Reinforcement Learning Toolbox™

Reference

MATLAB®

R2020a

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

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)

Contents

Functions

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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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.

See Also

Blocks
RL Agent

Functions
`createIntegratedEnv` | `rlFiniteSetSpec` | `rlNumericSpec` | `rlSimulinkEnv`

Topics
“Create Simulink Environments for Reinforcement Learning”

Introduced in R2019a

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']`.

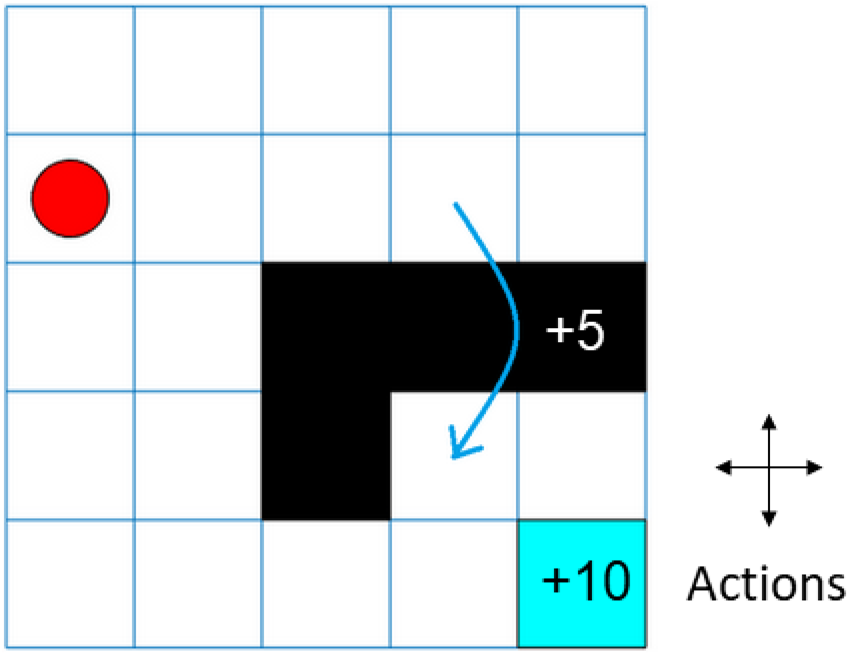
`GW = createGridWorld(m,n,moves)` creates a grid world `GW` of size `m`-by-`n` with actions specified by `moves`.

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]
```

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;
```

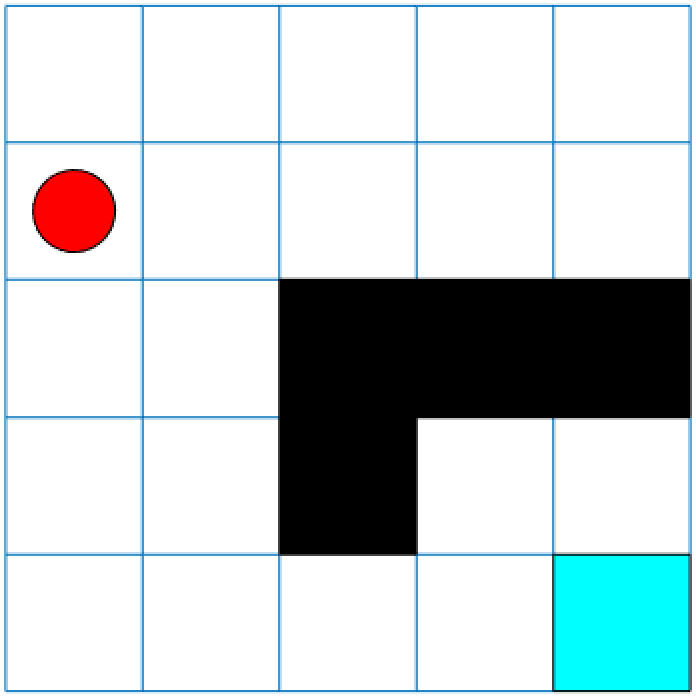
Now, use `rlMDPEnv` to create a grid world environment using the `GridWorld` object `GW`.

```
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.

See Also
`rlMDPEnv` | `rlPredefinedEnv`

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

Introduced in R2019a

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 a Simulink Model

This example shows how to use `createIntegratedEnv` to create an environment object starting from a 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.

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 state variables and sample time.

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

% sample time
Ts = 0.4;
```

Create the Simulink model `IntegratedEnv` 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','IntegratedEnv')
```

```
env = 
  SimulinkEnvWithAgent with properties:

             Model: "IntegratedEnv"
        AgentBlock: "IntegratedEnv/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 data specifications for the reference model.

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

```
agentBlk = 
'IntegratedEnv/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"
```

This is useful in cases in which you need to modify descriptions, limits or names in `observationInfo` and `actionInfo` and later create an environment from the integrated model `IntegratedEnv`, using the function `rlSimulinkEnv`.

Create an Integrated Environment with Specified Port Names

This example shows how to call the function `createIntegratedEnv` with using Name and Value pairs to create an integrated (closed loop) Simulink environment and the corresponding environment object.

The first argument of the `createIntegratedEnv` function is the name of the *reference* Simulink model which contains the system that the agent needs to interact with. 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.

Open the open loop water tank model.

```
open_system('rlWatertankOpenloop.slx');
```

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

```
Ts=1;
```

Since 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`), use Name and Value pairs to specify the correct name when calling the function `createIntegratedEnv`.

```
env=createIntegratedEnv('rlWatertankOpenloop','IntegratedWatertank','ActionPortName','u','Observa
```

```
env = 
  SimulinkEnvWithAgent with properties:

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

This creates the new model `IntegratedWatertank` which contains the reference 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.

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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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

See Also

Blocks
RL Agent

Functions
`bus2RLSpec` | `rlFiniteSetSpec` | `rlNumericSpec` | `rlSimulinkEnv`

Topics
“Create Simulink Environments for Reinforcement Learning”

Introduced in R2019a

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.

See Also
`createGridWorld` | `rlMDPEnv`

Topics
"Train Reinforcement Learning Agent in MDP Environment"

Introduced in R2019a

generatePolicyFunction

Package: `rl.agent`

Create function that evaluates trained policy of reinforcement learning agent

Syntax

```
generatePolicyFunction(agent)
generatePolicyFunction(agent,Name,Value)
```

Description

`generatePolicyFunction(agent)` creates a function that evaluates the learned policy of the specified agent using the default function, policy, and data file names. After generating the policy evaluation function, you can:

- Generate code for the function 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.

`generatePolicyFunction(agent,Name,Value)` specifies the function, policy, and data file names 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: 29-Feb-2020 09:51:55
```

```
actionSet = [-10 10];
% Select action from sampled probabilities
probabilities = localEvaluate(observation1);
% Normalize the probabilities
p = probabilities(:)'/sum(probabilities);
% Determine which action to take
edges = min([0 cumsum(p)],1);
edges(end) = 1;
[~,actionIndex] = histc(rand(1,1),edges); %#ok<HISTC>
action1 = actionSet(actionIndex);
end
%% Local Functions
function probabilities = localEvaluate(observation1)
persistent policy
if isempty(policy)
    policy = coder.loadDeepLearningNetwork('agentData.mat','policy');
end
probabilities = predict(policy,observation1);
end
```

For a given observation, the policy function evaluates a probability for each potential action using the actor network. Then, the policy function randomly selects an action based on these probabilities.

Since the actor network for this PG agent has a single input layer and single output layer, you can generate code for this network using the Deep Learning Toolbox™ generation functionality. 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: 29-Feb-2020 09:44:06

actionSet = [1;2;3;4];
```

```
numActions = numel(actionSet);
q = zeros(1,numActions);
for i = 1:numActions
    q(i) = localEvaluate(observation1,actionSet(i));
end
[~,actionIndex] = max(q);
action1 = actionSet(actionIndex);
end
%% Local Functions
function q = localEvaluate(observation1,action)
persistent policy
if isempty(policy)
    s = coder.load('policyFile.mat','policy');
    policy = s.policy;
end
actionSet = [1;2;3;4];
observationSet = [1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18;19;20;21;22;23;24;25];
actionIndex = rl.codegen.getElementIndex(actionSet,action);
observationIndex = rl.codegen.getElementIndex(observationSet,observation1);
q = policy(observationIndex,actionIndex);
end
```

For a given observation, the policy function looks up the value function for each potential action using the Q table. Then, the policy function selects the action for which the value function is greatest.

You can generate code for this policy function using MATLAB® Coder™

For more information, see “Deploy Trained Reinforcement Learning Policies”

Input Arguments

`agent` — Trained reinforcement learning agent
reinforcement learning agent object

Trained reinforcement learning agent, specified as one of the following:

- `rlQAgent` object
- `rlSARSAAgent` object
- `rlDDPGAgent` object
- `rlTD3Agent` object
- `rlACAgent` object
- `rlPGAgent` object that estimates a baseline value function using a critic

Since Deep Learning Toolbox™ code generation and prediction functionality do not support deep neural networks with more than one input layer, `generatePolicyFunction` does not support the following agent configurations.

- DQN agent with deep neural network critic representations
- Any agent with deep neural network actor or critic representations with multiple observation input layers

To train your agent, use the `train` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FunctionName',"computeAction"`

`FunctionName` — Name of the generated function
`'evaluatePolicy'` (default) | string | character vector

Name of the generated function, specified as the name-value pair consisting of `'FunctionName'` and a string or character vector.

`PolicyName` — Name of the policy variable within the generated function
`'policy'` (default) | string | character vector

Name of the policy variable within the generated function, specified as the name-value pair consisting of `'PolicyName'` and a string or character vector.

`MATFileName` — Name of agent data file
`'agentData'` (default) | string | character vector

Name of the agent data file, specified as the name-value pair consisting of `'MATFileName'` and a string or character vector.

See Also
`train`

Topics
“Train Reinforcement Learning Agents”
“Reinforcement Learning Agents”
“Create Policy and Value Function Representations”
“Deploy Trained Reinforcement Learning Policies”

Introduced in R2019a

getAction

Obtain action from agent or actor representation given environment observations

Syntax

```
agentAction = getAction(agent,obs)

actorAction = getAction(actorRep,obs)
[actorAction,nextState] = getAction(actorRep,obs)
```

Description

Agent

`agentAction = getAction(agent,obs)` returns the action derived from the policy of a reinforcement learning agent given environment observations.

Actor Representation

`actorAction = getAction(actorRep,obs)` returns the action derived from policy representation `actorRep` given environment observations `obs`.

`[actorAction,nextState] = getAction(actorRep,obs)` returns the updated state of the actor representation when the actor uses a recurrent neural network as a function approximator.

Examples

Get Actions from Agent

Create an environment interface and obtain its observation and action specifications. For this environment load the predefined environment used for the discrete cart-pole system.

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

Create a critic representation.

```
statePath = [
    imageInputLayer([4 1 1],'Normalization','none','Name','state')
    fullyConnectedLayer(24, 'Name', 'CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(24,'Name','CriticStateFC2')];
actionPath = [
    imageInputLayer([1 1 1],'Normalization','none','Name','action')
    fullyConnectedLayer(24, 'Name', 'CriticActionFC1')];
commonPath = [
    additionLayer(2,'Name','add')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(1,'Name','output')];
criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork, actionPath);
criticNetwork = addLayers(criticNetwork, commonPath);
criticNetwork = connectLayers(criticNetwork,'CriticStateFC2','add/in1');
criticNetwork = connectLayers(criticNetwork,'CriticActionFC1','add/in2');
```

Create a representation for the critic.

```
criticOpts = rlRepresentationOptions('LearnRate',0.01,'GradientThreshold',1);
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation',{'state'},'Action',{'action'},criticOpts);
```

Specify agent options, and create a DQN agent using the environment and critic.

```
agentOpts = rlDQNAgentOptions(...
    'UseDoubleDQN',false, ...
    'TargetUpdateMethod',"periodic", ...
    'TargetUpdateFrequency',4, ...
    'ExperienceBufferLength',100000, ...
    'DiscountFactor',0.99, ...
    'MiniBatchSize',256);
agent = rlDQNAgent(critic,agentOpts);
```

Obtain a discrete action from the agent for a single observation. For this example, use a random observation array.

```
act = getAction(agent,{rand(4,1)})
```

```
act = 10
```

You can also obtain actions for a batch of observations. For example, obtain actions for a batch of 10 observations.

```
actBatch = getAction(agent,{rand(4,1,10)});
size(actBatch)
```

```
ans = 1×2

     1    10
```

`actBatch` contains one action for each observation in the batch, with each action being one of the possible discrete actions.

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]);
numObs = obsinfo.Dimension(1);
numAct = actinfo.Dimension(1);
```

Create a recurrent deep neural network for the actor.

```
net = [imageInputLayer([4 1 1],'Normalization','none','Name','state')
            fullyConnectedLayer(10,'Name','fc1')
            reluLayer('Name','relu1')
            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'});
```

Obtain an action from this actor for a random batch of 20 observations.

```
act = getAction(actor,{rand(4,1,10)})
```

```
act = 1×1 cell array
    {2×1×10 single}
```

`act` contains the two computed actions for all 10 observations in the batch.

Input Arguments

agent — Reinforcement learning agent
`rlQAgent` | `rlSARSAAgent` | `rlDQNAgent` | `rlPGAgent` | `rlDDPGAgent` | `rlTD3Agent` | `rlACAgent` | `rlPPOAgent`

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

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

actorRep — Actor representation
`rlDeterministicActorRepresentation` object | `rlStochasticActorRepresentation` object

Actor representation, specified as either an `rlDeterministicActorRepresentation` or `rlStochasticActorRepresentation` 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 `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`.

Output Arguments

`agentAction` — Action value from agent

array

Action value from agent, returned as 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 actor and critic in `agent` do not use recurrent neural networks, then $L_S = 1$.

Note When agents such as `rlACAgent`, `rlPGAgent`, or `rlPPOAgent` use an `rlStochasticActorRepresentation` actor with a continuous action space, the constraints set by the action specification are not enforced by the agent. In these cases, you must enforce action space constraints within the environment.

`actorAction` — Action value from actor representation

single-element cell array

Action value from actor representation, returned as a single-element cell array that contains an array of dimensions M_A-by-L_B-by-L_S, where:

- M_A corresponds to the dimensions of the action specification.
- L_B is the batch size.
- L_S is the sequence length for a recurrent neural network. If `actorRep` does not use a recurrent neural network, then $L_S = 1$.

Note `rlStochasticActorRepresentation` actors with continuous action spaces do not enforce constraints set by the action specification. In these cases, you must enforce action space constraints within the environment.

`nextState` — Actor representation updated state

cell array

Actor representation updated state, returned as a cell array. If `actorRep` 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:

```
valueRep = setState(actorRep,state);
```

See Also

`getMaxQValue` | `getValue`

Topics

“Custom Agents”
“Train Reinforcement Learning Policy Using Custom Training Loop”

Introduced in R2020a

getActionInfo

Obtain action data specifications from reinforcement learning environment or agent

Syntax

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

Description

`actInfo = getActionInfo(env)` extracts action information from reinforcement learning environment `env`.

`actInfo = getActionInfo(agent)` extracts action information from reinforcement learning agent `agent`.

Examples

Extract Action and Observation Information from Reinforcement Learning Environment

Extract action and observation information that you can use to create other environments or agents.

The reinforcement learning environment for this example is the simple longitudinal dynamics for ego car and lead car. The training goal is to make the ego car travel at a set velocity while maintaining a safe distance from lead car by controlling longitudinal acceleration (and braking). This example uses the same vehicle model as the “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox) example.

Open the model and create the reinforcement learning environment.

```
mdl = 'rlACCMdl';
open_system(mdl);
agentblk = [mdl '/RL Agent'];
% create the observation info
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 Info
actInfo = rlNumericSpec([1 1],'LowerLimit',-3,'UpperLimit',2);
actInfo.Name = 'acceleration';
% define environment
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 `SimulinkWithAgent` object with the above properties.

Extract the action and observation information from the reinforcement learning environment `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
`SimulinkEnvWithAgent` object

Reinforcement learning environment from which the action information has to be extracted, specified as a `SimulinkEnvWithAgent` object.

For more information on reinforcement learning environments, see “Create Simulink Environments for Reinforcement Learning”.

agent — Reinforcement learning agent
`rlQAgent` object | `rlSARSAAgent` object | `rlDQNAgent` object | `rlDDPGAgent` object | `rlPGAgent` object | `rlACAgent` object

Reinforcement learning agent from which the action information has to be extracted, specified as one of the following objects:

- `rlQAgent`
- `rlSARSAAgent`

- `rlDQNAgent`
- `rlDDPGAgent`
- `rlPGAgent`
- `rlACAgent`

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

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

See Also

`getObservationInfo` | `rlACAgent` | `rlDDPGAgent` | `rlDQNAgent` | `rlFiniteSetSpec` | `rlNumericSpec` | `rlPGAgent` | `rlQAgent` | `rlSARSAAgent`

Topics
“Create Simulink Environments for Reinforcement Learning”
“Reinforcement Learning Agents”

Introduced in R2019a

getActor

Package: `rl.agent`

Get actor representation from reinforcement learning agent

Syntax

```
actor = getActor(agent)
```

Description

`actor = getActor(agent)` returns the actor representation object for 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 representation from the agent.

```
actor = getActor(agent);
```

Obtain the learnable parameters from the actor.

```
params = getLearnableParameters(actor);
```

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.

```
agent = setActor(agent,actor);
```

Input Arguments

agent — Reinforcement learning agent
`rlDDPGAgent` object | `rlTD3Agent` object | `rlPGAgent` object | `rlACAgent` object | `rlPPOAgent` object

Reinforcement learning agent that contains an actor representation, specified as one of the following:

- `rlDDPGAgent` object
- `rlTD3Agent` object
- `rlACAgent` object
- `rlPGAgent` object
- `rlPPOAgent` object

Output Arguments

`actor` — Actor representation
`rlDeterministicActorRepresentation` object | `rlStochasticActorRepresentation` object

Actor representation object, specified as one of the following:

- `rlDeterministicActorRepresentation` object — Returned when `agent` is an `rlDDPGAgent` or `rlTD3Agent` object
- `rlStochasticActorRepresentation` object — Returned when `agent` is an `rlACAgent`, `rlPGAgent`, or `rlPPOAgent` object

See Also

`getCritic` | `getLearnableParameters` | `setActor` | `setCritic` | `setLearnableParameters`

Topics
"Create Policy and Value Function Representations"
"Import Policy and Value Function Representations"

Introduced in R2019a

getCritic

Package: `rl.agent`

Get critic representation from reinforcement learning agent

Syntax

```
critic = getCritic(agent)
```

Description

`critic = getCritic(agent)` returns the critic representation object for 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 representation from the agent.

```
critic = getCritic(agent);
```

Obtain the learnable parameters from the critic.

```
params = getLearnableParameters(critic);
```

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.

```
agent = setCritic(agent,critic);
```

Input Arguments

agent — Reinforcement learning agent
`rlQAgent` object | `rlSARSAAgent` object | `rlDQNAgent` object | `rlDDPGAgent` object | `rlTD3Agent` object | `rlPGAgent` object | `rlACAgent` object | `rlPPOAgent` object

Reinforcement learning agent that contains a critic representation, specified as one of the following:

- `rlQAgent` object
- `rlSARSAAgent` object
- `rlDQNAgent` object
- `rlDDPGAgent` object
- `rlTD3Agent` object
- `rlACAgent` object
- `rlPPOAgent` object
- `rlPGAgent` object that estimates a baseline value function using a critic

Output Arguments

`critic` — Critic representation
`rlValueRepresentation` object | `rlQValueRepresentation` object | two-element row vector of `rlQValueRepresentation` objects

Critic representation object, returned as one of the following:

- `rlValueRepresentation` object — Returned when `agent` is an `rlACAgent`, `rlPGAgent`, or `rlPPOAgent` object
- `rlQValueRepresentation` object — Returned when `agent` is an `rlQAgent`, `rlSARSAAgent`, `rlDQNAgent`, `rlDDPGAgent`, or `rlTD3Agent` object with a single critic
- Two-element row vector of `rlQValueRepresentation` objects — Returned when `agent` is an `rlTD3Agent` object with two critics

See Also

`getActor` | `getLearnableParameters` | `setActor` | `setCritic` | `setLearnableParameters`

Topics
“Create Policy and Value Function Representations”
“Import Policy and Value Function Representations”

Introduced in R2019a

getLearnableParameters

Package: `rl.representation`

Obtain learnable parameter values from policy or value function representation

Syntax

```
val = getLearnableParameters(rep)
```

Description

`val = getLearnableParameters(rep)` returns the values of the learnable parameters from the reinforcement learning policy or value function representation `rep`.

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 representation from the agent.

```
critic = getCritic(agent);
```

Obtain the learnable parameters from the critic.

```
params = getLearnableParameters(critic);
```

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.

```
agent = setCritic(agent,critic);
```

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 representation from the agent.

```
actor = getActor(agent);
```

Obtain the learnable parameters from the actor.

```
params = getLearnableParameters(actor);
```

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.

```
agent = setActor(agent,actor);
```

Input Arguments

rep — Policy or value function representation
`rlValueRepresentation` object | `rlQValueRepresentation` object | `rlDeterministicActorRepresentation` object | `rlStochasticActorRepresentation` object

Policy or value function representation, specified as one of the following:

- `rlValueRepresentation` object — Value function representation
- `rlQValueRepresentation` object — Q-value function representation
- `rlDeterministicActorRepresentation` object — Actor representation with deterministic actions
- `rlStochasticActorRepresentation` object — Actor representation with stochastic actions

To create a policy or value function representation, use one of the following methods:

- Create a representation using the corresponding representation object.
- Obtain the existing value function representation from an agent using `getCritic`
- Obtain the existing policy representation from an agent using `getActor`.

Output Arguments

val — Learnable parameter values
cell array

Learnable parameter values for the representation 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.

Compatibility Considerations

`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

`getActor` | `getCritic` | `setActor` | `setCritic` | `setLearnableParameters`

Topics

"Create Policy and Value Function Representations"
"Import Policy and Value Function Representations"

Introduced in R2019a

getMaxQValue

Obtain maximum state-value function estimate for Q-value function representation with discrete action space

Syntax

```
[maxQ,maxActionIndex] = getValue(qValueRep,obs)
[maxQ,maxActionIndex,state] = getValue( ___ )
```

Description

`[maxQ,maxActionIndex] = getValue(qValueRep,obs)` returns the maximum estimated state-value function for Q-value function representation `qValueRep` given environment observations `obs`. `getMaxQValue` determines the discrete action for which the Q-value estimate is greatest and returns that Q value (`maxQ`) and the corresponding action index (`maxActionIndex`).

`[maxQ,maxActionIndex,state] = getValue( ___ )` returns the state of the representation. Use this syntax when `qValueRep` is a recurrent neural network.

Examples

Obtain Maximum Q-Value Function Estimates

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 deep neural network for a multi-output Q-value function representation.

```
criticNetwork = [
    imageInputLayer([4 1 1],'Normalization','none','Name','state')
    fullyConnectedLayer(50, 'Name', 'CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    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);
```

Obtain value function estimates for each possible discrete action using random observations.

```
obs = rand(4,1);
val = getValue(critic,{obs})
val = 2×1 single column vector

    0.0139
   -0.1851
```

`val` contains two value function estimates, one for each possible discrete action.

You can obtain the maximum Q-value function estimate across all the discrete actions.

```
[maxVal,maxIndex] = getMaxQValue(critic,{obs})
```

```
maxVal = single
    0.0139

maxIndex = 1
```

`maxVal` corresponds to the maximum entry in `val`.

You can also obtain maximum Q-value function estimates for a batch of observations. For example, obtain value function estimates for a batch of 10 observations.

```
[batchVal,batchIndex] = getMaxQValue(critic,{rand(4,1,10)});
```

Input Arguments

`qValueRep` — Q-value representation
`rlQValueRepresentation` object

Q-value representation, specified as an `rlQValueRepresentation` 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 `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`.

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 `qValueRep` 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 `qValueRep` does not use a recurrent neural network, then $L_S = 1$.

`state` — Representation state
cell array

Representation state, returned as a cell array. If `qValueRep` 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:

```
valueRep = setState(qValueRep,state);
```

See Also
`getAction` | `getValue`

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

Introduced in R2020a

getObservationInfo

Obtain observation data specifications from reinforcement learning environment or agent

Syntax

```
obsInfo = getObservationInfo(env)
obsInfo = getObservationInfo(agent)
```

Description

`obsInfo = getObservationInfo(env)` extracts observation information from reinforcement learning environment `env`.

`obsInfo = getObservationInfo(agent)` extracts observation information from reinforcement learning agent `agent`.

Examples

Extract Action and Observation Information from Reinforcement Learning Environment

Extract action and observation information that you can use to create other environments or agents.

The reinforcement learning environment for this example is the simple longitudinal dynamics for ego car and lead car. The training goal is to make the ego car travel at a set velocity while maintaining a safe distance from lead car by controlling longitudinal acceleration (and braking). This example uses the same vehicle model as the “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox) example.

Open the model and create the reinforcement learning environment.

```
mdl = 'rlACCMdl';
open_system(mdl);
agentblk = [mdl '/RL Agent'];
% create the observation info
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 Info
actInfo = rlNumericSpec([1 1],'LowerLimit',-3,'UpperLimit',2);
actInfo.Name = 'acceleration';
% define environment
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 `SimulinkWithAgent` object with the above properties.

Extract the action and observation information from the reinforcement learning environment `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
`SimulinkEnvWithAgent` object

Reinforcement learning environment from which the observation information has to be extracted, specified as a `SimulinkEnvWithAgent` object.

For more information on reinforcement learning environments, see “Create Simulink Environments for Reinforcement Learning”.

agent — Reinforcement learning agent
`rlQAgent` object | `rlSARSAAgent` object | `rlDQNAgent` object | `rlDDPGAgent` object | `rlPGAgent` object | `rlACAgent` object

Reinforcement learning agent from which the observation information has to be extracted, specified as one of the following objects:

- `rlQAgent`
- `rlSARSAAgent`

- `rlDQNAgent`
- `rlDDPGAgent`
- `rlPGAgent`
- `rlACAgent`

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

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

See Also

`getActionInfo` | `rlACAgent` | `rlDDPGAgent` | `rlDQNAgent` | `rlFiniteSetSpec` | `rlNumericSpec` | `rlPGAgent` | `rlQAgent` | `rlSARSAAgent`

Topics
“Create Simulink Environments for Reinforcement Learning”
“Reinforcement Learning Agents”

Introduced in R2019a

getValue

Obtain estimated value function representation

Syntax

```
value = getValue(valueRep,obs)
value = getValue(qValueRep,obs)
value = getValue(qValueRep,obs,act)
[value,state] = getValue( ___ )
```

Description

`value = getValue(valueRep,obs)` returns the estimated value function for the state value function representation `valueRep` given environment observations `obs`.

`value = getValue(qValueRep,obs)` returns the estimated state-action value functions for the multiple Q-value function representation `qValueRep` given environment observations `obs`. In this case, `qValueRep` has as many outputs as there are possible discrete actions, and `getValue` returns the state-value function for each action.

`value = getValue(qValueRep,obs,act)` returns the estimated state-action value function for the single-output Q-value function representation `qValueRep` given environment observations `obs` and actions `act`. In this case, `getValue` returns the state-value function for the given observation and action inputs.

`[value,state] = getValue( ___ )` returns the state of the representation. Use this syntax when `valueRep` or `qValueRep` is a recurrent neural network.

Examples

Obtain State Value Function Estimates

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 deep neural network for the critic.

```
criticNetwork = [
    imageInputLayer([4 1 1],'Normalization','none','Name','state')
    fullyConnectedLayer(8,'Name','fc')
    reluLayer('Name','relu')
    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);
```

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
    -0.0899
```

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 Multi-Output Q-Value Function Estimates

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 deep neural network for a multi-output Q-value function representation.

```
criticNetwork = [
    imageInputLayer([4 1 1],'Normalization','none','Name','state')
    fullyConnectedLayer(50, 'Name', 'CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    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);
```

Obtain value function estimates for each possible discrete action using random observations.

```
val = getValue(critic,{rand(4,1)})
```

```
val = 2×1 single column vector
```

```
    0.0139
   -0.1851
```

`val` contains two 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.

```
batchVal = getValue(critic,{rand(4,1,10)});
size(batchVal)
```

```
ans = 1×2

     2    10
```

`batchVal` contains two value function estimates for each observation in the batch.

Obtain Single-Output Q-Value Function Estimates

Create observation specifications for two observation input channels.

```
obsinfo = [rlNumericSpec([8 3]), rlNumericSpec([4 1])];
```

Create action specification.

```
actinfo = rlNumericSpec([2 1]);
```

Create a deep neural network for the critic. This network has three input channels (two for observations and one for actions).

```
observationPath1 = [
    imageInputLayer([8 3 1],'Normalization','none','Name','state1')
    fullyConnectedLayer(10, 'Name', 'fc1')
    additionLayer(3,'Name','add')
    reluLayer('Name','relu1')
    fullyConnectedLayer(10,'Name','fc4')
    reluLayer('Name', 'relu2')
    fullyConnectedLayer(1,'Name','fc5')];
observationPath2 = [
    imageInputLayer([4 1 1],'Normalization','none','Name','state2')
    fullyConnectedLayer(10, 'Name','fc2')];
actionPath = [
    imageInputLayer([2 1 1],'Normalization','none','Name','action');
    fullyConnectedLayer(10, 'Name', 'fc3')];
net = layerGraph(observationPath1);
net = addLayers(net,observationPath2);
net = addLayers(net,actionPath);
net = connectLayers(net,'fc2','add/in2');
net = connectLayers(net,'fc3','add/in3');
```

Create the critic representation with this network.

```
c = rlQValueRepresentation(net,obsinfo,actinfo,...
    'Observation',{'state1','state2'},'Action',{'action'});
```

Create random observation set of batch size 64 for each channel.

```
batchobs_ch1 = rand(8,3,64);
batchobs_ch2 = rand(4,1,64);
```

Create 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.

```
qvalue = getValue(c,{batchobs_ch1,batchobs_ch2},{batchact});
```

Input Arguments

valueRep — Value function representation
`rlValueRepresentation` object

Value function representation, specified as an `rlValueRepresentation` object.

qValueRep — Q-value function representation
`rlQValueRepresentation` object

Q-value function representation, specified as an `rlQValueRepresentation` 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 `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`.

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 `valueRep` or `qValueRep` 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`.

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 representation (`valueRep`), N = 1.
  - For a single-output state-action value representation (`qValueRep`), N = 1.
  - For a multi-output state-action value representation (`qValueRep`), 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` — Representation state
cell array

Representation state for a recurrent neural network, returned as a cell array. If `valueRep` or `qValueRep` 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:

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

See Also
`getAction` | `getMaxQValue`

Topics
“Custom Agents”
“Train Reinforcement Learning Policy Using Custom Training Loop”

Introduced in R2020a

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.

See Also

Topics
“Create MATLAB Environments for Reinforcement Learning”

Introduced in R2019a

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')
```

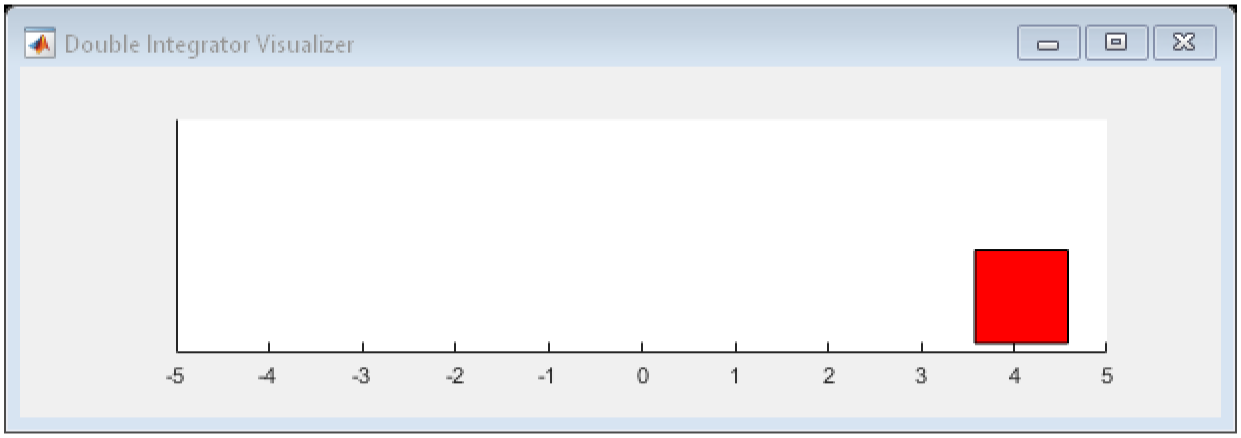
```
env = 
  DoubleIntegratorContinuousAction with properties:

             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
```

Continuous Simple Pendulum Model Reinforcement Learning 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' | 'CartPole-Continuous' | 'DoubleIntegrator-Discrete' | 'DoubleIntegrator-Continuous' | 'SimplePendulumWithImage-Discrete' | 'SimplePendulumWithImage-Continuous' | 'WaterFallGridWorld-Deterministic' | 'WaterFallGridWorld-Stochastic' | 'SimplePendulumModel-Discrete' | 'SimplePendulumModel-Continuous' | 'CartPoleSimscapeModel-Discrete' | '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'`

See Also

Topics
“Create MATLAB Environments for Reinforcement Learning”
“Create Simulink Environments for Reinforcement Learning”
“Load Predefined Control System Environments”
“Load Predefined Simulink Environments”

Introduced in R2019a

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 Policy and Value Function Representations".

`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” (Deep Learning Toolbox). For more information on creating deep neural networks for reinforcement learning, see “Create Policy and Value Function Representations”.

`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
spec object | array of spec objects

Observation specification, specified as a reinforcement learning spec object or an array of spec objects. You can extract `obsInfo` from an existing environment using `getObservationInfo`. Or, you can construct the specs manually using a spec command such as `rlFiniteSetSpec` or `rlNumericSpec`. This specification defines such information about the observations as the dimensions and names of the observation signals.

`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
spec object

Action specification, specified as a reinforcement learning spec object. You can extract `actInfo` from an existing environment using `getActionInfo`. Or, you can construct the spec manually using a spec command such as `rlFiniteSetSpec` or `rlNumericSpec`. This specification defines such information about the action as the dimensions and name of the action signal.

For linear basis function representations, the action signal must be a scalar, a column vector, or a discrete action.

`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".

Compatibility Considerations

`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.
`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` | `rlDeterministicActorRepresentation` | `rlQValueRepresentation` | `rlRepresentationOptions` | `rlStochasticActorRepresentation` | `rlValueRepresentation`

Topics
“Create Policy and Value Function Representations”
“Reinforcement Learning Agents”

Introduced in R2019a

rlSimulinkEnv

Create a reinforcement learning environment using a dynamic model implemented in Simulink

Syntax

```
env = rlSimulinkEnv(mdl,agentBlock,obsInfo,actInfo)
env = rlSimulinkEnv( ___ ,'UseFastRestart',fastRestartToggle)
```

Description

`env = rlSimulinkEnv(mdl,agentBlock,obsInfo,actInfo)` creates a reinforcement learning environment object `env` using the Simulink model name `mdl`, the path to the agent block `agentBlock`, observation information `obsInfo`, and action information `actInfo`.

`env = rlSimulinkEnv( ___ ,'UseFastRestart',fastRestartToggle)` creates a reinforcement learning environment object `env` with additional option to enable fast restart.

Examples

Reinforcement Learning Environment for Simulink models

For this example, consider the `rlSimplePendulumModel` Simulink model. The model is a simple frictionless pendulum that is initially hanging in a downward position.

Open the model.

```
mdl = 'rlSimplePendulumModel';
open_system(mdl)
```

Assign the agent block path information, and create `rlNumericSpec` and `rlFiniteSetSpec` objects for the observation and action information. You can use dot notation to assign property values of the `rlNumericSpec` and `rlFiniteSetSpec` objects.

```
agentBlk = [mdl '/RL Agent'];
obsInfo = rlNumericSpec([3 1])
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [3 1]
       DataType: "double"
```

```
actInfo = rlFiniteSetSpec([2 1])
```

```
actInfo = 
  rlFiniteSetSpec with properties:
       Elements: [2x1 double]
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

```
obsInfo.Name = 'observations';
actInfo.Name = 'torque';
```

Create the reinforcement learning environment for the Simulink model using information extracted in the previous steps.

```
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, consider randomly initializing `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'
```

Input Arguments

`mdl` — Simulink model name

string | character vector

Simulink model name, specified as a string or character vector.

`agentBlock` — Agent block path

string | character vector

Agent block path, specified as a string or character vector. The specified agent block can be inside of a model reference.

For more information on configuring an agent block for reinforcement learning, see RL Agent.

`obsInfo` — Observation information

array of `rlNumericSpec` objects | array of `rlFiniteSetSpec` objects

Observation information, specified as an array of one of the following:

- `rlNumericSpec` objects
- `rlFiniteSetSpec` objects
- A mix of `rlNumericSpec` and `rlFiniteSetSpec` objects

For more information, see `getObservationInfo`.

`actInfo` — Action information
array of `rlNumericSpec` objects | array of `rlFiniteSetSpec` objects

Action information, specified as an array of one of the following:

- `rlNumericSpec` objects
- `rlFiniteSetSpec` objects
- A mix of `rlNumericSpec` and `rlFiniteSetSpec` objects

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 Environments for Reinforcement Learning”.

See Also

RL Agent | `getActionInfo` | `getObservationInfo` | `rlFiniteSetSpec` | `rlNumericSpec`

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)

Introduced in R2019a

setActor

Package: `rl.agent`

Set actor representation of reinforcement learning agent

Syntax

```
newAgent = setActor(oldAgent,actor)
```

Description

`newAgent = setActor(oldAgent,actor)` returns a new reinforcement learning agent, `newAgent`, that uses the specified actor representation. Apart from the actor representation, the new agent has the same configuration as the specified original agent, `oldAgent`.

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 representation from the agent.

```
actor = getActor(agent);
```

Obtain the learnable parameters from the actor.

```
params = getLearnableParameters(actor);
```

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.

```
agent = setActor(agent,actor);
```

Add Layer to Actor Representation

Assume that you have an existing reinforcement learning agent, `agent`. For this example, load the trained agent from “Train DDPG Agent to Control Double Integrator System”:

```
load('DoubleIntegDDPG.mat','agent')
```

Further, assume that this agent has an actor representation that contains the following shallow neural network structure:

```
oldActorNetwork = [
        imageInputLayer([2 1 1],'Normalization','none','Name','state')
        fullyConnectedLayer(1,'Name','action')];
```

Create the new network with the additional fully connected hidden layer:

```
newActorNetwork = [
        imageInputLayer([2 1 1],'Normalization','none','Name','state')
        fullyConnectedLayer(3,'Name','hidden');
        fullyConnectedLayer(1,'Name','action')];
```

Create the corresponding actor representation:

```
actor = rlDeterministicActorRepresentation(newActorNetwork,...
    getObservationInfo(agent),getActionInfo(agent),...
    'Observation',{'state'},...
    'Action',{'action'})
```

```
actor = 
  rlDeterministicActorRepresentation with properties:

         ActionInfo: [1x1 rl.util.rlNumericSpec]
    ObservationInfo: [1x1 rl.util.rlNumericSpec]
            Options: [1x1 rl.option.rlRepresentationOptions]
```

Set the actor representation of the agent to the new augmented actor:

```
agent = setActor(agent,actor);
```

To check your agent, use getAction to return the action from a random observation.

```
getAction(agent,{rand(2,1)})
```

```
ans = single
    1.4134
```

You can now test and train the agent against the environment.

Input Arguments

oldAgent — Reinforcement learning agent

rlDDPGAgent object | rlTD3Agent object | rlPGAgent object | rlACAgent object | rlPPOAgent object

Reinforcement learning agent that contains an actor representation, specified as one of the following:

- `rlDDPGAgent` object
- `rlTD3Agent` object
- `rlACAgent` object

- `rlPGAgent` object
- `rlPPOAgent` object

`actor` — Actor representation
`rlDeterministicActorRepresentation` object | `rlStochasticActorRepresentation` object

Actor representation object, specified as one of the following:

- `rlDeterministicActorRepresentation` object — Specify when `agent` is an `rlDDPGAgent` or `rlTD3Agent` object
- `rlStochasticActorRepresentation` object — Specify when `agent` is an `rlACAgent`, `rlPGAgent`, or `rlPPOAgent` object

The input and output layers of the specified representation must match the observation and action specifications of the original agent.

To create a policy or value function representation, use one of the following methods:

- Create a representation using the corresponding representation object.
- Obtain the existing policy representation from an agent using `getActor`.

Output Arguments

`newAgent` — Updated reinforcement learning agent
`rlDDPGAgent` object | `rlTD3Agent` object | `rlPGAgent` object | `rlACAgent` object | `rlPPOAgent` object

Updated reinforcement learning agent, returned as an agent object that uses the specified actor representation. Apart from the actor representation, the new agent has the same configuration as `oldAgent`.

See Also
`getActor` | `getCritic` | `getLearnableParameters` | `setCritic` | `setLearnableParameters`

Topics
"Create Policy and Value Function Representations"
"Import Policy and Value Function Representations"

Introduced in R2019a

setCritic

Package: `rl.agent`

Set critic representation of reinforcement learning agent

Syntax

```
newAgent = setActor(oldAgent,critic)
```

Description

`newAgent = setActor(oldAgent,critic)` returns a new reinforcement learning agent, `newAgent`, that uses the specified critic representation. Apart from the critic representation, the new agent has the same configuration as the specified original agent, `oldAgent`.

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 representation from the agent.

```
critic = getCritic(agent);
```

Obtain the learnable parameters from the critic.

```
params = getLearnableParameters(critic);
```

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.

```
agent = setCritic(agent,critic);
```

Add Layer to Critic Representation

Assume that you have an existing reinforcement learning agent, `agent`. For this example, load the trained agent from “Train AC Agent to Balance Cart-Pole System”:

```
load('MATLABCartpoleAC.mat')
```

Further, assume that this agent has a critic representation that contains the following shallow neural network structure with two layers:

```
oldCriticNetwork = [
        imageInputLayer([4 1 1],'Normalization','none','Name','state')
        fullyConnectedLayer(1,'Name','CriticFC')];
```

Create the new network with an additional 5-neurons fully connected hidden layer:

```
newCriticNetwork = [
        oldCriticNetwork(1)
        fullyConnectedLayer(5,'Name','hidden');
        oldCriticNetwork(2)];
```

Create the critic using the new network with the additional fully connected layer.

```
critic = rlValueRepresentation(newCriticNetwork,getObservationInfo(agent),'Observation',{'state'
```

Set the critic representation of the agent to the new critic.

```
newAgent = setCritic(agent,critic);
```

Input Arguments

`oldAgent` — Original reinforcement learning agent

`rlQAgent` object | `rlSARSAAgent` object | `rlDQNAgent` object | `rlDDPGAgent` object | `rlTD3Agent` object | `rlPGAgent` object | `rlACAgent` object | `rlPPOAgent` object

Original reinforcement learning agent that contains a critic representation, specified as one of the following:

- `rlQAgent` object
- `rlSARSAAgent` object
- `rlDQNAgent` object
- `rlDDPGAgent` object
- `rlTD3Agent` object
- `rlACAgent` object
- `rlPPOAgent` object
- `rlPGAgent` object that estimates a baseline value function using a critic

`critic` — Critic representation

`rlValueRepresentation` object | `rlQValueRepresentation` object | two-element row vector of `rlQValueRepresentation` objects

Critic representation object, specified as one of the following:

- `rlValueRepresentation` object — Returned when `agent` is an `rlACAgent`, `rlPGAgent`, or `rlPPOAgent` object
- `rlQValueRepresentation` object — Returned when `agent` is an `rlQAgent`, `rlSARSAAgent`, `rlDQNAgent`, `rlDDPGAgent`, or `rlTD3Agent` object with a single critic

- Two-element row vector of `rlQValueRepresentation` objects — Returned when `agent` is an `rlTD3Agent` object with two critics

Output Arguments

`newAgent` — Updated reinforcement learning agent
`rlQAgent` object | `rlSARSAAgent` object | `rlDQNAgent` object | `rlDDPGAgent` object | `rlTD3Agent` object | `rlPGAgent` object | `rlACAgent` object | `rlPPOAgent` object

Updated reinforcement learning agent, returned as an agent object that uses the specified critic representation. Apart from the critic representation, the new agent has the same configuration as `oldAgent`.

See Also

`getActor` | `getCritic` | `getLearnableParameters` | `setActor` | `setLearnableParameters`

Topics
"Create Policy and Value Function Representations"
"Import Policy and Value Function Representations"

Introduced in R2019a

setLearnableParameters

Package: `rl.representation`

Set learnable parameter values of policy or value function representation

Syntax

```
newRep = setLearnableParameters(oldRep,val)
```

Description

`newRep = setLearnableParameters(oldRep,val)` returns a new policy or value function representation, `newRep`, with the same structure as the original representation, `oldRep`, and the learnable parameter values specified in `val`.

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 representation from the agent.

```
critic = getCritic(agent);
```

Obtain the learnable parameters from the critic.

```
params = getLearnableParameters(critic);
```

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.

```
agent = setCritic(agent,critic);
```

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 representation from the agent.

```
actor = getActor(agent);
```

Obtain the learnable parameters from the actor.

```
params = getLearnableParameters(actor);
```

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.

```
agent = setActor(agent,actor);
```

Input Arguments

oldRep — Original policy or value function representation
`rlValueRepresentation` object | `rlQValueRepresentation` object | `rlDeterministicActorRepresentation` object | `rlStochasticActorRepresentation` object

Original policy or value function representation, specified as one of the following:

- `rlValueRepresentation` object — Value function representation
- `rlQValueRepresentation` object — Q-value function representation
- `rlDeterministicActorRepresentation` object — Actor representation with deterministic actions
- `rlStochasticActorRepresentation` object — Actor representation with stochastic actions

To create a policy or value function representation, use one of the following methods:

- Create a representation using the corresponding representation object.
- Obtain the existing value function representation from an agent using `getCritic`
- Obtain the existing policy representation from an agent using `getActor`.

val — Learnable parameter values
cell array

Learnable parameter values for the representation object, specified as a cell array. The parameters in `val` must be compatible with the structure and parameterization of `oldRep`.

To obtain a cell array of learnable parameter values from an existing representation, which you can then modify, use the `getLearnableParameters` function.

Output Arguments

newRep — New policy or value function representation
`rlValueRepresentation` | `rlQValueRepresentation` |
`rlDeterministicActorRepresentation` | `rlStochasticActorRepresentation`

New policy or value function representation, returned as a representation object of the same type as `oldRep`. `newRep` has the same structure as `oldRep` but with parameter values from `val`.

Compatibility Considerations

`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
`getActor` | `getCritic` | `getLearnableParameters` | `setActor` | `setCritic`

Topics
"Create Policy and Value Function Representations"
"Import Policy and Value Function Representations"

Introduced in R2019a

sim

Package: `rl.env`

Simulate a trained reinforcement learning agent within a specified environment

Syntax

```
experience = sim(env,agent,simOpts)
experience = sim(agent,env,simOpts)
```

Description

`experience = sim(env,agent,simOpts)` simulates a reinforcement learning environment against an agent configured for that environment..

`experience = sim(agent,env,simOpts)` performs the same simulation as the previous syntax.

Examples

Simulate a 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
```

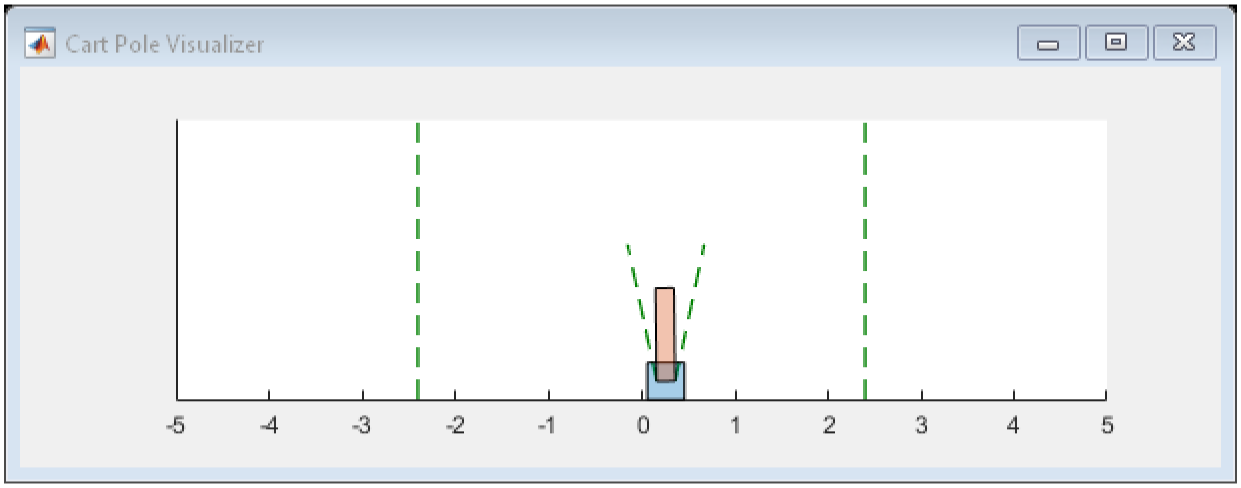
```
agent = 
  rlPGAgent with properties:
    AgentOptions: [1x1 rl.option.rlPGAgentOptions]
```

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);
```

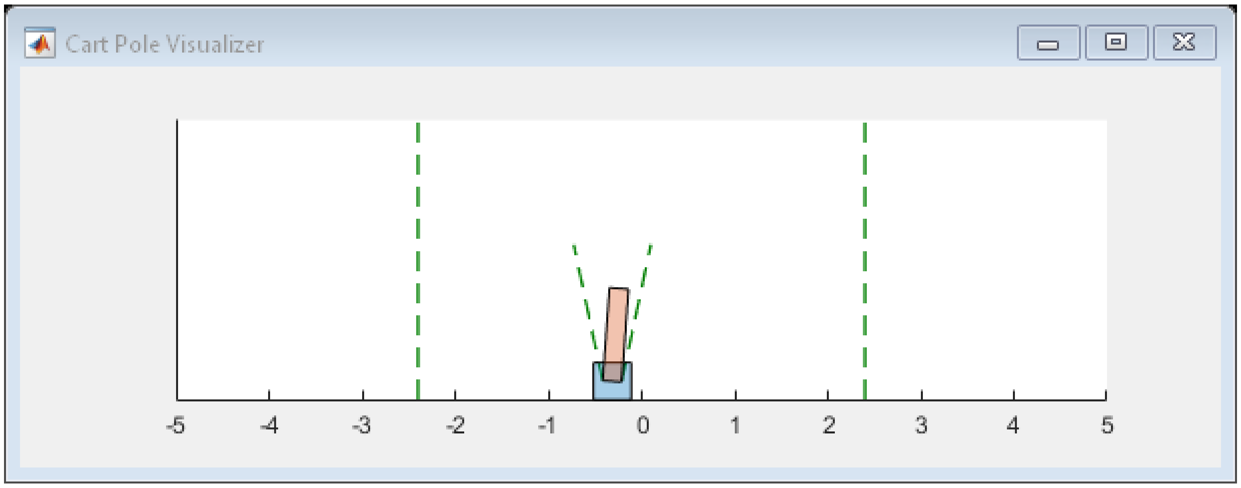
For the predefined cart-pole environment used in this example. you can use `plot` to generate a visualization of the cart-pole system. When you simulate the environment, this plot updates automatically so that you can watch the system evolve during the simulation.

```
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 is a timeseries or a structure of timeseries data. 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]
```

Input Arguments

`env` — Environment
reinforcement learning environment object

Environment in which the agent acts, specified as a reinforcement learning environment object, such as:

- A predefined MATLAB or Simulink environment created using `rlPredefinedEnv`
- A custom MATLAB environment you create with functions such as `rlFunctionEnv` or `rlCreateEnvTemplate`
- A custom Simulink environment you create using `rlSimulinkEnv`

For more information about creating and configuring environments, see:

- “Create MATLAB Environments for Reinforcement Learning”
- “Create Simulink Environments for Reinforcement Learning”

When `env` is a Simulink environment, calling `sim` compiles and simulates the model associated with the environment.

`agent` — Agent
reinforcement learning agent object

Agent to train, specified as a reinforcement learning agent object, such as an `rlACAgent` or `rlDDPGAgent` object, or a custom agent. Before simulation, you must configure the actor and critic representations of the agent. 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 such parameters and options 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 f elements in the array is equal to the number of simulations specified by the `NumSimulations` option of `rlSimulationOptions` The fields of the `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 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:

- 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.

See Also
`rlSimulationOptions` | `train`

Topics
"Train Reinforcement Learning Agents"

Introduced in R2019a

train

Package: rl.agent

Train a reinforcement learning agent within a specified environment

Syntax

```
trainStats = train(agent,env,trainOpts)
```

Description

`trainStats = train(agent,env,trainOpts)` trains a reinforcement learning agent with a specified environment. After each training episode, `train` updates the parameters of `agent` to maximize the expected long-term reward of the environment. When training terminates, the agent reflects the state of training at termination.

Use the training options `trainOpts` to specify training parameters such as the criteria for termination of training, when to save agents, the maximum number of episodes to train, and the maximum number of steps per episode.

Examples

Train a Reinforcement Learning Agent

Configure the training parameters and train a reinforcement learning agent. Typically, before training, you must configure your environment and agent. 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 RLTrainExample.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: [4×1 double]
```

```
agent
```

```
agent = 
  rlPGAgent with properties:

    AgentOptions: [1×1 rl.option.rlPGAgentOptions]
```

To train this agent, you must first specify 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 1000 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 = 1000;
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`.

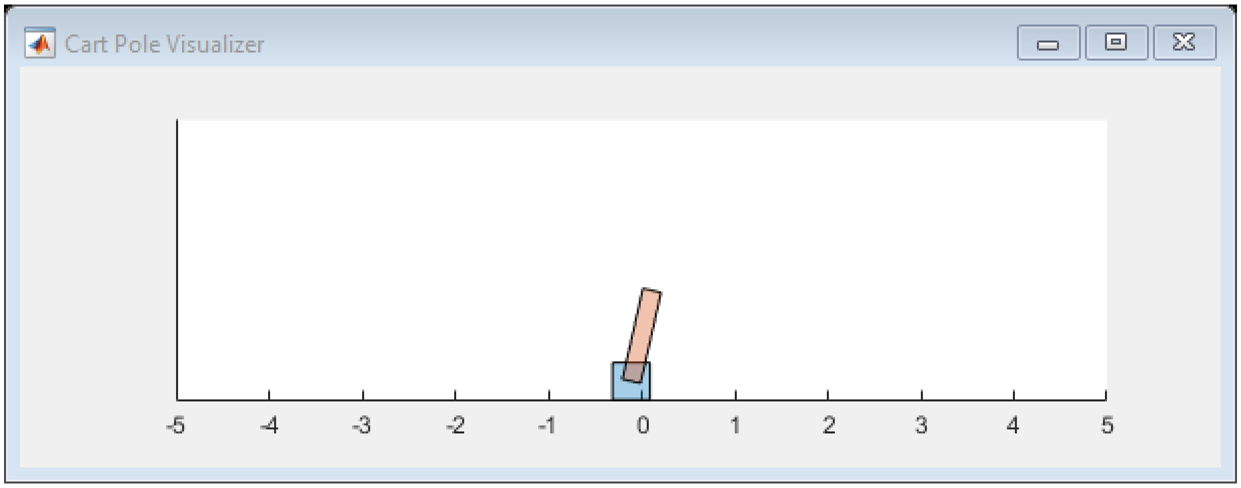
```
trainOpts.SaveAgentCriteria = "EpisodeReward";
trainOpts.SaveAgentValue = 500;
trainOpts.SaveAgentDirectory = "savedAgents";
```

Finally, 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";
```

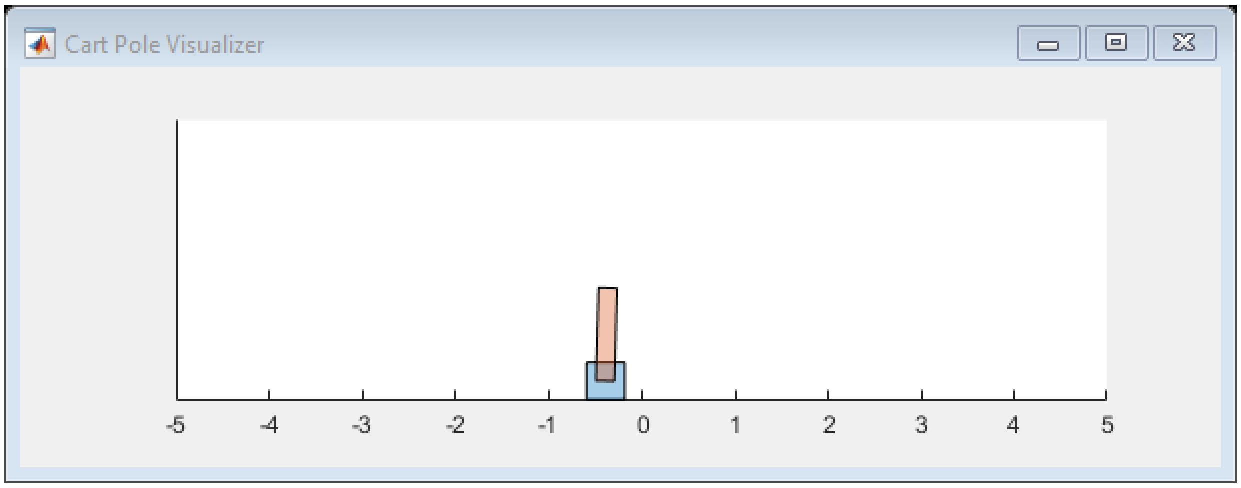
You are now ready to train the PG agent. For the predefined cart-pole environment used in this example. you can use `plot` to generate a visualization of the cart-pole system.

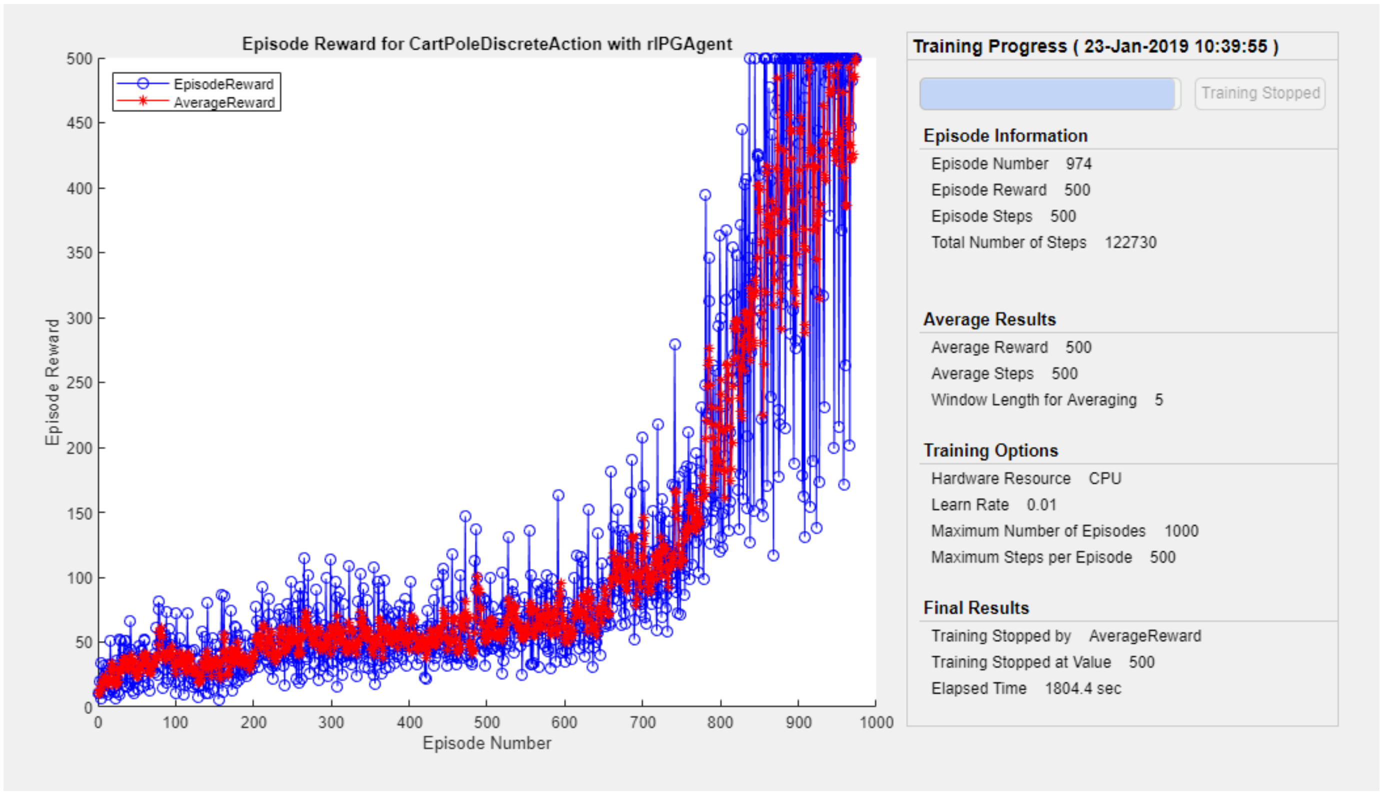
```
plot(env)
```



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);
```





Training Progress (23-Jan-2019 10:39:55)

Episode Information

Episode Number	974
Episode Reward	500
Episode Steps	500
Total Number of Steps	122730

Average Results

Average Reward	500
Average Steps	500
Window Length for Averaging	5

Training Options

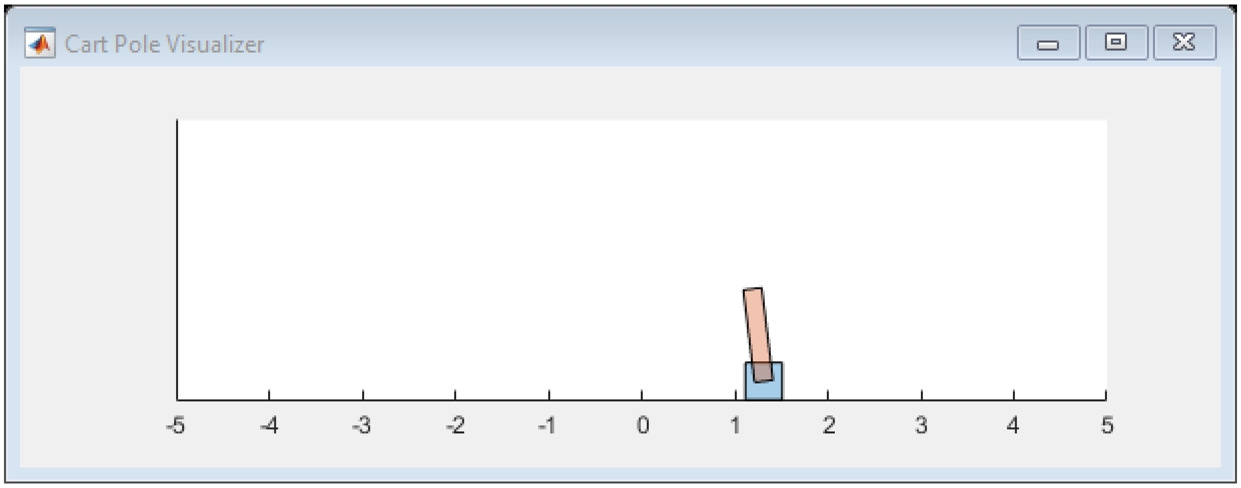
Hardware Resource	CPU
Learn Rate	0.01
Maximum Number of Episodes	1000
Maximum Steps per Episode	500

Final Results

Training Stopped by	AverageReward
Training Stopped at Value	500
Elapsed Time	1804.4 sec

The 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.

Input Arguments

agent — Agent
reinforcement learning agent object

Agent to train, specified as a reinforcement learning agent object, such as an `rlACAgent` or `rlDDPGAgent` object, or a custom agent. Before training, you must configure the actor and critic representations of the agent. 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 agent acts, specified as a reinforcement learning environment object, such as:

- A predefined MATLAB or Simulink environment created using `rlPredefinedEnv`
- A custom MATLAB environment you create with functions such as `rlFunctionEnv` or `rlCreateEnvTemplate`
- A custom Simulink environment you create using `rlSimulinkEnv`

For more information about creating and configuring environments, see:

- “Create MATLAB Environments for Reinforcement Learning”
- “Create Simulink Environments for Reinforcement Learning”

When `env` is a Simulink environment, calling `train` compiles and simulates the model associated with the environment.

`trainOpts` — Training parameters and options
`rlTrainingOptions` object

Training parameters and options, specified as an `rlTrainingOptions` object. Use this argument to specify such parameters and options as:

- Criteria for ending training
- Criteria for saving candidate agents
- How to display training progress
- Options for parallel computing

For details, see `rlTrainingOptions`.

Output Arguments

`trainStats` — Training episode data
structure

Training episode data, returned as a structure containing the following fields.

`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 long-term reward for each episode
column vector

Critic estimate of 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.

Tips

- `train` updates the agent as training progresses. To preserve the original agent parameters for later use, save the agent 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.

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`. For more information, see `rlTrainingOptions`.

See Also

`rlTrainingOptions` | `sim`

Topics

"Train Reinforcement Learning Agents"

Introduced in R2019a

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 1-90). 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 Environments for Reinforcement Learning"
- "Create Simulink Environments for Reinforcement Learning"

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.

See Also

`rlCreateEnvTemplate` | `rlFunctionEnv` | `rlSimulinkEnv`

Topics

"Create Simulink Environment and Train Agent"
"Create Custom MATLAB Environment from Template"

Introduced in R2019a

2

Objects

quadraticLayer

Quadratic layer for actor or critic network

Description

A `QuadraticLayer` is a deep neural network layer that takes an input vector and outputs a vector of quadratic monomials constructed from the input elements. 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]`.

The quadratic layer is useful when you need a layer whose output is some quadratic function of its inputs. For instance, inserting a `QuadraticLayer` into a network lets you recreate the structure of quadratic value functions such as those used in LQR controller design. For an example that uses a `QuadraticLayer`, see "Train DDPG Agent to Control Double Integrator System".

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 2-2 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'

  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 = 1×6

     1     2     4     3     6     9
```

You can incorporate `qLayer` into an actor network or critic network for reinforcement learning.

See Also

`scalingLayer` | `softplusLayer`

Topics

“Train DDPG Agent to Control Double Integrator System”
“Create Policy and Value Function Representations”

Introduced in R2019a

rlACAgent

Actor-critic 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 goal of this agent is to optimize the policy (actor) directly and train a critic to estimate the return or future rewards.

For more information see “Actor-Critic Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlACAgent(actor,critic,agentOptions)
```

Description

`agent = rlACAgent(actor,critic,agentOptions)` creates an actor-critic agent with the specified actor and critic networks and sets the `AgentOptions` property.

Input Arguments

`actor` — Actor network representation
`rlStochasticActorRepresentation` object

Actor network representation for the policy, specified as an `rlStochasticActorRepresentation` object. For more information on creating actor representations, see “Create Policy and Value Function Representations”.

`critic` — Critic network representation
`rlValueRepresentation` object

Critic network representation for estimating the discounted long-term reward, specified as an `rlValueRepresentation`. For more information on creating critic representations, see “Create Policy and Value Function Representations”.

Properties

`AgentOptions` — Agent options
`rlACAgentOptions` object

Agent options, specified as an `rlACAgentOptions` object.

Object Functions

train	Train a reinforcement learning agent within a specified environment
sim	Simulate a trained reinforcement learning agent within a specified environment
getActor	Get actor representation from reinforcement learning agent
setActor	Set actor representation of reinforcement learning agent
getCritic	Get critic representation from reinforcement learning agent
setCritic	Set critic representation of reinforcement learning agent
generatePolicyFunction	Create function that evaluates trained policy of reinforcement learning agent

Examples

Create Actor-Critic Agent

Create an environment interface and obtain its observation and action specifications.

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

Create a critic representation.

```
% create the network to be used as approximator in the critic
criticNetwork = [
    imageInputLayer([4 1 1],'Normalization','none','Name','state')
    fullyConnectedLayer(1,'Name','CriticFC')];

% set some options for the critic
criticOpts = rlRepresentationOptions('LearnRate',8e-3,'GradientThreshold',1);

% create the critic
critic = rlValueRepresentation(criticNetwork,obsInfo,'Observation',{'state'},criticOpts);
```

Create an actor representation.

```
% create the network to be used as approximator in the actor
actorNetwork = [
    imageInputLayer([4 1 1],'Normalization','none','Name','state')
    fullyConnectedLayer(2,'Name','action')];

% set some options for the actor
actorOpts = rlRepresentationOptions('LearnRate',8e-3,'GradientThreshold',1);

% create the actor
actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'state'},actorOpts);
```

Specify agent options, and create an AC agent using the environment, actor, and critic.

```
agentOpts = rlACAgentOptions('NumStepsToLookAhead',32,'DiscountFactor',0.99);
agent = rlACAgent(actor,critic,agentOpts)
```

```
agent = 
  rlACAgent with properties:
    AgentOptions: [1x1 rl.option.rlACAgentOptions]
```

To check your agent, use getAction to return the action from a random observation.

```
getAction(agent,{rand(4,1)})
```

```
ans = -10
```

You can now test and train the agent against the environment.

See Also

`rlACAgentOptions`

Topics

"Actor-Critic Agents"
"Reinforcement Learning Agents"
"Train Reinforcement Learning Agents"

Introduced in R2019a

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 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 2-7 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
`1` (default) | positive integer

Number of steps to look ahead in model training, specified as a positive integer. For AC agents, the number of steps to look ahead corresponds to the training episode length.

`EntropyLossWeight` — Entropy loss weight
`0` (default) | scalar value between `0` and `1`

Entropy loss weight, specified as a scalar value between `0` and `1`, inclusive. A higher 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.

The entropy loss function for episode step t is:

$$H_t = E \sum_{k=1}^{M} \mu_k(S_t|\theta_\mu) \ln \mu_k(S_t|\theta_\mu)$$

Here:

- E is the entropy loss weight.
- M is the number of possible actions.
- $\mu_k(S_t)$ is the probability of taking action A_k when in state S_t following the current policy.

When gradients are computed during training, an additional gradient component is computed for minimizing this loss function.

`SampleTime` — Sample time of agent
1 (default) | positive scalar

Sample time of agent, specified as a positive scalar.

`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 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: 1
      EntropyLossWeight: 0
             SampleTime: 1
         DiscountFactor: 0.9500
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

See Also

Topics
"Actor-Critic Agents"

Introduced in R2019a

rlDDPGAgent

Deep deterministic policy gradient 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.

For more information, see “Deep Deterministic Policy Gradient Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlDDPGAgent(actor,critic,agentOptions)
```

Description

`agent = rlDDPGAgent(actor,critic,agentOptions)` creates a DDPG agent with the specified actor and critic networks and sets the `AgentOptions` property.

Input Arguments

`actor` — Actor network representation
`rlDeterministicActorRepresentation` object

Actor network representation, specified as an `rlDeterministicActorRepresentation`. For more information on creating actor representations, see “Create Policy and Value Function Representations”.

`critic` — Critic network representation
`rlQValueRepesentation` object

Critic network representation, specified as an `rlQValueRepresentation` object. For more information on creating critic representations, see “Create Policy and Value Function Representations”.

Properties

`AgentOptions` — Agent options
`rlDDPGAgentOptions` object

Agent options, specified as an `rlDDPGAgentOptions` object.

ExperienceBuffer — Experience buffer
ExperienceBuffer object

Experience buffer, specified as an `ExperienceBuffer` object. During training the agent stores each of its experiences (*S*,*A*,*R*,*S*') 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*.

For more information on how the agent samples experience from the buffer during training, see “Deep Deterministic Policy Gradient Agents”.

Object Functions

train	Train a reinforcement learning agent within a specified environment
sim	Simulate a trained reinforcement learning agent within a specified environment
getActor	Get actor representation from reinforcement learning agent
setActor	Set actor representation of reinforcement learning agent
getCritic	Get critic representation from reinforcement learning agent
setCritic	Set critic representation of reinforcement learning agent
generatePolicyFunction	Create function that evaluates trained policy of reinforcement learning agent

Examples

Create a DDPG Agent

Create a DDPG agent with actor and critic and obtain its observation and action specifications.

```
% load predefined environment
env = rlPredefinedEnv("DoubleIntegrator-Continuous");

% get observation and specification info
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a critic representation.

```
% create a network to be used as underlying critic approximator
statePath = imageInputLayer([obsInfo.Dimension(1) 1 1], 'Normalization', 'none', 'Name', 'state'
actionPath = imageInputLayer([numel(actInfo) 1 1], 'Normalization', 'none', 'Name', 'action');
commonPath = [concatenationLayer(1,2,'Name','concat')
             quadraticLayer('Name','quadratic')
             fullyConnectedLayer(1,'Name','StateValue','BiasLearnRateFactor', 0, 'Bias', 0)];
criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork, actionPath);
criticNetwork = addLayers(criticNetwork, commonPath);
criticNetwork = connectLayers(criticNetwork,'state','concat/in1');
criticNetwork = connectLayers(criticNetwork,'action','concat/in2');
```

```
% set some options for the critic
criticOpts = rlRepresentationOptions('LearnRate',5e-3,'GradientThreshold',1);

% create the critic based on the network approximator
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation',{'state'},'Action',{'action'},criticOpts);
```

Create an actor representation.

```
% create a network to be used as underlying actor approximator
actorNetwork = [
    imageInputLayer([obsInfo.Dimension(1) 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(numel(actInfo), 'Name', 'action', 'BiasLearnRateFactor', 0, 'Bias', 0)];

% set some options for the actor
actorOpts = rlRepresentationOptions('LearnRate',1e-04,'GradientThreshold',1);

% create the actor based on the network approximator
actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'state'},'Action',{'action'},actorOpts);
```

Specify agent options, and create a PG agent using the environment, actor, and critic.

```
agentOpts = rlDDPGAgentOptions(...
    'SampleTime',env.Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',32);
agent = rlDDPGAgent(actor,critic,agentOpts);
```

To check your agent, use getAction to return the action from a random observation.

```
getAction(agent,{rand(2,1)})
```

```
ans = single
    -0.4719
```

You can now test and train the agent against the environment.

See Also

`rlDDPGAgentOptions`

Topics

"Deep Deterministic Policy Gradient Agents"
"Reinforcement Learning Agents"
"Train Reinforcement Learning Agents"

Introduced in R2019a

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 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 2-13 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 2-15.

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 variance of each action to a different value while using the same decay rate for both variances.

```
opt = rlDDPGAgentOptions;
opt.ExplorationModel.Variance = [0.1 0.2];
opt.ExplorationModel.VarianceDecayRate = 1e-4;
```

`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` — Flag for clearing the experience buffer
`true` (default) | `false`

Flag for clearing the experience buffer before training, specified as a logical value.

`SaveExperienceBufferWithAgent` — Flag for saving the experience buffer
`false` (default) | `true`

Flag for saving the experience buffer data when saving the agent, specified as a logical value. This option applies both when saving candidate agents during training and when saving agents using the `save` function.

For some agents, such as those with a large experience buffer and image-based observations, the memory required for saving their experience buffer is large. In such cases, to not save the experience buffer data, set `SaveExperienceBufferWithAgent` to `false`.

If you plan to further train your saved agent, you can start training with the previous experience buffer as a starting point. In this case, set `SaveExperienceBufferWithAgent` to `true`.

`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 steps ahead
`1` (default) | positive integer

Number of steps to look ahead during training, specified as a positive integer.

`ExperienceBufferLength` — Experience buffer size
`10000` (default) | positive integer

Experience buffer size, specified as a positive integer. During training, the agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.

`SampleTime` — Sample time of agent
`1` (default) | positive scalar

Sample time of agent, specified as a positive scalar.

`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 reinforcement learning agent

Examples

Create DDPG Agent Options Object

This example shows how to create a DDPG agent option object.

Create an `rlDDPGAgentOptions` object that specifies the mini-batch size.

```
opt = rlDDPGAgentOptions('MiniBatchSize',48)
```

```
opt = 
  rlDDPGAgentOptions with properties:

                           NoiseOptions: [1x1 rl.option.OrnsteinUhlenbeckActionNoise]
                     TargetSmoothFactor: 1.0000e-03
                  TargetUpdateFrequency: 1
    ResetExperienceBufferBeforeTraining: 1
          SaveExperienceBufferWithAgent: 0
                          MiniBatchSize: 48
                    NumStepsToLookAhead: 1
                 ExperienceBufferLength: 10000
                             SampleTime: 1
                         DiscountFactor: 0.9900
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

Algorithms

Noise Model

An `OrnsteinUhlenbeckActionNoise` object has the following numeric value properties.

Property	Description
`InitialAction`	Initial value of action for noise model
`Mean`	Noise model mean
`MeanAttractionConstant`	Constant specifying how quickly the noise model output is attracted to the mean
`Variance`	Noise model variance

Property	Description
`VarianceDecayRate`	Decay rate of the variance
`VarianceMin`	Minimum variance

At each sample time step, the noise model is updated using the following formula, where `Ts` is the agent sample time.

```
x(k) = x(k-1) + MeanAttractionConstant.*(Mean - x(k-1)).*Ts
       + Variance.*randn(size(Mean)).*sqrt(Ts)
```

At each sample time step, the variance decays as shown in the following code.

```
decayedVariance = Variance.*(1 - VarianceDecayRate);
Variance = max(decayedVariance,VarianceMin);
```

For continuous action signals, it is important to set the noise variance appropriately to encourage exploration. It is common to have `Variance*sqrt(Ts)` be 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 variance. Also, to increase exploration, you can reduce the `VarianceDecayRate`.

Compatibility Considerations

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;`

See Also

Topics
"Deep Deterministic Policy Gradient Agents"

Introduced in R2019a

rlDeterministicActorRepresentation

Deterministic actor representation 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 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 Policy and Value Function Representations”.

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. `observationInfo` 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” (Deep Learning Toolbox). For more information on creating deep neural networks for reinforcement learning, see “Create Policy and Value Function Representations”.

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
specification object | array of specification objects

Observation specifications, a reinforcement learning specification object or an array of specification objects defining properties such as dimensions, data type, and names of the observation signals.

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
`rlNumericSpec` object

Action specifications for a continuous action space, a `rlNumericSpec` object defining properties such as dimensions, data type and name of the action signals. The deterministic actor representation does not support discrete actions.

You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlNumericSpec`.

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 reinforcement learning agent
rlTD3Agent	Twin-delayed deep deterministic policy gradient reinforcement learning agent
getAction	Obtain action from agent or actor representation 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 4 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 2 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-dimensional vector (the observation vector just defined by `obsInfo`), and its output must be the action (here called `myact`) and be a two-dimensional vector, as defined by `actInfo`.

```
net = [imageInputLayer([4 1 1], '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: [1×1 rl.util.rlNumericSpec]
    ObservationInfo: [1×1 rl.util.rlNumericSpec]
            Options: [1×1 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 = 
  2×1 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'});
```

See Also

Functions
`getActionInfo` | `getObservationInfo` | `rlRepresentationOptions`

Topics
"Create Policy and Value Function Representations"
"Reinforcement Learning Agents"

Introduced in R2020a

rlDQNAgent

Deep Q-network 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.

For more information, “Deep Q-Network Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlDQNAgent(critic,agentOptions)
```

Description

`agent = rlDQNAgent(critic,agentOptions)` creates a DQN agent with the specified critic network and sets the `AgentOptions` property.

Input Arguments

`critic` — Critic network representation
`rlQValueRepresentation` object

Critic network representation, specified as an `rlQValueRepresentation` object. For more information on creating critic representations, see “Create Policy and Value Function Representations”.

Your critic representation can use a recurrent neural network as its function approximator. However, only the multi-output Q-value function representation supports recurrent neural networks. For an example, see “Create DQN Agent with Recurrent Neural Network” on page 2-26.

Properties

`AgentOptions` — Agent options
`rlDQNAgentOptions` object

Agent options, specified as an `rlDQNAgentOptions` object.

`ExperienceBuffer` — Experience buffer
`ExperienceBuffer` object

Experience buffer, specified as an `ExperienceBuffer` object. During training the agent stores each of its experiences (*S*,*A*,*R*,*S*') 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*.

For more information on how the agent samples experience from the buffer during training, see “Deep Q-Network Agents”.

Object Functions

train	Train a reinforcement learning agent within a specified environment
sim	Simulate a trained reinforcement learning agent within a specified environment
getActor	Get actor representation from reinforcement learning agent
setActor	Set actor representation of reinforcement learning agent
getCritic	Get critic representation from reinforcement learning agent
setCritic	Set critic representation of reinforcement learning agent
generatePolicyFunction	Create function that evaluates trained policy of reinforcement learning agent

Examples

Create a DQN Agent

Create an environment interface and obtain its observation and action specifications. For this environment load the predefined environment used for the discrete cart-pole system.

```
% load predefined environment
env = rlPredefinedEnv("CartPole-Discrete");

% get observation and specification info
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a critic representation.

```
% create a critic network to be used as underlying approximator
statePath = [
    imageInputLayer([4 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(24, 'Name', 'CriticStateFC1')
    reluLayer('Name', 'CriticRelu1')
    fullyConnectedLayer(24, 'Name', 'CriticStateFC2')];
actionPath = [
    imageInputLayer([1 1 1], 'Normalization', 'none', 'Name', 'action')
    fullyConnectedLayer(24, 'Name', 'CriticActionFC1')];
commonPath = [
    additionLayer(2,'Name', 'add')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(1, 'Name', 'output')];
criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork, actionPath);
criticNetwork = addLayers(criticNetwork, commonPath);
criticNetwork = connectLayers(criticNetwork,'CriticStateFC2','add/in1');
criticNetwork = connectLayers(criticNetwork,'CriticActionFC1','add/in2');

% set some options for the critic
criticOpts = rlRepresentationOptions('LearnRate',0.01,'GradientThreshold',1);

% create the critic based on the network approximator
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation',{'state'},'Action',{'action'},criticOpts);
```

Specify agent options, and create a DQN agent using the environment and critic.

```
agentOpts = rlDQNAgentOptions(...
    'UseDoubleDQN',false, ...
    'TargetUpdateMethod',"periodic", ...
    'TargetUpdateFrequency',4, ...
    'ExperienceBufferLength',100000, ...
    'DiscountFactor',0.99, ...
    'MiniBatchSize',256);

agent = rlDQNAgent(critic,agentOpts)
```

```
agent = 
  rlDQNAgent with properties:

        AgentOptions: [1x1 rl.option.rlDQNAgentOptions]
    ExperienceBuffer: [1x1 rl.util.ExperienceBuffer]
```

To check your agent, use getAction to return the action from a random observation.

```
getAction(agent,{rand(4,1)})
```

```
ans = 10
```

You can now test and train the agent against the environment.

Create DQN Agent with 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 an `lstmLayer` as one of the other network layers.

For DQN agents, only the multi-output Q-value function representation supports recurrent neural networks.

```
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);
```

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',20);
agentOptions.EpsilonGreedyExploration.EpsilonDecay = 1e-4;
agent = rlDQNAgent(critic,agentOptions);
```

See Also

`rlDQNAgentOptions`

Topics

"Deep Q-Network Agents"
"Reinforcement Learning Agents"
"Train Reinforcement Learning Agents"

Introduced in R2019a

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 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 2-28 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 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
`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.
`EpsilonMin`	Minimum value of `Epsilon`
`EpsilonDecay`	Decay rate

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)
```

To specify exploration options, use dot notation after creating the `rlDQNAgentOptions` object. For example, set the epsilon value to `0.9`.

```
opt = rlDQNAgentOptions;
opt.EpsilonGreedyExploration.Epsilon = 0.9;
```

If your agent converges on local optima too quickly, promote agent exploration by increasing `Epsilon`.

`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.

`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` — Flag for clearing the experience buffer
`true` (default) | `false`

Flag for clearing the experience buffer before training, specified as a logical value.

`SaveExperienceBufferWithAgent` — Flag for saving the experience buffer
`false` (default) | `true`

Flag for saving the experience buffer data when saving the agent, specified as a logical value. This option applies both when saving candidate agents during training and when saving agents using the `save` function.

For some agents, such as those with a large experience buffer and image-based observations, the memory required for saving their experience buffer is large. In such cases, to not save the experience buffer data, set `SaveExperienceBufferWithAgent` to `false`.

If you plan to further train your saved agent, you can start training with the previous experience buffer as a starting point. In this case, set `SaveExperienceBufferWithAgent` to `true`.

`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 steps ahead
`1` (default) | positive integer

Number of steps to look ahead during training, specified as a positive integer.

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 updates the critic using a mini-batch of experiences randomly sampled from the buffer.

`SampleTime` — Sample time of agent
`1` (default) | positive scalar

Sample time of agent, specified as a positive scalar.

`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 reinforcement learning agent

Examples

Create DQN Agent Options Object

This example shows how to create a 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: [1×1 rl.option.EpsilonGreedyExploration]
                          SequenceLength: 1
                      TargetSmoothFactor: 1.0000e-03
                   TargetUpdateFrequency: 1
    ResetExperienceBufferBeforeTraining: 1
          SaveExperienceBufferWithAgent: 0
                           MiniBatchSize: 48
                    NumStepsToLookAhead: 1
                 ExperienceBufferLength: 10000
                              SampleTime: 1
                          DiscountFactor: 0.9900
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

Compatibility Considerations

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;`

See Also

Topics
"Deep Q-Network Agents"

Introduced in R2019a

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 (default)

Name of the `rlFiniteSetSpec` object, specified as a string. Use this property to set a meaningful name for your finite set.

`Description` — Description of the `rlFiniteSetSpec` object
string (default)

Description of the `rlFiniteSetSpec` object, specified as a string. Use this property to specify a meaningful description of the finite set values.

`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`.

DataType — Information about the type of data

string (default)

This property is read-only.

Information about the type of data, specified as a string.

Object Functions

rlSimulinkEnv	Create a reinforcement learning environment using a dynamic model implemented in Simulink
rlFunctionEnv	Specify custom reinforcement learning environment dynamics using functions
rlRepresentation	(Not recommended) Model representation for reinforcement learning agents

Examples

Reinforcement Learning Environment for Simulink models

For this example, consider the `rlSimplePendulumModel` Simulink model. The model is a simple frictionless pendulum that is initially hanging in a downward position.

Open the model.

```
mdl = 'rlSimplePendulumModel';
open_system(mdl)
```

Assign the agent block path information, and create `rlNumericSpec` and `rlFiniteSetSpec` objects for the observation and action information. You can use dot notation to assign property values of the `rlNumericSpec` and `rlFiniteSetSpec` objects.

```
agentBlk = [mdl '/RL Agent'];
obsInfo = rlNumericSpec([3 1])
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [3 1]
       DataType: "double"
```

```
actInfo = rlFiniteSetSpec([2 1])
```

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [2x1 double]
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [1 1]
```

```
       DataType: "double"
```

```
obsInfo.Name = 'observations';
actInfo.Name = 'torque';
```

Create the reinforcement learning environment for the Simulink model using information extracted in the previous steps.

```
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, consider randomly initializing `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"
```

See Also

`getActionInfo` | `getObservationInfo` | `rlFunctionEnv` | `rlNumericSpec` |
`rlRepresentation` | `rlSimulinkEnv`

Introduced in R2019a

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(obsInfo,actInfo,stepfcn,resetfcn)
```

Description

`env = rlFunctionEnv(obsInfo,actInfo,stepfcn,resetfcn)` creates a reinforcement learning environment using the observation specification and agent specification you provide. You also provide your own MATLAB functions that define step and reset behavior for the environment.

Input Arguments

`obsInfo` — Observation specification
spec object

Observation specification, specified as a reinforcement learning spec object created with a spec command such as `rlFiniteSetSpec` or `rlNumericSpec`. This specification defines such information about the observations as the dimensions and names of the observation signals.

`actInfo` — Action specification
spec object

Action specification, specified as a reinforcement learning spec object created with a spec command such as `rlFiniteSetSpec` or `rlNumericSpec`. The specification defines such information about the actions as the dimensions and names of the action signals.

`stepfcn` — Step behavior for the environment
function | function handle | anonymous function

Step behavior for the environment, specified as a function, function handle, or anonymous function. `stepfcn` sets the value of the StepFcn property.

`resetfcn` — Reset behavior for the environment
function | function handle | anonymous function

Reset behavior for the environment, specified as a function, function handle, or anonymous function. `resetfcn` sets the value of the ResetFcn property.

Properties

`StepFcn` — Step behavior for the environment
function | function handle | anonymous function

Step behavior for the environment, specified as a function, function handle, or anonymous function. When you create an `rlFunctionEnv` object, the `stepfcn` input argument sets the value of this property.

`StepFcn` is a function that you provide which describes how the environment advances to the next state from a given action. This function must have two inputs and four outputs, as illustrated by the following signature:

```
[Observation,Reward,IsDone,LoggedSignals] = myStepFunction(Action,LoggedSignals)
```

Thus, 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:

- `Action` and `Observation` — The current action and the returned observation. These values must match the dimensions and data types specified in `actInfo` and `obsInfo`, respectively.
- `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.

To use additional input arguments beyond this required set, specify `StepFcn` using a function handle or an anonymous function. 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 | function handle | anonymous function

Reset behavior for the environment, specified as a function, function handle, or anonymous function. When you create a `rlFunctionEnv` object, the `resetfcn` input argument sets the value of this property.

The reset function that you provide must have no inputs and two outputs, as illustrated by the following signature.

```
[InitialObservation,LoggedSignals] = myResetFunction
```

Thus, the reset function computes the initial values of the observation signals. For instance, `sim` calls the reset function to reset the environment at the start of each simulation, and `train` calls it at the start of each training episode. Therefore, you might create a reset function that randomizes certain state values, such that each training episode begins from different initial conditions.

The `InitialObservation` output must match the dimensions and data type of `obsInfo`.

To pass information from the reset condition into the first step, specify that information in the reset function as the output structure `LoggedSignals`.

To use input arguments with your reset function, specify `ResetFcn` using a function handle or an anonymous function. 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 or agent
getObservationInfo	Obtain observation data specifications from reinforcement learning environment or agent
sim	Simulate a trained reinforcement learning agent within a 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”.

See Also

`rlCreateEnvTemplate` | `rlPredefinedEnv`

Topics

“Create MATLAB Environment Using Custom Functions”

Introduced in R2019a

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

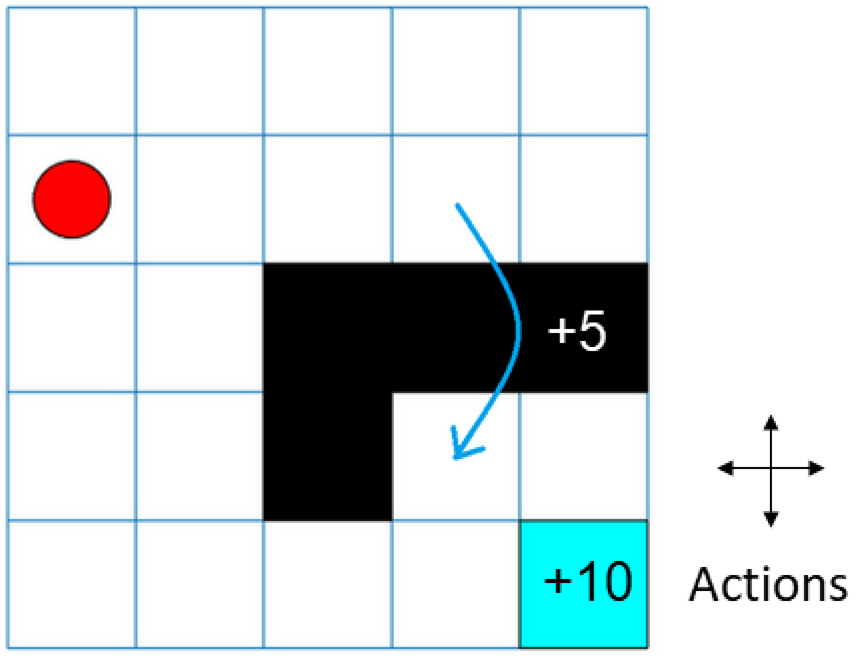
getActionInfo	Obtain action data specifications from reinforcement learning environment or agent
getObservationInfo	Obtain observation data specifications from reinforcement learning environment or agent
sim	Simulate a trained reinforcement learning agent within a specified environment
train	Train a reinforcement learning agent 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]
```

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;
```

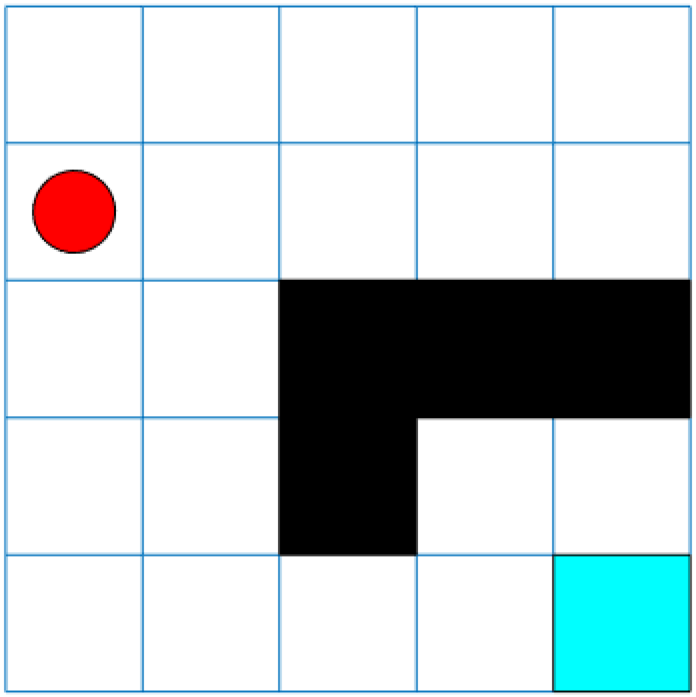
Now, use `rlMDPEnv` to create a grid world environment using the `GridWorld` object `GW`.

```
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)
```



See Also

`createGridWorld` | `rlPredefinedEnv`

Topics

"Train Reinforcement Learning Agent in Basic Grid World"
"Create Custom Grid World Environments"
"Train Reinforcement Learning Agent in MDP Environment"

Introduced in R2019a

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 2-45 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.

`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.

`Name` — Name of the `rlNumericSpec` object
string (default)

Name of the `rlNumericSpec` object, specified as a string.

`Description` — Description of the `rlNumericSpec` object
string (default)

Description of the `rlNumericSpec` object, specified as a string.

Dimension — Dimension of the data space
numeric vector (default)

This property is read-only.

Dimension of the data space, specified as a numeric vector.

DataType — Information about the type of data
string (default)

This property is read-only.

Information about the type of data, specified as a string.

Object Functions

rlSimulinkEnv	Create a reinforcement learning environment using a dynamic model implemented in Simulink
rlFunctionEnv	Specify custom reinforcement learning environment dynamics using functions
rlRepresentation	(Not recommended) Model representation for reinforcement learning agents

Examples

Reinforcement Learning Environment for Simulink models

For this example, consider the `rlSimplePendulumModel` Simulink model. The model is a simple frictionless pendulum that is initially hanging in a downward position.

Open the model.

```
mdl = 'rlSimplePendulumModel';
open_system(mdl)
```

Assign the agent block path information, and create `rlNumericSpec` and `rlFiniteSetSpec` objects for the observation and action information. You can use dot notation to assign property values of the `rlNumericSpec` and `rlFiniteSetSpec` objects.

```
agentBlk = [mdl '/RL Agent'];
obsInfo = rlNumericSpec([3 1])
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [3 1]
       DataType: "double"
```

```
actInfo = rlFiniteSetSpec([2 1])
```

```
actInfo = 
  rlFiniteSetSpec with properties:
       Elements: [2x1 double]
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

```
obsInfo.Name = 'observations';
actInfo.Name = 'torque';
```

Create the reinforcement learning environment for the Simulink model using information extracted in the previous steps.

```
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, consider randomly initializing `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'
```

See Also

`getActionInfo` | `getObservationInfo` | `rlFiniteSetSpec` | `rlFunctionEnv` | `rlRepresentation` | `rlSimulinkEnv`

Topics

"Train DDPG Agent for Adaptive Cruise Control"

Introduced in R2019a

rlPGAgent

Policy gradient 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 which directly computes an optimal policy that maximizes the long-term reward.

For more information on PG agents, see “Policy Gradient Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlPGAgent(actor)
agent = rlPGAgent(actor,critic)
agent = rlPGAgent(___,agentOptions)
```

Description

`agent = rlPGAgent(actor)` creates a PG 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 PG agent with the specified actor and critic networks. By default, the `UseBaseline` option is `true` in this case.

`agent = rlPGAgent(___,agentOptions)` creates a PG agent and sets the `AgentOptions` property.

Input Arguments

`actor` — Actor network representation
`rlStochasticActorRepresentation` object

Actor network representation, specified as an `rlStochasticActorRepresentation`. For more information on creating actor representations, see “Create Policy and Value Function Representations”.

`critic` — Critic network representation
`rlValueRepresentation` object

Critic network representation, specified as an `rlValueRepresentation` object. For more information on creating critic representations, see “Create Policy and Value Function Representations”.

Properties

`AgentOptions` — Agent options
`rlPGAgentOptions` object

Agent options, specified as an `rlPGAgentOptions` object.

Object Functions

train	Train a reinforcement learning agent within a specified environment
sim	Simulate a trained reinforcement learning agent within a specified environment
getActor	Get actor representation from reinforcement learning agent
setActor	Set actor representation of reinforcement learning agent
getCritic	Get critic representation from reinforcement learning agent
setCritic	Set critic representation of reinforcement learning agent
generatePolicyFunction	Create function that evaluates trained policy of reinforcement learning agent

Examples

Create a PG Agent

Create an environment interface.

```
% load predefined environment
env = rlPredefinedEnv("DoubleIntegrator-Discrete");

% get observation and specification info
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a critic representation to use as a baseline.

```
% create a network to be used as underlying critic approximator
baselineNetwork = [
    imageInputLayer([obsInfo.Dimension(1) 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(8, 'Name', 'BaselineFC')
    reluLayer('Name', 'CriticRelu1')
    fullyConnectedLayer(1, 'Name', 'BaselineFC2', 'BiasLearnRateFactor', 0)];

% set some options for the critic
baselineOpts = rlRepresentationOptions('LearnRate',5e-3,'GradientThreshold',1);

% create the critic based on the network approximator
baseline = rlValueRepresentation(baselineNetwork,obsInfo,'Observation',{'state'},baselineOpts);
```

Create an actor representation.

```
% create a network to be used as underlying actor approximator
actorNetwork = [
    imageInputLayer([obsInfo.Dimension(1) 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(numel(actInfo.Elements), 'Name', 'action', 'BiasLearnRateFactor', 0)];

% set some options for the actor
```

```
actorOpts = rlRepresentationOptions('LearnRate',5e-3,'GradientThreshold',1);

% create the actor based on the network approximator
actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'state'},actorOpts);
```

Specify agent options, and create a PG agent using the environment, actor, and critic.

```
agentOpts = rlPGAgentOptions(...
    'UseBaseline',true, ...
    'DiscountFactor', 0.99);
agent = rlPGAgent(actor,baseline,agentOpts)
```

```
agent = 
  rlPGAgent with properties:

    AgentOptions: [1x1 rl.option.rlPGAgentOptions]
```

To check your agent, use getAction to return the action from a random observation.

```
getAction(agent,{rand(2,1)})
```

```
ans = -2
```

You can now test and train the agent against the environment.

See Also

`rlPGAgentOptions`

Topics

"Policy Gradient Agents"
"Reinforcement Learning Agents"
"Train Reinforcement Learning Agents"

Introduced in R2019a

rlPGAgentOptions

Options for policy gradient 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 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 2-51 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`

Instruction to use baseline for learning, specified as a logical values. 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.

`SampleTime` — Sample time of agent
`1` (default) | positive scalar

Sample time of agent, specified as a positive scalar.

`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.

`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 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.

The entropy loss function for episode step t is:

$$H_t = E \sum_{k=1}^{M} \mu_k(S_t|\theta_\mu) \ln \mu_k(S_t|\theta_\mu)$$

Here:

- E is the entropy loss weight.
- M is the number of possible actions.
- $\mu_k(S_t)$ is the probability of taking action A_k following the current policy.

When gradients are computed during training, an additional gradient component is computed for minimizing this loss function.

Object Functions

rlPGAgent	Policy gradient 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
           SampleTime: 1
       DiscountFactor: 0.9000
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

See Also

Topics
"Policy Gradient Agents"

Introduced in R2019a

rlPPOAgent

Proximal policy optimization reinforcement learning agent

Description

The 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.

For more information on PPO agents, see “Proximal Policy Optimization Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlPPOAgent(actor,critic,agentOptions)
```

Description

`agent = rlPPOAgent(actor,critic,agentOptions)` creates a proximal policy optimization (PPO) agent with the specified actor and critic networks and sets the `AgentOptions` property.

Input Arguments

`actor` — Actor network representation
`rlStochasticActorRepresentation` object

Actor network representation for the policy, specified as an `rlStochasticActorRepresentation` object. For more information on creating actor representations, see “Create Policy and Value Function Representations”.

Your actor representation can use a recurrent neural network as its function approximator. In this case, your critic must also use a recurrent neural network. For an example, see “Create PPO Agent with Recurrent Neural Networks” on page 2-56.

`critic` — Critic network representation
`rlValueRepresentation` object

Critic network representation for estimating the discounted long-term reward, specified as an `rlValueRepresentation`. For more information on creating critic representations, see “Create Policy and Value Function Representations”.

Your critic representation 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 2-56.

Properties

AgentOptions — Agent options
rlPPOAgentOptions object

Agent options, specified as an `rlPPOAgentOptions` object.

Object Functions

train	Train a reinforcement learning agent within a specified environment
sim	Simulate a trained reinforcement learning agent within a specified environment
getActor	Get actor representation from reinforcement learning agent
setActor	Set actor representation of reinforcement learning agent
getCritic	Get critic representation from reinforcement learning agent
setCritic	Set critic representation of reinforcement learning agent
generatePolicyFunction	Create function that evaluates trained policy of reinforcement learning agent

Examples

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);
```

Create a critic representation.

```
% create the network to be used as approximator in the critic
criticNetwork = [
    imageInputLayer([4 1 1],'Normalization','none','Name','state')
    fullyConnectedLayer(1,'Name','CriticFC')];

% set some options for the critic
criticOpts = rlRepresentationOptions('LearnRate',8e-3,'GradientThreshold',1);

% create the critic
critic = rlValueRepresentation(criticNetwork,obsInfo,'Observation',{'state'},criticOpts);
```

Create an actor representation.

```
% create the network to be used as approximator in the actor
actorNetwork = [
    imageInputLayer([4 1 1],'Normalization','none','Name','state')
    fullyConnectedLayer(2,'Name','action')];

% set some options for the actor
actorOpts = rlRepresentationOptions('LearnRate',8e-3,'GradientThreshold',1);

% create the actor
actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'state'},actorOpts);
```

Specify agent options, and create a PPO agent using the environment, actor, and critic.

```
agentOpts = rlPPOAgentOptions(...
    'ExperienceHorizon',1024, ...
    'DiscountFactor',0.95);
agent = rlPPOAgent(actor,critic,agentOpts)
```

```
agent = 
  rlPPOAgent with properties:

    AgentOptions: [1x1 rl.option.rlPPOAgentOptions]
```

To check your agent, use getAction to return the action from a random observation.

```
getAction(agent,{rand(4,1)})
```

```
ans = -10
```

You can now test and train the agent against the environment.

Create PPO Agent with Recurrent Neural Networks

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 an `lstmLayer` as one of the other network layers.

```
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);
```

Similarly, define a recurrent neural network for the actor.

```
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);
```

Create a PPO agent using the actor and critic representations.

```
agentOptions = rlPPOAgentOptions(...
    'AdvantageEstimateMethod', 'finite-horizon', ...
    'ClipFactor', 0.1);
agent = rlPPOAgent(actor,critic,agentOptions);
```

See Also
`rlPPOAgentOptions`

Topics
“Proximal Policy Optimization Agents”
“Reinforcement Learning Agents”
“Train Reinforcement Learning Agents”

Introduced in R2019b

rlPPOAgentOptions

Options for proximal policy optimization reinforcement learning 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 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 2-58 using name-value pairs. For example, `rlPPOAgentOptions('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

`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.

The `ExperienceHorizon` value must be greater than or equal to the `MiniBatchSize` 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 greater `0` and `1`

Entropy loss weight, specified as a scalar value between `0` and `1`. A higher 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.

For episode step *t*, the entropy loss function, which is added to the loss function for actor updates, is:

$$H_t = E\sum_{k=1}^{M} \mu_k(S_t|\theta_\mu)\ln\mu_k(S_t|\theta_\mu)$$

Here:

- E is the entropy loss weight.
- M is the number of possible actions.
- $\mu_k(S_t|\theta_\mu)$ is the probability of taking action A_k when in state S_t following the current policy.

`MiniBatchSize` — Mini-batch size
`128` (default) | positive integer

Mini-batch size used for each learning epoch, specified as a positive integer.

The `MiniBatchSize` value must be less than or equal to the `ExperienceHorizon` value.

`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 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"`

`SampleTime` — Sample time of agent
`1` (default) | positive scalar

Sample time of agent, specified as a positive scalar.

`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 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
                 SampleTime: 1
             DiscountFactor: 0.9900
```

You can modify options using dot notation. For example, set the agent sample time to `0.5`.

```
opt.SampleTime = 0.5;
```

See Also

Topics
"Proximal Policy Optimization Agents"

Introduced in R2019b

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.

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 network representation
`rlQValueRepresentation` object

Critic network representation, specified as an `rlQValueRepresentation` object. For more information on creating critic representations, see “Create Policy and Value Function Representations”.

Properties

`AgentOptions` — Agent options
`rlQAgentOptions` object

Agent options, specified as an `rlQAgentOptions` object.

Object Functions

train	Train a reinforcement learning agent within a specified environment
sim	Simulate a trained reinforcement learning agent within a specified environment
getActor	Get actor representation from reinforcement learning agent

setActor	Set actor representation of reinforcement learning agent
getCritic	Get critic representation from reinforcement learning agent
setCritic	Set critic representation of reinforcement learning agent
generatePolicyFunction	Create function that evaluates trained policy of reinforcement learning agent

Examples

Create a Q-Learning Agent

Create an environment interface.

```
env = rlPredefinedEnv("BasicGridWorld");
```

Create a critic Q-value function representation using a Q-table derived from the environment observation and action specifications.

```
qTable = rlTable(getObservationInfo(env),getActionInfo(env));
critic = rlQValueRepresentation(qTable,getObservationInfo(env),getActionInfo(env));
```

Create a Q-learning agent using the specified critic value function and an epsilon value of `0.05`.

```
opt = rlQAgentOptions;
opt.EpsilonGreedyExploration.Epsilon = 0.05;

agent = rlQAgent(critic,opt)
```

```
agent = 
  rlQAgent with properties:

    AgentOptions: [1x1 rl.option.rlQAgentOptions]
```

To check your agent, use getAction to return the action from a random observation.

```
getAction(agent,{randi(25)})
```

```
ans = 1
```

You can now test and train the agent against the environment.

See Also

Functions
`rlQAgentOptions`

Topics
“Q-Learning Agents”
“Reinforcement Learning Agents”
“Train Reinforcement Learning Agents”

Introduced in R2019a

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 2-63 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 numeric value properties.

Property	Description
`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.
`EpsilonMin`	Minimum value of `Epsilon`

Property	Description
EpsilonDecay	Decay rate

Epsilon is updated using the following formula when it is greater than EpsilonMin:

```
Epsilon = Epsilon*(1-EpsilonDecay)
```

To specify exploration options, use dot notation after creating the rlQAgentOptions object. For example, set the probability threshold to 0.9.

```
opt = rlQAgentOptions;
opt.EpsilonGreedyExploration.Epsilon = 0.9;
```

SampleTime — Sample time of agent
1 (default) | positive scalar

Sample time of agent, specified as a positive scalar.

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

This example shows how to create an options object for a Q-Learning agent.

Create an rlQAgentOptions object that specifies the agent sample time.

```
opt = rlQAgentOptions('SampleTime',0.5)
```

```
opt = 
  rlQAgentOptions with properties:

    EpsilonGreedyExploration: [1x1 rl.option.EpsilonGreedyExploration]
                  SampleTime: 0.5000
              DiscountFactor: 0.9900
```

You can modify options using dot notation. For example, set the agent discount factor to 0.95.

```
opt.DiscountFactor = 0.95;
```

See Also

Topics
"Q-Learning Agents"

Introduced in R2019a

rlQValueRepresentation

Q-Value function critic representation for reinforcement learning agents

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` or `rlDQNAgent`. For more information on creating representations, see “Create Policy and Value Function Representations”.

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 = rlValueRepresentation( ___ ,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`.

`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

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” (Deep Learning Toolbox). For more information on creating deep neural networks for reinforcement learning, see “Create Policy and Value Function Representations”.

`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 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'}`

`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
specification object | array of specification objects

Observation specifications, a reinforcement learning specification object or an array of specification objects defining properties such as the dimensions, data type, and names of the observation signals.

You can extract `observationInfo` from an existing environment or agent using `getObservationInfo`. You can also construct the specifications manually using a specification command such as `rlFiniteSetSpec` or `rlNumericSpec`.

`ActionInfo` — Action specifications
specification object

Action specifications, a reinforcement learning specification object, defining properties such as the dimensions, data type and name of the action signals. You can extract `actionInfo` from an existing environment or agent using `getActionInfo`. You can also construct the specification manually using `rlFiniteSetSpec` or `rlNumericSpec`.

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 reinforcement learning agent
rlTD3Agent	Twin-delayed deep deterministic policy gradient reinforcement learning agent
rlDQNAgent	Deep Q-network reinforcement learning agent
rlQAgent	Q-learning reinforcement learning agent
rlSARSAAgent	SARSA reinforcement learning agent
getValue	Obtain estimated value function representation
getMaxQValue	Obtain maximum state-value function estimate for Q-value function representation with discrete action space

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 4 doubles.

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

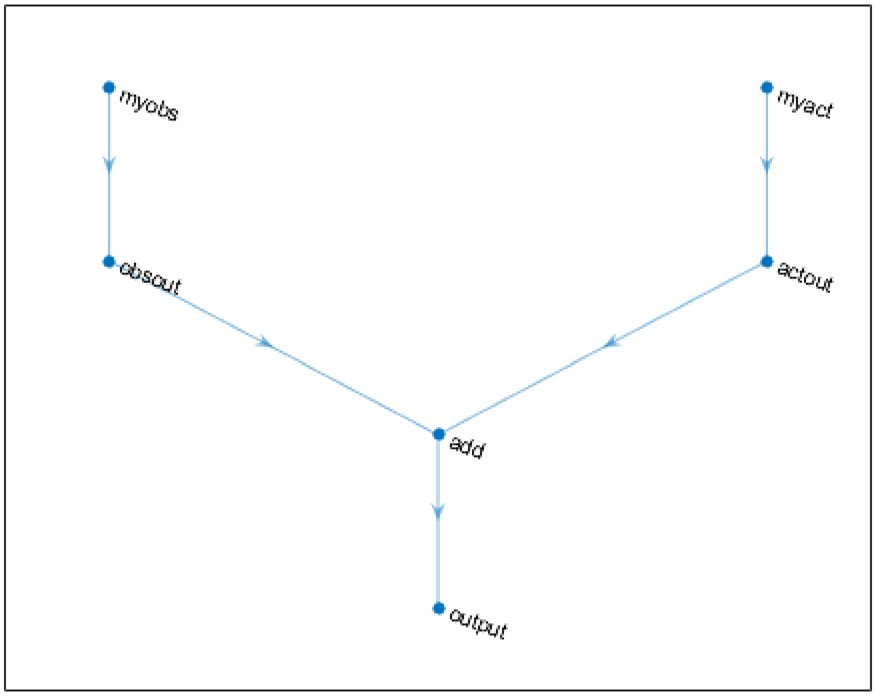
Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that a single action is a column vector containing 2 doubles.

```
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-dimensional vector (the observation vector defined by `obsInfo`). The action input (here called `myact`) must accept a two-dimensional 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 = [imageInputLayer([4 1 1], 'Normalization','none','Name','myobs')
    fullyConnectedLayer(1,'Name','obsout')];

% action path layers
actPath = [imageInputLayer([2 1 1], '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');
```

```
% plot network
plot(net)
```



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 4 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 3 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-dimensional 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 = [imageInputLayer([4 1 1], '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. Note that 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: [1×1 rl.util.rlFiniteSetSpec]
    ObservationInfo: [1×1 rl.util.rlNumericSpec]
            Options: [1×1 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 = 
  3×1 dlarray

    2.4007
    3.7697
    1.7370
```

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);
```

See Also

Functions
`getActionInfo` | `getObservationInfo` | `rlRepresentationOptions`

Topics
“Create Policy and Value Function Representations”
“Reinforcement Learning Agents”

Introduced in R2020a

rlRepresentationOptions

Options set for reinforcement learning agent representations (critics and actors)

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 2-78 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 the comma-separated pair consisting of `'LearnRate'` and 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 the comma-separated pair consisting of `'Optimizer'` and 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" (Deep Learning Toolbox) 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 the comma-separated pair consisting of `'OptimizerParameters'` and an `OptimizerParameters` object.

The `OptimizerParameters` object has the following properties.

`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.Epsilon = 1e-7;
```

`GradientThreshold` — Threshold value for gradient
`Inf` (default) | positive scalar

Threshold value for the representation gradient, specified as the comma-separated pair consisting of `'GradientThreshold'` and `Inf` or a positive scalar. If the gradient exceeds this value, the gradient is clipped as specified by the `GradientThresholdOption`. 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 the comma-separated pair consisting of `'GradientThresholdMethod'` and one of the following strings:

- `"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” (Deep Learning Toolbox) 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 the comma-separated pair consisting of `'L2RegularizationFactor'` and a nonnegative scalar. For more information, see “L2 Regularization” (Deep Learning Toolbox) 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 for training an agent that uses the representation, specified as the comma-separated pair consisting of `'UseDevice'` and either `"cpu"` or `"gpu"`.

The "gpu" option requires Parallel Computing Toolbox™. To use a GPU for training a network, you must also have a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

Example: 'UseDevice',"gpu"

Object Functions

rlValueRepresentation	Value function critic representation for reinforcement learning agents
rlQValueRepresentation	Q-Value function critic representation for reinforcement learning agents
rlDeterministicActorRepresentation	Deterministic actor representation for reinforcement learning agents
rlStochasticActorRepresentation	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
```

See Also

Topics
"Create Policy and Value Function Representations"
"Reinforcement Learning Agents"

Introduced in R2019a

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.

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 network representation
`rlQValueRepresentation` object

Critic network representation, specified as an `rlQValueRepresentation` object. For more information on creating critic representations, see “Create Policy and Value Function Representations”.

Properties

`AgentOptions` — Agent options
`rlSARSAAgentOptions` object

Agent options, specified as an `rlSARSAAgentOptions` object.

Object Functions

train	Train a reinforcement learning agent within a specified environment
sim	Simulate a trained reinforcement learning agent within a specified environment
getActor	Get actor representation from reinforcement learning agent

setActor	Set actor representation of reinforcement learning agent
getCritic	Get critic representation from reinforcement learning agent
setCritic	Set critic representation of reinforcement learning agent
generatePolicyFunction	Create function that evaluates trained policy of reinforcement learning agent

Examples

Create a SARSA Agent

Create or load an environment interface. For this example load the Basic Grid World environment interface.

```
env = rlPredefinedEnv("BasicGridWorld");
```

Create a critic value function representation using a Q table derived from the environment observation and action specifications.

```
qTable = rlTable(getObservationInfo(env),getActionInfo(env));
critic = rlQValueRepresentation(qTable,getObservationInfo(env),getActionInfo(env));
```

Create a SARSA agent using the specified critic value function and an epsilon value of `0.05`.

```
opt = rlSARSAAgentOptions;
opt.EpsilonGreedyExploration.Epsilon = 0.05;

agent = rlSARSAAgent(critic,opt)
```

```
agent = 
  rlSARSAAgent with properties:

    AgentOptions: [1x1 rl.option.rlSARSAAgentOptions]
```

To check your agent, use getAction to return the action from a random observation.

```
getAction(agent,{randi(25)})
```

```
ans = 1
```

You can now test and train the agent against the environment.

See Also
`rlSARSAAgentOptions`

Topics
"SARSA Agents"
"Reinforcement Learning Agents"
"Train Reinforcement Learning Agents"

Introduced in R2019a

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 2-85 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 numeric value properties.

Property	Description
`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.
`EpsilonMin`	Minimum value of `Epsilon`

Property	Description
EpsilonDecay	Decay rate

`Epsilon` is updated using the following formula when it is greater than `EpsilonMin`:

```
Epsilon = Epsilon*(1-EpsilonDecay)
```

To specify exploration options, use dot notation after creating the `rlSARSAAgentOptions` object. For example, set the probability threshold to `0.9`.

```
opt = rlSARSAAgentOptions;
opt.EpsilonGreedyExploration.Epsilon = 0.9;
```

`SampleTime` — Sample time of agent
`1` (default) | positive scalar

Sample time of agent, specified as a positive scalar.

`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

This example shows how to create a SARSA agent option 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]
                  SampleTime: 0.5000
              DiscountFactor: 0.9900
```

You can modify options using dot notation. For example, set the agent discount factor to `0.95`.

```
opt.DiscountFactor = 0.95;
```

See Also

Topics
"SARSA Agents"

Introduced in R2019a

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 2-88 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 the comma-separated pair consisting of `'UseParallel'` and either `true` or `false`. Setting this option to `true` configures simulation to use parallel computing. To specify options for parallel simulation, use the `ParallelizationOptions` property.

Using parallel computing requires Parallel Computing Toolbox software.

For more information about training using parallel computing, see “Train Reinforcement Learning Agents”.

Example: `'UseParallel',true`

`ParallelizationOptions` — Options to control parallel simulation
`ParallelTraining` object

Parallelization options to control parallel simulation, specified as the comma-separated pair consisting of `'ParallelizationOptions'` and 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 a trained reinforcement learning agent within a 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]
```

See Also

Topics
“Reinforcement Learning Agents”

Introduced in R2019a

rlStochasticActorRepresentation

Stochastic actor representation for reinforcement learning agents

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 Policy and Value Function Representations".

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 `actor` respectively to the inputs `observationInfo` and `actionInfo`.

`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 non-negative) 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 variances (which must be non-negative) of the Gaussian distributions for the dimensions of the action space.

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” (Deep Learning Toolbox). For more information on creating deep neural networks for reinforcement learning, see “Create Policy and Value Function Representations”.

`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.

`ObservationInfo` — Observation specifications
specification object | array of specification objects

Observation specifications, a reinforcement learning specification object or an array of specification objects defining properties such as dimensions, data type, and names of the observation signals.

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
specification object

Action specifications, a reinforcement learning specification object defining properties such as 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 using `rlFiniteSetSpec` or `rlNumericSpec`.

For custom basis function representations, the action signal must be a scalar, a column vector, or a discrete action.

Object Functions

rlACAgent	Actor-critic reinforcement learning agent
rlPGAgent	Policy gradient reinforcement learning agent
rlPPOAgent	Proximal policy optimization reinforcement learning agent
getAction	Obtain action from agent or actor representation 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-dimensional vector (the observation vector just defined by `obsInfo`), and its output (here called `actionProb`) must be a three-dimensional 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 = [  imageInputLayer([4 1 1], '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 four-dimensional space, so that a single observation is a column vector containing 6 doubles.

```
obsInfo = rlNumericSpec([6 1]);
```

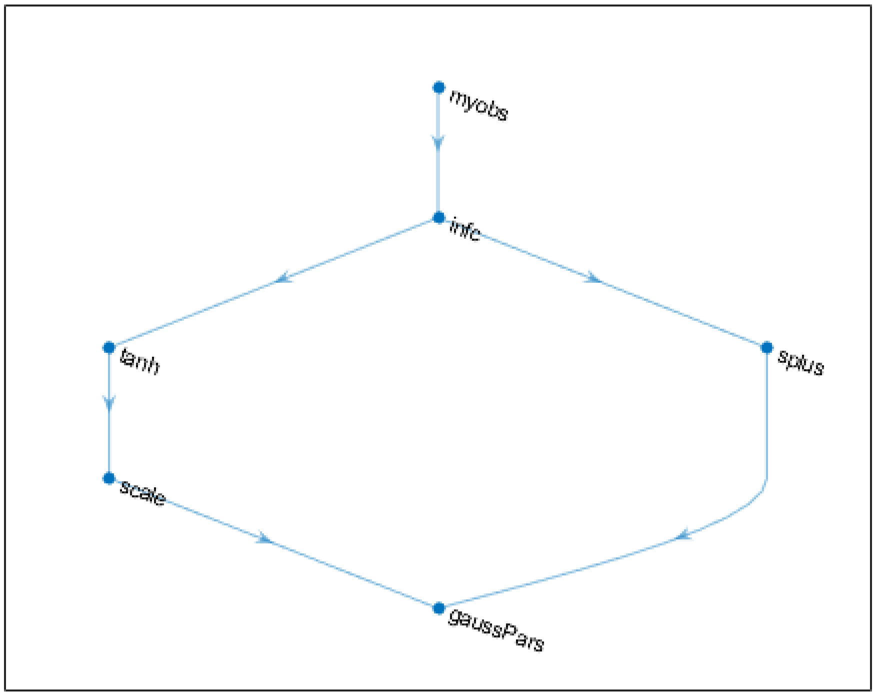
Create an action specification object (or alternatively use `getActionInfo` to extract the specification object from an environment). For this example, define the action space as a continuous two-dimensional space, so that a single action is a column vector containing 2 doubles both between `-10` and `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 variances of every action. The network has one path for the mean value (which is scaled to the output range) and another path for the variance (where a softplus layer enforces non-negativity).

```
% observation 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');
             scalingLayer('Name','scale','Scale',actInfo.UpperLimit) ];
```

```
% path layers for variance (2 by 1 input and output)
% using softplus layer to make it non negative)
variancePath =  softplusLayer('Name', 'splus');

% conctatenate two inputs (along dimension #3) to form a single (4 by 1) output layer
outLayer = concatenationLayer(3,2,'Name','gaussPars');

% add layers to network object
net = layerGraph(inPath);
net = addLayers(net,meanPath);
net = addLayers(net,variancePath);
net = addLayers(net,outLayer);

% connect layers
net = connectLayers(net,'infc','tanh/in');              % connect output of inPath to meanPath in
net = connectLayers(net,'infc','splus/in');             % connect output of inPath to variancePat
net = connectLayers(net,'scale','gaussPars/in1');       % connect output of meanPath to gaussPars
net = connectLayers(net,'splus','gaussPars/in2');       % connect output of variancePath to gauss

% plot network (the output obtained by concatenation is a 2+2
plot(net)
```



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','myobs')
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`, or `rlPGAgent` 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: [1×1 rl.util.rlFiniteSetSpec]
    ObservationInfo: [1×1 rl.util.rlNumericSpec]
            Options: [1×1 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 = 1×1 cell array
    {[7]}
```

You can now use the actor (along with an critic) to create a suitable discrete action space agent.

Create Stochastic Actor from 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);
```

See Also

Functions
`getActionInfo` | `getObservationInfo` | `rlRepresentationOptions`

Topics
"Create Policy and Value Function Representations"
"Reinforcement Learning Agents"

Introduced in R2020a

rlTable

Value table or Q table

Description

Value tables and Q tables are one way to represent critic networks for reinforcement learning. 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 representation using an `rlTable` object, use an `rlValueRepresentation` or `rlQValueRepresentation` 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

rlValueRepresentation	Value function critic representation for reinforcement learning agents
rlQValueRepresentation	Q-Value function critic representation for reinforcement learning agents

Examples

Create a Value Table

This example shows how to use `rlTable` to create a value table. You can use such a table to represent the critic of an actor-critic agent with a finite observation space.

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]
```

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]
```

See Also

Topics
"Create Policy and Value Function Representations"

Introduced in R2019a

rlTD3Agent

Twin-delayed deep deterministic policy gradient reinforcement learning agent

Description

The twin-delayed 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.

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 Policy Gradient Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

```
agent = rlTD3Agent(actor,critics,agentOptions)
```

Description

`agent = rlTD3Agent(actor,critics,agentOptions)` creates an agent with the specified actor and critic representations and sets the `AgentOptions` property. To create a:

- TD3 agent, specify a two-element row vector of critic representations.
- Delayed DDPG agent, specify a single critic representation.

Input Arguments

`actor` — Actor network representation
`rlDeterministicActorRepresentation` object

Actor network representation, specified as an `rlDeterministicActorRepresentation` object. For more information on creating actor representations, see “Create Policy and Value Function Representations”.

`critics` — Critic network representations
`rlQValueRepresentation` object | two-element row vector of `rlQValueRepresentation` objects

Critic network representations, specified as one of the following:

- `rlQValueRepresentation` 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 `rlQValueRepresentation` objects — Create a TD3 agent with two critic value functions. The two critic networks must be unique `rlQValueRepresentation` objects with the same observation and action specifications. The representations can either have the different structures or the same structure but with different initial parameters.

For more information on creating critic representations, see "Create Policy and Value Function Representations".

Properties

`AgentOptions` — Agent options
`rlTD3AgentOptions` object

Agent options, specified as an `rlTD3AgentOptions` object.

`ExperienceBuffer` — Experience buffer
`ExperienceBuffer` object

Experience buffer, specified as an `ExperienceBuffer` object. During training the agent stores each of its experiences (*S*,*A*,*R*,*S*') 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*.

For more information on how the agent samples experience from the buffer during training, see "Twin-Delayed Deep Deterministic Policy Gradient Agents".

Object Functions

train	Train a reinforcement learning agent within a specified environment
sim	Simulate a trained reinforcement learning agent within a specified environment
getActor	Get actor representation from reinforcement learning agent
setActor	Set actor representation of reinforcement learning agent
getCritic	Get critic representation from reinforcement learning agent
setCritic	Set critic representation of reinforcement learning agent
generatePolicyFunction	Create function that evaluates trained policy of reinforcement learning agent

Examples

Create TD3 Agent

Create environment and obtain observation and action specifications.

```
env = rlPredefinedEnv("DoubleIntegrator-Continuous");
obsInfo = getObservationInfo(env);
numObs = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
numAct = numel(actInfo);
```

Create two Q-value critic representations. First, create a critic deep neural network structure.

```
statePath1 = [
    imageInputLayer([numObs 1 1],'Normalization','none','Name','observation')
    fullyConnectedLayer(400,'Name','CriticStateFC1')
    reluLayer('Name','CriticStateRelu1')
    fullyConnectedLayer(300,'Name','CriticStateFC2')
    ];
actionPath1 = [
    imageInputLayer([numAct 1 1],'Normalization','none','Name','action')
    fullyConnectedLayer(300,'Name','CriticActionFC1')
    ];
commonPath1 = [
    additionLayer(2,'Name','add')
    reluLayer('Name','CriticCommonRelu1')
    fullyConnectedLayer(1,'Name','CriticOutput')
    ];

criticNet = layerGraph(statePath1);
criticNet = addLayers(criticNet,actionPath1);
criticNet = addLayers(criticNet,commonPath1);
criticNet = connectLayers(criticNet,'CriticStateFC2','add/in1');
criticNet = connectLayers(criticNet,'CriticActionFC1','add/in2');
```

Critic the critic representations. Use the same network structure ofr both critics. The TD3 agent initializes the two networks using different default parameters.

```
criticOptions = rlRepresentationOptions('Optimizer','adam','LearnRate',1e-3,...
                                        'GradientThreshold',1,'L2RegularizationFactor',2e-4);
critic1 = rlQValueRepresentation(criticNet,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
critic2 = rlQValueRepresentation(criticNet,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
```

Create an actor deep neural network.

```
actorNet = [
    imageInputLayer([numObs 1 1],'Normalization','none','Name','observation')
    fullyConnectedLayer(400,'Name','ActorFC1')
    reluLayer('Name','ActorRelu1')
    fullyConnectedLayer(300,'Name','ActorFC2')
    reluLayer('Name','ActorRelu2')
    fullyConnectedLayer(numAct,'Name','ActorFC3')
    tanhLayer('Name','ActorTanh1')
    ];
```

Create a deterministic actor representation.

```
actorOptions = rlRepresentationOptions('Optimizer','adam','LearnRate',1e-3,...
                                       'GradientThreshold',1,'L2RegularizationFactor',1e-5);
actor  = rlDeterministicActorRepresentation(actorNet,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'ActorTanh1'},actorOptions);
```

Specify agent options.

```
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;
```

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);
```

See Also
`rlTD3AgentOptions`

Topics
“Twin-Delayed Deep Deterministic Policy Gradient Agents”
“Reinforcement Learning Agents”
“Train Reinforcement Learning Agents”
“Train Biped Robot to Walk Using Reinforcement Learning Agents”

Introduced in R2020a

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 Policy Gradient Agents”.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents”.

Creation

Syntax

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 2-108 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 2-111.

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 variance of each action to a different value while using the same decay rate for both variances.

```
opt = rlTD3AgentOptions;
opt.ExplorationModel.Variance = [0.1 0.2];
opt.ExplorationModel.VarianceDecayRate = 1e-4;
```

`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 2-111.

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 variance of each action to a different value while using the same decay rate for both variances.

```
opt = rlTD3AgentOptions;
opt.TargetPolicySmoothModel.Variance = [0.1 0.2];
opt.TargetPolicySmoothModel.VarianceDecayRate = 1e-4;
```

`PolicyUpdateFrequency` — Number of steps between policy updates
2 (default) | positive integer

Number of steps between policy updates, specified as a positive integer.

`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` — Flag for clearing the experience buffer
`true` (default) | `false`

Flag for clearing the experience buffer before training, specified as a logical value.

`SaveExperienceBufferWithAgent` — Flag for saving the experience buffer
`false` (default) | `true`

Flag for saving the experience buffer data when saving the agent, specified as a logical value. This option applies both when saving candidate agents during training and when saving agents using the `save` function.

For some agents, such as those with a large experience buffer and image-based observations, the memory required for saving their experience buffer is large. In such cases, to not save the experience buffer data, set `SaveExperienceBufferWithAgent` to `false`.

If you plan to further train your saved agent, you can start training with the previous experience buffer as a starting point. In this case, set `SaveExperienceBufferWithAgent` to `true`.

`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 steps ahead
`1` (default) | positive integer

Number of steps to look ahead during training, specified as a positive integer.

`ExperienceBufferLength` — Experience buffer size
`10000` (default) | positive integer

Experience buffer size, specified as a positive integer. During training, the agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.

`SampleTime` — Sample time of agent
`1` (default) | positive scalar

Sample time of agent, specified as a positive scalar.

`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 policy gradient reinforcement learning agent

Examples

Create TD3 Agent Options Object

This example shows how to create a TD3 agent option 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
                     TargetSmoothFactor: 0.0050
                  TargetUpdateFrequency: 2
    ResetExperienceBufferBeforeTraining: 1
          SaveExperienceBufferWithAgent: 0
                          MiniBatchSize: 48
                    NumStepsToLookAhead: 1
                 ExperienceBufferLength: 10000
                             SampleTime: 1
                  DiscountFactor: 0.9900
```

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
`Mean`	Noise model mean
`Variance`	Noise model variance
`VarianceDecayRate`	Decay rate of the variance
`VarianceMin`	Minimum variance, which must be less than `Variance`
`LowerLimit`	Noise sample lower limit
`UpperLimit`	Noise sample upper limit

Gaussian noise is sampled as shown in the following code.

```
x = Mean + rand(ActionSize).*Variance
x = min(max(x,LowerLimit),UpperLimit);
```

At each sample time step, the variance decays as shown in the following code.

```
decayedVariance = Variance.*(1 - VarianceDecayRate);
Variance = max(decayedVariance,VarianceMin);
```

Ornstein-Uhlenbeck Action Noise

An `OrnsteinUhlenbeckActionNoise` object has the following numeric value properties.

Property	Description
`InitialAction`	Initial value of action for noise model
`Mean`	Noise model mean
`MeanAttractionConstant`	Constant specifying how quickly the noise model output is attracted to the mean
`Variance`	Noise model variance
`VarianceDecayRate`	Decay rate of the variance
`VarianceMin`	Minimum variance

At each sample time step, the noise model is updated using the following formula, where `Ts` is the agent sample time.

```
x(k) = x(k-1) + MeanAttractionConstant.*(Mean - x(k-1)).*Ts
       + Variance.*randn(size(Mean)).*sqrt(Ts)
```

At each sample time step, the variance decays as shown in the following code.

```
decayedVariance = Variance.*(1 - VarianceDecayRate);
Variance = max(decayedVariance,VarianceMin);
```

For continuous action signals, it is important to set the noise variance appropriately to encourage exploration. It is common to have `Variance*sqrt(Ts)` be 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 variance. Also, to increase exploration, you can reduce the `VarianceDecayRate`.

See Also

Topics
"Twin-Delayed Deep Deterministic Policy Gradient Agents"

Introduced in R2020a

rlTrainingOptions

Options for training reinforcement learning agents

Description

Use an `rlTrainingOptions` object to specify training options for an agent. To train an agent, use `train`.

For more information on agents training and simulation, see “Train Reinforcement Learning Agents”.

Creation

Syntax

```
trainOpts = rlTrainingOptions
opt = rlTrainingOptions(Name,Value)
```

Description

`trainOpts = rlTrainingOptions` returns the default options for training a reinforcement learning agent. Use training options to specify parameters about the training session such as the maximum number of episodes to train, criteria for stopping training, criteria for saving agents, and criteria for using parallel computing. After configuring the options, use `trainOpts` as an input argument for `train`.

`opt = rlTrainingOptions(Name,Value)` creates a training options set with the specified “Properties” on page 2-113 using one or more name-value pair 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 the comma-separated pair consisting of `'MaxEpisodes'` and a positive integer. Regardless of other criteria for termination, training terminates after this many episodes.

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 the comma-separated pair consisting of `'MaxStepsPerEpisode'` 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 episode if those termination conditions are not met.

Example: `'MaxStepsPerEpisode',1000`

`ScoreAveragingWindowLength` — Window length for averaging

5 (default) | positive integer

Window length for averaging scores, rewards, and numbers of steps, specified as the comma-separated pair consisting of `'ScoreAveragingWindowLength'` and a positive integer. For options expressed in terms of averages, this is the number of episodes included in the average. For instance suppose that `StopTrainingCriteria` is `"AverageReward"`, and `StopTrainingValue` is 500. Training terminates when the reward averaged over the number of episodes specified by this parameter is 500 or greater.

Example: `'ScoreAveragingWindowLength',10`

`StopTrainingCriteria` — Training termination condition

`"AverageSteps"` (default) | `"AverageReward"` | `"EpisodeCount"` | ...

Training termination condition, specified as the comma-separated pair consisting of `'StopTrainingCriteria'` and 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

Critical value of training termination condition, specified as the comma-separated pair consisting of `'StopTrainingValue'` and a scalar. Training terminates when the termination condition specified by the `StopTrainingCriteria` option equals or exceeds this value. For instance, if `StopTrainingCriteria` is `"AverageReward"`, and `StopTrainingValue` is 100, then 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 agent during training

`"none"` (default) | `"EpisodeReward"` | `"AverageReward"` | `"EpisodeCount"` | ...

Condition for saving agent during training, specified as the comma-separated pair consisting of `'SaveAgentCriteria'` and one of the following strings:

- `"none"` — Do not save any agents during training.
- `"EpisodeReward"` — Save agent when the reward in the current episode equals or exceeds the critical value.

- `"AverageSteps"` — Save 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 agent when the running average reward over all episodes equals or exceeds the critical value.
- `"GlobalStepCount"` — Save 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 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 agent
`"none"` (default) | 500 | scalar

Critical value of condition for saving agent, specified as the comma-separated pair consisting of `'SaveAgentValue'` and `"none"` or a numeric scalar.

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 `SaveAgentValue` option for more details.

Example: `'SaveAgentValue',100`

`SaveAgentDirectory` — Folder for saved agents
`"savedAgents"` (default) | string | character vector

Folder for saved agents, specified as the comma-separated pair consisting of `'SaveAgentDirectory'` and 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 agent in a MAT-file in this folder. If the folder doesn't 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 the comma-separated pair consisting of `'UseParallel'` and either `true` or `false`. Setting this option to `true` configures training to use parallel computing. To specify options for parallel training, use the `ParallelizationOptions` property.

Using parallel computing requires Parallel Computing Toolbox software.

For more information about training using parallel computing, see “Train Reinforcement Learning Agents”.

Example: `'UseParallel',true`

`ParallelizationOptions` — Options to control parallel training
`ParallelTraining` object

Parallelization options to control parallel training, specified as the comma-separated pair consisting of `'ParallelizationOptions'` and 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.
- `"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.

`DataToSendFromWorkers` — Type of data that workers send to the host
`"experiences"` (default) | `"gradients"`

Type of data that workers send to the host, specified as one of the following strings:

- `"experiences"` — Send experience data (observation, action, reward, next observation, is done) to the host. For agents with gradients, the host computes gradients from the experiences.
- `"gradients"` — Compute and send gradients to the host. The host applies gradients to update networks parameters.

Note AC and PG agents accept only `DataToSendFromWorkers = "gradients"`. DQN and DDPG agents accept only `DataToSendFromWorkers = "experiences"`.

`StepsUntilDataIsSent` — When workers send data to host
`–1` (default) | positive integer

When workers send data to host and receive updated parameters, specified as `–1` or a positive integer. This number indicates how many steps to compute during the episode before sending data to the host. When this option is –1, the worker waits until the end of the episode and then sends all step data to the host. Otherwise, the worker waits the specified number of steps before sending data.

Note

- AC agents do not accept `StepUntilDataIsSent = -1`. For A3C training, set `StepUntilDataIsSent` equal to the `NumStepToLookAhead` AC agent option.
- PG agents accept only `StepUntilDataIsSent = -1`.

`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 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.

`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.

`Verbose` — Display training progress on the command line

`false` (0) (default) | `true` (1)

Display training progress on 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.

`StopOnError` — Stop training when error occurs

`"on"` (default) | `"off"`

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.

`Plots` — Display training progress with the Episode Manager
`"training-progress"` (default) | `"none"`

Display training progress with the 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"`.

Object Functions

train	Train a reinforcement learning agent 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 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 the Reinforcement Learning Episode Manager for displaying training results. 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.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',1000,...
    'MaxStepsPerEpisode',1000,...
    'StopTrainingCriteria',"AverageReward",...
    'StopTrainingValue',480,...
    'Verbose',true,...
    'Plots',"training-progress")
```

```
trainOpts = 
  rlTrainingOptions with properties:

                   MaxEpisodes: 1000
            MaxStepsPerEpisode: 1000
    ScoreAveragingWindowLength: 5
          StopTrainingCriteria: "AverageReward"
             StopTrainingValue: 480
             SaveAgentCriteria: "none"
                SaveAgentValue: "none"
                   UseParallel: 0
        ParallelizationOptions: [1×1 rl.option.ParallelTraining]
            SaveAgentDirectory: "savedAgents"
                   StopOnError: "on"
                       Verbose: 1
                         Plots: "training-progress"
```

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
    ScoreAveragingWindowLength: 5
          StopTrainingCriteria: "AverageReward"
             StopTrainingValue: 480
             SaveAgentCriteria: "none"
                SaveAgentValue: "none"
                   UseParallel: 0
        ParallelizationOptions: [1×1 rl.option.ParallelTraining]
            SaveAgentDirectory: "savedAgents"
                   StopOnError: "on"
                       Verbose: 1
                         Plots: "training-progress"
```

You can now use `trainOpts` as an input argument to the `train` command.

See Also

Topics
"Reinforcement Learning Agents"

Introduced in R2019a

rlValueRepresentation

Value function critic representation for reinforcement learning agents

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` or `rlPGAgent`. For an example of this workflow, see “Create Actor and Critic Representations” on page 2-123. For more information on creating representations, see “Create Policy and Value Function Representations”.

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`.

`rlValueRepsentation` objects support recurrent deep neural networks.

For a list of deep neural network layers, see “List of Deep Learning Layers” (Deep Learning Toolbox). For more information on creating deep neural networks for reinforcement learning, see “Create Policy and Value Function Representations”.

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
specification object | array of specification objects

Observation specifications, a reinforcement learning specification object or an array of specification objects defining properties such as the dimensions, data types, and names of the observation signals. You can extract ObservationInfo from an existing environment or agent using `getObservationInfo`. You can also construct the specs manually using a specification command such as `rlFiniteSetSpec` or `rlNumericSpec`.

Object Functions

rlACAgent	Actor-critic reinforcement learning agent
rlPGAgent	Policy gradient reinforcement learning agent
rlPPOAgent	Proximal policy optimization reinforcement learning agent
getValue	Obtain estimated value function representation

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-dimensional 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 = [imageInputLayer([4 1 1], '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 = [
    imageInputLayer([numObservation 1 1],'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 = [
    imageInputLayer([4 1 1], '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: [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, using the current parameter vector.

```
v = getValue(critic,{[2 4 6 8]'})
```

```
v = 
  1×1 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);
```

See Also

Functions
`getActionInfo` | `getObservationInfo` | `rlRepresentationOptions`

Topics
"Create Policy and Value Function Representations"
"Reinforcement Learning Agents"

Introduced in R2020a

scalingLayer

Scaling layer for actor or critic network

Description

A `ScalingLayer` is a deep neural network layer that 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 2-128 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 with the same dimensions as the input array — Specify different scale factors for each element of the input array.

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 with the same dimensions as the input array — Specify a different bias for each element of the input array.

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

  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 and that you want to apply a different scaling factor and bias to each out using a `scalingLayer`. Since the `tanhLayer` outputs its channels along the third dimension, the scale and bias must be 1-by-1-by-3 arrays.

```
scale = reshape([2.5 0.4 10],[1 1 3]);
bias = reshape([5 0 -50],[1 1 3]);
```

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 = reshape([10 10 10],[1 1 3]);
predict(sLayer,testData)
```

```
ans = 
ans(:,:,1) =

    30


ans(:,:,2) =

     4


ans(:,:,3) =

    50
```

See Also

`quadraticLayer` | `softplusLayer`

Topics

"Train DDPG Agent to Swing Up and Balance Pendulum"
"Create Policy and Value Function Representations"

Introduced in R2019a

softplusLayer

Softplus layer for actor or critic network

Description

A `SoftplusLayer` is a deep neural network layer that implements the softplus activation $Y = \log(1 + e^X)$, 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 2-131 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.

```
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'

  Show all properties
```

You can incorporate `sLayer` into an actor network for reinforcement learning.

See Also

`quadraticLayer` | `scalingLayer`

Topics

"Create Policy and Value Function Representations"

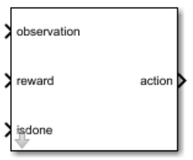
Introduced in R2020a

3

Blocks

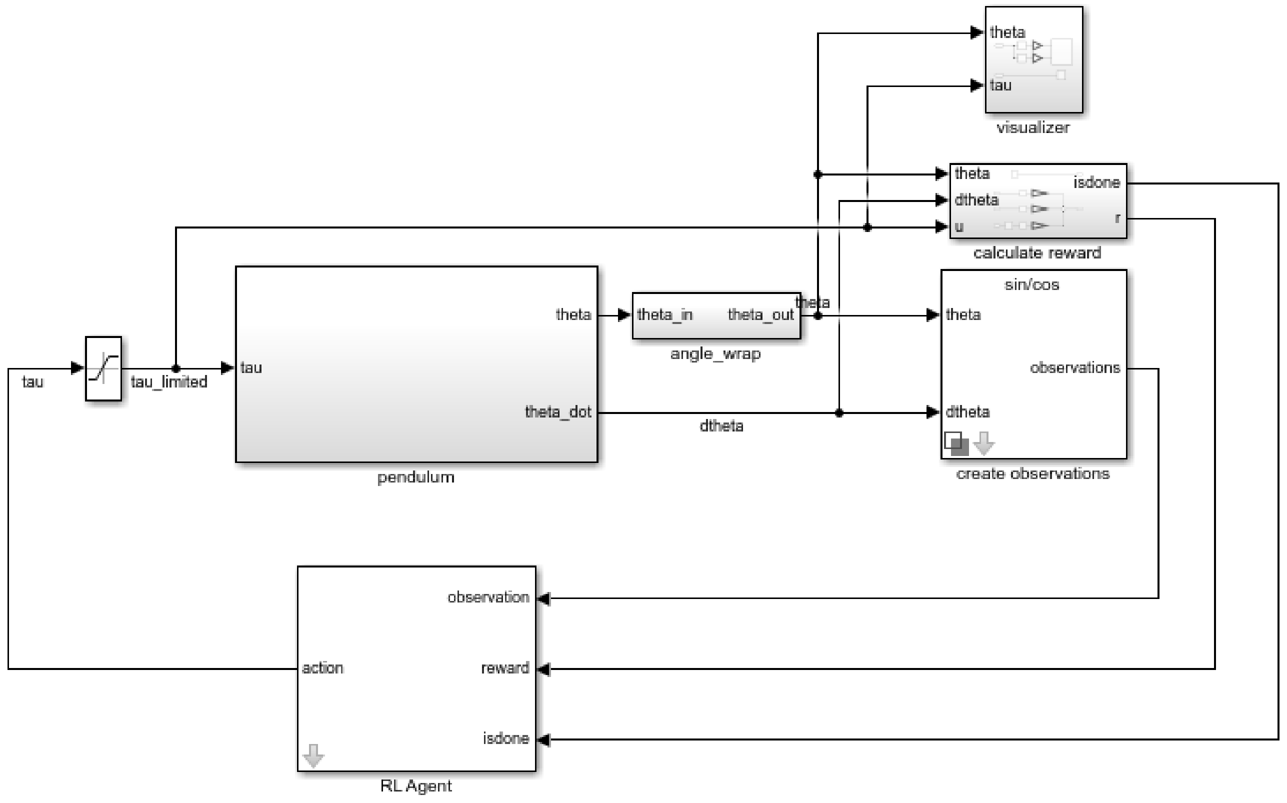
RL Agent

Reinforcement learning agent
Library: 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 as an agent object 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` 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 Environments for Reinforcement Learning". 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.

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`.

`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 **Provide cumulative reward signal** parameter.

Parameters

`Agent object` — Agent to train
`agent` (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”.

`Provide cumulative reward signal` — Add cumulative reward output port
`off` (default) | `on`

Enable the `cumulative_reward` block output by selecting this parameter.

See Also

`bus2RLSpec` | `createIntegratedEnv`

Topics
“Create Simulink Environments for Reinforcement Learning”
“Create Simulink Environment and Train Agent”

Introduced in R2019a