPOAgentOptions` object to specify options for trust region policy optimization (TRPO) agents. To create a TRPO agent, use `rlTRPOAgent`.

For more information on TRPO agents, see “Trust Region Policy Optimization (TRPO) Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
opt = rlTRPOAgentOptions
opt = rlTRPOAgentOptions(Name,Value)
```

Description

`opt = rlTRPOAgentOptions` creates an `rlPPOAgentOptions` object for use as an argument when creating a TRPO agent using all default settings. You can modify the object properties using dot notation.

`opt = rlTRPOAgentOptions(Name,Value)` sets option properties on page 3-455 using name-value arguments. For example, `rlTRPOAgentOptions('DiscountFactor',0.95)` creates an option set with a discount factor of `0.95`. You can specify multiple name-value arguments. Enclose each property name in quotes.

Properties

`ExperienceHorizon` — Number of steps the agent interacts with the environment before learning
`512` (default) | positive integer

Number of steps the agent interacts with the environment before learning from its experience, specified as a positive integer. When the agent is trained in parallel, `ExperienceHorizon` is ignored, and the whole episode is used to compute the gradients.

The `ExperienceHorizon` value must be greater than or equal to the `MiniBatchSize` value.

`MiniBatchSize` — Mini-batch size
`128` (default) | positive integer

Mini-batch size used for each learning epoch, specified as a positive integer. When the agent uses a recurrent neural network, `MiniBatchSize` is treated as the training trajectory length.

The `MiniBatchSize` value must be less than or equal to the `ExperienceHorizon` value.

`EntropyLossWeight` — Entropy loss weight

`0.01` (default) | scalar value between `0` and `1`

Entropy loss weight, specified as a scalar value between `0` and `1`. A higher entropy loss weight value promotes agent exploration by applying a penalty for being too certain about which action to take. Doing so can help the agent move out of local optima.

When gradients are computed during training, an additional gradient component is computed for minimizing the entropy loss. For more information, see “Entropy Loss”.

`NumEpoch` — Number of epochs

`1` (default) | positive integer

Number of epochs for which the actor and critic networks learn from the current experience set, specified as a positive integer.

`AdvantageEstimateMethod` — Method for estimating advantage values

`"gae"` (default) | `"finite-horizon"`

Method for estimating advantage values, specified as one of the following:

- `"gae"` — Generalized advantage estimator
- `"finite-horizon"` — Finite horizon estimation

For more information on these methods, see the training algorithm information in “Proximal Policy Optimization (PPO) Agents”.

`GAEFactor` — Smoothing factor for generalized advantage estimator

`0.95` (default) | scalar value between `0` and `1`

Smoothing factor for generalized advantage estimator, specified as a scalar value between `0` and `1`, inclusive. This option applies only when the `AdvantageEstimateMethod` option is `"gae"`

`ConjugateGradientDamping` — Conjugate gradient damping factor

`1e-4` (default) | nonnegative scalar

Conjugate gradient damping factor for numerical stability, specified as a nonnegative scalar.

`KLDivergenceLimit` — Upper limit for KL divergence

`0.01` (default) | positive scalar

Upper limit for the Kullback-Leibler (KL) divergence between the old policy and the current policy, specified as a positive scalar.

`NumIterationsConjugateGradient` — Maximum number of iterations for conjugate gradient decent

`10` (default) | positive integer

Maximum number of iterations for conjugate gradient decent, specified as positive integer.

`NumIterationsLineSearch` — Number of iterations for line search

`10` (default) | positive integer

Number of iterations for line search, specified as a positive integer.

Typically, the default value works well for most cases.

`ConjugateGradientResidualTolerance` — Conjugate gradient residual tolerance factor

`1e-8` (default) | positive scalar

Conjugate gradient residual tolerance, specified as a positive scalar. Once the residual for the conjugate gradient algorithm is below this tolerance, the algorithm stops.

Typically, the default value works well for most cases.

`NormalizedAdvantageMethod` — Method for normalizing advantage function

`"none"` (default) | `"current` | `"moving"`

Method for normalizing advantage function values, specified as one of the following:

- `"none"` — Do not normalize advantage values
- `"current"` — Normalize the advantage function using the mean and standard deviation for the current mini-batch of experiences.
- `"moving"` — Normalize the advantage function using the mean and standard deviation for a moving window of recent experiences. To specify the window size, set the `AdvantageNormalizingWindow` option.

In some environments, you can improve agent performance by normalizing the advantage function during training. The agent normalizes the advantage function by subtracting the mean advantage value and scaling by the standard deviation.

`AdvantageNormalizingWindow` — Window size for normalizing advantage function

`1e6` (default) | positive integer

Window size for normalizing advantage function values, specified as a positive integer. Use this option when the `NormalizedAdvantageMethod` option is `"moving"`.

`CriticOptimizerOptions` — Critic optimizer options

`rlOptimizerOptions` object

Critic optimizer options, specified as an `rlOptimizerOptions` object. It allows you to specify training parameters of the critic approximator such as learning rate, gradient threshold, as well as the optimizer algorithm and its parameters. For more information, see `rlOptimizerOptions` and `rlOptimizer`.

`SampleTime` — Sample time of agent

`1` (default) | positive scalar | `-1`

Sample time of agent, specified as a positive scalar or as `-1`. Setting this parameter to `-1` allows for event-based simulations.

Within a Simulink environment, the RL Agent block in which the agent is specified to execute every `SampleTime` seconds of simulation time. If `SampleTime` is `-1`, the block inherits the sample time from its parent subsystem.

Within a MATLAB environment, the agent is executed every time the environment advances. In this case, `SampleTime` is the time interval between consecutive elements in the output experience returned by `sim` or `train`. If `SampleTime` is `-1`, the time interval between consecutive elements in the returned output experience reflects the timing of the event that triggers the agent execution.

DiscountFactor — Discount factor

0.99 (default) | positive scalar less than or equal to 1

Discount factor applied to future rewards during training, specified as a positive scalar less than or equal to 1.

Object Functions

rlTRPOAgent	Trust region policy optimization (TRPO) reinforcement learning agent

Examples

Create TRPO Agent Options Object

Create a TRPO agent options object, specifying the discount factor.

```
opt = rlTRPOAgentOptions(DiscountFactor=0.9)
```

```
opt = 
  rlTRPOAgentOptions with properties:

                    ExperienceHorizon: 512
                        MiniBatchSize: 128
                    EntropyLossWeight: 0.0100
                             NumEpoch: 3
              AdvantageEstimateMethod: "gae"
                            GAEFactor: 0.9500
             ConjugateGradientDamping: 0.1000
                    KLDivergenceLimit: 0.0100
        NumIterationsConjugateGradient: 10
              NumIterationsLineSearch: 10
    ConjugateGradientResidualTolerance: 1.0000e-08
             NormalizedAdvantageMethod: "none"
            AdvantageNormalizingWindow: 1000000
                CriticOptimizerOptions: [1x1 rl.option.rlOptimizerOptions]
                           SampleTime: 1
                       DiscountFactor: 0.9000
                           InfoToSave: [1x1 struct]
```

You can modify options using dot notation. For example, set the agent sample time to `0.1`.

```
opt.SampleTime = 0.1;
```

Version History

Introduced in R2021b

R2022a: Simulation and deployment: UseDeterministicExploitation will be removed

Warns starting in R2022a

The property `UseDeterministicExploitation` of the `rlTRPOAgentOptions` object will be removed in a future release. Use the `UseExplorationPolicy` property of `rlTRPOAgent` instead.

Previously, you set `UseDeterministicExploitation` as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.AgentOptions.UseDeterministicExploitation = true;
  ```
- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.AgentOptions.UseDeterministicExploitation = false;
  ```

Starting in R2022a, set `UseExplorationPolicy` as follows.

- Force the agent to always select the action with maximum likelihood, thereby using a greedy deterministic policy for simulation and deployment.

  ```
  agent.UseExplorationPolicy = false;
  ```
- Allow the agent to select its action by sampling its probability distribution for simulation and policy deployment, thereby using a stochastic policy that explores the observation space.

  ```
  agent.UseExplorationPolicy = true;
  ```

Similarly to `UseDeterministicExploitation`, `UseExplorationPolicy` affects only simulation and deployment; it does not affect training.

See Also

Objects
`rlTRPOAgent` | `rlAgentInitializationOptions`

Topics
"Trust Region Policy Optimization (TRPO) Agents"

rlValueFunction

Value function approximator object for reinforcement learning agents

Description

This object implements a value function approximator object that you can use as a critic for a reinforcement learning agent. A value function is a mapping from an environment observation to the value of a policy. Specifically, its output is a scalar that represents the expected discounted cumulative long-term reward when an agent starts from the state corresponding to the given observation and executes actions according to a given policy afterwards. After you create an `rlValueFunction` critic, use it to create an agent such as an `rlACAgent`, `rlPGAgent`, or `rlPPOAgent` agent. For an example of this workflow, see “Create Actor and Critic Representations” on page 3-479. For more information on creating value functions, see “Create Policies and Value Functions”.

Creation

Syntax

```
critic = rlValueFunction(net,observationInfo)
critic = rlValueFunction(net,ObservationInputNames=netObsNames)

critic = rlValueFunction(tab,observationInfo)

critic = rlValueFunction({basisFcn,W0},observationInfo)

critic = rlValueFunction( ___ ,UseDevice=useDevice)
```

Description

`critic = rlValueFunction(net,observationInfo)` creates the value-function object `critic` using the deep neural network `net` as approximation model, and sets the `ObservationInfo` property of `critic` to the `observationInfo` input argument. The network input layers are automatically associated with the environment observation channels according to the dimension specifications in `observationInfo`.

`critic = rlValueFunction(net,ObservationInputNames=netObsNames)` specifies the network input layer names to be associated with the environment observation channels. The function assigns, in sequential order, each environment observation channel specified in `observationInfo` to the layer specified by the corresponding name in the string array `netObsNames`. Therefore, the network input layers, ordered as the names in `netObsNames`, must have the same data type and dimensions as the observation channels, as ordered in `observationInfo`.

`critic = rlValueFunction(tab,observationInfo)` creates the value function object `critic` with a *discrete observation space*, from the table `tab`, which is an `rlTable` object containing a column array with as many elements as the number of possible observations. The function sets the `ObservationInfo` property of `critic` to the `observationInfo` input argument, which in this case must be a scalar `rlFiniteSetSpec` object.

`critic = rlValueFunction({basisFcn,W0},observationInfo)` creates the value function object `critic` using a custom basis function as underlying approximator. The first input argument is a two-element cell array whose first element is the handle `basisFcn` to a custom basis function and whose second element is the initial weight vector `W0`. The function sets the `ObservationInfo` property of `critic` to the `observationInfo` input argument.

`critic = rlValueFunction( ___ ,UseDevice=useDevice)` specifies the device used to perform computations for the `critic` object, and sets the `UseDevice` property of `critic` to the `useDevice` input argument. You can use this syntax with any of the previous input-argument combinations.

Input Arguments

net — Deep neural network

array of `Layer` objects | `layerGraph` object | `DAGNetwork` object | `SeriesNetwork` object | `dlNetwork` object (preferred)

Deep neural network used as the underlying approximator within the critic, specified as one of the following:

- Array of `Layer` objects
- `layerGraph` object
- `DAGNetwork` object
- `SeriesNetwork` object
- `dlnetwork` object

Note Among the different network representation options, `dlnetwork` is preferred, since it has built-in validation checks and supports automatic differentiation. If you pass another network object as an input argument, it is internally converted to a `dlnetwork` object. However, best practice is to convert other representations to `dlnetwork` explicitly *before* using it to create a critic or an actor for a reinforcement learning agent. You can do so using `dlnet=dlnetwork(net)`, where `net` is any Deep Learning Toolbox neural network object. The resulting `dlnet` is the `dlnetwork` object that you use for your critic or actor. This practice allows a greater level of insight and control for cases in which the conversion is not straightforward and might require additional specifications.

The network must have as many input layers as the number of environment observation channels (with each input layer receiving input from an observation channel), and a single output layer returning a scalar value.

`rlValueFunction` objects support recurrent deep neural networks.

The learnable parameters of the critic are the weights of the deep neural network. For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

netObsNames — Network input layers names corresponding to the environment observation channels

string array | cell array of character vectors

Network input layers names corresponding to the environment observation channels, specified as a string array or a cell array of character vectors. When you use this argument after `'ObservationInputNames'`, the function assigns, in sequential order, each environment observation channel specified in `observationInfo` to each network input layer specified by the corresponding name in the string array `netObsNames`. Therefore, the network input layers, ordered as the names in `netObsNames`, must have the same data type and dimensions as the observation channels, as ordered in `observationInfo`.

Example: `{"NetInput1_airspeed","NetInput2_altitude"}`

`tab` — Value table
`rlTable` object

Value table, specified as an `rlTable` object containing a column vector with length equal to the number of possible observations from the environment. Each element is the predicted discounted cumulative long-term reward when the agent starts from the given observation and takes the best possible action. The elements of this vector are the learnable parameters of the representation.

`basisFcn` — Custom basis function
function handle

Custom basis function, specified as a function handle to a user-defined function. The user defined function can either be an anonymous function or a function on the MATLAB path. The output of the critic is the scalar `c = W'*B`, where `W` is a weight vector containing the learnable parameters and `B` is the column vector returned by the custom basis function.

Your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here, `obs1` to `obsN` are inputs in the same order and with the same data type and dimensions as the environment observation channels defined in `observationInfo`.

For an example on how to use a basis function to create a value function critic with a mixed continuous and discrete observation space, see “Create Mixed Observation Space Value Function Critic from Custom Basis Function” on page 3-473.

Example: `@(obs1,obs2,obs3) [obs3(1)*obs1(1)^2; abs(obs2(5)+obs1(2))]`

`W0` — Initial value of basis function weights
column vector

Initial value of the basis function weights `W`, specified as a column vector having the same length as the vector returned by the basis function.

Properties

`ObservationInfo` — Observation specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

`rlValueFucntion` sets the `ObservationInfo` property of `critic` to the input argument `observationInfo`.

You can extract `ObservationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually.

UseDevice — Computation device used for training and simulation

"cpu" (default) | "gpu"

Computation device used to perform operations such as gradient computation, parameter update and prediction during training and simulation, specified as either "cpu" or "gpu".

The "gpu" option requires both Parallel Computing Toolbox software and a CUDA enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use gpuDevice (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating an agent on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

To speed up training by using parallel processing over multiple cores, you do not need to use this argument. Instead, when training your agent, use an rlTrainingOptions object in which the UseParallel option is set to true. For more information about training using multicore processors and GPUs for training, see “Train Agents Using Parallel Computing and GPUs”.

Example: "gpu"

Object Functions

rlACAgent	Actor-critic (AC) reinforcement learning agent
rlPGAgent	Policy gradient (PG) reinforcement learning agent
rlPPOAgent	Proximal policy optimization (PPO) reinforcement learning agent
getValue	Obtain estimated value from a critic given environment observations and actions
evaluate	Evaluate function approximator object given observation (or observation-action) input data
gradient	Evaluate gradient of function approximator object given observation and action input data
accelerate	Option to accelerate computation of gradient for approximator object based on neural network
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object
setModel	Set function approximation model for actor or critic
getModel	Get function approximator model from actor or critic

Examples

Create Value Function Critic from Deep Neural Network

Create an observation specification object (or alternatively use getObservationInfo to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (which returns the content of the observation channel, as specified by obsInfo) and one output layer (returning the scalar value). Note that prod(obsInfo.Dimension) returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects.

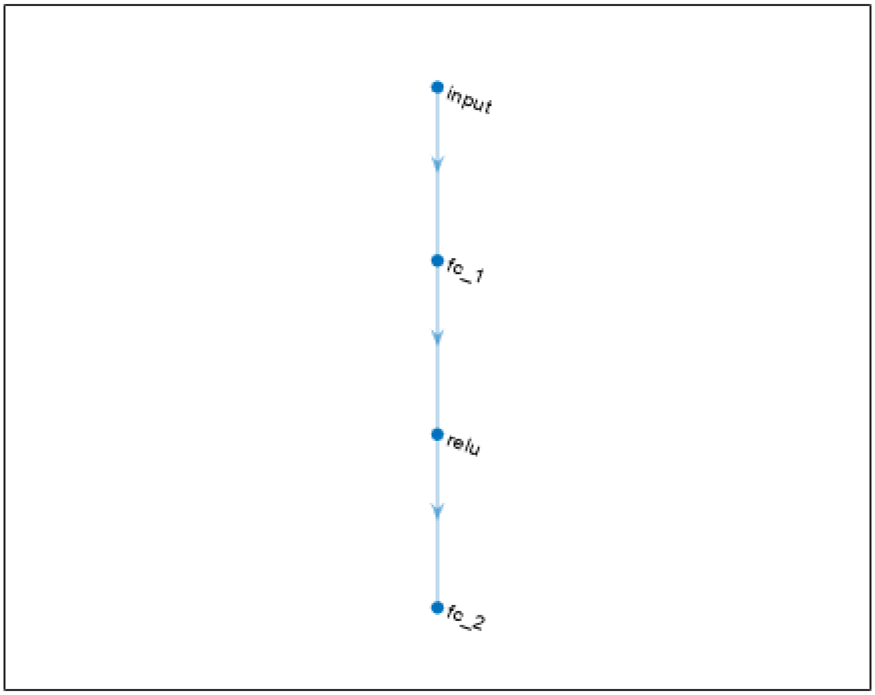
```
net = [
    featureInputLayer(prod(obsInfo.Dimension));
    fullyConnectedLayer(10);
    reluLayer;
    fullyConnectedLayer(1)
    ];
```

Convert the network to a dlnetwork object.

```
dlnet = dlnetwork(net);
```

You can plot the network using plot and display its main characteristics, like the number of weights, using summary.

```
plot(dlnet)
```



```
summary(dlnet)

   Initialized: true

   Number of learnables: 61

   Inputs:
      1   'input'   4 features
```

Create the critic using the network and the observation specification object.

```
critic = rlValueFunction(dlnet,obsInfo)

critic = 
  rlValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

To check your critic, use `getValue` to return the value of a random observation, using the current network weights.

```
v = getValue(critic,{rand(obsInfo.Dimension)})

v = single
    0.5196
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given observation specification object. Examples of agents that can work with a continuous observation space, and use a value function critic, are `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, and `rlTRPOAgent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Actor and Critic

Create an actor and a critic that you can use to define a reinforcement learning agent such as an Actor-Critic (AC) agent. For this example, create actor and critic for an agent that can be trained against the cart-pole environment described in “Train AC Agent to Balance Cart-Pole System”.

First, create the environment. Then, extract the observation and action specifications from the environment. You need these specifications to define the agent and critic.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (receiving the content of the observation channel, as specified by `obsInfo`) and one output layer (returning the scalar value).

Define the network as an array of layer objects, and get the dimension of the observation space from the environment specification objects. Name the network input layer `criticNetInput`.

```
CriticNet = [
    featureInputLayer(prod(obsInfo.Dimension));
    fullyConnectedLayer(10);
    reluLayer;
    fullyConnectedLayer(10);
    reluLayer;
    fullyConnectedLayer(1)];
```

Convert the network to a `dlnetwork` object.

```
CriticNet = dlnetwork(CriticNet);
```

To display the network main characteristics, use `summary`.

```
summary(CriticNet)
```

```
   Initialized: true

   Number of learnables: 171

   Inputs:
      1   'input'   4 features
```

Create the critic using CriticNet and the environment specification object. Set the observation name to observation, which is the name of the criticNetwork input layer.

```
critic = rlValueFunction(CriticNet,obsInfo)
```

```
critic = 
  rlValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

Check your critic using getValue to return the value of a random observation, given the current network weights.

```
v = getValue(critic,{rand(obsInfo.Dimension)})
```

```
v = single
    -0.3229
```

AC agents use a parametrized stochastic policy, which for discrete action spaces is implemented by a discrete categorical actor.

This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by obsInfo) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by actInfo.

You can obtain the number of actions from the actInfo specification. Name the network output actorNetOutput.

```
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(10);
    reluLayer;
    fullyConnectedLayer(10);
    reluLayer;
    fullyConnectedLayer(numel(actInfo.Elements)) ];
```

Convert the network to a dlnetwork object.

```
actorNet = dlnetwork(actorNet);
```

To display the network main characteristics, use summary.

```
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 182

   Inputs:
      1   'input'   4 features
```

Create the actor using rlDiscreteCategoricalActor together with the observation and action specifications.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo)
```

```
actor = 
  rlDiscreteCategoricalActor with properties:

       Distribution: [1x1 rl.distribution.rlDiscreteGaussianDistribution]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
          UseDevice: "cpu"
```

To check your actor, use `getAction` to return a random action from a given observation, using the current network weights.

```
a = getAction(actor,{rand(obsInfo.Dimension)})
```

```
a = 1x1 cell array
    {[-10]}
```

To return the probability distribution of the possible actions as a function of a random observation and given the current network weights, use `evaluate`.

```
prb = evaluate(actor,{rand(obsInfo.Dimension)})
```

```
prb = 1x1 cell array
    {2x1 single}
```

```
prb{1}
```

```
ans = 2x1 single column vector

    0.5917
    0.4083
```

Specify the optimization options for the actor and the critic using `rlOptimizerOptions`. These options control the learning of the network parameters. For both networks, set the gradient threshold to 1. For this example, set the learning rate to 0.01. For the actor network, set the learning rate to 0.05.

```
criticOpts = rlOptimizerOptions( ...
    LearnRate=1e-2,...
    GradientThreshold=1);

actorOpts = rlOptimizerOptions( ...
    LearnRate=5e-2,...
    GradientThreshold=1);
```

Specify agent options, including the objects previously created for both actor and critic.

```
agentOpts = rlACAgentOptions(...
    NumStepsToLookAhead=32,...
    DiscountFactor=0.99,...
    CriticOptimizerOptions=criticOpts,...
    ActorOptimizerOptions=actorOpts);
```

Create an AC agent using the actor, the critic and the agent options object.

```
agent = rlACAgent(actor,critic,agentOpts)
```

```
agent = 
  rlACAgent with properties:

            AgentOptions: [1x1 rl.option.rlACAgentOptions]
    UseExplorationPolicy: 1
         ObservationInfo: [1x1 rl.util.rlNumericSpec]
              ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

To check your agent, use `getAction` to return a random action from a given observation, using the current actor and critic network weights.

```
act = getAction(agent,{rand(obsInfo.Dimension)})
```

```
act = 1x1 cell array
    {[-10]}
```

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

For additional examples showing how to create actors and critics for different agent types, see “Train DDPG Agent to Control Double Integrator System” and “Train DQN Agent to Balance Cart-Pole System”.

Create Value Function Critic from Table

Create a finite set observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment with a discrete observation space). For this example, define the observation space as a finite set consisting of four possible values 1, 3, 5 and 7.

```
obsInfo = rlFiniteSetSpec([1 3 5 7]);
```

A value-function critic takes the current observation as input and returns a single scalar value as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

Since both observation and action spaces are discrete and low-dimensional, use a table to model the value function within the critic. `rlTable` creates a value table object from the observation and action specifications objects.

```
vTable = rlTable(obsInfo);
```

The table is a column vector in which each entry stores the value of the corresponding observation, under the given policy. You can access the table using the `Table` property of the `vTable` object. The initial value of each element is zero.

```
vTable.Table
```

```
ans = 4×1

     0
     0
     0
     0
```

You can also initialize the table to any value, in this case, an array containing all the integers from 1 to 4.

```
vTable.Table = reshape(1:4,4,1)
```

```
vTable = 
  rlTable with properties:

    Table: [4x1 double]
```

Create the critic using the table and the observation specification object.

```
critic = rlValueFunction(vTable,obsInfo)
```

```
critic = 
  rlValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlFiniteSetSpec]
          UseDevice: "cpu"
```

To check your critic, use the getValue function to return the value of a given observation, using the current table entries.

```
v = getValue(critic,{7})
```

```
v = 4
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given observation specification object. Examples of agents that can work with discrete observation spaces, and use a value function critic, are rlACAgent, rlPGAgent, rlPPOAgent. rlTRPOAgent does not support actors or critics that use tables.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Value Function Critic from Custom Basis Function

Create an observation specification object (or alternatively use getObservationInfo to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a custom basis function. Create a custom function that returns a vector of three elements, given an observation as input.

```
myBasisFcn = @(myobs) [
    myobs(2)^2;
    myobs(3)+exp(myobs(1));
    abs(myobs(4)) ]
```

```
myBasisFcn = function_handle with value:
    @(myobs)[myobs(2)^2;myobs(3)+exp(myobs(1));abs(myobs(4))]
```

The output of the critic is the scalar `W'*myBasisFcn(myobs)`, which represents the estimated value of the observation under the given policy. Here `W` is a weight column vector which must have the same size as the custom basis function output. The elements of `W` are the learnable parameters.

Define an initial parameter vector.

```
W0 = [3;5;2];
```

Create the critic. The first argument is a two-element cell containing both the handle to the custom function and the initial weight vector. The second argument is the observation specification object.

```
critic = rlValueFunction({myBasisFcn,W0},obsInfo)
```

```
critic = 
  rlValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

To check your critic, use the `getValue` function to return the value of a given observation, using the current parameter vector.

```
v = getValue(critic,{[2 4 6 8]'})
```

```
v = 130.9453
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given observation specification object. Examples of agents that can work with continuous observation spaces, and use a value function critic, are `rlACAgent`, `rlPGAgent`, `rlPPOAgent`. `rlTRPOAgent` does not support actors or critics that use custom basis functions.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Value Function Critic from Recurrent Neural Network

Create an environment and obtain observation and action information.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
```

A value-function critic takes the current observation as input and returns a single scalar value as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a recurrent neural network with one input layer (receiving the content of the observation channel, as specified by obsInfo) and one output layer (returning the scalar value).

Define the network as an array of layer objects. To create a recurrent network, use a sequenceInputLayer as the input layer (with size equal to the number of dimensions of the observation channel) and include at least one lstmLayer.

```
myNet = [
    sequenceInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(8)
    reluLayer
    lstmLayer(8)
    fullyConnectedLayer(1) ];
```

Convert the network to a dlnetwork object.

```
dlNet = dlnetwork(myNet);
```

Display a summary of network characteristics.

```
summary(dlNet)
```

```
   Initialized: true

   Number of learnables: 593

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create a value function representation object for the critic.

```
critic = rlValueFunction(dlNet,obsInfo)
```

```
critic = 
  rlValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
          UseDevice: "cpu"
```

To check your critic, use the getValue function to return the value of a random observation, using the current network weights.

```
v = getValue(critic,{rand(obsInfo.Dimension)})
```

```
v = single
    0.0017
```

You can use getState and setState to extract and set the current state of the recurrent neural network in the critic.

```
getState(critic)
```

```
ans=2×1 cell array
    {8x1 single}
    {8x1 single}
```

```
critic = setState(critic, ...
    {-0.01*single(rand(8,1)), ...
      0.01*single(rand(8,1))});
```

To evaluate the critic using sequential observations, use the sequence length (time) dimension. For example, obtain actions for 5 independent sequences each one consisting of 9 sequential observations.

```
[value,state] = getValue(critic, ...
    {rand([obsInfo.Dimension 5 9])});
```

Display the value corresponding to the seventh element of the observation sequence in the fourth sequence.

```
value(1,4,7)
```

```
ans = single
    0.0766
```

Display the updated state of the recurrent neural network.

```
state
```

```
state=2×1 cell array
    {8x5 single}
    {8x5 single}
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given observation specification object. Examples of agents that can work with continuous observation spaces, and use a value function critic, are `rlACAgent`, `rlPGAgent`, `rlPPOAgent`. `rlTRPOAgent` does not support actors or critics that use recurrent networks.

For more information on input and output format for recurrent neural networks, see the Algorithms section of `lstmLayer`. For more information on creating approximator objects such as critics and critics, see “Create Policies and Value Functions”.

Create Mixed Observation Space Value Function Critic from Custom Basis Function

Create a finite-set observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as two channels where the first one is a single observation labeled 7, 5, 3, or 1, and the second one is a vector over a continuous three-dimensional space.

```
obsInfo = [rlFiniteSetSpec([7 5 3 1]) rlNumericSpec([3 1])];
```

A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a custom basis function. Create a custom function that returns a vector of four elements, given the content of the two observation channels as input.

```
myBasisFcn = @(obsA,obsB) [ obsA(1) + norm(obsB);
                            obsA(1) - norm(obsB);
                            obsA(1)^2 + obsB(3);
                            obsA(1)^2 - obsB(3)];
```

The output of the critic is the scalar `W'*myBasisFcn(myobs)`, which represents the estimated value of the observation under the given policy. Here `W` is a weight column vector which must have the same size as the custom basis function output. The elements of `W` are the learnable parameters.

Define an initial parameter vector.

```
W0 = ones(4,1);
```

Create the critic. The first argument is a two-element cell containing both the handle to the custom function and the initial weight vector. The second argument is the observation specification object.

```
critic = rlValueFunction({myBasisFcn,W0},obsInfo)
```

```
critic = 
  rlValueFunction with properties:

    ObservationInfo: [2x1 rl.util.RLDataSpec]
          UseDevice: "cpu"
```

To check your critic, use the `getValue` function to return the value of a given observation, using the current parameter vector.

```
v = getValue(critic,{5,[0.1 0.1 0.1]'})
```

```
v = 60
```

Note that the critic does not enforce the set constraint for the discrete set element.

```
v = getValue(critic,{-3,[0.1 0.1 0.1]'})
```

```
v = 12
```

You can now use the critic (along with an actor) to create an agent for the environment described by the given observation specification object. Examples of agents that can work with mixed observation spaces, and use a value function critic, are `rlACAgent`, `rlPGAgent`, `rlPPOAgent`. `rlTRPOAgent` does not support actors or critics that use custom basis functions.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Version History

Introduced in R2022a

See Also

Functions
getValue | getModel | evaluate | getObservationInfo | getActionInfo

Objects
rlNumericSpec | rlFiniteSetSpec | rlQValueFunction | rlVectorQValueFunction | rlTable | rlPGAgent | rlACAgent | rlPPOAgent | rlTRPOAgent

Topics
"Train PG Agent to Balance Cart-Pole System"
"Train AC Agent to Balance Cart-Pole System"
"Train PPO Agent for a Lander Vehicle"
"Load Predefined Control System Environments"
"Create Policies and Value Functions"
"Reinforcement Learning Agents"

rlValueRepresentation

(Not recommended) Value function critic representation for reinforcement learning agents

Note `rlValueRepresentation` is not recommended. Use `rlValueFunction` instead. For more information, see “rlValueRepresentation is not recommended”.

Description

This object implements a value function approximator to be used as a critic within a reinforcement learning agent. A value function is a function that maps an observation to a scalar value. The output represents the expected total long-term reward when the agent starts from the given observation and takes the best possible action. Value function critics therefore only need observations (but not actions) as inputs. After you create an `rlValueRepresentation` critic, use it to create an agent relying on a value function critic, such as an `rlACAgent`, `rlPGAgent`, or `rlPPOAgent`. For an example of this workflow, see “Create Actor and Critic Representations” on page 3-479. For more information on creating representations, see “Create Policies and Value Functions”.

Creation

Syntax

```
critic = rlValueRepresentation(net,observationInfo,'Observation',obsName)
critic = rlValueRepresentation(tab,observationInfo)
critic = rlValueRepresentation({basisFcn,W0},observationInfo)
critic = rlValueRepresentation( ___ ,options)
```

Description

`critic = rlValueRepresentation(net,observationInfo,'Observation',obsName)` creates the value function based `critic` from the deep neural network `net`. This syntax sets the ObservationInfo property of `critic` to the input `observationInfo`. `obsName` must contain the names of the input layers of `net`.

`critic = rlValueRepresentation(tab,observationInfo)` creates the value function based `critic` with a *discrete observation space*, from the value table `tab`, which is an `rlTable` object containing a column array with as many elements as the possible observations. This syntax sets the ObservationInfo property of `critic` to the input `observationInfo`.

`critic = rlValueRepresentation({basisFcn,W0},observationInfo)` creates the value function based `critic` using a custom basis function as underlying approximator. The first input argument is a two-elements cell in which the first element contains the handle `basisFcn` to a custom basis function, and the second element contains the initial weight vector `W0`. This syntax sets the ObservationInfo property of `critic` to the input `observationInfo`.

`critic = rlValueRepresentation( ___ ,options)` creates the value function based `critic` using the additional option set `options`, which is an `rlRepresentationOptions` object. This syntax sets the Options property of `critic` to the `options` input argument. You can use this syntax with any of the previous input-argument combinations.

Input Arguments

`net` — Deep neural network

array of `Layer` objects | `layerGraph` object | `DAGNetwork` object | `SeriesNetwork` object | `dlNetwork` object

Deep neural network used as the underlying approximator within the critic, specified as one of the following:

- Array of `Layer` objects
- `layerGraph` object
- `DAGNetwork` object
- `SeriesNetwork` object
- `dlnetwork` object

The network input layers must be in the same order and with the same data type and dimensions as the signals defined in ObservationInfo. Also, the names of these input layers must match the observation names listed in `obsName`.

`rlValueRepresentation` objects support recurrent deep neural networks.

For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

`obsName` — Observation names

string | character vector | cell array of character vectors

Observation names, specified as a cell array of strings or character vectors. The observation names must be the names of the input layers in `net`. These network layers must be in the same order and with the same data type and dimensions as the signals defined in ObservationInfo.

Example: `{'my_obs'}`

`tab` — Value table

`rlTable` object

Value table, specified as an `rlTable` object containing a column vector with length equal to the number of observations. The element `i` is the expected cumulative long-term reward when the agent starts from the given observation `s` and takes the best possible action. The elements of this vector are the learnable parameters of the representation.

`basisFcn` — Custom basis function

function handle

Custom basis function, specified as a function handle to a user-defined function. The user defined function can either be an anonymous function or a function on the MATLAB path. The output of the critic is `c = W'*B`, where `W` is a weight vector and `B` is the column vector returned by the custom basis function. `c` is the expected cumulative long term reward when the agent starts from the given observation and takes the best possible action. The learnable parameters of this representation are the elements of `W`.

When creating a value function critic representation, your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here `obs1` to `obsN` are observations in the same order and with the same data type and dimensions as the signals defined in ObservationInfo.

Example: `@(obs1,obs2,obs3) [obs3(1)*obs1(1)^2; abs(obs2(5)+obs1(2))]`

`W0` — Initial value of the basis function weights

column vector

Initial value of the basis function weights, `W`, specified as a column vector having the same length as the vector returned by the basis function.

Properties

`Options` — Representation options

`rlRepresentationOptions` object

Representation options, specified as an `rlRepresentationOptions` object. Available options include the optimizer used for training and the learning rate.

`ObservationInfo` — Observation specifications

`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. These objects define properties such as the dimensions, data types, and names of the observation signals.

`rlValueRepresentation` sets the `ObservationInfo` property of `critic` to the input `observationInfo`.

You can extract `ObservationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually.

Object Functions

rlACAgent	Actor-critic (AC) reinforcement learning agent
rlPGAgent	Policy gradient (PG) reinforcement learning agent
rlPPOAgent	Proximal policy optimization (PPO) reinforcement learning agent
getValue	Obtain estimated value from a critic given environment observations and actions

Examples

Create Value Function Critic from Deep Neural Network

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing 4 doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create a deep neural network to approximate the value function within the critic. The input of the network (here called `myobs`) must accept a four-element vector (the observation vector defined by `obsInfo`), and the output must be a scalar (the value, representing the expected cumulative long-term reward when the agent starts from the given observation).

```
net = [featureInputLayer(4, 'Normalization','none','Name','myobs')
       fullyConnectedLayer(1,'Name','value')];
```

Create the critic using the network, observation specification object, and name of the network input layer.

```
critic = rlValueRepresentation(net,obsInfo,'Observation',{'myobs'})
```

```
critic = 
  rlValueRepresentation with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your critic, use the `getValue` function to return the value of a random observation, using the current network weights.

```
v = getValue(critic,{rand(4,1)})
```

```
v = single
    0.7904
```

You can now use the critic (along with an actor) to create an agent relying on a value function critic (such as `rlACAgent` or `rlPGAgent`).

Create Actor and Critic Representations

Create an actor representation and a critic representation that you can use to define a reinforcement learning agent such as an Actor Critic (AC) agent.

For this example, create actor and critic representations for an agent that can be trained against the cart-pole environment described in “Train AC Agent to Balance Cart-Pole System”. First, create the environment. Then, extract the observation and action specifications from the environment. You need these specifications to define the agent and critic representations.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

For a state-value-function critic such as those used for AC or PG agents, the inputs are the observations and the output should be a scalar value, the state value. For this example, create the critic representation using a deep neural network with one output, and with observation signals corresponding to `x`, `xdot`, `theta`, and `thetadot` as described in “Train AC Agent to Balance Cart-Pole System”. You can obtain the number of observations from the `obsInfo` specification. Name the network layer input `'observation'`.

```
numObservation = obsInfo.Dimension(1);
criticNetwork = [
    featureInputLayer(numObservation,'Normalization','none','Name','observation')
    fullyConnectedLayer(1,'Name','CriticFC')];
```

Specify options for the critic representation using `rlRepresentationOptions`. These options control the learning of the critic network parameters. For this example, set the learning rate to 0.05 and the gradient threshold to 1.

```
repOpts = rlRepresentationOptions('LearnRate',5e-2,'GradientThreshold',1);
```

Create the critic representation using the specified neural network and options. Also, specify the action and observation information for the critic. Set the observation name to `'observation'`, which is the of the `criticNetwork` input layer.

```
critic = rlValueRepresentation(criticNetwork,obsInfo,'Observation',{'observation'},repOpts)
```

```
critic = 
  rlValueRepresentation with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

Similarly, create a network for the actor. An AC agent decides which action to take given observations using an actor representation. For an actor, the inputs are the observations, and the output depends on whether the action space is discrete or continuous. For the actor of this example, there are two possible discrete actions, –10 or 10. To create the actor, use a deep neural network with the same observation input as the critic, that can output these two values. You can obtain the number of actions from the `actInfo` specification. Name the output `'action'`.

```
numAction = numel(actInfo.Elements);
actorNetwork = [
    featureInputLayer(numObservation,'Normalization','none','Name','observation')
    fullyConnectedLayer(numAction,'Name','action')];
```

Create the actor representation using the observation name and specification and the same representation options.

```
actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'observation'},repOpts)
```

```
actor = 
  rlStochasticActorRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

Create an AC agent using the actor and critic representations.

```
agentOpts = rlACAgentOptions(...
    'NumStepsToLookAhead',32,...
    'DiscountFactor',0.99);
agent = rlACAgent(actor,critic,agentOpts)
```

```
agent = 
  rlACAgent with properties:
    AgentOptions: [1x1 rl.option.rlACAgentOptions]
```

For additional examples showing how to create actor and critic representations for different agent types, see:

- “Train DDPG Agent to Control Double Integrator System”
- “Train DQN Agent to Balance Cart-Pole System”

Create Value Function Critic from Table

Create a finite set observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment with a discrete observation space). For this example, define the observation space as a finite set consisting of 4 possible values.

```
obsInfo = rlFiniteSetSpec([1 3 5 7]);
```

Create a table to approximate the value function within the critic.

```
vTable = rlTable(obsInfo);
```

The table is a column vector in which each entry stores the expected cumulative long-term reward for each possible observation as defined by `obsInfo`. You can access the table using the `Table` property of the `vTable` object. The initial value of each element is zero.

```
vTable.Table
```

```
ans = 4×1

     0
     0
     0
     0
```

You can also initialize the table to any value, in this case, an array containing all the integers from `1` to `4`.

```
vTable.Table = reshape(1:4,4,1)
```

```
vTable = 
  rlTable with properties:

    Table: [4x1 double]
```

Create the critic using the table and the observation specification object.

```
critic = rlValueRepresentation(vTable,obsInfo)
```

```
critic = 
  rlValueRepresentation with properties:

    ObservationInfo: [1x1 rl.util.rlFiniteSetSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your critic, use the `getValue` function to return the value of a given observation, using the current table entries.

```
v = getValue(critic,{7})
```

```
v = 4
```

You can now use the critic (along with an actor) to create an agent relying on a value function critic (such as `rlACAgent` or `rlPGAgent agent`).

Create Value Function Critic from Custom Basis Function

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that a single observation is a column vector containing 4 doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create a custom basis function to approximate the value function within the critic. The custom basis function must return a column vector. Each vector element must be a function of the observations defined by `obsInfo`.

```
myBasisFcn = @(myobs) [myobs(2)^2; myobs(3)+exp(myobs(1)); abs(myobs(4))]
```

```
myBasisFcn = function_handle with value:
    @(myobs)[myobs(2)^2;myobs(3)+exp(myobs(1));abs(myobs(4))]
```

The output of the critic is the scalar `W'*myBasisFcn(myobs)`, where `W` is a weight column vector which must have the same size of the custom basis function output. This output is the expected cumulative long term reward when the agent starts from the given observation and takes the best possible action. The elements of W are the learnable parameters.

Define an initial parameter vector.

```
W0 = [3;5;2];
```

Create the critic. The first argument is a two-element cell containing both the handle to the custom function and the initial weight vector. The second argument is the observation specification object.

```
critic = rlValueRepresentation({myBasisFcn,W0},obsInfo)
```

```
critic = 
  rlValueRepresentation with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

To check your critic, use the `getValue` function to return the value of a given observation, using the current parameter vector.

```
v = getValue(critic,{[2 4 6 8]'})
```

```
v = 
  1x1 dlarray

  130.9453
```

You can now use the critic (along with an with an actor) to create an agent relying on a value function critic (such as `rlACAgent` or `rlPGAgent`).

Create Value Function Critic from Recurrent Neural Network

Create an environment and obtain observation and action information.

```
env = rlPredefinedEnv('CartPole-Discrete');
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
numObs = obsInfo.Dimension(1);
numDiscreteAct = numel(actInfo.Elements);
```

Create a recurrent deep neural network for the critic. To create a recurrent neural network, use a `sequenceInputLayer` as the input layer and include at least one `lstmLayer`.

```
criticNetwork = [
    sequenceInputLayer(numObs,'Normalization','none','Name','state')
    fullyConnectedLayer(8, 'Name','fc')
    reluLayer('Name','relu')
    lstmLayer(8,'OutputMode','sequence','Name','lstm')
    fullyConnectedLayer(1,'Name','output')];
```

Create a value function representation object for the critic.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-2,'GradientThreshold',1);
critic = rlValueRepresentation(criticNetwork,obsInfo,...
    'Observation','state',criticOptions);
```

Version History

Introduced in R2020a

R2022a: `rlValueRepresentation` is not recommended

Not recommended starting in R2022a

`rlValueRepresentation` is not recommended. Use `rlValueFunction` instead.

The following table shows some typical uses of `rlValueRepresentation`, and how to update your code with `rlValueFunction` instead. Each table entry is related to different approximator objects, the first one uses a neural network, the second one uses a table, the third one uses a basis function.

`rlValueRepresentation`: Not Recommended	**`rlValueFunction`: Recommended**
`myCritic = rlValueRepresentation(net,obsInfo,'Observation',obsNames)`, where the neural network `net` has observations as inputs and a single scalar output.	`myCritic = rlValueFunction(net,obsInfo,'ObservationInputNames',obsNames)`. Use this syntax to create a state value function object for a critic that does not require action input.
`rep = rlValueRepresentation(tab,obsInfo)` where the table `tab` contains a column vector with as many elements as the number of possible observations.	`rep = rlValueFunction(tab,obsInfo)`. Use this syntax to create a value function object for a critic that does not require action input.
`rep = rlValueRepresentation({basisFcn,W0},obsInfo)`, where the basis function has only observations as inputs and `W0` is a column vector.	`rep = rlValueFunction({basisFcn,W0},obsInfo)`. Use this syntax to create a value function object for a critic that does not require action input.

See Also

Functions
`getActionInfo` | `getObservationInfo`

Objects
`rlValueFunction`

Topics
"Create Policies and Value Functions"
"Reinforcement Learning Agents"

rlVectorQValueFunction

Vector Q-value function approximator for reinforcement learning agents

Description

This object implements a vector Q-value function approximator that you can use as a critic with a discrete action space for a reinforcement learning agent. A vector Q-value function is a mapping from an environment observation to a vector in which each element represents the expected discounted cumulative long-term reward when an agent starts from the state corresponding to the given observation and executes the action corresponding to the element number (and follows a given policy afterwards). A vector Q-value function critic therefore needs only the observation as input. After you create an `rlVectorQValueFunction` critic, use it to create an agent such as `rlQAgent`, `rlDQNAgent`, `rlSARSAAgent`. For more information on creating representations, see “Create Policies and Value Functions”.

Creation

Syntax

```
critic = rlVectorQValueFunction(net,observationInfo,actionInfo)
critic = rlVectorQValueFunction(net,observationInfo,ObservationInputNames=
netObsNames)
```

```
critic = rlVectorQValueFunction({basisFcn,W0},observationInfo,actionInfo)
```

```
critic = rlVectorQValueFunction( ___ ,UseDevice=useDevice)
```

Description

`critic = rlVectorQValueFunction(net,observationInfo,actionInfo)` creates the *multi-output* Q-value function `critic` with a *discrete action space*. Here, `net` is the deep neural network used as an approximation model, and must have only the observations as input and a single output layer having as many elements as the number of possible discrete actions. The network input layers are automatically associated with the environment observation channels according to the dimension specifications in `observationInfo`. This function sets the `ObservationInfo` and `ActionInfo` properties of `critic` to the `observationInfo` and `actionInfo` input arguments, respectively.

`critic = rlVectorQValueFunction(net,observationInfo,ObservationInputNames=netObsNames)` specifies the names of the network input layers to be associated with the environment observation channels. The function assigns, in sequential order, each environment observation channel specified in `observationInfo` to the layer specified by the corresponding name in the string array `netObsNames`. Therefore, the network input layers, ordered as the names in `netObsNames`, must have the same data type and dimensions as the observation channels, as ordered in `observationInfo`.

`critic = rlVectorQValueFunction({basisFcn,W0},observationInfo,actionInfo)` creates the *multi-output* Q-value function `critic` with a *discrete action space* using a custom basis function as underlying approximation model. The first input argument is a two-element cell array whose first element is the handle `basisFcn` to a custom basis function and whose second element is the initial weight matrix `W0`. Here the basis function must have only the observations as inputs, and `W0` must have as many columns as the number of possible actions. The function sets the ObservationInfo and ActionInfo properties of `critic` to the input arguments `observationInfo` and `actionInfo`, respectively.

`critic = rlVectorQValueFunction( ___ ,UseDevice=useDevice)` specifies the device used to perform computations for the `critic` object, and sets the `UseDevice` property of `critic` to the `useDevice` input argument. You can use this syntax with any of the previous input-argument combinations.

Input Arguments

net — Deep neural network

array of `Layer` objects | `layerGraph` object | `DAGNetwork` object | `SeriesNetwork` object | `dlNetwork` object (preferred)

Deep neural network used as the underlying approximator within the critic,specified as one of the following:

- Array of `Layer` objects
- `layerGraph` object
- `DAGNetwork` object
- `SeriesNetwork` object
- `dlnetwork` object

Note Among the different network representation options, `dlnetwork` is preferred, since it has built-in validation checks and supports automatic differentiation. If you pass another network object as an input argument, it is internally converted to a `dlnetwork` object. However, best practice is to convert other representations to `dlnetwork` explicitly *before* using it to create a critic or an actor for a reinforcement learning agent. You can do so using `dlnet=dlnetwork(net)`, where `net` is any Deep Learning Toolbox neural network object. The resulting `dlnet` is the `dlnetwork` object that you use for your critic or actor. This practice allows a greater level of insight and control for cases in which the conversion is not straightforward and might require additional specifications.

The network must have as many input layers as the number of environment observation channels (with each input layer receiving input from an observation channel), and a single output layer with as many elements as the number of possible discrete actions.

`rlQValueFunction` objects support recurrent deep neural networks.

The learnable parameters of the critic are the weights of the deep neural network. For a list of deep neural network layers, see “List of Deep Learning Layers”. For more information on creating deep neural networks for reinforcement learning, see “Create Policies and Value Functions”.

netObsNames — Network input layers names corresponding to the environment observation channels

string array | cell array of character vectors

Network input layers names corresponding to the environment observation channels, specified as a string array or a cell array of character vectors. When you use this argument after

`'ObservationInputNames'`, the function assigns, in sequential order, each environment observation channel specified in `observationInfo` to each network input layer specified by the corresponding name in the string array `netObsNames`. Therefore, the network input layers, ordered as the names in `netObsNames`, must have the same data type and dimensions as the observation channels, as ordered in `observationInfo`.

Example: `{"NetInput1_airspeed","NetInput2_altitude"}`

`basisFcn` — Custom basis function
function handle

Custom basis function, specified as a function handle to a user-defined MATLAB function. The user defined function can either be an anonymous function or a function on the MATLAB path. The output of the critic is the vector `c = W'*B`, where `W` is a matrix containing the learnable parameters, and `B` is the column vector returned by the custom basis function. Each element of `a` approximates the value of executing the corresponding action from the observed state.

Your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here, `obs1` to `obsN` are inputs in the same order and with the same data type and dimensions as the channels defined in `observationInfo`.

Example: `@(obs1,obs2) [act(2)*obs1(1)^2; abs(obs2(5))]`

`W0` — Initial value of basis function weights
matrix

Initial value of the basis function weights `W`, specified as a matrix having as many rows as the length of the basis function output vector and as many columns as the number of possible actions.

Properties

`ObservationInfo` — Observation specifications
`rlFiniteSetSpec` object | `rlNumericSpec` object | array

Observation specifications, specified as an `rlFiniteSetSpec` or `rlNumericSpec` object or an array containing a mix of such objects. Each element in the array defines the properties of an environment observation channel, such as its dimensions, data type, and name.

`rlVectorQValueFucntion` sets the `ObservationInfo` property of `critic` to the input argument `observationInfo`.

You can extract `ObservationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually.

`ActionInfo` — Action specifications
`rlFiniteSetSpec` object

Action specifications, specified as an `rlFiniteSetSpec` object. This object defines the properties of the environment action channel, such as its dimensions, data type, and name.

Note Only one action channel is allowed.

rlVectorQValueFucntion sets the ActionInfo property of critic to the input actionInfo.

You can extract ActionInfo from an existing environment or agent using getActionInfo. You can also construct the specifications manually.

UseDevice — Computation device used for training and simulation

"cpu" (default) | "gpu"

Computation device used to perform operations such as gradient computation, parameter update and prediction during training and simulation, specified as either "cpu" or "gpu".

The "gpu" option requires both Parallel Computing Toolbox software and a CUDA enabled NVIDIA GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use gpuDevice (Parallel Computing Toolbox) to query or select a local GPU device to be used with MATLAB.

Note Training or simulating an agent on a GPU involves device-specific numerical round-off errors. These errors can produce different results compared to performing the same operations using a CPU.

To speed up training by using parallel processing over multiple cores, you do not need to use this argument. Instead, when training your agent, use an rlTrainingOptions object in which the UseParallel option is set to true. For more information about training using multicore processors and GPUs for training, see “Train Agents Using Parallel Computing and GPUs”.

Example: "gpu"

Object Functions

rlDQNAgent	Deep Q-network (DQN) reinforcement learning agent
rlQAgent	Q-learning reinforcement learning agent
rlSARSAAgent	SARSA reinforcement learning agent
getValue	Obtain estimated value from a critic given environment observations and actions
getMaxQValue	Obtain maximum estimated value over all possible actions from a Q-value function critic with discrete action space, given environment observations
evaluate	Evaluate function approximator object given observation (or observation-action) input data
gradient	Evaluate gradient of function approximator object given observation and action input data
accelerate	Option to accelerate computation of gradient for approximator object based on neural network
getLearnableParameters	Obtain learnable parameter values from agent, function approximator, or policy object
setLearnableParameters	Set learnable parameter values of agent, function approximator, or policy object
setModel	Set function approximation model for actor or critic
getModel	Get function approximator model from actor or critic

Examples

Create Multi-Output Q-Value Function Critic from Deep Neural Network

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create a finite set action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment with a discrete action space). For this example, define the action space as a finite set consisting of three possible actions (labeled 7, 5, and 3).

```
actInfo = rlFiniteSetSpec([7 5 3]);
```

A vector Q-value function takes only the observation as input and returns as output a single vector with as many elements as the number of possible actions. The value of each output element represents the expected discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy afterwards.

To model the parametrized vector Q-value function within the critic, use a neural network with one input layer (receiving the content of the observation channel, as specified by `obsInfo`) and one output layer (returning the vector of values for all the possible actions, as defined by `actInfo`).

Define the network as an array of layer objects, and get the dimension of the observation space and the number of possible actions from the environment specification objects.

```
net = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert the network to a `dlnetwork` object, and display the number of weights.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 403

   Inputs:
      1   'input'   4 features
```

Create the critic using the network, as well as the observation and action specification objects.

```
critic = rlVectorQValueFunction(net,obsInfo,actInfo)
```

```
critic = 
  rlVectorQValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
```

```
          UseDevice: "cpu"
```

To check your critic, use `getValue` to return the values of a random observation, using the current network weights. There is one value for each of the three possible actions.

```
v = getValue(critic,{rand(obsInfo.Dimension)})
```

```
v = 3x1 single column vector

    0.0761
   -0.5906
    0.2072
```

You can now use the critic to create an agent for the environment described by the given specification objects. Examples of agents that can work with a discrete action space, a continuous observation space, and use a vector Q-value function critic, are `rlQAgent`, `rlDQNAgent`, and `rlSARSAAgent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Multi-Output Q-Value Function Critic from Deep Neural Network Specifying Layer Names

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as a continuous four-dimensional space, so that there is a single observation channel that carries a column vector containing four doubles.

```
obsInfo = rlNumericSpec([4 1]);
```

Create a finite set action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment with a discrete action space). For this example, define the action space as a finite set consisting of three possible values (labeled 7, 5, and 3).

```
actInfo = rlFiniteSetSpec([7 5 3]);
```

A vector Q-value function takes only the observation as input and returns as output a single vector with as many elements as the number of possible actions. The value of each output element represents the expected discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy afterwards.

To model the parametrized vector Q-value function within the critic, use a neural network with one input layer (receiving the content of the observation channel, as specified by `obsInfo`) and one output layer (returning the vector of values for all the possible actions, as defined by `actInfo`).

Define the network as an array of layer objects, and get the dimension of the observation space and the number of possible actions from the environment specification objects. Name the network input `netObsIn` (so you can later explicitly associate it with the observation input channel).

```
net = [
    featureInputLayer(obsInfo.Dimension(1),Name="netObsIn")
    fullyConnectedLayer(32)
    tanhLayer
    fullyConnectedLayer(numel(actInfo.Elements)) ];
```

Convert the network to a `dlnetwork` object and display the number of its learnable parameters.

```
net = dlnetwork(net)
```

```
net = 
  dlnetwork with properties:

         Layers: [4x1 nnet.cnn.layer.Layer]
    Connections: [3x2 table]
     Learnables: [4x3 table]
          State: [0x3 table]
     InputNames: {'netObsIn'}
    OutputNames: {'fc_2'}
    Initialized: 1

  View summary with summary.
```

```
summary(net)
```

```
   Initialized: true

   Number of learnables: 259

   Inputs:
      1   'netObsIn'   4 features
```

Create the critic using the network, the observations specification object, and the name of the network input layer. The specified network input layer, `netObsIn`, is associated with the environment observation, and therefore must have the same data type and dimension as the observation channel specified in `obsInfo`.

```
critic = rlVectorQValueFunction(net, ...
    obsInfo,actInfo, ...
    ObservationInputNames="netObsIn")
```

```
critic = 
  rlVectorQValueFunction with properties:

    ObservationInfo: [1x1 rl.util.rlNumericSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
          UseDevice: "cpu"
```

To check your critic, use the `getValue` function to return the values of a random observation, using the current network weights. The function returns one value for each of the three possible actions.

```
v = getValue(critic,{rand(obsInfo.Dimension)})
```

```
v = 3x1 single column vector

    0.0435
    0.1906
    0.7386
```

You can now use the critic to create an agent for the environment described by the given specification objects. Examples of agents that can work with a discrete action space, a continuous observation space, and use a vector Q-value function critic, are `rlQAgent`, `rlDQNAgent`, and `rlSARSAAgent`.

For more information on creating approximator objects such as actors and critics, see "Create Policies and Value Functions".

Create Multi-Output Q-Value Function Critic from Custom Basis Function

Create an observation specification object (or alternatively use `getObservationInfo` to extract the specification object from an environment). For this example, define the observation space as consisting of two channels, the first carrying a two-by-two continuous matrix and the second carrying scalar that can assume only two values, 0 and 1.

```
obsInfo = [rlNumericSpec([2 2])
           rlFiniteSetSpec([0 1])];
```

Create a finite set action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment with a discrete action space). For this example, define the action space as a finite set consisting of three possible vectors, [1 2], [3 4], and [5 6].

```
actInfo = rlFiniteSetSpec({[1 2],[3 4],[5 6]});
```

A vector Q-value function takes only the observation as input and returns as output a single vector with as many elements as the number of possible actions.

To model the parametrized vector Q-value function within the critic, use a custom basis function with two inputs (which receive the content of the environment observation channels, as specified by `obsInfo`).

Create a function that returns a vector of four elements, given an observation as input.

```
myBasisFcn = @(obsA,obsB) [obsA(1,1)+obsB(1)^2;
                           obsA(2,1)-obsB(1)^2;
                           obsA(1,2)^2+obsB(1);
                           obsA(2,2)^2-obsB(1)];
```

The output of the critic is the vector `c = W'*myBasisFcn(obsA,obsB)`, where `W` is a weight matrix which must have as many rows as the length of the basis function output and as many columns as the number of possible actions.

Each element of `c` represents the expected cumulative long term reward when an agent starts from the given observation and takes the action corresponding to the position of the considered element (and follows the policy afterwards). The elements of W are the learnable parameters.

Define an initial parameter matrix.

```
W0 = rand(4,3);
```

Create the critic. The first argument is a two-element cell containing both the handle to the custom function and the initial parameter matrix. The second and third arguments are, respectively, the observation and action specification objects.

```
critic = rlVectorQValueFunction({myBasisFcn,W0},obsInfo,actInfo)
critic = 
  rlVectorQValueFunction with properties:

    ObservationInfo: [2x1 rl.util.RLDataSpec]
         ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
          UseDevice: "cpu"
```

To check your critic, use the getValue function to return the values of a random observation, using the current parameter matrix. The function returns one value for each of the three possible actions.

```
v = getValue(critic,{rand(2,2),0})
```

```
v = 3×1

    1.3192
    0.8420
    1.5053
```

Note that the critic does not enforce the set constraint for the discrete set elements.

```
v = getValue(critic,{rand(2,2),-1})
```

```
v = 3×1

    2.7890
    1.8375
    3.0855
```

You can now use the critic to create an agent for the environment described by the given specification objects. Examples of agents that can work with a discrete action space, a mixed observation space, and use a vector Q-value function critic, are rlQAgent, rlDQNAgent, and rlSARSAAgent.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Create Multi-Output Q-Value Function Critic from Recurrent Neural Network

Create an environment and obtain observation and action specification objects.

```
env = rlPredefinedEnv("CartPole-Discrete");
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

A vector Q-value function takes only the observation as input and returns as output a single vector with as many elements as the number of possible actions. The value of each output element represents the expected discounted cumulative long-term reward for taking the action from the state corresponding to the current observation, and following the policy afterwards.

To model the parametrized vector Q-value function within the critic, use a recurrent neural network with one input layers (receiving the content of the observation channel, as specified by obsInfo) and one output layer (returning the vector of values for all the possible actions, as defined by actInfo).

Define the network as an array of layer objects, and get the dimension of the observation space and the number of possible actions from the environment specification objects. To create a recurrent network, use a `sequenceInputLayer` as the input layer (with size equal to the number of dimensions of the observation channel) and include at least one `lstmLayer`.

```
net = [
    sequenceInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(50)
    reluLayer
    lstmLayer(20)
    fullyConnectedLayer(20)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    ];
```

Convert the network to a `dlnetwork` object, and display the number of weights.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 6.3k

   Inputs:
      1   'sequenceinput'   Sequence input with 4 dimensions
```

Create the critic using the network, as well as the observation and action specification objects.

```
critic = rlVectorQValueFunction(net, ...
    obsInfo,actInfo);
```

To check your critic, use the `getValue` function to return the value of a random observation and action, using the current network weights.

```
v = getValue(critic,{rand(obsInfo.Dimension)})
```

```
v = 2x1 single column vector

    0.0136
    0.0067
```

You can use `getState` and `setState` to extract and set the current state of the recurrent neural network in the critic.

```
getState(critic)
```

```
ans=2×1 cell array
    {20x1 single}
    {20x1 single}
```

```
critic = setState(critic, ...
    {-0.01*single(rand(20,1)), ...
      0.01*single(rand(20,1))});
```

To evaluate the critic using sequential observations, use the sequence length (time) dimension. For example, obtain actions for 5 independent sequences each one consisting of 9 sequential observations.

```
[value,state] = getValue(critic, ...
    {rand([obsInfo.Dimension 5 9])});
```

Display the value corresponding to the seventh element of the observation sequence in the fourth sequence.

```
value(1,4,7)
```

```
ans = single
    0.0394
```

Display the updated state of the recurrent neural network.

```
state
```

```
state=2×1 cell array
    {20x5 single}
    {20x5 single}
```

You can now use the critic to create an agent for the environment described by the given specification objects. Examples of agents that can work with a discrete action space, a continuous observation space, and use a vector Q-value function critic, are `rlQAgent`, `rlDQNAgent`, and `rlSARSAAgent`.

For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions”.

Version History

Introduced in R2022a

See Also

Functions

`getValue` | `getMaxQValue` | `getModel` | `evaluate` | `getObservationInfo` | `getActionInfo`

Objects

`rlNumericSpec` | `rlFiniteSetSpec` | `rlValueFunction` | `rlQValueFunction` | `rlQAgent` | `rlSARSAAgent` | `rlDQNAgent`

Topics

“Train DQN Agent to Balance Cart-Pole System”
“Train DQN Agent to Swing Up and Balance Pendulum”
“Load Predefined Control System Environments”
“Create Policies and Value Functions”
“Reinforcement Learning Agents”

scalingLayer

Scaling layer for actor or critic network

Description

A scaling layer linearly scales and biases an input array `U`, giving an output `Y = Scale.*U + Bias`. You can incorporate this layer into the deep neural networks you define for actors or critics in reinforcement learning agents. This layer is useful for scaling and shifting the outputs of nonlinear layers, such as `tanhLayer` and sigmoid.

For instance, a `tanhLayer` gives bounded output that falls between –1 and 1. If your actor network output has different bounds (as defined in the actor specification), you can include a `ScalingLayer` as an output to scale and shift the actor network output appropriately.

The parameters of a `ScalingLayer` object are not learnable.

Creation

Syntax

```
sLayer = scalingLayer
sLayer = scalingLayer(Name,Value)
```

Description

`sLayer = scalingLayer` creates a scaling layer with default property values.

`sLayer = scalingLayer(Name,Value)` sets properties on page 3-496 using name-value pairs. For example, `scalingLayer('Scale',0.5)` creates a scaling layer that scales its input by 0.5. Enclose each property name in quotes.

Properties

Name — Name of layer
`'scaling'` (default) | character vector

Name of layer, specified as a character vector. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with this layer and `Name` is set to `''`, then the software automatically assigns a name to the layer at training time.

`Description` — Description of layer
`'Scaling layer'` (default) | character vector

This property is read-only.

Description of layer, specified as a character vector. When you create the scaling layer, you can use this property to give it a description that helps you identify its purpose.

`Scale` — Element-wise scale on input

1 (default) | scalar | array

Element-wise scale on the input to the scaling layer, specified as one of the following:

- Scalar — Specify the same scale factor for all elements of the input array.
- Array expandable to the same dimensions as the input array — Specify different scale factors for each element of the input array.

Note `Scale` and `Bias` must have the same size if they are both arrays.

The scaling layer takes an input `U` and generates the output `Y = Scale.*U + Bias`.

`Bias` — Element-wise bias on input

0 (default) | scalar | array

Element-wise bias on the input to the scaling layer, specified as one of the following:

- Scalar — Specify the same bias for all elements of the input array.
- Array expandable to the same dimensions as the input array — Specify a different bias for each element of the input array.

Note `Scale` and `Bias` must have the same size if they are both arrays.

The scaling layer takes an input `U` and generates the output `Y = Scale.*U + Bias`.

Examples

Create Scaling Layer

Create a scaling layer that converts an input array `U` to the output array `Y = 0.1.*U - 0.4`.

```
sLayer = scalingLayer(Scale=0.1,Bias=-0.4)
```

```
sLayer = 
  ScalingLayer with properties:

     Name: 'scaling'
    Scale: 0.1000
     Bias: -0.4000

   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

Confirm that the scaling layer scales and offsets an input array as expected.

```
predict(sLayer,[10,20,30])

ans = 1×3

    0.6000    1.6000    2.6000
```

You can incorporate `sLayer` into an actor network or critic network for reinforcement learning.

Specify Different Scale and Bias for Each Input Element

Assume that the layer preceding the `scalingLayer` is a `tanhLayer` with three outputs aligned along the first dimension, and that you want to apply a different scaling factor and bias to each out using a `scalingLayer`.

```
scale = [2.5 0.4 10]';
bias = [5 0 -50]';
```

Create the `scalingLayer` object.

```
sLayer = scalingLayer(Scale=scale,Bias=bias);
```

Confirm that the scaling layer applies the correct scale and bias values to an array with the expected dimensions.

```
testData = [10 10 10]';
predict(sLayer,testData)

ans = 3×1

    30
     4
    50
```

Version History

Introduced in R2019a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Functions

`quadraticLayer` | `softplusLayer`

Topics

"Train DDPG Agent to Swing Up and Balance Pendulum"
"Create Policies and Value Functions"

SimulinkEnvWithAgent

Reinforcement learning environment with a dynamic model implemented in Simulink

Description

The `SimulinkEnvWithAgent` object represents a reinforcement learning environment that uses a dynamic model implemented in Simulink. The environment object acts as an interface such that when you call `sim` or `train`, these functions in turn call the Simulink model to generate experiences for the agents.

Creation

To create a `SimulinkEnvWithAgent` object, use one of the following functions.

- `rlSimulinkEnv` — Create an environment using a Simulink model with at least one RL Agent block.
- `createIntegratedEnv` — Use a reference model as a reinforcement learning environment.
- `rlPredefinedEnv` — Create a predefined reinforcement learning environment.

Properties

`Model` — Simulink model name
string | character vector

Simulink model name, specified as a string or character vector. The specified model must contain one or more RL Agent blocks.

`AgentBlock` — Agent block paths
string | string array

Agent block paths, specified as a string or string array.

If `Model` contains a single RL Agent block for training, then `AgentBlock` is a string containing the block path.

If `Model` contains multiple RL Agent blocks for training, then `AgentBlock` is a string array, where each element contains the path of one agent block.

`Model` can contain RL Agent blocks whose path is not included in `AgentBlock`. Such agent blocks behave as part of the environment and select actions based on their current policies. When you call `sim` or `train`, the experiences of these agents are not returned and their policies are not updated.

The agent blocks can be inside of a model reference. For more information on configuring an agent block for reinforcement learning, see RL Agent.

`ResetFcn` — Reset behavior for environment
function handle | anonymous function handle

Reset behavior for the environment, specified as a function handle or anonymous function handle. The function must have a single `Simulink.SimulationInput` input argument and a single `Simulink.SimulationInput` output argument.

The reset function sets the initial state of the Simulink environment. For example, you can create a reset function that randomizes certain block states such that each training episode begins from different initial conditions.

If you have an existing reset function `myResetFunction` on the MATLAB path, set `ResetFcn` using a handle to the function.

```
env.ResetFcn = @(in)myResetFunction(in);
```

If your reset behavior is simple, you can implement it using an anonymous function handle. For example, the following code sets the variable `x0` to a random value.

```
env.ResetFcn = @(in) setVariable(in,'x0',rand());
```

The `sim` function calls the reset function to reset the environment at the start of each simulation, and the `train` function calls it at the start of each training episode.

`UseFastRestart` — Option to toggle fast restart

`"on"` (default) | `"off"`

Option to toggle fast restart, specified as either `"on"` or `"off"`. Fast restart allows you to perform iterative simulations without compiling a model or terminating the simulation each time.

For more information on fast restart, see “How Fast Restart Improves Iterative Simulations” (Simulink).

Object Functions

train	Train reinforcement learning agents within a specified environment
sim	Simulate trained reinforcement learning agents within specified environment
getObservationInfo	Obtain observation data specifications from reinforcement learning environment, agent, or experience buffer
getActionInfo	Obtain action data specifications from reinforcement learning environment, agent, or experience buffer

Examples

Create Simulink Environment Using Agent in Workspace

Create a Simulink environment using the trained agent and corresponding Simulink model from the “Create Simulink Environment and Train Agent” example.

Load the agent in the MATLAB® workspace.

```
load rlWaterTankDDPGAgent
```

Create an environment for the `rlwatertank` model, which contains an RL Agent block. Since the agent used by the block is already in the workspace, you do not need to pass the observation and action specifications to create the environment.

```
env = rlSimulinkEnv("rlwatertank","rlwatertank/RL Agent")
env = 
SimulinkEnvWithAgent with properties:

           Model : rlwatertank
      AgentBlock : rlwatertank/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

Validate the environment by performing a short simulation for two sample times.

```
validateEnvironment(env)
```

You can now train and simulate the agent within the environment by using `train` and `sim`, respectively.

Create Reinforcement Learning Environment for Simulink Model

For this example, consider the `rlSimplePendulumModel` Simulink model. The model is a simple frictionless pendulum that initially hangs in a downward position.

Open the model.

```
mdl = "rlSimplePendulumModel";
open_system(mdl)
```

Create `rlNumericSpec` and `rlFiniteSetSpec` objects for the observation and action information, respectively.

The observation is a vector containing three signals: the sine, cosine, and time derivative of the angle.

```
obsInfo = rlNumericSpec([3 1])
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [3 1]
       DataType: "double"
```

The action is a scalar expressing the torque and can be one of three possible values, `-2` Nm, `0` Nm and `2` Nm.

```
actInfo = rlFiniteSetSpec([-2 0 2])
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [3x1 double]
           Name: [0x0 string]
```

```
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

You can use dot notation to assign property values for the `rlNumericSpec` and `rlFiniteSetSpec` objects.

```
obsInfo.Name = "observations";
actInfo.Name = "torque";
```

Assign the agent block path information, and create the reinforcement learning environment for the Simulink model using the information extracted in the previous steps.

```
agentBlk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,agentBlk,obsInfo,actInfo)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

You can also include a reset function using dot notation. For this example, randomly initialize `theta0` in the model workspace.

```
env.ResetFcn = @(in) setVariable(in,"theta0",randn,"Workspace",mdl)
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : @(in)setVariable(in,"theta0",randn,"Workspace",mdl)
  UseFastRestart : on
```

Create Simulink Environment for Multiple Agents

Create an environment for the Simulink model from the example “Train Multiple Agents to Perform Collaborative Task”.

Load the agents in the MATLAB workspace.

```
load rlCollaborativeTaskAgents
```

Create an environment for the `rlCollaborativeTask` model, which has two agent blocks. Since the agents used by the two blocks (`agentA` and `agentB`) are already in the workspace, you do not need to pass their observation and action specifications to create the environment.

```
env = rlSimulinkEnv( ...
    "rlCollaborativeTask", ...
    ["rlCollaborativeTask/Agent A","rlCollaborativeTask/Agent B"])
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlCollaborativeTask
      AgentBlock : [
                     rlCollaborativeTask/Agent A
                     rlCollaborativeTask/Agent B
                   ]
        ResetFcn : []
  UseFastRestart : on
```

You can now simulate or train the agents within the environment using `sim` or `train`, respectively.

Create Continuous Simple Pendulum Model Environment

Use the predefined `"SimplePendulumModel-Continuous"` keyword to create a continuous simple pendulum model reinforcement learning environment.

```
env = rlPredefinedEnv("SimplePendulumModel-Continuous")
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

Create Environment from Simulink Model

This example shows how to use `createIntegratedEnv` to create an environment object starting from a Simulink model that implements the system with which the agent. Such a system is often referred to as *plant, open-loop* system, or *reference* system, while the whole (integrated) system including the agent is often referred to as the *closed-loop* system.

For this example, use the flying robot model described in “Train DDPG Agent to Control Flying Robot” as the reference (open-loop) system.

Open the flying robot model.

```
open_system('rlFlyingRobotEnv')
```

Initialize the state variables and sample time.

```
% initial model state variables
theta0 = 0;
x0 = -15;
y0 = 0;

% sample time
Ts = 0.4;
```

Create the Simulink model myIntegratedEnv containing the flying robot model connected in a closed loop to the agent block. The function also returns the reinforcement learning environment object env to be used for training.

```
env = createIntegratedEnv('rlFlyingRobotEnv','myIntegratedEnv')
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : myIntegratedEnv
      AgentBlock : myIntegratedEnv/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

The function can also return the block path to the RL Agent block in the new integrated model, as well as the observation and action specifications for the reference model.

```
[~,agentBlk,observationInfo,actionInfo] = createIntegratedEnv( ...
    'rlFlyingRobotEnv','myIntegratedEnv')
```

```
agentBlk = 
'myIntegratedEnv/RL Agent'

observationInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "observation"
    Description: [0x0 string]
      Dimension: [7 1]
       DataType: "double"


actionInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "action"
    Description: [0x0 string]
      Dimension: [2 1]
       DataType: "double"
```

Returning the block path and specifications is useful in cases in which you need to modify descriptions, limits, or names in observationInfo and actionInfo. After modifying the specifications, you can then create an environment from the integrated model IntegratedEnv using the rlSimulinkEnv function.

Version History

Introduced in R2019a

See Also

Functions
`rlSimulinkEnv` | `rlPredefinedEnv` | `train` | `sim`

Objects
`rlNumericSpec` | `rlFiniteSetSpec`

Blocks
RL Agent

Topics
“Create Simulink Reinforcement Learning Environments”

softplusLayer

Softplus layer for actor or critic network

Description

A softplus layer applies the softplus activation function $Y = \log(1 + e^X)$, which ensures that the output is always positive. This activation function is a smooth continuous version of `reluLayer`. You can incorporate this layer into the deep neural networks you define for actors in reinforcement learning agents. This layer is useful for creating continuous Gaussian policy deep neural networks, for which the standard deviation output must be positive.

Creation

Syntax

```
sLayer = softplusLayer
sLayer = softplusLayer(Name,Value)
```

Description

`sLayer = softplusLayer` creates a softplus layer with default property values.

`sLayer = softplusLayer(Name,Value)` sets properties on page 3-507 using name-value pairs. For example, `softplusLayer('Name','softlayer')` creates a softplus layer and assigns the name `'softlayer'`.

Properties

Name — Name of layer
`'softplus'` (default) | character vector

Name of layer, specified as a character vector. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with this layer and `Name` is set to `''`, then the software automatically assigns a name to the layer at training time.

`Description` — Description of layer
`'Softplus layer'` (default) | character vector

This property is read-only.

Description of layer, specified as a character vector. When you create the softplus layer, you can use this property to give it a description that helps you identify its purpose.

Examples

Create Softplus Layer

Create s softplus layer object.

```
sLayer = softplusLayer;
```

You can specify the name of the softplus layer. For example, if the softplus layer represents the standard deviation of a Gaussian policy deep neural network, you can specify an appropriate name.

```
sLayer = softplusLayer(Name="stddev")
```

```
sLayer = 
  SoftplusLayer with properties:

    Name: 'stddev'

   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

You can incorporate `sLayer` into an actor network for reinforcement learning.

Version History

Introduced in R2020a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Functions

`quadraticLayer` | `scalingLayer`

Topics

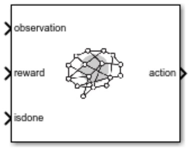
“Create Policies and Value Functions”

4

Blocks

RL Agent

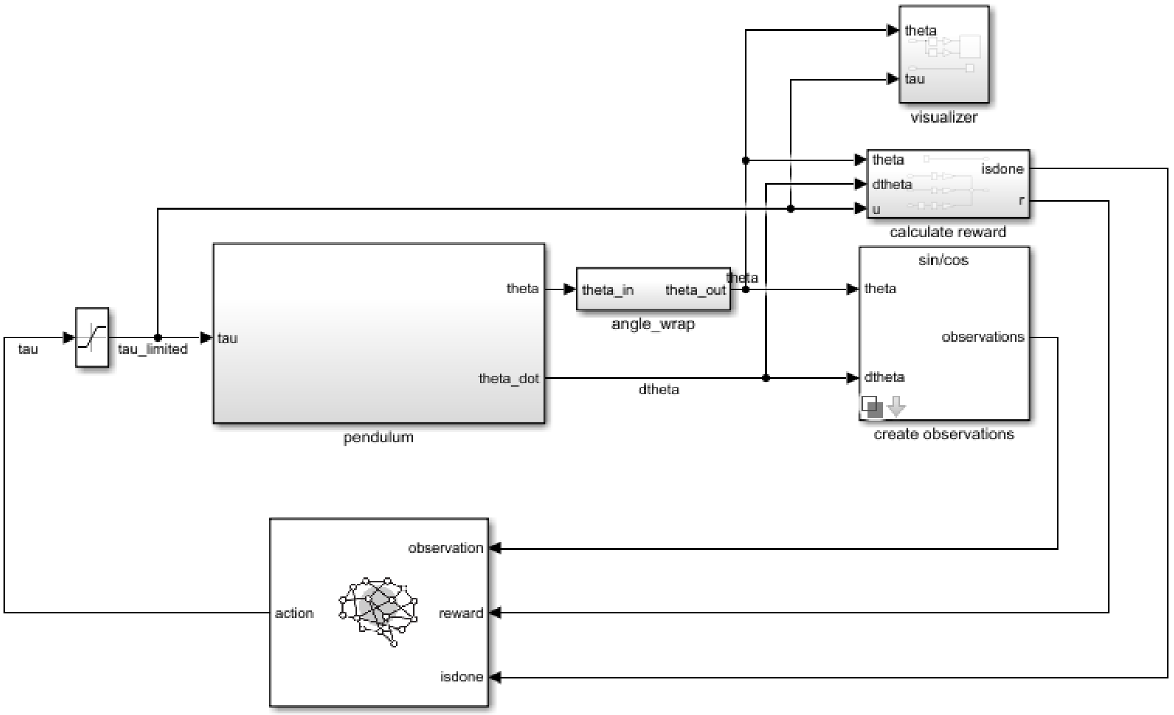
Reinforcement learning agent



Libraries:
Reinforcement Learning Toolbox

Description

Use the RL Agent block to simulate and train a reinforcement learning agent in Simulink. You associate the block with an agent stored in the MATLAB workspace or a data dictionary, such as an `rlACAgent` or `rlDDPGAgent` object. You connect the block so that it receives an observation and a computed reward. For instance, consider the following block diagram of the `rlSimplePendulumModel` model.



The **observation** input port of the RL Agent block receives a signal that is derived from the instantaneous angle and angular velocity of the pendulum. The **reward** port receives a reward calculated from the same two values and the applied action. You configure the observations and reward computations that are appropriate to your system.

The block uses the agent to generate an action based on the observation and reward you provide. Connect the **action** output port to the appropriate input for your system. For instance, in the `rlSimplePendulumModel`, the **action** output port is a torque applied to the pendulum system. For more information about this model, see "Train DQN Agent to Swing Up and Balance Pendulum".

To train a reinforcement learning agent in Simulink, you generate an environment from the Simulink model. You then create and configure the agent for training against that environment. For more information, see "Create Simulink Reinforcement Learning Environments". When you call `train` using the environment, `train` simulates the model and updates the agent associated with the block.

Ports

Input

observation — Environment observations
scalar | vector | nonvirtual bus

This port receives observation signals from the environment. Observation signals represent measurements or other instantaneous system data. If you have multiple observations, you can use a Mux block to combine them into a vector signal. To use a nonvirtual bus signal, use `bus2RLSpec`.

reward — Reward from environment
scalar

This port receives the reward signal, which you compute based on the observation data. The reward signal is used during agent training to maximize the expectation of the long-term reward.

isdone — Flag to terminate episode simulation
logical

Use this signal to specify conditions under which to terminate a training episode. You must configure logic appropriate to your system to determine the conditions for episode termination. One application is to terminate an episode that is clearly going well or going poorly. For instance, you can terminate an episode if the agent reaches its goal or goes irrecoverably far from its goal.

external action — External action signal
scalar | vector

Use this signal to provide an external action to the block. This signal can be a control action from a human expert, which can be used for safe or imitation learning applications. When the value of the **use external action** signal is `1`, the passes the **external action** signal to the environment through the **action** block output. The block also uses the external action to update the agent policy based on the resulting observations and rewards.

Dependencies

To enable this port, select the **External action inputs** parameter.

last action — Last action applied to environment signal
scalar | vector

For some applications, the action applied to the environment can differ from the action output by the RL Agent block. For example, the Simulink model can contain a saturation block on the action output signal.

In such cases, to improve learning results, you can enable this input port and connect the actual action signal that is applied to the environment.

Note The **last action** port should be used only with off-policy agents, otherwise training can produce unexpected results.

Dependencies

To enable this port, select the **Last action input** parameter.

use external action — Use external action signal
`0` | `1`

Use this signal to pass the **external action** signal to the environment.

When the value of the **use external action** signal is `1` the block passes the **external action** signal to the environment. The block also uses the external action to update the agent policy.

When the value of the **use external action** signal is `0` the block does not pass the **external action** signal to the environment and does not update the policy using the external action. Instead, the action from the block uses the action from the agent policy.

Dependencies

To enable this port, select the **External action inputs** parameter.

Output

action — Agent action
scalar | vector | nonvirtual bus

Action computed by the agent based on the observation and reward inputs. Connect this port to the inputs of your system. To use a nonvirtual bus signal, use `bus2RLSpec`.

Note Continuous action-space agents such as `rlACAgent`, `rlPGAgent`, or `rlPPOAgent` (the ones using an `rlContinuousGaussianActor` object), do not enforce constraints set by the action specification. In these cases, you must enforce action space constraints within the environment.

cumulative reward — Total reward
scalar | vector

Cumulative sum of the reward signal during simulation. Observe or log this signal to track how the cumulative reward evolves over time.

Dependencies

To enable this port, select the **Cumulative reward output** parameter.

Parameters

Agent object — Agent to train
`agentObj` (default) | agent object

Enter the name of an agent object stored in the MATLAB workspace or a data dictionary, such as an `rlACAgent` or `rlDDPGAgent` object. For information about agent objects, see “Reinforcement Learning Agents”.

If the RL Agent block is within a conditionally executed subsystem, such as a Triggered Subsystem or a Function-Call Subsystem, you must specify the sample time of the agent object as `-1` so that the block can inherit the sample time of its parent subsystem.

Programmatic Use
Block Parameter: `Agent`
Type: string, character vector
Default: `"agentObj"`

Generate greedy policy block — Generate greedy policy block controller
button

Generate a Policy block that implements a greedy policy for the agent specified in **Agent object** by calling the `generatePolicyBlock` block function. To generate a greedy policy, the block sets the `UseExplorationPolicy` property of the agent to `false` before generating the policy block..

The generated block is added to a new Simulink model and the policy data is saved in a MAT-file in the current working folder.

External action inputs — Add input ports for external action
`off` (default) | `on`

Enable the **external action** and **use external action** block input ports by selecting this parameter.

Programmatic Use
Block Parameter: `ExternalActionAsInput`
Type: string, character vector
Values: `"off"` | `"on"`
Default: `"off"`

Last action input — Add input ports for last action applied to environment
`off` (default) | `on`

Enable the **last action** block input port by selecting this parameter.

Programmatic Use
Block Parameter: `ProvideLastAction`
Type: string, character vector
Values: `"off"` | `"on"`
Default: `"off"`

Cumulative reward output — Add cumulative reward output port
`off` (default) | `on`

Enable the **cumulative reward** block output by selecting this parameter.

Programmatic Use
Block Parameter: `ProvideCumRwd`
Type: string, character vector
Values: `"off"` | `"on"`
Default: `"off"`

Use strict observation data types — Enforce strict data types for observations
`off` (default) | `on`

Select this parameter to enforce the observation data types. In this case, if the data type of the signal connected to the **observation** input port does not match the data type in the `ObservationInfo` property of the agent, the block attempts to cast the signal to the correct data type. If casting the data type is not possible, the block generates an error.

Enforcing strict data types:

- Lets you validate that the block is getting the correct data types.
- Allows other blocks to inherit their data type from the **observation** port.

Programmatic Use
Block Parameter: `UseStrictObservationDataTypes`
Type: string, character vector
Values: `"off"` | `"on"`
Default: `"off"`

Version History

Introduced in R2019a

See Also

Functions

`bus2RLSpec` | `createIntegratedEnv`

Blocks

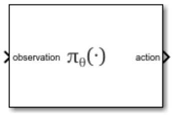
Policy

Topics

“Create Simulink Environment and Train Agent”
“Create Simulink Reinforcement Learning Environments”

Policy

Reinforcement learning policy



Libraries:
Reinforcement Learning Toolbox

Description

Use the Policy block to simulate a reinforcement learning policy in Simulink and to generate code (using Simulink Coder) for deployment purposes. This block takes an observation as input and outputs an action. You associate the block with a MAT-file that contains the information needed to fully characterize the policy, and which can be generated by `generatePolicyFunction` or `generatePolicyBlock`.

Ports

Input

observation — Environment observations
scalar | vector | nonvirtual bus

This port receives observation signals from the environment. Observation signals represent measurements or other instantaneous system data. If you have multiple observations, you can use a Mux block to combine them into a vector signal. To use a nonvirtual bus signal, use `bus2RLSpec`.

Output

action — Policy action
scalar | vector | nonvirtual bus

Action computed by the policy based on the observation input. Connect this port to the inputs of your system. To use a nonvirtual bus signal, use `bus2RLSpec`.

Note Policy blocks generated from a continuous action-space `rlStochasticActorPolicy` object or a continuous action-space `rlACAgent`, `rlPGAgent`, or `rlPPOAgent` object, do not enforce the constraints set by the action specification. In these cases, you must enforce action space constraints within the environment.

Parameters

Policy block data MAT file — Policy block data MAT file
`blockAgentData.mat` (default) | file name

Enter the name of the MAT-file containing the information needed to fully characterize the policy. This file is generated by `generatePolicyFunction` or `generatePolicyBlock`. When you generate the block using `generatePolicyBlock` and specify a non-default `dataFileName` argument, then the generated block has this parameter set to the specified file name, so that the block is associated with the generated data file.

To use a Policy block within a conditionally executed subsystem, such as a Triggered Subsystem or a Function-Call Subsystem, you must generate its data file from an agent or policy object which has its `SampleTime` property set to `-1`. Doing so allows the block to inherit the sample time of its parent subsystem.

Programmatic Use
Block Parameter: `MATFile`
Type: string, character vector
Default: `"blockAgentData.mat"`

Tips

- When using Embedded Coder® to generate parallel code, enabling the **Generate parallel for loops** optimization parameter improves the performance when the data being processed is large in size. However, if the network and the data is small, the overhead of initializing the threads for parallelization significantly reduces the performance. In this case, disable **Generate parallel for loops**. See Generate parallel for-loops (Embedded Coder) and `coder.MexCodeConfig` for more information.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Functions

`bus2RLSpec` | `createIntegratedEnv` | `generatePolicyFunction` | `generatePolicyBlock`

Objects

`rlMaxQPolicy` | `rlEpsilonGreedyPolicy` | `rlAdditiveNoisePolicy` |
`rlDeterministicActorPolicy` | `rlStochasticActorPolicy`

Blocks

RL Agent

Topics

“Create Simulink Environment and Train Agent”
“Create Policies and Value Functions”
“Create Simulink Reinforcement Learning Environments”