Reinforcement Learning Toolbox™

User's Guide

MATLAB®

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

Contents

Create Agents

3

Define Policies and Value Functions

4

5

Deploy Trained Policies

6

Getting Started

Reinforcement Learning Toolbox Product Description

Design and train policies using reinforcement learning

Reinforcement Learning Toolbox™ provides functions and blocks for training policies using reinforcement learning algorithms including DQN, A2C, and DDPG. You can use these policies to implement controllers and decision-making algorithms for complex systems such as robots and autonomous systems. You can implement the policies using deep neural networks, polynomials, or look-up tables.

The toolbox lets you train policies by enabling them to interact with environments represented by MATLAB® or Simulink® models. You can evaluate algorithms, experiment with hyperparameter settings, and monitor training progress. To improve training performance, you can run simulations in parallel on the cloud, computer clusters, and GPUs (with Parallel Computing Toolbox™ and MATLAB Parallel Server™).

Through the ONNX™ model format, existing policies can be imported from deep learning frameworks such as TensorFlow™ Keras and PyTorch (with Deep Learning Toolbox™). You can generate optimized C, C++, and CUDA code to deploy trained policies on microcontrollers and GPUs.

The toolbox includes reference examples for using reinforcement learning to design controllers for robotics and automated driving applications.

What Is Reinforcement Learning?

Reinforcement learning is a goal-directed computational approach where a computer learns to perform a task by interacting with an unknown dynamic environment. This learning approach enables a computer to make a series of decisions to maximize the cumulative reward for the task without human intervention and without being explicitly programmed to achieve the task. The following diagram shows a general representation of a reinforcement learning scenario.
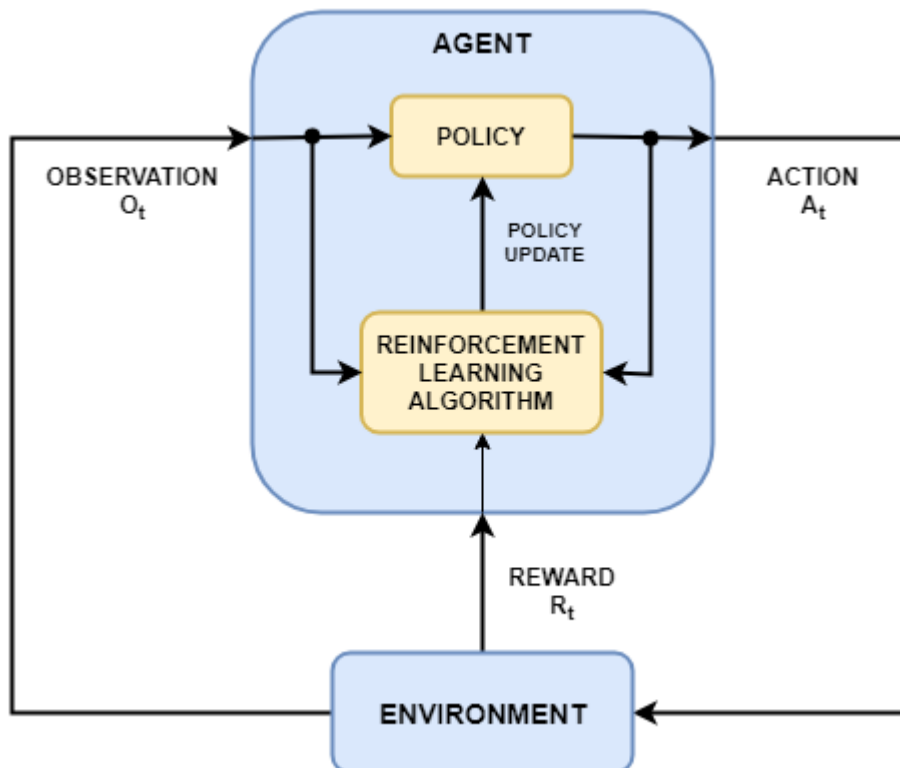
The goal of reinforcement learning is to train an *agent* to complete a task within an unknown *environment*. The agent receives *observations* and a *reward* from the environment and sends *actions* to the environment. The reward is a measure of how successful an action is with respect to completing the task goal.

The agent contains two components: a *policy* and a *learning algorithm*.

- The policy is a mapping that selects actions based on the observations from the environment. Typically, the policy is a function approximator with tunable parameters, such as a deep neural network.
- The learning algorithm continuously updates the policy parameters based on the actions, observations, and reward. The goal of the learning algorithm is to find an optimal policy that maximizes the cumulative reward received during the task.

In other words, reinforcement learning involves an agent learning the optimal behavior through repeated trial-and-error interactions with the environment without human involvement.

As an example, consider the task of parking a vehicle using an automated driving system. The goal of this task is for the vehicle computer (**agent**) to park the vehicle in the correct position and

orientation. To do so, the controller uses readings from cameras, accelerometers, gyroscopes, a GPS receiver, and lidar (**observations**) to generate steering, braking, and acceleration commands (**actions**). The action commands are sent to the actuators that control the vehicle. The resulting observations depend on the actuators, sensors, vehicle dynamics, road surface, wind, and many other less-important factors. All these factors, that is, everything that is not the agent, make up the **environment** in reinforcement learning.

To learn how to generate the correct actions from the observations, the computer repeatedly tries to park the vehicle using a trial-and-error process. To guide the learning process, you provide a signal that is one when the car successfully reaches the desired position and orientation and zero otherwise (**reward**). During each trial, the computer selects actions using a mapping (**policy**) initialized with some default values. After each trial, the computer updates the mapping to maximize the reward (**learning algorithm**). This process continues until the computer learns an optimal mapping that successfully parks the car.

Reinforcement Learning Workflow

The general workflow for training an agent using reinforcement learning includes the following steps.

1 **Formulate Problem** — Define the task for the agent to learn, including how the agent interacts with the environment and any primary and secondary goals the agent must achieve.

2 **Create Environment** — Define the environment within which the agent operates, including the interface between agent and environment and the environment dynamic model. For more information, see "Create MATLAB Environments for Reinforcement Learning" on page 2-2 and "Create Simulink Environments for Reinforcement Learning" on page 2-5.

3 **Define Reward** — Specify the reward signal that the agent uses to measure its performance against the task goals and how this signal is calculated from the environment. For more information, see "Define Reward Signals" on page 2-7.

4 **Create Agent** — Create the agent, which includes defining a policy representation and configuring the agent learning algorithm. For more information, see "Create Policy and Value Function Representations" on page 4-2 and "Reinforcement Learning Agents" on page 3-2.

5 **Train Agent** — Train the agent policy representation using the defined environment, reward, and agent learning algorithm. For more information, see "Train Reinforcement Learning Agents" on page 5-2.

6 **Validate Agent** — Evaluate the performance of the trained agent by simulating the agent and environment together. For more information, see "Train Reinforcement Learning Agents" on page 5-2.

7   **Deploy Policy** — Deploy the trained policy representation using, for example, generated GPU code. For more information, see "Deploy Trained Reinforcement Learning Policies" on page 6-2.

Training an agent using reinforcement learning is an iterative process. Decisions and results in later stages can require you to return to an earlier stage in the learning workflow. For example, if the training process does not converge to an optimal policy within a reasonable amount of time, you might have to update any of the following before retraining the agent:

- Training settings
- Learning algorithm configuration
- Policy representation
- Reward signal definition
- Action and observation signals
- Environment dynamics

See Also

More About

- "Reinforcement Learning for Control Systems Applications" on page 1-6
- "Create Simulink Environment and Train Agent" on page 1-19

Reinforcement Learning for Control Systems Applications

The behavior of a reinforcement learning policy—that is, how the policy observes the environment and generates actions to complete a task in an optimal manner—is similar to the operation of a controller in a control system. Reinforcement learning can be translated to a control system representation using the following mapping.
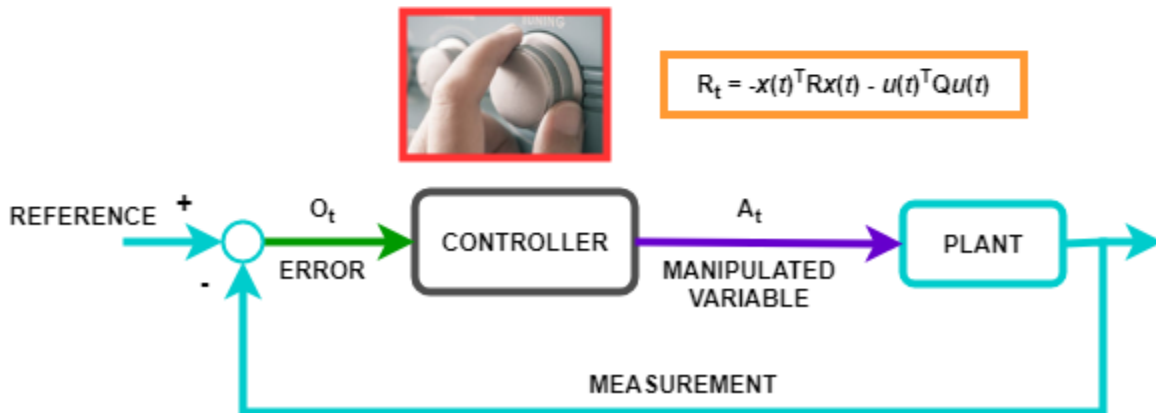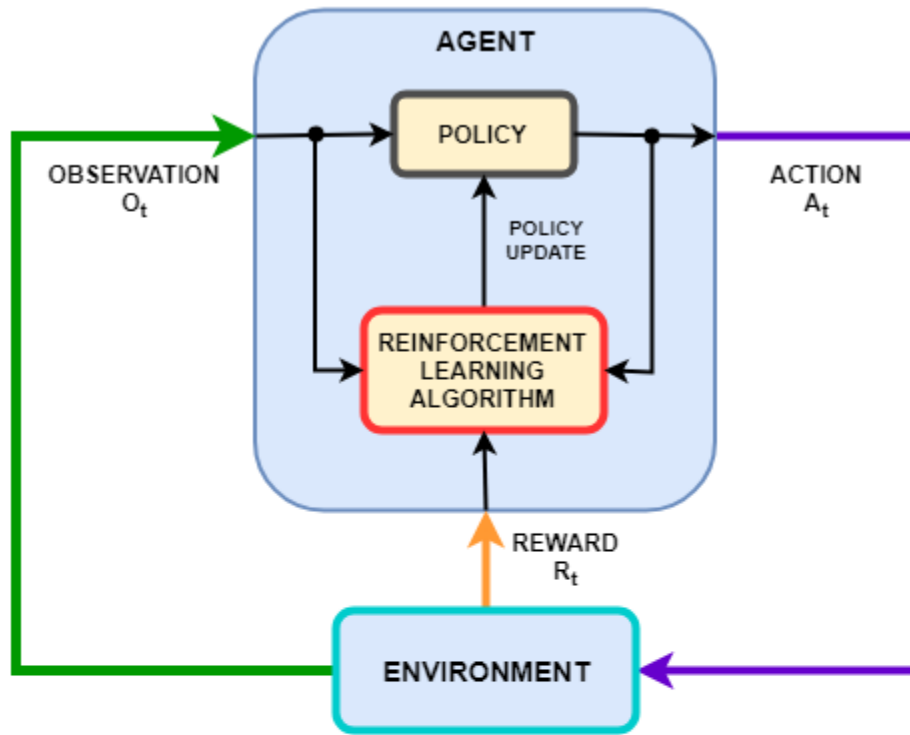
Reinforcement Learning	Control Systems
Policy	Controller

Reinforcement Learning	Control Systems
Environment	Everything that is not the controller — In the preceding diagram, the environment includes the plant, the reference signal, and the calculation of the error. In general, the environment can also include additional elements, such as: • Measurement noise • Disturbance signals • Filters • Analog-to-digital and digital-to-analog converters
Observation	Any measurable value from the environment that is visible to the agent — In the preceding diagram, the controller can see the error signal from the environment. You can also create agents that observe, for example, the reference signal, measurement signal, and measurement signal rate of change.
Action	Manipulated variables or control actions
Reward	Function of the measurement, error signal, or some other performance metric — For example, you can implement reward functions that minimize the steady-state error while minimizing control effort.
Learning Algorithm	Adaptation mechanism of an adaptive controller

Many control problems encountered in areas such as robotics and automated driving require complex, nonlinear control architectures. Techniques such as gain scheduling, robust control, and nonlinear model predictive control (MPC) can be used for these problems, but often require significant domain expertise from the control engineer. For example, gains and parameters are difficult to tune. The resulting controllers can pose implementation challenges, such as the computational intensity of nonlinear MPC.

You can use deep neural networks, trained using reinforcement learning, to implement such complex controllers. These systems can be self-taught without intervention from an expert control engineer. Also, once the system is trained, you can deploy the reinforcement learning policy in a computationally efficient way.

You can also use reinforcement learning to create an end-to-end controller that generates actions directly from raw data, such as images. This approach is attractive for video-intensive applications, such as automated driving, since you do not have to manually define and select image features.

See Also

More About

Train Reinforcement Learning Agent in Basic Grid World

This example shows how to solve a grid world environment using reinforcement learning by training Q-learning and SARSA agents. For more information on these agents, see "Q-Learning Agents" on page 3-6 and "SARSA Agents" on page 3-8.

This grid world environment has the following configuration and rules:

1   The grid world is 5-by-5 and bounded by borders, with four 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 a 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 a reward of +5.

5   The agent is blocked by obstacles (black cells).

6   All other actions result in –1 reward.
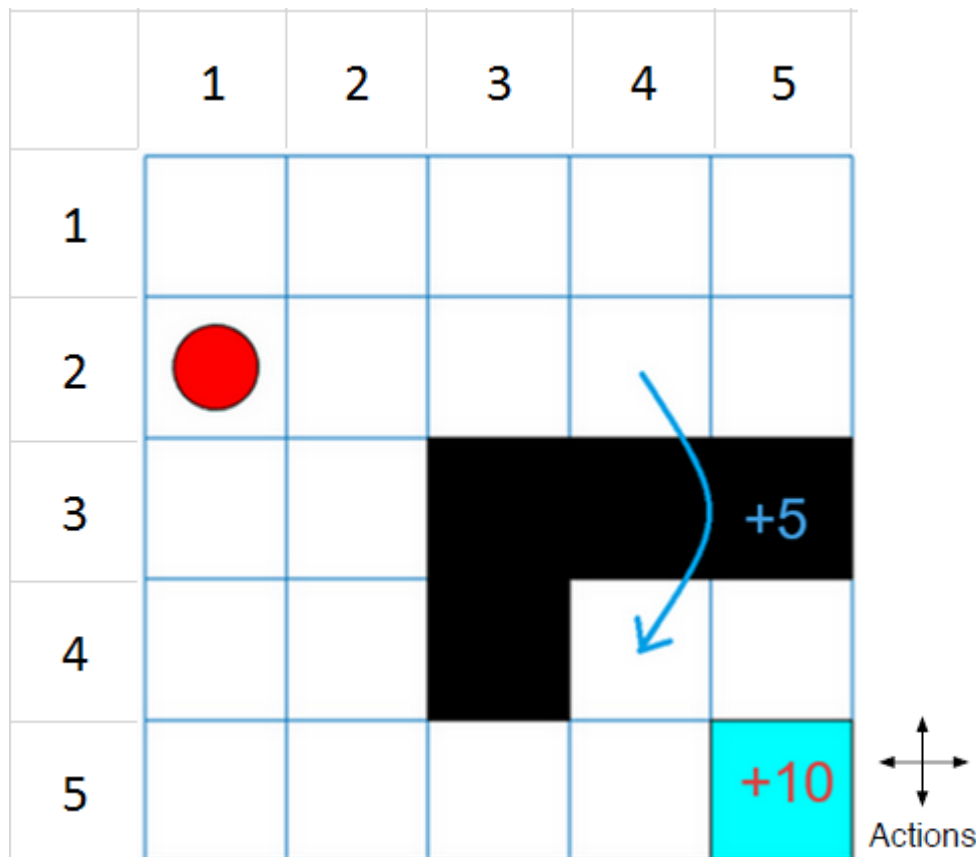
Create Grid World Environment

Create the basic grid world environment.

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

To specify that the initial state of the agent is always [2,1], create a reset function that returns the state number for the initial agent state. This function is called at the start of each training episode

and simulation. States are numbered starting at position [1,1]. The state number increases as you move down the first column and then down each subsequent column. Therefore, create an anonymous function handle that sets the initial state to 2.

```
env.ResetFcn = @() 2;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create Q-Learning Agent

To create a Q-learning agent, first create a Q table using the observation and action specifications from the grid world environment. Set the learning rate of the representation to 1.

```
qTable = rlTable(getObservationInfo(env),getActionInfo(env));
qRepresentation = rlQValueRepresentation(qTable,getObservationInfo(env),getActionInfo(env));
qRepresentation.Options.LearnRate = 1;
```

Next, create a Q-learning agent using this table representation and configure the epsilon-greedy exploration. For more information on creating Q-learning agents, see `rlQAgent` and `rlQAgentOptions`.

```
agentOpts = rlQAgentOptions;
agentOpts.EpsilonGreedyExploration.Epsilon = .04;
qAgent = rlQAgent(qRepresentation,agentOpts);
```

Train Q-Learning Agent

To train the agent, first specify the training options. For this example, use the following options:

- Train for at most 200 episodes. Specify that each episode lasts for most 50 time steps.
- Stop training when the agent receives an average cumulative reward greater than 10 over 30 consecutive episodes.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions;
trainOpts.MaxStepsPerEpisode = 50;
trainOpts.MaxEpisodes= 200;
trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = 11;
trainOpts.ScoreAveragingWindowLength = 30;
```

Train the Q-learning agent using the `train` function. Training can take several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(qAgent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('basicGWQAgent.mat','qAgent')
end
```

The **Episode Manager** window opens and displays the training progress.
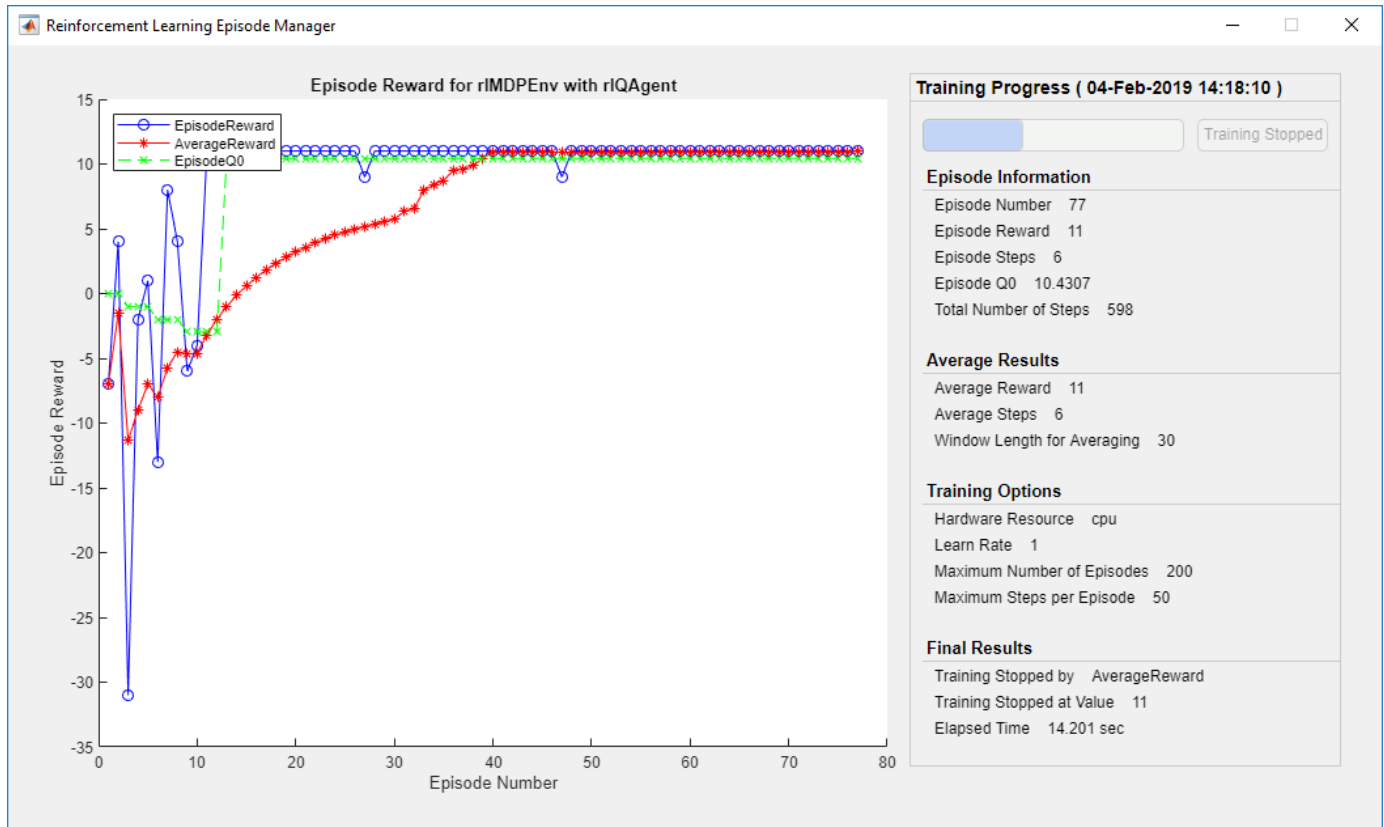
Validate Q-Learning Results

To validate the training results, simulate the agent in the training environment.

Before running the simulation, visualize the environment and configure the visualization to maintain a trace of the agent states.

```
plot(env)
env.Model.Viewer.ShowTrace = true;
env.Model.Viewer.clearTrace;
```

Simulate the agent in the environment using the `sim` function.

```
sim(qAgent,env)
```
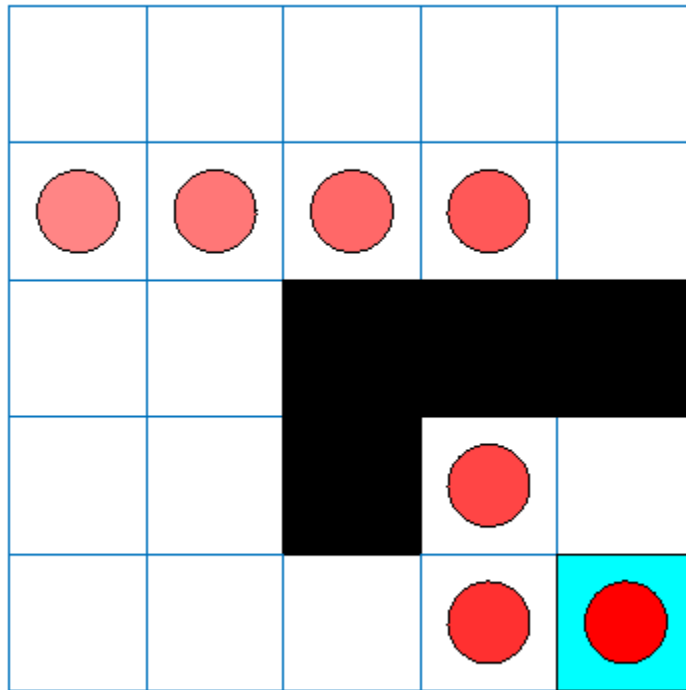
The agent trace shows that the agent successfully finds the jump from cell [2,4] to cell [4,4].

Create and Train SARSA Agent

To create a SARSA agent, use the same Q table representation and epsilon-greedy configuration as for the Q-learning agent. For more information on creating SARSA agents, see `rlSARSAAgent` and `rlSARSAAgentOptions`.

```
agentOpts = rlSARSAAgentOptions;
agentOpts.EpsilonGreedyExploration.Epsilon = 0.04;
sarsaAgent = rlSARSAAgent(qRepresentation,agentOpts);
```

Train the SARSA agent using the `train` function. Training can take several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(sarsaAgent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('basicGWSarsaAgent.mat','sarsaAgent')
end
```
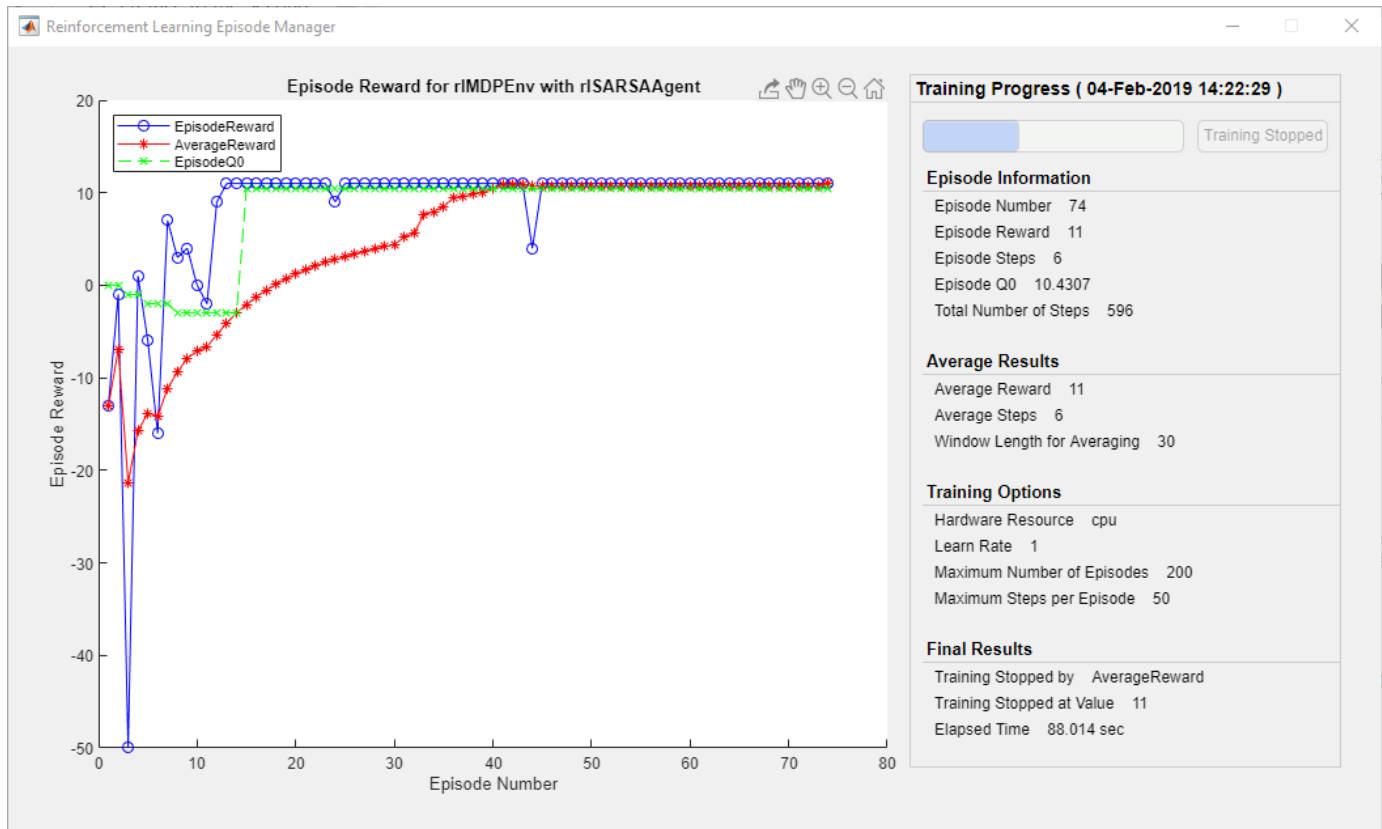
Validate SARSA Training

To validate the training results, simulate the agent in the training environment.

```
plot(env)
env.Model.Viewer.ShowTrace = true;
env.Model.Viewer.clearTrace;
```

Simulate the agent in the environment.

```
sim(sarsaAgent,env)
```
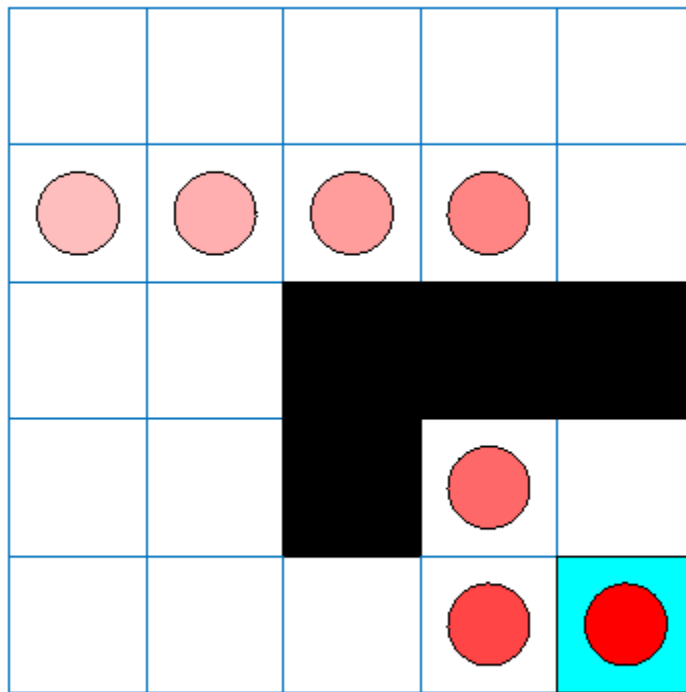
The SARSA agent finds the same grid world solution as the Q-learning agent.

See Also
`createGridWorld` | `rlMDPEnv`

More About
- "Reinforcement Learning Agents" on page 3-2
- "Train Reinforcement Learning Agents" on page 5-2

Train Reinforcement Learning Agent in MDP Environment

This example shows how to train a Q-learning agent to solve a generic Markov decision process (MDP) environment. For more information on these agents, see "Q-Learning Agents" on page 3-6.

The MDP environment has the following graph.
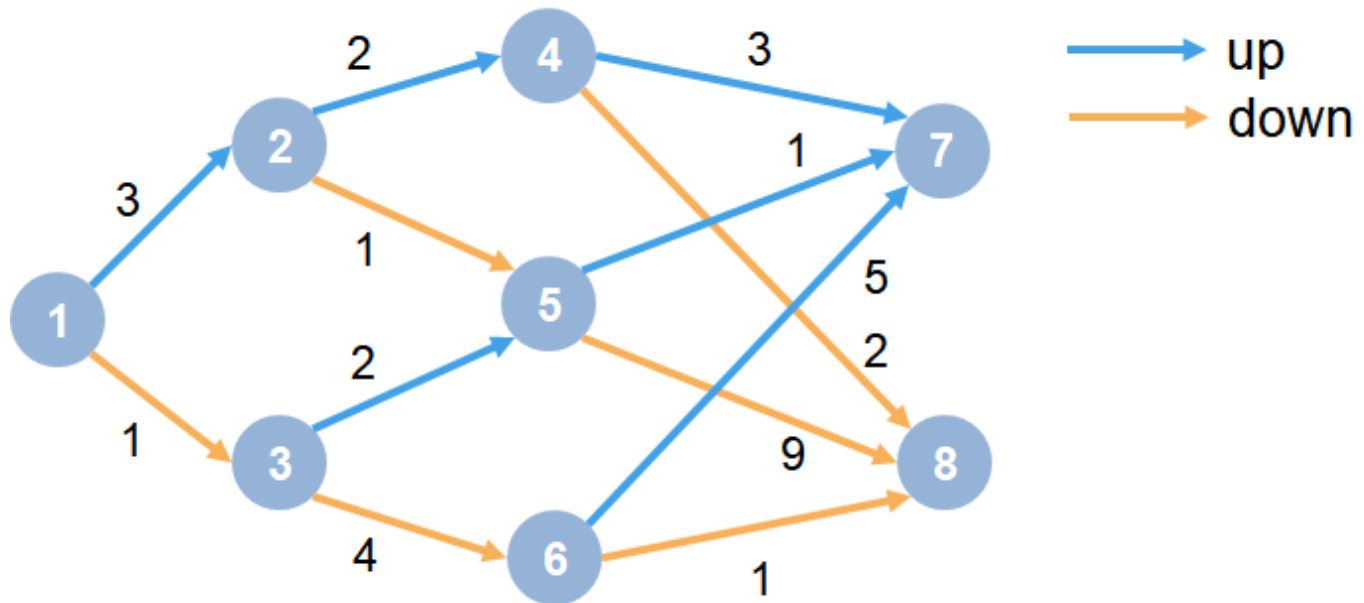
Here:

1   Each circle represents a state.
2   At each state there is a decision to go up or down.
3   The agent begins from state 1.
4   The agent receives a reward equal to the value on each transition in the graph.
5   The training goal is to collect the maximum cumulative reward.

Create MDP Environment

Create an MDP model with eight states and two actions ("up" and "down").

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

To model the transitions from the above graph, modify the state transition matrix and reward matrix of the MDP. By default, these matrices contain zeros. For more information on creating an MDP model and the properties of an MDP object, see `createMDP`.

Specify the state transition and reward matrices for the MDP. For example, in the following commands:

- The first two lines specify the transition from state 1 to state 2 by taking action 1 ("up") and a reward of +3 for this transition.
- The next two lines specify the transition from state 1 to state 3 by taking action 2 ("down") and a reward of +1 for this transition.

```
MDP.T(1,2,1) = 1;
MDP.R(1,2,1) = 3;
MDP.T(1,3,2) = 1;
MDP.R(1,3,2) = 1;
```

Similarly, specify the state transitions and rewards for the remaining rules in the graph.

```
% 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 states "s7" and "s8" as terminal states of the MDP.

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

Create the reinforcement learning MDP environment for this process model.

```
env = rlMDPEnv(MDP);
```

To specify that the initial state of the agent is always state 1, specify a reset function that returns the initial agent state. This function is called at the start of each training episode and simulation. Create an anonymous function handle that sets the initial state to 1.

```
env.ResetFcn = @() 1;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create Q-Learning Agent

To create a Q-learning agent, first create a Q table using the observation and action specifications from the MDP environment. Set the learning rate of the representation to 1.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
qTable = rlTable(obsInfo, actInfo);
qRepresentation = rlQValueRepresentation(qTable, obsInfo, actInfo);
qRepresentation.Options.LearnRate = 1;
```

Next, create a Q-learning agent using this table representation, configuring the epsilon-greedy exploration. For more information on creating Q-learning agents, see `rlQAgent` and `rlQAgentOptions`.

```
agentOpts = rlQAgentOptions;
agentOpts.DiscountFactor = 1;
agentOpts.EpsilonGreedyExploration.Epsilon = 0.9;
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 0.01;
qAgent = rlQAgent(qRepresentation,agentOpts);
```

Train Q-Learning Agent

To train the agent, first specify the training options. For this example, use the following options:

- Train for at most 200 episodes, with each episode lasting at most 50 time steps.
- Stop training when the agent receives an average cumulative reward greater than 10 over 30 consecutive episodes.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions;
trainOpts.MaxStepsPerEpisode = 50;
trainOpts.MaxEpisodes = 200;
trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = 13;
trainOpts.ScoreAveragingWindowLength = 30;
```

Train the agent using the `train` function. This may take several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(qAgent,env,trainOpts);
else
    % Load pretrained agent for the example.
    load('genericMDPQAgent.mat','qAgent');
end
```
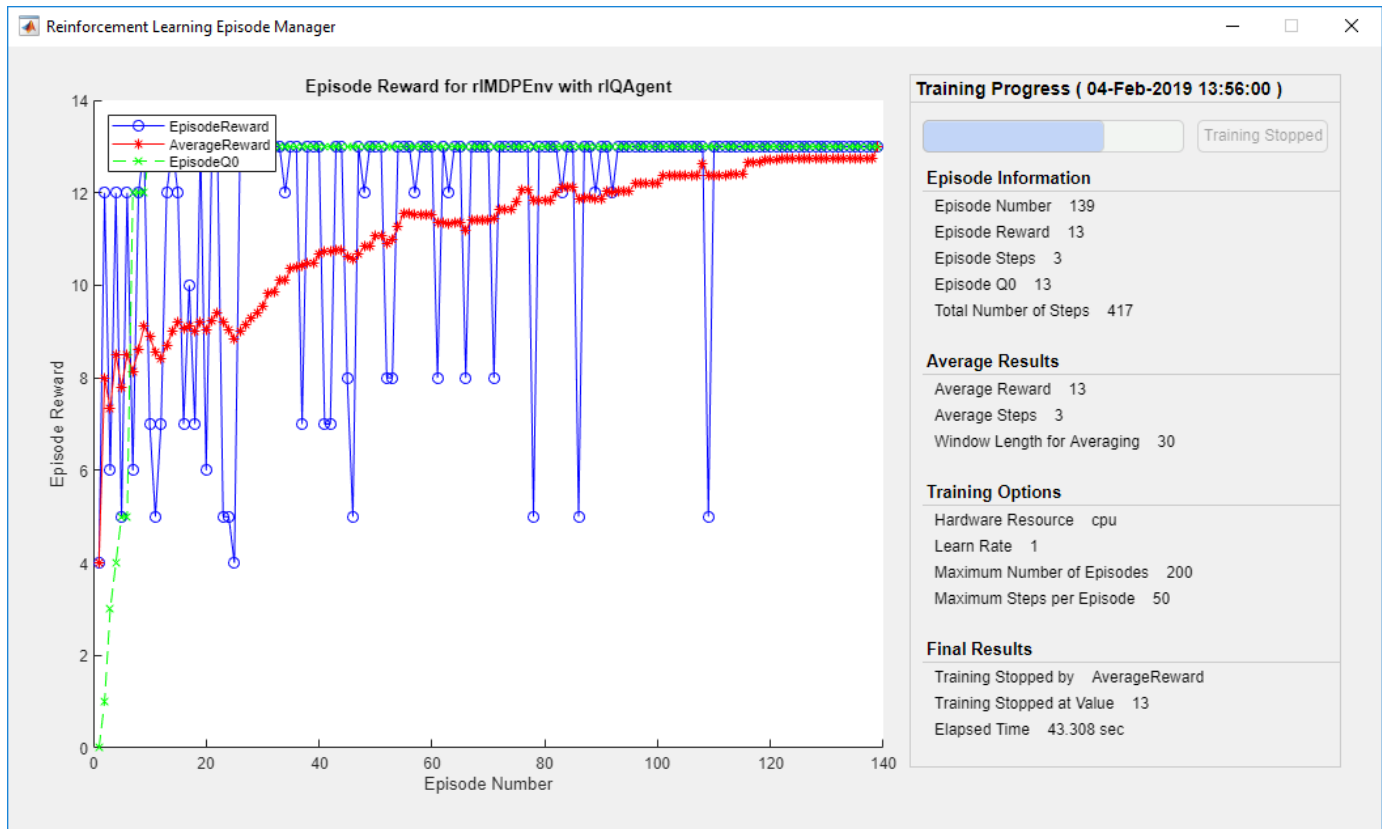
Validate Q-Learning Results

To validate the training results, simulate the agent in the training environment using the `sim` function. The agent successfully finds the optimal path which results in cumulative reward of 13.

```
Data = sim(qAgent,env);
cumulativeReward = sum(Data.Reward)
```

cumulativeReward = 13

Since the discount factor is set to 1, the values in the Q table of the trained agent match the undiscounted returns of the environment.

```
QTable = getLearnableParameters(getCritic(qAgent));
QTable{1}
```

ans = *8×2*

```
    13    12
     5    10
    11     9
     3     2
     1     9
     5     1
     0     0
     0     0
```

```
TrueTableValues = [13,12;5,10;11,9;3,2;1,9;5,1;0,0;0,0]
```

```
TrueTableValues = 8×2

    13    12
     5    10
    11     9
     3     2
     1     9
     5     1
     0     0
     0     0
```

See Also
createMDP | rlMDPEnv

More About

Create Simulink Environment and Train Agent

This example shows how to convert the PI controller in the `watertank` Simulink® model to a reinforcement learning deep deterministic policy gradient (DDPG) agent. For an example that trains a DDPG agent in MATLAB®, see "Train DDPG Agent to Control Double Integrator System" on page 5-31.

Water Tank Model

The original model for this example is the water tank model. The goal is to control the level of the water in the tank. For more information about the water tank model, see "watertank Simulink Model" (Simulink Control Design).
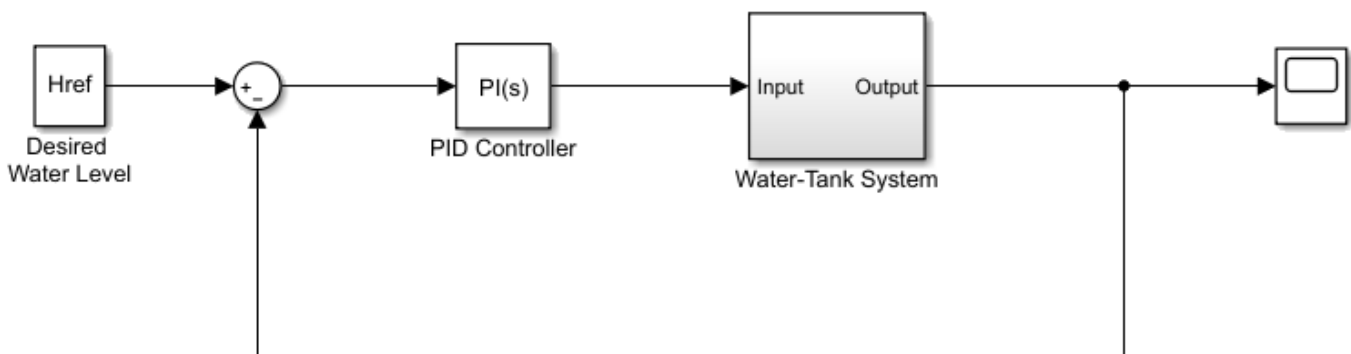
Modify the original model by making the following changes:

1 Delete the PID Controller.

2 Insert the RL Agent block.

3 Connect the observation vector $[\int e \, \mathrm{dt} \; e \; h]^T$, where h is the height of the tank, $e = r - h$, and r is the reference height.

4 Set up the reward reward $= 10(|e| < 0.1) - 1(|e| \geq 0.1) - 100(h \leq 0 \,||\, h \geq 20)$.

5 Configure the termination signal such that the simulation stops if $h \leq 0$ or $h \geq 20$.

The resulting model is `rlwatertank.slx`. For more information on this model and the changes, see "Create Simulink Environments for Reinforcement Learning" on page 2-5.

```
open_system('rlwatertank')
```
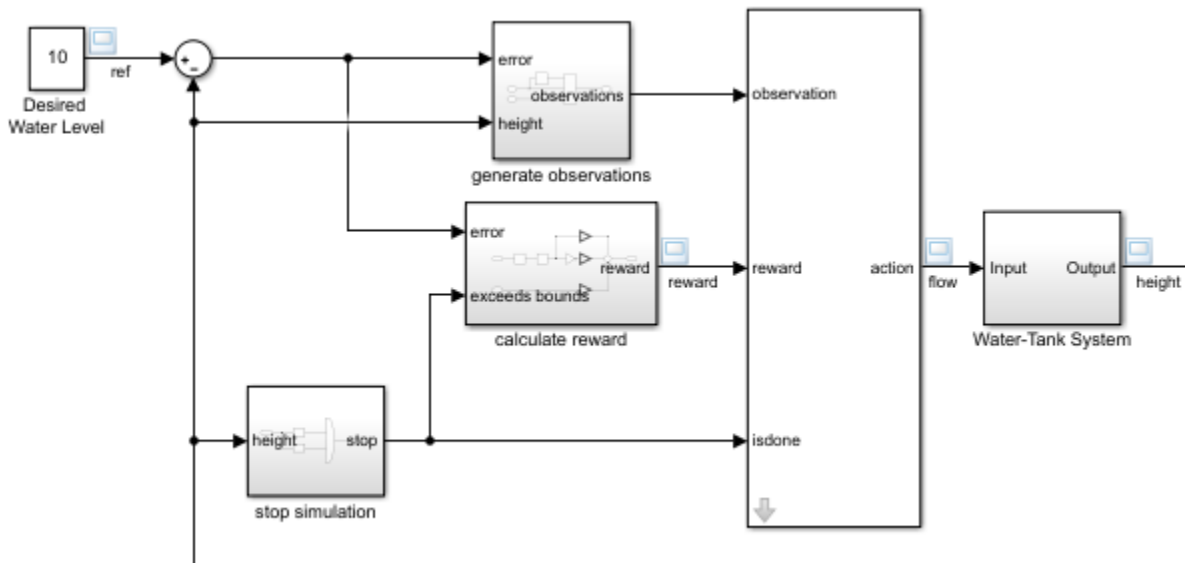
Create Environment Interface

Creating an environment model includes defining the following:

- Action and observation signals that the agent uses to interact with the environment. For more information, see `rlNumericSpec` and `rlFiniteSetSpec`.

- Reward signal that the agent uses to measure its success. For more information, see "Define Reward Signals" on page 2-7.

Define the observation specification `obsInfo` and action specification `actInfo`.

```
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 = actInfo.Dimension(1);
```

Build the environment interface object.

```
env = rlSimulinkEnv('rlwatertank','rlwatertank/RL Agent',...
    obsInfo,actInfo);
```

Set a custom reset function that randomizes the reference values for the model.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Specify the simulation time `Tf` and the agent sample time `Ts` in seconds.

```
Ts = 1.0;
Tf = 200;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG Agent

Given observations and actions, a DDPG agent approximates the long-term reward using a critic value function representation. To create the critic, first create a deep neural network with two inputs, the observation and action, and one output. For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
statePath = [
    featureInputLayer(numObservations,'Normalization','none','Name','State')
    fullyConnectedLayer(50,'Name','CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(25,'Name','CriticStateFC2')];
actionPath = [
    featureInputLayer(numActions,'Normalization','none','Name','Action')
    fullyConnectedLayer(25,'Name','CriticActionFC1')];
commonPath = [
    additionLayer(2,'Name','add')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(1,'Name','CriticOutput')];

criticNetwork = layerGraph();
criticNetwork = addLayers(criticNetwork,statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);
criticNetwork = connectLayers(criticNetwork,'CriticStateFC2','add/in1');
criticNetwork = connectLayers(criticNetwork,'CriticActionFC1','add/in2');
```

View the critic network configuration.

```
figure
plot(criticNetwork)
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOpts = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation specifications for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,'Observation',{'State'},'Action',{
```

Given observations, a DDPG agent decides which action to take using an actor representation. To create the actor, first create a deep neural network with one input, the observation, and one output, the action.

Construct the actor in a similar manner to the critic. For more information, see `rlDeterministicActorRepresentation`.

```
actorNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','State')
    fullyConnectedLayer(3, 'Name','actorFC')
    tanhLayer('Name','actorTanh')
    fullyConnectedLayer(numActions,'Name','Action')
    ];
```

```
actorOptions = rlRepresentationOptions('LearnRate',1e-04,'GradientThreshold',1);
```

```
actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,'Observation',{'State'},
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOpts = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'DiscountFactor',1.0, ...
    'MiniBatchSize',64, ...
    'ExperienceBufferLength',1e6);
agentOpts.NoiseOptions.Variance = 0.3;
agentOpts.NoiseOptions.VarianceDecayRate = 1e-5;
```

Then, create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent`.

```
agent = rlDDPGAgent(actor,critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run each training for at most `5000` episodes. Specify that each episode lasts for at most `ceil(Tf/Ts)` (that is `200`) time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than `800` over `20` consecutive episodes. At this point, the agent can control the level of water in the tank.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes, ...
    'MaxStepsPerEpisode',maxsteps, ...
    'ScoreAveragingWindowLength',20, ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',800);
```

Train the agent using the `train` function. Training is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('WaterTankDDPG.mat','agent')
end
```
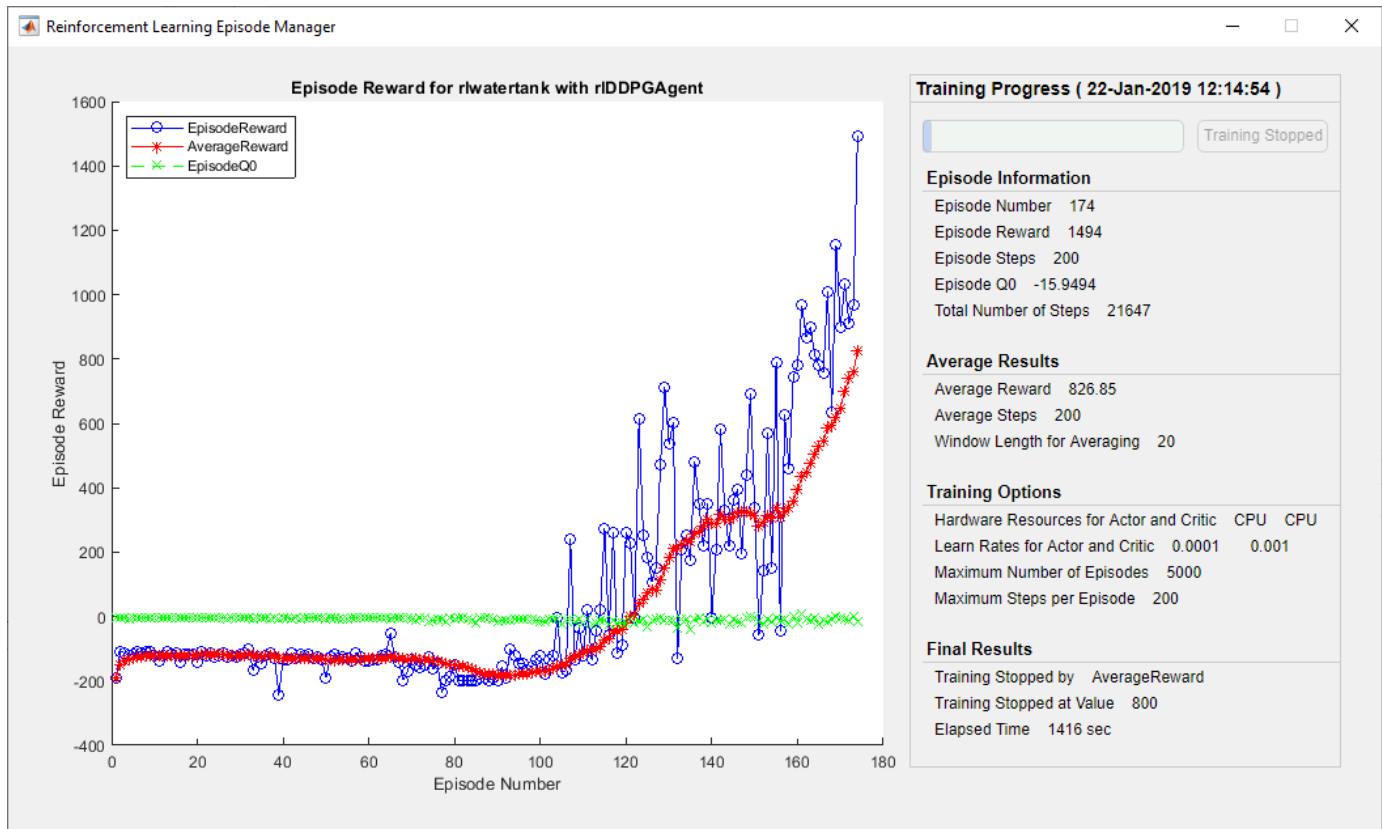
Validate Trained Agent

Validate the learned agent against the model by simulation.

```
simOpts = rlSimulationOptions('MaxSteps',maxsteps,'StopOnError','on');
experiences = sim(env,agent,simOpts);
```
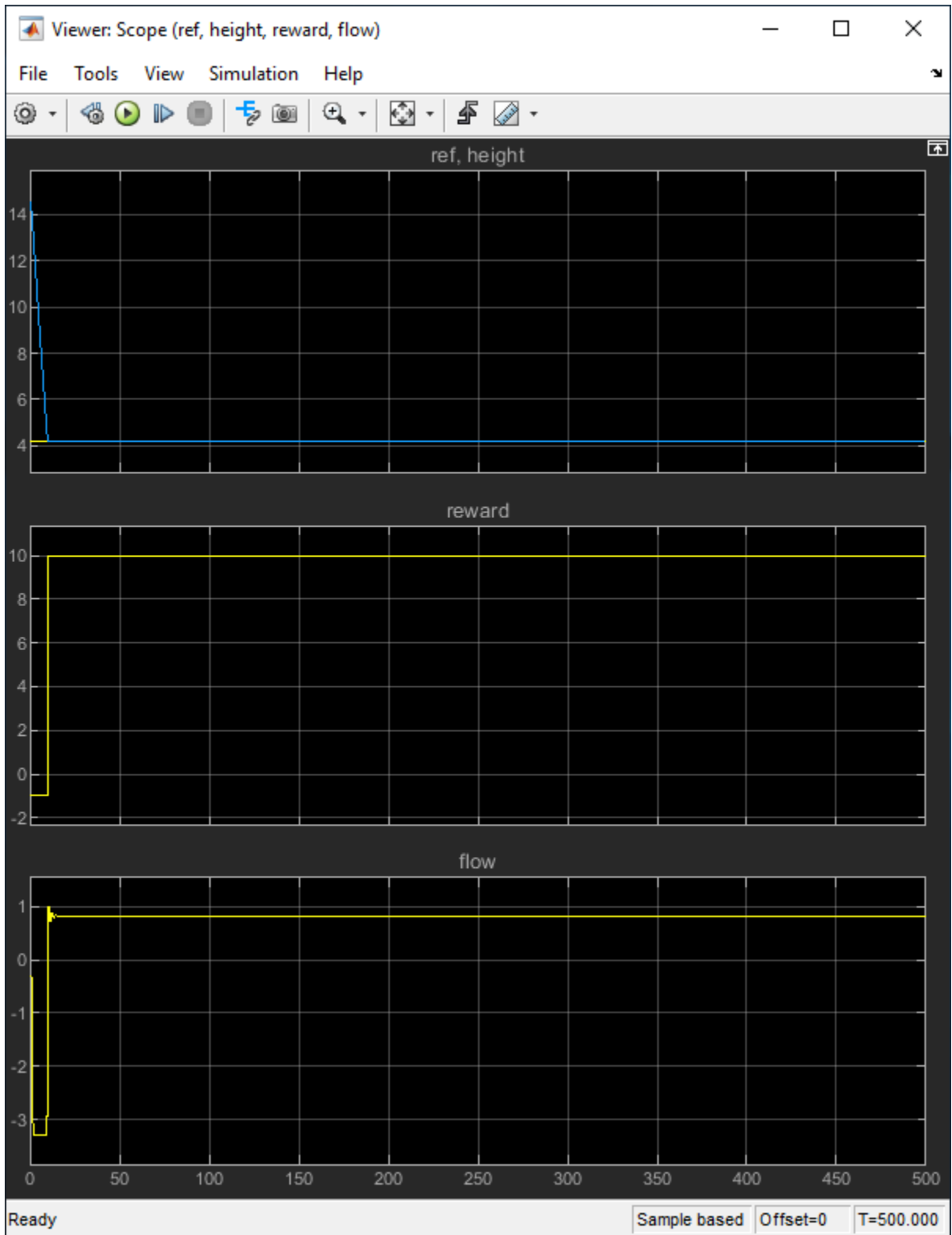
Local Function

```matlab
function in = localResetFcn(in)

% randomize reference signal
blk = sprintf('rlwatertank/Desired \nWater Level');
h = 3*randn + 10;
while h <= 0 || h >= 20
    h = 3*randn + 10;
end
in = setBlockParameter(in,blk,'Value',num2str(h));

% randomize initial height
h = 3*randn + 10;
while h <= 0 || h >= 20
    h = 3*randn + 10;
end
blk = 'rlwatertank/Water-Tank System/H';
in = setBlockParameter(in,blk,'InitialCondition',num2str(h));

end
```

See Also

`train`

More About

- "Train Reinforcement Learning Agents" on page 5-2
- "Create Simulink Environments for Reinforcement Learning" on page 2-5

2

Create Environments

Create MATLAB Environments for Reinforcement Learning

In a reinforcement learning scenario, where you are training an agent to complete a task, the environment models the external system (that is the world) with which the agent interacts. In control systems applications, this external system is often referred to as *the plant*.

As shown in the following figure, the environment:

1 Receives actions from the agent.
2 Returns observations in response to the actions.
3 Generates a reward measuring how well the action contributes to achieving the task.
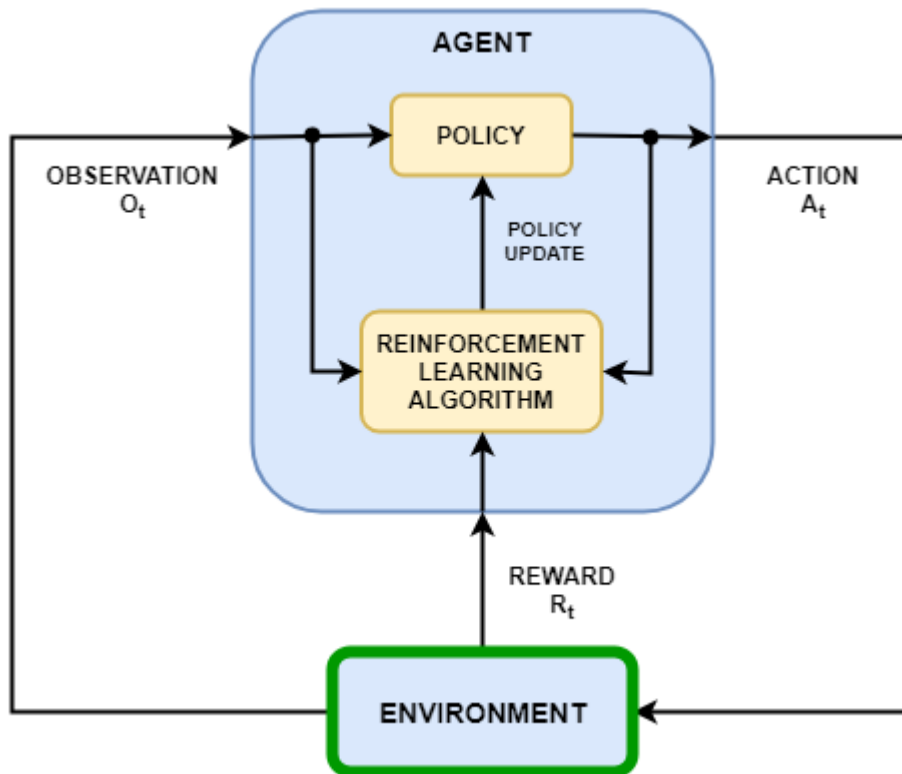
Creating an environment model involves defining:

- Action and observation signals that the agent uses to interact with the environment.
- A reward signal that the agent uses to measure its success. For more information, see "Define Reward Signals" on page 2-7.
- The environment initial condition and its dynamic behavior.

Action and Observation Signals

When you create the environment object, you must specify the action and observation signals that the agent uses to interact with the environment. You can create both discrete and continuous action and observation spaces. For more information, see `rlNumericSpec` and `rlFiniteSetSpec`, respectively.

What signals you select as actions and observations depends on your application. For example, for control system applications, the integrals (and sometimes derivatives) of error signals are often useful observations. Also, for reference-tracking applications, having a time-varying reference signal as an observation is helpful.

When you define your observation signals, ensure that all the environment states (or their estimation) are included in the observation vector. This is a good practice because the agent is often a static function which lacks internal memory or state, and so it might not be able to successfully reconstruct the environment state internally.

For example, an image observation of a swinging pendulum has position information but does not have enough information, by itself, to determine the pendulum velocity. In this case, you can measure or estimate the pendulum velocity as an additional entry in the observation vector.

Predefined MATLAB Environments

The Reinforcement Learning Toolbox software provides some predefined MATLAB environments for which the actions, observations, rewards, and dynamics are already defined. You can use these environments to:

- Learn reinforcement learning concepts.
- Gain familiarity with Reinforcement Learning Toolbox software features.
- Test your own reinforcement learning agents.

For more information, see "Load Predefined Grid World Environments" on page 2-9 and "Load Predefined Control System Environments" on page 2-15.

Custom MATLAB Environments

You can create the following types of custom MATLAB environments for your own applications:

- Grid worlds with specified size, rewards, and obstacles
- Environments with dynamics specified using custom functions
- Environments specified by creating and modifying a template environment object

Once you create a custom environment object, you can train an agent in the same manner as in a predefined environment. For more information on training agents, see "Train Reinforcement Learning Agents" on page 5-2.

Custom Grid Worlds

You can create custom grid worlds of any size with your own custom reward, state transition, and obstacle configurations. To create a custom grid world environment:

1 Create a grid world model using the `createGridWorld` function. For example, create a grid world named `gw` with ten rows and nine columns.

    ```
    gw = createGridWorld(10,9);
    ```
2 Configure the grid world by modifying the properties of the model. For example, specify the terminal state as the location `[7,9]`

    ```
    gw.TerminalStates = "[7,9]";
    ```

3 A grid world needs to be included in a Markov decision process (MDP) environment. Create an MDP environment for this grid world, which the agent uses to interact with the grid world model.

```
env = rlMDPEnv(gw);
```

For more information on Custom Grid Worlds see "Create Custom Grid World Environments" on page 2-28.

Specify Custom Functions

For simple environments, you can define a custom environment object by creating an `rlFunctionEnv` object and specifying your own custom *reset* and *step* functions.

- At the beginning of each training episode, the agent calls the reset function to set the environment initial condition. For example, you can specify known initial state values or place the environment into a random initial state.
- The step function defines the dynamics of the environment, that is, how the state changes as a function of the current state and the agent action. At each training time step, the state of the model is updated using the step function.

For more information, see "Create MATLAB Environment Using Custom Functions" on page 2-33.

Create and Modify Template Environment

For more complex environments, you can define a custom environment by creating and modifying a template environment. To create a custom environment:

1 Create an environment template class using the `rlCreateEnvTemplate` function.
2 Modify the template environment, specifying environment properties, required environment functions, and optional environment functions.
3 Validate your custom environment using `validateEnvironment`.

For more information, see "Create Custom MATLAB Environment from Template" on page 2-40.

See Also
`rlCreateEnvTemplate` | `rlFunctionEnv` | `rlPredefinedEnv`

More About
- "What Is Reinforcement Learning?" on page 1-3
- "Create Simulink Environments for Reinforcement Learning" on page 2-5

Create Simulink Environments for Reinforcement Learning

In a reinforcement learning scenario, where you are training an agent to complete task, the environment models the dynamics with which the agent interacts. As shown in the following figure, the environment:

1 Receives actions from the agent.

2 Outputs observations in response to the actions.

3 Generates a reward measuring how well the action contributes to achieving the task.
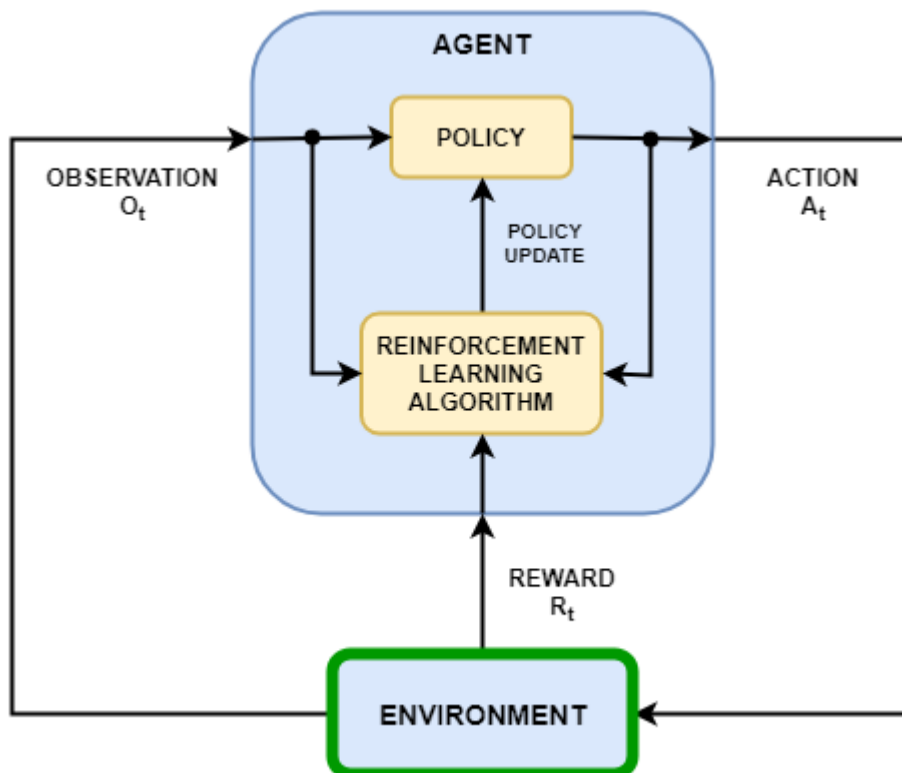
Creating an environment model includes defining the following:

- Action and observation signals that the agent uses to interact with the environment.
- Reward signal that the agent uses to measure its success. For more information, see "Define Reward Signals" on page 2-7.
- Environment dynamic behavior.

Action and Observation Signals

When you create an environment object, you must specify the action and observation signals that the agent uses to interact with the environment. You can create both discrete and continuous action spaces. For more information, see `rlNumericSpec` and `rlFiniteSetSpec`, respectively.

What signals you select as actions and observations depends on your application. For example, for control system applications, the integrals (and sometimes derivatives) of error signals are often

useful observations. Also, for reference-tracking applications, having a time-varying reference signal as an observation is helpful.

When you define your observation signals, ensure that all the system states are observable through the observations. For example, an image observation of a swinging pendulum has position information but does not have enough information to determine the pendulum velocity. In this case, you can specify the pendulum velocity as a separate observation.

Predefined Simulink Environments

Reinforcement Learning Toolbox software provides predefined Simulink environments for which the actions, observations, rewards, and dynamics are already defined. You can use these environments to:

- Learn reinforcement learning concepts.
- Gain familiarity with Reinforcement Learning Toolbox software features.
- Test your own reinforcement learning agents.

For more information, see "Load Predefined Simulink Environments" on page 2-22.

Custom Simulink Environments

To specify your own custom reinforcement learning environment, create a Simulink model with an RL Agent block. In this model, connect the action, observation, and reward signals to the RL Agent block. For an example, see "Water Tank Reinforcement Learning Environment Model" on page 2-46.

For the action and observation signals, you must create specification objects using `rlNumericSpec` for continuous signals and `rlFiniteSetSpec` for discrete signals. For bus signals, create specifications using `bus2RLSpec`.

For the reward signal, construct a scalar signal in the model and connect this signal to the RL Agent block. For more information, see "Define Reward Signals" on page 2-7.

After configuring the Simulink model, create an environment object for the model using the `rlSimulinkEnv` function.

If you have a reference model with an appropriate action input port, observation output port, and scalar reward output port, you can automatically create a Simulink model that includes this reference model and an RL Agent block. For more information, see `createIntegratedEnv`. This function returns the environment object, action specifications, and observation specifications for the model.

Your environment can include third-party functionality. For more information, see "Integrate with Existing Simulation or Environment" (Simulink).

See Also
`createIntegratedEnv` | `rlPredefinedEnv` | `rlSimulinkEnv`

More About
- "What Is Reinforcement Learning?" on page 1-3
- "Create MATLAB Environments for Reinforcement Learning" on page 2-2

Define Reward Signals

To guide the learning process, reinforcement learning uses a scalar reward signal generated from the environment. This signal measures the performance of the agent with respect to the task goals. In other words, for a given observation (state), the reward measures he effectiveness of taking a particular action. During training, an agent updates its policy based on the rewards received for different state-action combinations. For more information on the different types of agents and how they use the reward signal during training, see "Reinforcement Learning Agents" on page 3-2.

In general, you provide a positive reward to encourage certain agent actions and a negative reward (penalty) to discourage other actions. A well-designed reward signal guides the agent to maximize the expectation of the long-term reward. What constitutes a well-designed reward depends on your application and the agent goals.

For example, when an agent must perform a task for as long as possible, a common strategy is to provide a small positive reward for each time step that the agent successfully performs the task and a large penalty when the agent fails. This approach encourages longer training episodes while heavily discouraging episodes that fail. For an example that uses this approach, see "Train DQN Agent to Balance Cart-Pole System" on page 5-8.

If your reward function incorporates multiple signals, such as position, velocity, and control effort, you must consider the relative sizes of the signals and scale their contributions to the reward signal accordingly.

You can specify either continuous or discrete reward signals. In either case, you must provide a reward signal that provides rich information when the action and observation signals change.

Continuous Rewards

A continuous reward function varies continuously with changes in the environment observations and actions. In general, continuous reward signals improve convergence during training and can lead to simpler network structures.

An example of a continuous reward is the quadratic regulator (QR) cost function, where the long-term reward can be expressed as

$$J_i = -\left(s_\tau^T Q_\tau s_\tau + \sum_{j=i}^{\tau} s_j^T Q_j s_j + a_j^T R_j a_j + 2 s_j^T N_j a_j \right)$$

Here, Q_τ, Q, R, and N are the weight matrices. Q_τ is the terminal weight matrix, applied only at the end of the episode. Also, s is the observation vector, a is the action vector, and τ is the terminal iteration of the episode. The instantaneous reward for this cost function is

$$r_i = s_i^T Q_i s_i + a_i^T R_i a_i + 2 s_i^T N_i a_i$$

This QR reward structure encourages driving s to zero with minimal action effort. A QR-based reward structure is a good reward to choose for regulation or stationary point problems, such as pendulum swing-up or regulating the position of the double integrator. For training examples that use a QR reward, see "Train DQN Agent to Swing Up and Balance Pendulum" on page 5-37 and "Train DDPG Agent to Control Double Integrator System" on page 5-31.

Smooth continuous rewards, such as the QR regulator, are good for fine-tuning parameters and can provide policies similar to optimal controllers (LQR/MPC).

Discrete Rewards

A discrete reward function varies discontinuously with changes in the environment observations or actions. These types of reward signals can make convergence slower and can require more complex network structures. Discrete rewards are usually implemented as *events* that occur in the environment—for example, when an agent receives a positive reward if it exceeds some target value or a penalty when it violates some performance constraint.

While discrete rewards can slow down convergence, they can also guide the agent toward better reward regions in the state space of the environment. For example, a region-based reward, such as a fixed reward when the agent is near a target location, can emulate final-state constraints. Also, a region-based penalty can encourage an agent to avoid certain areas of the state space.

Mixed Rewards

In many cases, providing a mixed reward signal that has a combination of continuous and discrete reward components is beneficial. The discrete reward signal can be used to drive the system away from bad states, and the continuous reward signal can improve convergence by providing a smooth reward near target states. For example, in "Train DDPG Agent to Control Flying Robot" on page 5-90, the reward function has three components: r_1, r_2, and r_3.

$$r_1 = 10\left(\left(x_t^2 + y_t^2 + \theta_t^2\right) < 0.5\right)$$

$$r_2 = -100(|x_t| \geq 20 \,||\, |y_t| \geq 20)$$

$$r_3 = -\left(0.2(R_{t-1} + L_{t-1})^2 + 0.3(R_{t-1} - L_{t-1})^2 + 0.03x_t^2 + 0.03y_t^2 + 0.02\theta_t^2\right)$$

$$r = r_1 + r_2 + r_3$$

Here:

- r_1 is a region-based continuous reward that applies only near the target location of the robot.
- r_2 is a discrete signal that provides a large penalty when the robot moves far from the target location.
- r_3 is a continuous QR penalty that applies for all robot states.

See Also

More About

Load Predefined Grid World Environments

Reinforcement Learning Toolbox software provides several predefined grid world environments for which the actions, observations, rewards, and dynamics are already defined. You can use these environments to:

- Learn reinforcement learning concepts.
- Gain familiarity with Reinforcement Learning Toolbox software features.
- Test your own reinforcement learning agents.

You can load the following predefined MATLAB grid world environments using the `rlPredefinedEnv` function.

Environment	Agent Task
Basic grid world	Move from a starting location to a target location on a two-dimensional grid by selecting moves from the discrete action space {N,S,E,W}.
Waterfall grid world	Move from a starting location to a target location on a larger two-dimensional grid with unknown deterministic or stochastic dynamics.

For more information on the properties of grid world environments, see "Create Custom Grid World Environments" on page 2-28.

You can also load predefined MATLAB control system environments. For more information, see "Load Predefined Control System Environments" on page 2-15.

Basic Grid World

The basic grid world environment is a two-dimensional 5-by-5 grid with a starting location, terminal location, and obstacles. The environment also contains a special jump from state [2,4] to state [4,4]. The goal of the agent is to move from the starting location to the terminal location while avoiding obstacles and maximizing the total reward.

To create a basic grid world environment, use the `rlPredefinedEnv` function. This function creates an `rlMDPEnv` object representing the grid world.

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

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

- Agent location is a red circle. By default, the agent starts in state [1,1].
- Terminal location is a blue square.
- Obstacles are black squares.

```
plot(env)
```
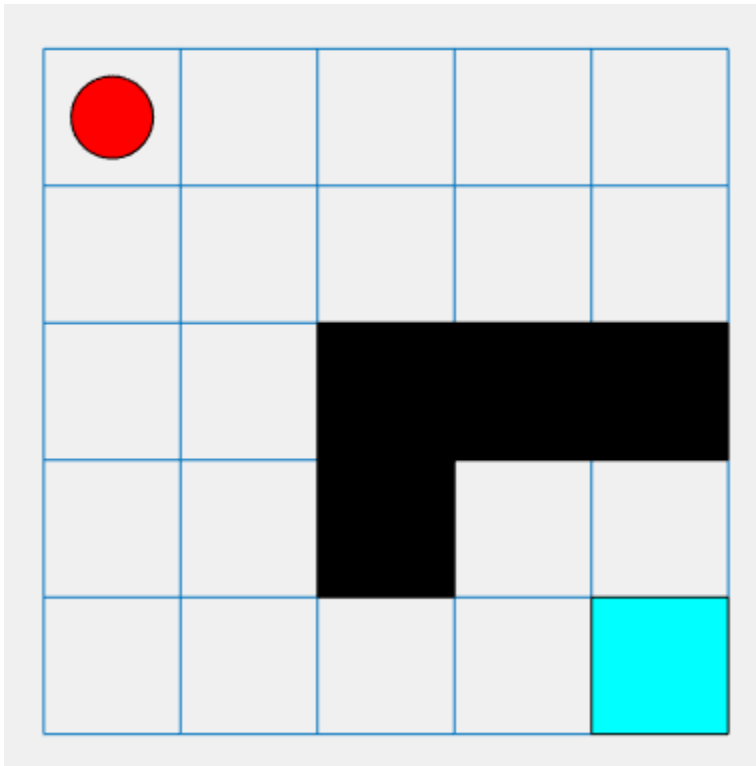
Actions

The agent can move in one of four possible directions (north, south, east, or west).

Rewards

The agent receives the following rewards or penalties:

- `+10` reward for reaching the terminal state at [5,5]
- `+5` reward for jumping from state [2,4] to state [4,4]
- `-1` penalty for every other action

Deterministic Waterfall Grid Worlds

The deterministic waterfall grid world environment is a two-dimensional 8-by-7 grid with a starting location and terminal location. The environment includes a waterfall that pushes the agent toward the bottom of the grid. The goal of the agent is to move from the starting location to the terminal location while maximizing the total reward.

To create a deterministic waterfall grid world, use the `rlPredefinedEnv` function. This function creates an `rlMDPEnv` object representing the grid world.

```
env = rlPredefinedEnv('WaterFallGridWorld-Deterministic');
```

As with the basic grid world, you can visualize the environment, where the agent is a red circle and the terminal location is a blue square.

```
plot(env)
```
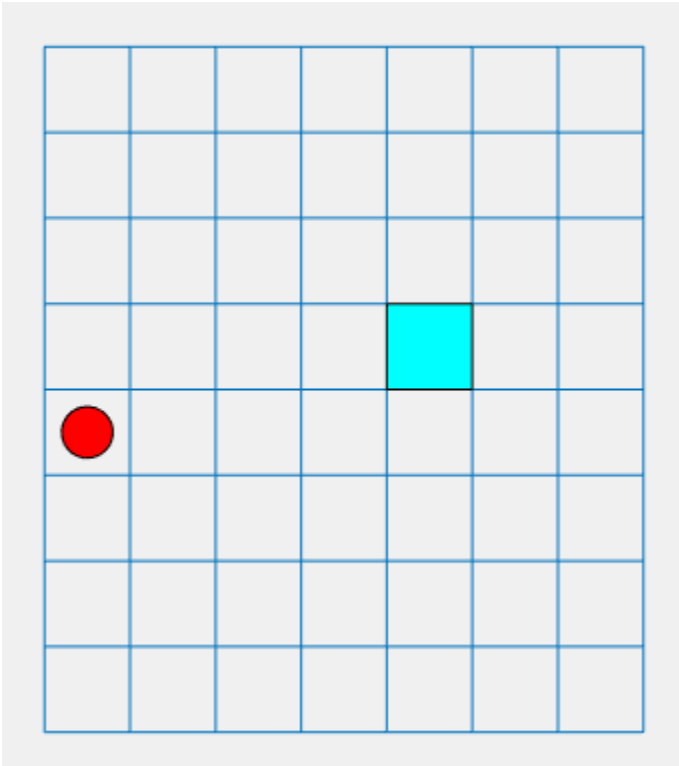
Actions

The agent can move in one of four possible directions (north, south, east, or west).

Rewards

The agent receives the following rewards or penalties:

- `+10` reward for reaching the terminal state at [4,5]
- `-1` penalty for every other action

Waterfall Dynamics

In this environment, a waterfall pushes the agent toward the bottom of the grid.
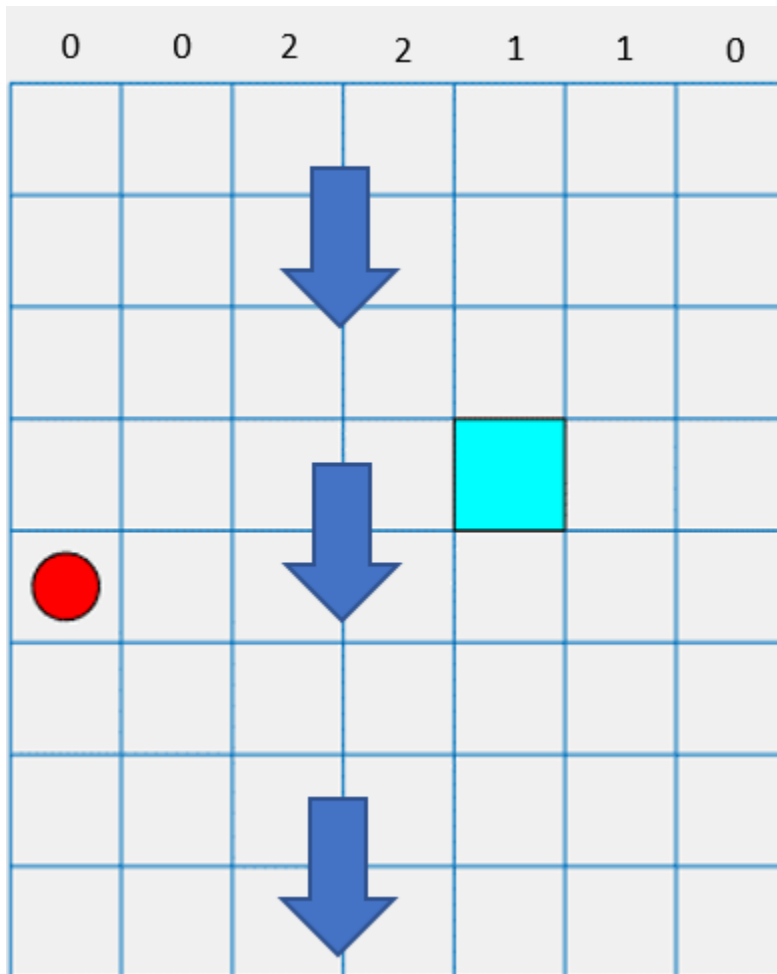
The intensity of the waterfall varies between the columns, as shown at the top of the preceding figure. When the agent moves into a column with a nonzero intensity, the waterfall pushes it downward by the indicated number of squares. For example, if the agent goes east from state [5,2], it reaches state [7,3].

Stochastic Waterfall Grid Worlds

The stochastic waterfall grid world environment is a two-dimensional 8-by-7 grid with a starting location and terminal locations. The environment includes a waterfall that pushes the agent towards the bottom of the grid with a stochastic intensity. The goal of the agent is to move from the starting location to the target terminal location while avoiding the penalty terminal states along the bottom of the grid and maximizing the total reward.

To create a stochastic waterfall grid world, use the `rlPredefinedEnv` function. This function creates an `rlMDPEnv` object representing the grid world.

```
env = rlPredefinedEnv('WaterFallGridWorld-Stochastic');
```

As with the basic grid world, you can visualize the environment, where the agent is a red circle and the terminal location is a blue square.

```
plot(env)
```
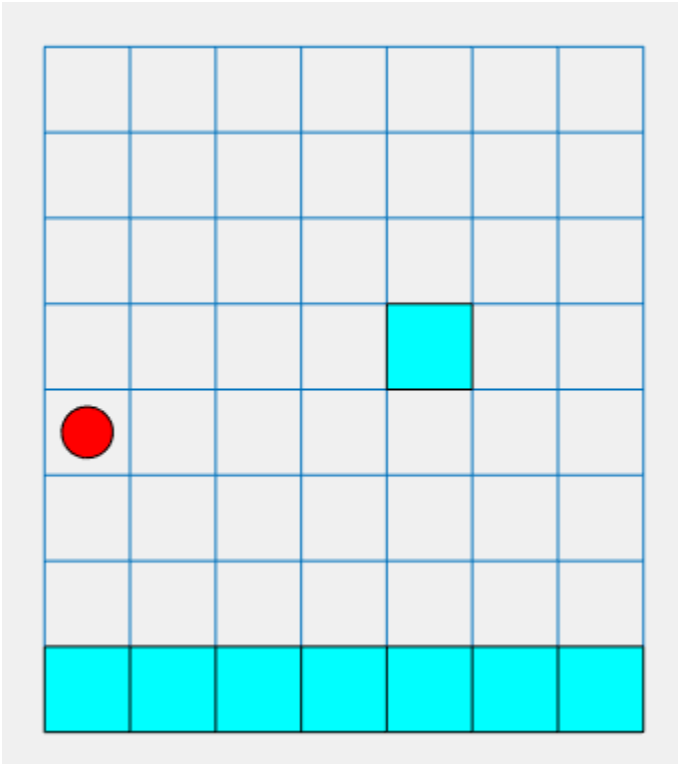
Actions

The agent can move in one of four possible directions (north, south, east, or west).

Rewards

The agent receives the following rewards or penalties:

- +10 reward for reaching the terminal state at [4,5]
- -10 penalty for reaching any terminal state in the bottom row of the grid
- -1 penalty for every other action

Waterfall Dynamics

In this environment, a waterfall pushes the agent towards the bottom of the grid with a stochastic intensity. The baseline intensity matches the intensity of the deterministic waterfall environment. However, in the stochastic waterfall case, the agent has an equal chance of experiencing the indicated intensity, one level above that intensity, or one level below that intensity. For example, if the agent goes east from state [5,2], it has an equal chance of reaching state [6,3], [7,3], or [8,3].

See Also
rlMDPEnv | rlPredefinedEnv | train

More About
- "Create MATLAB Environments for Reinforcement Learning" on page 2-2

- "Load Predefined Control System Environments" on page 2-15
- "Train Reinforcement Learning Agents" on page 5-2

Load Predefined Control System Environments

Reinforcement Learning Toolbox software provides several predefined control system environments for which the actions, observations, rewards, and dynamics are already defined. You can use these environments to:

- Learn reinforcement learning concepts.
- Gain familiarity with Reinforcement Learning Toolbox software features.
- Test your own reinforcement learning agents.

You can load the following predefined MATLAB control system environments using the `rlPredefinedEnv` function.

Environment	Agent Task
Cart-pole	Balance a pole on a moving cart by applying forces to the cart using either a discrete or continuous action space.
Double integrator	Control a second-order dynamic system using either a discrete or continuous action space.
Simple pendulum with image observation	Swing up and balance a simple pendulum using either a discrete or continuous action space.

You can also load predefined MATLAB grid world environments. For more information, see "Load Predefined Grid World Environments" on page 2-9.

Cart-Pole Environments

The goal of the agent in the predefined cart-pole environments is to balance a pole on a moving cart by applying horizontal forces to the cart. The pole is considered successfully balanced if both of the following conditions are satisfied:

- The pole angle remains within a given threshold of the vertical position, where the vertical position is zero radians.
- The magnitude of the cart position remains below a given threshold.

There are two cart-pole environment variants, which differ by the agent action space.

- Discrete — Agent can apply a force of either F_{max} or $-F_{max}$ to the cart, where F_{max} is the `MaxForce` property of the environment.
- Continuous — Agent can apply any force within the range $[-F_{max}, F_{max}]$.

To create a cart-pole environment, use the `rlPredefinedEnv` function.

- Discrete action space

  ```
  env = rlPredefinedEnv('CartPole-Discrete');
  ```
- Continuous action space

  ```
  env = rlPredefinedEnv('CartPole-Continuous');
  ```

You can visualize the cart-pole environment using the `plot` function. The plot displays the cart as a blue square and the pole as a red rectangle.
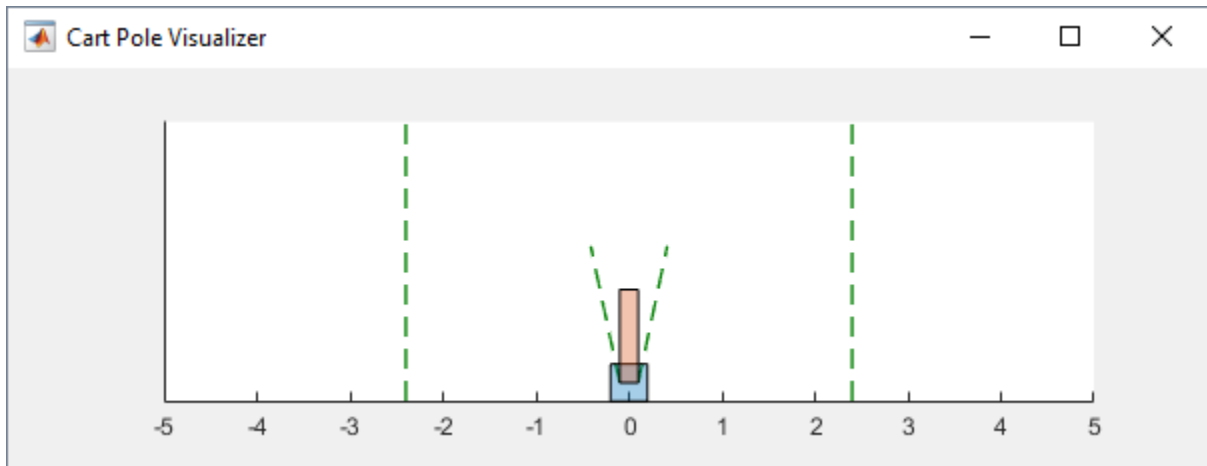
```
plot(env)
```


To visualize the environment during training, call `plot` before training and keep the visualization figure open.

For examples showing how to train agents in cart-pole environments, see the following:

- "Train DQN Agent to Balance Cart-Pole System" on page 5-8
- "Train PG Agent to Balance Cart-Pole System" on page 5-14
- "Train AC Agent to Balance Cart-Pole System" on page 5-19

Environment Properties

Property	Description	Default
`Gravity`	Acceleration due to gravity in meters per second	`9.8`
`MassCart`	Mass of the cart in kilograms	`1`
`MassPole`	Mass of the pole in kilograms	`0.1`
`Length`	Half the length of the pole in meters	`0.5`
`MaxForce`	Maximum horizontal force magnitude in newtons	`10`
`Ts`	Sample time in seconds	`0.02`
`ThetaThresholdRadians`	Pole angle threshold in radians	`0.2094`
`XThreshold`	Cart position threshold in meters	`2.4`
`RewardForNotFalling`	Reward for each time step the pole is balanced	`1`
`PenaltyForFalling`	Reward penalty for failing to balance the pole	Discrete — `-5` Continuous — `-50`

Property	Description	Default
State	Environment state, specified as a column vector with the following state variables: • Cart position • Derivative of cart position • Pole angle • Derivative of pole angle	[0 0 0 0]'

Actions

In the cart-pole environments, the agent interacts with the environment using a single action signal, the horizontal force applied to the cart. The environment contains a specification object for this action signal. For the environment with a:

• Discrete action space, the specification is an `rlFiniteSetSpec` object.

• Continuous action space, the specification is an `rlNumericSpec` object.

For more information on obtaining action specifications from an environment, see `getActionInfo`.

Observations

In the cart-pole system, the agent can observe all the environment state variables in `env.State`. For each state variable, the environment contains an `rlNumericSpec` observation specification. All the states are continuous and unbounded.

For more information on obtaining observation specifications from an environment, see `getObservationInfo`.

Reward

The reward signal for this environment consists of two components.

• A positive reward for each time step that the pole is balanced, that is, the cart and pole both remain within their specified threshold ranges. This reward accumulates over the entire training episode. To control the size of this reward, use the `RewardForNotFalling` property of the environment.

• A one-time negative penalty if either the pole or cart moves outside of their threshold range. At this point, the training episode stops. To control the size of this penalty, use the `PenaltyForFalling` property of the environment.

Double Integrator Environments

The goal of the agent in the predefined double integrator environments is to control the position of a mass in a second-order system by applying a force input. Specifically, the second-order system is a double integrator with a gain.

Training episodes for these environments end when either of the following events occurs:

• The mass moves beyond a given threshold from the origin.

• The norm of the state vector is less than a given threshold.

There are two double integrator environment variants, which differ by the agent action space.

- Discrete — Agent can apply a force of either F_{max} or $-F_{max}$ to the cart, where F_{max} is the MaxForce property of the environment.
- Continuous — Agent can apply any force within the range $[-F_{max}, F_{max}]$.

To create a double integrator environment, use the rlPredefinedEnv function.

- Discrete action space

  ```
  env = rlPredefinedEnv('DoubleIntegrator-Discrete');
  ```
- Continuous action space

  ```
  env = rlPredefinedEnv('DoubleIntegrator-Continuous');
  ```

You can visualize the double integrator environment using the plot function. The plot displays the mass as a red rectangle.
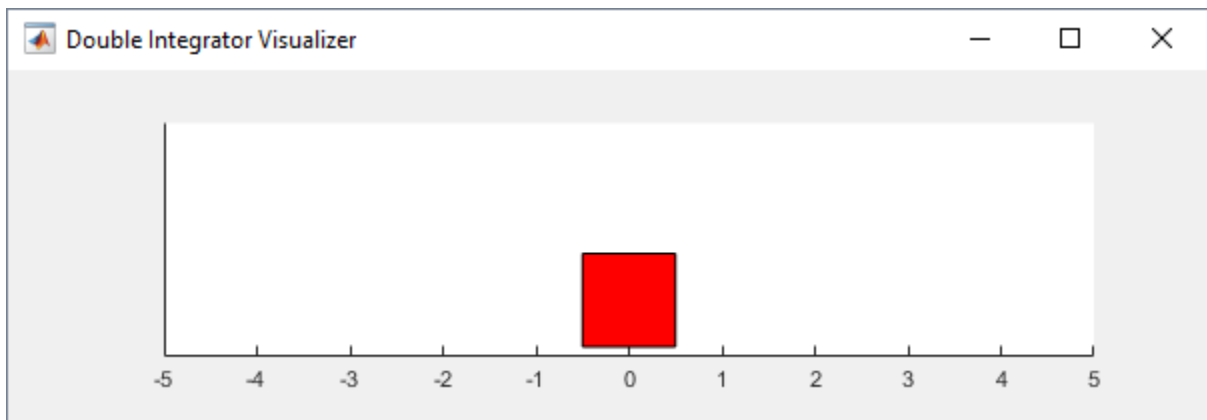
```
plot(env)
```


To visualize the environment during training, call plot before training and keep the visualization figure open.

For examples showing how to train agents in double integrator environments, see the following:

- "Train DDPG Agent to Control Double Integrator System" on page 5-31
- "Train PG Agent with Baseline to Control Double Integrator System" on page 5-25

Environment Properties

Property	Description	Default
Gain	Gain for the double integrator	1
Ts	Sample time in seconds	0.1
MaxDistance	Distance magnitude threshold in meters	5
GoalThreshold	State norm threshold	0.01
Q	Weight matrix for observation component of reward signal	[10 0; 0 1]

Property	Description	Default
R	Weight matrix for action component of reward signal	0.01
MaxForce	Maximum input force in newtons	Discrete: 2 Continuous: Inf
State	Environment state, specified as a column vector with the following state variables: • Mass position • Derivative of mass position	[0 0]'

Actions

In the double integrator environments, the agent interacts with the environment using a single action signal, the force applied to the mass. The environment contains a specification object for this action signal. For the environment with a:

• Discrete action space, the specification is an `rlFiniteSetSpec` object.
• Continuous action space, the specification is an `rlNumericSpec` object.

For more information on obtaining action specifications from an environment, see `getActionInfo`.

Observations

In the double integrator system, the agent can observe both of the environment state variables in `env.State`. For each state variable, the environment contains an `rlNumericSpec` observation specification. Both states are continuous and unbounded.

For more information on obtaining observation specifications from an environment, see `getObservationInfo`.

Reward

The reward signal for this environment is the discrete-time equivalent of the following continuous-time reward, which is analogous to the cost function of an LQR controller.

$$reward = -\int (x'Qx + u'Ru)\, dt$$

Here:

• `Q` and `R` are environment properties.
• x is the environment state vector.
• u is the input force.

This reward is the episodic reward, that is, the cumulative reward across the entire training episode.

Simple Pendulum Environments with Image Observation

This environment is a simple frictionless pendulum that is initially hangs in a downward position. The training goal is to make the pendulum stand upright without falling over using minimal control effort.

There are two simple pendulum environment variants, which differ by the agent action space.

- Discrete — Agent can apply a torque of `-2`, `-1`, `0`, `1`, or `2` to the pendulum.
- Continuous — Agent can apply any torque within the range `[-2,2]`.

To create a simple pendulum environment, use the `rlPredefinedEnv` function.

- Discrete action space

  `env = rlPredefinedEnv('SimplePendulumWithImage-Discrete');`

- Continuous action space

  `env = rlPredefinedEnv('SimplePendulumWithImage-Continuous');`

For examples showing how to train an agent in this environment, see the following:

- "Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation" on page 5-65
- "Create Agent Using Deep Network Designer and Train Using Image Observations" on page 5-73

Environment Properties

Property	Description	Default
`Mass`	Pendulum mass	1
`RodLength`	Pendulum length	1
`RodInertia`	Pendulum moment of inertia	0
`Gravity`	Acceleration due to gravity in meters per second	9.81
`DampingRatio`	Damping on pendulum motion	0
`MaximumTorque`	Maximum input torque in newtons	2
`Ts`	Sample time in seconds	0.05
`State`	Environment state, specified as a column vector with the following state variables: • Pendulum angle • Pendulum angular velocity	`[0 0 ]'`
`Q`	Weight matrix for observation component of reward signal	`[1 0;0 0.1]`
`R`	Weight matrix for action component of reward signal	`1e-3`

Actions

In the simple pendulum environments, the agent interacts with the environment using a single action signal, the torque applied at the base of the pendulum. The environment contains a specification object for this action signal. For the environment with a:

- Discrete action space, the specification is an `rlFiniteSetSpec` object.
- Continuous action space, the specification is an `rlNumericSpec` object.

For more information on obtaining action specifications from an environment, see `getActionInfo`.

Observations

In the simple pendulum environment, the agent receives the following observation signals:

- 50-by-50 grayscale image of the pendulum position
- Derivative of the pendulum angle

For each observation signal, the environment contains an `rlNumericSpec` observation specification. All the observations are continuous and unbounded.

For more information on obtaining observation specifications from an environment, see `getObservationInfo`.

Reward

The reward signal for this environment is

$$r_t = -\left(\theta_t^2 + 0.1 * \dot{\theta}_t^2 + 0.001 * u_{t-1}^2\right)$$

Here:

- θ_t is the pendulum angle of displacement from the upright position.
- $\dot{\theta}_t$ is the derivative of the pendulum angle.
- u_{t-1} is the control effort from the previous time step.

See Also
`rlPredefinedEnv` | `train`

More About
- "Create MATLAB Environments for Reinforcement Learning" on page 2-2
- "Load Predefined Grid World Environments" on page 2-9
- "Train Reinforcement Learning Agents" on page 5-2

Load Predefined Simulink Environments

Reinforcement Learning Toolbox software provides predefined Simulink environments for which the actions, observations, rewards, and dynamics are already defined. You can use these environments to:

- Learn reinforcement learning concepts.
- Gain familiarity with Reinforcement Learning Toolbox software features.
- Test your own reinforcement learning agents.

You can load the following predefined Simulink environments using the `rlPredefinedEnv` function.

Environment	Agent Task
Simple pendulum Simulink model	Swing up and balance a simple pendulum using either a discrete or continuous action space.
Cart-pole Simscape™ model	Balance a pole on a moving cart by applying forces to the cart using either a discrete or continuous action space.

For predefined Simulink environments, the environment dynamics, observations, and reward signal are defined in a corresponding Simulink model. The `rlPredefinedEnv` function creates a `SimulinkEnvWithAgent` object that the `train` function uses to interact with the Simulink model.

Simple Pendulum Simulink Model

This environment is a simple frictionless pendulum that initially hangs in a downward position. The training goal is to make the pendulum stand upright without falling over using minimal control effort. The model for this environment is defined in the `rlSimplePendulumModel` Simulink model.

```
open_system('rlSimplePendulumModel')
```
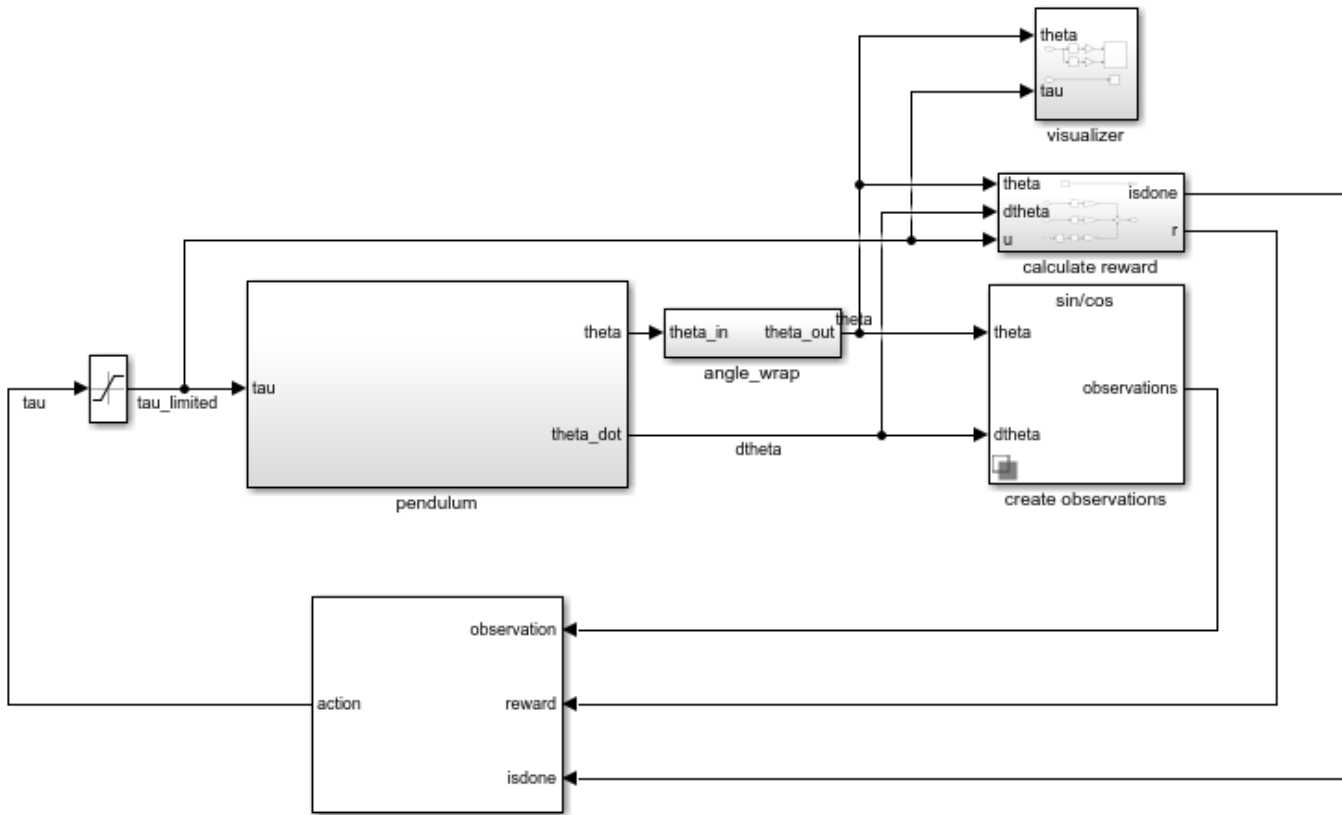
There are two simple pendulum environment variants, which differ by the agent action space.

- Discrete — Agent can apply a torque of either T_{max}, 0, or $-T_{max}$ to the pendulum, where T_{max} is the max_tau variable in the model workspace.
- Continuous — Agent can apply any torque within the range $[-T_{max}, T_{max}]$.

To create a simple pendulum environment, use the rlPredefinedEnv function.

- Discrete action space

  env = rlPredefinedEnv('SimplePendulumModel-Discrete');

- Continuous action space

  env = rlPredefinedEnv('SimplePendulumModel-Continuous');

For examples that train agents in the simple pendulum environment, see:

- "Train DQN Agent to Swing Up and Balance Pendulum" on page 5-37
- "Train DDPG Agent to Swing Up and Balance Pendulum" on page 5-44

Actions

In the simple pendulum environments, the agent interacts with the environment using a single action signal, the torque applied at the base of the pendulum. The environment contains a specification object for this action signal. For the environment with a:

- Discrete action space, the specification is an `rlFiniteSetSpec` object.
- Continuous action space, the specification is an `rlNumericSpec` object.

For more information on obtaining action specifications from an environment, see `getActionInfo`.

Observations

In the simple pendulum environment, the agent receives the following three observation signals, which are constructed within the create observations subsystem.

- Sine of the pendulum angle
- Cosine of the pendulum angle
- Derivative of the pendulum angle

For each observation signal, the environment contains an `rlNumericSpec` observation specification. All the observations are continuous and unbounded.

For more information on obtaining observation specifications from an environment, see `getObservationInfo`.

Reward

The reward signal for this environment, which is constructed in the calculate reward subsystem, is

$$r_t = -\left(\theta_t^2 + 0.1 * \dot{\theta}_t^2 + 0.001 * u_{t-1}^2\right)$$

Here:

- θ_t is the pendulum angle of displacement from the upright position.
- $\dot{\theta}_t$ is the derivative of the pendulum angle.
- u_{t-1} is the control effort from the previous time step.

Cart-Pole Simscape Model

The goal of the agent in the predefined cart-pole environments is to balance a pole on a moving cart by applying horizontal forces to the cart. The pole is considered successfully balanced if both of the following conditions are satisfied:

- The pole angle remains within a given threshold of the vertical position, where the vertical position is zero radians.
- The magnitude of the cart position remains below a given threshold.

The model for this environment is defined in the `rlCartPoleSimscapeModel` Simulink model. The dynamics of this model are defined using Simscape Multibody™.

```
open_system('rlCartPoleSimscapeModel')
```
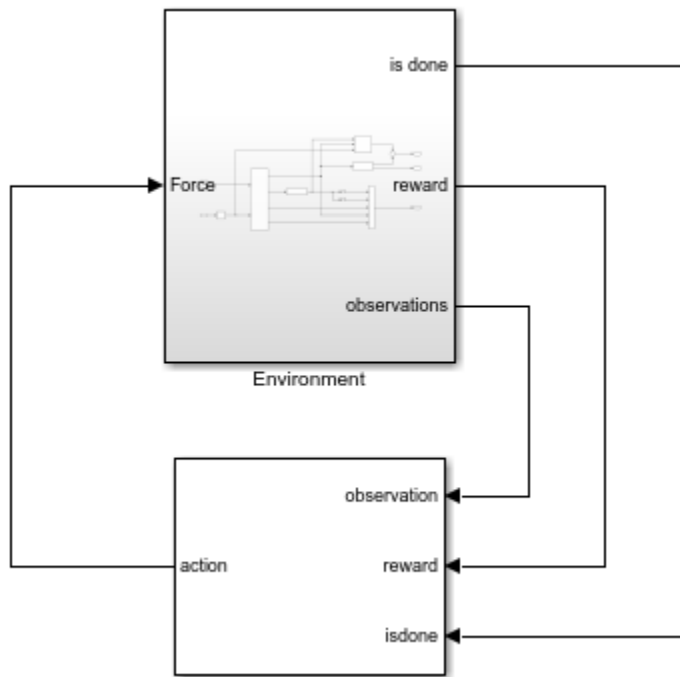
In the Environment subsystem, the model dynamics are defined using Simscape components and the reward and observation are constructed using Simulink blocks.

```
open_system('rlCartPoleSimscapeModel/Environment')
```
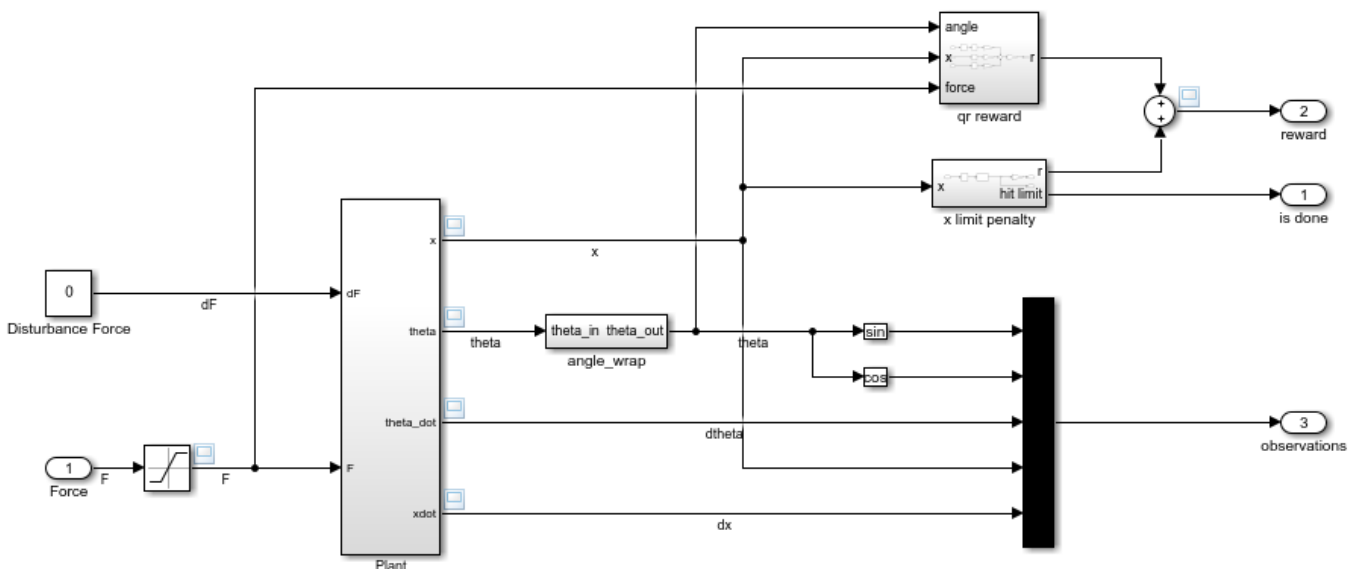

There are two cart-pole environment variants, which differ by the agent action space.

* Discrete — Agent can apply a force of 15, 0, or -15 to the cart.
* Continuous — Agent can apply any force within the range [-15,15].

To create a cart-pole environment, use the `rlPredefinedEnv` function.

- Discrete action space

  ```
  env = rlPredefinedEnv('CartPoleSimscapeModel-Discrete');
  ```
- Continuous action space

  ```
  env = rlPredefinedEnv('CartPoleSimscapeModel-Continuous');
  ```

For an example that trains an agent in this cart-pole environment, see "Train DDPG Agent to Swing Up and Balance Cart-Pole System" on page 5-51.

Actions

In the cart-pole environments, the agent interacts with the environment using a single action signal, the force applied to the cart. The environment contains a specification object for this action signal. For the environment with a:

- Discrete action space, the specification is an `rlFiniteSetSpec` object.
- Continuous action space, the specification is an `rlNumericSpec` object.

For more information on obtaining action specifications from an environment, see `getActionInfo`.

Observations

In the cart-pole environment, the agent receives the following five observation signals.

- Sine of the pole angle
- Cosine of the pole angle
- Derivative of the pendulum angle
- Cart position
- Derivative of cart position

For each observation signal, the environment contains an `rlNumericSpec` observation specification. All the observations are continuous and unbounded.

For more information on obtaining observation specifications from an environment, see `getObservationInfo`.

Reward

The reward signal for this environment is the sum of three components ($r = r_{qr} + r_n + r_p$):

- A quadratic regulator control reward, constructed in the `Environment/qr reward` subsystem.

  $$r_{qr} = -\left(0.1 * x^2 + 0.5 * \theta^2 + 0.005 * u_{t-1}^2\right)$$
- An additional reward for when the pole is near the upright position, constructed in the `Environment/near upright reward` subsystem.

  $$r_n = 10 * (|\theta| < 0.175)$$
- A cart limit penalty, constructed in the `Environment/x limit penalty` subsystem. This subsystem generates a negative reward when the magnitude of the cart position exceeds a given threshold.

$$r_p = -100 * (|x| \geq 3.5)$$

Here:

- x is the cart position.
- θ is the pole angle of displacement from the upright position.
- u_{t-1} is the control effort from the previous time step.

See Also

Blocks
RL Agent

Functions
`rlPredefinedEnv` | `train`

More About

- "Create Simulink Environments for Reinforcement Learning" on page 2-5
- "Train Reinforcement Learning Agents" on page 5-2

Create Custom Grid World Environments

A grid world is a two-dimensional, cell-based environment where the agent starts from one cell and moves toward the terminal cell while collecting as much reward as possible. Grid world environments are useful for applying reinforcement learning algorithms to discover optimal paths and policies for agents on the grid to arrive at the terminal goal in the fewest moves.
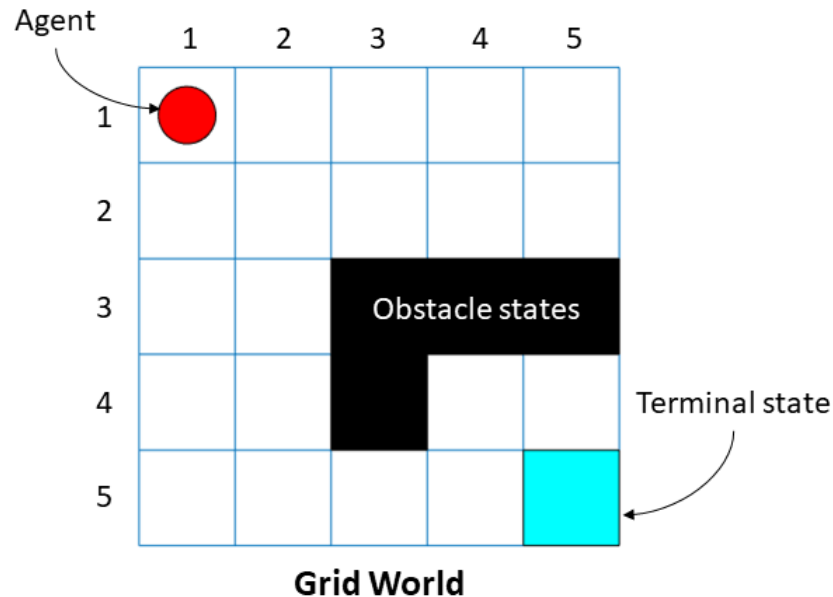
Grid World

Reinforcement Learning Toolbox lets you create custom MATLAB grid world environments for your own applications. To create a custom grid world environment:

1 Create the grid world model.
2 Configure the grid world model.
3 Use the grid world model to create your own grid world environment.
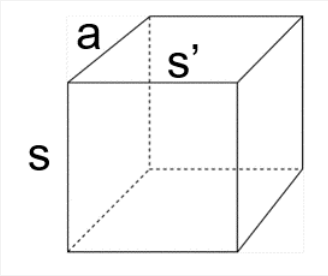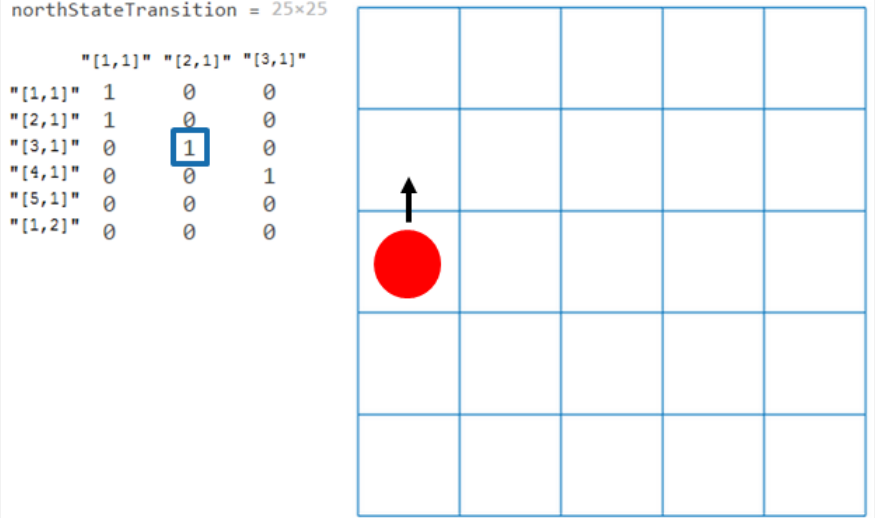
Grid World Model
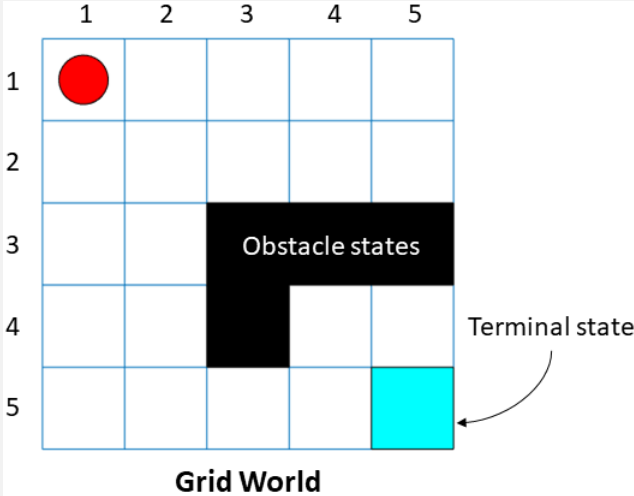
You can create your own grid world model using the `createGridWorld` function. Specify the grid size when creating the `GridWorld` model object.

The `GridWorld` object has the following properties.

Property	Read-Only	Description
GridSize	Yes	Dimensions of the grid world, displayed as an *m*-by-*n* array. Here, *m* represents the number of grid rows and *n* is the number of grid columns.

Property	Read-Only	Description
CurrentState	No	Name of the current state of the agent, specified as a string. You can use this property to set the initial state of the agent. The agent always starts from cell `[1,1]` by default. The agent starts from the `CurrentState` once you use the reset function in the `rlMDPEnv` environment object.

States	Yes	A string vector containing the state names of the grid world. For instance, for a 2-by-2 grid world model `GW`, specify the following:
		
		`GW.States = ["[1,1]";`
		`            "[2,1]";`
		`            "[1,2]";`
		`            "[2,2]"];`

Actions	Yes	A string vector containing the list of possible actions that the agent can use. You can set the actions when you create the grid world model by using the `moves` argument: `GW = createGridWorld(m,n,moves)` Specify `moves` as either `'Standard'` or `'Kings'`. {TABLE_BELOW}

moves	Gw.Actions
'Standard'	['N';'S';'E';'W']
'Kings'	['N';'S';'E';'W';'NE';'NW';'SE';'SW']

Property	Read-Only	Description
T	No	State transition matrix, specified as a 3-D array. T is a probability matrix that indicates the likelihood of the agent moving from the current state s to any possible next state s' by performing action a. T can be denoted as $T(s, s', a) = probability(s'|s, a)$. For instance, consider a 5-by-5 deterministic grid world object GW with the agent in cell [3,1]. View the state transition matrix for the north direction. `northStateTransition = GW.T(:,:,1)` From the above figure, the value of `northStateTransition(3,2)` is 1 since the agent moves from cell [3,1] to cell [2,1] with action 'N'. A probability of 1 indicates that from a given state, if the agent goes north, it has a 100% chance of moving one cell north on the grid. For an example showing how to set up the state transition matrix, see "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8.

Property	Read-Only	Description
R	No	Reward transition matrix, specified as a 3-D array. R determines how much reward the agent receives after performing an action in the environment. R has the same shape and size as the state transition matrix T. The reward transition matrix R can be denoted as $$r = R(s, s', a).$$ Set up R such that there is a reward to the agent after every action. For instance, you can set up a positive reward if the agent transitions over obstacle states and when it reaches the terminal state. You can also set up a default reward of -11 for all actions the agent takes, independent of the current state and next state. For an example that show how to set up the reward transition matrix, see "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8.
ObstacleStates	No	ObstacleStates are states that cannot be reached in the grid world, specified as a string vector. Consider the following 5-by-5 grid world model GW. **Grid World** The black cells are obstacle states, and you can specify them using the following syntax: `GW.ObstacleStates = ["[3,3]";"[3,4]";"[3,5]";"[4,3]"];` For a workflow example, see "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8.

Property	Read-Only	Description
`TerminalStates`	No	`TerminalStates` are the final states in the grid world, specified as a string vector. Consider the previous 5-by-5 grid world model `GW`. The blue cell is the terminal state and you can specify it by: `GW.TerminalStates = "[5,5]";` For a workflow example, see "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8.

Grid World Environment

You can create a Markov decision process (MDP) environment using `rlMDPEnv` from the grid world model from the previous step. 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. The agent uses the grid world environment object `rlMDPEnv` to interact with the grid world model object `GridWorld`.

For more information, see `rlMDPEnv` and "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8.

See Also

`createGridWorld` | `rlMDPEnv` | `rlPredefinedEnv`

More About

•   "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8

Create MATLAB Environment Using Custom Functions

This example shows how to create a cart-pole environment by supplying custom dynamic functions in MATLAB®.

Using the `rlFunctionEnv` function, you can create a MATLAB reinforcement learning environment from an observation specification, an action specification, and user-defined `step` and `reset` functions. You can then train a reinforcement learning agent in this environment. The necessary `step` and `reset` functions are already defined for this example.

Creating an environment using custom functions is useful for environments with less complex dynamics, environments with no special visualization requirements, or environments with interfaces to third-party libraries. For more complex environments, you can create an environment object using a template class. For more information, see "Create Custom MATLAB Environment from Template" on page 2-40.

For more information on creating reinforcement learning environments, see "Create MATLAB Environments for Reinforcement Learning" on page 2-2 and "Create Simulink Environments for Reinforcement Learning" on page 2-5.

Cart-Pole MATLAB Environment

The cart-pole environment is a pole attached to an unactuated joint on a cart, which moves along a frictionless track. The training goal is to make the pendulum stand upright without falling over.
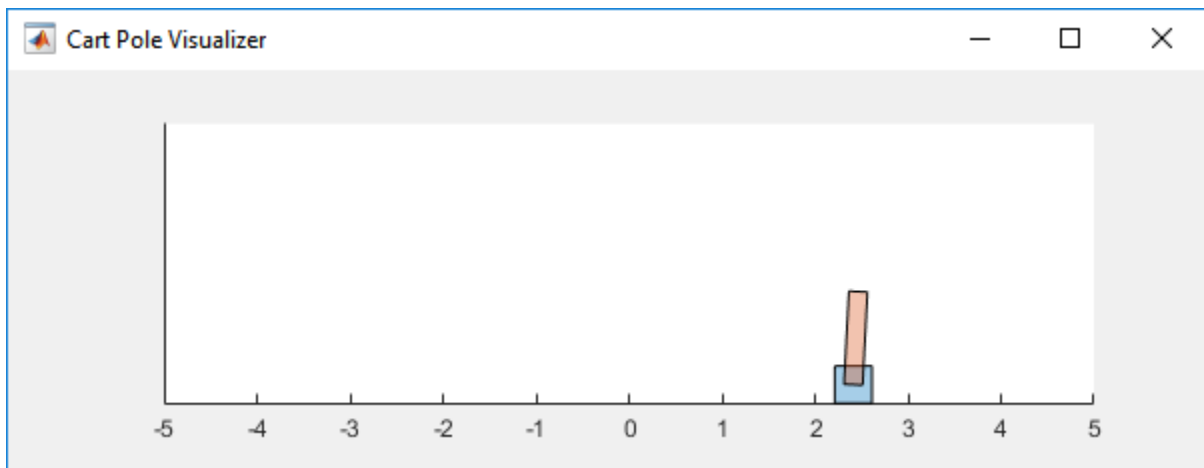
For this environment:

- The upward balanced pendulum position is `0` radians, and the downward hanging position is `pi` radians.

- The pendulum starts upright with an initial angle that is between –0.5 and 0.05.

- The force action signal from the agent to the environment is from –10 to 10 N.

- The observations from the environment are the cart position, cart velocity, pendulum angle, and pendulum angle derivative.

- The episode terminates if the pole is more than 12 degrees from vertical, or if the cart moves more than 2.4 m from the original position.

- A reward of +1 is provided for every time step that the pole remains upright. A penalty of –10 is applied when the pendulum falls.

For more information on this model, see "Load Predefined Control System Environments" on page 2-15.

Observation and Action Specifications

The observations from the environment are the cart position, cart velocity, pendulum angle, and pendulum angle derivative.

```
ObservationInfo = rlNumericSpec([4 1]);
ObservationInfo.Name = 'CartPole States';
ObservationInfo.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` or `10` N.

```
ActionInfo = rlFiniteSetSpec([-10 10]);
ActionInfo.Name = 'CartPole Action';
```

For more information on specifying environment actions and observations, see `rlNumericSpec` and `rlFiniteSetSpec`.

Create Environment Using Function Names

To define a custom environment, first specify the custom `step` and `reset` functions. These functions must be in your current working folder or on the MATLAB path.

The custom `reset` function sets the default state of the environment. This function must have the following signature.

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

To pass information from one step to the next, such as the environment state, use `LoggedSignals`. For this example, `LoggedSignals` contains the states of the cart-pole environment: the position and velocity of the cart, the pendulum angle, and the pendulum angle derivative. The `reset` function sets the cart angle to a random value each time the environment is reset.

For this example, use the custom reset function defined in `myResetFunction.m`.

```
type myResetFunction.m

function [InitialObservation, LoggedSignal] = myResetFunction()
% Reset function to place custom cart-pole environment into a random
% initial state.

% Theta (randomize)
T0 = 2 * 0.05 * rand() - 0.05;
% Thetadot
Td0 = 0;
% X
X0 = 0;
% Xdot
Xd0 = 0;

% Return initial environment state variables as logged signals.
```

```
LoggedSignal.State = [X0;Xd0;T0;Td0];
InitialObservation = LoggedSignal.State;

end
```

The custom `step` function specifies how the environment advances to the next state based on a given action. This function must have the following signature.

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

To get the new state, the environment applies the dynamic equation to the current state stored in `LoggedSignals`, which is similar to giving an initial condition to a differential equation. The new state is stored in `LoggedSignals` and returned as an output.

For this example, use the custom step function defined in `myStepFunction.m`. For implementation simplicity, this function redefines physical constants, such as the cart mass, every time `step` is executed.

```
type myStepFunction.m
```

```
function [NextObs,Reward,IsDone,LoggedSignals] = myStepFunction(Action,LoggedSignals)
% Custom step function to construct cart-pole environment for the function
% name case.
%
% This function applies the given action to the environment and evaluates
% the system dynamics for one simulation step.

% Define the environment constants.

% Acceleration due to gravity in m/s^2
Gravity = 9.8;
% Mass of the cart
CartMass = 1.0;
% Mass of the pole
PoleMass = 0.1;
% Half the length of the pole
HalfPoleLength = 0.5;
% Max force the input can apply
MaxForce = 10;
% Sample time
Ts = 0.02;
% Pole angle at which to fail the episode
AngleThreshold = 12 * pi/180;
% Cart distance at which to fail the episode
DisplacementThreshold = 2.4;
% Reward each time step the cart-pole is balanced
RewardForNotFalling = 1;
% Penalty when the cart-pole fails to balance
PenaltyForFalling = -10;

% Check if the given action is valid.
if ~ismember(Action,[-MaxForce MaxForce])
    error('Action must be %g for going left and %g for going right.',...
        -MaxForce,MaxForce);
end
Force = Action;

% Unpack the state vector from the logged signals.
```

```
    State = LoggedSignals.State;
    XDot = State(2);
    Theta = State(3);
    ThetaDot = State(4);

    % Cache to avoid recomputation.
    CosTheta = cos(Theta);
    SinTheta = sin(Theta);
    SystemMass = CartMass + PoleMass;
    temp = (Force + PoleMass*HalfPoleLength*ThetaDot*ThetaDot*SinTheta)/SystemMass;

    % Apply motion equations.
    ThetaDotDot = (Gravity*SinTheta - CosTheta*temp) / ...
        (HalfPoleLength*(4.0/3.0 - PoleMass*CosTheta*CosTheta/SystemMass));
    XDotDot  = temp - PoleMass*HalfPoleLength*ThetaDotDot*CosTheta/SystemMass;

    % Perform Euler integration.
    LoggedSignals.State = State + Ts.*[XDot;XDotDot;ThetaDot;ThetaDotDot];

    % Transform state to observation.
    NextObs = LoggedSignals.State;

    % Check terminal condition.
    X = NextObs(1);
    Theta = NextObs(3);
    IsDone = abs(X) > DisplacementThreshold || abs(Theta) > AngleThreshold;

    % Get reward.
    if ~IsDone
        Reward = RewardForNotFalling;
    else
        Reward = PenaltyForFalling;
    end

end
```

Construct the custom environment using the defined observation specification, action specification, and function names.

```
env = rlFunctionEnv(ObservationInfo,ActionInfo,'myStepFunction','myResetFunction');
```

To verify the operation of your environment, `rlFunctionEnv` automatically calls `validateEnvironment` after creating the environment.

Create Environment Using Function Handles

You can also define custom functions that have additional input arguments beyond the minimum required set. For example, to pass the additional arguments `arg1` and `arg2` to both the step and rest function, use the following code.

```
[InitialObservation,LoggedSignals] = myResetFunction(arg1,arg2)
[Observation,Reward,IsDone,LoggedSignals] = myStepFunction(Action,LoggedSignals,arg1,arg2)
```

To use these functions with `rlFunctionEnv`, you must use anonymous function handles.

```
ResetHandle = @()myResetFunction(arg1,arg2);
StepHandle = @(Action,LoggedSignals) myStepFunction(Action,LoggedSignals,arg1,arg2);
```

For more information, see "Anonymous Functions".

Using additional input arguments can create a more efficient environment implementation. For example, `myStepFunction2.m` contains a custom `step` function that takes the environment constants as an input argument (`envConstants`). By doing so, this function avoids redefining the environment constants at each step.

type myStepFunction2.m

```
function [NextObs,Reward,IsDone,LoggedSignals] = myStepFunction2(Action,LoggedSignals,EnvConstant
% Custom step function to construct cart-pole environment for the function
% handle case.
%
% This function applies the given action to the environment and evaluates
% the system dynamics for one simulation step.

% Check if the given action is valid.
if ~ismember(Action,[-EnvConstants.MaxForce EnvConstants.MaxForce])
    error('Action must be %g for going left and %g for going right.',...
        -EnvConstants.MaxForce,EnvConstants.MaxForce);
end
Force = Action;

% Unpack the state vector from the logged signals.
State = LoggedSignals.State;
XDot = State(2);
Theta = State(3);
ThetaDot = State(4);

% Cache to avoid recomputation.
CosTheta = cos(Theta);
SinTheta = sin(Theta);
SystemMass = EnvConstants.MassCart + EnvConstants.MassPole;
temp = (Force + EnvConstants.MassPole*EnvConstants.Length*ThetaDot*ThetaDot*SinTheta)/SystemMass

% Apply motion equations.
ThetaDotDot = (EnvConstants.Gravity*SinTheta - CosTheta*temp)...
    / (EnvConstants.Length*(4.0/3.0 - EnvConstants.MassPole*CosTheta*CosTheta/SystemMass));
XDotDot  = temp - EnvConstants.MassPole*EnvConstants.Length*ThetaDotDot*CosTheta/SystemMass;

% Perform Euler integration.
LoggedSignals.State = State + EnvConstants.Ts.*[XDot;XDotDot;ThetaDot;ThetaDotDot];

% Transform state to observation.
NextObs = LoggedSignals.State;

% Check terminal condition.
X = NextObs(1);
Theta = NextObs(3);
IsDone = abs(X) > EnvConstants.XThreshold || abs(Theta) > EnvConstants.ThetaThresholdRadians;

% Get reward.
if ~IsDone
    Reward = EnvConstants.RewardForNotFalling;
else
    Reward = EnvConstants.PenaltyForFalling;
end

end
```

Create the structure that contains the environment constants.

```
% Acceleration due to gravity in m/s^2
envConstants.Gravity = 9.8;
% Mass of the cart
envConstants.MassCart = 1.0;
% Mass of the pole
envConstants.MassPole = 0.1;
% Half the length of the pole
envConstants.Length = 0.5;
% Max force the input can apply
envConstants.MaxForce = 10;
% Sample time
envConstants.Ts = 0.02;
% Angle at which to fail the episode
envConstants.ThetaThresholdRadians = 12 * pi/180;
% Distance at which to fail the episode
envConstants.XThreshold = 2.4;
% Reward each time step the cart-pole is balanced
envConstants.RewardForNotFalling = 1;
% Penalty when the cart-pole fails to balance
envConstants.PenaltyForFalling = -5;
```

Create an anonymous function handle to the custom `step` function, passing `envConstants` as an additional input argument. Because `envConstants` is available at the time that `StepHandle` is created, the function handle includes those values. The values persist within the function handle even if you clear the variables.

```
StepHandle = @(Action,LoggedSignals) myStepFunction2(Action,LoggedSignals,envConstants);
```

Use the same `reset` function, specifying it as a function handle rather than by using its name.

```
ResetHandle = @myResetFunction;
```

Create the environment using the custom function handles.

```
env2 = rlFunctionEnv(ObservationInfo,ActionInfo,StepHandle,ResetHandle);
```

Validate Custom Functions

Before you train an agent in your environment, the best practice is to validate the behavior of your custom functions. To do so, you can initialize your environment using the `reset` function and run one simulation step using the `step` function. For reproducibility, set the random generator seed before validation.

Validate the environment created using function names.

```
rng(0);
InitialObs = reset(env)

InitialObs = 4×1

        0
        0
   0.0315
        0
```

```
[NextObs,Reward,IsDone,LoggedSignals] = step(env,10);
NextObs
```

NextObs = 4×1

```
        0
   0.1947
   0.0315
  -0.2826
```

Validate the environment created using function handles.

```
rng(0);
InitialObs2 = reset(env2)
```

InitialObs2 = 4×1

```
        0
        0
   0.0315
        0
```

```
[NextObs2,Reward2,IsDone2,LoggedSignals2] = step(env2,10);
NextObs
```

NextObs = 4×1

```
        0
   0.1947
   0.0315
  -0.2826
```

Both environments initialize and simulate successfully, producing the same state values in `NextObs`.

See Also
`rlFunctionEnv`

More About
- "Create MATLAB Environments for Reinforcement Learning" on page 2-2
- "Create Custom MATLAB Environment from Template" on page 2-40

Create Custom MATLAB Environment from Template

You can define a custom reinforcement learning environment by creating and modifying a template environment class. You can use a custom template environment to:

- Implement more complex environment dynamics.
- Add custom visualizations to your environment.
- Create an interface to third-party libraries defined in languages such as C++, Java®, or Python®. For more information, see "External Language Interfaces".

For more information about creating MATLAB classes, see "User-Defined Classes".

You can create less complex custom reinforcement learning environments using custom functions, as described in "Create MATLAB Environment Using Custom Functions" on page 2-33.

Create Template Class

To define your custom environment, first create the template class file, specifying the name of the class. For this example, name the class `MyEnvironment`.

```
rlCreateEnvTemplate("MyEnvironment")
```

The software creates and opens the template class file. The template class is a subclass of the `rl.env.MATLABEnvironment` abstract class, as shown in the class definition at the start of the template file. This abstract class is the same one used by the other MATLAB reinforcement learning environment objects.

```
classdef MyEnvironment < rl.env.MATLABEnvironment
```

By default, the template class implements a simple cart-pole balancing model similar to the cart-pole predefined environments described in "Load Predefined Control System Environments" on page 2-15.

To define your environment dynamics modify the template class, specify the following:

- Environment properties
- Required environment methods
- Optional environment methods

Environment Properties

In the `properties` section of the template, specify any parameters necessary for creating and simulating the environment. These parameters can include:

- Physical constants — The sample environment defines the acceleration due to gravity (`Gravity`).
- Environment geometry — The sample environment defines the cart and pole masses (`CartMass` and `PoleMass`) and the half-length of the pole (`HalfPoleLength`).
- Environment constraints — The sample environment defines the pole angle and cart distance thresholds (`AngleThreshold` and `DisplacementThreshold`). The environment uses these values to detect when a training episode is finished.
- Variables required for evaluating the environment — The sample environment defines the state vector (`State`) and a flag for indicating when an episode is finished (`IsDone`).

- Constants for defining the actions or observation spaces — The sample environment defines the maximum force for the action space (`MaxForce`).
- Constants for calculating the reward signal — The sample environment defines the constants `RewardForNotFalling` and `PenaltyForFalling`.

```
properties
    % Specify and initialize the necessary properties of the environment
    % Acceleration due to gravity in m/s^2
    Gravity = 9.8

    % Mass of the cart
    CartMass = 1.0

    % Mass of the pole
    PoleMass = 0.1

    % Half the length of the pole
    HalfPoleLength = 0.5

    % Max force the input can apply
    MaxForce = 10

    % Sample time
    Ts = 0.02

    % Angle at which to fail the episode (radians)
    AngleThreshold = 12 * pi/180

    % Distance at which to fail the episode
    DisplacementThreshold = 2.4

    % Reward each time step the cart-pole is balanced
    RewardForNotFalling = 1

    % Penalty when the cart-pole fails to balance
    PenaltyForFalling = -10
end

properties
    % Initialize system state [x,dx,theta,dtheta]'
    State = zeros(4,1)
end

properties(Access = protected)
    % Initialize internal flag to indicate episode termination
    IsDone = false
end
```

Required Functions

A reinforcement learning environment requires the following functions to be defined. The `getObservationInfo`, `getActionInfo`, `sim`, and `validateEnvironment` functions are already defined in the base abstract class. To create your environment, you must define the constructor, `reset`, and `step` functions.

Function	Description
`getObservationInfo`	Return information about the environment observations
`getActionInfo`	Return information about the environment actions
`sim`	Simulate the environment with an agent
`validateEnvironment`	Validate the environment by calling the `reset` function and simulating the environment for one time step using `step`
`reset`	Initialize the environment state and clean up any visualization

Function	Description
step	Apply an action, simulate the environment for one step, and output the observations and rewards; also, set a flag indicating whether the episode is complete
Constructor function	A function with the same name as the class that creates an instance of the class

Sample Constructor Function

The sample cart-pole constructor function creates the environment by:

- Defining the action and observation specifications. For more information about creating these specifications, see rlNumericSpec and rlFiniteSetSpec.
- Calling the constructor of the base abstract class.

```
function this = MyEnvironment()
    % Initialize observation settings
    ObservationInfo = rlNumericSpec([4 1]);
    ObservationInfo.Name = 'CartPole States';
    ObservationInfo.Description = 'x, dx, theta, dtheta';

    % Initialize action settings
    ActionInfo = rlFiniteSetSpec([-1 1]);
    ActionInfo.Name = 'CartPole Action';

    % The following line implements built-in functions of the RL environment
    this = this@rl.env.MATLABEnvironment(ObservationInfo,ActionInfo);

    % Initialize property values and precompute necessary values
    updateActionInfo(this);
end
```

This sample constructor function does not include any input arguments. However, you can add input arguments for your custom constructor.

Sample reset Function

The sample cart-pole reset function sets the initial condition of the model and returns the initial values of the observations. It also generates a notification that the environment has been updated by calling the envUpdatedCallback function, which is useful for updating the environment visualization.

```
% Reset environment to initial state and return initial observation
function InitialObservation = reset(this)
    % Theta (+- .05 rad)
    T0 = 2 * 0.05 * rand - 0.05;
    % Thetadot
    Td0 = 0;
    % X
    X0 = 0;
    % Xdot
    Xd0 = 0;

    InitialObservation = [X0;Xd0;T0;Td0];
    this.State = InitialObservation;

    % (Optional) Use notifyEnvUpdated to signal that the
    % environment is updated (for example, to update the visualization)
    notifyEnvUpdated(this);
end
```

Sample step Function

The sample cart-pole `step` function:

- Processes the input action.
- Evaluates the environment dynamic equations for one time step.
- Computes and returns the updated observations.
- Computes and returns the reward signal.
- Checks if the episode is complete and returns the `IsDone` signal as appropriate.
- Generates a notification that the environment has been updated.

```
function [Observation,Reward,IsDone,LoggedSignals] = step(this,Action)
    LoggedSignals = [];

    % Get action
    Force = getForce(this,Action);

    % Unpack state vector
    XDot = this.State(2);
    Theta = this.State(3);
    ThetaDot = this.State(4);

    % Cache to avoid recomputation
    CosTheta = cos(Theta);
    SinTheta = sin(Theta);
    SystemMass = this.CartMass + this.PoleMass;
    temp = (Force + this.PoleMass*this.HalfPoleLength*ThetaDot^2*SinTheta)...
        /SystemMass;

    % Apply motion equations
    ThetaDotDot = (this.Gravity*SinTheta - CosTheta*temp)...
        / (this.HalfPoleLength*(4.0/3.0 - this.PoleMass*CosTheta*CosTheta/SystemMass));
    XDotDot  = temp - this.PoleMass*this.HalfPoleLength*ThetaDotDot*CosTheta/SystemMass;

    % Euler integration
    Observation = this.State + this.Ts.*[XDot;XDotDot;ThetaDot;ThetaDotDot];

    % Update system states
    this.State = Observation;

    % Check terminal condition
    X = Observation(1);
    Theta = Observation(3);
    IsDone = abs(X) > this.DisplacementThreshold || abs(Theta) > this.AngleThreshold;
    this.IsDone = IsDone;

    % Get reward
    Reward = getReward(this);

    % (Optional) Use notifyEnvUpdated to signal that the
    % environment has been updated (for example, to update the visualization)
    notifyEnvUpdated(this);
end
```

Optional Functions

You can define any other functions in your template class as required. For example, you can create helper functions that are called by either `step` or `reset`. The cart-pole template model implements a `getReward` function for computing the reward at each time step.

```
function Reward = getReward(this)
    if ~this.IsDone
        Reward = this.RewardForNotFalling;
```

```matlab
        else
            Reward = this.PenaltyForFalling;
        end
end
```

Environment Visualization

You can add a visualization to your custom environment by implementing the `plot` function. In the `plot` function:

- Create a figure or an instance of a visualizer class of your own implementation. For this example, you create a figure and store a handle to the figure within the environment object.

- Call the `envUpdatedCallback` function.

```matlab
function plot(this)
    % Initiate the visualization
    this.Figure = figure('Visible','on','HandleVisibility','off');
    ha = gca(this.Figure);
    ha.XLimMode = 'manual';
    ha.YLimMode = 'manual';
    ha.XLim = [-3 3];
    ha.YLim = [-1 2];
    hold(ha,'on');
    % Update the visualization
    envUpdatedCallback(this)
end
```

For this example, store the handle to the figure as a protected property of the environment object.

```matlab
properties(Access = protected)
    % Initialize internal flag to indicate episode termination
    IsDone = false

    % Handle to figure
    Figure
end
```

In the `envUpdatedCallback`, plot the visualization to the figure or use your custom visualizer object. For example, check if the figure handle has been set. If it has, then plot the visualization.

```matlab
function envUpdatedCallback(this)
    if ~isempty(this.Figure) && isvalid(this.Figure)
        % Set visualization figure as the current figure
        ha = gca(this.Figure);

        % Extract the cart position and pole angle
        x = this.State(1);
        theta = this.State(3);

        cartplot = findobj(ha,'Tag','cartplot');
        poleplot = findobj(ha,'Tag','poleplot');
        if isempty(cartplot) || ~isvalid(cartplot) ...
                || isempty(poleplot) || ~isvalid(poleplot)
            % Initialize the cart plot
            cartpoly = polyshape([-0.25 -0.25 0.25 0.25],[-0.125 0.125 0.125 -0.125]);
            cartpoly = translate(cartpoly,[x 0]);
            cartplot = plot(ha,cartpoly,'FaceColor',[0.8500 0.3250 0.0980]);
            cartplot.Tag = 'cartplot';

            % Initialize the pole plot
            L = this.HalfPoleLength*2;
```

```
        polepoly = polyshape([-0.1 -0.1 0.1 0.1],[0 L L 0]);
        polepoly = translate(polepoly,[x,0]);
        polepoly = rotate(polepoly,rad2deg(theta),[x,0]);
        poleplot = plot(ha,polepoly,'FaceColor',[0 0.4470 0.7410]);
        poleplot.Tag = 'poleplot';
    else
        cartpoly = cartplot.Shape;
        polepoly = poleplot.Shape;
    end

    % Compute the new cart and pole position
    [cartposx,~] = centroid(cartpoly);
    [poleposx,poleposy] = centroid(polepoly);
    dx = x - cartposx;
    dtheta = theta - atan2(cartposx-poleposx,poleposy-0.25/2);
    cartpoly = translate(cartpoly,[dx,0]);
    polepoly = translate(polepoly,[dx,0]);
    polepoly = rotate(polepoly,rad2deg(dtheta),[x,0.25/2]);

    % Update the cart and pole positions on the plot
    cartplot.Shape = cartpoly;
    poleplot.Shape = polepoly;

    % Refresh rendering in the figure window
    drawnow();
    end
end
```

The environment calls the `envUpdatedCallback` function, and therefore updates the visualization, whenever the environment is updated.

Create Custom Environment

After you define your custom environment class, create an instance of it in the MATLAB workspace. At the command line, type the following.

```
env = MyEnvironment;
```

If your constructor has input arguments, specify them after the class name. For example, `MyEnvironment(arg1,arg2)`.

After you create your environment, the best practice is to validate the environment dynamics. To do so, use the `validateEnvironment` function, which prints an error to the command window if your environment implementation has any issues.

```
validateEnvironment(env)
```

After validating the environment object, you can use it to train a reinforcement learning agent. For more information on training agents, see "Train Reinforcement Learning Agents" on page 5-2.

See Also
`rlCreateEnvTemplate` | `train`

More About
- "Create MATLAB Environments for Reinforcement Learning" on page 2-2
- "Create MATLAB Environment Using Custom Functions" on page 2-33
- "Define Reward Signals" on page 2-7

Water Tank Reinforcement Learning Environment Model

This example shows how to create a water tank reinforcement learning Simulink® environment that contains an RL Agent block in the place of a controller for the water level in a tank. To simulate this environment, you must create an agent and specify that agent in the RL Agent block. For an example that trains an agent using this environment, see "Create Simulink Environment and Train Agent" on page 1-19.

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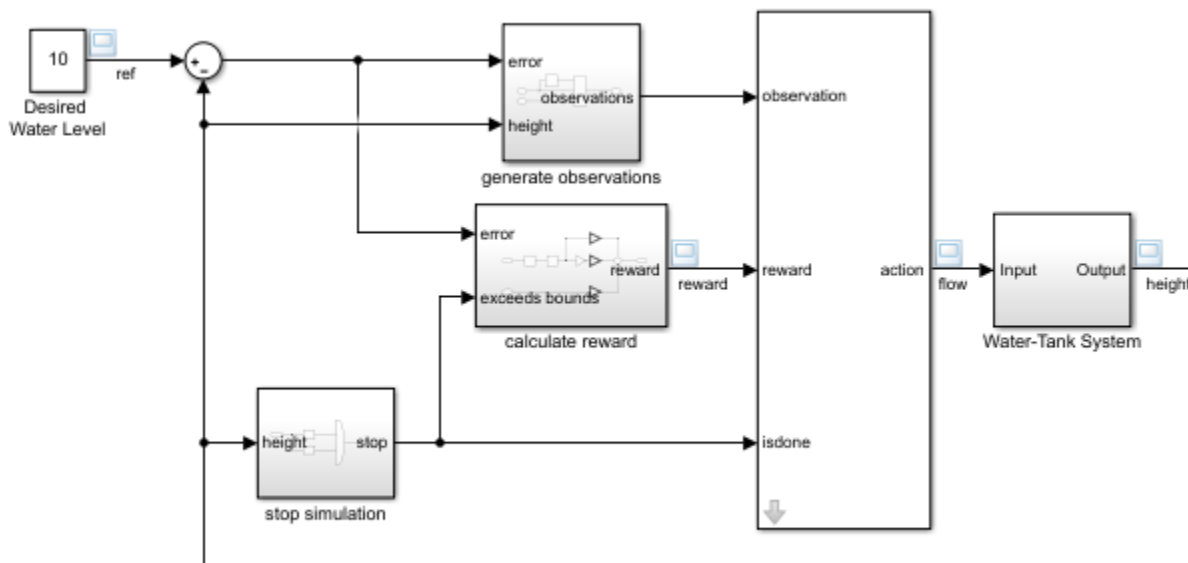

This model already contains an RL Agent block, which connects to the following signals:

- Scalar action output signal
- Vector of observation input signals
- Scalar reward input signal
- Logical input signal for stopping the simulation

Actions and Observations

A reinforcement learning environment receives action signals from the agent and generates observation signals in response to these actions. To create and train an agent, you must create action and observation specification objects.

The action signal for this environment is the flow rate control signal that is sent to the plant. To create a specification object for this continuous action signal, use the `rlNumericSpec` function.

```
actionInfo = rlNumericSpec([1 1]);
actionInfo.Name = 'flow';
```

If the action signal takes one of a discrete set of possible values, create the specification using the `rlFiniteSetSpec` function.

For this environment, there are three observation signals sent to the agent, specified as a vector signal. The observation vector is $[\int e \, dt \;\; e \;\; h]^T$, where:

- h is the height of the water in the tank.
- $e = r - h$, where r is the reference value for the water height.

Compute the observation signals in the generate observations subsystem.

```
open_system([mdl '/generate observations'])
```
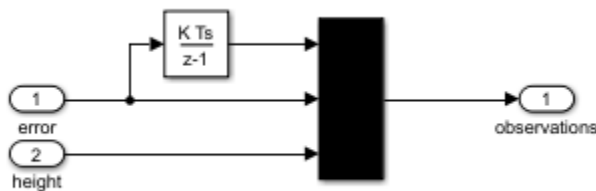

Create a three-element vector of observation specifications. Specify a lower bound of 0 for the water height, leaving the other observation signals unbounded.

```
observationInfo = rlNumericSpec([3 1],...
    'LowerLimit',[-inf -inf 0  ]',...
    'UpperLimit',[ inf  inf inf]');
observationInfo.Name = 'observations';
observationInfo.Description = 'integrated error, error, and measured height';
```

If the actions or observations are represented by bus signals, create specifications using the `bus2RLSpec` function.

Reward Signal

Construct a scalar reward signal. For this example, specify the following reward.

$$reward = 10(|e| < 0.1) - 1(|e| \geq 0.1) - 100(h \leq 0 \, || \, h \geq 20)$$

The reward is positive when the error is below `0.1` and negative otherwise. Also, there is a large reward penalty when the water height is outside the `0` to `20` range.

Construct this reward in the calculate reward subsystem.

```
open_system([mdl '/calculate reward'])
```
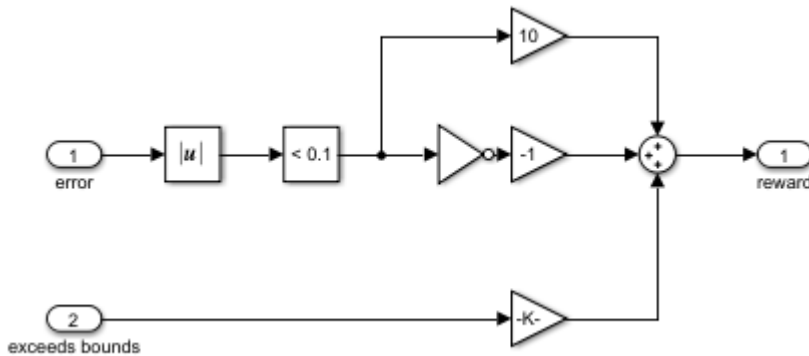
Stop Signal

To terminate training episodes and simulations, specify a logical signal to the `isdone` input port of the block. For this example, terminate the episode if $h \leq 0$ or $h \geq 20$.

Compute this signal in the stop simulation subsystem.

```
open_system([mdl '/stop simulation'])
```
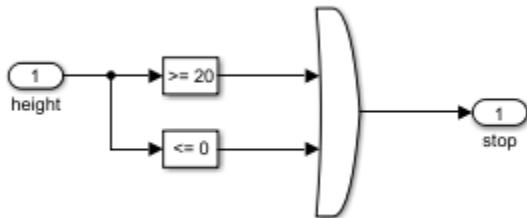

Create Environment Object

Create an environment object for the Simulink model.

```
env = rlSimulinkEnv(mdl,[mdl '/RL Agent'],observationInfo,actionInfo);
```

Reset Function

You can also create a custom reset function that randomizes parameters, variables, or states of the model. In this example, the reset function randomizes the reference signal and the initial water height and sets the corresponding block parameters.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Local Function

```
function in = localResetFcn(in)

% Randomize reference signal
blk = sprintf('rlwatertank/Desired \nWater Level');
h = 3*randn + 10;
while h <= 0 || h >= 20
```

```
    h = 3*randn + 10;
end
in = setBlockParameter(in,blk,'Value',num2str(h));

% Randomize initial height
h = 3*randn + 10;
while h <= 0 || h >= 20
    h = 3*randn + 10;
end
blk = 'rlwatertank/Water-Tank System/H';
in = setBlockParameter(in,blk,'InitialCondition',num2str(h));

end
```

See Also

rlSimulinkEnv

More About

- "Create Simulink Environments for Reinforcement Learning" on page 2-5

Create Agents

Reinforcement Learning Agents

The goal of reinforcement learning is to train an agent to complete a task within an uncertain environment. The agent receives observations and a reward from the environment and sends actions to the environment. The reward is a measure of how successful an action is with respect to completing the task goal.

The agent contains two components: a policy and a learning algorithm.

- The policy is a mapping that selects actions based on the observations from the environment. Typically, the policy is a function approximator with tunable parameters, such as a deep neural network.

- The learning algorithm continuously updates the policy parameters based on the actions, observations, and rewards. The goal of the learning algorithm is to find an optimal policy that maximizes the expected cumulative long-term reward received during the task.
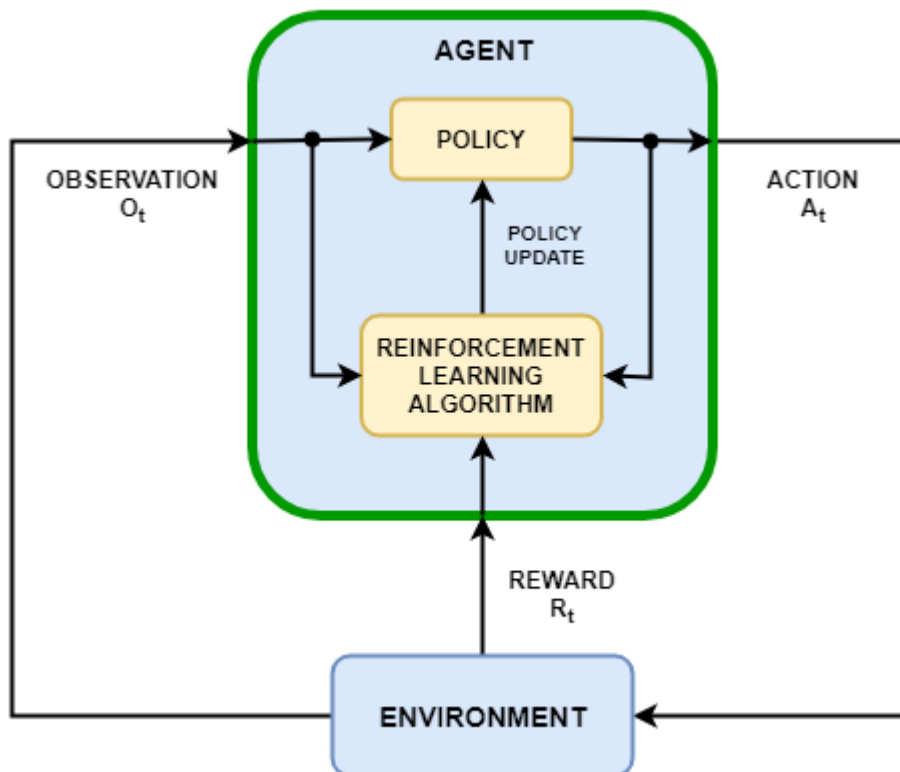
Depending on the learning algorithm, an agent maintains one or more parameterized function approximators for training the policy. Approximators can be used in two ways.

- **Critics** — For a given observation and action, a critic returns as output the expected value of the cumulative long-term reward for the task.

- **Actor** — For a given observation, an actor returns as output the action that maximizes the expected cumulative long-term reward.

Agents that use only critics to select their actions rely on an *indirect policy representation*. These agents are also referred to as *value-based*, and they use an approximator to represent a value

function or Q-value function. In general, these agents work better with discrete action spaces but can become computationally expensive for continuous action spaces.

Agents that use only actors to select their actions rely on a *direct policy representation*. These agents are also referred to as *policy-based*. The policy can be either deterministic or stochastic. In general, these agents are simpler and can handle continuous action spaces, though the training algorithm can be sensitive to noisy measurement and can converge on local minima.

Agents that use both an actor and a critic are referred to as *actor-critic* agents. In these agents, during training, the actor learns the best action to take using feedback from the critic (instead of using the reward directly). At the same time, the critic learns the value function from the rewards so that it can properly criticize the actor. In general, these agents can handle both discrete and continuous action spaces.

Built-In Agents

Reinforcement Learning Toolbox software provides the following built-in agents. You can train these agents in environments with either continuous or discrete observation spaces and the following action spaces.

The following tables summarize the types, action spaces, and representation for all the built-in agents. For each agent, the observation space can be either discrete or continuous.

Built-in Agents: Type and Action Space

Agent	Type	Action Space
"Q-Learning Agents" on page 3-6 (Q)	Value-Based	Discrete
"Deep Q-Network Agents" on page 3-10(DQN)	Value-Based	Discrete
"SARSA Agents" on page 3-8	Value-Based	Discrete
"Policy Gradient Agents" on page 3-13 (PG)	Policy-Based	Discrete or continuous
"Actor-Critic Agents" on page 3-25 (AC)	Actor-Critic	Discrete or continuous
"Proximal Policy Optimization Agents" on page 3-28 (PPO)	Actor-Critic	Discrete or continuous
"Deep Deterministic Policy Gradient Agents" on page 3-17 (DDPG)	Actor-Critic	Continuous
"Twin-Delayed Deep Deterministic Policy Gradient Agents" on page 3-21 (TD3)	Actor-Critic	Continuous
"Soft Actor-Critic Agents" on page 3-32 (SAC)	Actor-Critic	Continuous

Built-in Agents: Representation that You Must Use with Each Agent

Representation	Q, DQN, SARSA	PG	AC, PPO	SAC	DDPG, TD3
Value function critic *V(S)*, which you create using `rlValueRepresentation`		X (if baseline is used)	X		
Q-value function critic *Q(S,A)*, which you create using `rlQValueRepresentation`	X			X	X
Deterministic policy actor *π(S)*, which you create using `rlDeterministicActorRepresentation`					X
Stochastic policy actor *π(S)*, which you create using `rlStochasticActorRepresentation`		X	X	X	

Agent with default networks — All agents except Q-Learning and SARSA support default networks for actors and critics. You can create an agent with default actor and critic representations based on the observation and action specifications from the environment. To do so, perform the following steps.

1   Create observation specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getObservationInfo`.
2   Create action specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getActionInfo`.
3   If needed, specify the number of neurons in each learnable layer or whether to use an LSTM layer. To do so, create an agent initialization option object using `rlAgentInitializationOptions`.
4   If needed, specify agent options by creating an options object set for the specific agent.
5   Create the agent using the corresponding agent creation function. The resulting agent contains the appropriate actor and critic representations listed in the table above. The actor and critic use default agent-specific deep neural networks as internal approximators.

For more information on creating actor and critic function approximators, see "Create Policy and Value Function Representations" on page 4-2.

Choose the Type of Agent

When choosing an agent, a best practice is to start with a simpler (and faster to train) algorithm that is compatible with your action and observation spaces. You can then try progressively more complicated algorithms if the simpler ones do not perform as desired.

- **Discrete action and observation space** — For environments with a discrete action and observation space, the Q-learning agent is the simplest compatible agent, followed by DQN and PPO.

- **Discrete action space and continuous observation space** — For environments with a discrete action space and a continuous observation space, DQN is the simplest compatible agent followed by PPO.

- **Continuous action space** — For environments with both a continuous action and observation space, DDPG is the simplest compatible agent, followed by TD3, PPO, and SAC. For such environments, try DDPG first. In general:

  - TD3 is an improved, more complex version of DDPG.

  - PPO has more stable updates but requires more training.

  - SAC is an improved, more complex version of DDPG that generates stochastic policies.

Custom Agents

You can also train policies using other learning algorithms by creating a custom agent. To do so, you create a subclass of a custom agent class, defining the agent behavior using a set of required and optional methods. For more information, see "Custom Agents" on page 3-37.

See Also

rlACAgent | rlDDPGAgent | rlDQNAgent | rlPGAgent | rlPPOAgent | rlQAgent | rlSACAgent | rlSARSAAgent | rlTD3Agent

More About

- "What Is Reinforcement Learning?" on page 1-3

- "Train Reinforcement Learning Agents" on page 5-2

Q-Learning Agents

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 that trains a critic to estimate the return or future rewards.

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

Q-learning agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Discrete

During training, the agent explores the action space using epsilon-greedy exploration. During each control interval the agent selects a random action with probability ϵ, otherwise it selects an action greedily with respect to the value function with probability 1-ϵ. This greedy action is the action for which the value function is greatest.

Critic Function

To estimate the value function, a Q-learning agent maintains a critic $Q(S,A)$, which is a table or function approximator. The critic takes observation S and action A as inputs and returns the corresponding expectation of the long-term reward.

For more information on creating critics for value function approximation, see "Create Policy and Value Function Representations" on page 4-2.

When training is complete, the trained value function approximator is stored in critic $Q(S,A)$.

Agent Creation

To create a Q-learning agent:

1 Create a critic using an `rlQValueRepresentation` object.
2 Specify agent options using an `rlQAgentOptions` object.
3 Create the agent using an `rlQAgent` object.

Training Algorithm

Q-learning agents use the following training algorithm. To configure the training algorithm, specify options using an `rlQAgentOptions` object.

- Initialize the critic $Q(S,A)$ with random values.
- For each training episode:

    1 Set the initial observation S.
    2 Repeat the following for each step of the episode until S is a terminal state.

        a For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

$$A = \mathop{\mathrm{argmax}}_{A} Q(S, A)$$

       To specify ϵ and its decay rate, use the `EpsilonGreedyExploration` option.

b   Execute action A. Observe the reward R and next observation S'.

c   If S' is a terminal state, set the value function target y to R. Otherwise, set it to

$$y = R + \gamma \mathop{\max}_{A} Q(S', A)$$

       To set the discount factor γ, use the `DiscountFactor` option.

d   Compute the critic parameter update.

$$\Delta Q = y - Q(S, A)$$

e   Update the critic using the learning rate α.

$$Q(S, A) = Q(S, A) + \alpha * \Delta Q$$

       Specify the learning rate when you create the critic representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

f   Set the observation S to S'.

See Also

`rlQAgent` | `rlQAgentOptions`

More About

- "Reinforcement Learning Agents" on page 3-2
- "Create Policy and Value Function Representations" on page 4-2
- "Train Reinforcement Learning Agents" on page 5-2
- "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8

SARSA Agents

The SARSA algorithm is a model-free, online, on-policy reinforcement learning method. A SARSA agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards.

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

SARSA agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Discrete

During training, the agent explores the action space using epsilon-greedy exploration. During each control interval, the agent either selects a random action with probability ϵ or selects an action greedily with respect to the value function with probability 1-ϵ. This greedy action is the action for which the value function is greatest.

Critic Function

To estimate the value function, a SARSA agent maintains a critic $Q(S,A)$, which is a table or function approximator. The critic takes observation S and action A as inputs and returns the corresponding expectation of the long-term reward.

For more information on creating critics for value function approximation, see "Create Policy and Value Function Representations" on page 4-2.

When training is complete, the trained value function approximator is stored in critic $Q(S,A)$.

Agent Creation

To create a SARSA agent:

1   Create a critic using an `rlQValueRepresentation` object.
2   Specify agent options using an `rlSARSAAgentOptions` object.
3   Create the agent using an `rlSARSAAgent` object.

Training Algorithm

SARSA agents use the following training algorithm. To configure the training algorithm, specify options using an `rlSARSAAgentOptions` object.

- Initialize the critic $Q(S,A)$ with random values.
- For each training episode:

    1   Set the initial observation S.
    2   For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

$$A = \underset{A}{\mathrm{argmax}}\, Q(S, A)$$

To specify ϵ and its decay rate, use the `EpsilonGreedyExploration` option.

3 Repeat the following for each step of the episode until S is a terminal state:

    a Execute action A. Observe the reward R and next observation S'.

    b Select an action A' by following the policy from state S'.

$$A' = \max_{A'} Q(S', A')$$

    c If S' is a terminal state, set the value function target y to R. Otherwise, set it to

$$y = R + \gamma Q(S', A')$$

To set the discount factor γ, use the `DiscountFactor` option.

    d Compute the critic parameter update.

$$\Delta Q = y - Q(S, A)$$

    e Update the critic using the learning rate α.

$$Q(S, A) = Q(S, A) + \alpha * \Delta Q$$

Specify the learning rate when you create the critic representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

    f Set the observation S to S'.

    g Set the action A to A'.

See Also

`rlSARSAAgent` | `rlSARSAAgentOptions`

More About

- "Reinforcement Learning Agents" on page 3-2
- "Create Policy and Value Function Representations" on page 4-2
- "Train Reinforcement Learning Agents" on page 5-2
- "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8

Deep Q-Network Agents

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 on Q-learning, see "Q-Learning Agents" on page 3-6.

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

DQN agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Discrete

During training, the agent:

- Updates the critic properties at each time step during learning.
- Explores the action space using epsilon-greedy exploration. During each control interval, the agent either selects a random action with probability ϵ or selects an action greedily with respect to the value function with probability 1-ϵ. This greedy action is the action for which the value function is greatest.
- Stores past experiences using a circular experience buffer. The agent updates the critic based on a mini-batch of experiences randomly sampled from the buffer.

Critic Function

To estimate the value function, a DQN agent maintains two function approximators:

- Critic $Q(S,A)$ — The critic takes observation S and action A as inputs and returns the corresponding expectation of the long-term reward.
- Target critic $Q'(S,A)$ — To improve the stability of the optimization, the agent periodically updates the target critic based on the latest critic parameter values.

Both $Q(S,A)$ and $Q'(S,A)$ have the same structure and parameterization.

For more information on creating critics for value function approximation, see "Create Policy and Value Function Representations" on page 4-2.

When training is complete, the trained value function approximator is stored in critic $Q(S,A)$.

Agent Creation

You can create a DQN agent with a critic representation based on the observation and action specifications from the environment. To do so, perform the following steps.

1   Create observation specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getObservationInfo`.
2   Create action specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getActionInfo`.

3 If needed, specify the number of neurons in each learnable layer or whether to use an LSTM layer. To do so, create an agent initialization option object using `rlAgentInitializationOptions`.

4 If needed, specify agent options using an `rlDQNAgentOptions` object.

5 Create the agent using an `rlDQNAgent` object.

Alternatively, you can create actor and critic representations and use these representations to create your agent. In this case, ensure that the input and output dimensions of the actor and critic representations match the corresponding action and observation specifications of the environment.

1 Create a critic using an `rlQValueRepresentation` object.

2 Specify agent options using an `rlDQNAgentOptions` object.

3 Create the agent using an `rlDQNAgent` object.

DQN agents support critics that use recurrent deep neural networks as functions approximators.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Training Algorithm

DQN agents use the following training algorithm, in which they update their critic model at each time step. To configure the training algorithm, specify options using an `rlDQNAgentOptions` object.

- Initialize the critic $Q(s,a)$ with random parameter values θ_Q, and initialize the target critic with the same values: $\theta_{Q'} = \theta_Q$.

- For each training time step:

  1 For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

  $$A = \underset{A}{\mathrm{argmax}}\, Q(S, A | \theta_Q)$$

  To specify ϵ and its decay rate, use the `EpsilonGreedyExploration` option.

  2 Execute action A. Observe the reward R and next observation S'.

  3 Store the experience (S,A,R,S') in the experience buffer.

  4 Sample a random mini-batch of M experiences (S_i,A_i,R_i,S'_i) from the experience buffer. To specify M, use the `MiniBatchSize` option.

  5 If S'_i is a terminal state, set the value function target y_i to R_i. Otherwise, set it to

  $$A_{\max} = \underset{A'}{\mathrm{argmax}}\, Q(S'_i, A' | \theta_Q)$$
  $$y_i = R_i + \gamma Q'(S'_i, A_{\max} | \theta_{Q'}) \quad \text{(double DQN)}$$

  $$y_i = R_i + \gamma \underset{A'}{\max}\, Q'(S'_i, A' | \theta_{Q'}) \quad \text{(DQN)}$$

  To set the discount factor γ, use the `DiscountFactor` option. To use double DQN, set the `UseDoubleDQN` option to `true`.

6   Update the critic parameters by one-step minimization of the loss L across all sampled experiences.

$$L = \frac{1}{M} \sum_{i=1}^{M} (y_i - Q(S_i, A_i|\theta_Q))^2$$

7   Update the target critic parameters depending on the target update method For more information, see "Target Update Methods" on page 3-12.

8   Update the probability threshold ϵ for selecting a random action based on the decay rate you specify in the `EpsilonGreedyExploration` option.

Target Update Methods

DQN agents update their target critic parameters using one of the following target update methods.

- **Smoothing** — Update the target parameters at every time step using smoothing factor τ. To specify the smoothing factor, use the `TargetSmoothFactor` option.

  $$\theta_{Q'} = \tau\theta_Q + (1 - \tau)\theta_{Q'}$$

- **Periodic** — Update the target parameters periodically without smoothing (`TargetSmoothFactor = 1`). To specify the update period, use the `TargetUpdateFrequency` parameter.

- **Periodic Smoothing** — Update the target parameters periodically with smoothing.

To configure the target update method, create a `rlDQNAgentOptions` object, and set the `TargetUpdateFrequency` and `TargetSmoothFactor` parameters as shown in the following table.

Update Method	TargetUpdateFrequency	TargetSmoothFactor
Smoothing (default)	1	Less than 1
Periodic	Greater than 1	1
Periodic smoothing	Greater than 1	Less than 1

References

[1] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing Atari with Deep Reinforcement Learning." *ArXiv:1312.5602 [Cs]*, December 19, 2013. https://arxiv.org/abs/1312.5602.

See Also

`rlDQNAgent` | `rlDQNAgentOptions`

More About

- "Reinforcement Learning Agents" on page 3-2
- "Create Policy and Value Function Representations" on page 4-2
- "Train Reinforcement Learning Agents" on page 5-2

Policy Gradient Agents

The policy gradient (PG) algorithm is a model-free, online, on-policy reinforcement learning method. A PG agent is a policy-based reinforcement learning agent that directly computes an optimal policy that maximizes the long-term reward.

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

PG agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Discrete or continuous	Discrete or continuous

During training, a PG agent:

- Estimates probabilities of taking each action in the action space and randomly selects actions based on the probability distribution.
- Completes a full training episode using the current policy before learning from the experience and updating the policy parameters.

Actor and Critic Functions

PG agents represent the policy using an actor function approximator $\mu(S)$. The actor takes observation S and returns the probabilities of taking each action in the action space when in state S.

To reduce the variance during gradient estimation, PG agents can use a baseline value function, which is estimated using a critic function approximator, $V(S)$. The critic computes the value function for a given observation state.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Agent Creation

You can create a PG agent with default actor and critic representations based on the observation and action specifications from the environment. To do so, perform the following steps.

1   Create observation specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getObservationInfo`.
2   Create action specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getActionInfo`.
3   If needed, specify the number of neurons in each learnable layer or whether to use an LSTM layer. To do so, create an agent initialization option object using `rlAgentInitializationOptions`.
4   If needed, specify agent options using an `rlPGAgentOptions` object.
5   Create the agent using an `rlPGAgent` object.

Alternatively, you can create actor and critic representations and use these representations to create your agent. In this case, ensure that the input and output dimensions of the actor and critic representations match the corresponding action and observation specifications of the environment.

1. Create an actor representation using an `rlStochasticActorRepresentation` object.
2. If you are using a baseline function, create a critic using an `rlValueRepresentation` object.
3. Specify agent options using the `rlPGAgentOptions` object.
4. Create the agent using an `rlPGAgent` object.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Training Algorithm

PG agents use the REINFORCE (Monte Carlo policy gradient) algorithm either with or without a baseline. To configure the training algorithm, specify options using an `rlPGAgentOptions` object.

REINFORCE Algorithm

1. Initialize the actor $\mu(S)$ with random parameter values θ_μ.
2. For each training episode, generate the episode experience by following actor policy $\mu(S)$. To select an action, the actor generates probabilities for each action in the action space, then the agent randomly selects an action based on the probability distribution. The agent takes actions until it reaches the terminal state S_T. The episode experience consists of the sequence

   $$S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T, S_T$$

   Here, S_t is a state observation, A_{t+1} is an action taken from that state, S_{t+1} is the next state, and R_{t+1} is the reward received for moving from S_t to S_{t+1}.

3. For each state in the episode sequence, that is, for $t = 1, 2, ..., T$-1, calculate the return G_t, which is the discounted future reward.

   $$G_t = \sum_{k=t}^{T} \gamma^{k-t} R_k$$

4. Accumulate the gradients for the actor network by following the policy gradient to maximize the expected discounted reward. If the `EntropyLossWeight` option is greater than zero, then additional gradients are accumulated to minimize the entropy loss function.

   $$d\theta_\mu = \sum_{t=1}^{T-1} G_t \nabla_{\theta_\mu} \ln\mu(S_t|\theta_\mu)$$

5. Update the actor parameters by applying the gradients.

   $$\theta_\mu = \theta_\mu + \alpha d\theta_\mu$$

   Here, α is the learning rate of the actor. Specify the learning rate when you create the actor representation by setting the `LearnRate` option in the `rlRepresentationOptions` object. For simplicity, this step shows a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer you specify using `rlRepresentationOptions`.

6. Repeat steps 2 through 5 for each training episode until training is complete.

REINFORCE with Baseline Algorithm

1. Initialize the actor $\mu(S)$ with random parameter values θ_μ.

2   Initialize the critic $V(S)$ with random parameter values θ_Q.

3   For each training episode, generate the episode experience by following the actor policy $\mu(S)$. The episode experience consists of the sequence

$$S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T, S_T$$

4   For $t = 1, 2, ..., T$:

- Calculate the return G_t, which is the discounted future reward.

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} R_k$$

- Compute the advantage function δ_t using the baseline value function estimate from the critic.

$$\delta_t = G_t - V(S_t|\theta_V)$$

5   Accumulate the gradients for the critic network.

$$d\theta_V = \sum_{t=1}^{T-1} \delta_t \nabla_{\theta_V} V(S_t|\theta_V)$$

6   Accumulate the gradients for the actor network. If the `EntropyLossWeight` option is greater than zero, then additional gradients are accumulated to minimize the entropy loss function.

$$d\theta_\mu = \sum_{t=1}^{T-1} \delta_t \nabla_{\theta_\mu} \ln \mu\left(S_t|\theta_\mu\right)$$

7   Update the critic parameters θ_V.

$$\theta_V = \theta_V + \beta d\theta_V$$

Here, β is the learning rate of the critic. Specify the learning rate when you create the critic representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

8   Update the actor parameters θ_μ.

$$\theta_\mu = \theta_\mu + \alpha d\theta_\mu$$

9   Repeat steps 3 through 8 for each training episode until training is complete.

For simplicity, the actor and critic updates in this algorithm show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer you specify using `rlRepresentationOptions`.

References

[1] Williams, Ronald J. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8, no. 3–4 (May 1992): 229–56. https://doi.org/10.1007/BF00992696.

See Also

`rlPGAgent` | `rlPGAgentOptions`

More About

Deep Deterministic Policy Gradient Agents

The deep deterministic policy gradient (DDPG) algorithm is a model-free, online, off-policy reinforcement learning method. A DDPG agent is an actor-critic reinforcement learning agent that computes an optimal policy that maximizes the long-term reward.

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

DDPG agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Continuous

During training, a DDPG agent:

- Updates the actor and critic properties at each time step during learning.
- Stores past experiences using a circular experience buffer. The agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.
- Perturbs the action chosen by the policy using a stochastic noise model at each training step.

Actor and Critic Functions

To estimate the policy and value function, a DDPG agent maintains four function approximators:

- Actor $\mu(S)$ — The actor takes observation S and returns the corresponding action that maximizes the long-term reward.
- Target actor $\mu'(S)$ — To improve the stability of the optimization, the agent periodically updates the target actor based on the latest actor parameter values.
- Critic $Q(S,A)$ — The critic takes observation S and action A as inputs and returns the corresponding expectation of the long-term reward.
- Target critic $Q'(S,A)$ — To improve the stability of the optimization, the agent periodically updates the target critic based on the latest critic parameter values.

Both $Q(S,A)$ and $Q'(S,A)$ have the same structure and parameterization, and both $\mu(S)$ and $\mu'(S)$ have the same structure and parameterization.

When training is complete, the trained optimal policy is stored in actor $\mu(S)$.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Agent Creation

You can create a DDPG agent with default actor and critic representations based on the observation and action specifications from the environment. To do so, perform the following steps.

1. Create observation specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getObservationInfo`.
2. Create action specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getActionInfo`.

3   If needed, specify the number of neurons in each learnable layer or whether to use an LSTM layer. To do so, create an agent initialization option object using `rlAgentInitializationOptions`.

4   If needed, specify agent options using an `rlDDPGAgentOptions` object.

5   Create the agent using an `rlDDPGAgent` object.

Alternatively, you can create actor and critic representations and use these representations to create your agent. In this case, ensure that the input and output dimensions of the actor and critic representations match the corresponding action and observation specifications of the environment.

1   Create an actor using an `rlDeterministicActorRepresentation` object.

2   Create a critic using an `rlQValueRepresentation` object.

3   Specify agent options using an `rlDDPGAgentOptions` object.

4   Create the agent using an `rlDDPGAgent` object.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Training Algorithm

DDPG agents use the following training algorithm, in which they update their actor and critic models at each time step. To configure the training algorithm, specify options using an `rlDDPGAgentOptions` object.

- Initialize the critic $Q(S,A)$ with random parameter values θ_Q, and initialize the target critic with the same random parameter values: $\theta_{Q'} = \theta_Q$.

- Initialize the actor $\mu(S)$ with random parameter values θ_μ, and initialize the target actor with the same parameter values: $\theta_{\mu'} = \theta_\mu$.

- For each training time step:

  1   For the current observation S, select action $A = \mu(S) + N$, where N is stochastic noise from the noise model. To configure the noise model, use the `NoiseOptions` option.

  2   Execute action A. Observe the reward R and next observation S'.

  3   Store the experience (S,A,R,S') in the experience buffer.

  4   Sample a random mini-batch of M experiences (S_i,A_i,R_i,S'_i) from the experience buffer. To specify M, use the `MiniBatchSize` option.

  5   If S'_i is a terminal state, set the value function target y_i to R_i. Otherwise, set it to

  $$y_i = R_i + \gamma Q'\big(S_i',\, \mu'(S_i'|\theta_\mu)\big|\theta_{Q'}\big)$$

  The value function target is the sum of the experience reward R_i and the discounted future reward. To specify the discount factor γ, use the `DiscountFactor` option.

  To compute the cumulative reward, the agent first computes a next action by passing the next observation S_i' from the sampled experience to the target actor. The agent finds the cumulative reward by passing the next action to the target critic.

  6   Update the critic parameters by minimizing the loss L across all sampled experiences.

  $$L = \frac{1}{M}\sum_{i=1}^{M}\big(y_i - Q(S_i, A_i|\theta_Q)\big)^2$$

7 Update the actor parameters using the following sampled policy gradient to maximize the expected discounted reward.

$$\nabla_{\theta_\mu} J \approx \frac{1}{M} \sum_{i=1}^{M} G_{ai} G_{\mu i}$$

$$G_{ai} = \nabla_A Q(S_i, A | \theta_Q) \quad \text{where } A = \mu(S_i | \theta_\mu)$$

$$G_{\mu i} = \nabla_{\theta_\mu} \mu(S_i | \theta_\mu)$$

Here, G_{ai} is the gradient of the critic output with respect to the action computed by the actor network, and $G_{\mu i}$ is the gradient of the actor output with respect to the actor parameters. Both gradients are evaluated for observation S_i.

8 Update the target actor and critic parameters depending on the target update method. For more information see "Target Update Methods" on page 3-19.

For simplicity, the actor and critic updates in this algorithm show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer you specify using `rlRepresentationOptions`.

Target Update Methods

DDPG agents update their target actor and critic parameters using one of the following target update methods.

- **Smoothing** — Update the target parameters at every time step using smoothing factor τ. To specify the smoothing factor, use the `TargetSmoothFactor` option.

    $$\theta_{Q'} = \tau \theta_Q + (1 - \tau) \theta_{Q'} \quad \text{(critic parameters)}$$

    $$\theta_{\mu'} = \tau \theta_\mu + (1 - \tau) \theta_{\mu'} \quad \text{(actor parameters)}$$

- **Periodic** — Update the target parameters periodically without smoothing (`TargetSmoothFactor = 1`). To specify the update period, use the `TargetUpdateFrequency` parameter.
- **Periodic Smoothing** — Update the target parameters periodically with smoothing.

To configure the target update method, create a `rlDDPGAgentOptions` object, and set the `TargetUpdateFrequency` and `TargetSmoothFactor` parameters as shown in the following table.

Update Method	TargetUpdateFrequency	TargetSmoothFactor
Smoothing (default)	1	Less than 1
Periodic	Greater than 1	1
Periodic smoothing	Greater than 1	Less than 1

References

[1] Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous Control with Deep Reinforcement Learning." *ArXiv:1509.02971 [Cs, Stat]*, September 9, 2015. https://arxiv.org/abs/1509.02971.

See Also

`rlDDPGAgent` | `rlDDPGAgentOptions`

More About

Twin-Delayed Deep Deterministic Policy Gradient Agents

The twin-delayed deep deterministic policy gradient (TD3) algorithm is a model-free, online, off-policy reinforcement learning method. A TD3 agent is an actor-critic reinforcement learning agent that computes an optimal policy that maximizes the long-term reward.

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

The TD3 algorithm is an extension of the DDPG algorithm. DDPG agents can overestimate value functions, which can produce suboptimal policies. To reduce value function overestimation includes the following modifications of the DDPG algorithm.

1   A TD3 agent learns two Q-value functions and uses the minimum value function estimate during policy updates.
2   A TD3 agent updates the policy and targets less frequently than the Q functions.
3   When updating the policy, a TD3 agent adds noise to the target action, which makes the policy less likely to exploit actions with high Q-value estimates.

You can use a TD3 agent to implement one of the following training algorithms, depending on the number of critics you specify.

- TD3 — Train the agent with two Q-value functions. This algorithm implements all three of the preceding modifications.
- Delayed DDPG — Train the agent with a single Q-value function. This algorithm trains a DDPG agent with target policy smoothing and delayed policy and target updates.

TD3 agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Continuous

During training, a TD3 agent:

- Updates the actor and critic properties at each time step during learning.
- Stores past experiences using a circular experience buffer. The agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.
- Perturbs the action chosen by the policy using a stochastic noise model at each training step.

Actor and Critic Functions

To estimate the policy and value function, a TD3 agent maintains the following function approximators:

- Deterministic actor $\mu(S)$ — The actor takes observation S and returns the corresponding action that maximizes the long-term reward.
- Target actor $\mu'(S)$ — To improve the stability of the optimization, the agent periodically updates the target actor based on the latest actor parameter values.
- One or two Q-value critics $Q_k(S,A)$ — The critics take observation S and action A as inputs and returns the corresponding expectation of the long-term reward.

- One or two target critics $Q'_k(S,A)$ — To improve the stability of the optimization, the agent periodically updates the target critics based on the latest parameter values of the critics. The number of target critics matches the number of critics.

Both $\mu(S)$ and $\mu'(S)$ have the same structure and parameterization.

For each critic, $Q_k(S,A)$ and $Q'_k(S,A)$ have the same structure and parameterization.

When using two critics, $Q_1(S,A)$ and $Q_2(S,A)$, each critic can have a different structure, though TD3 works best when the critics have the same structure. When the critics have the same structure, they must have different initial parameter values.

When training is complete, the trained optimal policy is stored in actor $\mu(S)$.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Agent Creation

You can create a TD3 agent with default actor and critic representations based on the observation and action specifications from the environment. To do so, perform the following steps.

1   Create observation specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getObservationInfo`.
2   Create action specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getActionInfo`.
3   If needed, specify the number of neurons in each learnable layer or whether to use an LSTM layer. To do so, create an agent initialization option object using `rlAgentInitializationOptions`.
4   If needed, specify agent options using an `rlTD3AgentOptions` object.
5   Create the agent using an `rlTD3Agent` object.

Alternatively, you can create actor and critic representations and use these representations to create your agent. In this case, ensure that the input and output dimensions of the actor and critic representations match the corresponding action and observation specifications of the environment.

1   Create an actor using an `rlDeterministicActorRepresentation` object.
2   Create one or two critics using `rlQValueRepresentation` objects.
3   Specify agent options using an `rlTD3AgentOptions` object.
4   Create the agent using an `rlTD3Agent` object.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Training Algorithm

TD3 agents use the following training algorithm, in which they update their actor and critic models at each time step. To configure the training algorithm, specify options using an `rlDDPGAgentOptions` object. Here, $K = 2$ is the number of critics and k is the critic index.

- Initialize each critic $Q_k(S,A)$ with random parameter values θ_{Qk}, and initialize each target critic with the same random parameter values: $\theta_{Qk'} = \theta_{Qk}$.

- Initialize the actor $\mu(S)$ with random parameter values θ_μ, and initialize the target actor with the same parameter values: $\theta_{\mu'} = \theta_\mu$.

- For each training time step:

  1 For the current observation S, select action $A = \mu(S) + N$, where N is stochastic noise from the noise model. To configure the noise model, use the `ExplorationModel` option.

  2 Execute action A. Observe the reward R and next observation S'.

  3 Store the experience (S,A,R,S') in the experience buffer.

  4 Sample a random mini-batch of M experiences (S_i,A_i,R_i,S'_i) from the experience buffer. To specify M, use the `MiniBatchSize` option.

  5 If S'_i is a terminal state, set the value function target y_i to R_i. Otherwise, set it to

  $$y_i = R_i + \gamma * \min_k \left(Q_k'\left(S_i', \mathrm{clip}\left(\mu'\left(S_i'\middle|\theta_\mu\right) + \varepsilon\right)\middle|\theta_{Qk'}\right)\right)$$

  The value function target is the sum of the experience reward R_i and the minimum discounted future reward from the critics. To specify the discount factor γ, use the `DiscountFactor` option.

  To compute the cumulative reward, the agent first computes a next action by passing the next observation S'_i from the sampled experience to the target actor. Then, the agent adds noise ε to the computed action using the `TargetPolicySmoothModel`, and clips the action based on the upper and lower noise limits. The agent finds the cumulative rewards by passing the next action to the target critics.

  6 At every time training step, update the parameters of each critic by minimizing the loss L_k across all sampled experiences.

  $$L_k = \frac{1}{M}\sum_{i=1}^{M}\left(y_i - Q_k\left(S_i, A_i\middle|\theta_{Qk}\right)\right)^2$$

  7 Every D_1 steps, update the actor parameters using the following sampled policy gradient to maximize the expected discounted reward. To set D_1, use the `PolicyUpdateFrequency` option.

  $$\nabla_{\theta_\mu}J \approx \frac{1}{M}\sum_{i=1}^{M}G_{ai}G_{\mu i}$$

  $$G_{ai} = \nabla_A \min_k\left(Q_k\left(S_i, A\middle|\theta_Q\right)\right) \quad \text{where } A = \mu\left(S_i\middle|\theta_\mu\right)$$

  $$G_{\mu i} = \nabla_{\theta_\mu}\mu\left(S_i\middle|\theta_\mu\right)$$

  Here, G_{ai} is the gradient of the minimum critic output with respect to the action computed by the actor network, and $G_{\mu i}$ is the gradient of the actor output with respect to the actor parameters. Both gradients are evaluated for observation S_i.

  8 Every D_2 steps, update the target actor and critics depending on the target update method. To specify D_2, use the `TargetUpdateFrequency` option. For more information, see "Target Update Methods" on page 3-24.

For simplicity, the actor and critic updates in this algorithm show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer you specify using `rlRepresentationOptions`.

Target Update Methods

TD3 agents update their target actor and critic parameters using one of the following target update methods.

- **Smoothing** — Update the target parameters at every time step using smoothing factor τ. To specify the smoothing factor, use the `TargetSmoothFactor` option.

  $\theta_{Qk'} = \tau\theta_{Qk} + (1 - \tau)\theta_{Qk'}$ (critic parameters)

  $\theta_{\mu'} = \tau\theta_{\mu} + (1 - \tau)\theta_{\mu'}$ (actor parameters)

- **Periodic** — Update the target parameters periodically without smoothing (`TargetSmoothFactor = 1`). To specify the update period, use the `TargetUpdateFrequency` parameter.

  $\theta_{Qk'} = \theta_{Qk}$

  $\theta_{\mu'} = \theta_{\mu}$

- **Periodic Smoothing** — Update the target parameters periodically with smoothing.

To configure the target update method, create a `rlTD3AgentOptions` object, and set the `TargetUpdateFrequency` and `TargetSmoothFactor` parameters as shown in the following table.

Update Method	`TargetUpdateFrequency`	`TargetSmoothFactor`
Smoothing (default)	1	Less than 1
Periodic	Greater than 1	1
Periodic smoothing	Greater than 1	Less than 1

References

[1] Fujimoto, Scott, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". *ArXiv:1802.09477 [Cs, Stat]*, 22 October 2018. https://arxiv.org/abs/1802.09477.

See Also
`rlTD3Agent` | `rlTD3AgentOptions`

More About
- "Reinforcement Learning Agents" on page 3-2
- "Create Policy and Value Function Representations" on page 4-2
- "Train Reinforcement Learning Agents" on page 5-2
- "Train Biped Robot to Walk Using Reinforcement Learning Agents" on page 5-168

Actor-Critic Agents

You can use the actor-critic (AC) agent, which uses a model-free, online, on-policy reinforcement learning method, to implement actor-critic algorithms, such as A2C and A3C. The goal of this agent is to optimize the policy (actor) directly and train a critic to estimate the return or future rewards [1].

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

AC agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Discrete or continuous	Discrete or continuous

During training, an AC agent:

- Estimates probabilities of taking each action in the action space and randomly selects actions based on the probability distribution.
- Interacts with the environment for multiple steps using the current policy before updating the actor and critic properties.

Actor and Critic Functions

To estimate the policy and value function, an AC agent maintains two function approximators:

- Actor $\mu(S)$ — The actor takes observation S and returns the probabilities of taking each action in the action space when in state S.
- Critic $V(S)$ — The critic takes observation S and returns the corresponding expectation of the discounted long-term reward.

When training is complete, the trained optimal policy is stored in actor $\mu(S)$.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Agent Creation

You can create an AC agent with default actor and critic representations based on the observation and action specifications from the environment. To do so, perform the following steps.

1 Create observation specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getObservationInfo`.

2 Create action specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getActionInfo`.

3 If needed, specify the number of neurons in each learnable layer or whether to use an LSTM layer. To do so, create an agent initialization option object using `rlAgentInitializationOptions`.

4 If needed, specify agent options using an `rlACAgentOptions` object.

5 Create the agent using an `rlACAgent` object.

Alternatively, you can create actor and critic representations and use these representations to create your agent. In this case, ensure that the input and output dimensions of the actor and critic representations match the corresponding action and observation specifications of the environment.

1 Create an actor using an `rlStochasticActorRepresentation` object.
2 Create a critic using an `rlValueRepresentation` object.
3 Specify agent options using an `rlACAgentOptions` object.
4 Create the agent using an `rlACAgent` object.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Training Algorithm

AC agents use the following training algorithm. To configure the training algorithm, specify options using an `rlACAgentOptions` object.

1 Initialize the actor $\mu(S)$ with random parameter values θ_μ.
2 Initialize the critic $V(S)$ with random parameter values θ_V.
3 Generate N experiences by following the current policy. The episode experience sequence is

$$S_{ts}, A_{ts}, R_{ts+1}, S_{ts+1}, ..., S_{ts+N-1}, A_{ts+N-1}, R_{ts+N}, S_{ts+N}$$

Here, S_t is a state observation, A_t is an action taken from that state, S_{t+1} is the next state, and R_{t+1} is the reward received for moving from S_t to S_{t+1}.

When in state S_t, the agent computes the probability of taking each action in the action space using $\mu(S_t)$ and randomly selects action A_t based on the probability distribution.

ts is the starting time step of the current set of N experiences. At the beginning of the training episode, $ts = 1$. For each subsequent set of N experiences in the same training episode, $ts = ts + N$.

For each training episode that does not contain a terminal state, N is equal to the `NumStepsToLookAhead` option value. Otherwise, N is less than `NumStepsToLookAhead` and S_N is the terminal state.

4 For each episode step $t = ts+1, ts+2, ..., ts+N$, compute the return G_t, which is the sum of the reward for that step and the discounted future reward. If S_{ts+N} is not a terminal state, the discounted future reward includes the discounted state value function, computed using the critic network V.

$$G_t = \sum_{k=t}^{ts+N} \left(\gamma^{k-t} R_k\right) + b\gamma^{N-t+1} V(S_{ts+N}|\theta_V)$$

Here, b is 0 if S_{ts+N} is a terminal state and 1 otherwise.

To specify the discount factor γ, use the `DiscountFactor` option.

5 Compute the advantage function D_t.

$$D_t = G_t - V(S_t|\theta_V)$$

6   Accumulate the gradients for the actor network by following the policy gradient to maximize the expected discounted reward.

$$d\theta_\mu = \sum_{t=1}^{N} \nabla_{\theta_\mu} \ln\mu(S_t|\theta_\mu) * D_t$$

7   Accumulate the gradients for the critic network by minimizing the mean squared error loss between the estimated value function $V_{(t)}$ and the computed target return G_t across all N experiences. If the `EntropyLossWeight` option is greater than zero, then additional gradients are accumulated to minimize the entropy loss function.

$$d\theta_V = \sum_{t=1}^{N} \nabla_{\theta_V} (G_t - V(S_t|\theta_V))^2$$

8   Update the actor parameters by applying the gradients.

$$\theta_\mu = \theta_\mu + \alpha d\theta_\mu$$

Here, α is the learning rate of the actor. Specify the learning rate when you create the actor representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

9   Update the critic parameters by applying the gradients.

$$\theta_V = \theta_V + \beta d\theta_V$$

Here, β is the learning rate of the critic. Specify the learning rate when you create the critic representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

10   Repeat steps 3 through 9 for each training episode until training is complete.

For simplicity, the actor and critic updates in this algorithm show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer specified using `rlRepresentationOptions`.

References

[1] Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning." *ArXiv:1602.01783 [Cs]*, February 4, 2016. https://arxiv.org/abs/1602.01783.

See Also
`rlACAgent` | `rlACAgentOptions`

More About

- "Reinforcement Learning Agents" on page 3-2
- "Create Policy and Value Function Representations" on page 4-2
- "Train Reinforcement Learning Agents" on page 5-2

Proximal Policy Optimization Agents

Proximal policy optimization (PPO) is a model-free, online, on-policy, policy gradient reinforcement learning method. This algorithm is a type of policy gradient training that alternates between sampling data through environmental interaction and optimizing a clipped surrogate objective function using stochastic gradient descent. The clipped surrogate objective function improves training stability by limiting the size of the policy change at each step [1].

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

PPO agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Discrete or continuous	Discrete or continuous

During training, a PPO agent:

- Estimates probabilities of taking each action in the action space and randomly selects actions based on the probability distribution.
- Interacts with the environment for multiple steps using the current policy before using mini-batches to update the actor and critic properties over multiple epochs.

Actor and Critic Functions

To estimate the policy and value function, a PPO agent maintains two function approximators:

- Actor $\mu(S)$ — The actor takes observation S and returns the probabilities of taking each action in the action space when in state S.
- Critic $V(S)$ — The critic takes observation S and returns the corresponding expectation of the discounted long-term reward.

When training is complete, the trained optimal policy is stored in actor $\mu(S)$.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Agent Creation

You can create a PPO agent with default actor and critic representations based on the observation and action specifications from the environment. To do so, perform the following steps.

1. Create observation specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getObservationInfo`.
2. Create action specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getActionInfo`.
3. Specify agent options using an `rlPPOAgentOptions` object.
4. Create the agent using an `rlPPOAgent` object.

Alternatively, you can create actor and critic representations and use these representations to create your agent. In this case, ensure that the input and output dimensions of the actor and critic representations match the corresponding action and observation specifications of the environment.

1 Create an actor using an `rlStochasticActorRepresentation` object.

2 Create a critic using an `rlValueRepresentation` object.

3 If needed, specify the number of neurons in each learnable layer or whether to use an LSTM layer. To do so, create an agent initialization option object using `rlAgentInitializationOptions`.

4 If needed, specify agent options using an `rlPPOAgentOptions` object.

5 Create the agent using the `rlPPOAgent` function.

PPO agents support actors and critics that use recurrent deep neural networks as functions approximators.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Training Algorithm

PPO agents use the following training algorithm. To configure the training algorithm, specify options using an `rlPPOAgentOptions` object.

1 Initialize the actor $\mu(S)$ with random parameter values θ_μ.

2 Initialize the critic $V(S)$ with random parameter values θ_V.

3 Generate N experiences by following the current policy. The experience sequence is

$$S_{ts}, A_{ts}, R_{ts+1}, S_{ts+1}, ..., S_{ts+N-1}, A_{ts+N-1}, R_{ts+N}, S_{ts+N}$$

Here, S_t is a state observation, A_t is an action taken from that state, S_{t+1} is the next state, and R_{t+1} is the reward received for moving from S_t to S_{t+1}.

When in state S_t, the agent computes the probability of taking each action in the action space using $\mu(S_t)$ and randomly selects action A_t based on the probability distribution.

ts is the starting time step of the current set of N experiences. At the beginning of the training episode, $ts = 1$. For each subsequent set of N experiences in the same training episode, $ts \leftarrow ts + N$.

For each experience sequence that does not contain a terminal state, N is equal to the `ExperienceHorizon` option value. Otherwise, N is less than `ExperienceHorizon` and S_N is the terminal state.

4 For each episode step $t = ts+1, ts+2, ..., ts+N$, compute the return and advantage function using the method specified by the `AdvantageEstimateMethod` option.

- **Finite Horizon** (`AdvantageEstimateMethod = "finite-horizon"`) — Compute the return G_t, which is the sum of the reward for that step and the discounted future reward [2].

$$G_t = \sum_{k=t}^{ts+N} \left(\gamma^{k-t} R_k \right) + b\gamma^{N-t+1} V(S_{ts+N}|\theta_V)$$

Here, b is 0 if S_{ts+N} is a terminal state and 1 otherwise. That is, if S_{ts+N} is not a terminal state, the discounted future reward includes the discounted state value function, computed using the critic network V.

Compute the advantage function D_t.

$$D_t = G_t - V(S_t|\theta_V)$$

- **Generalized Advantage Estimator** (`AdvantageEstimateMethod = "gae"`) — Compute the advantage function D_t, which is the discounted sum of temporal difference errors [3].

$$D_t = \sum_{k=t}^{ts+N-1} (\gamma\lambda)^{k-t}\delta_k$$

$$\delta_k = R_t + b\gamma V(S_t|\theta_V)$$

Here, b is 0 if S_{ts+N} is a terminal state and 1 otherwise. λ is a smoothing factor specified using the `GAEFactor` option.

Compute the return G_t.

$$G_t = D_t - V(S_t|\theta_V)$$

To specify the discount factor γ for either method, use the `DiscountFactor` option.

5 Learn from mini-batches of experiences over K epochs. To specify K, use the `NumEpoch` option. For each learning epoch:

a Sample a random mini-batch data set of size M from the current set of experiences. To specify M, use the `MiniBatchSize` option. Each element of the mini-batch data set contains a current experience and the corresponding return and advantage function values.

b Update the critic parameters by minimizing the loss L_{critic} across all sampled mini-batch data.

$$L_{critic}(\theta_V) = \frac{1}{M}\sum_{i=1}^{M} (G_i - V(S_i|\theta_V))^2$$

c Update the actor parameters by minimizing the loss L_{actor} across all sampled mini-batch data. If the `EntropyLossWeight` option is greater than zero, then additional entropy loss is added to L_{actor}, which encourages policy exploration.

$$L_{actor}(\theta_\mu) = -\frac{1}{M}\sum_{i=1}^{M} \min(r_i(\theta_\mu) * D_i, c_i(\theta_\mu) * D_i)$$

$$r_i(\theta_\mu) = \frac{\mu_{Ai}(S_i|\theta_\mu)}{\mu_{Ai}(S_i|\theta_{\mu, old})}$$

$$c_i(\theta_\mu) = \max(\min(r_i(\theta_\mu), 1 + \varepsilon), 1 - \varepsilon)$$

Here:

- D_i and G_i are the advantage function and return value for the ith element of the mini-batch, respectively.
- $\mu_i(S_i|\theta_\mu)$ is the probability of taking action A_i when in state S_i, given the updated policy parameters θ_μ.
- $\mu_i(S_i|\theta_{\mu,old})$ is the probability of taking action A_i when in state S_i, given the previous policy parameters ($\theta_{\mu,old}$) from before the current learning epoch.

- ε is the clip factor specified using the `ClipFactor` option.

6 Repeat steps 3 through 5 until the training episode reaches a terminal state.

References

[1] Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal Policy Optimization Algorithms." *ArXiv:1707.06347 [Cs]*, July 19, 2017. https://arxiv.org/abs/1707.06347.

[2] Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning." *ArXiv:1602.01783 [Cs]*, February 4, 2016. https://arxiv.org/abs/1602.01783.

[3] Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. "High-Dimensional Continuous Control Using Generalized Advantage Estimation." *ArXiv:1506.02438 [Cs]*, October 20, 2018. https://arxiv.org/abs/1506.02438.

See Also

`rlPPOAgent` | `rlPPOAgentOptions`

More About

- "Reinforcement Learning Agents" on page 3-2
- "Create Policy and Value Function Representations" on page 4-2
- "Train Reinforcement Learning Agents" on page 5-2

Soft Actor-Critic Agents

The soft actor-critic (SAC) algorithm is a model-free, online, off-policy, actor-critic reinforcement learning method. The SAC algorithm computes an optimal policy that maximizes both the long-term expected reward and the entropy of the policy. The policy entropy is a measure of policy uncertainty given the state. A higher entropy value promotes more exploration. Maximizing both the reward and the entropy balances exploration and exploitation of the environment.

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

Reinforcement Learning Toolbox software, uses two Q-value function critics, which prevents overestimation of the value function. Other implementations of the SAC algorithm use an additional value function critic.

SAC agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Discrete or continuous	Continuous

During training, a SAC agent:

- Updates the actor and critic properties at regular intervals during learning.
- Estimates the mean and standard deviation for selecting an action in the continuous action space and randomly selects actions based on the probability distribution.
- Updates an entropy weight term that balances the expected return and the entropy of the policy.
- Stores past experience using a circular experience buffer. The agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.

Actor and Critic Functions

To estimate the policy and value function, a SAC agent maintains the following function approximators:

- Stochastic actor $\mu(S)$ — The actor takes observation S and returns the action probability density function. The agent randomly selects actions based on this density function.
- One or two Q-value critics $Q_k(S,A)$ — The critics take observation S and action A as inputs and return the corresponding expectation of the value function, which includes both the long-term reward and entropy.
- One or two target critics $Q'_k(S,A)$ — To improve the stability of the optimization, the agent periodically updates the target critics based on the latest parameter values of the critics. The number of target critics matches the number of critics.

When using two critics, $Q_1(S,A)$ and $Q_2(S,A)$, each critic can have a different structure. When the critics have the same structure, they must have different initial parameter values.

For each critic, $Q_k(S,A)$ and $Q'_k(S,A)$ have the same structure and parameterization.

When training is complete, the trained optimal policy is stored in actor $\mu(S)$.

Action Generation

The actor in a SAC agent generates mean and standard deviation outputs. To select an action, the actor first randomly selects an unbounded action from a Gaussian distribution with these parameters. During training, the SAC agent uses the unbounded probability distribution to compute the entropy of the policy for the given observation.

If the action space of the SAC agent is bounded, the actor generates bounded actions by applying *tanh* and *scaling* operations to the unbounded action.
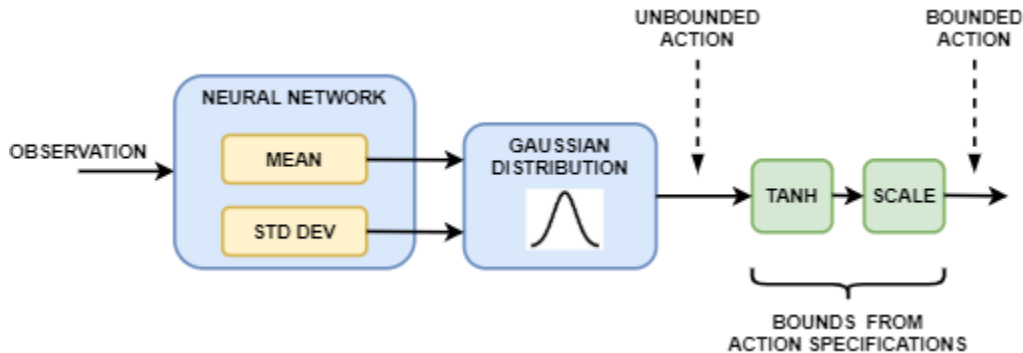
Agent Creation

You can create a SAC agent with default actor and critic representations based on the observation and action specifications from the environment. To do so, perform the following steps.

1   Create observation specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getObservationInfo`.

2   Create action specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getActionInfo`.

3   If needed, specify the number of neurons in each learnable layer or whether to use an LSTM layer. To do so, create an agent initialization option object using `rlAgentInitializationOptions`.

4   If needed, specify agent options using an `rlSACAgentOptions` object.

5   Create the agent using an `rlSACAgent` object.

Alternatively, you can create actor and critic representations and use these representations to create your agent. In this case, ensure that the input and output dimensions of the actor and critic representations match the corresponding action and observation specifications of the environment.

1   Create a stochastic actor using an `rlStochasticActorRepresentation` object. For SAC agents, the actor network must not contain a `tanhLayer` and `scalingLayer` in the mean output path.

2   Create one or two critics using `rlQValueRepresentation` objects.

3   Specify agent options using an `rlSACAgentOptions` object.

4   Create the agent using an `rlSACAgent` object.

SAC agents do not support actors and critics that use recurrent deep neural networks as function approximators.

For more information on creating actors and critics for function approximation, see "Create Policy and Value Function Representations" on page 4-2.

Training Algorithm

SAC agents use the following training algorithm, in which they periodically update their actor and critic models and entropy weight. To configure the training algorithm, specify options using an `rlSACAgentOptions` object. Here, $K = 2$ is the number of critics and k is the critic index.

- Initialize each critic $Q_k(S,A)$ with random parameter values θ_{Qk}, and initialize each target critic with the same random parameter values: $\theta_{Qk'} = \theta_{Qk}$.

- Initialize the actor $\mu(S)$ with random parameter values θ_μ.

- Perform a warm start by taking a sequence of actions following the initial random policy in $\mu(S)$. For each action, store the experience in the experience buffer. To specify the number of warm up actions, use the `NumWarmStartSteps` option.

- For each training time step:

    1 For the current observation S, select action A using the policy in $\mu(S)$.

    2 Execute action A. Observe the reward R and next observation S'.

    3 Store the experience (S,A,R,S') in the experience buffer.

    4 Sample a random mini-batch of M experiences (S_i,A_i,R_i,S'_i) from the experience buffer. To specify M, use the `MiniBatchSize` option.

    5 Every D_C steps, update the parameters of each critic by minimizing the loss L_k across all sampled experiences. To specify D_C, use the `CriticUpdateFrequency` option.

    $$L_k = \frac{1}{M} \sum_{i=1}^{M} \left(y_i - Q_k\left(S_i, A_i | \theta_{Qk}\right) \right)^2$$

    If S'_i is a terminal state, the value function target y_i is equal to the experience reward R_i. Otherwise, the value function target is the sum of R_i, the minimum discounted future reward from the critics, and the weighted entropy H.

    $$y_i = R_i + \gamma * \min_k \left(Q_{k'}\left(S'_i, A'_i | \theta_{Qk'}\right) \right) + \alpha H\left(\mu\left(S'_i | \theta_\mu\right)\right)$$

    Here:

    - A'_i is the bounded action derived the unbounded output of the actor $\mu(S'_i)$

    - γ is the discount factor, which you specify using the `DiscountFactor` option.

    - H is the policy entropy, which is computed for the unbounded output of the actor

    - α is the entropy tuning weight, which the SAC agent tunes during training.

    6 Every D_A steps, update the actor parameters by minimizing the following objective function. To set D_A, use the `PolicyUpdateFrequency` option.

    $$J_\mu = \frac{1}{M} \sum_{i=1}^{M} \left(-\min_k \left(Q_{k'}\left(S_i, A_i | \theta_{Qk'}\right) \right) - \alpha H\left(\mu\left(S_i | \theta_\mu\right)\right) \right)$$

    7 Every D_A steps, also update the entropy weight by minimizing the following loss function.

$$L_\alpha = \frac{1}{M} \sum_{i=1}^{M} \left(\alpha H' - \alpha H\big(\mu\big(S_i|\theta_\mu\big)\big) \right)^2$$

Here, H' is the target entropy, which you specify using the `EntropyWeightOptions.TargetEntropy` option.

8   Every D_T steps, update the target critics depending on the target update method. To specify D_T, use the `TargetUpdateFrequency` option. For more information, see "Target Update Methods" on page 3-24.

9   Repeat steps 4 through 8 N_G times, where N_G is the number of gradient steps, which you specify using the `NumGradientStepsPerUpdate` option.

Target Update Methods

SAC agents update their target critic parameters using one of the following target update methods.

- **Smoothing** — Update the target critic parameters at every time step using smoothing factor τ. To specify the smoothing factor, use the `TargetSmoothFactor` option.

  $$\theta_{Qk'} = \tau \theta_{Qk} + (1 - \tau)\theta_{Qk'}$$

- **Periodic** — Update the target critic parameters periodically without smoothing (`TargetSmoothFactor = 1`). To specify the update period, use the `TargetUpdateFrequency` parameter.

  $$\theta_{Qk'} = \theta_{Qk}$$

- **Periodic Smoothing** — Update the target parameters periodically with smoothing.

To configure the target update method, create a `rlSACAgentOptions` object, and set the `TargetUpdateFrequency` and `TargetSmoothFactor` parameters as shown in the following table.

Update Method	TargetUpdateFrequency	TargetSmoothFactor
Smoothing (default)	1	Less than 1
Periodic	Greater than 1	1
Periodic smoothing	Greater than 1	Less than 1

References

[1] Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, et al. 'Soft Actor-Critic Algorithms and Applications'. *ArXiv:1812.05905 [Cs, Stat]*, 29 January 2019. https://arxiv.org/abs/1812.05905.

See Also
`rlSACAgent` | `rlSACAgentOptions`

More About
- "Reinforcement Learning Agents" on page 3-2
- "Create Policy and Value Function Representations" on page 4-2

- "Train Reinforcement Learning Agents" on page 5-2

Custom Agents

To implement your own custom reinforcement learning algorithms, you can create a custom agent by creating a subclass of a custom agent class. You can then train and simulate this agent in MATLAB and Simulink environments. For more information about creating classes in MATLAB, see "User-Defined Classes".

Create Template Class

To define your custom agent, first create a class that is a subclass of the `rl.agent.CustomAgent` class. As an example, this topic describes the custom LQR agent trained in "Train Custom LQR Agent" on page 5-227. As a starting point for your own agent, you can open and modify this custom agent class. To add the example files to the MATLAB path and open the file, at the MATLAB command line, type the following code.

```
addpath(fullfile(matlabroot,'examples','rl','main'));
edit LQRCustomAgent.m
```

After saving the class to your own working folder, you can remove the example files from the path.

```
rmpath(fullfile(matlabroot,'examples','rl','main'));
```

This class has the following class definition, which indicates the agent class name and the associated abstract agent.

```
classdef LQRCustomAgent < rl.agent.CustomAgent
```

To define your agent, you must specify the following:

- Agent properties
- Constructor function
- Critic representation that estimates the discounted long-term reward (if required for learning)
- Actor representation that selects an action based on the current observation (if required for learning)
- Required agent methods
- Optional agent methods

Agent Properties

In the `properties` section of the class file, specify any parameters necessary for creating and training the agent. These parameters can include:

- Discount factor for discounting future rewards
- Configuration parameters for exploration models, such as noise models or epsilon-greedy exploration
- Experience buffers for using replay memory
- Mini-batch sizes for sampling from the experience buffer
- Number of steps to look ahead during training

For more information on potential agent properties, see the option objects for the built-in Reinforcement Learning Toolbox agents.

The `rl.Agent.CustomAgent` class already includes properties for the agent sample time (`SampleTime`) and the action and observation specifications (`ActionInfo` and `ObservationInfo`, respectively).

The custom LQR agent defines the following agent properties.

```
properties
    % Q
    Q

    % R
    R

    % Feedback gain
    K

    % Discount factor
    Gamma = 0.95

    % Critic
    Critic

    % Buffer for K
    KBuffer
    % Number of updates for K
    KUpdate = 1

    % Number for estimator update
    EstimateNum = 10
end

properties (Access = private)
    Counter = 1
    YBuffer
    HBuffer
end
```

Constructor Function

To create your custom agent, you must define a constructor function that:

- Defines the action and observation specifications. For more information about creating these specifications, see `rlNumericSpec` and `rlFiniteSetSpec`.
- Creates actor and critic representations as required by your training algorithm. For more information, see "Create Policy and Value Function Representations" on page 4-2.
- Configures agent properties.
- Calls the constructor of the base abstract class.

For example, the `LQRCustomAgent` constructor defines continuous action and observation spaces and creates a critic representation. The `createCritic` function is an optional helper function that defines the critic representation.

```
function obj = LQRCustomAgent(Q,R,InitialK)
    % Check the number of input arguments
    narginchk(3,3);
```

```
    % Call the abstract class constructor
    obj = obj@rl.agent.CustomAgent();

    % Set the Q and R matrices
    obj.Q = Q;
    obj.R = R;

    % Define the observation and action spaces
    obj.ObservationInfo = rlNumericSpec([size(Q,1),1]);
    obj.ActionInfo = rlNumericSpec([size(R,1),1]);

    % Create the critic representation
    obj.Critic = createCritic(obj);

    % Initialize the gain matrix
    obj.K = InitialK;

    % Initialize the experience buffers
    obj.YBuffer = zeros(obj.EstimateNum,1);
    num = size(Q,1) + size(R,1);
    obj.HBuffer = zeros(obj.EstimateNum,0.5*num*(num+1));
    obj.KBuffer = cell(1,1000);
    obj.KBuffer{1} = obj.K;
end
end
```

Actor and Critic Representations

If your learning algorithm uses a critic representation to estimate the long-term reward, an actor for selecting an action, or both, you must add these as agent properties. You must then create these representations when you create your agent; that is, in the constructor function. For more information on creating actors and critics, see "Create Policy and Value Function Representations" on page 4-2.

For example, the custom LQR agent uses a critic representation, stored in its `Critic` property, and no actor. The critic creation is implemented in the `createCritic` helper function, which is called from the `LQRCustomAgent` constructor.

```
function critic = createCritic(obj)
    nQ = size(obj.Q,1);
    nR = size(obj.R,1);
    n = nQ+nR;
    w0 = 0.1*ones(0.5*(n+1)*n,1);
    critic = rlQValueRepresentation({@(x,u) computeQuadraticBasis(x,u,n),w0},...
        getObservationInfo(obj),getActionInfo(obj));
    critic.Options.GradientThreshold = 1;
end
```

In this case, the critic is an `rlQValueRepresentation` object. To create such a representation, you must specify the handle to a custom basis function, in this case the `computeQuadraticBasis` function. For more information on this critic representation, see "Train Custom LQR Agent" on page 5-227.

Required Functions

To create a custom reinforcement learning agent you must define the following implementation functions. To call these functions in your own code, use the wrapper methods from the abstract base

class. For example, to call `getActionImpl`, use `getAction`. The wrapper methods have the same input and output arguments as the implementation methods.

Function	Description
`getActionImpl`	Selects an action by evaluating the agent policy for a given observation
`getActionWithExplorationImpl`	Selects an action using the exploration model of the agent
`learnImpl`	Learns from the current experiences and returns an action with exploration

Within your implementation functions, to evaluate your actor and critic representations, you can use the `getValue`, `getAction`, and `getMaxQValue` functions.

- To evaluate an `rlValueRepresentation` critic with only observation input signals, obtain the state value function `V` using the following syntax.

  ```
  V = getValue(Critic,Observation);
  ```

- To evaluate an `rlQValueRepresentation` critic with both observation and action input signals, obtain the state-action value function `Q` using the following syntax.

  ```
  Q = getValue(Critic,[Observation,Action]);
  ```

- To evaluate an `rlQValueRepresentation` critic with only observation input signals, obtain the state-action value function `Q` for all possible discrete actions using the following syntax.

  ```
  Q = getValue(Critic,Observation);
  ```

- A discrete action space `rlQValueRepresentation` critic, obtain the maximum Q state-action value function `Q` for all possible discrete actions using the following syntax.

  ```
  [MaxQ,MaxActionIndex] = getMaxQValue(Critic,Observation);
  ```

- To evaluate an actor representation (`rlStochasticActorRepresentation` or `rlDeterministicActorRepresentation`), obtain the action `A` using the following syntax.

  ```
  A = getAction(Actor,Observation);
  ```

For each of these cases, if your actor or critic network uses a recurrent neural network, the functions can also return the current values of the network state after obtaining the corresponding network output.

getActionImpl Function

The `getActionImpl` function is evaluates the policy of your agent and selects an action. This function must have the following signature, where `obj` is the agent object, `Observation` is the current observation, and `action` is the selected action.

```
function action = getActionImpl(obj,Observation)
```

For the custom LQR agent, you select an action by applying the *u=-Kx* control law.

```
function action = getActionImpl(obj,Observation)
    % Given the current state of the system, return an action
    action = -obj.K*Observation{:};
end
```

getActionWithExplorationImpl Function

The `getActionWithExplorationImpl` function selects an action using the exploration model of your agent. Using this function you can implement algorithms such as epsilon-greedy exploration. This function must have the following signature, where `obj` is the agent object, `Observation` is the current observation, and `action` is the selected action.

```
function action = getActionWithExplorationImpl(obj,Observation)
```

For the custom LQR agent, the `getActionWithExplorationImpl` function adds random white noise to an action selected using the current agent policy.

```
function action = getActionWithExplorationImpl(obj,Observation)
    % Given the current observation, select an action
    action = getAction(obj,Observation);

    % Add random noise to the action
    num = size(obj.R,1);
    action = action + 0.1*randn(num,1);
end
```

learnImpl Function

The `learnImpl` function defines how the agent learns from the current experience. This function implements the custom learning algorithm of your agent by updating the policy parameters and selecting an action with exploration. This function must have the following signature, where `obj` is the agent object, `exp` is the current agent experience, and `action` is the selected action.

```
function action = learnImpl(obj,exp)
```

The agent experience is the cell array `exp = {state,action,reward,nextstate,isdone}`.

- `state` is the current observation.
- `action` is the current action.
- `reward` is the current reward.
- `nextState` is the next observation.
- `isDone` is a logical flag indicating that the training episode is complete.

For the custom LQR agent, the critic parameters are updated every `N` steps.

```
function action = learnImpl(obj,exp)
    % Parse the experience input
    x = exp{1}{1};
    u = exp{2}{1};
    dx = exp{4}{1};
    y = (x'*obj.Q*x + u'*obj.R*u);
    num = size(obj.Q,1) + size(obj.R,1);

    % Wait N steps before updating the critic parameters
    N = obj.EstimateNum;
    h1 = computeQuadraticBasis(x,u,num);
    h2 = computeQuadraticBasis(dx,-obj.K*dx,num);
    H = h1 - obj.Gamma* h2;
    if obj.Counter<=N
        obj.YBuffer(obj.Counter) = y;
        obj.HBuffer(obj.Counter,:) = H;
```

```
            obj.Counter = obj.Counter + 1;
        else
            % Update the critic parameters based on the batch of
            % experiences
            H_buf = obj.HBuffer;
            y_buf = obj.YBuffer;
            theta = (H_buf'*H_buf)\H_buf'*y_buf;
            obj.Critic = setLearnableParameters(obj.Critic,{theta});

            % Derive a new gain matrix based on the new critic parameters
            obj.K = getNewK(obj);

            % Reset the experience buffers
            obj.Counter = 1;
            obj.YBuffer = zeros(N,1);
            obj.HBuffer = zeros(N,0.5*num*(num+1));
            obj.KUpdate = obj.KUpdate + 1;
            obj.KBuffer{obj.KUpdate} = obj.K;
        end

        % Find and return an action with exploration
        action = getActionWithExploration(obj,exp{4});
end
```

Optional Functions

Optionally, you can define how your agent is reset at the start of training by specifying a `resetImpl` function with the following function signature, where `obj` is the agent object. Using this function, you can set the agent into a known or random condition before training.

```
function resetImpl(ob)
```

Also, you can define any other helper functions in your custom agent class as required. For example, the custom LQR agent defines a `createCritic` function for creating the critic representation and a `getNewK` function that derives the feedback gain matrix from the trained critic parameters.

Create Custom Agent

After you define your custom agent class, create an instance of it in the MATLAB workspace. For example, to create the custom LQR agent, define the `Q`, `R`, and `InitialK` values and call the constructor function.

```
Q = [10,3,1;3,5,4;1,4,9];
R = 0.5*eye(3);
K0 = place(A,B,[0.4,0.8,0.5]);
agent = LQRCustomAgent(Q,R,K0);
```

After validating the environment object, you can use it to train a reinforcement learning agent. For an example that trains the custom LQR agent, see "Train Custom LQR Agent" on page 5-227.

See Also
```
train
```

More About

- "Reinforcement Learning Agents" on page 3-2
- "Create Policy and Value Function Representations" on page 4-2
- "Train Reinforcement Learning Agents" on page 5-2

Define Policies and Value Functions

Create Policy and Value Function Representations

A reinforcement learning policy is a mapping that selects the action that the agent takes based on observations from the environment. During training, the agent tunes the parameters of its policy representation to maximize the expected cumulative long-term reward.

Reinforcement learning agents estimate policies and value functions using function approximators called actor and critic representations respectively. The actor represents the policy that selects the best action to take, based on the current observation. The critic represents the value function that estimates the expected cumulative long-term reward for the current policy.

Before creating an agent, you must create the required actor and critic representations using deep neural networks, linear basis functions, or lookup tables. The type of function approximators you use depends on your application.

For more information on agents, see "Reinforcement Learning Agents" on page 3-2.

Actors and Critic Representations

The Reinforcement Learning Toolbox software supports the following types of representations:

- $V(S|\theta_V)$ — Critics that estimate the expected cumulative long-term reward based on a given observation S. You can create these critics using `rlValueRepresentation`.
- $Q(S,A|\theta_Q)$ — Critics that estimate the expected cumulative long-term reward for all possible discrete action based on a given observation S. You can create these critics using `rlQValueRepresentation`.
- $Q(S|\theta_Q)$ — Multi-output critics that estimate the expected cumulative long-term reward for all possible discrete actions A_i given observation S. You can create these critics using `rlQValueRepresentation`.
- $\mu(S|\theta_\mu)$ — Actors that select an action based on a given observation S. You can create these actors using either `rlDeterministicActorRepresentation` or `rlStochasticActorRepresentation`.

Each representation uses a function approximator with a corresponding set of parameters $(\theta_V, \theta_Q, \theta_\mu)$, which are computed during the learning process.
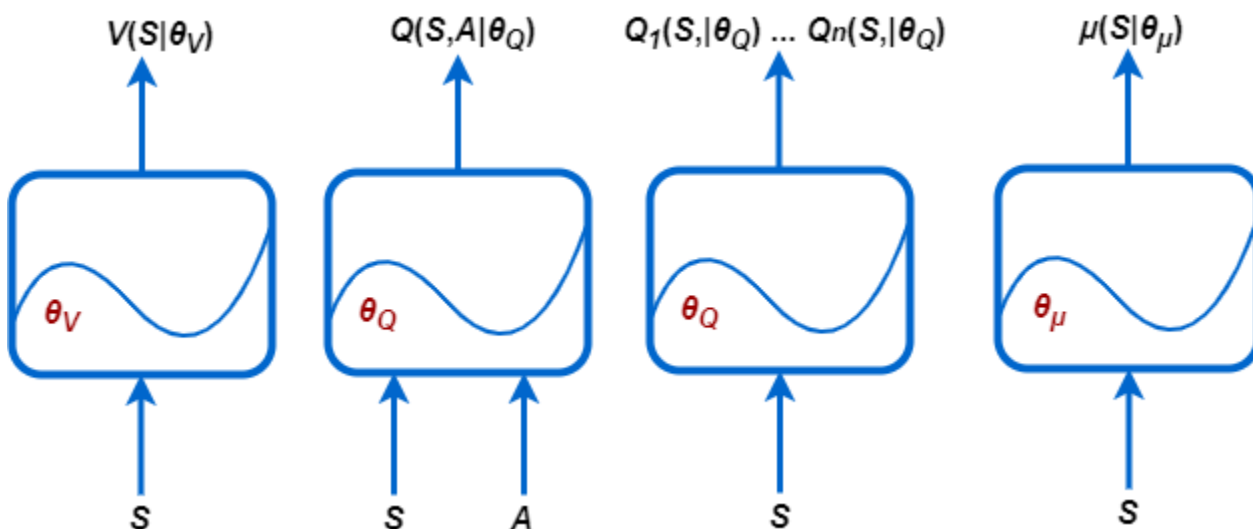
For systems with a limited number of discrete observations and discrete actions, you can store value functions in a lookup table. For systems that have many discrete observations and actions and for observation and action spaces that are continuous, storing the observations and actions is impractical. For such systems, you can represent your actors and critics using deep neural networks or custom (linear in the parameters) basis functions.

The following table summarizes the way in which you can use the four representation objects available with the Reinforcement Learning Toolbox software, depending on the action and observation spaces of your environment, and on the approximator and agent that you want to use.

Representations vs. Approximators and Agents

Representation	Supported Approximators	Observation Space	Action Space	Supported Agents
Value function critic, *V(S)* which you create using `rlValueRepresentation`	Table	Discrete	Not applicable	PG, AC, PPO
	Deep neural network or custom basis function	Discrete or continuous	Not applicable	PG, AC, PPO
Q-value function critic, *Q(S,A)* which you create using `rlQValueRepresentation`	Table	Discrete	Discrete	Q, DQN, SARSA
	Deep neural network or custom basis function	Discrete or continuous	Discrete	Q, DQN, SARSA
	Deep neural network or custom basis function	Discrete or continuous	Continuous	DDPG, TD3
Multi-output Q-value function critic, *Q(S,A)* which you create using `rlQValueRepresentation`	Deep neural network or custom basis function	Discrete or continuous	Discrete	Q, DQN, SARSA
Deterministic policy actor, *π(S)* which you create using `rlDeterministicActorRepresentation`	Deep neural network or custom basis function	Discrete or continuous	Continuous	DDPG, TD3
Stochastic policy actor, *π(S)* which you create using `rlStochasticActorRepresentation`	Deep neural network or custom basis function	Discrete or continuous	Discrete	PG, AC, PPO
	Deep neural network	Discrete or continuous	Continuous	PG, AC, PPO, SAC

For more information on agents, see "Reinforcement Learning Agents" on page 3-2.

Table Approximators

Representations based on lookup tables are appropriate for environments with a limited number of *discrete* observations and actions. You can create two types of lookup table representations:

- Value tables, which store rewards for corresponding observations
- Q-tables, which store rewards for corresponding observation-action pairs

To create a table representation, first create a value table or Q-table using the `rlTable` function. Then, create a representation for the table using either an `rlValueRepresentation` or `rlQValueRepresentation` object. To configure the learning rate and optimization used by the representation, use an `rlRepresentationOptions` object.

Deep Neural Network Approximators

You can create actor and critic function approximators using deep neural networks. Doing so uses the Deep Learning Toolbox software features.

Network Input and Output Dimensions

The dimensions of your actor and critic networks must match the corresponding action and observation specifications from the training environment object. To obtain the action and observation dimensions for environment `env`, use the `getActionInfo` and `getObservationInfo` functions, respectively. Then access the `Dimensions` property of the specification objects.

```
actInfo = getActionInfo(env);
actDimensions = actInfo.Dimensions;

obsInfo = getObservationInfo(env);
obsDimensions = obsInfo.Dimensions;
```

Networks for value function critics (such as the ones used in AC, PG, or PPO agents) must take only observations as inputs and must have a single scalar output. For these networks, the dimensions of the input layers must match the dimensions of the environment observation specifications. For more information, see `rlValueRepresentation`.

Networks for single-output Q-value function critics (such as the ones used in Q, DQN, SARSA, DDPG, TD3, and SAC agents) must take both observations and actions as inputs, and must have a single scalar output. For these networks, the dimensions of the input layers must match the dimensions of the environment specifications for both observations and actions. For more information, see `rlQValueRepresentation`.

Networks for multi-output Q-value function critics (such as those used in Q, DQN, and SARSA agents) take only observations as inputs and must have a single output layer with output size equal to the number of discrete actions. For these networks the dimensions of the input layers must match the dimensions of the environment observations. specifications. For more information, see `rlQValueRepresentation`.

For actor networks, the dimensions of the input layers must match the dimensions of the environment observation specifications.

- Networks used in actors with a discrete action space (such as the ones in PG, AC, and PPO agents) must have a single output layer with an output size equal to the number of possible discrete actions.

- Networks used in deterministic actors with a continuous action space (such as the ones in DDPG and TD3 agents) must have a single output layer with an output size matching the dimension of the action space defined in the environment action specification.
- Networks used in stochastic actors with a continuous action space (such as the ones in PG, AC, PPO, and SAC agents) must have a single output layer with output size having twice the dimension of the action space defined in the environment action specification. These networks must have two separate paths, the first producing the mean values (which must be scaled to the output range) and the second producing the standard deviations (which must be non-negative).

For more information, see `rlDeterministicActorRepresentation` and `rlStochasticActorRepresentation`.

Build Deep Neural Network

Deep neural networks consist of a series of interconnected layers. The following table lists some common deep learning layers used in reinforcement learning applications. For a full list of available layers, see "List of Deep Learning Layers".

Layer	Description
`featureInputLayer`	Inputs feature data and applies normalization
`imageInputLayer`	Inputs vectors and 2-D images and applies normalization.
`sigmoidLayer`	Applies a sigmoid function to the input such that the output is bounded in the interval (0,1).
`tanhLayer`	Applies a hyperbolic tangent activation layer to the input.
`reluLayer`	Sets any input values that are less than zero to zero.
`fullyConnectedLayer`	Multiplies the input vector by a weight matrix, and add a bias vector.
`convolution2dLayer`	Applies sliding convolutional filters to the input.
`additionLayer`	Adds the outputs of multiple layers together.
`concatenationLayer`	Concatenates inputs along a specified dimension.
`sequenceInputLayer`	Provides inputs sequence data to a network.
`lstmLayer`	Applies a Long Short-Term Memory layer to the input. Supported for DQN and PPO agents.

The `bilstmLayer` and `batchNormalizationLayer` layers are not supported for reinforcement learning.

The Reinforcement Learning Toolbox software provides the following layers, which contain no tunable parameters (that is, parameters that change during training).

Layer	Description
`scalingLayer`	Applies a linearly scale and bias to an input array. This layer is useful for scaling and shifting the outputs of nonlinear layers, such as `tanhLayer` and sigmoid layer.

Layer	Description
quadraticLayer	Creates a vector of quadratic monomials constructed from the elements of the input array. This layer is useful when you need an output that is some quadratic function of its inputs, such as for an LQR controller.
softplusLayer	Implements the softplus activation $Y = \log(1 + e^x)$, which ensures that the output is always positive. This is a smoothed version of the rectified linear unit (ReLU).

You can also create your own custom layers. For more information, see "Define Custom Deep Learning Layers".

For reinforcement learning applications, you construct your deep neural network by connecting a series of layers for each input path (observations or actions) and for each output path (estimated rewards or actions). You then connect these paths together using the connectLayers function.

You can also create your deep neural network using the **Deep Network Designer** app. For an example, see "Create Agent Using Deep Network Designer and Train Using Image Observations" on page 5-73.

When you create a deep neural network, you must specify names for the first layer of each input path and the final layer of the output path.

The following code creates and connects the following input and output paths:

- An observation input path, observationPath, with the first layer named 'observation'.

- An action input path, actionPath, with the first layer named 'action'.

- An estimated value function output path, commonPath, which takes the outputs of observationPath and actionPath as inputs. The final layer of this path is named 'output'.

```
observationPath = [
    imageInputLayer([4 1 1],'Normalization','none','Name','observation')
    fullyConnectedLayer(24,'Name','CriticObsFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(24,'Name','CriticObsFC2')];
actionPath = [
    imageInputLayer([1 1 1],'Normalization','none','Name','action')
    fullyConnectedLayer(24,'Name','CriticActFC1')];
commonPath = [
    additionLayer(2,'Name','add')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(1,'Name','output')];
criticNetwork = layerGraph(observationPath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);
criticNetwork = connectLayers(criticNetwork,'CriticObsFC2','add/in1');
criticNetwork = connectLayers(criticNetwork,'CriticActFC1','add/in2');
```

For all observation and action input paths, you must specify an imageInputLayer as the first layer in the path.

You can view the structure of your deep neural network using the plot function.

```
plot(criticNetwork)
```
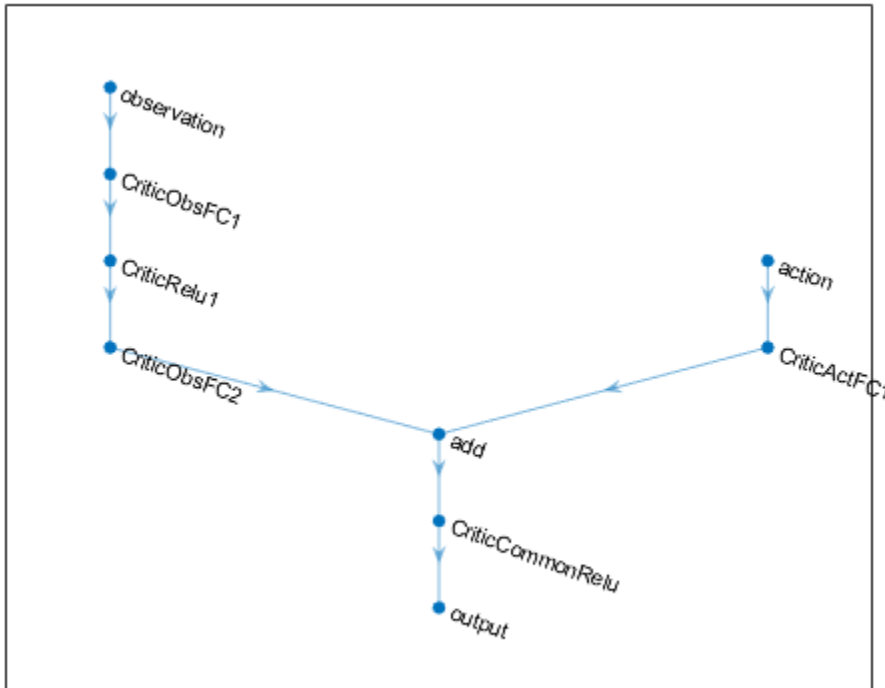
For PG and AC agents, the final output layers of your deep neural network actor representation are a `fullyConnectedLayer` and a `softmaxLayer`. When you specify the layers for your network, you must specify the `fullyConnectedLayer` and you can optionally specify the `softmaxLayer`. If you omit the `softmaxLayer`, the software automatically adds one for you.

Determining the number, type, and size of layers for your deep neural network representation can be difficult and is application dependent. However, the most critical component in deciding the characteristics of the function approximator is whether it is able to approximate the optimal policy or discounted value function for your application, that is, whether it has layers that can correctly learn the features of your observation, action, and reward signals.

Consider the following tips when constructing your network.

- For continuous action spaces, bound actions with a `tanhLayer` followed by a `ScalingLayer`, if necessary.
- Deep dense networks with `reluLayer` layers can be fairly good at approximating many different functions. Therefore, they are often a good first choice.
- Start with the smallest possible network that you think can approximate the optimal policy or value function.
- When you approximate strong nonlinearities or systems with algebraic constraints, adding more layers is often better than increasing the number of outputs per layer. In general, the ability of the approximator to represent more complex functions grows only polynomially in the size of the layers, but grows exponentially with the number of layers. In other words, more layers allow approximating more complex and nonlinear compositional functions, although this generally requires more data and longer training times. Networks with fewer layers can require exponentially more units to successfully approximate the same class of functions, and might fail to learn and generalize correctly.

- For on-policy agents (the ones that learn only from experience collected while following the current policy), such as AC and PG agents, parallel training works better if your networks are large (for example, a network with two hidden layers with 32 nodes each, which has a few hundred parameters). On-policy parallel updates assume each worker updates a different part of the network, such as when they explore different areas of the observation space. If the network is small, the worker updates can correlate with each other and make training unstable.

Create and Configure Representation

To create a critic representation for your deep neural network, use an `rlValueRepresentation` or `rlQValueRepresentation` object. To create an actor representation for your deep neural network, use an `rlDeterministicActorRepresentation` or `rlStochasticActorRepresentation` object. To configure the learning rate and optimization used by the representation, use an `rlRepresentationOptions` object.

For example, create a Q-value representation object for the critic network `criticNetwork`, specifying a learning rate of `0.0001`. When you create the representation, pass the environment action and observation specifications to the `rlQValueRepresentation` object, and specify the names of the network layers to which the observations and actions are connected (in this case `'observation'` and `'action'`).

```
opt = rlRepresentationOptions('LearnRate',0.0001);
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
        'Observation',{'observation'},'Action',{'action'},opt);
```

When you create your deep neural network and configure your representation object, consider using the following approach as a starting point.

1. Start with the smallest possible network and a high learning rate (`0.01`). Train this initial network to see if the agent converges quickly to a poor policy or acts in a random manner. If either of these issues occur, rescale the network by adding more layers or more outputs on each layer. Your goal is to find a network structure that is just big enough, does not learn too fast, and shows signs of learning (an improving trajectory of the reward graph) after an initial training period.

2. Once you settle on a good network architecture, a low initial learning rate can allow you to see if the agent is on the right track, and help you check that your network architecture is satisfactory for the problem. A low learning rate makes tuning parameters is much easier, especially for difficult problems.

Also, consider the following tips when configuring your deep neural network representation.

- Be patient with DDPG and DQN agents, since they might not learn anything for some time during the early episodes, and they typically show a dip in cumulative reward early in the training process. Eventually, they can show signs of learning after the first few thousand episodes.

- For DDPG and DQN agents, promoting exploration of the agent is critical.

- For agents with both actor and critic networks, set the initial learning rates of both representations to the same value. For some problems, setting the critic learning rate to a higher value than that of the actor can improve learning results.

Recurrent Neural Networks

When creating representations for use with a PPO or DQN agent, you can use recurrent neural networks. These networks are deep neural networks with a `sequenceInputLayer` input layer and at least one layer that has hidden state information, such as an `lstmLayer`. They can be especially

useful when the environment has states that cannot be included in the observation vector. For more information and examples, see `rlValueRepresentation`, `rlQValueRepresentation`, `rlDeterministicActorRepresentation`, and `rlStochasticActorRepresentation`.

Custom Basis Function Approximators

Custom (linear in the parameters) basis function approximators have the form `f = W'B`, where `W` is a weight array and `B` is the column vector output of a custom basis function that you must create. The learnable parameters of a linear basis function representation are the elements of `W`.

For value function critic representations, (such as the ones used in AC, PG or PPO agents), `f` is a scalar value, so `W` must be a column vector with the same length as `B`, and `B` must be a function of the observation. For more information, see `rlValueRepresentation`.

For single-output Q-value function critic representations, (such as the ones used in Q, DQN, SARSA, DDPG, TD3, and SAC agents), `f` is a scalar value, so `W` must be a column vector with the same length as `B`, and `B` must be a function of both the observation and action. For more information, see `rlQValueRepresentation`.

For multi-output Q-value function critic representations with discrete action spaces, (such as those used in Q, DQN, and SARSA agents), `f` is a vector with as many elements as the number of possible actions. Therefore `W` must be a matrix with as many columns as the number of possible actions and as many rows as the length of `B`. `B` must be only a function of the observation. For more information, see `rlQValueRepresentation`.

- For actors with a discrete action space (such as the ones in PG, AC, and PPO agents), `f` must be column vector with length equal to the number of possible discrete actions.
- For deterministic actors with a continuous action space (such as the ones in DDPG, and TD3 agents), the dimensions of `f` must match the dimensions of the agent action specification, which is either a scalar or a column vector.
- Stochastic actors with continuous action spaces cannot rely on custom basis functions (they can only use neural network approximators, due to the need to enforce positivity for the standard deviations).

For any actor representation, `W` must have as many columns as the number of elements in `f`, and as many rows as the number of elements in `B`. `B` must be only a function of the observation. For more information, see `rlDeterministicActorRepresentation`, and `rlStochasticActorRepresentation`.

For an example that trains a custom agent that uses a linear basis function representation, see "Train Custom LQR Agent" on page 5-227.

Create an Agent or Specify Agent Representations

Once you create your actor and critic representations, you can create a reinforcement learning agent that uses these representations. For example, create a PG agent using a given actor and critic network.

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

For more information on the different types of reinforcement learning agents, see "Reinforcement Learning Agents" on page 3-2.

You can obtain the actor and critic representations from an existing agent using `getActor` and `getCritic`, respectively.

You can also set the actor and critic of an existing agent using `setActor` and `setCritic`, respectively. When you specify a representation for an existing agent using these functions, the input and output layers of the specified representation must match the observation and action specifications of the original agent.

See Also

More About

- "Reinforcement Learning Agents" on page 3-2
- "Import Policy and Value Function Representations" on page 4-11

Import Policy and Value Function Representations

To create function approximators for reinforcement learning, you can import pretrained deep neural networks or deep neural network layer architectures using the Deep Learning Toolbox network import functionality. You can import:

- Open Neural Network Exchange (ONNX) models, which require the Deep Learning Toolbox Converter for ONNX Model Format support package software. For more information, `importONNXLayers`.
- TensorFlow-Keras networks, which require Deep Learning Toolbox Importer for TensorFlow-Keras Models support package software. For more information, see `importKerasLayers`.
- Caffe convolutional networks, which require Deep Learning Toolbox Importer for Caffe Models support package software. For more information, see `importCaffeLayers`.

After you import a deep neural network, you can create a policy or value function representation object using a representation object, such as `rlValueRepresentation`.

When you import deep neural network architectures, consider the following.

- Imported architectures must have a single input layer and a single output layer. Therefore, importing entire critic networks with observation and action input layers is not supported.
- The dimensions of the imported network architecture input and output layers must match the dimensions of the corresponding action, observation, or reward dimensions for your environment.
- After importing the network architecture, you must set the names of the input and output layers to match the names of the corresponding action and observation specifications.

For more information on the deep neural network architectures supported for reinforcement learning, see "Create Policy and Value Function Representations" on page 4-2.

Import Actor and Critic for Image Observation Application

As an example, assume that you have an environment with a 50-by-50 grayscale image observation signal and a continuous action space. To train a policy gradient agent, you require the following function approximators, both of which must have a single 50-by-50 image input observation layer and a single scalar output value.

- **Actor** — Selects an action value based on the current observation
- **Critic** — Estimates the expected long-term reward based on the current observation

Also, assume that you have the following network architectures to import:

- A deep neural network architecture for the actor with a 50-by-50 image input layer and a scalar output layer, which is saved in the ONNX format (`criticNetwork.onnx`).
- A deep neural network architecture for the critic with a 50-by-50 image input layer and a scalar output layer, which is saved in the ONNX format (`actorNetwork.onnx`).

To import the critic and actor networks, use the `importONNXLayers` function without specifying an output layer.

```
criticNetwork = importONNXLayers('criticNetwork.onnx');
actorNetwork = importONNXLayers('actorNetwork.onnx');
```

These commands generate a warning, which states that the network is trainable until an output layer is added. When you use an imported network to create an actor or critic representation, Reinforcement Learning Toolbox software automatically adds an output layer for you.

After you import the networks, create the actor and critic function approximator representations. To do so, first obtain the observation and action specifications from the environment.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create the critic representation, specifying the name of the input layer of the critic network as the observation name. Since the critic network has a single observation input and a single action output, use a value-function representation.

```
critic = rlValueRepresentation(criticNetwork,obsInfo,...
    'Observation',{criticNetwork.Layers(1).Name});
```

Create the actor representation, specifying the name of the input layer of the actor network as the observation name and the output layer of the actor network as the observation name. Since the actor network has a single scalar output, use a deterministic actor representation.

```
actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{actorNetwork.Layers(1).Name},...
    'Action',{actorNetwork.Layers(end).Name});
```

You can then:

- Create an agent using these representations. For more information, see "Reinforcement Learning Agents" on page 3-2.
- Set the actor and critic representation in an existing agent using `setActor` and `setCritic`, respectively.

See Also

More About
- "Create Policy and Value Function Representations" on page 4-2
- "Reinforcement Learning Agents" on page 3-2

5

Train and Validate Agents

Train Reinforcement Learning Agents

Once you have created an environment and reinforcement learning agent, you can train the agent in the environment using the `train` function. To configure your training, use the `rlTrainingOptions` function. For example, create a training option set `opt`, and train agent `agent` in environment `env`.

```
opt = rlTrainingOptions(...
    'MaxEpisodes',1000,...
    'MaxStepsPerEpisode',1000,...
    'StopTrainingCriteria',"AverageReward",...
    'StopTrainingValue',480);
trainStats = train(agent,env,opt);
```

For more information on creating agents, see "Reinforcement Learning Agents" on page 3-2. For more information on creating environments, see "Create MATLAB Environments for Reinforcement Learning" on page 2-2 and "Create Simulink Environments for Reinforcement Learning" on page 2-5.

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

```
save("initialAgent.mat","agent")
```

Training terminates automatically when the conditions you specify in the `StopTrainingCriteria` and `StopTrainingValue` options of your `rlTrainingOptions` object are satisfied. To manually terminate training in progress, type **Ctrl+C** or, 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`.

Training Algorithm

In general, training performs the following steps.

1 Initialize the agent.
2 For each episode:

    a Reset the environment.

    b Get the initial observation s_0 from the environment.

    c Compute the initial action $a_0 = \mu(s_0)$, where $\mu(s)$ is the current policy.

    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, perform the following steps.

        i Apply action a to the environment and 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 Terminate the episode if the termination conditions defined in the environment are met.

3 If the training termination condition is met, terminate training. Otherwise, begin the next episode.

The specifics of how the software performs these steps depend on the 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. For more information on agents and their training algorithms, see "Reinforcement Learning Agents" on page 3-2.

Episode Manager

By default, calling the `train` function opens the Reinforcement Learning Episode Manager, which lets you visualize the training progress. The Episode Manager plot shows the reward for each episode (**EpisodeReward**) and a running average reward value (**AverageReward**). Also, for agents that have critics, the plot shows the critic's estimate of the discounted long-term reward at the start of each episode (**EpisodeQ0**). The Episode Manager also displays various episode and training statistics. You can also use the `train` function to return episode and training information.
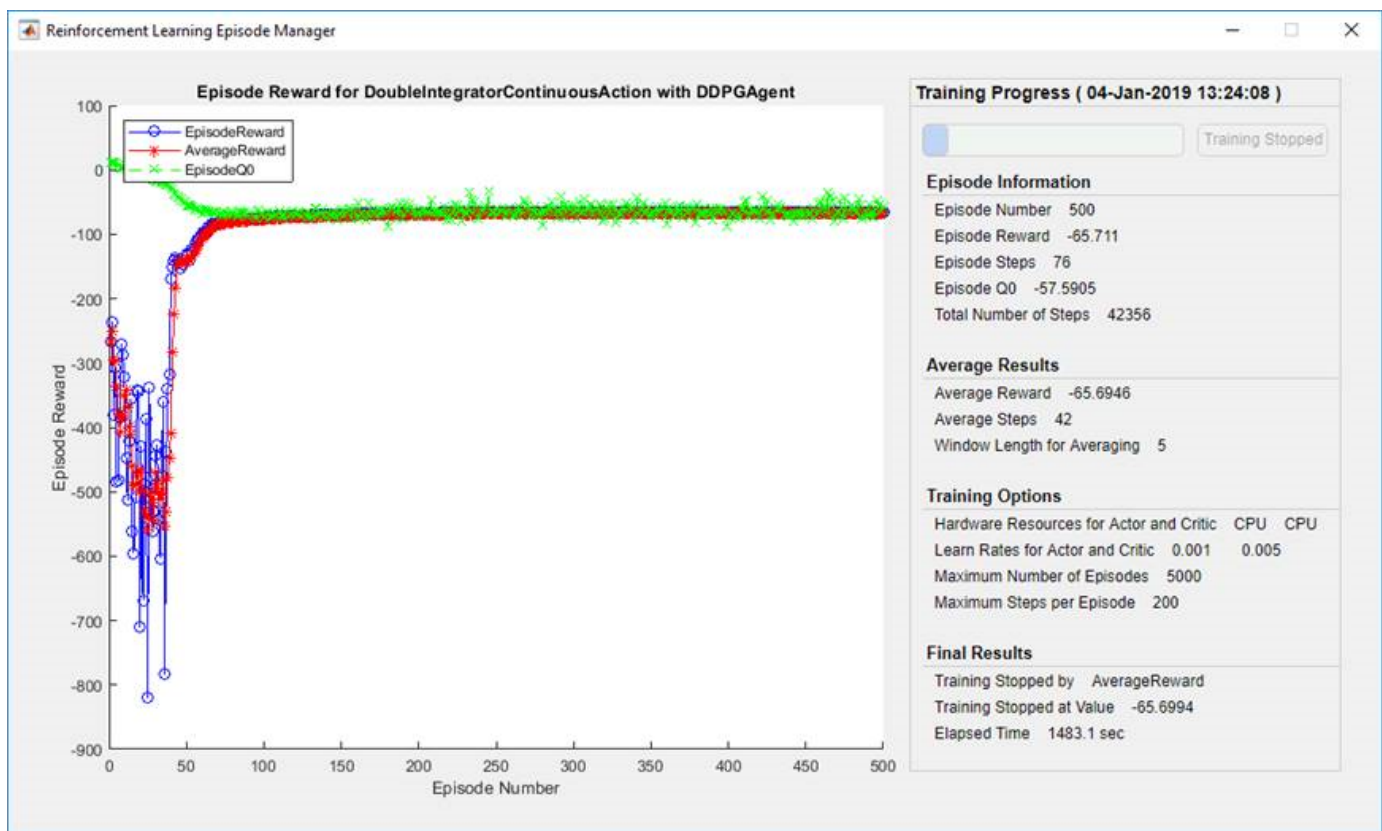
For agents with a critic, **Episode Q0** is the estimate of the discounted long-term reward at the start of each episode, given the initial observation of the environment. As training progresses, if the critic is well designed. **Episode Q0** approaches the true discounted long-term reward, as shown in the preceding figure.

To turn off the Reinforcement Learning Episode Manager, set the `Plots` option of `rlTrainingOptions` to `"none"`.

Save Candidate Agents

During training, you can save candidate agents that meet conditions you specify in the SaveAgentCriteria and SaveAgentValue options of your rlTrainingOptions object. 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. For example, save agents when the episode reward is greater than 100.

```
opt = rlTrainingOptions('SaveAgentCriteria',"EpisodeReward",'SaveAgentValue',100');
```

train stores saved agents in a MAT-file in the folder you specify using the SaveAgentDirectory option of rlTrainingOptions. Saved agents can be useful, for instance, to test candidate agents generated during a long-running training process. For details about saving criteria and saving location, see rlTrainingOptions.

After training is complete, you can save the final trained agent from the MATLAB workspace using the save function. For example, save the agent myAgent to the file finalAgent.mat in the current working directory.

```
save(opt.SaveAgentDirectory + "/finalAgent.mat",'agent')
```

By default, when DDPG and DQN agents are saved, the experience buffer data is not saved. 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 the SaveExperienceBufferWithAgent option to true. For some agents, such as those with large experience buffers and image-based observations, the memory required for saving the experience buffer is large. In these cases, you must ensure that enough memory is available for the saved agents.

Parallel Computing

You can accelerate agent training by running parallel training simulations. If you have Parallel Computing Toolbox software, you can run parallel simulations on multicore computers. If you have MATLAB Parallel Server software, you can run parallel simulations on computer clusters or cloud resources.

When you train agents using parallel computing, the host client sends copies of the agent and environment to each parallel worker. Each worker simulates the agent within the environment and sends their simulation data back to the host. The host agent learns from the data sent by the workers and sends the updated policy parameters back to the workers.
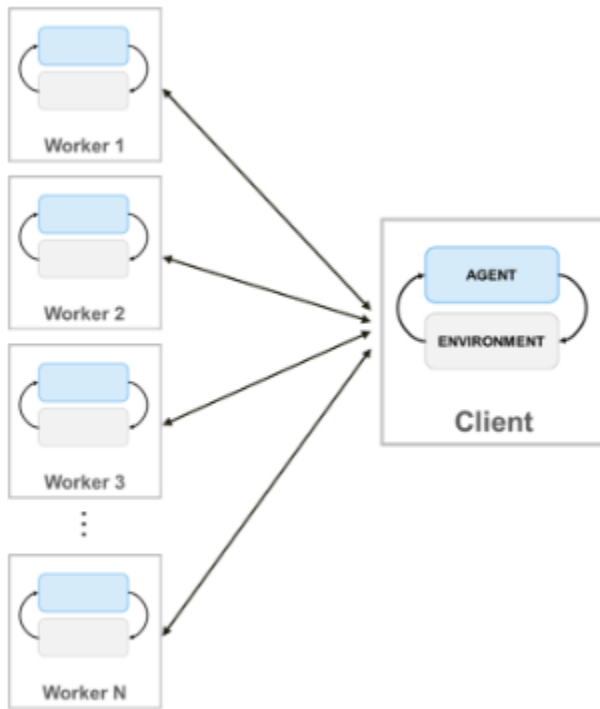
To create a parallel pool of N workers, use the following syntax.

```
pool = parpool(N);
```

If you do not create a parallel pool using `parpool`, the `train` function automatically creates one using your default parallel pool preferences. For more information on specifying these preferences, see "Specify Your Parallel Preferences" (Parallel Computing Toolbox).

For off-policy agents, such as DDPG and DQN agents, do not use all of your cores for parallel training. For example, if your CPU has six cores, train with four workers. Doing so provides more resources for the host client to compute gradients based on the experiences sent back from the workers. Limiting the number of workers is not necessary for on-policy agents, such as PG and AC agents, when the gradients are computed on the workers.

For more information on configuring your training to use parallel computing, see the `UseParallel` and `ParallelizationOptions` options in `rlTrainingOptions`.

To benefit from parallel computing, the computational cost for simulating the environment must be relatively expensive compared to the optimization of parameters when sending experiences back to the host. If the simulation of the environment is not expensive enough, the workers idle while waiting for the host to learn and send back updated parameters.

When sending experiences back from the workers, you can improve sample efficiency when the ratio (R) of the environment step complexity to the learning complexity is large. If the environment is fast to simulate (R is small), you are unlikely to get any benefit from experience-based parallelization. If the environment is expensive to simulate but it is also expensive to learn (for example, if the mini-batch size is large), then you are also unlikely to improve sample efficiency. However, in this case, for off-policy agents, you can reduce the mini-batch size to make R larger, which improves sample efficiency.

For an example that trains an agent using parallel computing in MATLAB, see "Train AC Agent to Balance Cart-Pole System Using Parallel Computing" on page 5-85. For an example that trains an agent using parallel computing in Simulink, see "Train DQN Agent for Lane Keeping Assist Using Parallel Computing" on page 5-160.

GPU Acceleration

When using deep neural network function approximators for your actor or critic representations, you can speed up training by performing representation operations on a GPU rather than a CPU. To do so, set the `UseDevice` option to `"gpu"`.

```
opt = rlRepresentationOptions('UseDevice',"gpu");
```

The size of any performance improvement depends on your specific application and network configuration.

Validate Trained Policy

To validate your trained agent, you can simulate the agent within the training environment using the `sim` function. To configure the simulation, use `rlSimulationOptions`.

When validating your agent, consider checking how your agent handles the following:

- Changes to simulation initial conditions — To change the model initial conditions, modify the reset function for the environment. For example reset functions, see "Create MATLAB Environment Using Custom Functions" on page 2-33, "Create Custom MATLAB Environment from Template" on page 2-40, and "Create Simulink Environments for Reinforcement Learning" on page 2-5.
- Mismatches between the training and simulation environment dynamics — To check such mismatches, create test environments in the same way that you created the training environment, modifying the environment behavior.

As with parallel training, if you have Parallel Computing Toolbox software, you can run multiple parallel simulations on multicore computers. If you have MATLAB Parallel Server software, you can run multiple parallel simulations on computer clusters or cloud resources. For more information on configuring your simulation to use parallel computing, see `UseParallel` and `ParallelizationOptions` in `rlSimulationOptions`.

Environment Visualization

If your training environment implements the `plot` method, you can visualize the environment behavior during training and simulation. If you call `plot(env)` before training or simulation, where `env` is your environment object, then the visualization updates during training to allow you to visualize the progress of each episode or simulation.

Environment visualization is not supported when training or simulating your agent using parallel computing.

For custom environments, you must implement your own `plot` method. For more information on creating a custom environments with a `plot` function, see "Create Custom MATLAB Environment from Template" on page 2-40.

See Also

`train`

More About

•   "Reinforcement Learning Agents" on page 3-2

Train DQN Agent to Balance Cart-Pole System

This example shows how to train a deep Q-learning network (DQN) agent to balance a cart-pole system modeled in MATLAB®.

For more information on DQN agents, see "Deep Q-Network Agents" on page 3-10. For an example that trains a DQN agent in Simulink®, see "Train DQN Agent to Swing Up and Balance Pendulum" on page 5-37.

Cart-Pole MATLAB Environment

The reinforcement learning environment for this example is a pole attached to an unactuated joint on a cart, which moves along a frictionless track. The training goal is to make the pole stand upright without falling over.
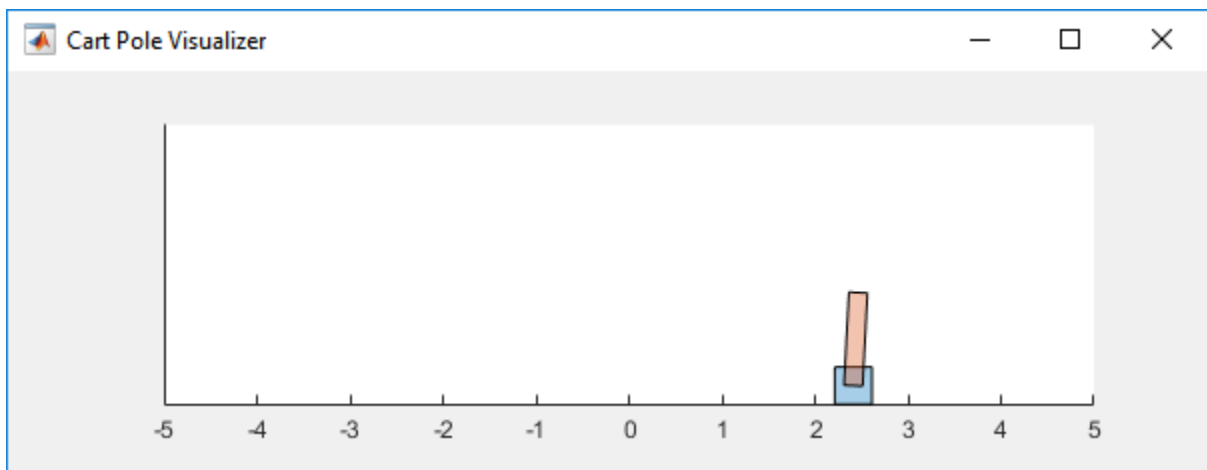
For this environment:

- The upward balanced pole position is `0` radians, and the downward hanging position is `pi` radians.
- The pole starts upright with an initial angle between –0.05 and 0.05 radians.
- The force action signal from the agent to the environment is from –10 to 10 N.
- The observations from the environment are the position and velocity of the cart, the pole angle, and the pole angle derivative.
- The episode terminates if the pole is more than 12 degrees from vertical or if the cart moves more than 2.4 m from the original position.
- A reward of +1 is provided for every time step that the pole remains upright. A penalty of –5 is applied when the pole falls.

For more information on this model, see "Load Predefined Control System Environments" on page 2-15.

Create Environment Interface

Create a predefined environment interface for the system.

```
env = rlPredefinedEnv("CartPole-Discrete")
```

```
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]
```

The interface has a discrete action space where the agent can apply one of two possible force values to the cart, –10 or 10 N.

Get the observation and action specification information.

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

```
obsInfo =
  rlNumericSpec with properties:

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

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

```
actInfo =
  rlFiniteSetSpec with properties:

       Elements: [-10 10]
           Name: "CartPole Action"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DQN Agent

A DQN agent approximates the long-term reward, given observations and actions, using a value-function critic.

DQN agents can use multi-output Q-value critic approximators, which are generally more efficient. A multi-output approximator has observations as inputs and state-action values as outputs. Each output element represents the expected cumulative long-term reward for taking the corresponding discrete action from the state indicated by the observation inputs.

To create the critic, first create a deep neural network with one input (the 4-dimensional observed state) and one output vector with two elements (one for the 10 N action, another for the –10 N action). For more information on creating value-function representations based on a neural network, see "Create Policy and Value Function Representations" on page 4-2.

```
dnn = [
    featureInputLayer(obsInfo.Dimension(1),'Normalization','none','Name','state')
    fullyConnectedLayer(24,'Name','CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(24, 'Name','CriticStateFC2')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(length(actInfo.Elements),'Name','output')];
```

View the network configuration.

```
figure
plot(layerGraph(dnn))
```


Specify some training options for the critic representation using `rlRepresentationOptions`.

```
criticOpts = rlRepresentationOptions('LearnRate',0.001,'GradientThreshold',1);
```

Create the critic representation using the specified neural network and options. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(dnn,obsInfo,actInfo,'Observation',{'state'},criticOpts);
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions`.

```
agentOpts = rlDQNAgentOptions(...
    'UseDoubleDQN',false, ...
    'TargetSmoothFactor',1, ...
    'TargetUpdateFrequency',4, ...
    'ExperienceBufferLength',100000, ...
    'DiscountFactor',0.99, ...
    'MiniBatchSize',256);
```

Then, create the DQN agent using the specified critic representation and agent options. For more information, see rlDQNAgent.

```
agent = rlDQNAgent(critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run one training session containing at most 1000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the Plots option) and disable the command line display (set the Verbose option to false).
- Stop training when the agent receives an moving average cumulative reward greater than 480. At this point, the agent can balance the cart-pole system in the upright position.

For more information, see rlTrainingOptions.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',1000, ...
    'MaxStepsPerEpisode',500, ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',480);
```

You can visualize the cart-pole system can be visualized by using the plot function during training or simulation.
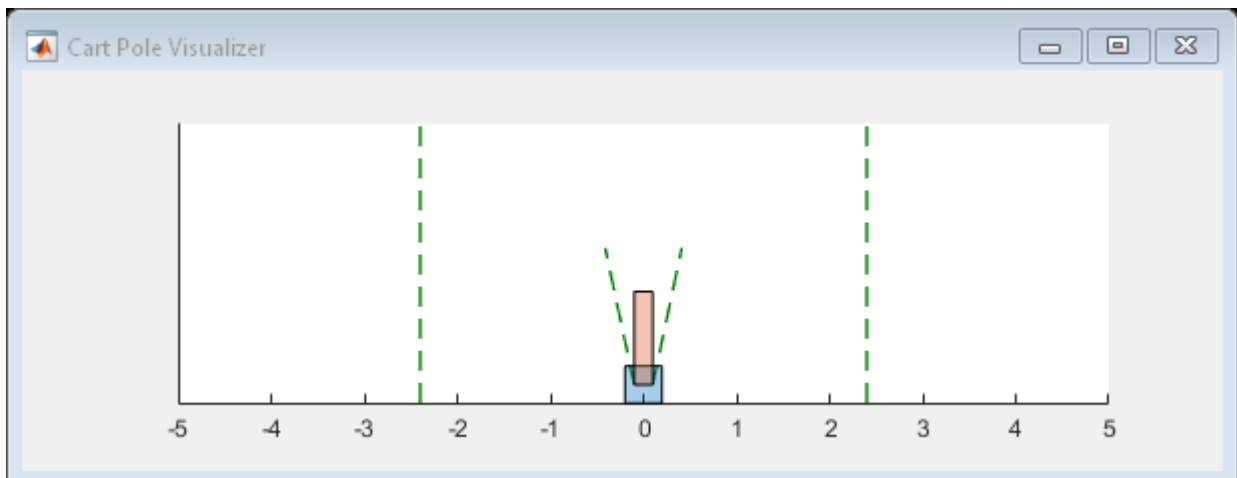
```
plot(env)
```

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('MATLABCartpoleDQNMulti.mat','agent')
end
```
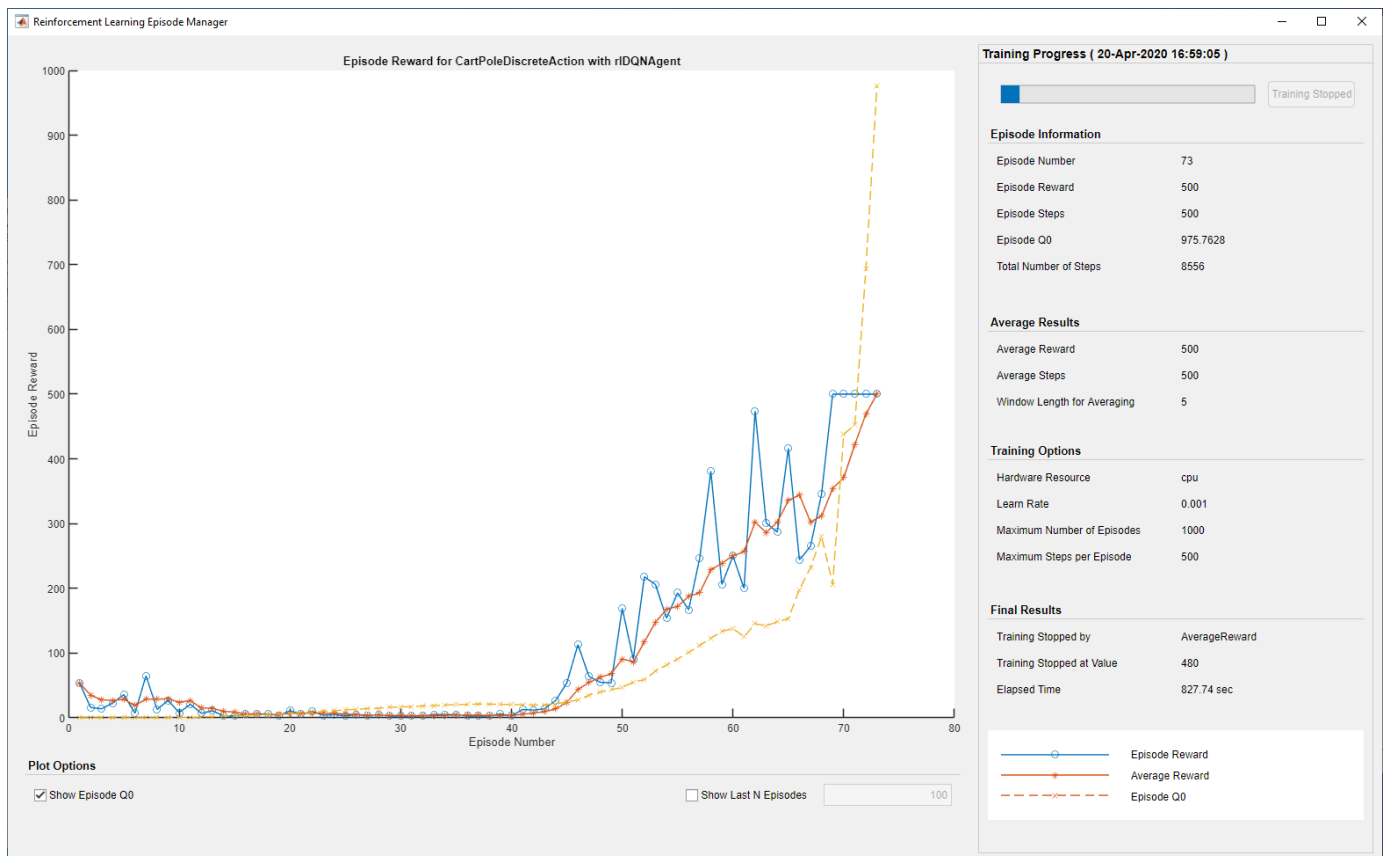

Simulate DQN Agent

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`. The agent can balance the cart-pole even when the simulation time increases to 500 steps.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```

```
totalReward = sum(experience.Reward)
```

```
totalReward = 500
```
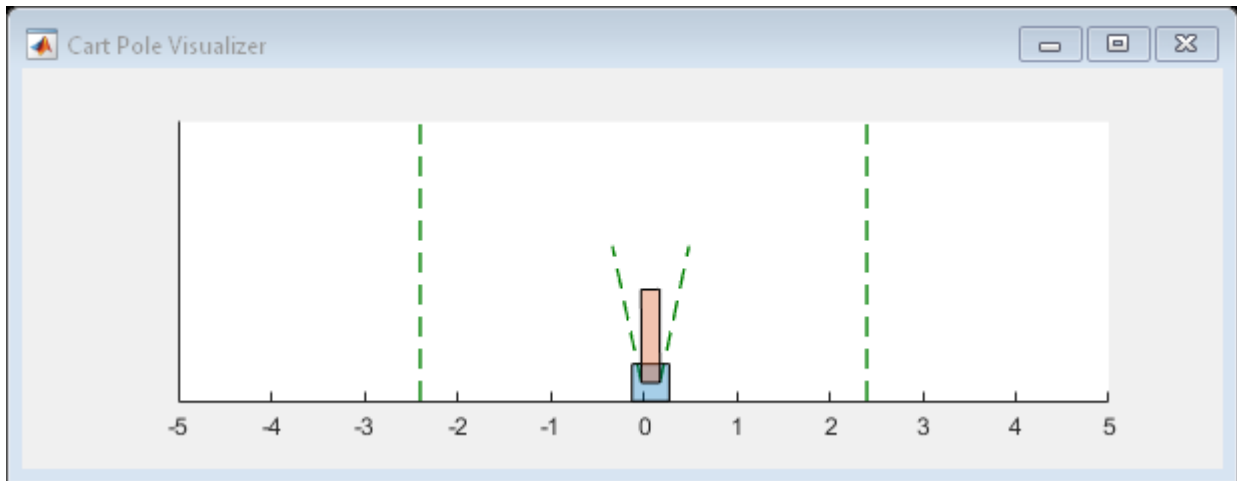
See Also

`train`

More About

- "Deep Q-Network Agents" on page 3-10
- "Train Reinforcement Learning Agents" on page 5-2
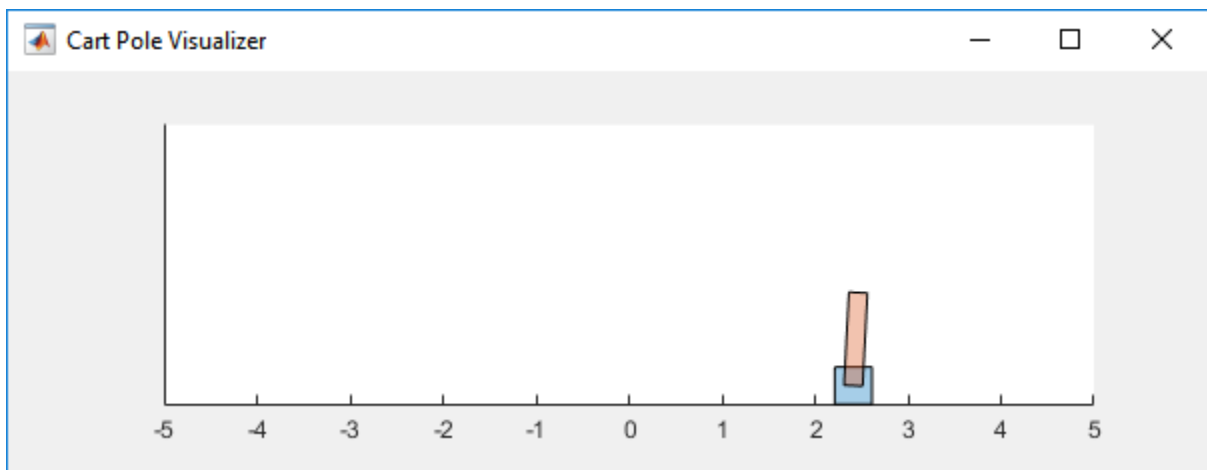- "Create Policy and Value Function Representations" on page 4-2

Train PG Agent to Balance Cart-Pole System

This example shows how to train a policy gradient (PG) agent to balance a cart-pole system modeled in MATLAB®. For more information on PG agents, see "Policy Gradient Agents" on page 3-13.

For an example that trains a PG agent with a baseline, see "Train PG Agent with Baseline to Control Double Integrator System" on page 5-25.

Cart-Pole MATLAB Environment

The reinforcement learning environment for this example is a pole attached to an unactuated joint on a cart, which moves along a frictionless track. The training goal is to make the pendulum stand upright without falling over.

For this environment:

- The upward balanced pendulum position is 0 radians, and the downward hanging position is pi radians.
- The pendulum starts upright with an initial angle between –0.05 and 0.05 radians.
- The force action signal from the agent to the environment is from –10 to 10 N.
- The observations from the environment are the position and velocity of the cart, the pendulum angle, and the pendulum angle derivative.
- The episode terminates if the pole is more than 12 degrees from vertical or if the cart moves more than 2.4 m from the original position.
- A reward of +1 is provided for every time step that the pole remains upright. A penalty of –5 is applied when the pendulum falls.

For more information on this model, see "Load Predefined Control System Environments" on page 2-15.

Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv("CartPole-Discrete")
```

```
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]
```

The interface has a discrete action space where the agent can apply one of two possible force values to the cart, –10 or 10 N.

Obtain the observation and action information from the environment interface.

```
obsInfo = getObservationInfo(env);
numObservations = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create PG Agent

A PG agent decides which action to take given observations using an actor representation. To create the actor, first create a deep neural network with one input (the observation) and one output (the action). The actor network has two outputs, which corresponds to the number of possible actions. For more information on creating a deep neural network policy representation, see "Create Policy and Value Function Representations" on page 4-2.

```
actorNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','state')
    fullyConnectedLayer(2,'Name','fc')
    softmaxLayer('Name','actionProb')
    ];
```

Specify options for the actor representation using `rlRepresentationOptions`.

```
actorOpts = rlRepresentationOptions('LearnRate',1e-2,'GradientThreshold',1);
```

Create the actor representation using the specified deep neural network and options. You must also specify the action and observation information for the critic, which you obtained from the environment interface. For more information, see `rlStochasticActorRepresentation`.

```
actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,actInfo,'Observation',{'state'},acto
```

Create the agent using the specified actor representation and the default agent options. For more information, see `rlPGAgent`.

```
agent = rlPGAgent(actor);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

- Run each training episode for at most 1000 episodes, with each episode lasting at most 200 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than 195 over 100 consecutive episodes. At this point, the agent can balance the pendulum in the upright position.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes', 1000, ...
    'MaxStepsPerEpisode', 200, ...
    'Verbose', false, ...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',195,...
    'ScoreAveragingWindowLength',100);
```

You can visualize the cart-pole system by using the `plot` function during training or simulation.
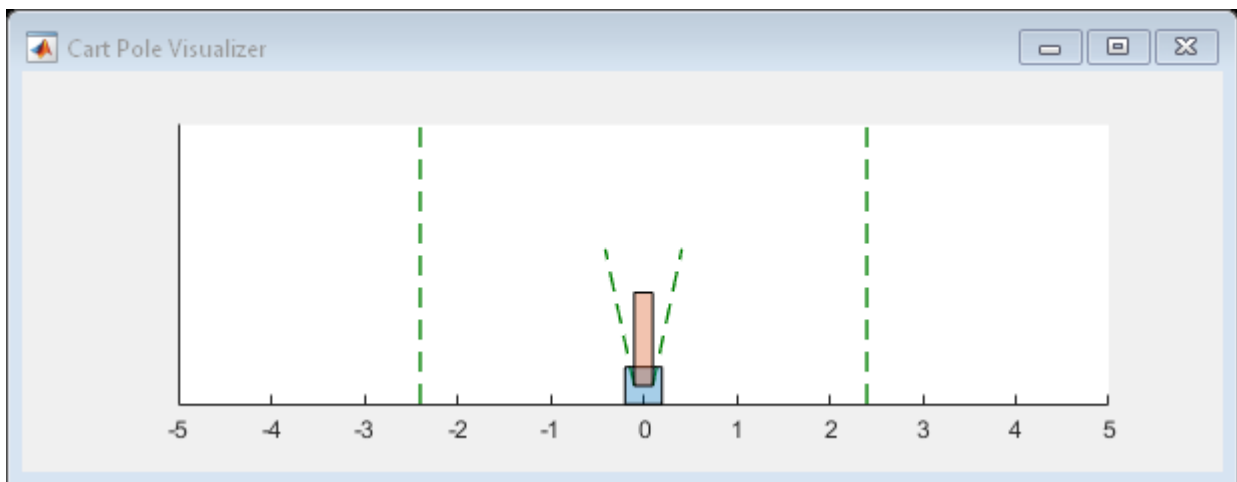
```
plot(env)
```


Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('MATLABCartpolePG.mat','agent');
end
```
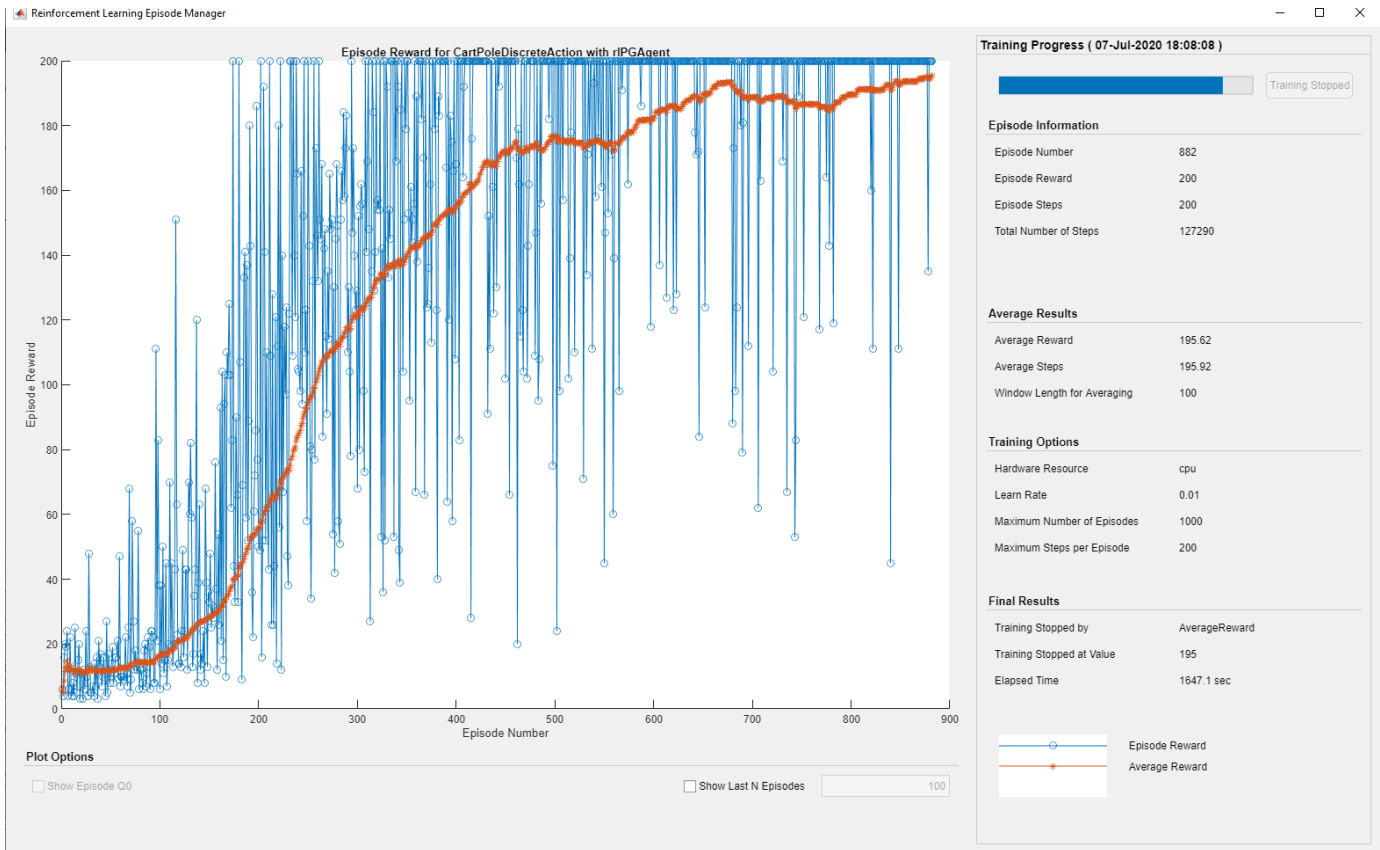
Simulate PG Agent

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`. The agent can balance the cart-pole system even when the simulation time increases to 500 steps.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```



```
totalReward = sum(experience.Reward)
```

```
totalReward = 500
```
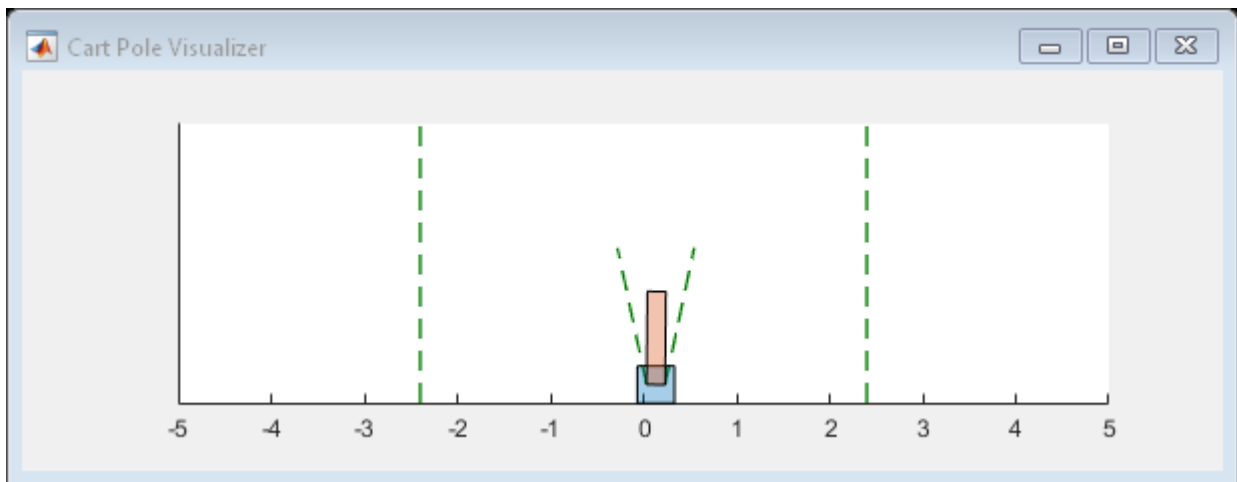
See Also

`train`

More About

- "Policy Gradient Agents" on page 3-13
- "Train Reinforcement Learning Agents" on page 5-2
- "Create Policy and Value Function Representations" on page 4-2

Train AC Agent to Balance Cart-Pole System

This example shows how to train an actor-critic (AC) agent to balance a cart-pole system modeled in MATLAB®.

For more information on AC agents, see "Actor-Critic Agents" on page 3-25. For an example showing how to train an AC agent using parallel computing, see "Train AC Agent to Balance Cart-Pole System Using Parallel Computing" on page 5-85.

Cart-Pole MATLAB Environment

The reinforcement learning environment for this example is a pole attached to an unactuated joint on a cart, which moves along a frictionless track. The training goal is to make the pendulum stand upright without falling over.
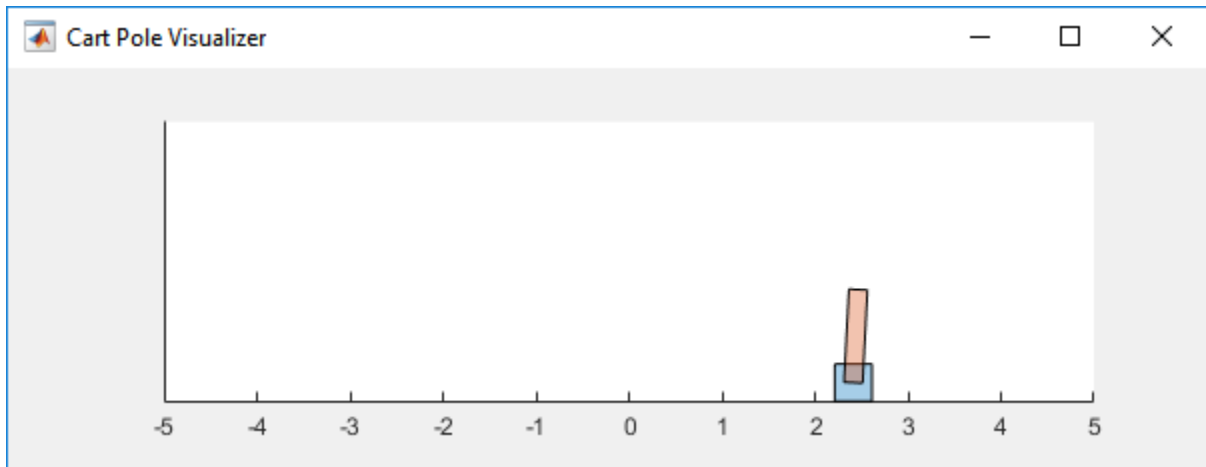
For this environment:

- The upward balanced pendulum position is `0` radians, and the downward hanging position is `pi` radians.
- The pendulum starts upright with an initial angle between –0.05 and 0.5 rad.
- The force action signal from the agent to the environment is from –10 to 10 N.
- The observations from the environment are the position and velocity of the cart, the pendulum angle, and the pendulum angle derivative.
- The episode terminates if the pole is more than 12 degrees from vertical or if the cart moves more than 2.4 m from the original position.
- A reward of +1 is provided for every time step that the pole remains upright. A penalty of –5 is applied when the pendulum falls.

For more information on this model, see "Load Predefined Control System Environments" on page 2-15.

Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv("CartPole-Discrete")
```

```
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]
```

```
env.PenaltyForFalling = -10;
```

The interface has a discrete action space where the agent can apply one of two possible force values to the cart, –10 or 10 N.

Obtain the observation and action information from the environment interface.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create AC Agent

An AC agent approximates the long-term reward, given observations and actions, using a critic value function representation. To create the critic, first create a deep neural network with one input (the observation) and one output (the state value). The input size of the critic network is 4 since the environment has four observations. For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

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

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOpts = rlRepresentationOptions('LearnRate',8e-3,'GradientThreshold',1);
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation information for the critic, which you obtain from the environment interface. For more information, see `rlValueRepresentation`.

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

An AC agent decides which action to take, given observations, using an actor representation. To create the actor, create a deep neural network with one input (the observation) and one output (the action). The output size of the actor network is 2 since the environment has two possible actions, –10 and 10.

Construct the actor in a similar manner to the critic. For more information, see `rlStochasticActorRepresentation`.

```
actorNetwork = [
    featureInputLayer(4,'Normalization','none','Name','state')
    fullyConnectedLayer(2,'Name','fc')
    softmaxLayer('Name','actionProb')];

actorOpts = rlRepresentationOptions('LearnRate',8e-3,'GradientThreshold',1);

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

To create the AC agent, first specify the AC agent options using `rlACAgentOptions`.

```
agentOpts = rlACAgentOptions(...
    'NumStepsToLookAhead',32, ...
    'DiscountFactor',0.99);
```

Then create the agent using the specified actor representation and the default agent options. For more information, see `rlACAgent`.

```
agent = rlACAgent(actor,critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

- Run each training episode for at most 1000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than 480 over 10 consecutive episodes. At this point, the agent can balance the pendulum in the upright position.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',1000,...
    'MaxStepsPerEpisode',500,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',480,...
    'ScoreAveragingWindowLength',10);
```

You can visualize the cart-pole system during training or simulation using the `plot` function.
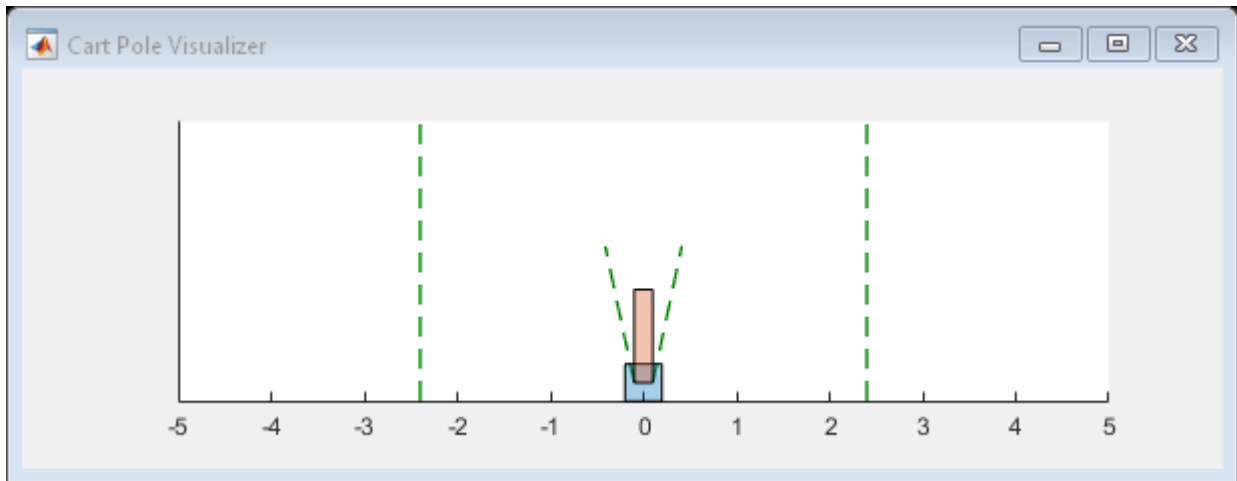
```
plot(env)
```

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('MATLABCartpoleAC.mat','agent');
end
```
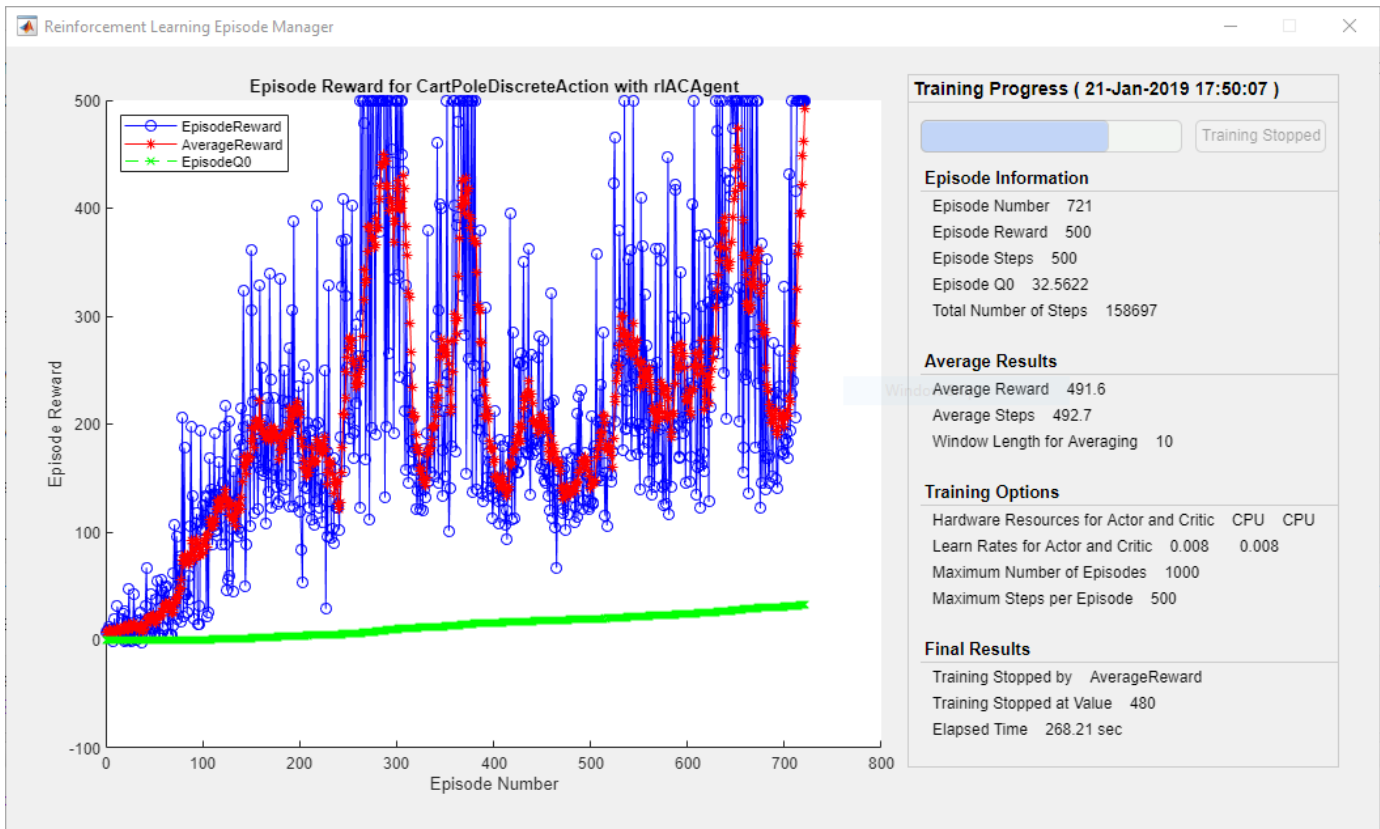
Simulate AC Agent

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```



```
totalReward = sum(experience.Reward)
```

```
totalReward = 500
```
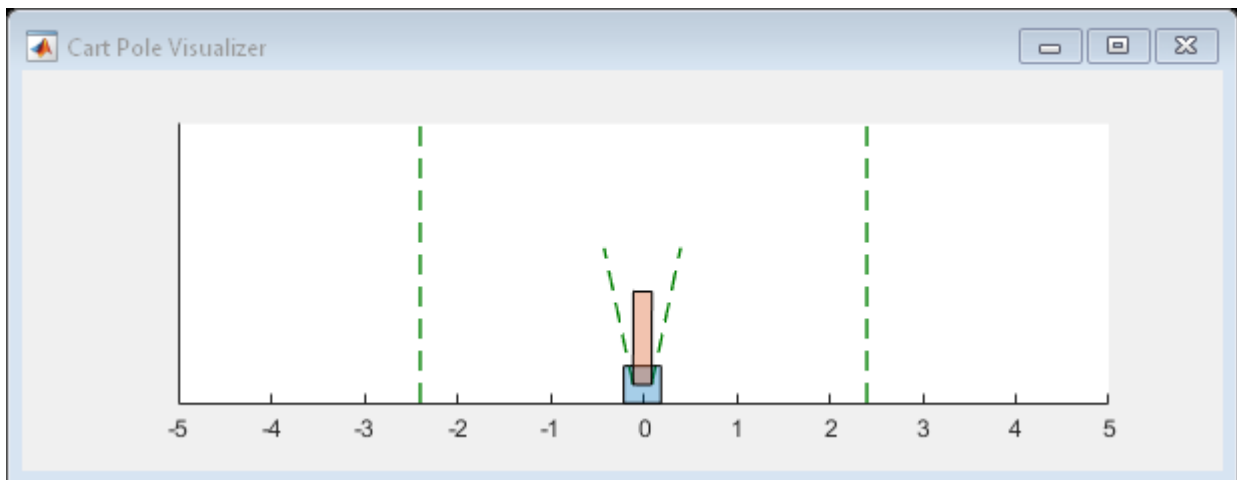
See Also

`train`

More About

- "Actor-Critic Agents" on page 3-25
- "Train Reinforcement Learning Agents" on page 5-2
- "Create Policy and Value Function Representations" on page 4-2

Train PG Agent with Baseline to Control Double Integrator System

This example shows how to train a policy gradient (PG) agent with baseline to control a second-order dynamic system modeled in MATLAB®.

For more information on the basic PG agent with no baseline, see the example "Train PG Agent to Balance Cart-Pole System" on page 5-14.

Double Integrator MATLAB Environment

The reinforcement learning environment for this example is a second-order double integrator system with a gain. The training goal is to control the position of a mass in the second-order system by applying a force input.
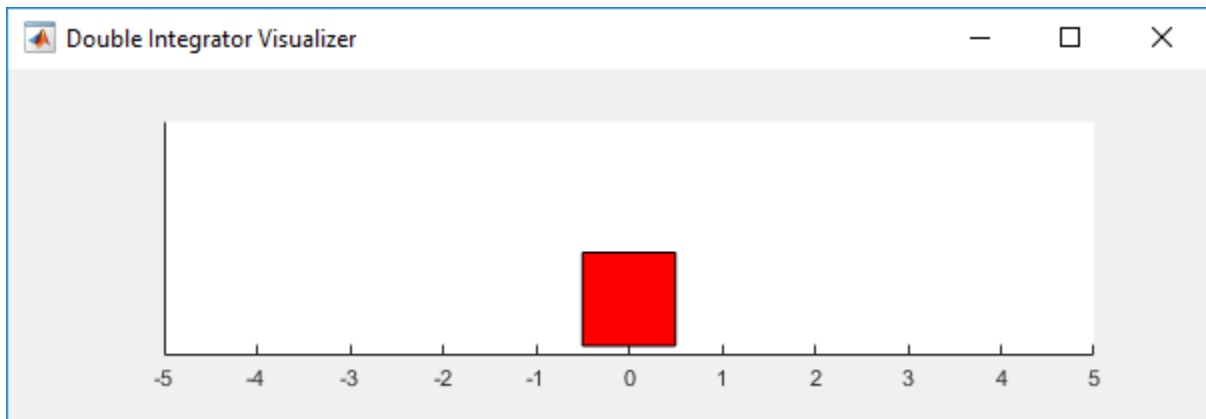
For this environment:

- The mass starts at an initial position between –2 and 2 units.
- The force action signal from the agent to the environment is from –2 to 2 N.
- The observations from the environment are the position and velocity of the mass.
- The episode terminates if the mass moves more than 5 m from the original position or if $|x| < 0.01$.
- The reward r_t, provided at every time step, is a discretization of $r(t)$:

$$r(t) = -(x(t)' \, Q \, x(t) + u(t)' \, R \, u(t))$$

Here:

- x is the state vector of the mass.
- u is the force applied to the mass.
- Q is the weights on the control performance; $Q = [10\ 0; 0\ 1]$.
- R is the weight on the control effort; $R = 0.01$.

For more information on this model, see "Load Predefined Control System Environments" on page 2-15.

Create Double Integrator MATLAB Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv("DoubleIntegrator-Discrete")

env =
  DoubleIntegratorDiscreteAction with properties:

            Gain: 1
              Ts: 0.1000
     MaxDistance: 5
   GoalThreshold: 0.0100
               Q: [2x2 double]
               R: 0.0100
        MaxForce: 2
           State: [2x1 double]
```

The interface has a discrete action space where the agent can apply one of three possible force values to the mass: -2, 0, or 2 N.

Obtain the observation and action information from the environment interface.

```
obsInfo = getObservationInfo(env);
numObservations = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
numActions = numel(actInfo.Elements);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create PG Agent Actor

A PG agent decides which action to take, given observations, using an actor representation. To create the actor, first create a deep neural network with one input (the observation) and one output (the action). For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
actorNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','state')
    fullyConnectedLayer(numActions,'Name','action','BiasLearnRateFactor',0)];
```

Specify options for the actor representation using `rlRepresentationOptions`.

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

Create the actor representation using the specified deep neural network and options. You must also specify the action and observation information for the critic, which you obtained from the environment interface. For more information, see `rlStochasticActorRepresentation`.

```
actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,actInfo,'Observation',{'state'},act
```

Create PG Agent Baseline

A baseline that varies with state can reduce the variance of the expected value of the update and thus improve the speed of learning for a PG agent. A possible choice for the baseline is an estimate of the state value function [1].

In this case, the baseline representation is a deep neural network with one input (the state) and one output (the state value).

Construct the baseline in a similar manner to the actor.

```
baselineNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','state')
    fullyConnectedLayer(8,'Name','BaselineFC')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(1,'Name','BaselineFC2','BiasLearnRateFactor',0)];

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

baseline = rlValueRepresentation(baselineNetwork,obsInfo,'Observation',{'state'},baselineOpts);
```

To create the PG agent with baseline, specify the PG agent options using `rlPGAgentOptions` and set the `UseBaseline` option set to `true`.

```
agentOpts = rlPGAgentOptions(...
    'UseBaseline',true, ...
    'DiscountFactor',0.99);
```

Then create the agent using the specified actor representation, baseline representation, and agent options. For more information, see `rlPGAgent`.

```
agent = rlPGAgent(actor,baseline,agentOpts);
```

Train Agent

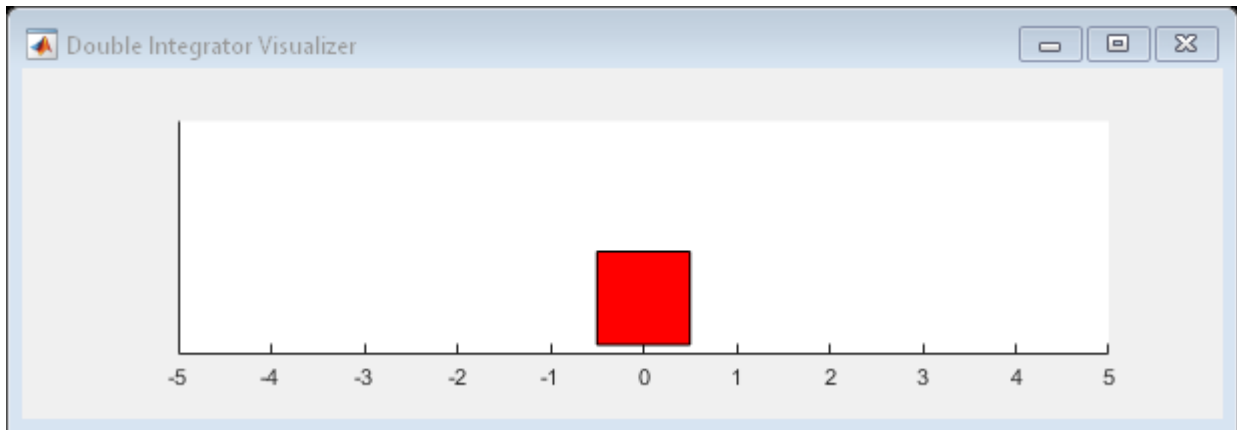To train the agent, first specify the training options. For this example, use the following options.

- Run at most 1000 episodes, with each episode lasting at most 200 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option).
- Stop training when the agent receives a moving average cumulative reward greater than –45. At this point, the agent can control the position of the mass using minimal control effort.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',1000, ...
    'MaxStepsPerEpisode',200, ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-43);
```

You can visualize the double integrator system using the `plot` function during training or simulation.

```
plot(env)
```

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained parameters for the example.
    load('DoubleIntegPGBaseline.mat');
end
```
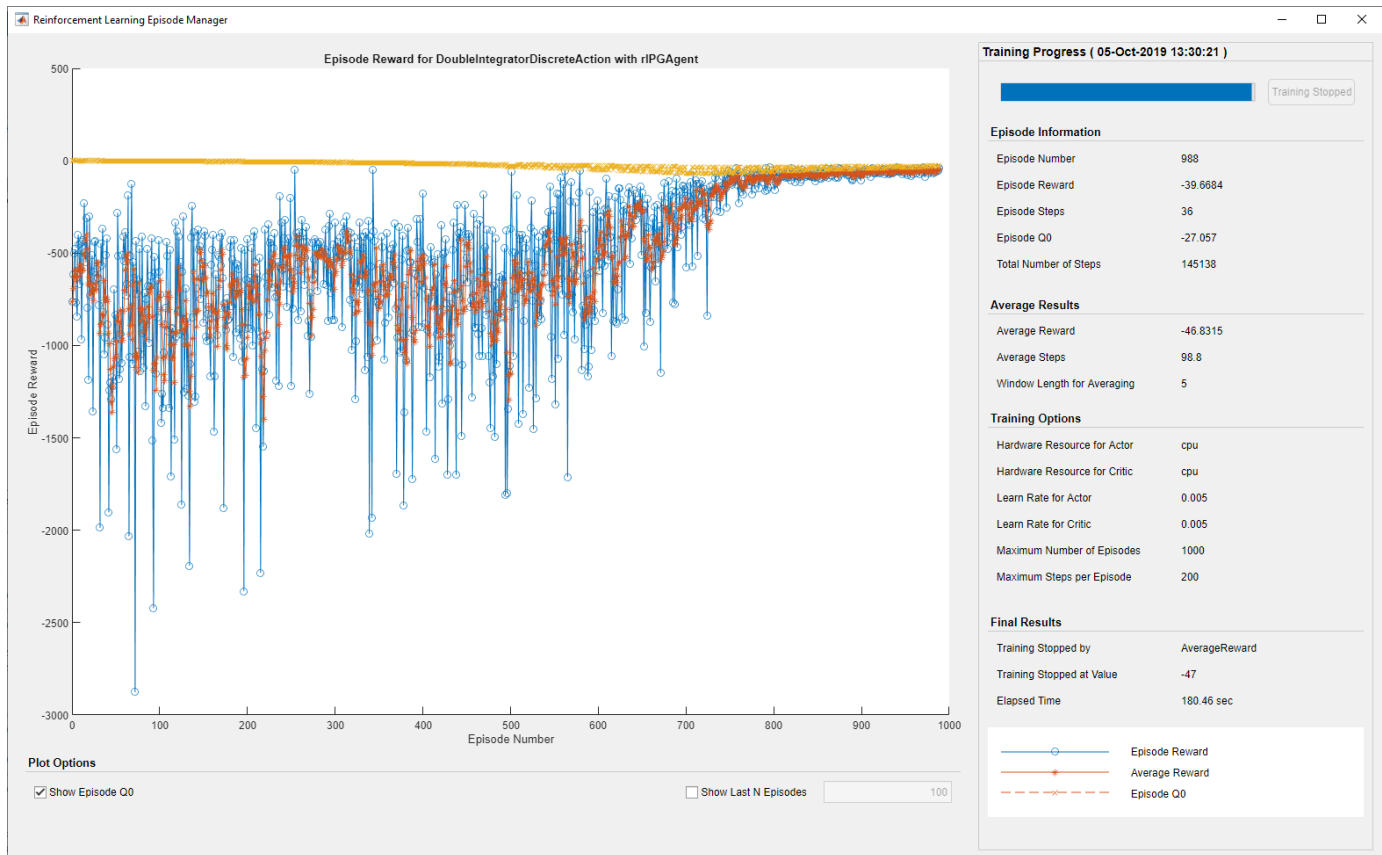
Simulate PG Agent

To validate the performance of the trained agent, simulate it within the double integrator environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```
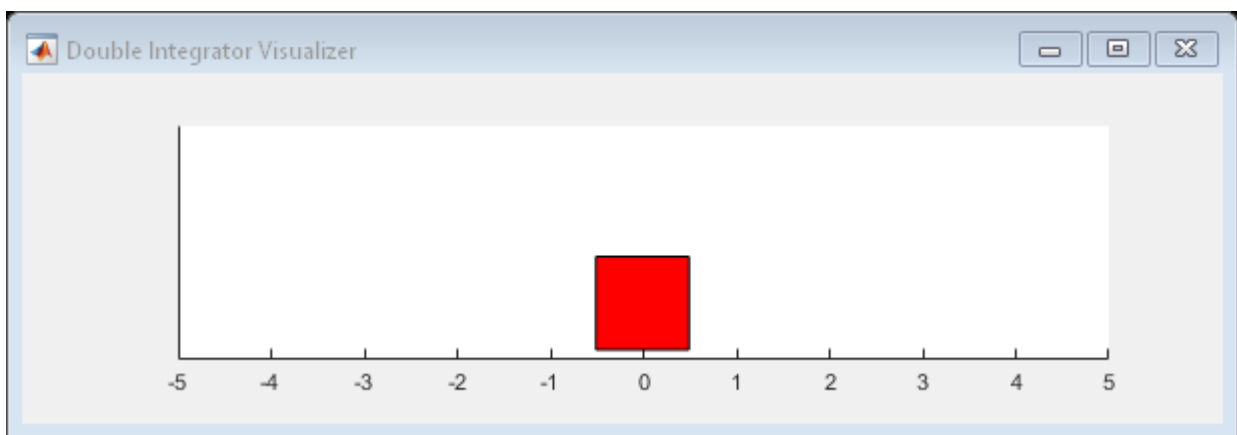


```
totalReward = sum(experience.Reward)
```

```
totalReward = -43.0392
```

References

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, MA: The MIT Press, 2018.

See Also
`rlPGAgent`

More About
- "Policy Gradient Agents" on page 3-13
- "Train PG Agent to Balance Cart-Pole System" on page 5-14
- "Train Reinforcement Learning Agents" on page 5-2
- "Create Policy and Value Function Representations" on page 4-2

Train DDPG Agent to Control Double Integrator System

This example shows how to train a deep deterministic policy gradient (DDPG) agent to control a second-order dynamic system modeled in MATLAB®.

For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17. For an example showing how to train a DDPG agent in Simulink®, see "Train DDPG Agent to Swing Up and Balance Pendulum" on page 5-44.

Double Integrator MATLAB Environment

The reinforcement learning environment for this example is a second-order double-integrator system with a gain. The training goal is to control the position of a mass in the second-order system by applying a force input.
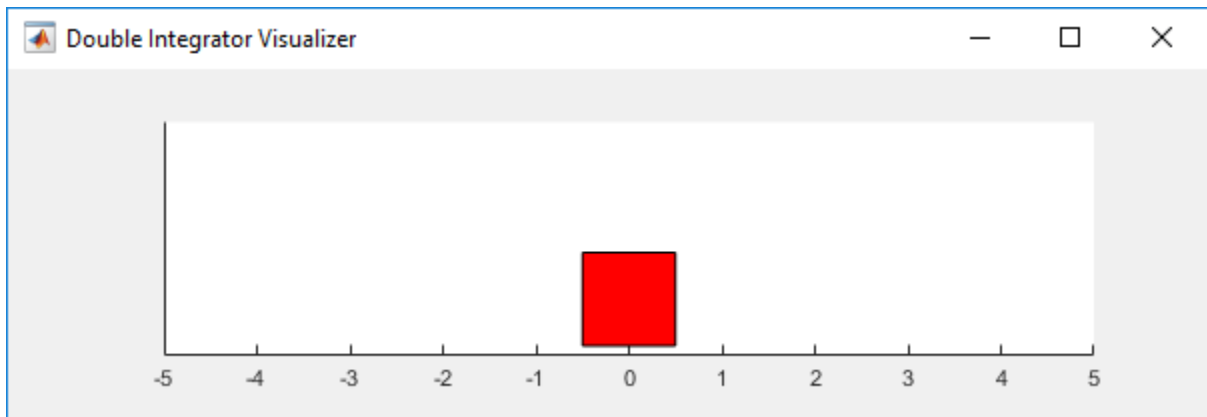
For this environment:

- The mass starts at an initial position between –4 and 4 units.
- The force action signal from the agent to the environment is from –2 to 2 N.
- The observations from the environment are the position and velocity of the mass.
- The episode terminates if the mass moves more than 5 m from the original position or if $|x| < 0.01$.
- The reward r_t, provided at every time step, is a discretization of $r(t)$:

$$r(t) = -(x(t)' \, Q \, x(t) + u(t)' \, R \, u(t))$$

Here:

- x is the state vector of the mass.
- u is the force applied to the mass.
- Q is the weights on the control performance; $Q = [10\ 0; 0\ 1]$.
- R is the weight on the control effort; $R = 0.01$.

For more information on this model, see "Load Predefined Control System Environments" on page 2-15.

Create Environment Interface

Create a predefined environment interface for the double integrator system.

```
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]
```

```
env.MaxForce = Inf;
```

The interface has a continuous action space where the agent can apply force values from `-Inf` to `Inf` to the mass.

Obtain the observation and action information from the environment interface.

```
obsInfo = getObservationInfo(env);
numObservations = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
numActions = numel(actInfo);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG Agent

A DDPG agent approximates the long-term reward, given observations and actions, using a critic value function representation. To create the critic, first create a deep neural network with two inputs (the state and action) and one output. For more information on creating a neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
statePath = imageInputLayer([numObservations 1 1],'Normalization','none','Name','state');
actionPath = imageInputLayer([numActions 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');
```

View the critic network configuration.
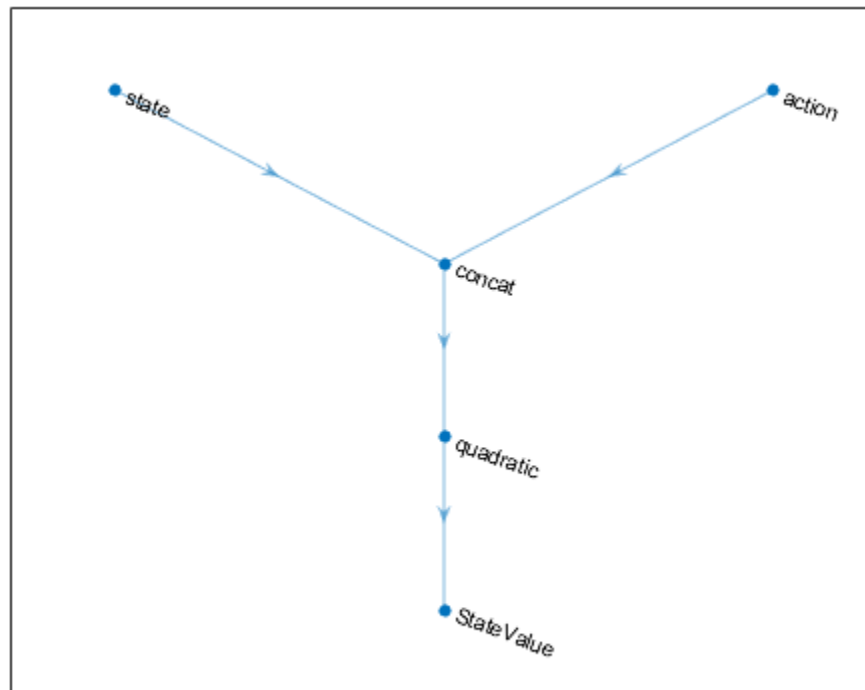
```
figure
plot(criticNetwork)
```

Specify options for the critic representation using `rlRepresentationOptions`.

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

Create the critic representation using the specified neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,'Observation',{'state'},'Action',{
```

A DDPG agent decides which action to take, given observations, using an actor representation. To create the actor, first create a deep neural network with one input (the observation) and one output (the action).

Construct the actor in a similar manner to the critic.

```
actorNetwork = [
    imageInputLayer([numObservations 1 1],'Normalization','none','Name','state')
    fullyConnectedLayer(numActions,'Name','action','BiasLearnRateFactor',0,'Bias',0)];
```

```
actorOpts = rlRepresentationOptions('LearnRate',1e-04,'GradientThreshold',1);
```

```
actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,'Observation',{'state'},
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOpts = rlDDPGAgentOptions(...
    'SampleTime',env.Ts,...
```

```
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',32);
agentOpts.NoiseOptions.Variance = 0.3;
agentOpts.NoiseOptions.VarianceDecayRate = 1e-6;
```

Create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent`.

```
agent = rlDDPGAgent(actor,critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

- Run at most 1000 episodes in the training session, with each episode lasting at most 200 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option).
- Stop training when the agent receives a moving average cumulative reward greater than –66. At this point, the agent can control the position of the mass using minimal control effort.
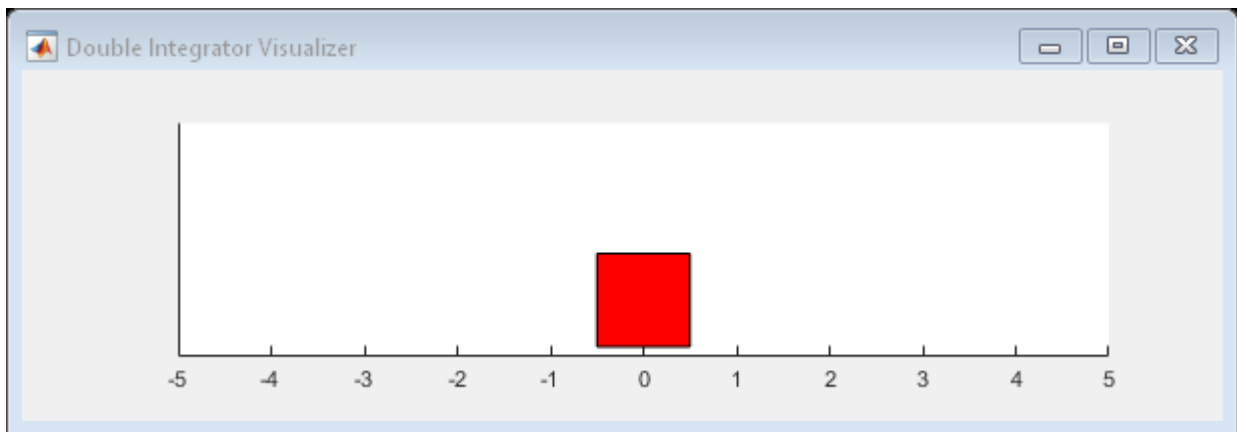
For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes', 5000, ...
    'MaxStepsPerEpisode', 200, ...
    'Verbose', false, ...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-66);
```

You can visualize the double integrator environment by using the `plot` function during training or simulation.

```
plot(env)
```


Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```matlab
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('DoubleIntegDDPG.mat','agent');
end
```
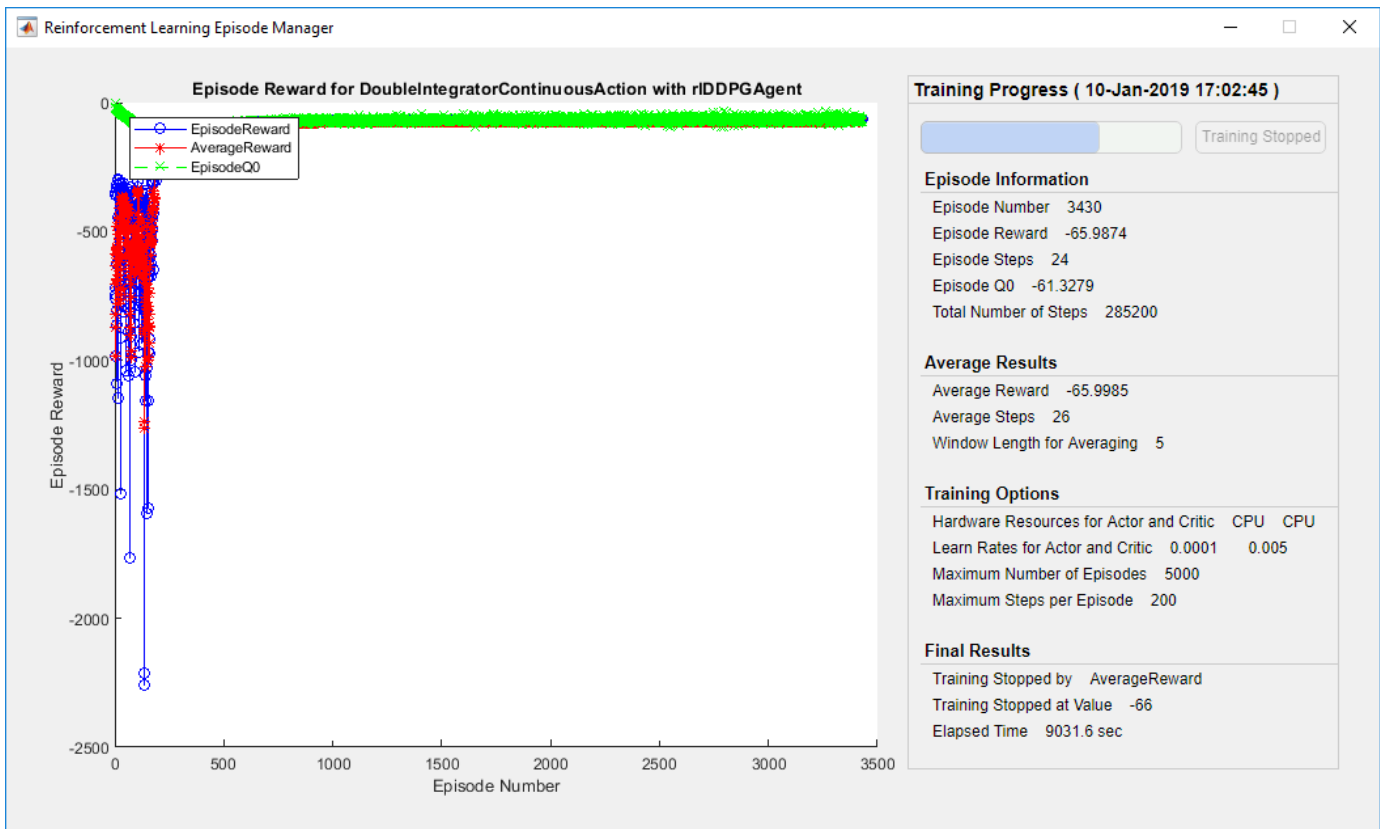

Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the double integrator environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```matlab
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```

```
totalReward = sum(experience.Reward)
```

```
totalReward = single
    -65.9933
```

See Also

`train`

More About

* "Deep Deterministic Policy Gradient Agents" on page 3-17
* "Train Reinforcement Learning Agents" on page 5-2
* "Create Policy and Value Function Representations" on page 4-2

Train DQN Agent to Swing Up and Balance Pendulum

This example shows how to train a deep Q-learning network (DQN) agent to swing up and balance a pendulum modeled in Simulink®.

For more information on DQN agents, see "Deep Q-Network Agents" on page 3-10. For an example that trains a DQN agent in MATLAB®, see "Train DQN Agent to Balance Cart-Pole System" on page 5-8.

Pendulum Swing-up Model

The reinforcement learning environment for this example is a simple frictionless pendulum that initially hangs in a downward position. The training goal is to make the pendulum stand upright without falling over using minimal control effort.

Open the model.

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


For this model:

- The upward balanced pendulum position is 0 radians, and the downward hanging position is `pi` radians.
- The torque action signal from the agent to the environment is from –2 to 2 N·m.

- The observations from the environment are the sine of the pendulum angle, the cosine of the pendulum angle, and the pendulum angle derivative.

- The reward r_t, provided at every timestep, is

$$r_t = -\left(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001u_{t-1}^2\right)$$

Here:

- θ_t is the angle of displacement from the upright position.

- $\dot{\theta}_t$ is the derivative of the displacement angle.

- u_{t-1} is the control effort from the previous time step.

For more information on this model, see "Load Predefined Simulink Environments" on page 2-22.

Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv('SimplePendulumModel-Discrete')

env =
SimulinkEnvWithAgent with properties:

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

The interface has a discrete action space where the agent can apply one of three possible torque values to the pendulum: –2, 0, or 2 N·m.

To define the initial condition of the pendulum as hanging downward, specify an environment reset function using an anonymous function handle. This reset function sets the model workspace variable `theta0` to `pi`.

```
env.ResetFcn = @(in)setVariable(in,'theta0',pi,'Workspace',mdl);
```

Get the observation and action specification information from the environment

```
obsInfo = getObservationInfo(env)

obsInfo =
  rlNumericSpec with properties:

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

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

```
actInfo =
  rlFiniteSetSpec with properties:

        Elements: [3x1 double]
            Name: "torque"
     Description: [0x0 string]
       Dimension: [1 1]
        DataType: "double"
```

Specify the simulation time `Tf` and the agent sample time `Ts` in seconds.

```
Ts = 0.05;
Tf = 20;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DQN Agent

A DQN agent approximates the long-term reward, given observations and actions, using a value function critic.

Since DQN has a discrete action space, it can rely on a multi-output critic approximator, which is generally a more efficient option than relying on a comparable single-output approximator. A multi-output approximator has only the observation as input and an output vector having as many elements as the number of possible discrete actions. Each output element represents the expected cumulative long-term reward following from the observation given as input, when the corresponding discrete action is taken.

To create the critic, first create a deep neural network with an input vector of three elements (for the sine, cosine, and derivative of the pendulum angle) and one output vector with three elements (–2, 0, or 2 Nm actions). For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
dnn = [
    featureInputLayer(3,'Normalization','none','Name','state')
    fullyConnectedLayer(24,'Name','CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(48,'Name','CriticStateFC2')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(3,'Name','output')];
```

View the critic network configuration.

```
figure
plot(layerGraph(dnn))
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOpts = rlRepresentationOptions('LearnRate',0.001,'GradientThreshold',1);
```

Create the critic representation using the specified deep neural network and options. You must also specify observation and action info for the critic. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(dnn,obsInfo,actInfo,'Observation',{'state'},criticOpts);
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions`.

```
agentOptions = rlDQNAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',3000,...
    'UseDoubleDQN',false,...
    'DiscountFactor',0.9,...
    'MiniBatchSize',64);
```

Then, create the DQN agent using the specified critic representation and agent options. For more information, see `rlDQNAgent`.

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

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

- Run each training for at most 1000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the Plots option) and disable the command line display (set the Verbose option to false).
- Stop training when the agent receives an average cumulative reward greater than –1100 over five consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.
- Save a copy of the agent for each episode where the cumulative reward is greater than –1100.

For more information, see rlTrainingOptions.

```
trainingOptions = rlTrainingOptions(...
    'MaxEpisodes',1000,...
    'MaxStepsPerEpisode',500,...
    'ScoreAveragingWindowLength',5,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-1100,...
    'SaveAgentCriteria','EpisodeReward',...
    'SaveAgentValue',-1100);
```

Train the agent using the train function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting doTraining to false. To train the agent yourself, set doTraining to true.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
    % Load the pretrained agent for the example.
    load('SimulinkPendulumDQNMulti.mat','agent');
end
```
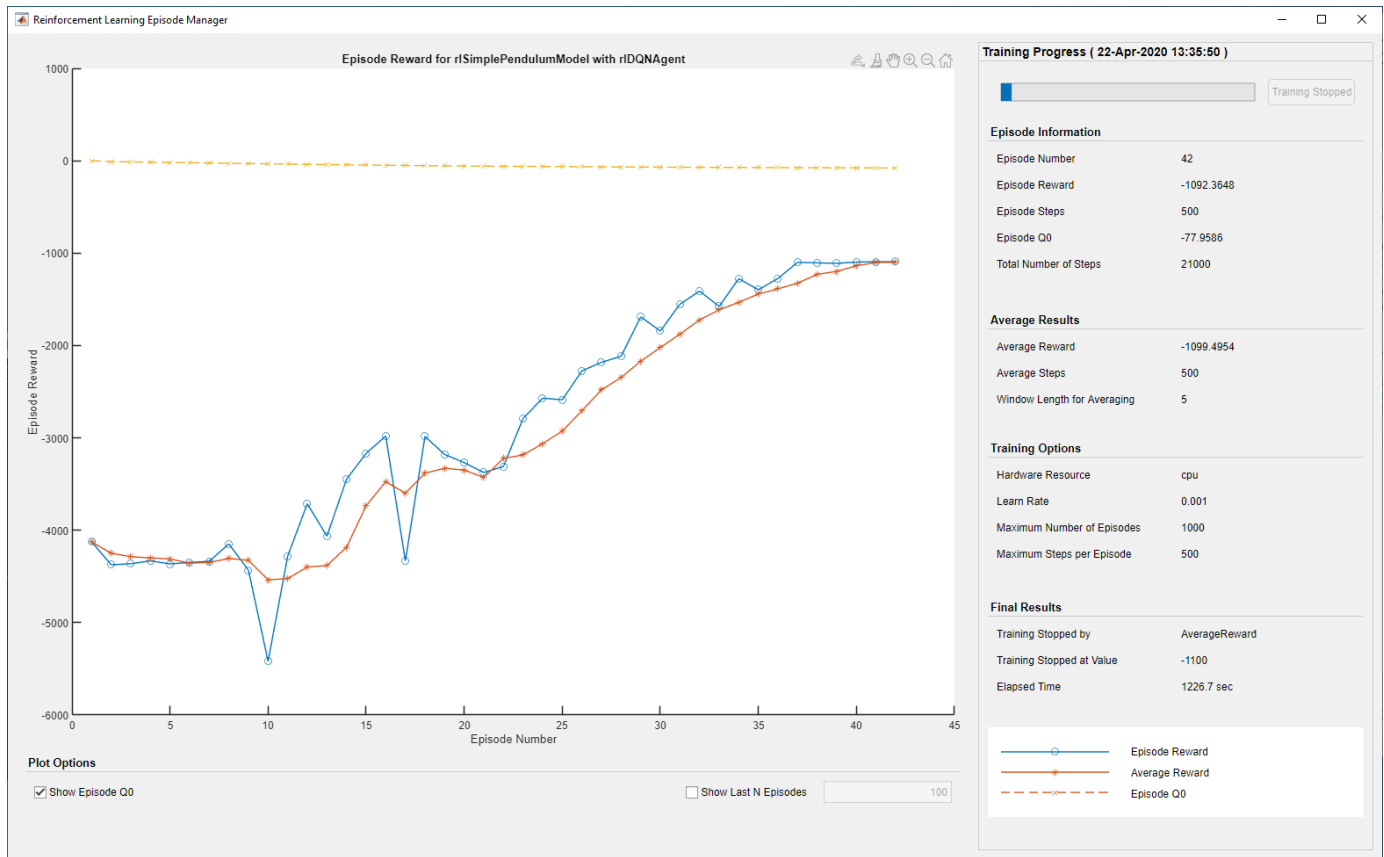
Simulate DQN Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```
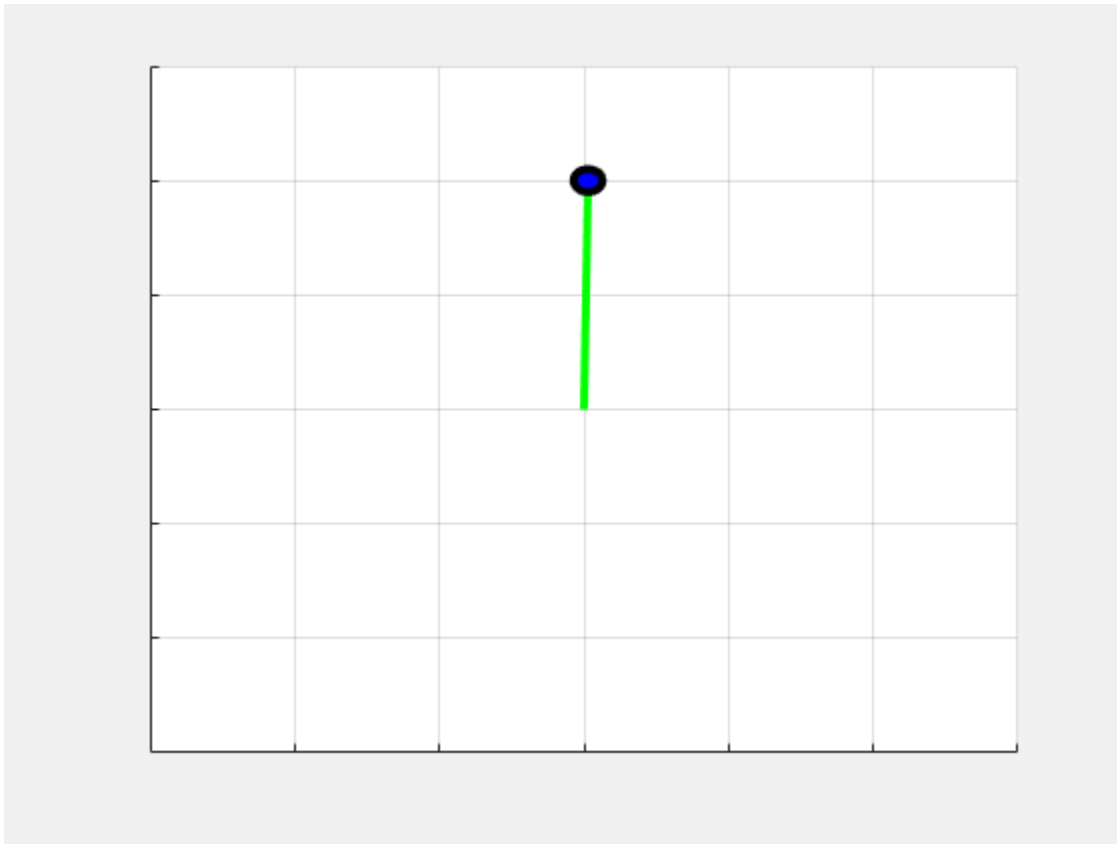
See Also
`rlDQNAgent`

More About
- "Deep Q-Network Agents" on page 3-10
- "Create Simulink Environments for Reinforcement Learning" on page 2-5

Train DDPG Agent to Swing Up and Balance Pendulum

This example shows how to train a deep deterministic policy gradient (DDPG) agent to swing up and balance a pendulum modeled in Simulink®.

For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17. For an example that trains a DDPG agent in MATLAB®, see "Train DDPG Agent to Control Double Integrator System" on page 5-31.

Pendulum Swing-Up Model

The reinforcement learning environment for this example is a simple frictionless pendulum that initially hangs in a downward position. The training goal is to make the pendulum stand upright without falling over using minimal control effort.

Open the model.

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


For this model:

- The upward balanced pendulum position is 0 radians, and the downward hanging position is `pi` radians.
- The torque action signal from the agent to the environment is from –2 to 2 N·m.

- The observations from the environment are the sine of the pendulum angle, the cosine of the pendulum angle, and the pendulum angle derivative.
- The reward r_t, provided at every time step, is

$$r_t = -\left(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001u_{t-1}^2\right)$$

Here:

- θ_t is the angle of displacement from the upright position.
- $\dot{\theta}_t$ is the derivative of the displacement angle.
- u_{t-1} is the control effort from the previous time step.

For more information on this model, see "Load Predefined Simulink Environments" on page 2-22.

Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv('SimplePendulumModel-Continuous')
```

```
env =
SimulinkEnvWithAgent with properties:

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

The interface has a continuous action space where the agent can apply torque values between –2 to 2 N·m to the pendulum.

Set the observations of the environment to be the sine of the pendulum angle, the cosine of the pendulum angle, and the pendulum angle derivative.

```
numObs = 3;
set_param('rlSimplePendulumModel/create observations','ThetaObservationHandling','sincos');
```

To define the initial condition of the pendulum as hanging downward, specify an environment reset function using an anonymous function handle. This reset function sets the model workspace variable theta0 to pi.

```
env.ResetFcn = @(in)setVariable(in,'theta0',pi,'Workspace',mdl);
```

Specify the simulation time Tf and the agent sample time Ts in seconds.

```
Ts = 0.05;
Tf = 20;
```

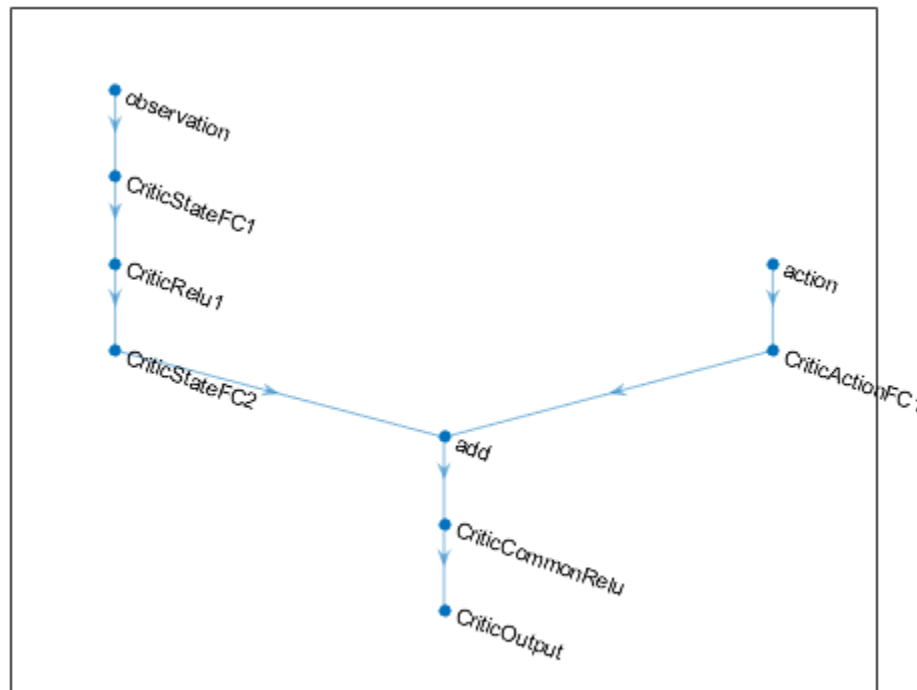Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG Agent

A DDPG agent approximates the long-term reward, given observations and actions, using a critic value function representation. To create the critic, first create a deep neural network with two inputs (the state and action) and one output. For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
statePath = [
    featureInputLayer(numObs,'Normalization','none','Name','observation')
    fullyConnectedLayer(400,'Name','CriticStateFC1')
    reluLayer('Name', 'CriticRelu1')
    fullyConnectedLayer(300,'Name','CriticStateFC2')];
actionPath = [
    featureInputLayer(1,'Normalization','none','Name','action')
    fullyConnectedLayer(300,'Name','CriticActionFC1','BiasLearnRateFactor',0)];
commonPath = [
    additionLayer(2,'Name','add')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(1,'Name','CriticOutput')];

criticNetwork = layerGraph();
criticNetwork = addLayers(criticNetwork,statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);

criticNetwork = connectLayers(criticNetwork,'CriticStateFC2','add/in1');
criticNetwork = connectLayers(criticNetwork,'CriticActionFC1','add/in2');
```

View the critic network configuration.

```
figure
plot(criticNetwork)
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOpts = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation`.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,'Observation',{'observation'},'Act:
```

A DDPG agent decides which action to take given observations using an actor representation. To create the actor, first create a deep neural network with one input, the observation, and one output, the action.

Construct the actor in a manner similar to the critic. For more information, see `rlDeterministicActorRepresentation`.

```
actorNetwork = [
    featureInputLayer(numObs,'Normalization','none','Name','observation')
    fullyConnectedLayer(400,'Name','ActorFC1')
    reluLayer('Name','ActorRelu1')
    fullyConnectedLayer(300,'Name','ActorFC2')
    reluLayer('Name','ActorRelu2')
    fullyConnectedLayer(1,'Name','ActorFC3')
    tanhLayer('Name','ActorTanh')
```

```
        scalingLayer('Name','ActorScaling','Scale',max(actInfo.UpperLimit))];

    actorOpts = rlRepresentationOptions('LearnRate',1e-04,'GradientThreshold',1);

    actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,'Observation',{'observat:
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOpts = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',128);
agentOpts.NoiseOptions.Variance = 0.6;
agentOpts.NoiseOptions.VarianceDecayRate = 1e-5;
```

Then create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent`.

```
agent = rlDDPGAgent(actor,critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

- Run training for at most 50000 episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –740 over five consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.
- Save a copy of the agent for each episode where the cumulative reward is greater than –740.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'ScoreAveragingWindowLength',5,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-740,...
    'SaveAgentCriteria','EpisodeReward',...
    'SaveAgentValue',-740);
```

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
```

```
      % Train the agent.
      trainingStats = train(agent,env,trainOpts);
else
      % Load the pretrained agent for the example.
      load('SimulinkPendulumDDPG.mat','agent')
end
```
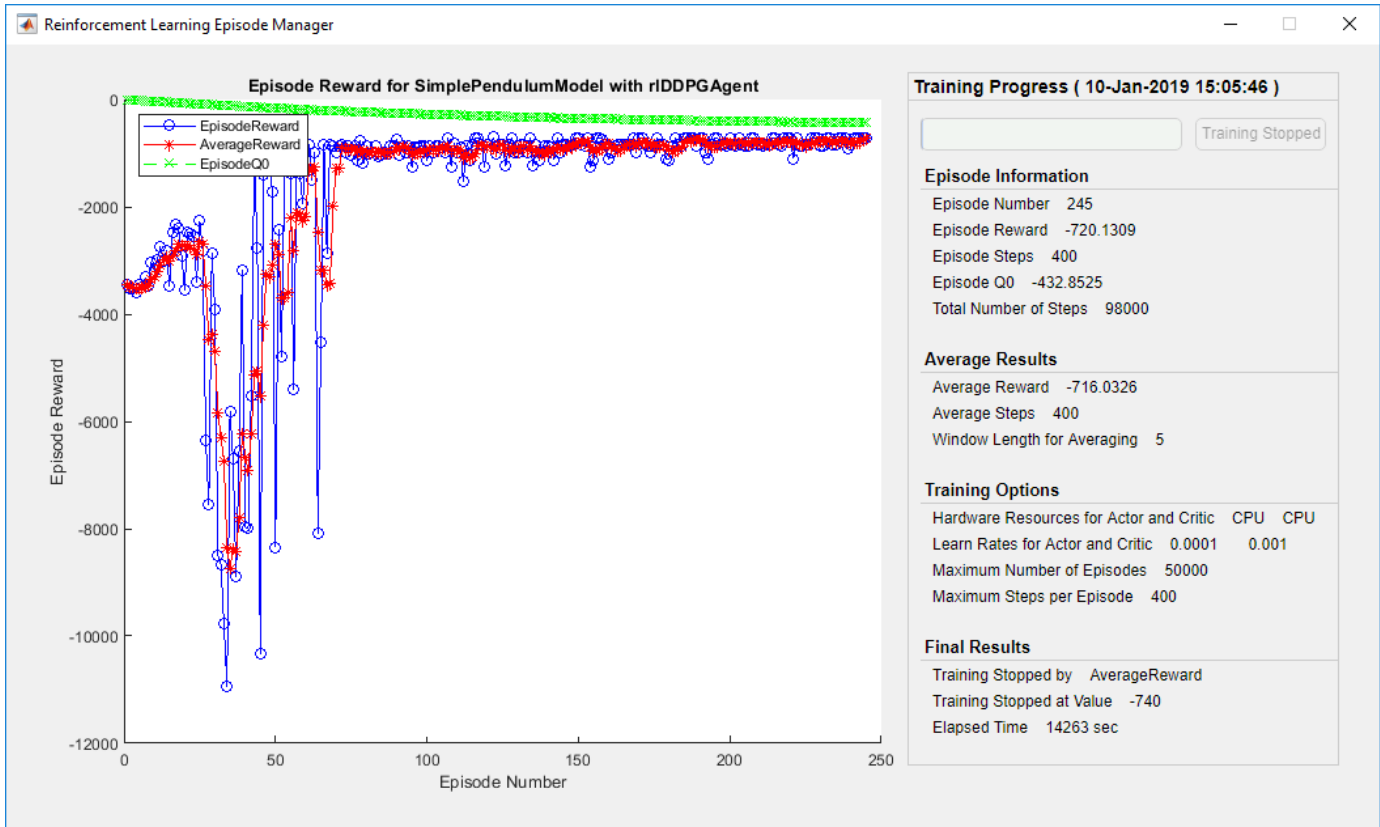

Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```
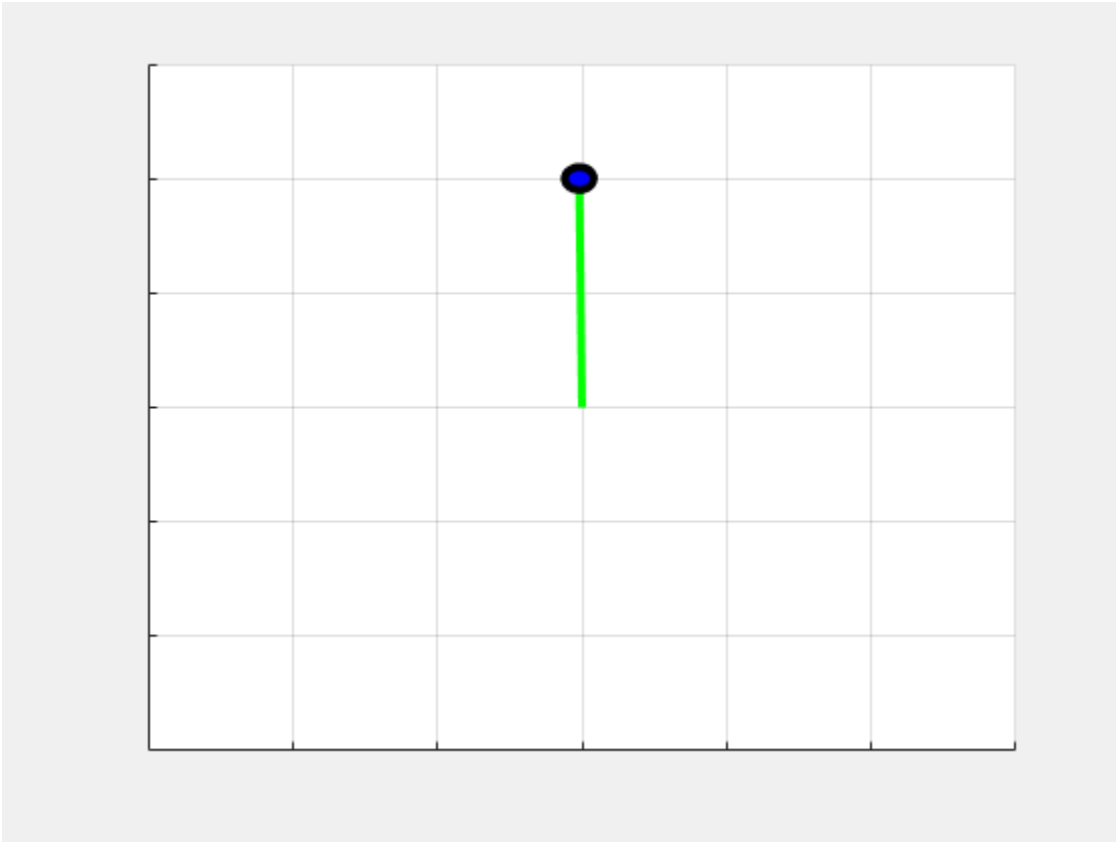
See Also
`rlDDPGAgent` | `rlSimulinkEnv` | `train`

More About
- "Create Simulink Environments for Reinforcement Learning" on page 2-5
- "Deep Deterministic Policy Gradient Agents" on page 3-17

Train DDPG Agent to Swing Up and Balance Cart-Pole System

This example shows how to train a deep deterministic policy gradient (DDPG) agent to swing up and balance a cart-pole system modeled in Simscape™ Multibody™.

For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17. For an example showing how to train a DDPG agent in MATLAB®, see "Train DDPG Agent to Control Double Integrator System" on page 5-31.

Cart-Pole Simscape Model

The reinforcement learning environment for this example is a pole attached to an unactuated joint on a cart, which moves along a frictionless track. The training goal is to make the pole stand upright without falling over using minimal control effort.

Open the model.

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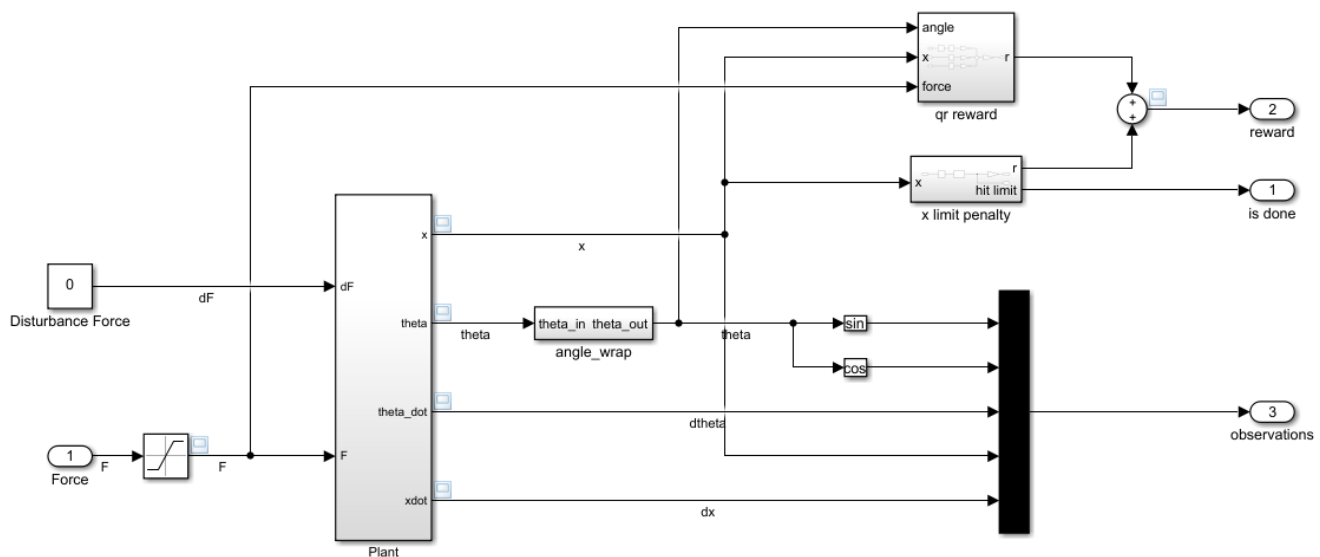

The cart-pole system is modeled using Simscape Multibody.

For this model:

- The upward balanced pole position is `0` radians, and the downward hanging position is `pi` radians.
- The force action signal from the agent to the environment is from –15 to 15 N.
- The observations from the environment are the position and velocity of the cart, and the sine, cosine, and derivative of the pole angle.
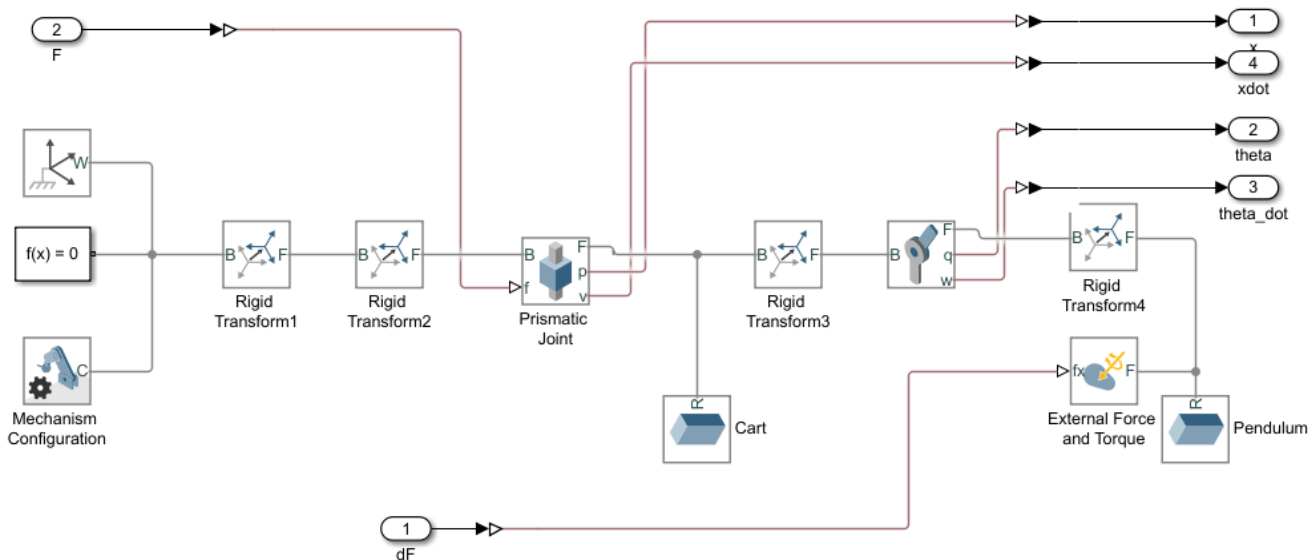- The episode terminates if the cart moves more than 3.5 m from the original position.
- The reward r_t, provided at every timestep, is

$$r_t = -0.1\left(5\theta_t^2 + x_t^2 + 0.05u_{t-1}^2\right) - 100B$$

Here:

- θ_t is the angle of displacement from the upright position of the pole.
- x_t is the position displacement from the center position of the cart.
- u_{t-1} is the control effort from the previous time step.
- B is a flag (1 or 0) that indicates whether the cart is out of bounds.

For more information on this model, see "Load Predefined Simulink Environments" on page 2-22.

Create Environment Interface

Create a predefined environment interface for the pole.

```
env = rlPredefinedEnv('CartPoleSimscapeModel-Continuous')
```

```
env =
SimulinkEnvWithAgent with properties:

        Model : rlCartPoleSimscapeModel
```

```
    AgentBlock : rlCartPoleSimscapeModel/RL Agent
      ResetFcn : []
UseFastRestart : on
```

The interface has a continuous action space where the agent can apply possible torque values from –15 to 15 N to the pole.

Obtain the observation and action information from the environment interface.

```
obsInfo = getObservationInfo(env);
numObservations = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
```

Specify the simulation time `Tf` and the agent sample time `Ts` in seconds

```
Ts = 0.02;
Tf = 25;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG Agent

A DDPG agent approximates the long-term reward, given observations and actions, using a critic value function representation. To create the critic, first create a deep neural network with two inputs (the state and action) and one output. The input size of action path is [1 1 1] since the agent can apply an action as one force value to the environment. For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
statePath = [
    featureInputLayer(numObservations,'Normalization','none','Name','observation')
    fullyConnectedLayer(128,'Name','CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(200,'Name','CriticStateFC2')];

actionPath = [
    featureInputLayer(1,'Normalization','none','Name','action')
    fullyConnectedLayer(200,'Name','CriticActionFC1','BiasLearnRateFactor',0)];

commonPath = [
    additionLayer(2,'Name','add')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(1,'Name','CriticOutput')];

criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);

criticNetwork = connectLayers(criticNetwork,'CriticStateFC2','add/in1');
criticNetwork = connectLayers(criticNetwork,'CriticActionFC1','add/in2');
```
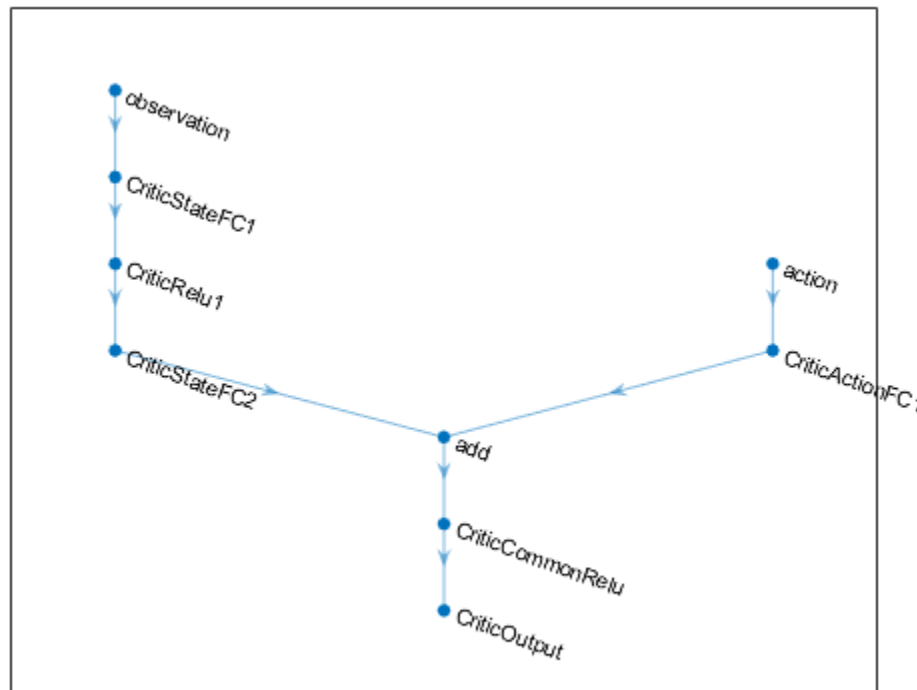
View the critic network configuration.

```
figure
plot(criticNetwork)
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation information for the critic, which you already obtained from the environment interface. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
```

A DDPG agent decides which action to take, given observations, using an actor representation. To create the actor, first create a deep neural network with one input (the observation) and one output (the action).

Construct the actor in a similar manner to the critic. For more information, see `rlDeterministicActorRepresentation`.

```
actorNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','observation')
    fullyConnectedLayer(128,'Name','ActorFC1')
    reluLayer('Name','ActorRelu1')
    fullyConnectedLayer(200,'Name','ActorFC2')
    reluLayer('Name','ActorRelu2')
    fullyConnectedLayer(1,'Name','ActorFC3')
    tanhLayer('Name','ActorTanh1')
    scalingLayer('Name','ActorScaling','Scale',max(actInfo.UpperLimit))];
```

```
actorOptions = rlRepresentationOptions('LearnRate',5e-04,'GradientThreshold',1);

actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'ActorScaling'},actorOptions);
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'MiniBatchSize',128);
agentOptions.NoiseOptions.Variance = 0.4;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Then, create the agent using the specified actor representation, critic representation and agent options. For more information, see `rlDDPGAgent`.

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

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

- Run each training episode for at most 2000 episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –400 over five consecutive episodes. At this point, the agent can quickly balance the pole in the upright position using minimal control effort.
- Save a copy of the agent for each episode where the cumulative reward is greater than –400.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 2000;
maxsteps = ceil(Tf/Ts);
trainingOptions = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'ScoreAveragingWindowLength',5,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-400,...
    'SaveAgentCriteria','EpisodeReward',...
    'SaveAgentValue',-400);
```

Train the agent using the `train` function. Training this agent process is computationally intensive and takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
```

```
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
    % Load the pretrained agent for the example.
    load('SimscapeCartPoleDDPG.mat','agent')
end
```
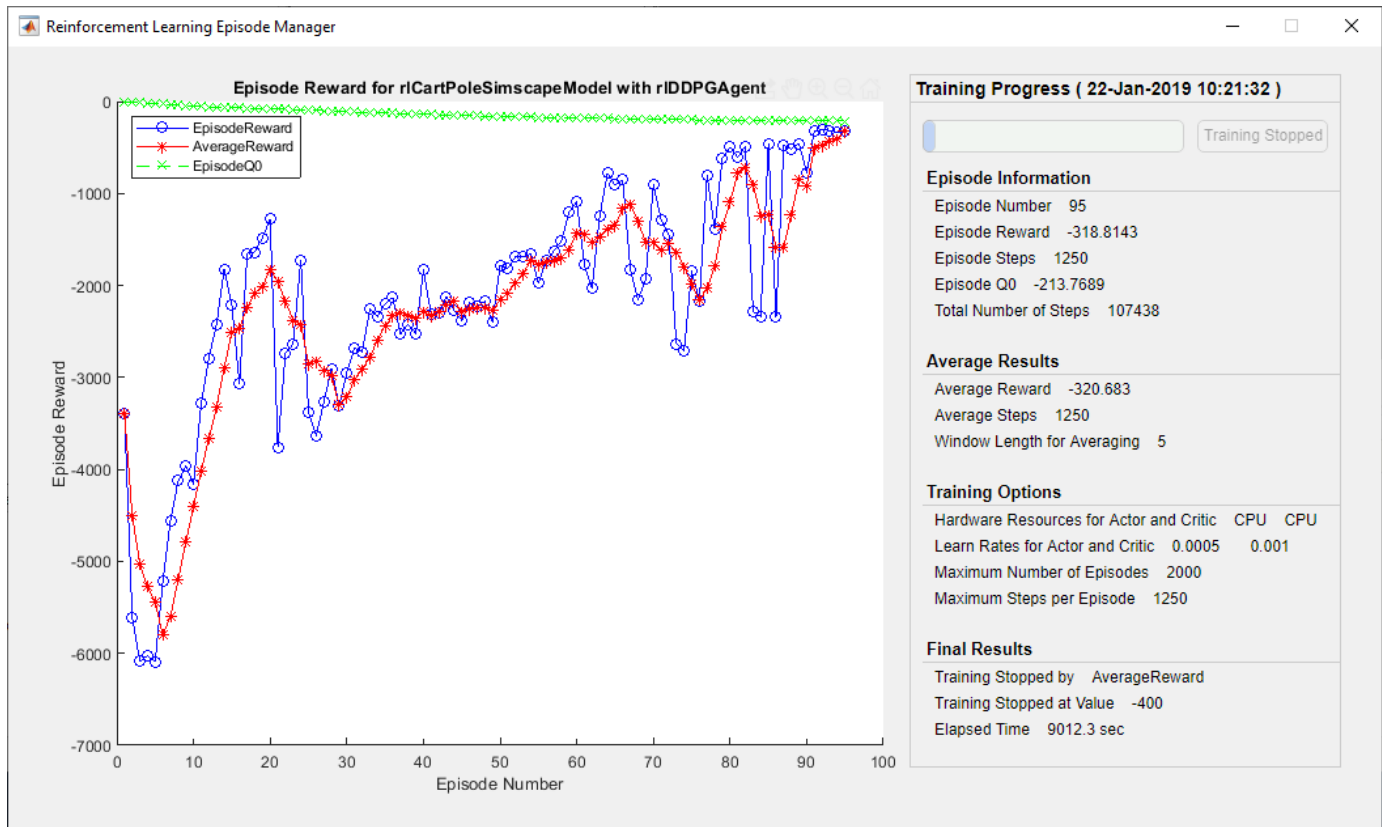

Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```
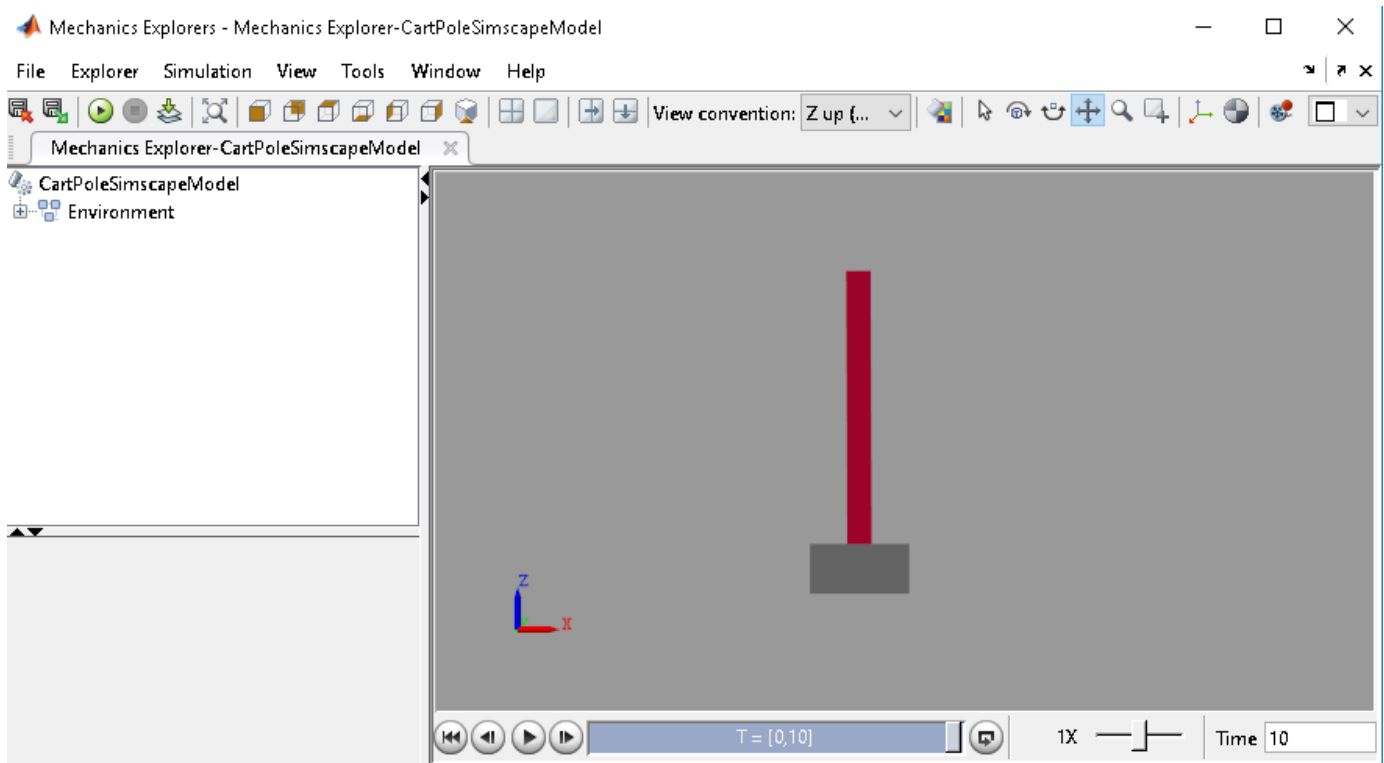
```
bdclose(mdl)
```

See Also
rlDDPGAgent | rlSimulinkEnv | train

More About
- "Create Simulink Environments for Reinforcement Learning" on page 2-5
- "Deep Deterministic Policy Gradient Agents" on page 3-17

Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal

This example shows how to convert a simple frictionless pendulum Simulink® model to a reinforcement learning environment interface, and trains a deep deterministic policy gradient (DDPG) agent in this environment.

For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17. For an example showing how to train a DDPG agent in MATLAB®, see "Train DDPG Agent to Control Double Integrator System" on page 5-31.

Pendulum Swing-Up Model with Bus

The starting model for this example is a simple frictionless pendulum. The training goal is to make the pendulum stand upright without falling over using minimal control effort.

Open the model.

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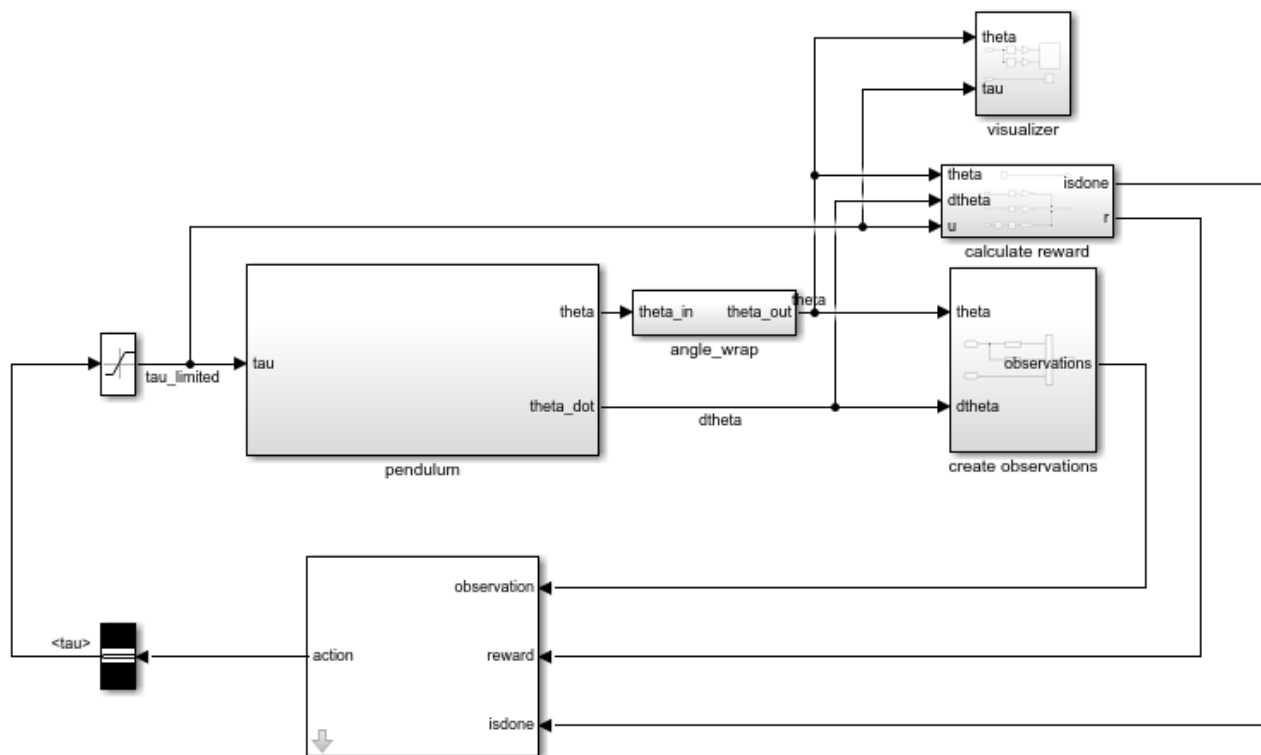

For this model:

- The upward balanced pendulum position is 0 radians, and the downward hanging position is pi radians.

- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are the sine of the pendulum angle, the cosine of the pendulum angle, and the pendulum angle derivative.
- Both the observation and action signals are Simulink buses.
- The reward r_t, provided at every time step, is

$$r_t = -\left(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001u_{t-1}^2\right)$$

Here:

- θ_t is the angle of displacement from the upright position.
- $\dot{\theta}_t$ is the derivative of the displacement angle.
- u_{t-1} is the control effort from the previous time step.

The model used in this example is similar to the simple pendulum model described in "Load Predefined Simulink Environments" on page 2-22. The difference is that the model in this example uses Simulink buses for the action and observation signals.

Create Environment Interface with Bus

The environment interface from a Simulink model is created using `rlSimulinkEnv`, which requires the name of the Simulink model, the path to the agent block, and observation and action reinforcement learning data specifications. For models that use bus signals for actions or observations, you can create the corresponding specifications using the `bus2RLSpec` function.

Specify the path to the agent block.

```
agentBlk = 'rlSimplePendulumModelBus/RL Agent';
```

Create the observation `Bus` object.

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

Create the action `Bus` object.

```
actBus = Simulink.Bus();
act(1) = Simulink.BusElement;
act(1).Name = 'tau';
act(1).Min = -2;
act(1).Max = 2;
actBus.Elements = act;
```

Create the action and observation specification objects using the Simulink buses.

```
obsInfo = bus2RLSpec('obsBus','Model',mdl);
actInfo = bus2RLSpec('actBus','Model',mdl);
```

Create the reinforcement learning environment for the pendulum model.

```
env = rlSimulinkEnv(mdl,agentBlk,obsInfo,actInfo);
```

To define the initial condition of the pendulum as hanging downward, specify an environment reset function using an anonymous function handle. This reset function sets the model workspace variable `theta0` to `pi`.

```
env.ResetFcn = @(in)setVariable(in,'theta0',pi,'Workspace',mdl);
```

Specify the simulation time `Tf` and the agent sample time `Ts` in seconds.

```
Ts = 0.05;
Tf = 20;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG Agent

A DDPG agent decides which action to take, given observations, using an actor representation. To create the actor, first create a deep neural network with three inputs (the observations) and one output (the action). The three observations can be combined using a `concatenationLayer`.

For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
sinThetaInput = featureInputLayer(1,'Normalization','none','Name','sin_theta');
cosThetaInput = featureInputLayer(1,'Normalization','none','Name','cos_theta');
dThetaInput   = featureInputLayer(1,'Normalization','none','Name','dtheta');
commonPath = [
    concatenationLayer(1,3,'Name','concat')
    fullyConnectedLayer(400, 'Name','ActorFC1')
    reluLayer('Name','ActorRelu1')
    fullyConnectedLayer(300,'Name','ActorFC2')
    reluLayer('Name','ActorRelu2')
    fullyConnectedLayer(1,'Name','ActorFC3')
    tanhLayer('Name','ActorTanh1')
    scalingLayer('Name','ActorScaling1','Scale',max(actInfo.UpperLimit))];

actorNetwork = layerGraph(sinThetaInput);
actorNetwork = addLayers(actorNetwork,cosThetaInput);
actorNetwork = addLayers(actorNetwork,dThetaInput);
actorNetwork = addLayers(actorNetwork,commonPath);

actorNetwork = connectLayers(actorNetwork,'sin_theta','concat/in1');
actorNetwork = connectLayers(actorNetwork,'cos_theta','concat/in2');
actorNetwork = connectLayers(actorNetwork,'dtheta','concat/in3');
```

View the actor network configuration.
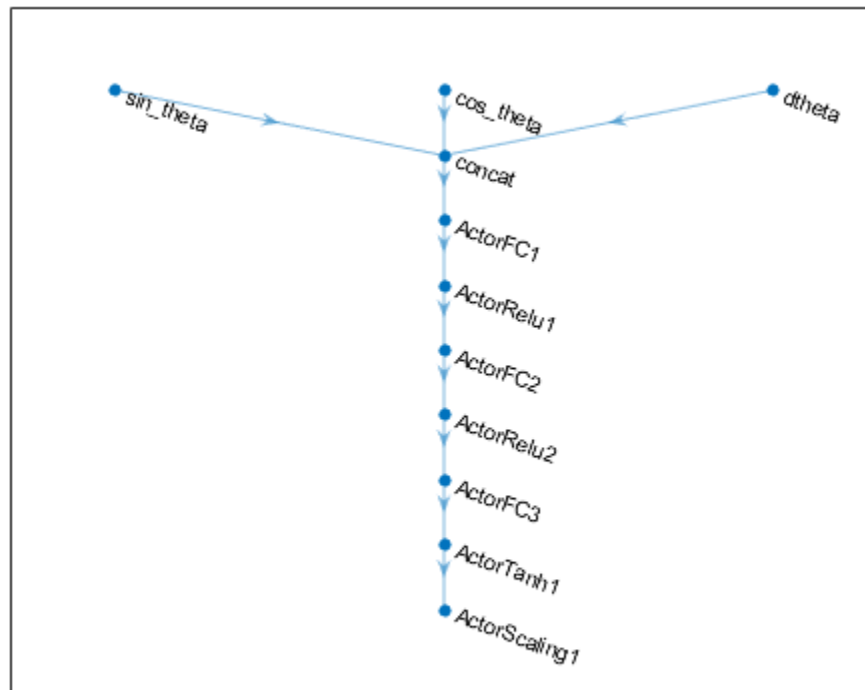
```
figure
plot(actorNetwork)
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
actorOptions = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1);
```

Create the actor representation using the specified deep neural network and options. You must also specify the action and observation info for the actor, which you obtained from the environment interface. For more information, see `rlDeterministicActorRepresentation`.

```
actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'sin_theta','cos_theta','dtheta'},'Action',{'ActorScaling1'},actorOptions);
```

A DDPG agent approximates the long-term reward given observations and actions using a critic value function representation. To create the critic, first create a deep neural network with two inputs, the observation and action, and one output, the state action value.

Construct the critic in a similar manner to the actor. For more information, see `rlQValueRepresentation`.

```
statePath = [
    concatenationLayer(1,3,'Name','concat')
    fullyConnectedLayer(400,'Name','CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(300,'Name','CriticStateFC2')];

actionPath = [
    featureInputLayer(1,'Normalization','none','Name', 'action')
    fullyConnectedLayer(300,'Name','CriticActionFC1','BiasLearnRateFactor', 0)];
```

```
commonPath = [
    additionLayer(2,'Name','add')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(1,'Name','CriticOutput')];

criticNetwork = layerGraph(sinThetaInput);
criticNetwork = addLayers(criticNetwork,cosThetaInput);
criticNetwork = addLayers(criticNetwork,dThetaInput);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,statePath);
criticNetwork = addLayers(criticNetwork,commonPath);

criticNetwork = connectLayers(criticNetwork,'sin_theta','concat/in1');
criticNetwork = connectLayers(criticNetwork,'cos_theta','concat/in2');
criticNetwork = connectLayers(criticNetwork,'dtheta','concat/in3');
criticNetwork = connectLayers(criticNetwork,'CriticStateFC2','add/in1');
criticNetwork = connectLayers(criticNetwork,'CriticActionFC1','add/in2');

criticOpts = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);

critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
                    'Observation',{'sin_theta','cos_theta','dtheta'},'Action',{'action'},cri
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOpts = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',128);
agentOpts.NoiseOptions.Variance = 0.6;
agentOpts.NoiseOptions.VarianceDecayRate = 1e-5;
```

Then create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent`.

```
agent = rlDDPGAgent(actor,critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run each training for at most 50000 episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –740 over five consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.
- Save a copy of the agent for each episode where the cumulative reward is greater than –740.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
```

```matlab
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'ScoreAveragingWindowLength',5,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-740);
```

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```matlab
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('SimulinkPendBusDDPG.mat','agent')
end
```
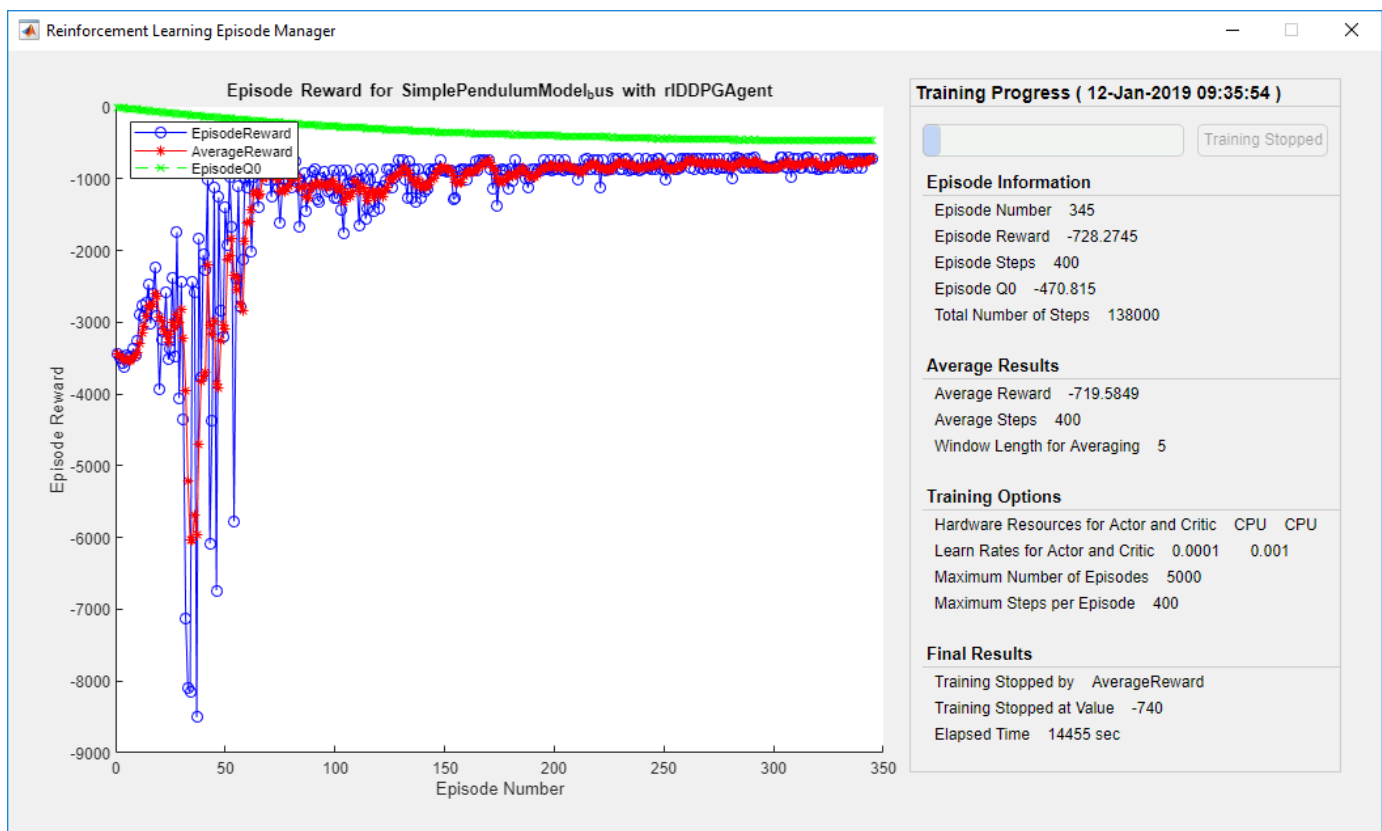

Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```
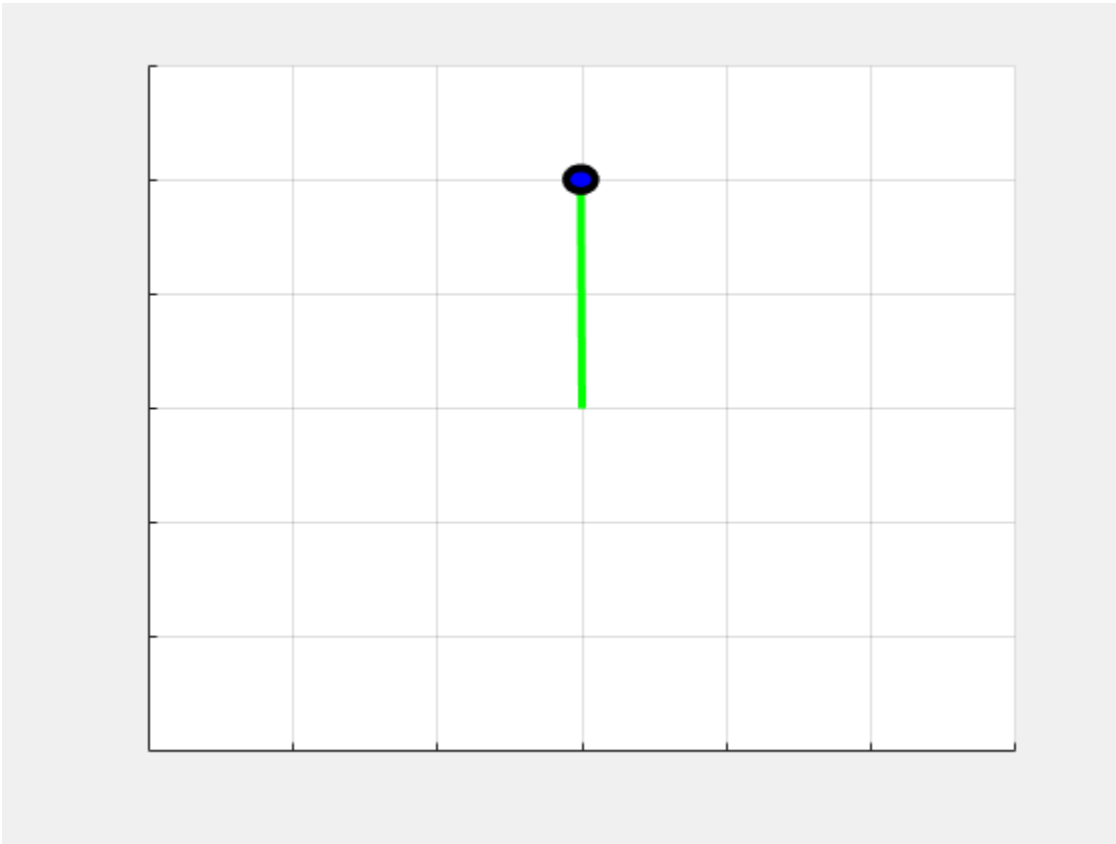

See Also
bus2RLSpec | rlDDPGAgent | rlSimulinkEnv | train

More About
- "Create Simulink Environments for Reinforcement Learning" on page 2-5
- "Deep Deterministic Policy Gradient Agents" on page 3-17

Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation

This example shows how to train a deep deterministic policy gradient (DDPG) agent to swing up and balance a pendulum with an image observation modeled in MATLAB®.

For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17.

Simple Pendulum with Image MATLAB Environment

The reinforcement learning environment for this example is a simple frictionless pendulum that initially hangs in a downward position. The training goal is to make the pendulum stand upright without falling over using minimal control effort.

For this environment:

- The upward balanced pendulum position is `0` radians, and the downward hanging position is `pi` radians.
- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are an image indicating the location of the pendulum mass and the pendulum angular velocity.
- The reward r_t, provided at every time step, is

$$r_t = -\left(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001u_{t-1}^2\right)$$

Here:

- θ_t is the angle of displacement from the upright position.
- $\dot{\theta}_t$ is the derivative of the displacement angle.
- u_{t-1} is the control effort from the previous time step.

For more information on this model, see "Load Predefined Control System Environments" on page 2-15.

Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv('SimplePendulumWithImage-Continuous')
```

```
env =
  SimplePendlumWithImageContinuousAction with properties:

            Mass: 1
       RodLength: 1
      RodInertia: 0
         Gravity: 9.8100
     DampingRatio: 0
    MaximumTorque: 2
              Ts: 0.0500
           State: [2x1 double]
```

```
                    Q: [2x2 double]
                    R: 1.0000e-03
```

The interface has a continuous action space where the agent can apply a torque between –2 to 2 N·m.

Obtain the observation and action specification from the environment interface.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG Agent

A DDPG agent approximates the long-term reward, given observations and actions, using a critic value function representation. To create the critic, first create a deep convolutional neural network (CNN) with three inputs (the image, angular velocity, and action) and one output. For more information on creating representations, see "Create Policy and Value Function Representations" on page 4-2.

```
hiddenLayerSize1 = 400;
hiddenLayerSize2 = 300;

imgPath = [
    imageInputLayer(obsInfo(1).Dimension,'Normalization','none','Name',obsInfo(1).Name)
    convolution2dLayer(10,2,'Name','conv1','Stride',5,'Padding',0)
    reluLayer('Name','relu1')
    fullyConnectedLayer(2,'Name','fc1')
    concatenationLayer(3,2,'Name','cat1')
    fullyConnectedLayer(hiddenLayerSize1,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSize2,'Name','fc3')
    additionLayer(2,'Name','add')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')
    ];
dthetaPath = [
    imageInputLayer(obsInfo(2).Dimension,'Normalization','none','Name',obsInfo(2).Name)
    fullyConnectedLayer(1,'Name','fc5','BiasLearnRateFactor',0,'Bias',0)
    ];
actPath =[
    imageInputLayer(actInfo(1).Dimension,'Normalization','none','Name','action')
    fullyConnectedLayer(hiddenLayerSize2,'Name','fc6','BiasLearnRateFactor',0,'Bias',zeros(hidden
    ];

criticNetwork = layerGraph(imgPath);
criticNetwork = addLayers(criticNetwork,dthetaPath);
criticNetwork = addLayers(criticNetwork,actPath);
criticNetwork = connectLayers(criticNetwork,'fc5','cat1/in2');
criticNetwork = connectLayers(criticNetwork,'fc6','add/in2');
```

View the critic network configuration.
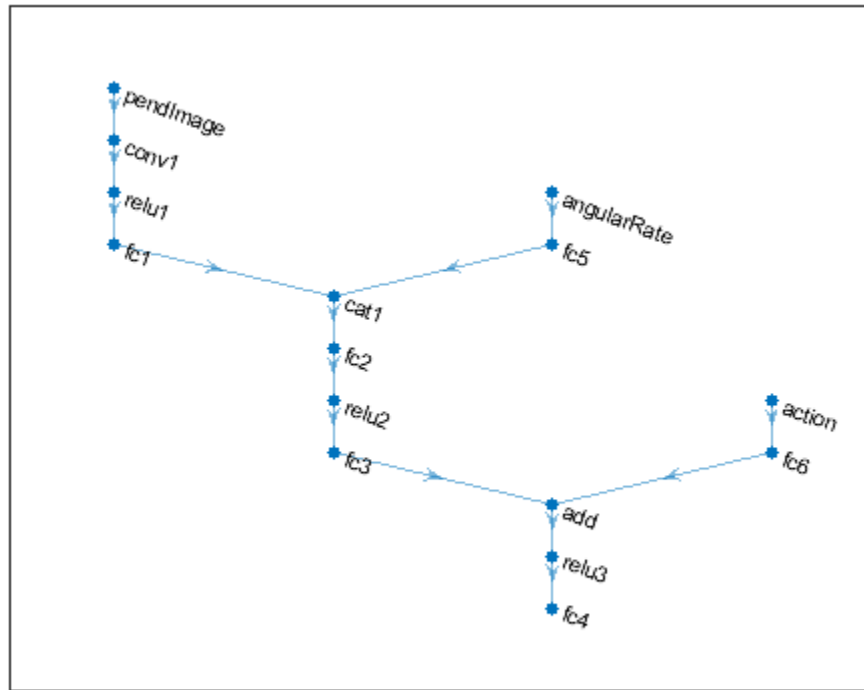
```
figure
plot(criticNetwork)
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Uncomment the following line to use the GPU to accelerate training of the critic CNN. Using a GPU requires Parallel Computing Toolbox™ software and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
% criticOptions.UseDevice = 'gpu';
```

Create the critic representation using the specified neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation',{'pendImage','angularRate'},'Action',{'action'},criticOptions);
```

A DDPG agent decides which action to take given observations using an actor representation. To create the actor, first create a deep convolutional neural network (CNN) with two inputs (the image and angular velocity) and one output (the action).

Construct the actor in a similar manner to the critic.

```
imgPath = [
    imageInputLayer(obsInfo(1).Dimension,'Normalization','none','Name',obsInfo(1).Name)
    convolution2dLayer(10,2,'Name','conv1','Stride',5,'Padding',0)
    reluLayer('Name','relu1')
```

```
    fullyConnectedLayer(2,'Name','fc1')
    concatenationLayer(3,2,'Name','cat1')
    fullyConnectedLayer(hiddenLayerSize1,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSize2,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')
    tanhLayer('Name','tanh1')
    scalingLayer('Name','scale1','Scale',max(actInfo.UpperLimit))
    ];
dthetaPath = [
    imageInputLayer(obsInfo(2).Dimension,'Normalization','none','Name',obsInfo(2).Name)
    fullyConnectedLayer(1,'Name','fc5','BiasLearnRateFactor',0,'Bias',0)
    ];

actorNetwork = layerGraph(imgPath);
actorNetwork = addLayers(actorNetwork,dthetaPath);
actorNetwork = connectLayers(actorNetwork,'fc5','cat1/in2');

actorOptions = rlRepresentationOptions('LearnRate',1e-04,'GradientThreshold',1);
```

Uncomment the following line to use the GPU to accelerate training of the actor CNN.

```
% actorOptions.UseDevice = 'gpu';
```

Create the actor representation using the specified neural network and options. For more information, see rlDeterministicActorRepresentation.

```
actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,'Observation',{'pendImage
```

View the actor network configuration.

```
figure
plot(actorNetwork)
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',env.Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',128);
agentOptions.NoiseOptions.Variance = 0.6;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;
```

Then create the agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent`.

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

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

- Run each training for at most 5000 episodes, with each episode lasting at most 400 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option).
- Stop training when the agent receives a moving average cumulative reward greater than -740 over ten consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = 400;
trainingOptions = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-740);
```

You can visualize the pendulum by using the `plot` function during training or simulation.
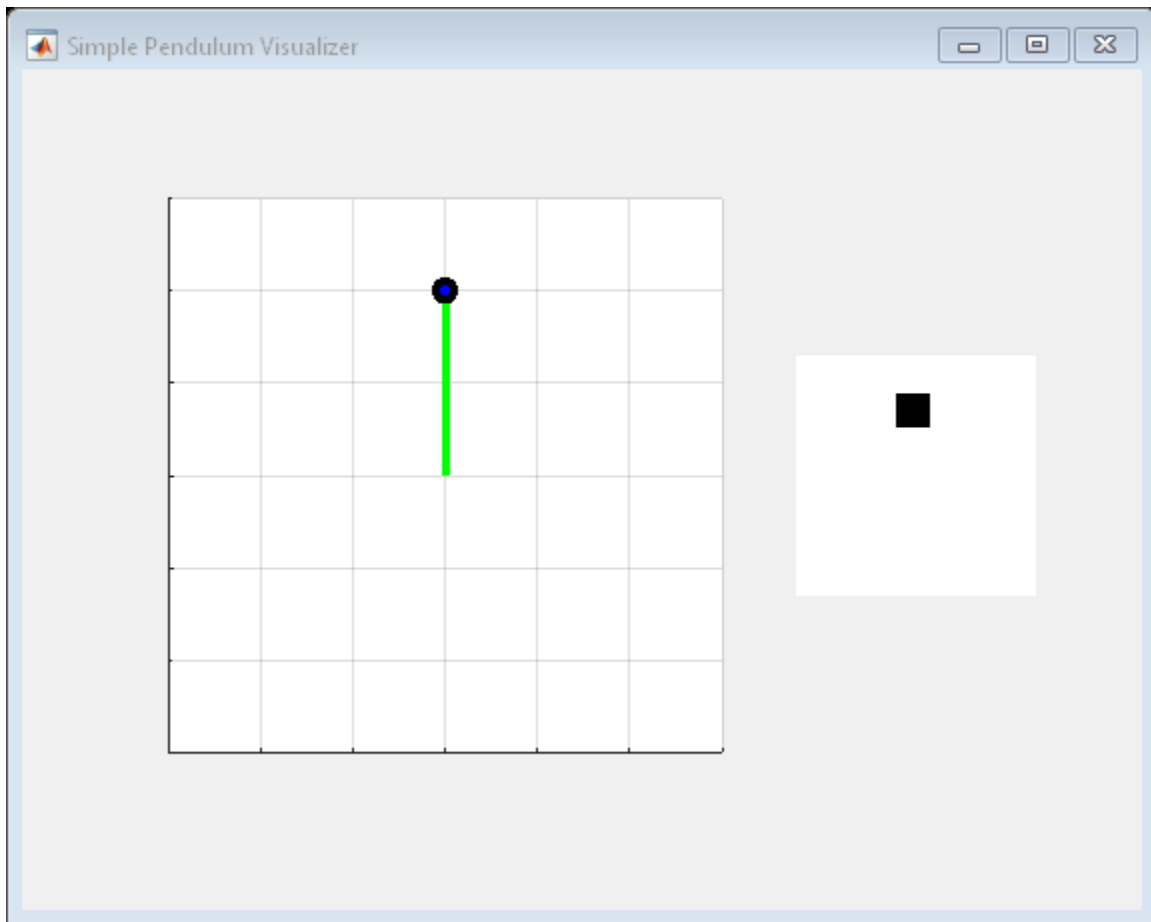
```
plot(env)
```


Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
    % Load pretrained agent for the example.
    load('SimplePendulumWithImageDDPG.mat','agent')
end
```
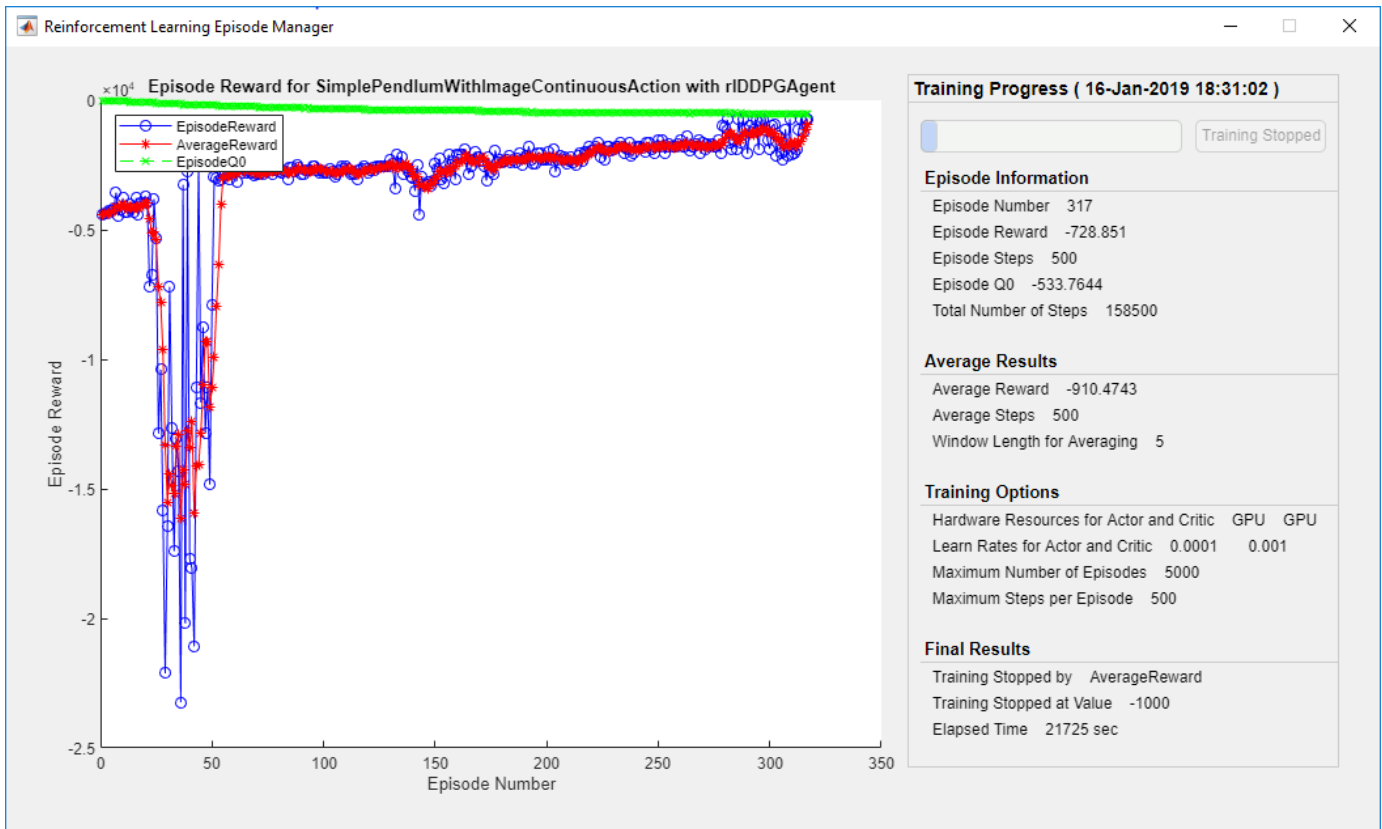
Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```
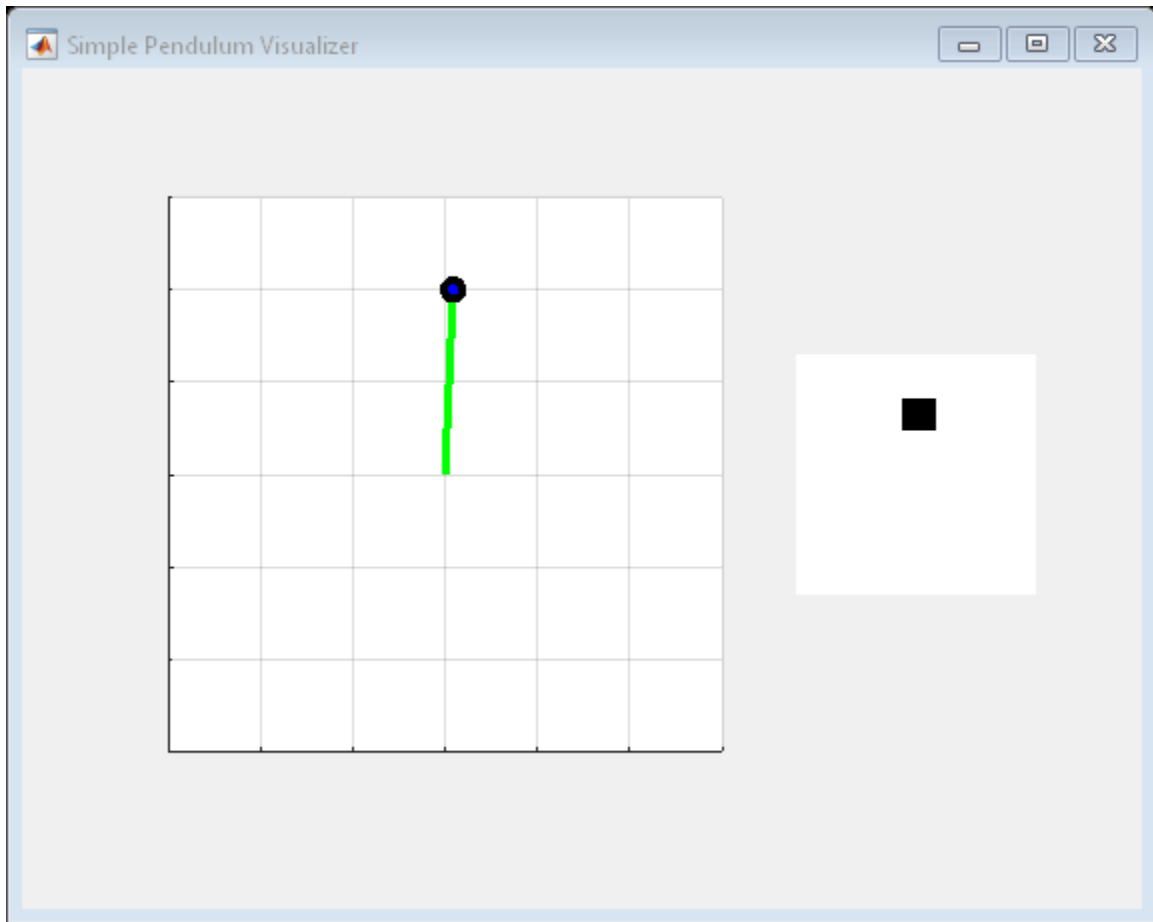
See Also
`train`

More About
- "Deep Deterministic Policy Gradient Agents" on page 3-17
- "Train Reinforcement Learning Agents" on page 5-2
- "Create Policy and Value Function Representations" on page 4-2

Create Agent Using Deep Network Designer and Train Using Image Observations

This example shows how to create a deep Q-learning network (DQN) agent that can swing up and balance a pendulum modeled in MATLAB®. In this example, you create the DQN agent using Deep Network Designer. For more information on DQN agents, see "Deep Q-Network Agents" on page 3-10.

Pendulum Swing-Up with Image MATLAB Environment

The reinforcement learning environment for this example is a simple frictionless pendulum that initially hangs in a downward position. The training goal is to make the pendulum stand upright without falling over using minimal control effort.
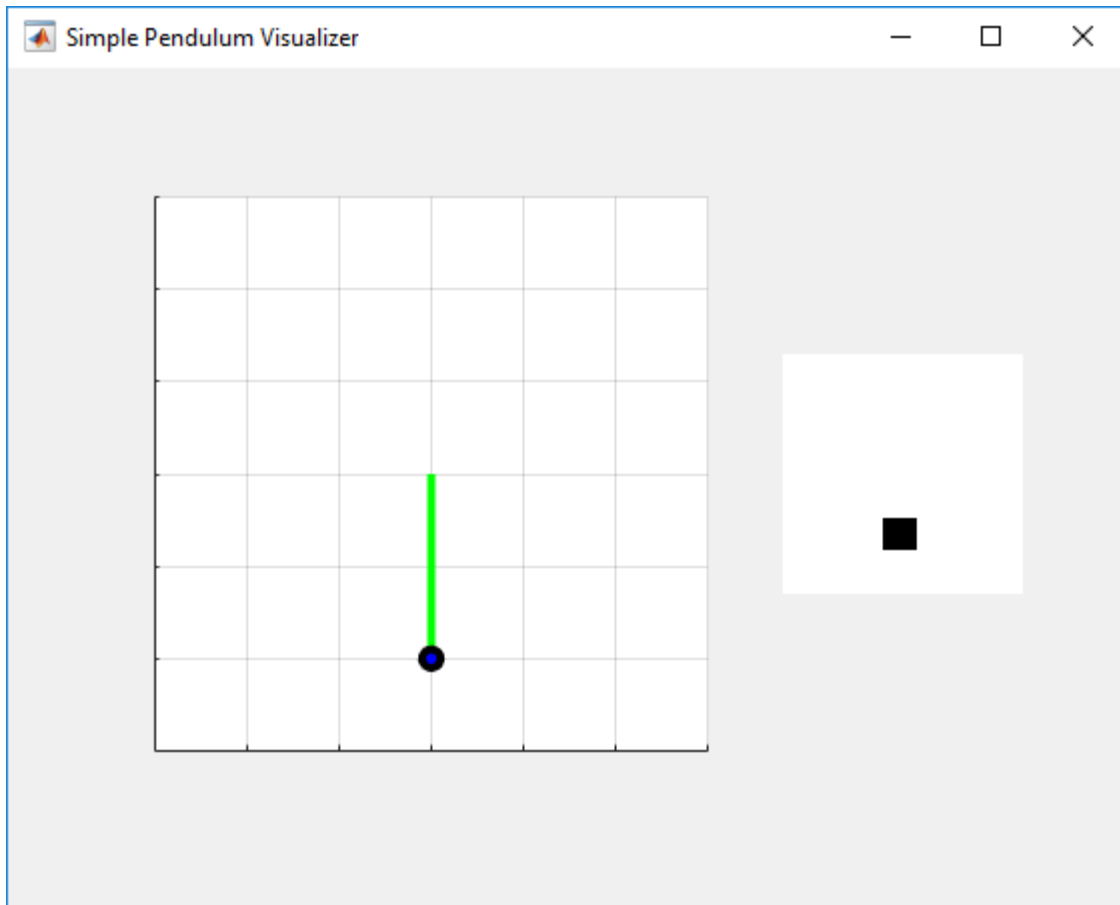
For this environment:

- The upward balanced pendulum position is `0` radians, and the downward hanging position is `pi` radians.
- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are the simplified grayscale image of the pendulum and the pendulum angle derivative.

- The reward r_t, provided at every time step, is

$$r_t = -\left(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001u_{t-1}^2\right)$$

Here:

- θ_t is the angle of displacement from the upright position.
- $\dot{\theta}_t$ is the derivative of the displacement angle.
- u_{t-1} is the control effort from the previous time step.

For more information on this model, see "Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation" on page 5-65.

Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv('SimplePendulumWithImage-Discrete');
```

The interface has two observations. The first observation, named `"pendImage"`, is a 50-by-50 grayscale image.

```
obsInfo = getObservationInfo(env);
obsInfo(1)
```

```
ans =
  rlNumericSpec with properties:

     LowerLimit: 0
     UpperLimit: 1
           Name: "pendImage"
    Description: [0x0 string]
      Dimension: [50 50]
       DataType: "double"
```

The second observation, named `"angularRate"`, is the angular velocity of the pendulum.

```
obsInfo(2)
```

```
ans =
  rlNumericSpec with properties:

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

The interface has a discrete action space where the agent can apply one of five possible torque values to the pendulum: -2, -1, 0, 1, or 2 N·m.

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

```
actInfo =
  rlFiniteSetSpec with properties:

        Elements: [-2 -1 0 1 2]
            Name: "torque"
     Description: [0x0 string]
       Dimension: [1 1]
        DataType: "double"
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Construct Critic Network Using Deep Network Designer

A DQN agent approximates the long-term reward, given observations and actions, using a critic value function representation. For this environment, the critic is a deep neural network with three inputs (two observations and one action), and one output. For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

You can construct the critic network interactively by using the Deep Network Designer app. To do so, you first create separate input paths for each observation and action. These paths learn lower level features from their respective inputs. You then create a common output path that combines the outputs from the input paths.

Create Image Observation Path

To create the image observation path, first drag an `ImageInputLayer` from the **Layer Library** pane to the canvas. Set the layer **InputSize** to `50,50,1` for the image observation, and set **Normalization** to `none`.
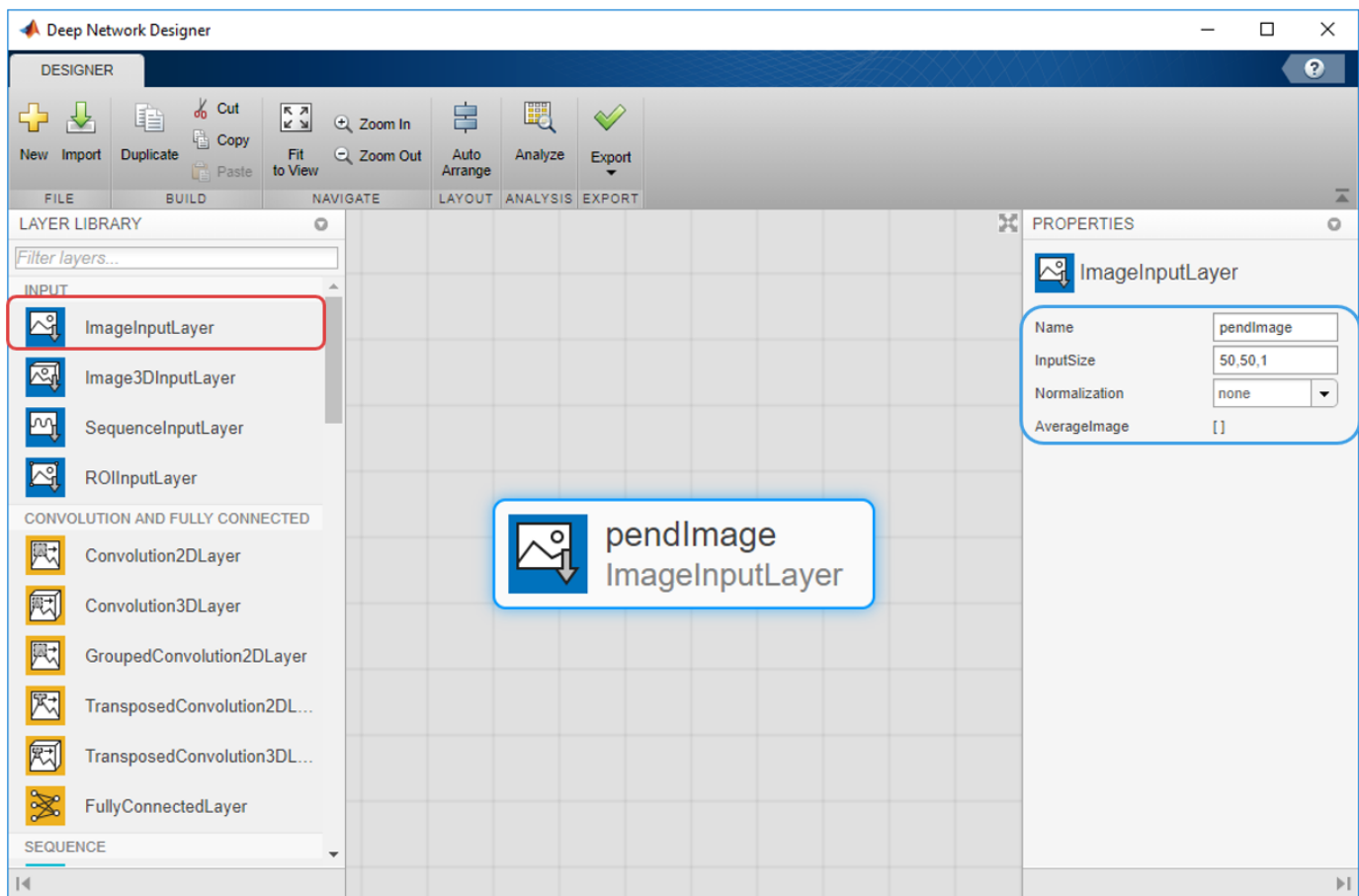
Second, drag a `Convolution2DLayer` to the canvas and connect the input of this layer to the output of the `ImageInputLayer`. Create a convolution layer with 2 filters (**NumFilters** property) that have a height and width of 10 (**FilterSize** property), and use a stride of 5 in the horizontal and vertical directions (**Stride** property).
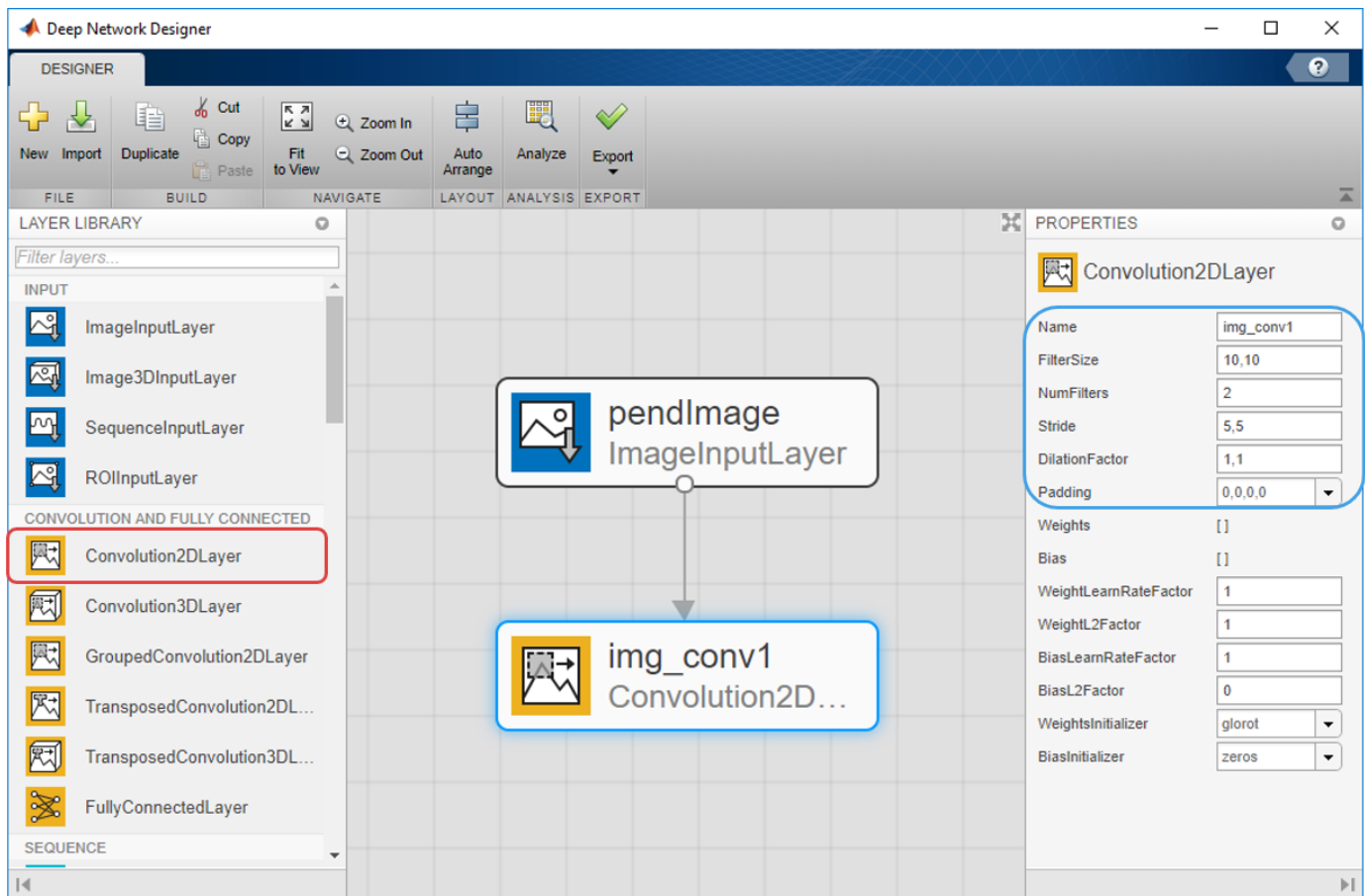
Finally, complete the image path network with two sets of `ReLULayer` and `FullyConnectedLayer` layers. The output sizes of the first and second `FullyConnectedLayer` layers are 400 and 300, respectively.
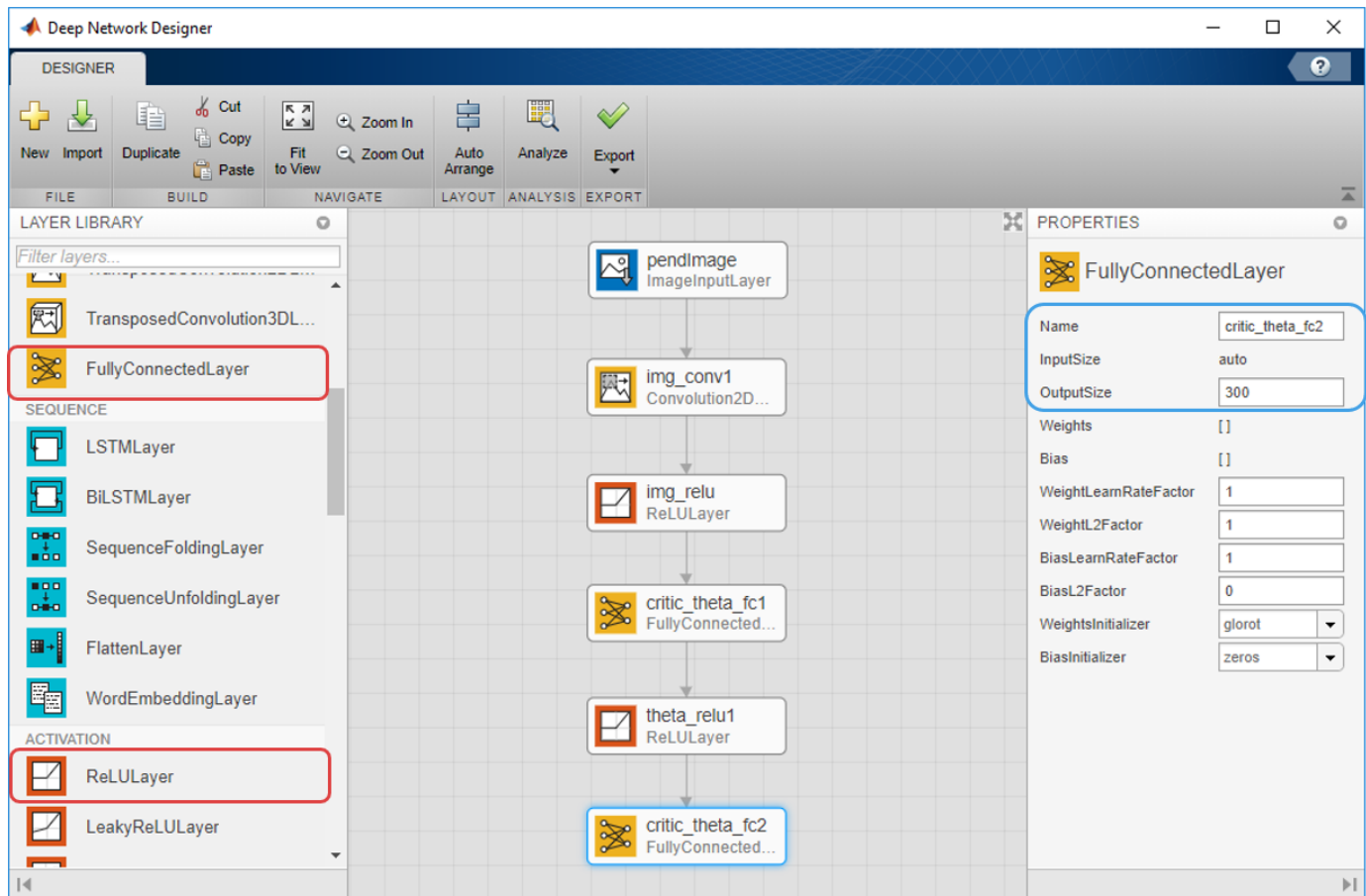
Create All Input Paths and Output Path

Construct the other input paths and output path in a similar manner. For this example, use the following options.

Angular velocity path (scalar input):

- `ImageInputLayer` — Set **InputSize** to 1,1 and **Normalization** to none.
- `FullyConnectedLayer` — Set **OutputSize** to 400.
- `ReLULayer`
- `FullyConnectedLayer` — Set **OutputSize** to 300.

Action path (scalar input):

- `ImageInputLayer` — Set **InputSize** to 1,1 and **Normalization** to none.
- `FullyConnectedLayer` — Set **OutputSize** to 300.

Output path:

- `AdditionLayer` — Connect the output of all input paths to the input of this layer.
- `ReLULayer`
- `FullyConnectedLayer` — Set **OutputSize** to 1 for the scalar value function.

Export Network from Deep Network Designer

To export the network to the MATLAB workspace, in **Deep Network Designer**, click **Export**. **Deep Network Designer** exports the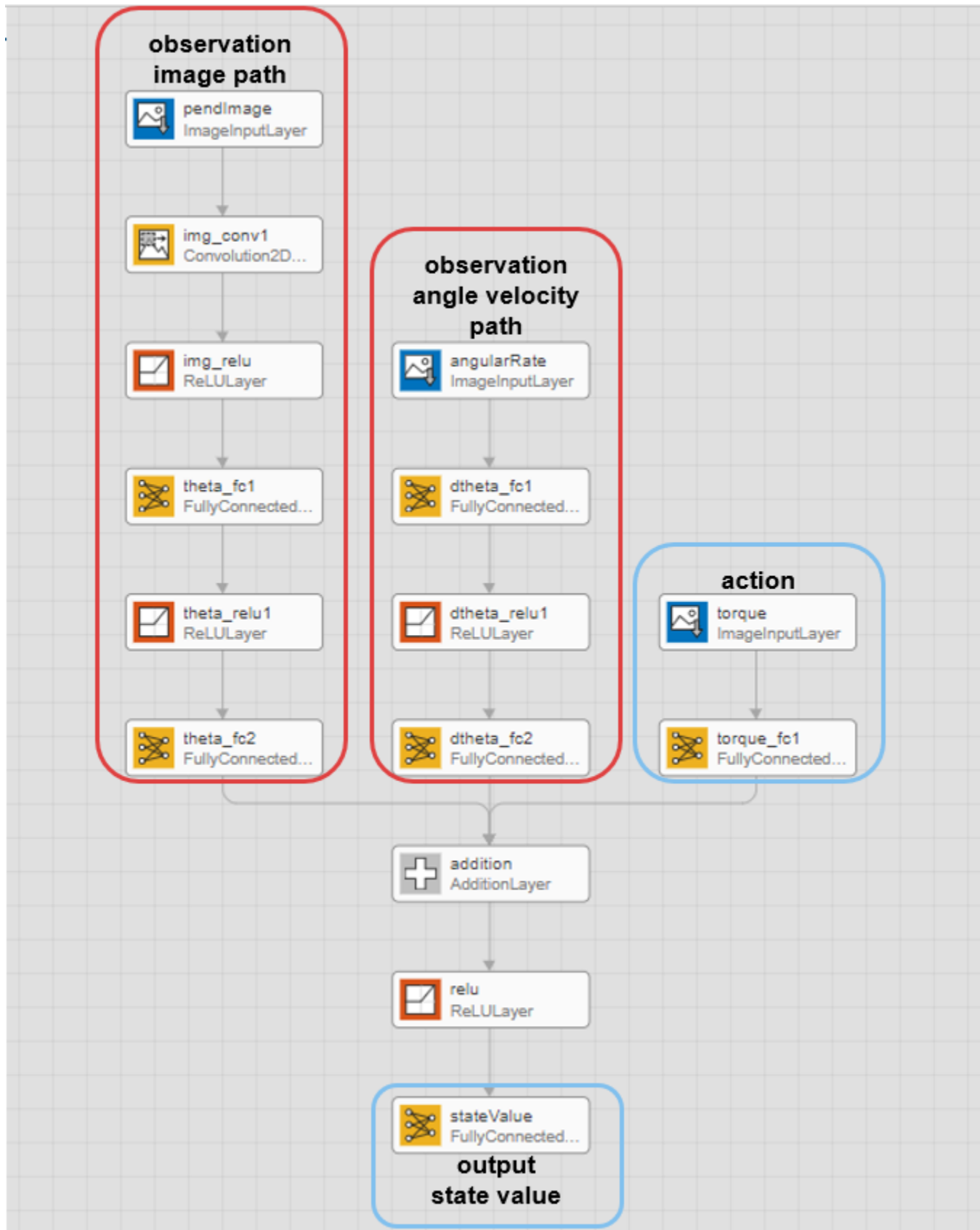 network as a new variable containing the network layers. You can create the critic representation using this layer network variable.

Alternatively, to generate equivalent MATLAB code for the network, click **Export > Generate Code**.
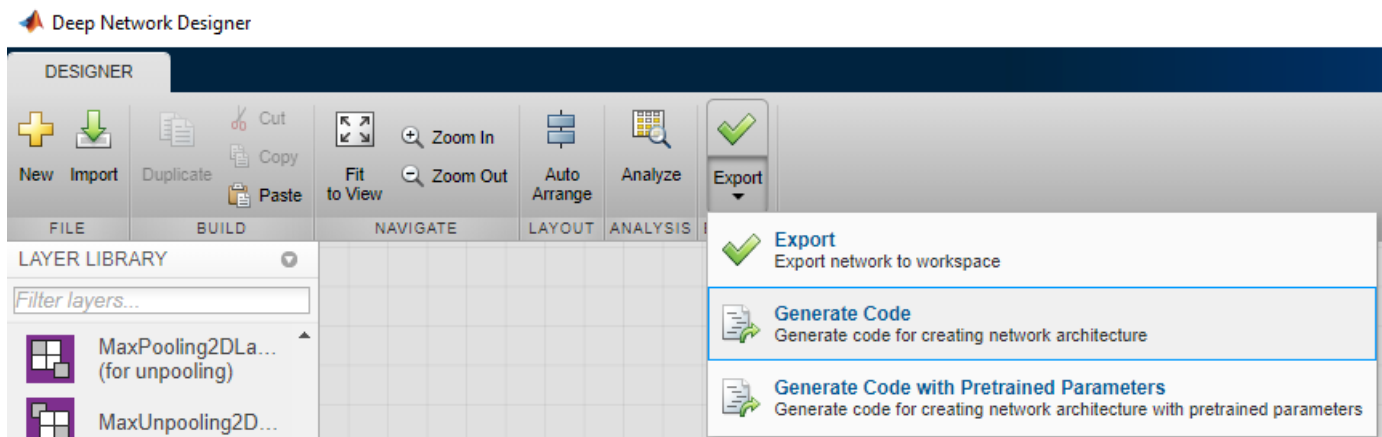
The generated code is as follows.

```
lgraph = layerGraph();
layers = [
    imageInputLayer([1 1 1],"Name","torque","Normalization","none")
    fullyConnectedLayer(300,"Name","torque_fc1")];
lgraph = addLayers(lgraph,layers);
layers = [
    imageInputLayer([1 1 1],"Name","angularRate","Normalization","none")
    fullyConnectedLayer(400,"Name","dtheta_fc1")
    reluLayer("Name","dtheta_relu1")
    fullyConnectedLayer(300,"Name","dtheta_fc2")];
lgraph = addLayers(lgraph,layers);
layers = [
    imageInputLayer([50 50 1],"Name","pendImage","Normalization","none")
    convolution2dLayer([10 10],2,"Name","img_conv1","Stride",[5 5])
    reluLayer("Name","img_relu")
    fullyConnectedLayer(400,"Name","theta_fc1")
    reluLayer("Name","theta_relu1")
    fullyConnectedLayer(300,"Name","theta_fc2")];
lgraph = addLayers(lgraph,layers);
layers = [
    additionLayer(3,"Name","addition")
    reluLayer("Name","relu")
    fullyConnectedLayer(1,"Name","stateValue")];
lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,"torque_fc1","addition/in3");
lgraph = connectLayers(lgraph,"theta_fc2","addition/in1");
lgraph = connectLayers(lgraph,"dtheta_fc2","addition/in2");
```

View the critic network configuration.

```
figure
plot(lgraph)
```
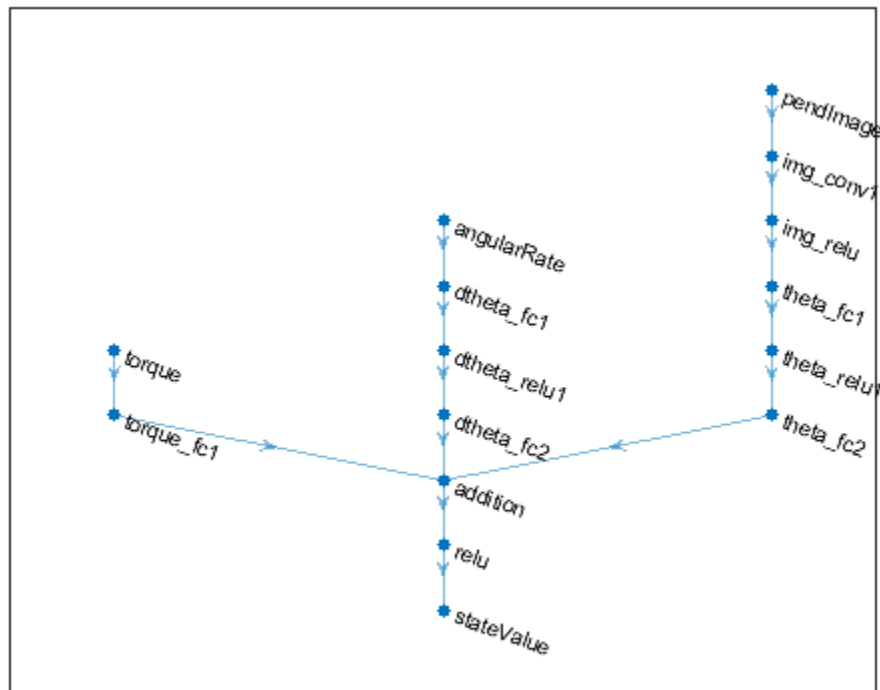
Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOpts = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Create the critic representation using the specified deep neural network `lgraph` and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(lgraph,obsInfo,actInfo,...
    'Observation',{'pendImage','angularRate'},'Action',{'torque'},criticOpts);
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions`.

```
agentOpts = rlDQNAgentOptions(...
    'UseDoubleDQN',false,...
    'TargetUpdateMethod',"smoothing",...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'SampleTime',env.Ts,...
    'MiniBatchSize',64);
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 1e-5;
```

Then, create the DQN agent using the specified critic representation and agent options. For more information, see `rlDQNAgent`.

```
agent = rlDQNAgent(critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

- Run each training for at most 5000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –1000 over the default window length of five consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',5000,...
    'MaxStepsPerEpisode',500,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-1000);
```

You can visualize the pendulum system during training or simulation by using the `plot` function.

```
plot(env)
```
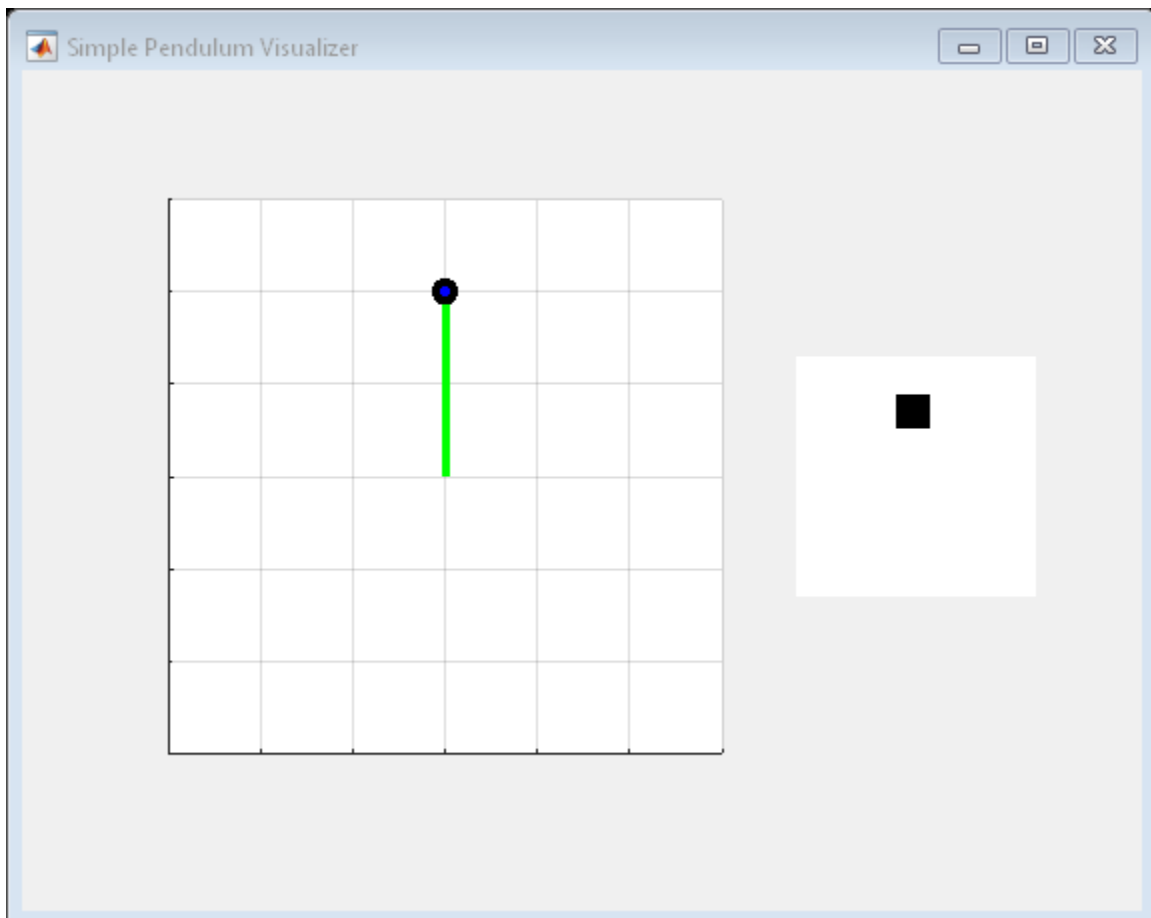
Train the agent using the `train` function. This is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load pretrained agent for the example.
    load('MATLABPendImageDQN.mat','agent');
end
```
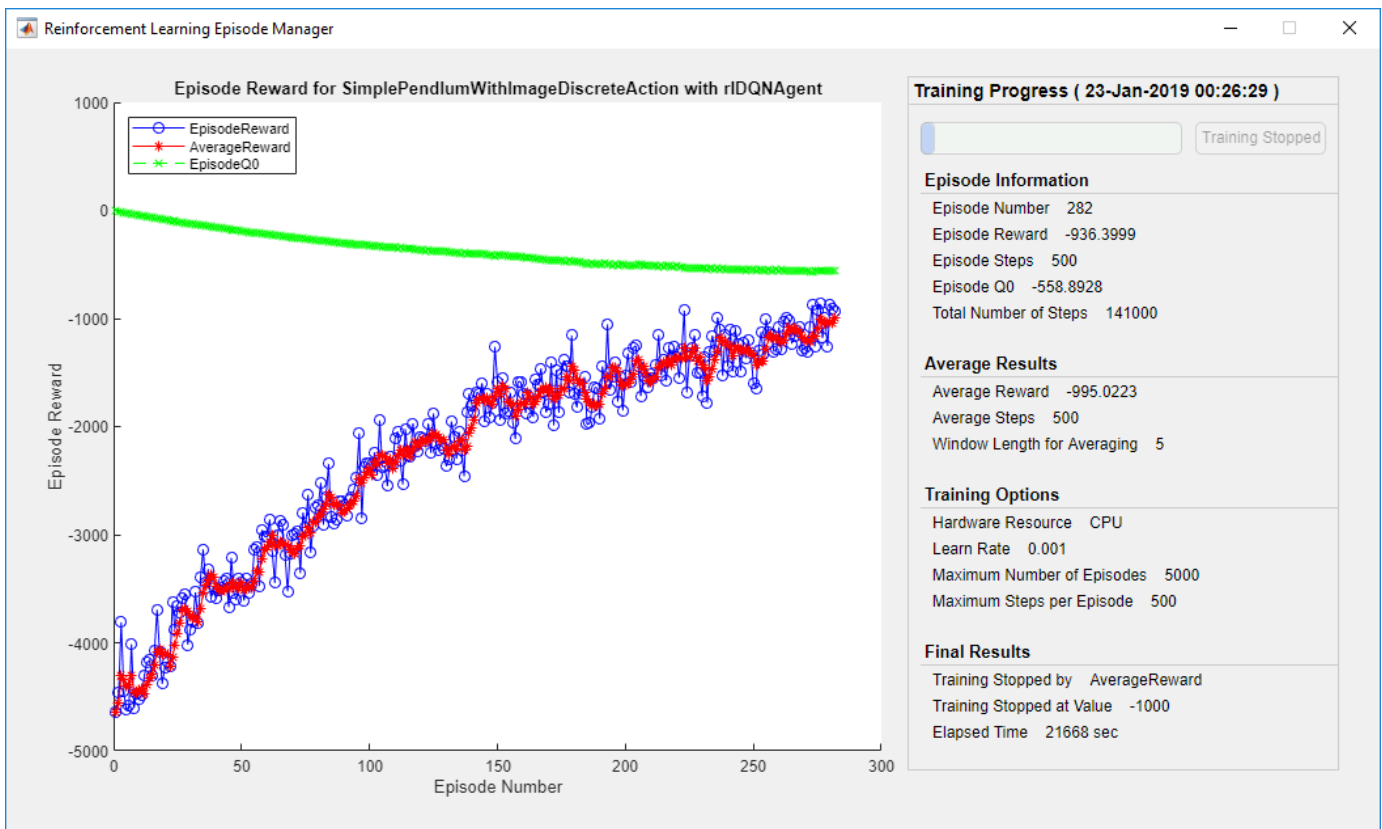

Simulate DQN Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```
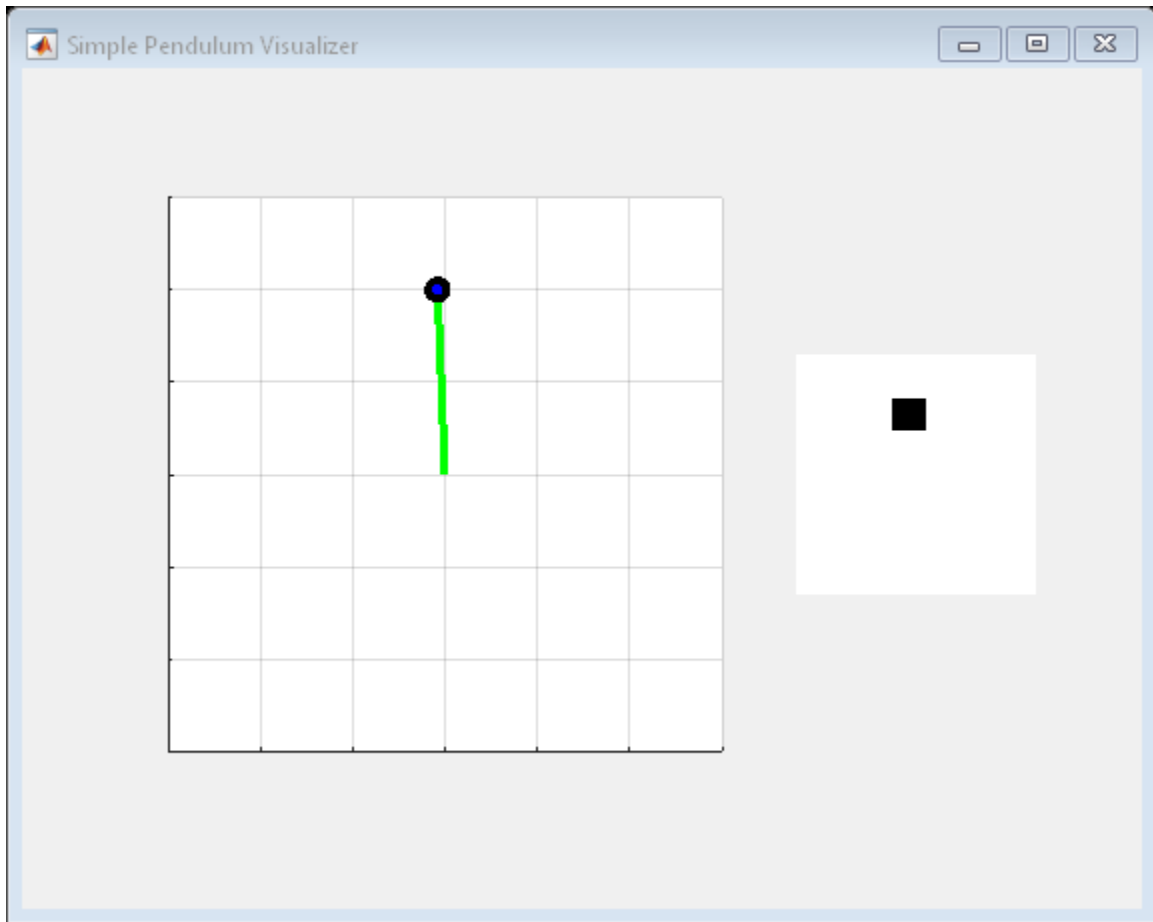
```
totalReward = sum(experience.Reward)
```

```
totalReward = -888.9802
```

See Also
Deep Network Designer | rlDQNAgent

More About

- "Train DQN Agent to Swing Up and Balance Pendulum" on page 5-37

Train AC Agent to Balance Cart-Pole System Using Parallel Computing

This example shows how to train an actor-critic (AC) agent to balance a cart-pole system modeled in MATLAB® by using asynchronous parallel training. For an example that shows how to train the agent without using parallel training, see "Train AC Agent to Balance Cart-Pole System" on page 5-19.

Actor Parallel Training

When you use parallel computing with AC agents, each worker generates experiences from its copy of the agent and the environment. After every **N** steps, the worker computes gradients from the experiences and sends the computed gradients back to the host agent. The host agent updates its parameters as follows.

- For asynchronous training, the host agent applies the received gradients without waiting for all workers to send gradients, and sends the updated parameters back to the worker that provided the gradients. Then, the worker continues to generate experiences from its environment using the updated parameters.
- For synchronous training, the host agent waits to receive gradients from all of the workers and updates its parameters using these gradients. The host then sends updated parameters to all the workers at the same time. Then, all workers continue to generate experiences using the updated parameters.

Create Cart-Pole MATLAB Environment Interface

Create a predefined environment interface for the cart-pole system. For more information on this environment, see "Load Predefined Control System Environments" on page 2-15.

```
env = rlPredefinedEnv("CartPole-Discrete");
env.PenaltyForFalling = -10;
```

Obtain the observation and action information from the environment interface.

```
obsInfo = getObservationInfo(env);
numObservations = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create AC Agent

An AC agent approximates the long-term reward, given observations and actions, using a critic value function representation. To create the critic, first create a deep neural network with one input (the observation) and one output (the state value). The input size of the critic network is 4 since the environment provides 4 observations. For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
criticNetwork = [
    featureInputLayer(4,'Normalization','none','Name','state')
    fullyConnectedLayer(32,'Name','CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(1, 'Name', 'CriticFC')];
```

```
criticOpts = rlRepresentationOptions('LearnRate',1e-2,'GradientThreshold',1);

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

An AC agent decides which action to take, given observations, using an actor representation. To create the actor, create a deep neural network with one input (the observation) and one output (the action). The output size of the actor network is 2 since the agent can apply 2 force values to the environment, –10 and 10.

```
actorNetwork = [
    featureInputLayer(4,'Normalization','none','Name','state')
    fullyConnectedLayer(32, 'Name','ActorStateFC1')
    reluLayer('Name','ActorRelu1')
    fullyConnectedLayer(2,'Name','action')];

actorOpts = rlRepresentationOptions('LearnRate',1e-2,'GradientThreshold',1);

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

To create the AC agent, first specify the AC agent options using `rlACAgentOptions`.

```
agentOpts = rlACAgentOptions(...
    'NumStepsToLookAhead',32,...
    'EntropyLossWeight',0.01,...
    'DiscountFactor',0.99);
```

Then create the agent using the specified actor representation and agent options. For more information, see `rlACAgent`.

```
agent = rlACAgent(actor,critic,agentOpts);
```

Parallel Training Options

To train the agent, first specify the training options. For this example, use the following options.

- Run each training for at most `1000` episodes, with each episode lasting at most `500` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option).
- Stop training when the agent receives an average cumulative reward greater than `500` over `10` consecutive episodes. At this point, the agent can balance the pendulum in the upright position.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',1000,...
    'MaxStepsPerEpisode', 500,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',500,...
    'ScoreAveragingWindowLength',10);
```

You can visualize the cart-pole system can during training or simulation using the `plot` function.

```
plot(env)
```
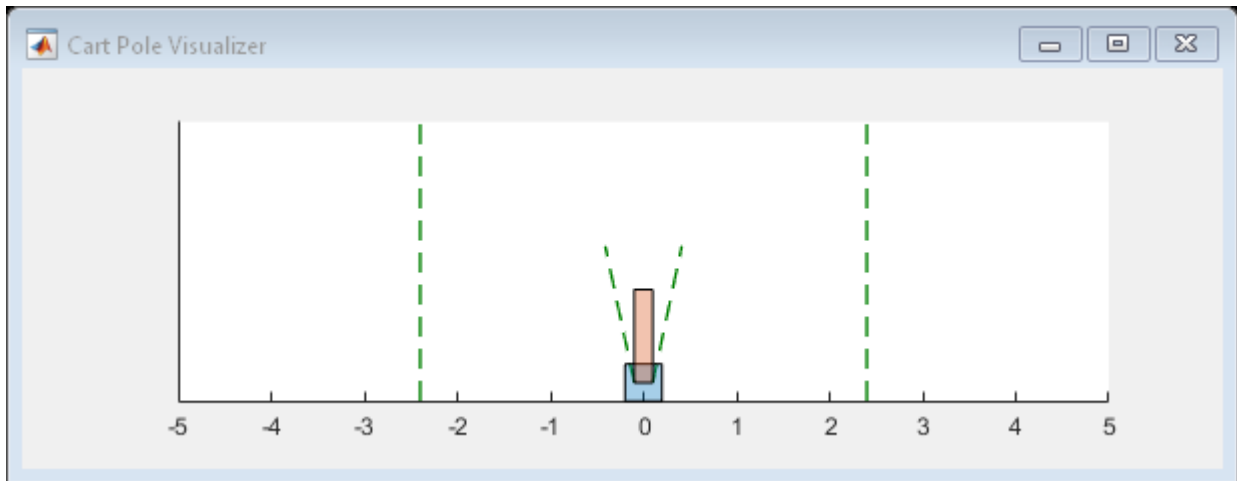
To train the agent using parallel computing, specify the following training options.

- Set the `UseParallel` option to `True`.
- Train the agent in parallel asynchronously by setting the `ParallelizationOptions.Mode` option to `"async"`.
- After every 32 steps, each worker computes gradients from experiences and send them to the host.
- The AC agent requires workers to send `"gradients"` to the host.
- The AC agent requires `'StepsUntilDataIsSent'` to be equal to `agentOptions.NumStepsToLookAhead`.

```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = "async";
trainOpts.ParallelizationOptions.DataToSendFromWorkers = "gradients";
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 32;
```

For more information, see `rlTrainingOptions`.

Train Agent

Train the agent using the `train` function. Training the agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`. Due to randomness in the asynchronous parallel training, you can expect different training results from the following training plot. The plot shows the result of training with six workers.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('MATLABCartpoleParAC.mat','agent');
end
```
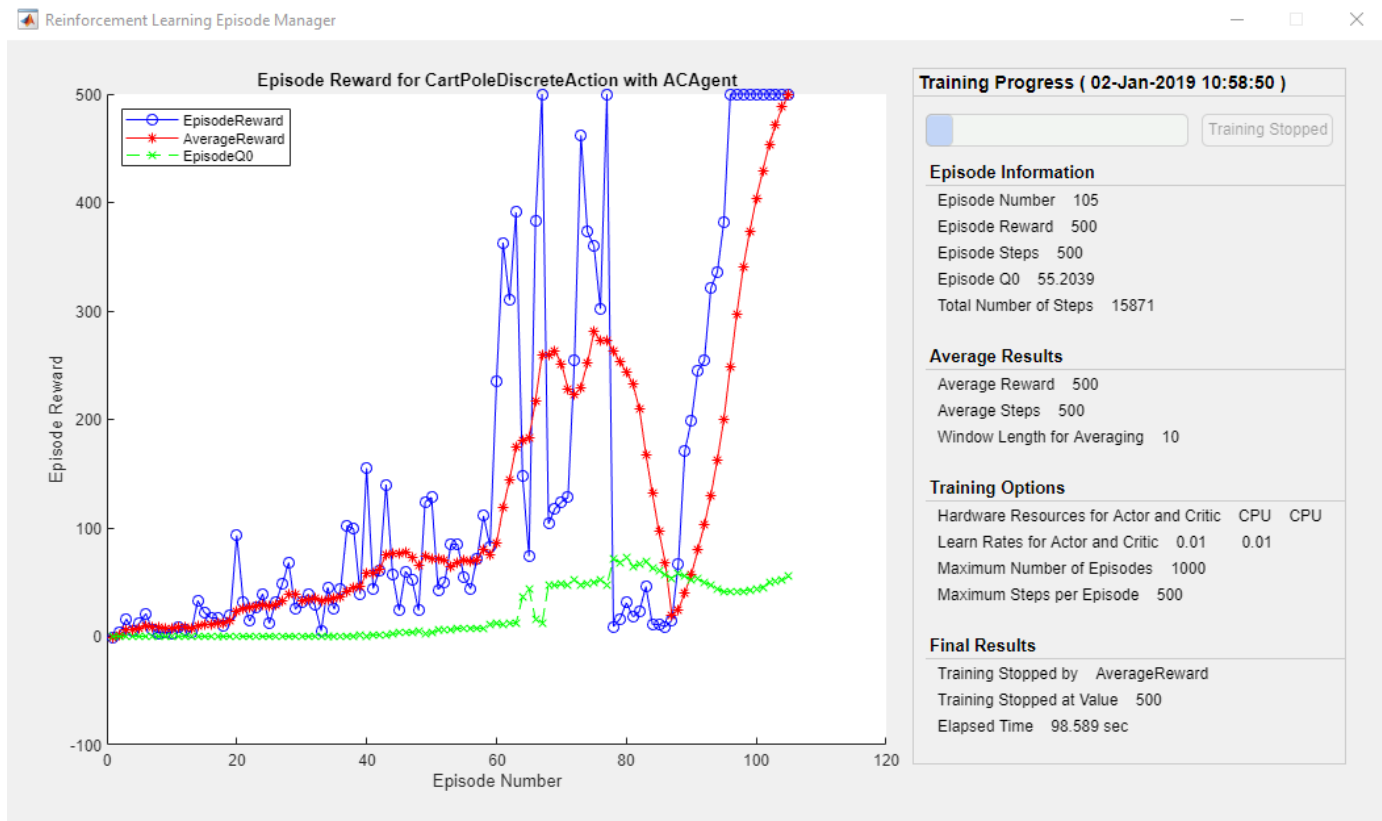
Simulate AC Agent

You can visualize the cart-pole system with the plot function during simulation.

```
plot(env)
```

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```
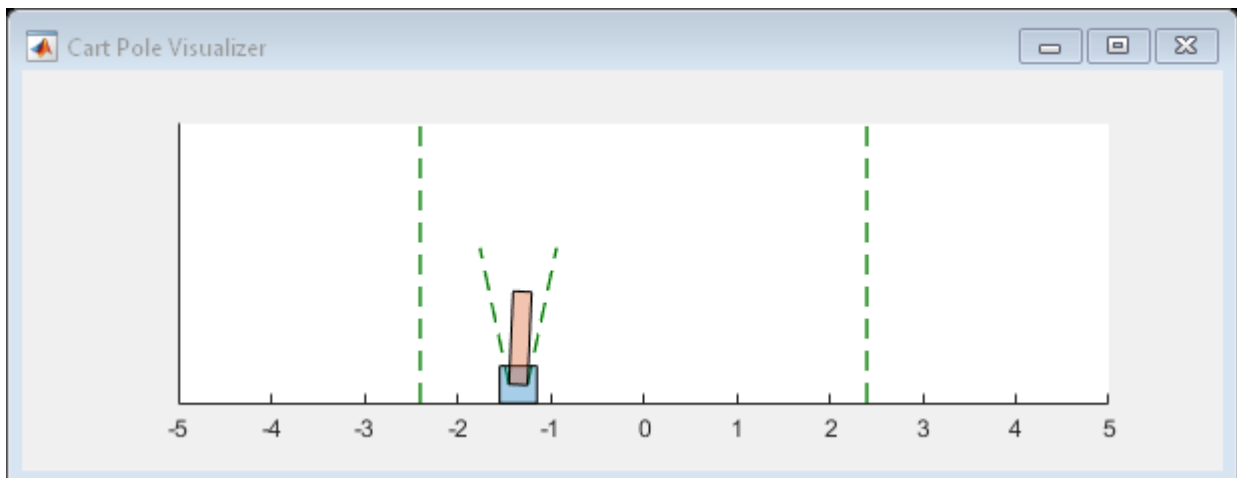
```
totalReward = sum(experience.Reward)
```

```
totalReward = 500
```

References

[1] Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 'Asynchronous Methods for Deep Reinforcement Learning'. *ArXiv:1602.01783 [Cs]*, 16 June 2016. https://arxiv.org/abs/1602.01783.

See Also

`rlTrainingOptions` | `train`

More About

- "Train Reinforcement Learning Agents" on page 5-2

Train DDPG Agent to Control Flying Robot

This example shows how to train a deep deterministic policy gradient (DDPG) agent to generate trajectories for a flying robot modeled in Simulink®. For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17.

Flying Robot Model

The reinforcement learning environment for this example is a flying robot with its initial condition randomized around a ring of radius 15 m. The orientation of the robot is also randomized. The robot has two thrusters mounted on the side of the body that are used to propel and steer the robot. The training goal is to drive the robot from its initial condition to the origin facing east.

Open the model.

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

Set the initial model state variables.

```
theta0 = 0;
x0 = -15;
y0 = 0;
```

Define the sample time `Ts` and the simulation duration `Tf`.

```
Ts = 0.4;
Tf = 30;
```

For this model:

- The goal orientation is `0` rad (robot facing east).
- The thrust from each actuator is bounded from `-1` to `1` N
- The observations from the environment are the position, orientation (sine and cosine of orientation), velocity, and angular velocity of the robot.
- The reward r_t provided at every time step is

$$r_1 = 10\left(\left(x_t^2 + y_t^2 + \theta_t^2\right) < 0.5\right)$$

$$r_2 = -100(|x_t| \geq 20 \; || \; |y_t| \geq 20)$$

$$r_3 = -\left(0.2(R_{t-1} + L_{t-1})^2 + 0.3(R_{t-1} - L_{t-1})^2 + 0.03x_t^2 + 0.03y_t^2 + 0.02\theta_t^2\right)$$

$$r_t = r_1 + r_2 + r_3$$

where:

- x_t is the position of the robot along the x-axis.
- y_t is the position of the robot along the y-axis.
- θ_t is the orientation of the robot.
- L_{t-1} is the control effort from the left thruster.

- R_{t-1} is the control effort from the right thruster.
- r_1 is the reward when the robot is close to the goal.
- r_2 is the penalty when the robot drives beyond 20 m in either the x or y direction. The simulation is terminated when $r_2 < 0$.
- r_3 is a QR penalty that penalizes distance from the goal and control effort.

Create Integrated Model

To train an agent for the `FlyingRobotEnv` model, use the `createIntegratedEnv` function to automatically generate an integrated model with the RL Agent block that is ready for training.

```
integratedMdl = 'IntegratedFlyingRobot';
[~,agentBlk,observationInfo,actionInfo] = createIntegratedEnv(mdl,integratedMdl);
```

Actions and Observations

Before creating the environment object, specify names for the observation and action specifications, and bound the thrust actions between `-1` and `1`.

The observation signals for this environment are observation $= \begin{bmatrix} x & y & \dot{x} & \dot{y} & \sin(\theta) & \cos(\theta) & \dot{\theta} \end{bmatrix}^T$.

```
numObs = prod(observationInfo.Dimension);
observationInfo.Name = 'observations';
```

The action signals for this environment are action $= \begin{bmatrix} T_R & T_L \end{bmatrix}^T$.

```
numAct = prod(actionInfo.Dimension);
actionInfo.LowerLimit = -ones(numAct,1);
actionInfo.UpperLimit =  ones(numAct,1);
actionInfo.Name = 'thrusts';
```

Create Environment Interface

Create an environment interface for the flying robot using the integrated model.

```
env = rlSimulinkEnv(integratedMdl,agentBlk,observationInfo,actionInfo);
```

Reset Function

Create a custom reset function that randomizes the initial position of the robot along a ring of radius 15 m and the initial orientation. For details on the reset function, see `flyingRobotResetFcn`.

```
env.ResetFcn = @(in) flyingRobotResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG agent

A DDPG agent approximates the long-term reward given observations and actions by using a critic value function representation. To create the critic, first create a deep neural network with two inputs (the observation and action) and one output. For more information on creating a neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
% Specify the number of outputs for the hidden layers.
hiddenLayerSize = 100;

observationPath = [
    featureInputLayer(numObs,'Normalization','none','Name','observation')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc2')
    additionLayer(2,'Name','add')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')];
actionPath = [
    featureInputLayer(numAct,'Normalization','none','Name','action')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc5')];

% Create the layer graph.
criticNetwork = layerGraph(observationPath);
criticNetwork = addLayers(criticNetwork,actionPath);

% Connect actionPath to observationPath.
criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');
```

Specify options for the critic using rlRepresentationOptions.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Create the critic representation using the specified neural network and options. You must also specify the action and observation specification for the critic. For more information, see rlQValueRepresentation.

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
```

A DDPG agent decides which action to take given observations by using an actor representation. To create the actor, first create a deep neural network with one input (the observation) and one output (the action).

Construct the actor similarly to the critic. For more information, see rlDeterministicActorRepresentation.

```
actorNetwork = [
    featureInputLayer(numObs,'Normalization','none','Name','observation')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numAct,'Name','fc4')
    tanhLayer('Name','tanh1')];

actorOptions = rlRepresentationOptions('LearnRate',1e-04,'GradientThreshold',1);

actor = rlDeterministicActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'tanh1'},actorOptions);
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6 ,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',256);
agentOptions.NoiseOptions.Variance = 1e-1;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;
```

Then, create the agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent`.

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

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run each training for at most `20000` episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than 415 over 10 consecutive episodes. At this point, the agent can drive the flying robot to the goal position.
- Save a copy of the agent for each episode where the cumulative reward is greater than 415.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 20000;
maxsteps = ceil(Tf/Ts);
trainingOptions = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'StopOnError',"on",...
    'Verbose',false,...
    'Plots',"training-progress",...
    'StopTrainingCriteria',"AverageReward",...
    'StopTrainingValue',415,...
    'ScoreAveragingWindowLength',10,...
    'SaveAgentCriteria',"EpisodeReward",...
    'SaveAgentValue',415);
```

Train the agent using the `train` function. Training is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
    % Load the pretrained agent for the example.
    load('FlyingRobotDDPG.mat','agent')
end
```
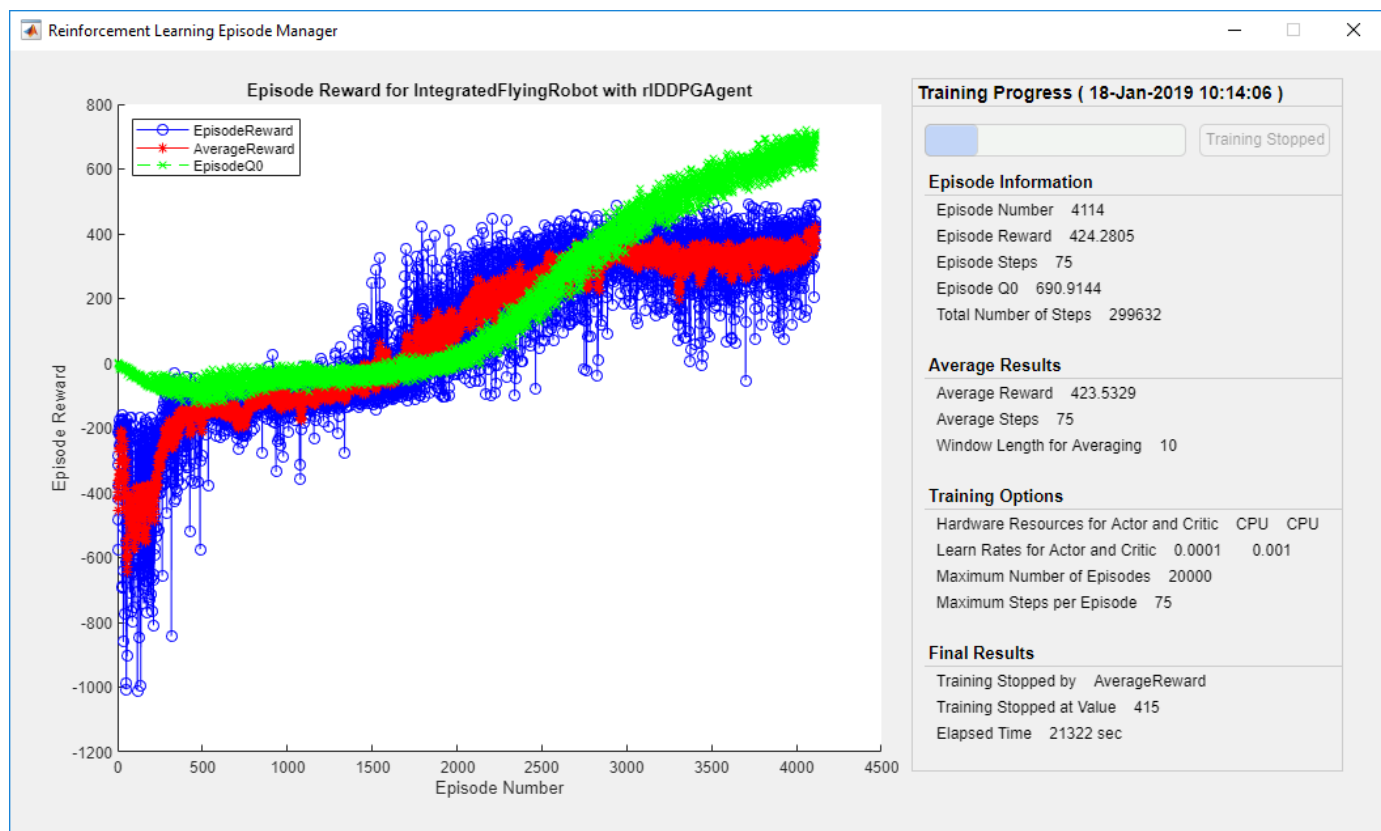
Simulate DDPG Agent

To validate the performance of the trained agent, simulate the agent within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',maxsteps);
experience = sim(env,agent,simOptions);
```
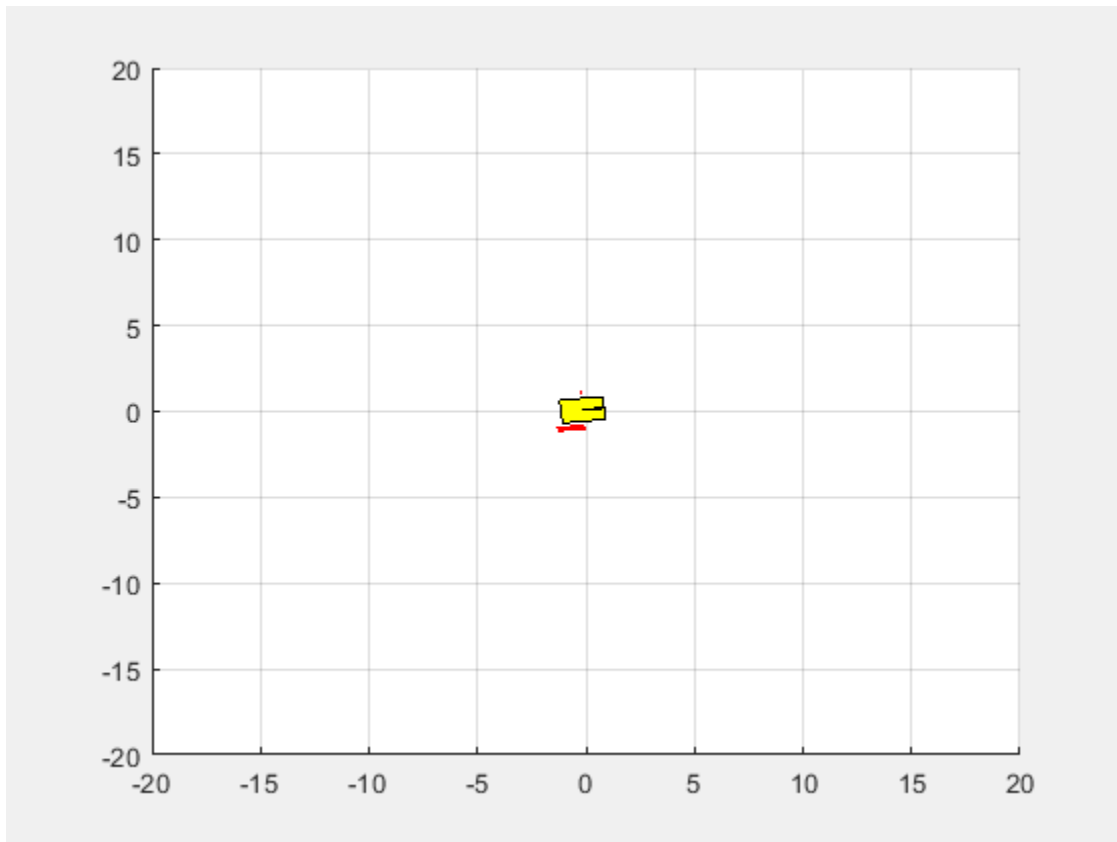
See Also
`rlDDPGAgent` | `train`

More About
- "Train Reinforcement Learning Agents" on page 5-2

Train PPO Agent to Land Rocket

This example shows how to train a proximal policy optimization (PPO) agent with a discrete action space to land a rocket on the ground. For more information on PPO agents, see "Proximal Policy Optimization Agents" on page 3-28.
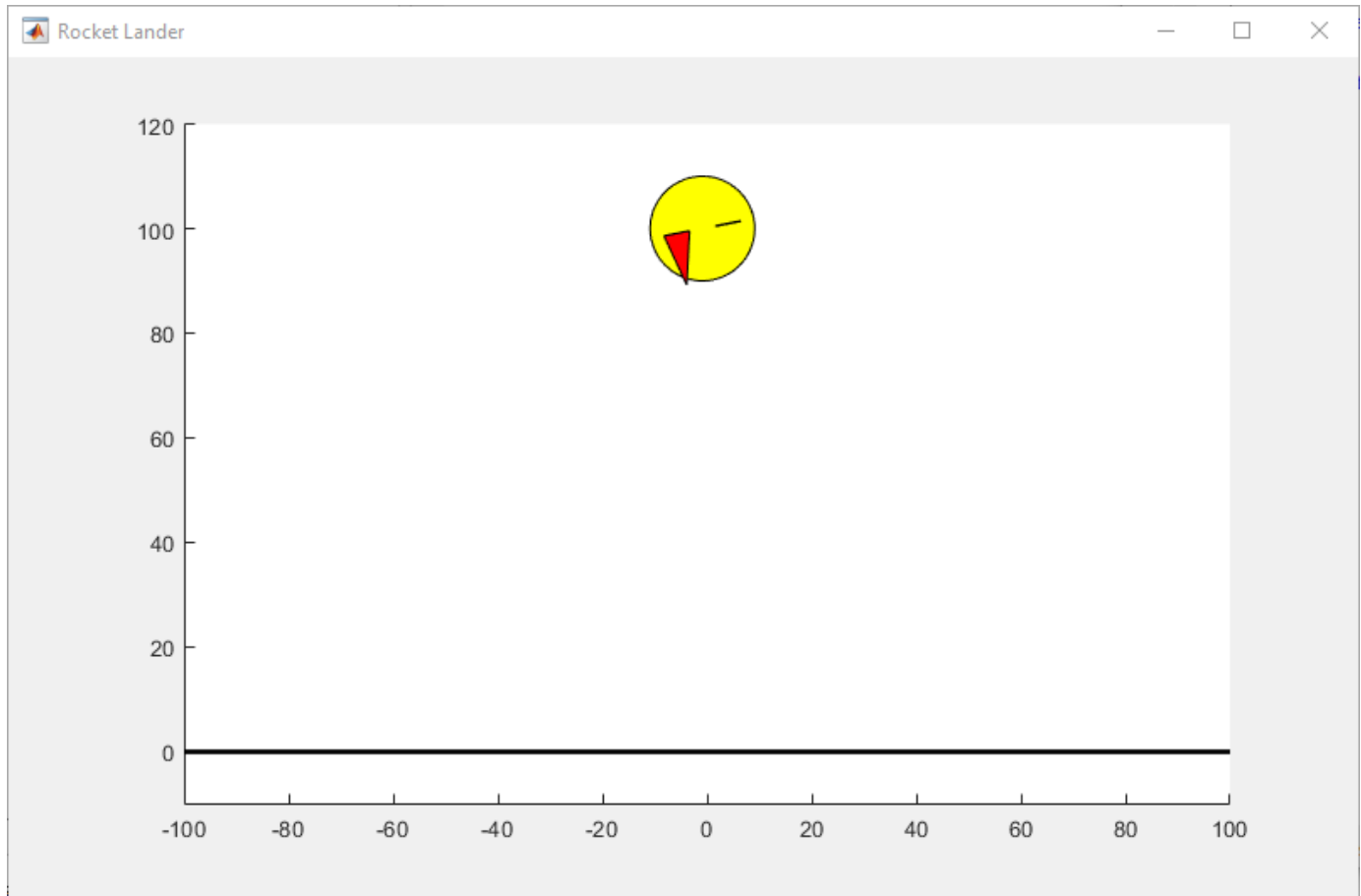
Environment

The environment in this example is a 3-DOF rocket represented by a circular disc with mass. The rocket has two thrusters for forward and rotational motion. Gravity acts vertically downwards, and there are no aerodynamic drag forces. The training goal is to make the robot land on the ground at a specified location.

For this environment:

- Motion of the rocket is bounded in X (horizontal axis) from -100 to 100 meters and Y (vertical axis) from 0 to 120 meters.
- The goal position is at (0,0) meters and the goal orientation is 0 radians.
- The maximum thrust applied by each thruster is 8.5 N.
- The sample time is 0.1 seconds.

- The observations from the environment are the rocket position (x, y), orientation (θ), velocity $(\dot{x}, \dot{y})$, angular velocity $(\dot{\theta})$, and a sensor reading that detects rough landing (-1), soft landing (1) or airborne (0) condition. The observations are normalized between -1 and 1.
- At the beginning of every episode, the rocket starts from a random initial x position and orientation. The altitude is always reset to 100 meters.
- The reward r_t provided at the time step t is as follows.

$$r_t = (s_t - s_{t-1}) - 0.1\theta_t^2 - 0.01\left(L_t^2 + R_t^2\right) + 500c$$

$$s_t = 1 - \left(\sqrt{\widehat{d}_t} + \frac{\sqrt{\widehat{v}_t}}{2}\right)$$

$$c = (y_t \le 0) \,\&\&\, (\dot{y}_t \ge -0.5 \,\&\&\, |\dot{x}_t| \le 0.5)$$

Here:

- $x_t, y_t, \dot{x}_t,$ and $\dot{y}_t$ are the positions and velocities of the rocket along the x and y axes.
- $\widehat{d}_t = \sqrt{x_t^2 + y_t^2}/d_{max}$ is the normalized distance of the rocket from the goal position.
- $\widehat{v}_t = \sqrt{\dot{x}_t^2 + \dot{y}_t^2}/v_{max}$ is the normalized speed of the rocket.
- d_{max} and v_{max} are the maximum distances and speeds.
- θ_t is the orientation with respect to the vertical axis.
- L_t and R_t are the action values for the left and right thrusters.
- c is a sparse reward for soft-landing with horizontal and vertical velocities less than 0.5 m/s.

Create MATLAB Environment

Create a MATLAB environment for the rocket lander using the `RocketLander` class.

```
env = RocketLander;
```

Obtain the observation and action specifications from the environment.

```
actionInfo = getActionInfo(env);
observationInfo = getObservationInfo(env);
numObs = observationInfo.Dimension(1);
numAct = numel(actionInfo.Elements);
```

Set a sample time for the environment

```
Ts = 0.1;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create PPO Agent

The PPO agent in this example operates on a discrete action space. At every time step, the agent selects one of the following discrete action pairs.

L, L – do nothing

L, M – fire right (med)

L, H – fire right (high)

M, L – fire left (med)

M, M – fire left (med) + right (med)

M, H – fire left (med) + right (high)

H, L – fire left (high)

H, M – fire left (high) + right (med)

H, H – fire left (high) + right (high)

Here, $L = 0.0, M = 0.5$ and $H = 1.0$ are normalized thrust values for each thruster.

To estimate the policy and value function, the agent maintains function approximators for the actor and critic, which are modeled using deep neural networks. The training can be sensitive to the initial network weights and biases, and results can vary with different sets of values. The network weights are randomly initialized to small values in this example.

Create the critic deep neural network with six inputs and one output. The output of the critic network is the discounted long-term reward for the input observations.

```
criticLayerSizes = [400 300];
actorLayerSizes = [400 300];

criticNetwork = [
        featureInputLayer(numObs,'Normalization','none','Name','observation')
        fullyConnectedLayer(criticLayerSizes(1),'Name','CriticFC1', ...
            'Weights',sqrt(2/numObs)*(rand(criticLayerSizes(1),numObs)-0.5), ...
            'Bias',1e-3*ones(criticLayerSizes(1),1))
        reluLayer('Name','CriticRelu1')
        fullyConnectedLayer(criticLayerSizes(2),'Name','CriticFC2', ...
            'Weights',sqrt(2/criticLayerSizes(1))*(rand(criticLayerSizes(2),criticLayerSizes(1))
            'Bias',1e-3*ones(criticLayerSizes(2),1))
        reluLayer('Name','CriticRelu2')
        fullyConnectedLayer(1,'Name','CriticOutput', ...
            'Weights',sqrt(2/criticLayerSizes(2))*(rand(1,criticLayerSizes(2))-0.5), ...
            'Bias',1e-3)];
```

Create the critic representation.

```
criticOpts = rlRepresentationOptions('LearnRate',1e-4);
critic = rlValueRepresentation(criticNetwork,observationInfo,'Observation',{'observation'},critic
```

Create the actor using a deep neural network with six inputs and two outputs. The outputs of the actor network are the probabilities of taking each possible action pair. Each action pair contains normalized action values for each thruster. The environment `step` function scales these values to determine the actual thrust values.

```
actorNetwork = [featureInputLayer(numObs,'Normalization','none','Name','observation')
        fullyConnectedLayer(actorLayerSizes(1),'Name','ActorFC1', ...
            'Weights',sqrt(2/numObs)*(rand(actorLayerSizes(1),numObs)-0.5), ...
            'Bias',1e-3*ones(actorLayerSizes(1),1))
        reluLayer('Name','ActorRelu1')
        fullyConnectedLayer(actorLayerSizes(2),'Name','ActorFC2', ...
            'Weights',sqrt(2/actorLayerSizes(1))*(rand(actorLayerSizes(2),actorLayerSizes(1))-0.5
```

```
        'Bias',1e-3*ones(actorLayerSizes(2),1))
    reluLayer('Name', 'ActorRelu2')
    fullyConnectedLayer(numAct,'Name','Action', ...
        'Weights',sqrt(2/actorLayerSizes(2))*(rand(numAct,actorLayerSizes(2))-0.5), ...
        'Bias',1e-3*ones(numAct,1))
    softmaxLayer('Name','actionProb')];
```

Create the actor using a stochastic actor representation.

```
actorOpts = rlRepresentationOptions('LearnRate',1e-4);
actor = rlStochasticActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},actorOpts);
```

Specify the agent hyperparameters using an `rlPPOAgentOptions` object.

```
agentOpts = rlPPOAgentOptions(...
                'ExperienceHorizon',600,...
                'ClipFactor',0.02,...
                'EntropyLossWeight',0.01,...
                'MiniBatchSize',128,...
                'NumEpoch',3,...
                'AdvantageEstimateMethod','gae',...
                'GAEFactor',0.95,...
                'SampleTime',Ts,...
                'DiscountFactor',0.997);
```

For these hyperparameters:

- The agent collects experiences until it reaches the experience horizon of 600 steps or episode termination and then trains from mini-batches of 128 experiences for 3 epochs.

- For improving training stability, use an objective function clip factor of 0.02.

- A discount factor value of 0.997 encourages long term rewards.

- Variance in critic output is reduced by using the Generalized Advantage Estimate method with a GAE factor of 0.95.

- The `EntropyLossWeight` term of 0.01 enhances exploration during training.

Create the PPO agent.

```
agent = rlPPOAgent(actor,critic,agentOpts);
```

Train Agent

To train the PPO agent, specify the following training options.

- Run the training for at most 20000 episodes, with each episode lasting at most 600 time steps.

- Stop the training when the average reward over 100 consecutive episodes is 430 or more.

- Save a copy of the agent for each episode where the episode reward is 700 or more.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',20000,...
    'MaxStepsPerEpisode',600,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',430,...
    'ScoreAveragingWindowLength',100,...
```

```
                'SaveAgentCriteria',"EpisodeReward",...
                'SaveAgentValue',700);
```

Train the agent using the `train` function. Due to the complexity of the environment, training process is computationally intensive and takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`.

```
doTraining = false;
if doTraining
    trainingStats = train(agent,env,trainOpts);
else
    load('rocketLanderAgent.mat');
end
```

An example training session is shown below. The actual results may vary because of randomness in the training process.
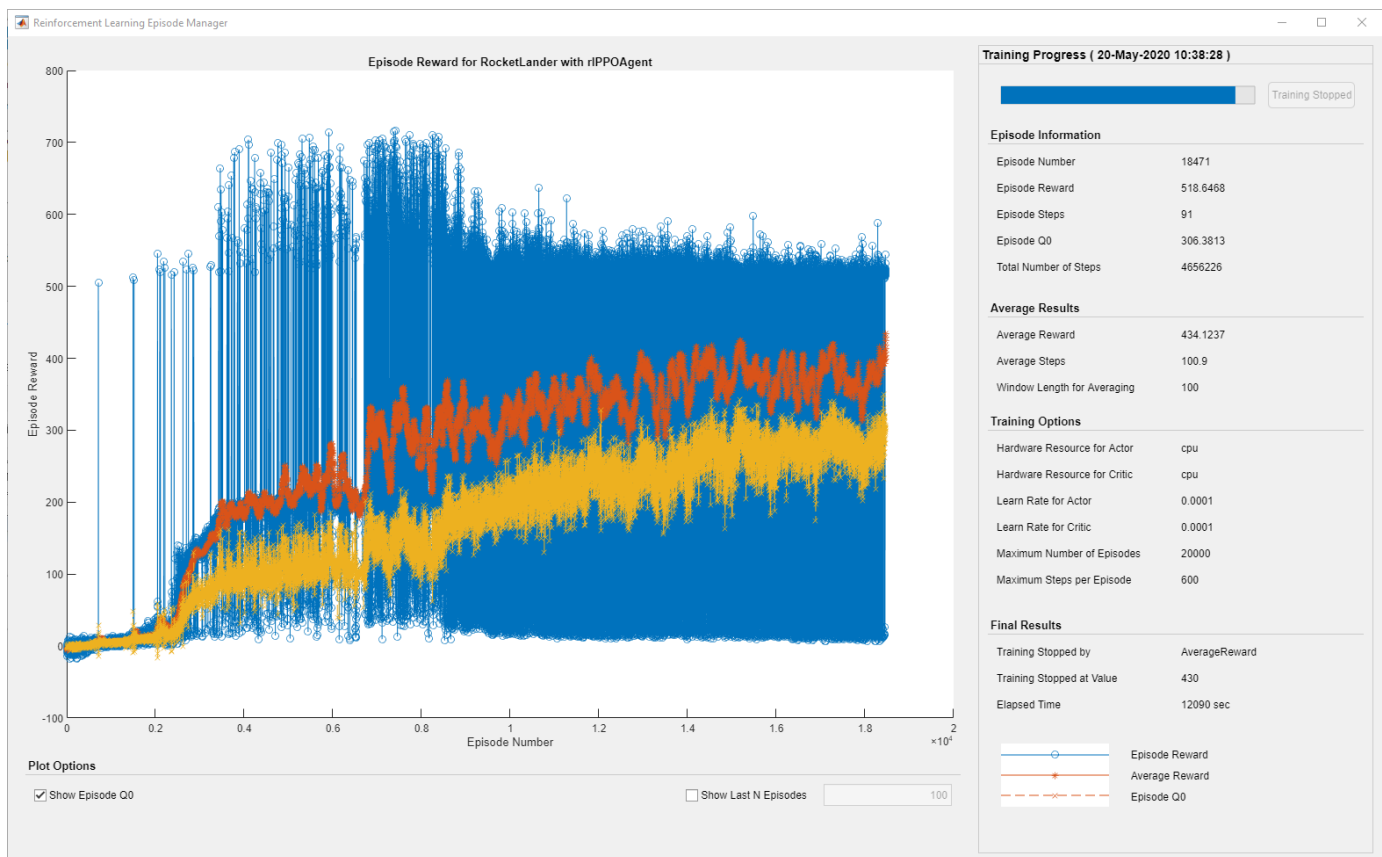
Simulate

Plot the rocket lander environment to visualize the simulation.

```
plot(env)
```

Simulate the trained agent within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',600);
simOptions.NumSimulations = 5;  % simulate the environment 5 times
experience = sim(env,agent,simOptions);
```
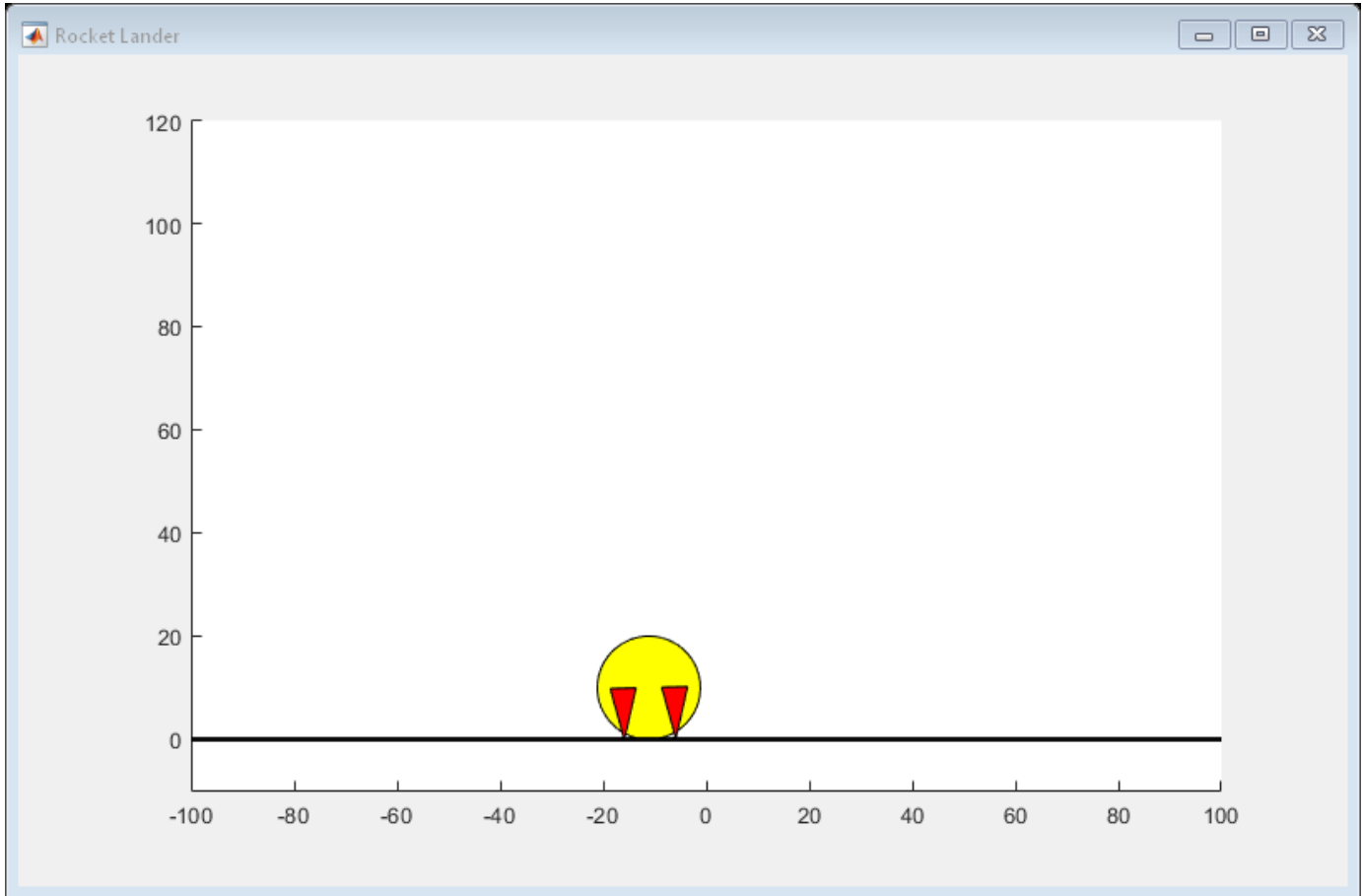

See Also
rlPPOAgent | train

More About
*   "Train Reinforcement Learning Agents" on page 5-2

Train Multiple Agents to Perform Collaborative Task

This example shows how to set up a multi-agent training session on a Simulink® environment. In the example, you train two agents to collaboratively perform the task of moving an object.
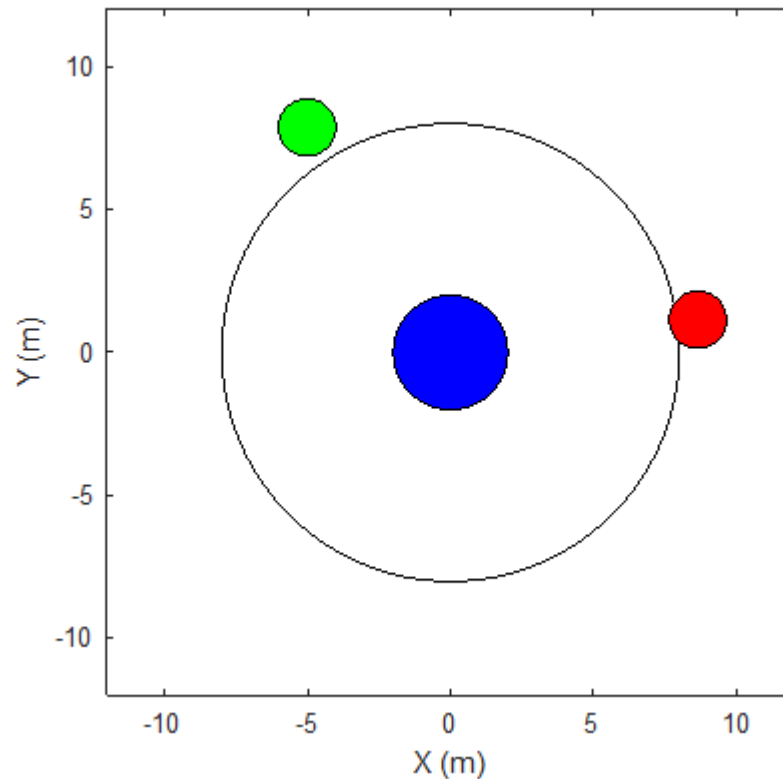
The environment in this example is a frictionless two dimensional surface containing elements represented by circles. A target object C is represented by the blue circle with a radius of 2 m, and robots A (red) and B (green) are represented by smaller circles with radii of 1 m each. The robots attempt to move object C outside a circular ring of a radius 8 m by applying forces through collision. All elements within the environment have mass and obey Newton's laws of motion. In addition, contact forces between the elements and the environment boundaries are modeled as spring and mass damper systems. The elements can move on the surface through the application of externally applied forces in the X and Y directions. There is no motion in the third dimension and the total energy of the system is conserved.

Create the set of parameters required for this example.

`rlCollaborativeTaskParams`

Open the Simulink model.

```
mdl = "rlCollaborativeTask";
open_system(mdl)
```
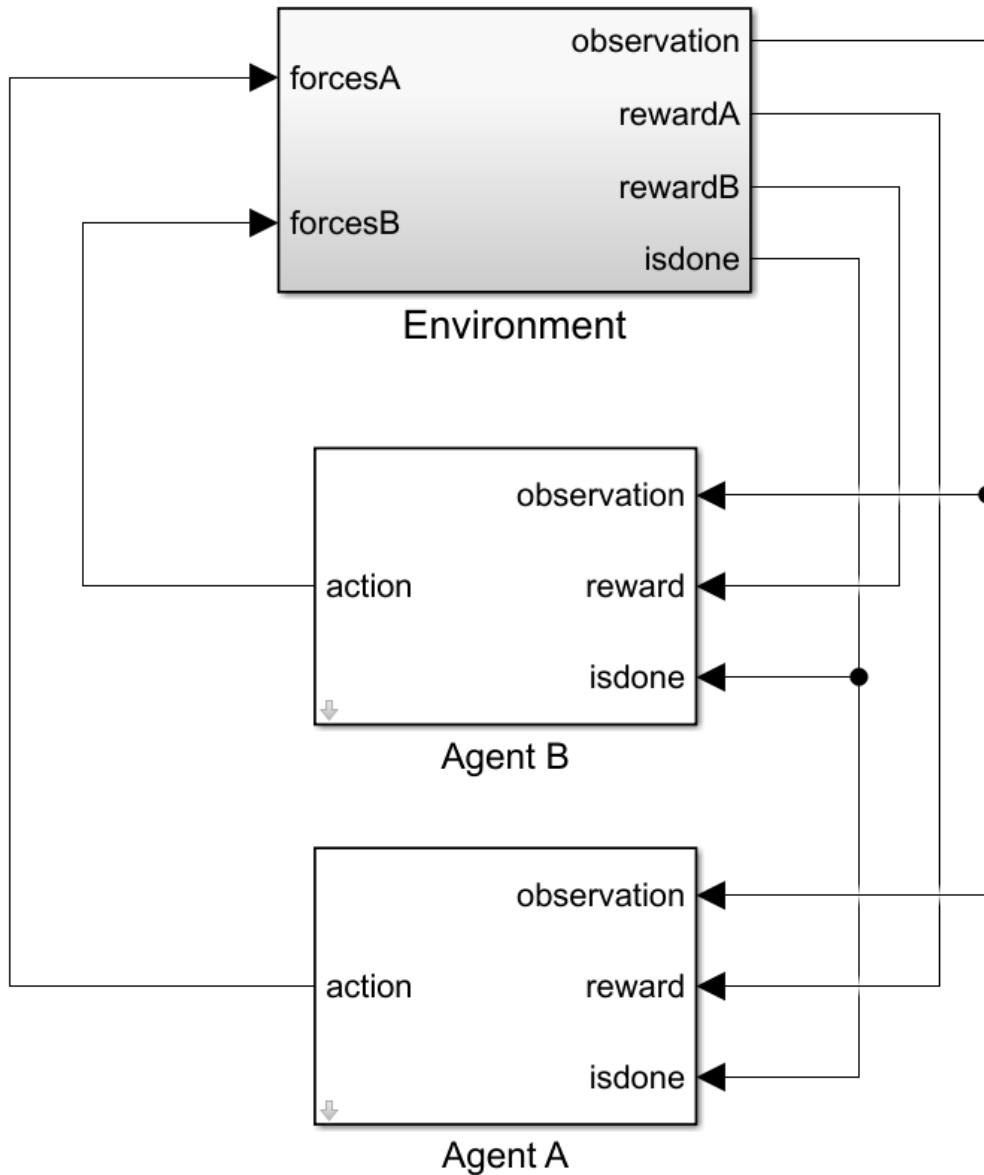
For this environment:

- The 2-dimensional space is bounded from –12 m to 12 m in both the X and Y directions.
- The contact spring stiffness and damping values are 100 N/m and 0.1 N/m/s, respectively.
- The agents share the same observations for positions, velocities of A, B, and C and the action values from the last time step.
- The simulation terminates when object C moves outside the circular ring.
- At each time step, the agents receive the following reward:

$$r_A = r_{\text{global}} + r_{\text{local}, A}$$

$$r_B = r_{\text{global}} + r_{\text{local}, B}$$

$$r_{\text{global}} = 0.001 d_c$$

$$r_{\text{local}, A} = -0.005 d_{\text{AC}} - 0.008 u_A^2$$

$$r_{\text{local}, B} = -0.005 d_{\text{BC}} - 0.008 u_B^2$$

Here:

- r_A and r_B are the rewards received by agents A and B, respectively.
- r_{global} is a team reward that is received by both agents as object C moves closer towards the boundary of the ring.
- $r_{\text{local}, A}$ and $r_{\text{local}, B}$ are local penalties received by agents A and B based on their distances from object C and the magnitude of the action from the last time step.
- d_C is the distance of object C from the center of the ring.
- d_{AC} and d_{BC} are the distances between agent A and object C and agent B and object C, respectively.
- u_A and u_B are the action values of agents A and B from the last time step.

This example uses proximal policy optimization (PPO) agents with discrete action spaces. To learn more about PPO agents, see "Proximal Policy Optimization Agents" on page 3-28. The agents apply external forces on the robots that result in motion. At every time step, the agents select the actions $u_{A,B} = [F_X, F_Y]$, where F_X, F_Y is one of the following pairs of externally applied forces.

$$F_X = -1.0\,N, \ F_Y = -1.0\,N$$

$$F_X = -1.0\,N, \ F_Y = 0$$

$$F_X = -1.0\,N, \ F_Y = 1.0\,N$$

$$F_X = 0, \quad\quad F_Y = -1.0\,N$$

$$F_X = 0, \quad\quad F_Y = 0$$

$$F_X = 0, \quad\quad F_Y = 1.0\,N$$

$$F_X = 1.0\,N, \quad F_Y = -1.0\,N$$

$$F_X = 1.0\,N, \quad F_Y = 0$$

$$F_X = 1.0\,N, \quad F_Y = 1.0\,N$$

Create Environment

To create a multi-agent environment, specify the block paths of the agents using a string array. Also, specify the observation and action specification objects using cell arrays. The order of the specification objects in the cell array must match the order specified in the block path array. When agents are available in the MATLAB workspace at the time of environment creation, the observation and action specification arrays are optional. For more information on creating multi-agent environments, see rlSimulinkEnv.

Create the I/O specifications for the environment. In this example, the agents are homogeneous and have the same I/O specifications.

```matlab
% Number of observations
numObs = 16;

% Number of actions
numAct = 2;

% Maximum value of externally applied force (N)
maxF = 1.0;

% I/O specifications for each agent
oinfo = rlNumericSpec([numObs,1]);
ainfo = rlFiniteSetSpec({
    [-maxF -maxF]
    [-maxF  0   ]
    [-maxF  maxF]
    [ 0    -maxF]
    [ 0     0   ]
    [ 0     maxF]
    [ maxF -maxF]
    [ maxF  0   ]
    [ maxF  maxF]});
oinfo.Name = 'observations';
ainfo.Name = 'forces';
```

Create the Simulink environment interface.

```matlab
blks = ["rlCollaborativeTask/Agent A", "rlCollaborativeTask/Agent B"];
obsInfos = {oinfo,oinfo};
actInfos = {ainfo,ainfo};
env = rlSimulinkEnv(mdl,blks,obsInfos,actInfos);
```

Specify a reset function for the environment. The reset function `resetRobots` ensures that the robots start from random initial positions at the beginning of each episode.

```matlab
env.ResetFcn = @(in) resetRobots(in,RA,RB,RC,boundaryR);
```

Create Agents

PPO agents rely on actor and critic representations to learn the optimal policy. In this example, the agents maintain neural network-based function approximators for the actor and critic.

Create the critic neural network and representation. The output of the critic network is the state value function $V(s)$ for state s.

```matlab
% Reset the random seed to improve reproducibility
rng(0)

% Critic networks
criticNetwork = [...
    featureInputLayer(oinfo.Dimension(1),'Normalization','none','Name','observation')
    fullyConnectedLayer(128,'Name','CriticFC1','WeightsInitializer','he')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(64,'Name','CriticFC2','WeightsInitializer','he')
    reluLayer('Name','CriticRelu2')
    fullyConnectedLayer(32,'Name','CriticFC3','WeightsInitializer','he')
```

```
    reluLayer('Name','CriticRelu3')
    fullyConnectedLayer(1,'Name','CriticOutput')];

% Critic representations
criticOpts = rlRepresentationOptions('LearnRate',1e-4);
criticA = rlValueRepresentation(criticNetwork,oinfo,'Observation',{'observation'},criticOpts);
criticB = rlValueRepresentation(criticNetwork,oinfo,'Observation',{'observation'},criticOpts);
```

The outputs of the actor network are the probabilities $\pi(a|s)$ of taking each possible action pair at a certain state s. Create the actor neural network and representation.

```
% Actor networks
actorNetwork = [...
    featureInputLayer(oinfo.Dimension(1),'Normalization','none','Name','observation')
    fullyConnectedLayer(128,'Name','ActorFC1','WeightsInitializer','he')
    reluLayer('Name','ActorRelu1')
    fullyConnectedLayer(64,'Name','ActorFC2','WeightsInitializer','he')
    reluLayer('Name','ActorRelu2')
    fullyConnectedLayer(32,'Name','ActorFC3','WeightsInitializer','he')
    reluLayer('Name','ActorRelu3')
    fullyConnectedLayer(numel(ainfo.Elements),'Name','Action')
    softmaxLayer('Name','SM')];

% Actor representations
actorOpts = rlRepresentationOptions('LearnRate',1e-4);
actorA = rlStochasticActorRepresentation(actorNetwork,oinfo,ainfo,...
    'Observation',{'observation'},actorOpts);
actorB = rlStochasticActorRepresentation(actorNetwork,oinfo,ainfo,...
    'Observation',{'observation'},actorOpts);
```

Create the agents. Both agents use the same options.

```
agentOptions = rlPPOAgentOptions(...
    'ExperienceHorizon',256,...
    'ClipFactor',0.125,...
    'EntropyLossWeight',0.001,...
    'MiniBatchSize',64,...
    'NumEpoch',3,...
    'AdvantageEstimateMethod','gae',...
    'GAEFactor',0.95,...
    'SampleTime',Ts,...
    'DiscountFactor',0.9995);
agentA = rlPPOAgent(actorA,criticA,agentOptions);
agentB = rlPPOAgent(actorB,criticB,agentOptions);
```

During training, agents collect experiences until either the experience horizon of 256 steps or the episode termination is reached, and then train from mini-batches of 64 experiences. This example uses an objective function clip factor of 0.125 to improve training stability and a discount factor of 0.9995 to encourage long-term rewards.

Train Agents

Specify the following training options to train the agents.

- Run the training for at most 1000 episodes, with each episode lasting at most 5000 time steps.
- Stop the training of an agent when its average reward over 100 consecutive episodes is –10 or more.

```
maxEpisodes = 1000;
maxSteps = 5e3;
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxEpisodes,...
    'MaxStepsPerEpisode',maxSteps,...
    'ScoreAveragingWindowLength',100,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-10);
```

To train multiple agents, specify an array of agents to the `train` function. The order of agents in the array must match the order of agent block paths specified during environment creation. Doing so ensures that the agent objects are linked to their appropriate I/O interfaces in the environment. Training these agents can take several hours to complete, depending on the available computational power.

The MAT file `rlCollaborativeTaskAgents` contains a set of pretrained agents. You can load the file and to view the performance of the agents. To train the agents yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    stats = train([agentA, agentB],env,trainOpts);
else
    load('rlCollaborativeTaskAgents.mat');
end
```

The following figure shows a snapshot of training progress. You can expect different results due to randomness in the training process.
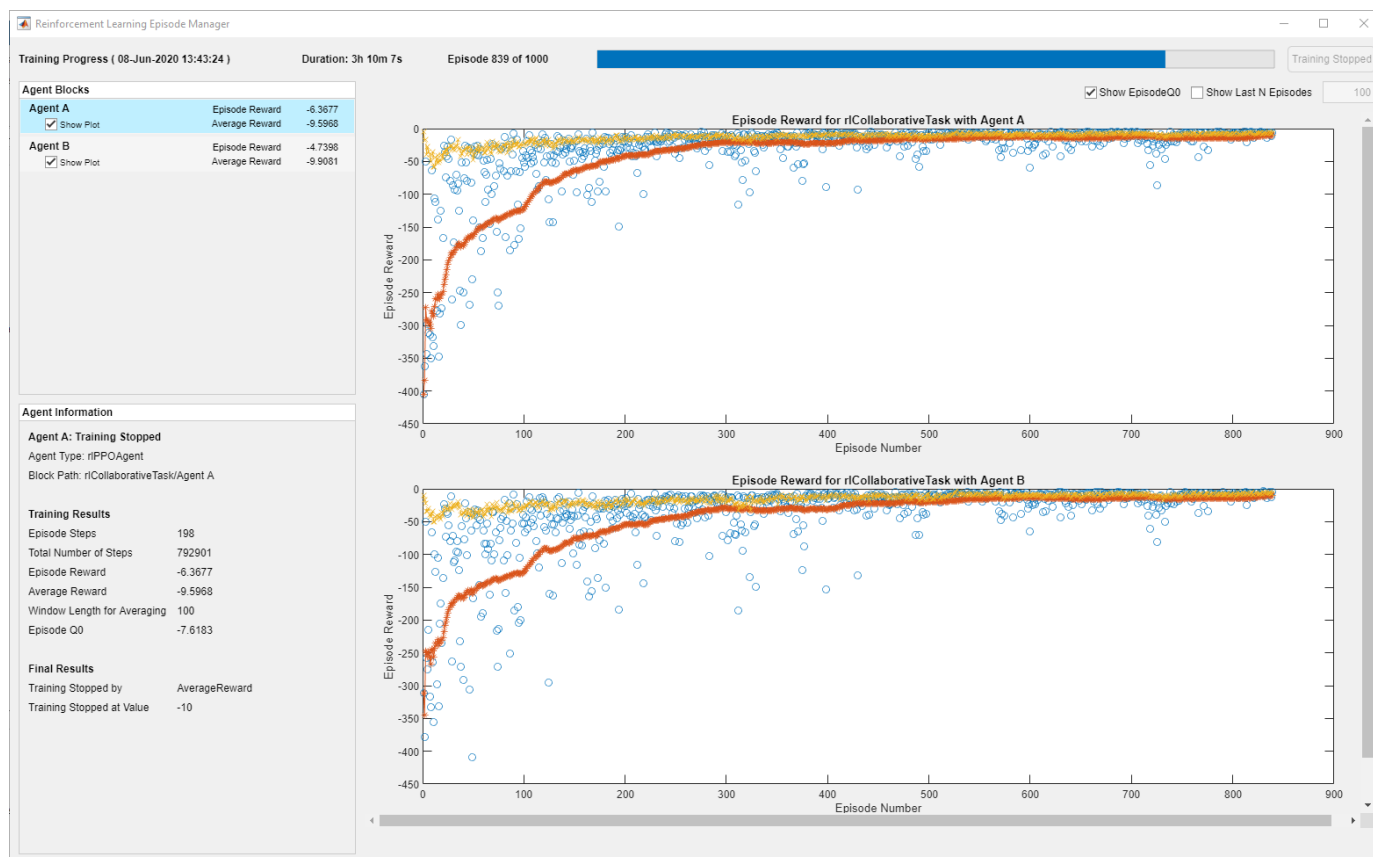
Simulate Agents

Simulate the trained agents within the environment.

```
simOptions = rlSimulationOptions('MaxSteps',maxSteps);
exp = sim(env,[agentA agentB],simOptions);
```
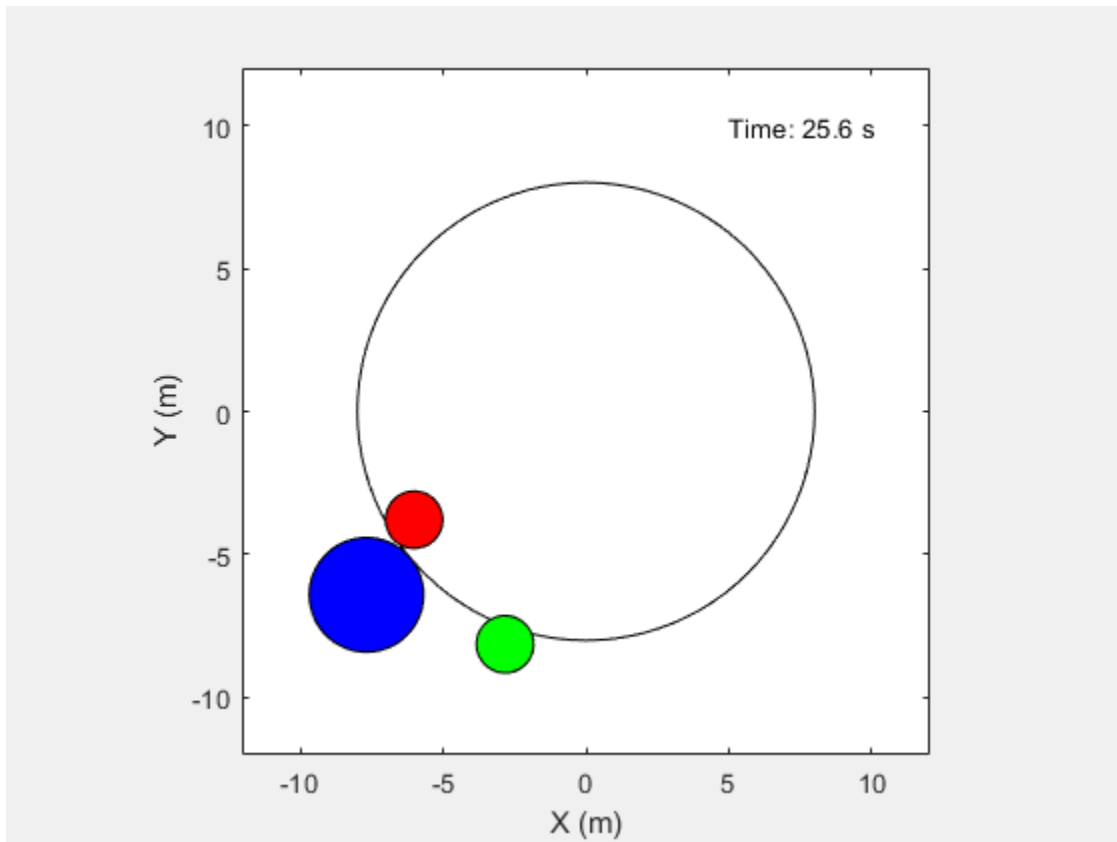
For more information on agent simulation, see `rlSimulationOptions` and `sim`.

See Also
`rlSimulinkEnv` | `train`

More About
*   "Train Reinforcement Learning Agents" on page 5-2

Train Multiple Agents for Area Coverage

This example demonstrates a multi-agent collaborative-competitive task in which you train three proximal policy optimization (PPO) agents to explore all areas within a grid-world environment.
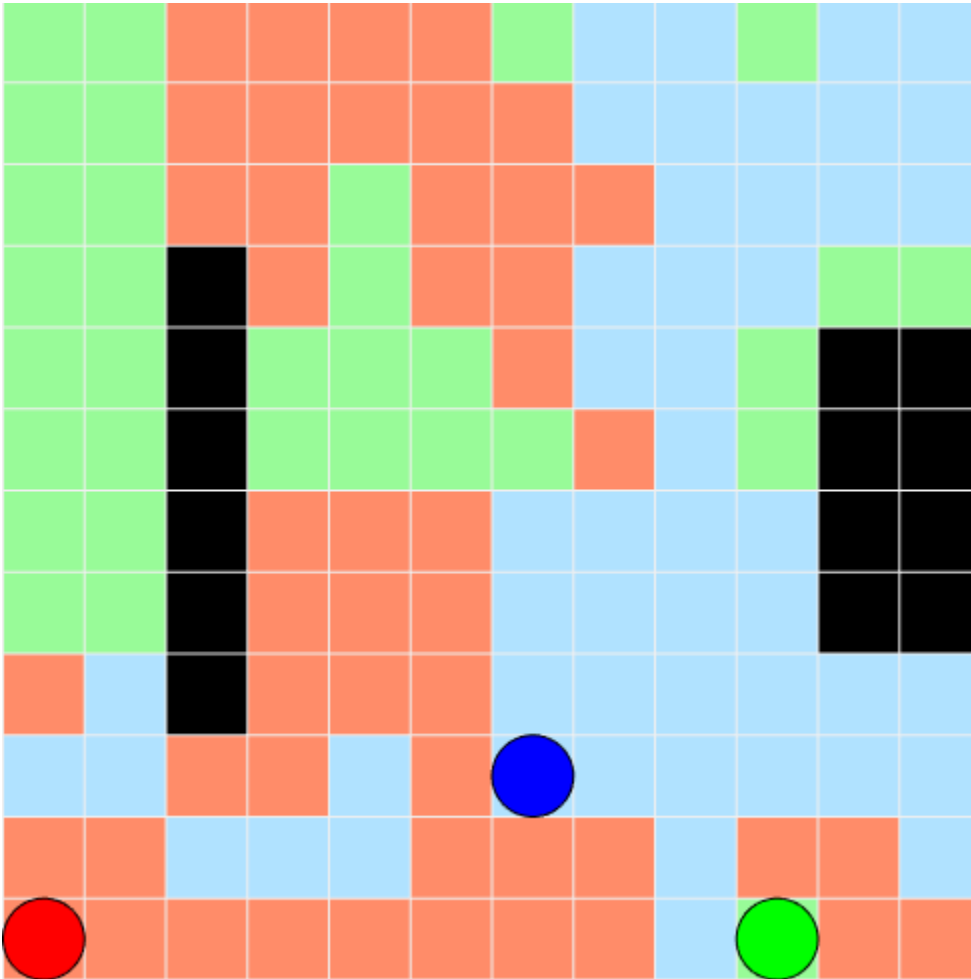
Multi-agent training is supported for Simulink® environments only. As shown in this example, if you define your environment behavior using a MATLAB® System object, you can incorporate it into a Simulink environment using a MATLAB System (Simulink) block.

Create Environment

The environment in this example is a 12x12 grid world containing obstacles, with unexplored cells marked in white and obstacles marked in black. There are three robots in the environment represented by the red, green, and blue circles. Three proximal policy optimization agents with discrete action spaces control the robots. To learn more about PPO agents, see "Proximal Policy Optimization Agents" on page 3-28.

The agents provide one of five possible movement actions (WAIT, UP, DOWN, LEFT, or RIGHT) to their respective robots. The robots decide whether an action is legal or illegal. For example, an action of moving LEFT when the robot is located next to the left boundary of the environment is deemed illegal. Similarly, actions for colliding against obstacles and other agents in the environment are

illegal actions and draw penalties. The environment dynamics are deterministic, which means the robots execute legal and illegal actions with 100% and 0% probabilities, respectively. The overall goal is to explore all cells as quickly as possible.

At each time step, an agent observes the state of the environment through a set of four images that identify the cells with obstacles, current position of the robot that is being controlled, position of other robots, and cells that have been explored during the episode. These images are combined to create a 4-channel 12x12 image observation set. The following figure shows an example of what the agent controlling the green robot observes for a given time step.
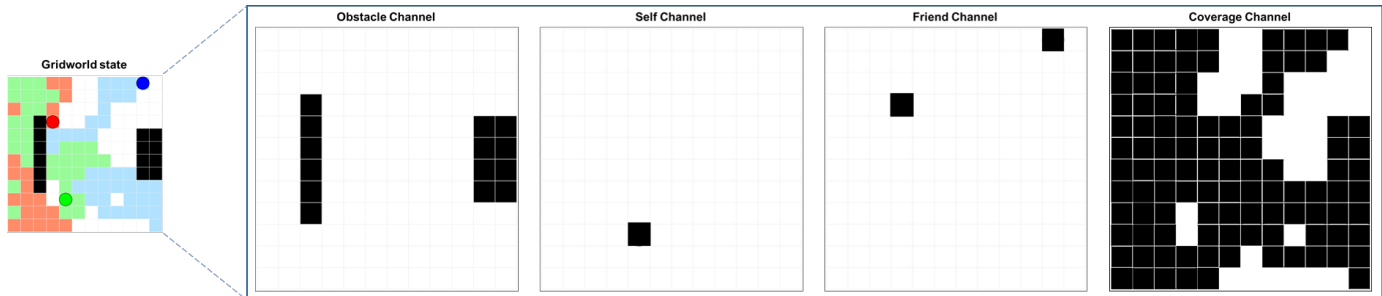
For the grid world environment:

- The search area is a 12x12 grid with obstacles.
- The observation for each agent is a 12x12x4 image.
- The discrete action set is a set of five actions (WAIT=0, UP=1, DOWN=2, LEFT=3, RIGHT=4).
- The simulation terminates when the grid is fully explored or the maximum number of steps is reached.

At each time step, agents receive the following rewards and penalties.

- +1 for moving to a previously unexplored cell (white).
- -0.5 for an illegal action (attempt to move outside the boundary or collide against other robots and obstacles)
- -0.05 for an action that results in movement (movement cost).
- -0.1 for an action that results in no motion (lazy penalty).
- If the grid is fully explored, +200 times the coverage contribution for that robot during the episode (ratio of cells explored to total cells)

Define the locations of obstacles within the grid using a matrix of indices. The first column contains the row indices, and the second column contains the column indices.

```
obsMat = [4 3; 5 3; 6 3; 7 3; 8 3; 9 3; 5 11; 6 11; 7 11; 8 11; 5 12; 6 12; 7 12; 8 12];
```

Initialize the robot positions.

```
sA0 = [2 2];
sB0 = [11 4];
sC0 = [3 12];
s0 = [sA0; sB0; sC0];
```

Specify the sample time, simulation time, and maximum number of steps per episode.

```
Ts = 0.1;
Tf = 100;
maxsteps = ceil(Tf/Ts);
```

Open the Simulink model.

```
mdl = "rlAreaCoverage";
open_system(mdl)
```
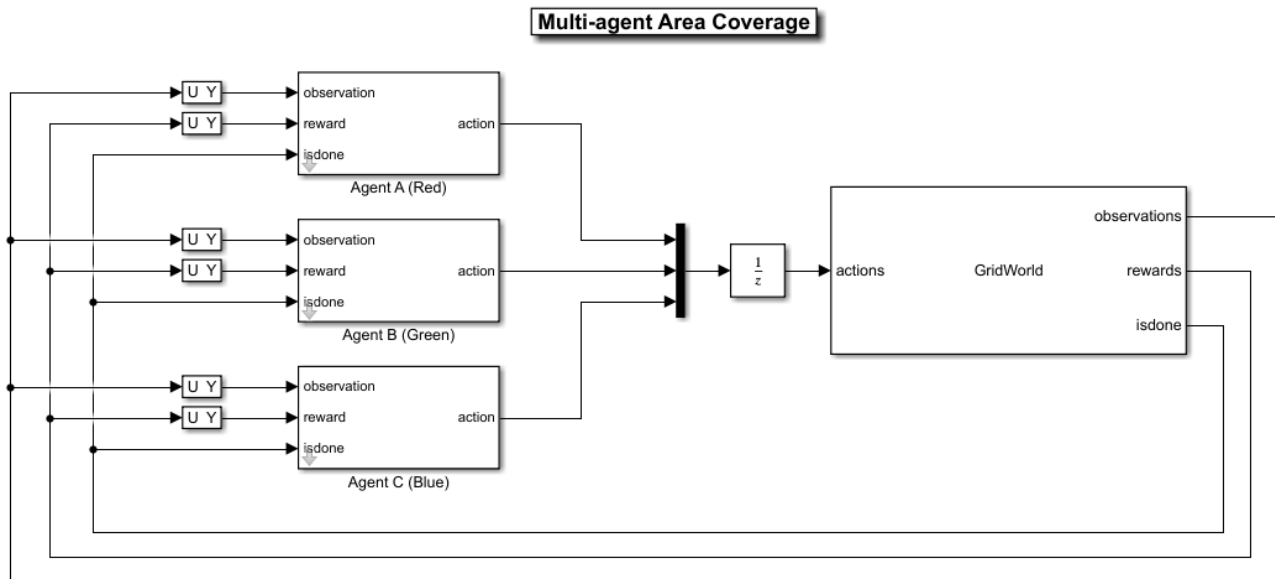

Copyright 2020 The MathWorks, Inc.

The `GridWorld` block is a MATLAB System block representing the training environment. The System object for this environment is defined in `GridWorld.m`.

In this example, the agents are homogeneous and have the same observation and action specifications. Create the observation and action specifications for the environment. For more information, see `rlNumericSpec` and `rlFiniteSetSpec`.

```
% Define observation specifications.
obsSize = [12 12 4];
oinfo = rlNumericSpec(obsSize);
oinfo.Name = 'observations';

% Define action specifications.
numAct = 5;
actionSpace = {0,1,2,3,4};
ainfo = rlFiniteSetSpec(actionSpace);
ainfo.Name = 'actions';
```

Specify the block paths for the agents

```
blks = mdl + ["/Agent A (Red)","/Agent B (Green)","/Agent C (Blue)"];
```

Create the environment interface, specifying the same observation and action specifications for all three agents.

```
env = rlSimulinkEnv(mdl,blks,{oinfo,oinfo,oinfo},{ainfo,ainfo,ainfo});
```

Specify a reset function for the environment. The reset function `resetMap` ensures that the robots start from random initial positions at the beginning of each episode. The random initialization makes the agents robust to different starting positions and improves training convergence.

```
env.ResetFcn = @(in) resetMap(in, obsMat);
```

Create Agents

PPO agents rely on actor and critic representations to learn the optimal policy. In this example, the agents maintain deep neural network-based function approximators for the actor and critic. Both the actor and critic have similar network structures with convolution and fully connected layers. The critic outputs a scalar value representing the state value $V(s)$. The actor outputs the probabilities $\pi(a|s)$ of taking each of the five actions. For more information, see `rlValueRepresentation` and `rlStochasticActorRepresentation`.

Set the random seed for reproducibility.

```
rng(0)
```

Create the actor and critic representations using the following steps.

1   Create the actor and critic deep neural networks.
2   Specify representation options for the actor and critic. In this example, specify the learning rates and the gradient thresholds. For more information, see `rlRepresentationOptions`.
3   Create the actor and critic representation objects.

Use the same network structure and representation options for all three agents.

```
for idx = 1:3
    % Create actor deep neural network.
    actorNetWork = [
        imageInputLayer(obsSize,'Normalization','none','Name','observations')
        convolution2dLayer(8,16,'Name','conv1','Stride',1,'Padding',1,'WeightsInitializer','he')
        reluLayer('Name','relu1')
        convolution2dLayer(4,8,'Name','conv2','Stride',1,'Padding','same','WeightsInitializer','l
        reluLayer('Name','relu2')
        fullyConnectedLayer(256,'Name','fc1','WeightsInitializer','he')
        reluLayer('Name','relu3')
        fullyConnectedLayer(128,'Name','fc2','WeightsInitializer','he')
        reluLayer('Name','relu4')
        fullyConnectedLayer(64,'Name','fc3','WeightsInitializer','he')
        reluLayer('Name','relu5')
        fullyConnectedLayer(numAct,'Name','output')
        softmaxLayer('Name','action')];

    % Create critic deep neural network.
    criticNetwork = [
        imageInputLayer(obsSize,'Normalization','none','Name','observations')
        convolution2dLayer(8,16,'Name','conv1','Stride',1,'Padding',1,'WeightsInitializer','he')
        reluLayer('Name','relu1')
        convolution2dLayer(4,8,'Name','conv2','Stride',1,'Padding','same','WeightsInitializer','l
        reluLayer('Name','relu2')
        fullyConnectedLayer(256,'Name','fc1','WeightsInitializer','he')
        reluLayer('Name','relu3')
        fullyConnectedLayer(128,'Name','fc2','WeightsInitializer','he')
```

```
            reluLayer('Name','relu4')
            fullyConnectedLayer(64,'Name','fc3','WeightsInitializer','he')
            reluLayer('Name','relu5')
            fullyConnectedLayer(1,'Name','output')];

        % Specify representation options for the actor and critic.
        actorOpts = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1);
        criticOpts = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1);

        % create actor and critic
        actor(idx) = rlStochasticActorRepresentation(actorNetWork,oinfo,ainfo,...
            'Observation',{'observations'},actorOpts);
        critic(idx) = rlValueRepresentation(criticNetwork,oinfo,...
            'Observation',{'observations'},criticOpts);
end
```

Specify the agent options using `rlPPOAgentOptions`. Use the same options for all three agents. During training, agents collect experiences until they reach the experience horizon of 128 steps and then train from mini-batches of 64 experiences. An objective function clip factor of 0.2 improves training stability, and a discount factor value of 0.995 encourages long-term rewards.

```
opt = rlPPOAgentOptions(...
    'ExperienceHorizon',128,...
    'ClipFactor',0.2,...
    'EntropyLossWeight',0.01,...
    'MiniBatchSize',64,...
    'NumEpoch',3,...
    'AdvantageEstimateMethod','gae',...
    'GAEFactor',0.95,...
    'SampleTime',Ts,...
    'DiscountFactor',0.995);
```

Create the agents using the defined actors, critics, and options.

```
agentA = rlPPOAgent(actor(1),critic(1),opt);
agentB = rlPPOAgent(actor(2),critic(2),opt);
agentC = rlPPOAgent(actor(3),critic(3),opt);
agents = [agentA,agentB,agentC];
```

Train Agents

Specify the following options for training the agents.

- Run the training for at most 1000 episodes, with each episode lasting at most 5000 time steps.
- Stop the training of an agent when its average reward over 100 consecutive episodes is 80 or more.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',1000,...
    'MaxStepsPerEpisode',maxsteps,...
    'Plots','training-progress',...
    'ScoreAveragingWindowLength',100,...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',80);
```

To train multiple agents, specify an array of agents to the `train` function. The order of the agents in the array must match the order of agent block paths specified during environment creation. Doing so

ensures that the agent objects are linked to the appropriate action and observation specifications in the environment.

Training is a computationally intensive process that takes several minutes to complete. To save time while running this example, load pretrained agent parameters by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    stats = train(agents,env,trainOpts);
else
    load('rlAreaCoverageParameters.mat');
    setLearnableParameters(agentA,agentAParams);
    setLearnableParameters(agentB,agentBParams);
    setLearnableParameters(agentC,agentCParams);
end
```

The following figure shows a snapshot of the training progress. You can expect different results due to randomness in the training process.
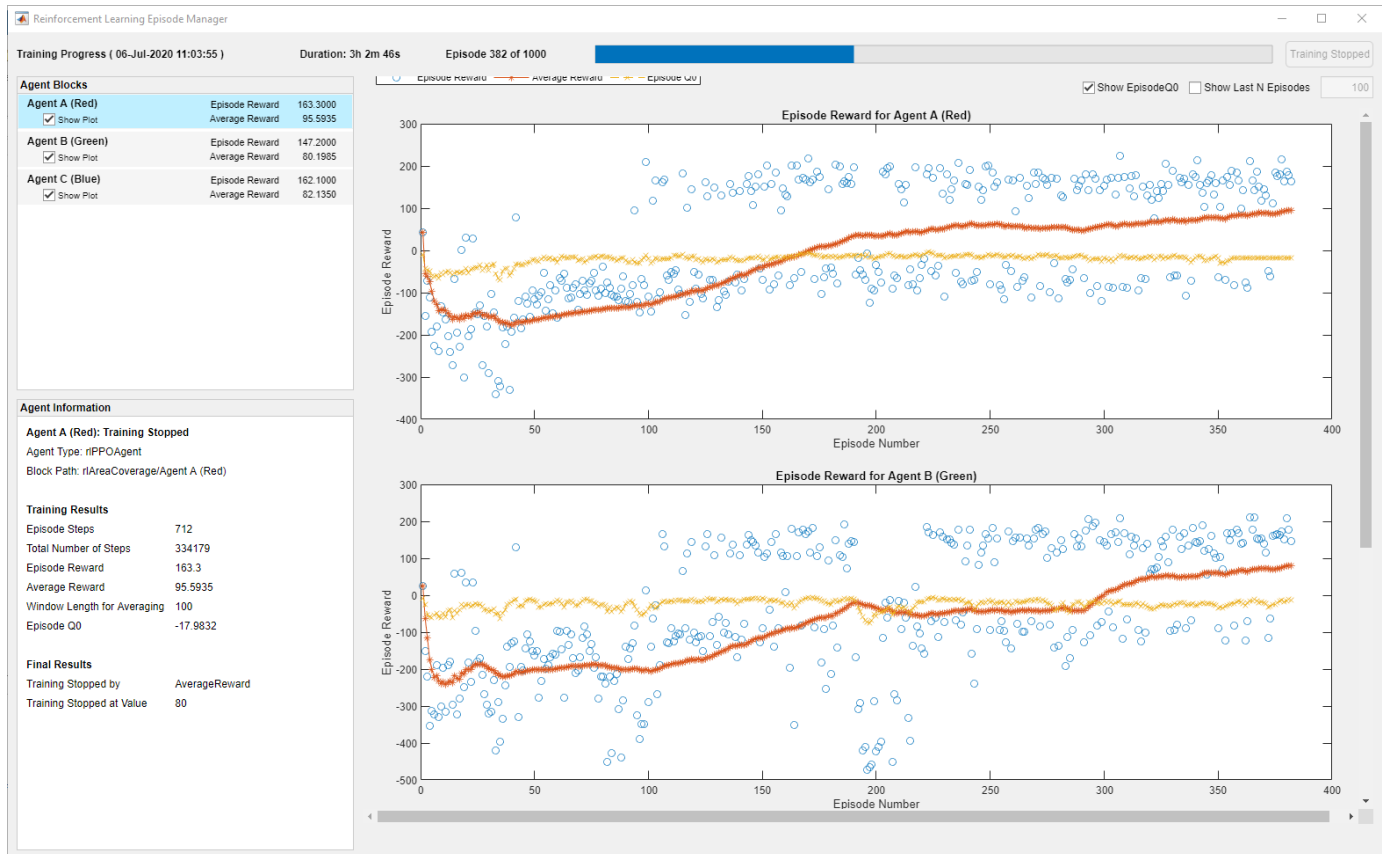
Simulate Agents

Simulate the trained agents within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
rng(0) % reset the random seed
simOpts = rlSimulationOptions('MaxSteps',maxsteps);
experience = sim(env,agents,simOpts);
```
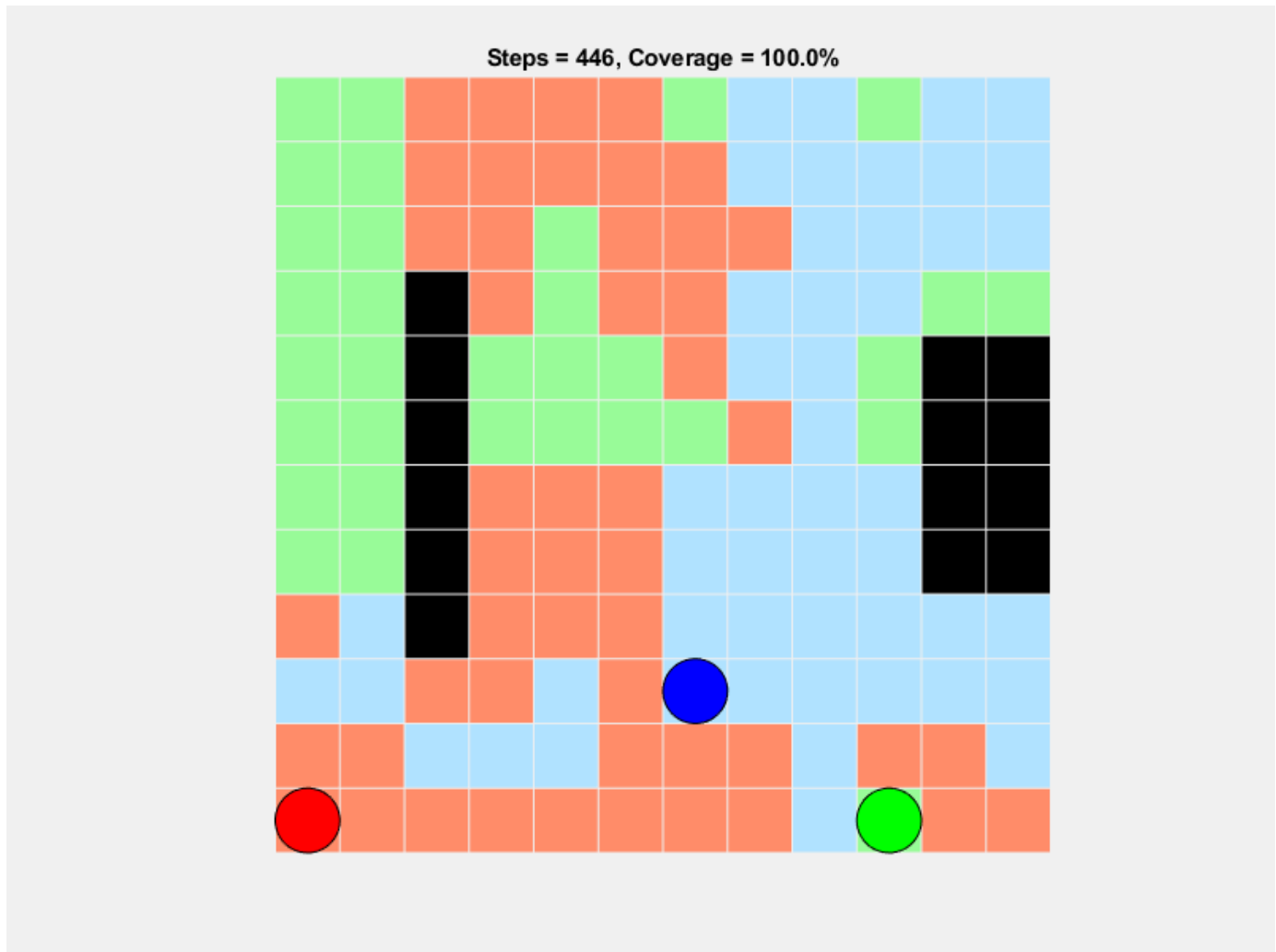

The agents successfully cover the entire grid world.

See Also

`rlSimulinkEnv` | `train`

More About

- "Train Reinforcement Learning Agents" on page 5-2

Train Multiple Agents for Path Following Control

This example shows how to train multiple agents to collaboratively perform path-following control (PFC) for a vehicle. The goal of PFC is to make the ego vehicle travel at a set velocity while maintaining a safe distance from a lead car by controlling longitudinal acceleration and braking, and also while keeping the vehicle travelling along the centerline of its lane by controlling the front steering angle. For more information on PFC, see Path Following Control System (Model Predictive Control Toolbox).

Overview

An example that trains a reinforcement learning agent to perform PFC is shown in "Train DDPG Agent for Path-Following Control" on page 5-152. In that example, a single deep deterministic policy gradient (DDPG) agent is trained to control both the longitudinal speed and lateral steering of the ego vehicle. In this example, you train two reinforcement learning agents — A DDPG agent provides continuous acceleration values for the longitudinal control loop and a deep Q-network (DQN) agent provides discrete steering angle values for the lateral control loop.

The trained agents perform PFC through cooperative behavior and achieve satisfactory results.

Create Environment

The environment for this example includes a simple bicycle model for the ego car and a simple longitudinal model for the lead car. The training goal is to make thbuild mpce ego car travel at a set velocity while maintaining a safe distance from lead car by controlling longitudinal acceleration and braking, while also keeping the ego car travelling along the centerline of its lane by controlling the front steering angle.

Load the environment parameters.

```
multiAgentPFCParams
```

Open the Simulink model.

```
mdl = "rlMultiAgentPFC";
open_system(mdl)
```
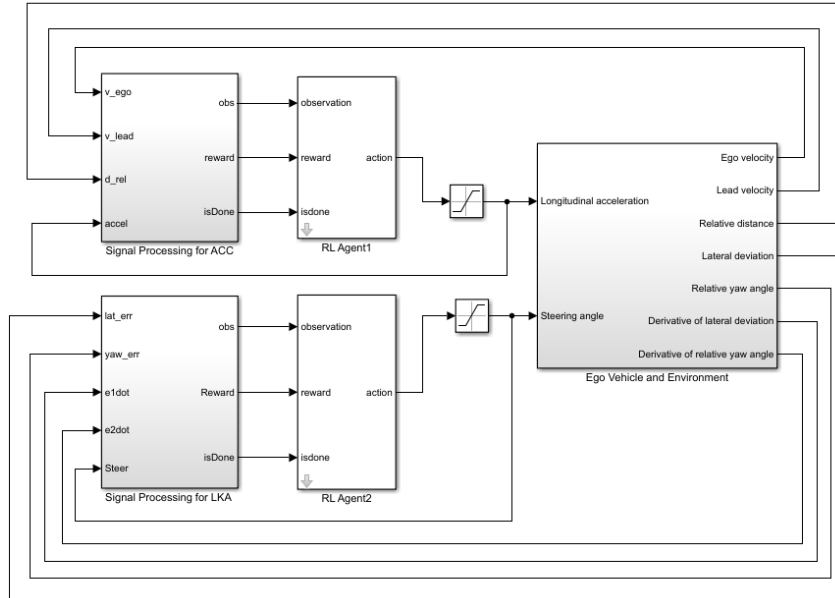
In this model, the two reinforcement learning agents (RL Agent1 and RL Agent2) provide longitudinal acceleration and steering angle signals, respectively.

The simulation terminates when any of the following conditions occur.

- $|e_1| > 1$ (magnitude of the lateral deviation exceeds 1)
- $V_{ego} < 0.5$ (longitudinal velocity of the ego car drops below 0.5.
- $D_{rel} < 0$ (distance between the ego and lead car is below zero)

For the longitudinal controller (RL Agent1):

- The reference velocity for the ego car V_{ref} is defined as follows. If the relative distance is less than the safe distance, the ego car tracks the minimum of the lead car velocity and driver-set velocity. In this manner, the ego car maintains some distance from the lead car. If the relative distance is greater than the safe distance, the ego car tracks the driver-set velocity. In this example, the safe distance is defined as a linear function of the ego car longitudinal velocity V, that is, $t_{gap}*V + D_{default}$. The safe distance determines the tracking velocity for the ego car.
- The observations from the environment contain the longitudinal measurements: the velocity error $e_V = V_{ref} - V$, its integral $\int e$, and the ego car longitudinal velocity V.
- The action signal consists of continuous acceleration values between -3 and 2 m/s^2.
- The reward r_t, provided at every time step t, is

$$r_t = -(10e_V^2 + 100a_{t-1}^2) \times 1e^{-3} - 10F_t + M_t$$

Here, a_{t-1} is the acceleration input from the previous time step, and:

- $F_t = 1$ if the simulation is terminated, otherwise $F_t = 0$.
- $M_t = 1$ if $e_V^2 < 1$, otherwise $M_t = 0$.

For the lateral controller (RL Agent2):

- The observations from the environment contain the lateral measurements: the lateral deviation e_1, relative yaw angle e_2, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.
- The action signal consists of discrete steering angle actions which take values from -15 degrees (-0.2618 rad) to 15 degrees (0.2618 rad) in steps of 1 degree (0.0175 rad).
- The reward r_t, provided at every time step t, is

$$r_t = -(100e_1^2 + 500u_{t-1}^2) \times 1e^{-3} - 10F_t + 2H_t$$

Here, u_{t-1} is the steering input from the previous time step, a_{t-1} is the acceleration input from the previous time step, and:

- $F_t = 1$ if the simulation is terminated, otherwise $F_t = 0$.
- $H_t = 1$ $e_1^2 < 0.01$, otherwise $H_t = 0$.

The logical terms in the reward functions (F_t, M_t, and H_t) penalize the agents if the simulation terminates early, while encouraging the agents to make both the lateral error and velocity error small.

Create the observation and action specifications for longitudinal control loop.

```
obsInfo1 = rlNumericSpec([3 1]);
actInfo1 = rlNumericSpec([1 1],'LowerLimit',-3,'UpperLimit',2);
```

Create the observation and action specifications for lateral control loop.

```
obsInfo2 = rlNumericSpec([6 1]);
actInfo2 = rlFiniteSetSpec((-15:15)*pi/180);
```

Combine the observation and action specifications as a cell array.

```
obsInfo = {obsInfo1,obsInfo2};
actInfo = {actInfo1,actInfo2};
```

Create a Simulink environment interface, specifying the block paths for both agent blocks. The order of the block paths must match the order of the observation and action specification cell arrays.

```
blks = mdl + ["/RL Agent1", "/RL Agent2"];
env = rlSimulinkEnv(mdl,blks,obsInfo,actInfo);
```

Specify a reset function for the environment using the `ResetFcn` property. The function `pfcResetFcn` randomly sets the initial poses of the lead and ego vehicles at the beginning of every episode during training.

```
env.ResetFcn = @pfcResetFcn;
```

Create Agents

For this example you create two reinforcement learning agents. First, fix the random seed for reproducibility.

```
rng(0)
```

Both agents operate at the same sample time in this example. Set the sample time value (in seconds).

```
Ts = 0.1;
```

Longitudinal Control

The agent for the longitudinal control loop is a DDPG agent. A DDPG agent approximates the long-term reward given observations and actions using a critic value function representation and selects actions using an actor policy representation. For more information on creating deep neural network value function and policy representations, see "Create Policy and Value Function Representations" on page 4-2.

Use the `createCCAgent` function to create a DDPG agent for longitudinal control. The structure of this agent is similar to the "Train DDPG Agent for Adaptive Cruise Control" on page 5-126 example.

```
agent1 = createACCAgent(obsInfo1,actInfo1,Ts);
```

Lateral Control

The agent for the lateral control loop is a DQN agent. A DQN agent approximates the long-term reward given observations and actions using a critic value function representation.

Use the `createLKAAgent` function to create a DQN agent for lateral control. The structure of this agent is similar to the "Train DQN Agent for Lane Keeping Assist" on page 5-134 example.

```
agent2 = createLKAAgent(obsInfo2,actInfo2,Ts);
```

Train Agents

Specify the training options. For this example, use the following options.

- Run each training episode for at most 5000 episodes, with each episode lasting at most `maxsteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Verbose` and `Plots` options).
- Stop training the DDPG and DQN agents when they receive an average reward greater than 480 and 1195, respectively. When one agent reaches its stop criteria, it simulates its own policy without learning while the other agent continues training.

```
Tf = 60; % simulation time
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',[480,1195]);
```

Train the agents using the `train` function. Training these agents is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    % Train the agent.
```

```
    trainingStats = train([agent1,agent2],env,trainingOpts);
else
    % Load pretrained agents for the example.
    load('rlPFCAgents.mat')
end
```

The following figure shows a snapshot of the training progress for the two agents.
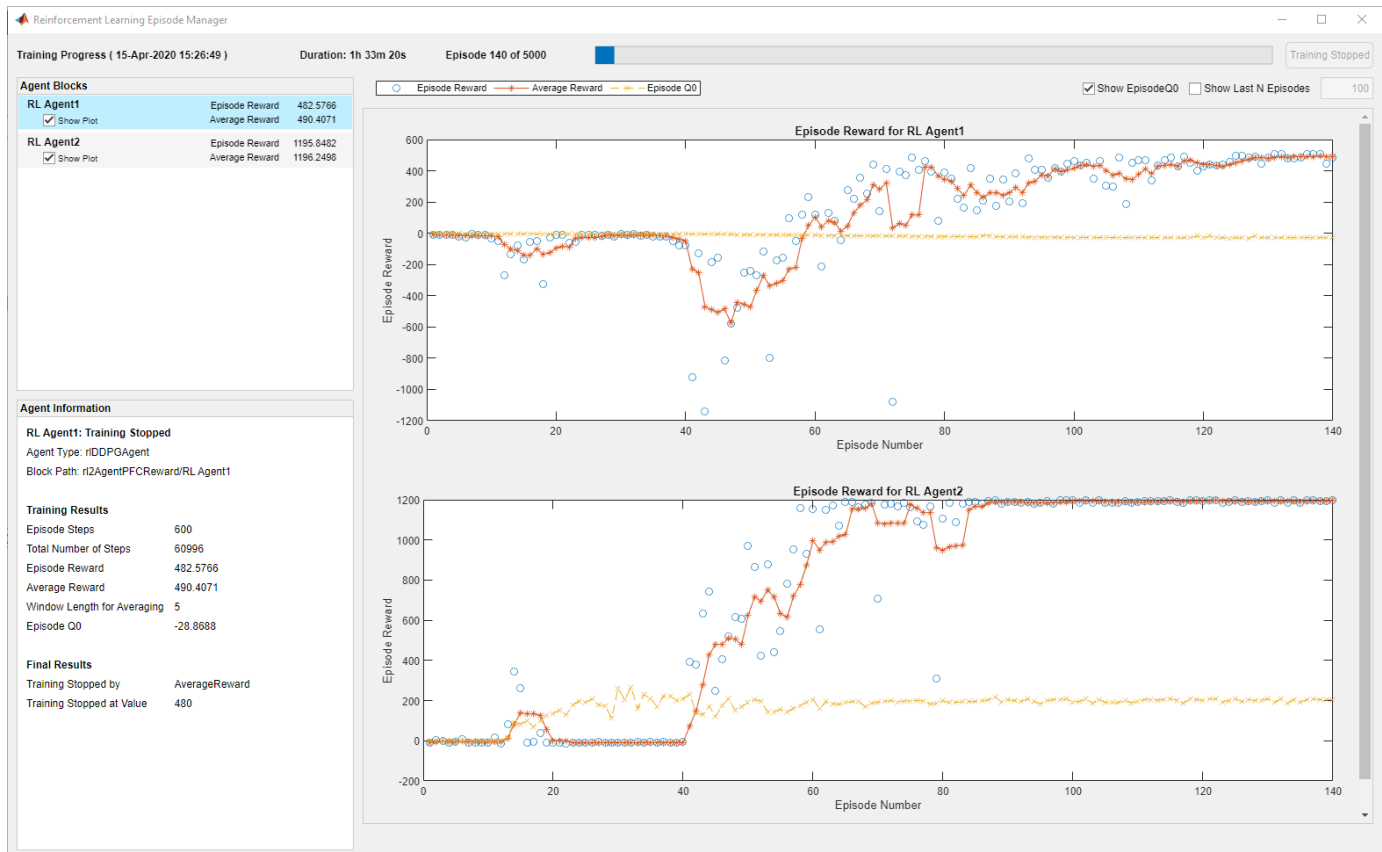
Simulate Agents

To validate the performance of the trained agents, simulate the agents within the Simulink environment by uncommenting the following commands. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,[agent1, agent2],simOptions);
```

To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.1;
x0_lead = 80;
sim(mdl)
```

The following plots show the results when the lead car is 70 m ahead of the ego car at the beginning of simulation.

- The lead car changes speed from 24 m/s to 30 m/s periodically (top-right plot). The ego car maintains a safe distance throughout the simulation (bottom-right plot).
- From 0 to 30 seconds, the ego car tracks the set velocity (top-right plot) and experiences some acceleration (top-left plot). After that, the acceleration is reduced to 0.
- The bottom-left plot shows the lateral deviation. As shown in the plot, the lateral deviation is greatly decreased within 1 second. The lateral deviation remains less than 0.1 m.
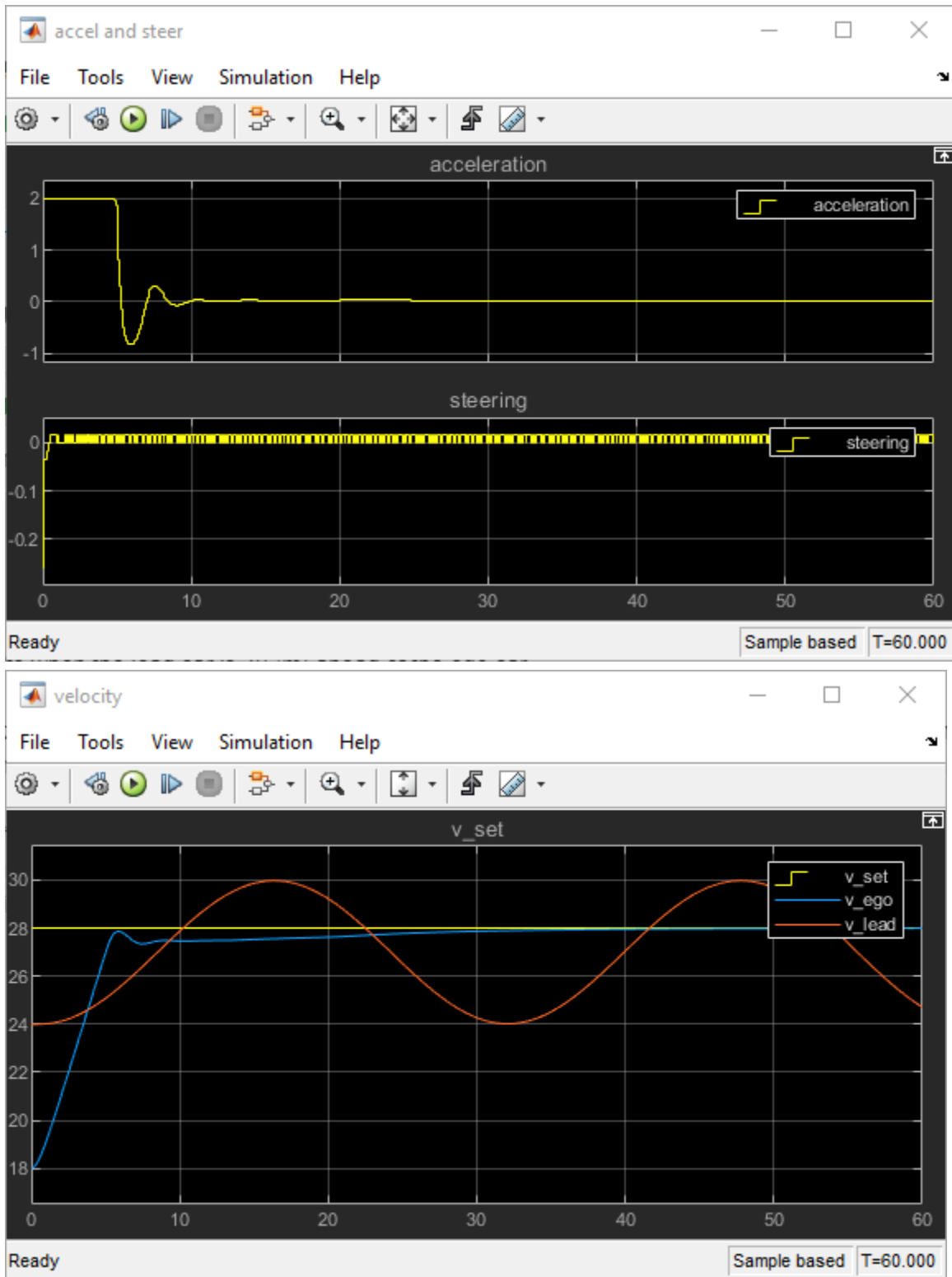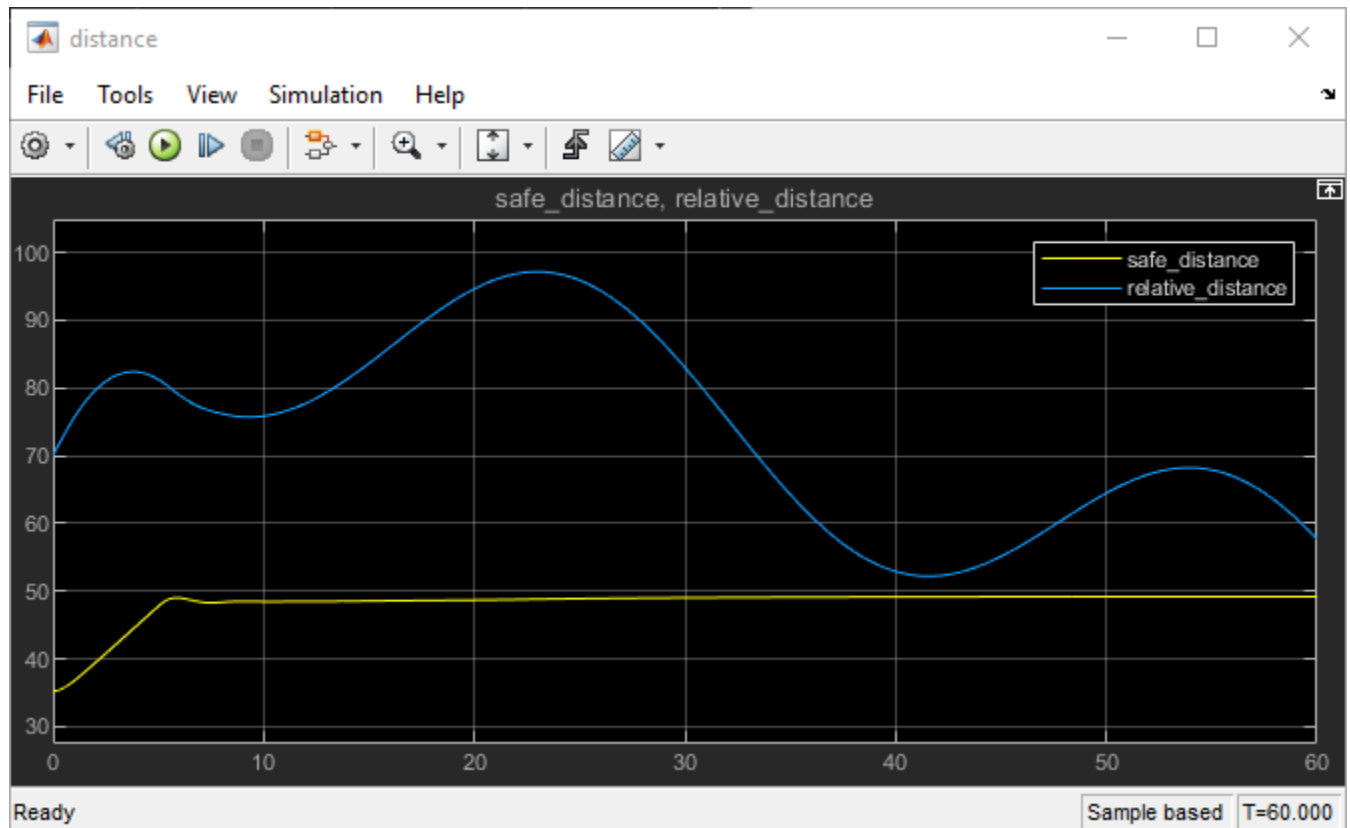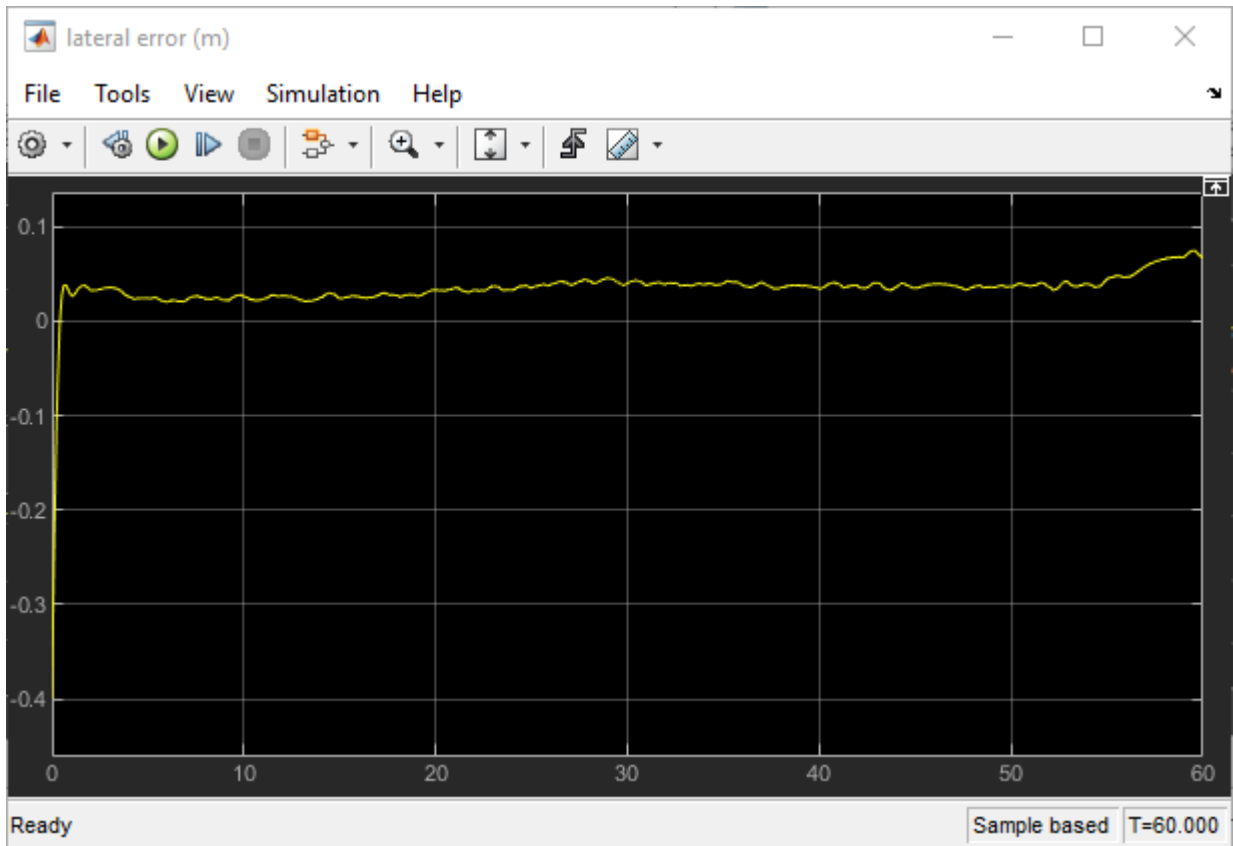
See Also

`rlSimulinkEnv` | `train`

More About

- "Train Reinforcement Learning Agents" on page 5-2

Train DDPG Agent for Adaptive Cruise Control

This example shows how to train a deep deterministic policy gradient (DDPG) agent for adaptive cruise control (ACC) in Simulink®. For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17.

Simulink Model

The reinforcement learning environment for this example is the simple longitudinal dynamics for an 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.

Specify the initial position and velocity for the two vehicles.

```
x0_lead = 50;    % initial position for lead car (m)
v0_lead = 25;    % initial velocity for lead car (m/s)
x0_ego = 10;     % initial position for ego car (m)
v0_ego = 20;     % initial velocity for ego car (m/s)
```

Specify standstill default spacing (m), time gap (s) and driver-set velocity (m/s).

```
D_default = 10;
t_gap = 1.4;
v_set = 30;
```

To simulate the physical limitations of the vehicle dynamics, constraint the acceleration to the range [−3,2] m/s^2.
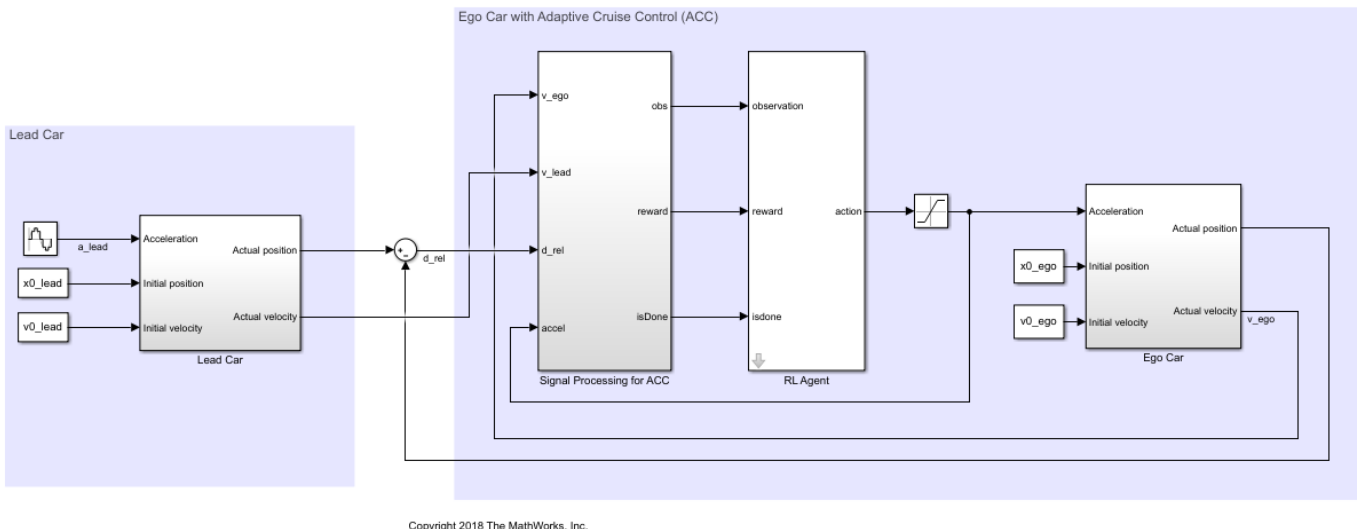
```
amin_ego = -3;
amax_ego = 2;
```

Define the sample time Ts and simulation duration Tf in seconds.

```
Ts = 0.1;
Tf = 60;
```

Open the model.

```
mdl = 'rlACCMdl';
open_system(mdl)
agentblk = [mdl '/RL Agent'];
```

Copyright 2018 The MathWorks, Inc.

For this model:

- The acceleration action signal from the agent to the environment is from –3 to 2 m/s^2.

- The reference velocity for the ego car V_{ref} is defined as follows. If the relative distance is less than the safe distance, the ego car tracks the minimum of the lead car velocity and driver-set velocity. In this manner, the ego car maintains some distance from the lead car. If the relative distance is greater than the safe distance, the ego car tracks the driver-set velocity. In this example, the safe distance is defined as a linear function of the ego car longitudinal velocity V; that is, $t_{gap}*V + D_{default}$. The safe distance determines the reference tracking velocity for the ego car.

- The observations from the environment are the velocity error $e = V_{ref} - V_{ego}$, its integral $\int e$, and the ego car longitudinal velocity V.

- The simulation is terminated when longitudinal velocity of the ego car is less than 0, or the relative distance between the lead car and ego car becomes less than 0.

- The reward r_t, provided at every time step t, is

$$r_t = -(0.1e_t^2 + u_{t-1}^2) + M_t$$

where u_{t-1} is the control input from the previous time step. The logical value $M_t = 1$ if velocity error $e_t^2 <\ = 0.25$; otherwise, $M_t = 0$.

Create Environment Interface

Create a reinforcement learning environment interface for the model.

Create the observation specification.

```
observationInfo = rlNumericSpec([3 1],'LowerLimit',-inf*ones(3,1),'UpperLimit',inf*ones(3,1));
observationInfo.Name = 'observations';
observationInfo.Description = 'information on velocity error and ego velocity';
```

Create the action specification.

```
actionInfo = rlNumericSpec([1 1],'LowerLimit',-3,'UpperLimit',2);
actionInfo.Name = 'acceleration';
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

To define the initial condition for the position of the lead car, specify an environment reset function using an anonymous function handle. The reset function `localResetFcn`, which is defined at the end of the example, randomizes the initial position of the lead car.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng('default')
```

Create DDPG agent

A DDPG agent approximates the long-term reward given observations and actions using a critic value function representation. To create the critic, first create a deep neural network with two inputs, the state and action, and one output. For more information on creating a neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
L = 48; % number of neurons
statePath = [
    featureInputLayer(3,'Normalization','none','Name','observation')
    fullyConnectedLayer(L,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(L,'Name','fc2')
    additionLayer(2,'Name','add')
    reluLayer('Name','relu2')
    fullyConnectedLayer(L,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')];

actionPath = [
    featureInputLayer(1,'Normalization','none','Name','action')
    fullyConnectedLayer(L, 'Name', 'fc5')];

criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork, actionPath);

criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');
```

View the critic network configuration.

```
plot(criticNetwork)
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1,'L2Regularization
```

Create the critic representation using the specified neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
```

A DDPG agent decides which action to take given observations by using an actor representation. To create the actor, first create a deep neural network with one input, the observation, and one output, the action.

Construct the actor similarly to the critic. For more information, see `rlDeterministicActorRepresentation`.

```
actorNetwork = [
    featureInputLayer(3,'Normalization','none','Name','observation')
    fullyConnectedLayer(L,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(L,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(L,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')
```
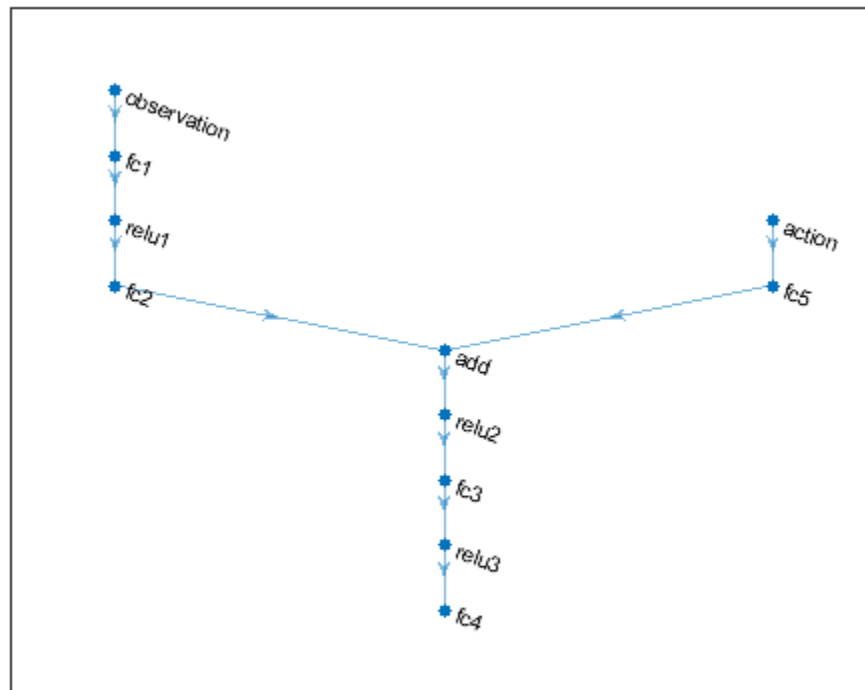
```
    tanhLayer('Name','tanh1')
    scalingLayer('Name','ActorScaling1','Scale',2.5,'Bias',-0.5)];

actorOptions = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1,'L2RegularizationFa
actor = rlDeterministicActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'ActorScaling1'},actorOptions);
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',64);
agentOptions.NoiseOptions.Variance = 0.6;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Then, create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent`.

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

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run each training episode for at most `5000` episodes, with each episode lasting at most 600 time steps.
- Display the training progress in the Episode Manager dialog box.
- Stop training when the agent receives an episode reward greater than 260.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue',260);
```

Train the agent using the `train` function. Training is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load a pretrained agent for the example.
    load('SimulinkACCDDPG.mat','agent')
end
```
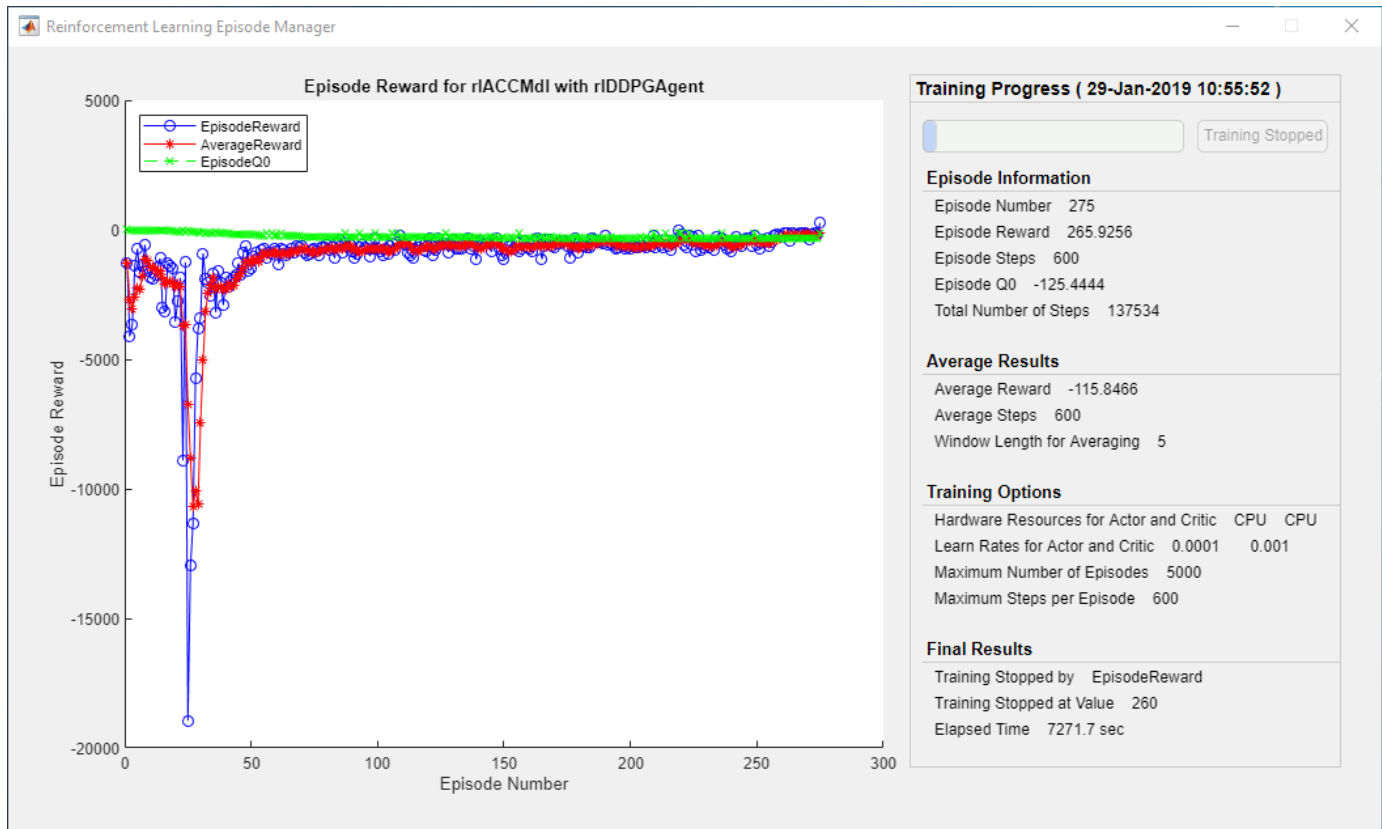
Simulate DDPG Agent

To validate the performance of the trained agent, simulate the agent within the Simulink environment by uncommenting the following commands. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
x0_lead = 80;
sim(mdl)
```

The following plots show the simulation results when lead car is 70 (m) ahead of the ego car.

- In the first 28 seconds, the relative distance is greater than the safe distance (bottom plot), so the ego car tracks set velocity (middle plot). To speed up and reach the set velocity, acceleration is positive (top plot).

- From 28 to 60 seconds, the relative distance is less than the safe distance (bottom plot), so the ego car tracks the minimum of the lead velocity and set velocity. From 28 to 36 seconds, the lead velocity is less than the set velocity (middle plot). To slow down and track the lead car velocity, acceleration is negative (top plot). From 36 to 60 seconds, the ego car adjusts its acceleration to track the reference velocity closely (middle plot). Within this time interval, the ego car tracks the set velocity from 43 to 52 seconds and tracks lead velocity from 36 to 43 seconds and 52 to 60 seconds.

Close the Simulink model.

```
bdclose(mdl)
```
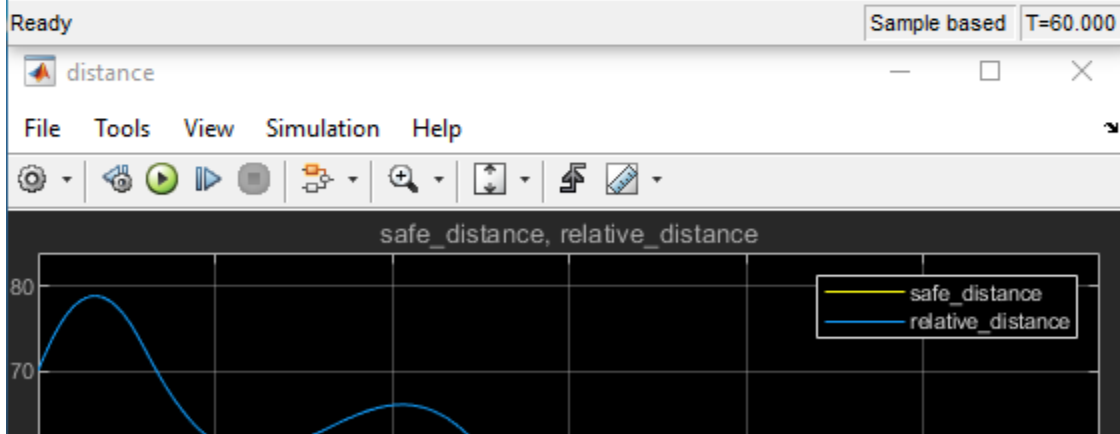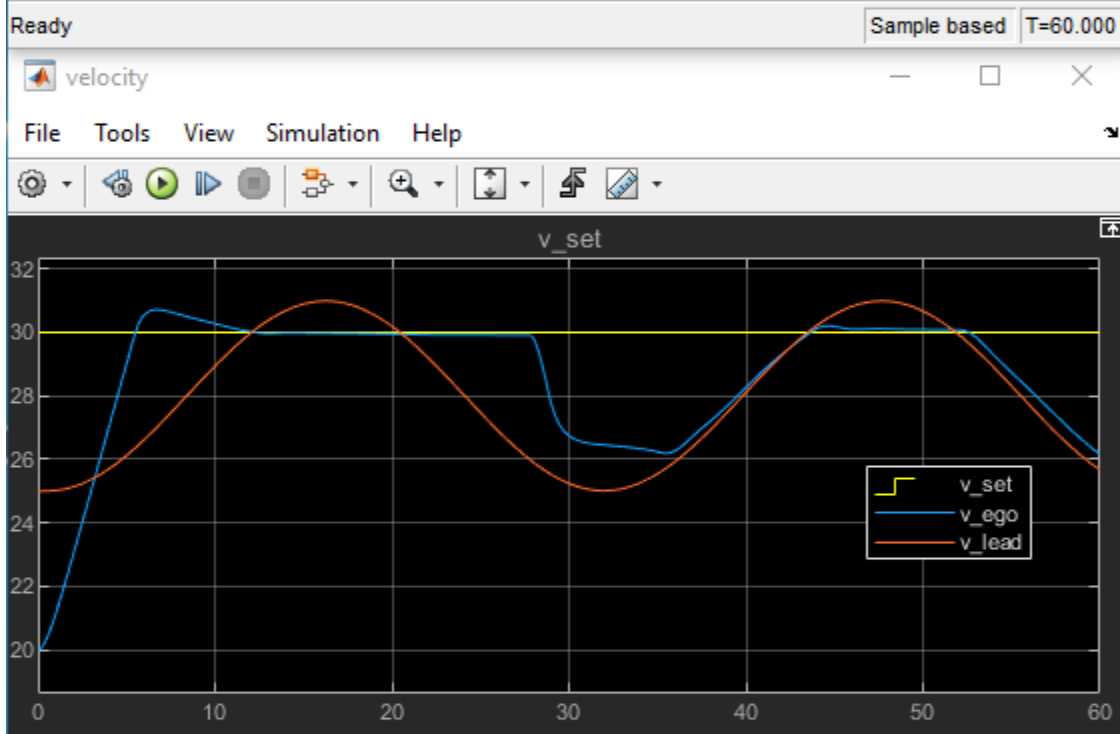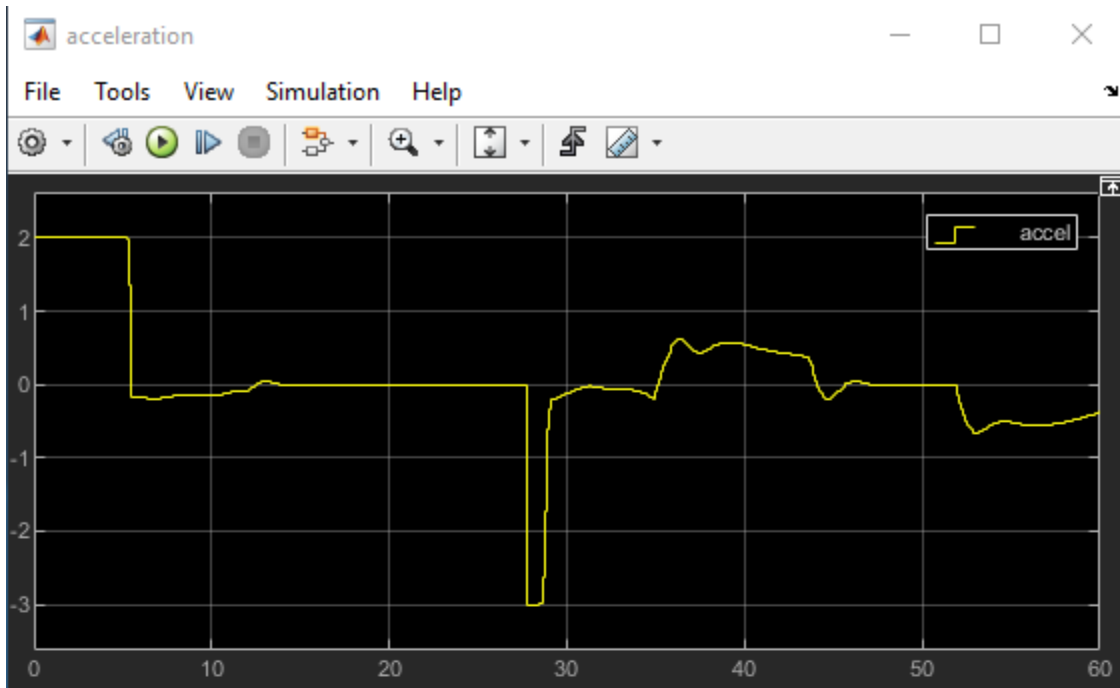
Reset Function

```
function in = localResetFcn(in)
% Reset the initial position of the lead car.
in = setVariable(in,'x0_lead',40+randi(60,1,1));
end
```

See Also
```
train
```

More About

- "Train Reinforcement Learning Agents" on page 5-2
- "Create Policy and Value Function Representations" on page 4-2

Train DQN Agent for Lane Keeping Assist

This example shows how to train a deep Q-learning network (DQN) agent for lane keeping assist (LKA) in Simulink®. For more information on DQN agents, see "Deep Q-Network Agents" on page 3-10.

Simulink Model for Ego Car

The reinforcement learning environment for this example is a simple bicycle model for ego vehicle dynamics. The training goal is to keep the ego vehicle traveling along the centerline of the lanes by adjusting the front steering angle. This example uses the same vehicle model as in "Lane Keeping Assist System Using Model Predictive Control" (Model Predictive Control Toolbox). The ego car dynamics are specified by the following parameters.

```
m = 1575;   % total vehicle mass (kg)
Iz = 2875;  % yaw moment of inertia (mNs^2)
lf = 1.2;   % longitudinal distance from center of gravity to front tires (m)
lr = 1.6;   % longitudinal distance from center of gravity to rear tires (m)
Cf = 19000; % cornering stiffness of front tires (N/rad)
Cr = 33000; % cornering stiffness of rear tires (N/rad)
Vx = 15;    % longitudinal velocity (m/s)
```

Define the sample time `Ts` and simulation duration `T` in seconds.

```
Ts = 0.1;
T = 15;
```

The output of the LKA system is the front steering angle of the ego car. To simulate the physical limitations of the ego car, constrain the steering angle to the range `[-0.5,0.5]` rad.

```
u_min = -0.5;
u_max = 0.5;
```

The curvature of the road is defined by a constant 0.001 (m^{-1}). The initial value for the lateral deviation is 0.2 m and the initial value for the relative yaw angle is –0.1 rad.

```
rho = 0.001;
e1_initial = 0.2;
e2_initial = -0.1;
```

Open the model.

```
mdl = 'rlLKAMdl';
open_system(mdl);
agentblk = [mdl '/RL Agent'];
```
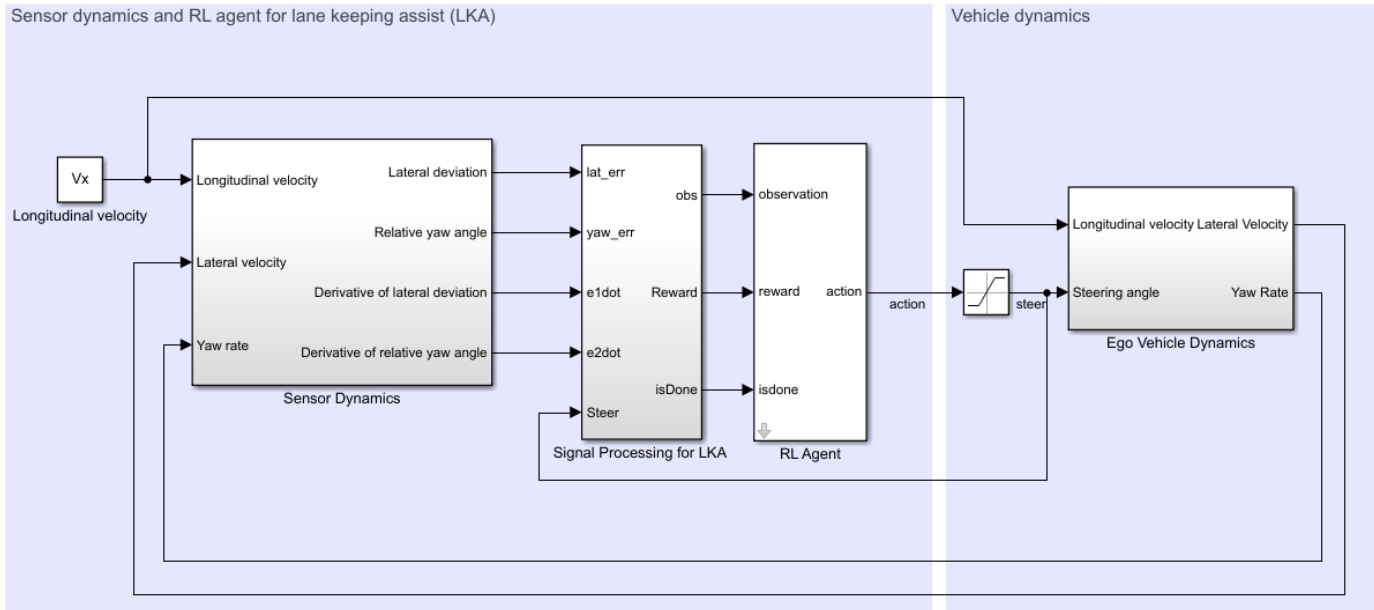
Copyright 2018 The MathWorks, Inc.

For this model:

- The steering-angle action signal from the agent to the environment is from –15 degrees to 15 degrees.
- The observations from the environment are the lateral deviation e_1, the relative yaw angle e_2, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.
- The simulation is terminated when the lateral deviation $|e_1| > 1$.
- The reward r_t, provided at every time step t, is

$$r_t = -(10e_1^2 + 5e_2^2 + 2u^2 + 5\dot{e}_1^2 + 5\dot{e}_2^2)$$

where u is the control input from the previous time step $t - 1$.

Create Environment Interface

Create a reinforcement learning environment interface for the ego vehicle. To do so, first create the observation and action specifications.

```
observationInfo = rlNumericSpec([6 1],'LowerLimit',-inf*ones(6,1),'UpperLimit',inf*ones(6,1));
observationInfo.Name = 'observations';
observationInfo.Description = 'information on lateral deviation and relative yaw angle';
actionInfo = rlFiniteSetSpec((-15:15)*pi/180);
actionInfo.Name = 'steering';
```

Then, create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

The interface has a discrete action space where the agent can apply one of 31 possible steering angles from –15 degrees to 15 degrees. The observation is the six-dimensional vector containing lateral deviation, relative yaw angle, as well as their derivatives and integrals with respect to time.

To define the initial condition for lateral deviation and relative yaw angle, specify an environment reset function using an anonymous function handle. This reset function randomizes the initial values for the lateral deviation and relative yaw

```
% angle.
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DQN agent

A DQN agent approximates the long-term reward, given observations and actions, using a value function critic representation.

DQN agents can use multi-output Q-value critic approximators, which are generally more efficient. A multi-output approximator has observations as inputs and state-action values as outputs. Each output element represents the expected cumulative long-term reward for taking the corresponding discrete action from the state indicated by the observation inputs.

To create the critic, first create a deep neural network with one input (the six-dimensional observed state) and one output vector with 31 elements (evenly spaced steering angles from -15 to 15 degrees). For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
nI = observationInfo.Dimension(1);   % number of inputs (6)
nL = 24;                             % number of neurons
nO = numel(actionInfo.Elements);     % number of outputs (31)

dnn = [
    featureInputLayer(nI,'Normalization','none','Name','state')
    fullyConnectedLayer(nL,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(nL,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(nO,'Name','fc3')];
```

View the network configuration.

```
figure
plot(layerGraph(dnn))
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1,'L2Regularization
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation information for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(dnn,observationInfo,actionInfo,...
    'Observation',{'state'},criticOptions);
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions`.

```
agentOptions = rlDQNAgentOptions(...
    'SampleTime',Ts,...
    'UseDoubleDQN',true,...
    'TargetSmoothFactor',1e-3,...
    'DiscountFactor',0.99,...
    'ExperienceBufferLength',1e6,...
    'MiniBatchSize',64);
```

Then, create the DQN agent using the specified critic representation and agent options. For more information, see `rlDQNAgent`.

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

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run each training episode for at most 5000 episodes, with each episode lasting at most `ceil(T/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option to `training-progress`) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the episode reward reaches -1.
- Save a copy of the agent for each episode where the cumulative reward is greater than -2.5.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(T/Ts);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue',-1,...
    'SaveAgentCriteria','EpisodeReward',...
    'SaveAgentValue',-2.5);
```

Train the agent using the `train` function. Training is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load the pretrained agent for the example.
    load('SimulinkLKADQNMulti.mat','agent')
end
```
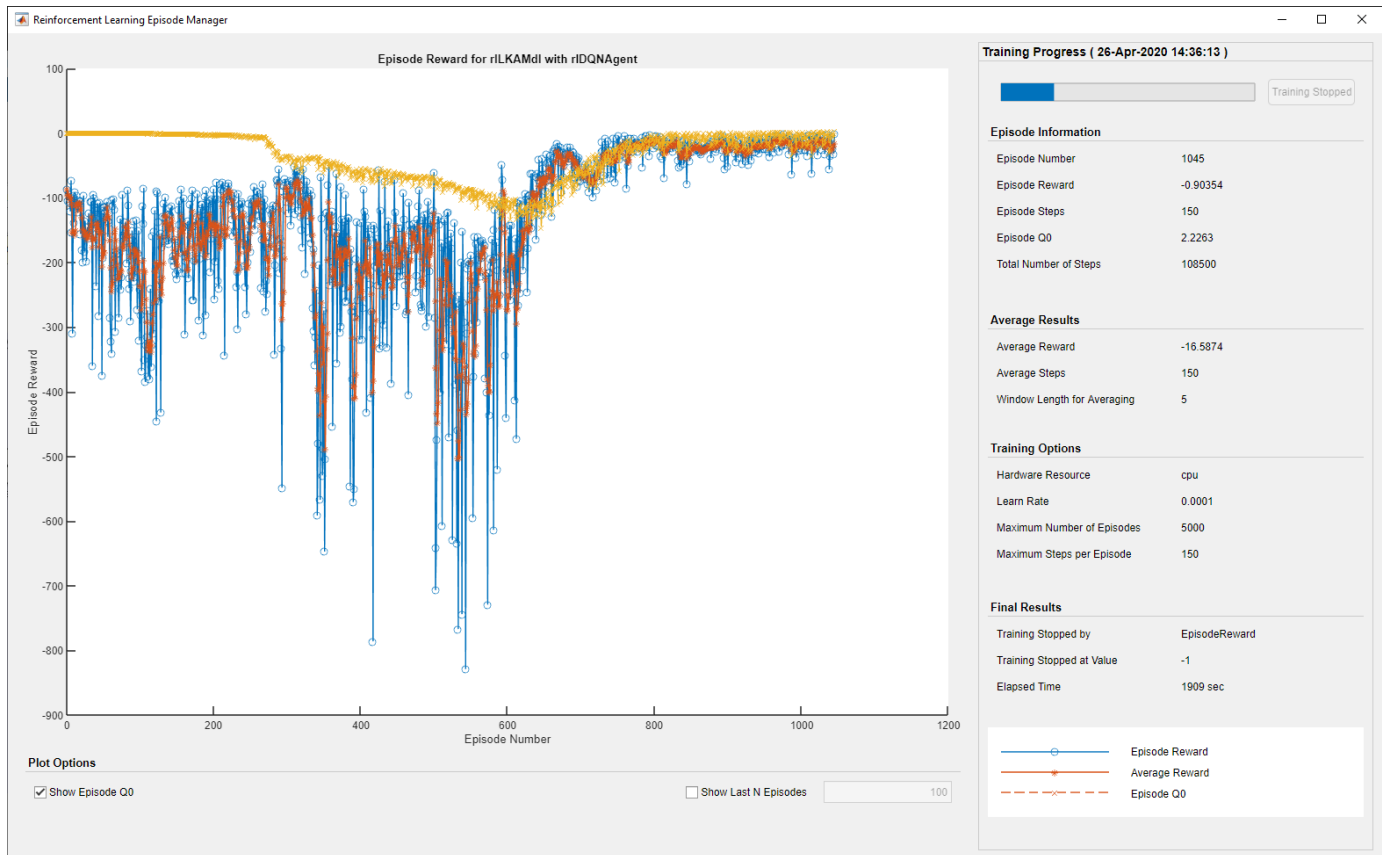
Simulate DQN Agent

To validate the performance of the trained agent, uncomment the following two lines and simulate the agent within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

To demonstrate the trained agent on deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.2;
sim(mdl)
```

As the plots show, the lateral error (top plot) and relative yaw angle (middle plot) are both driven close to zero. The vehicle starts from off the centerline (–0.4 m) and with a nonzero yaw angle error (0.2 rad). The lane keeping assist makes the ego car travel along the centerline after about 2.5 seconds. The steering angle (bottom plot) shows that the controller reaches steady state after about 2 seconds.
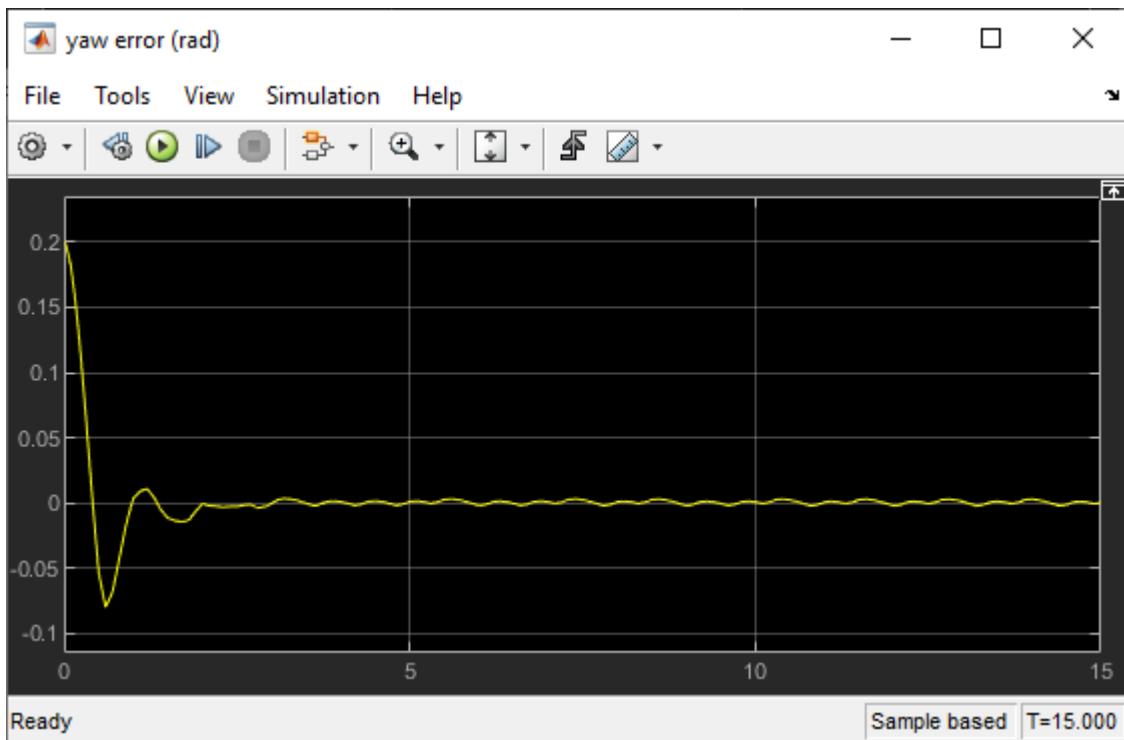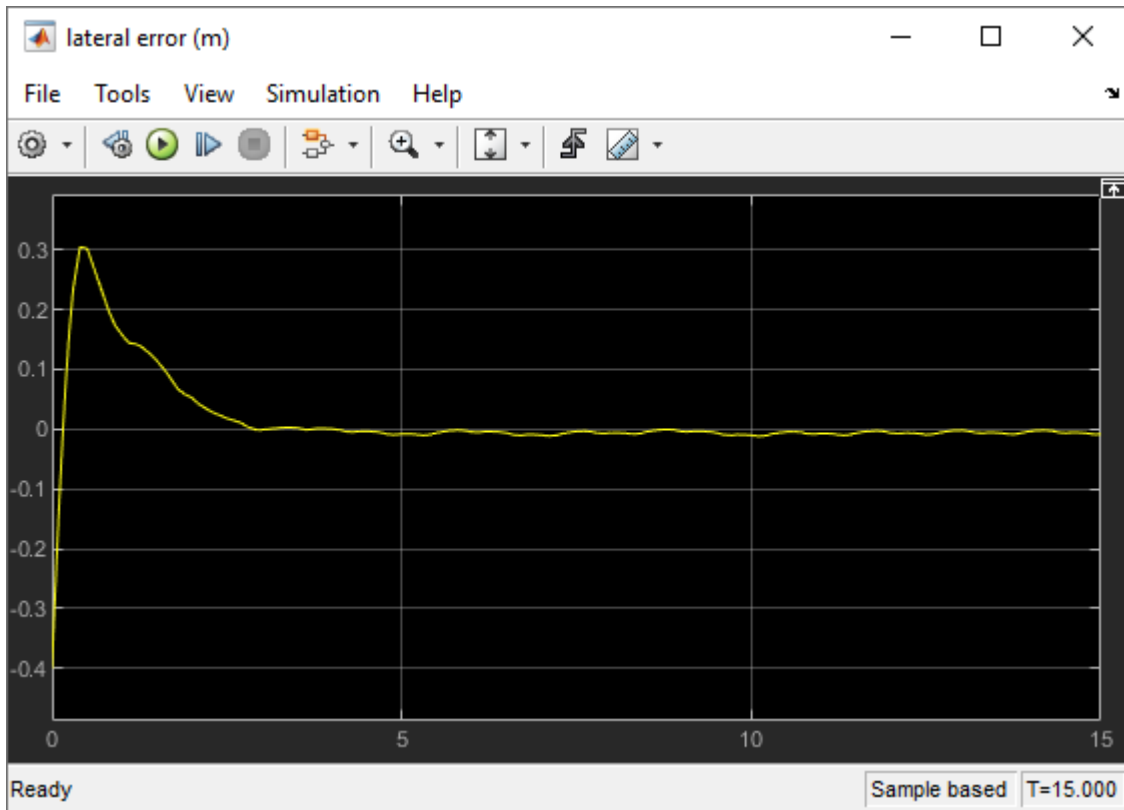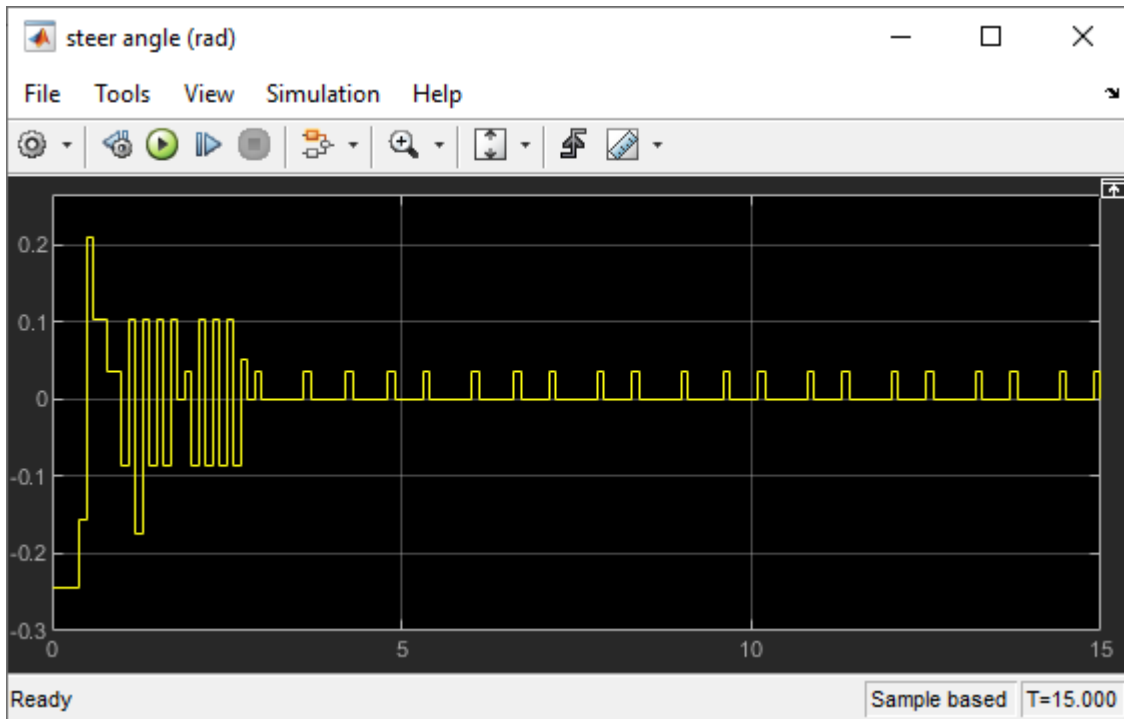
Close the Simulink model.

```
if ~doTraining
    %bdclose(mdl)
end
```

Reset Function

```
function in = localResetFcn(in)
    % reset
    in = setVariable(in,'e1_initial', 0.5*(-1+2*rand)); % random value for lateral deviation
    in = setVariable(in,'e2_initial', 0.1*(-1+2*rand)); % random value for relative yaw angle
end
```

See Also

`train`

More About

- "Train Reinforcement Learning Agents" on page 5-2
- "Create Policy and Value Function Representations" on page 4-2

Train PPO Agent for Automatic Parking Valet

This example demonstrates the design of a hybrid controller for an automatic search and parking task. The hybrid controller uses model predictive control (MPC) to follow a reference path in a parking lot and a trained reinforcement learning (RL) agent to perform the parking maneuver.

The automatic parking algorithm in this example executes a series of maneuvers while simultaneously sensing and avoiding obstacles in tight spaces. It switches between an adaptive MPC controller and an RL agent to complete the parking maneuver. The MPC controller moves the vehicle at a constant speed along a reference path while an algorithm searches for an empty parking spot. When a spot is found, the RL Agent takes over and executes a pretrained parking maneuver. Prior knowledge of the environment (the parking lot) including the locations of the empty spots and parked vehicles is available to the controllers.

Parking Lot

The parking lot is represented by the `ParkingLot` class, which stores information about the ego vehicle, empty parking spots, and static obstacles (parked cars). Each parking spot has a unique index number and an indicator light that is either green (free) or red (occupied). Parked vehicles are represented in black.

Create a `ParkingLot` object with a free spot at location 7.

```
freeSpotIdx = 7;
map = ParkingLot(freeSpotIdx);
```
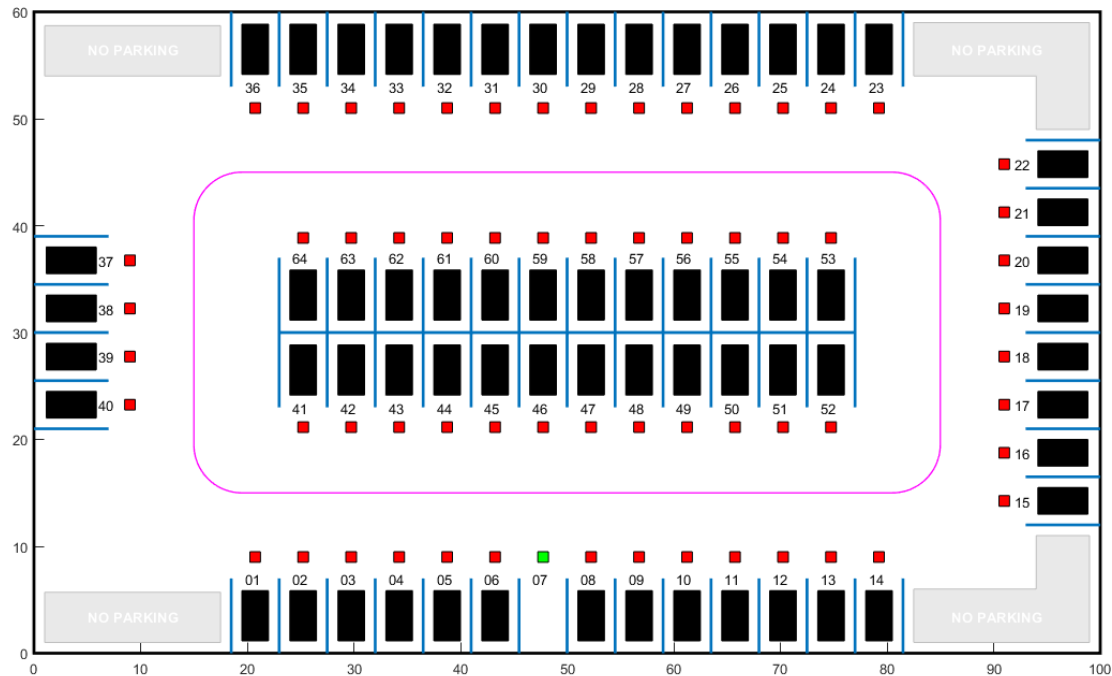

Specify an initial pose (X_0, Y_0, θ_0) for the ego vehicle. The target pose is determined based on the first available free spot as the vehicle navigates the parking lot.

```
egoInitialPose = [20, 15, 0];
```

Compute the target pose for the vehicle using the `createTargetPose` function. The target pose corresponds to the location in `freeSpotIdx`.

```
egoTargetPose = createTargetPose(map,freeSpotIdx)
```

egoTargetPose = *1×3*

    47.7500   4.9000   -1.5708

Sensor Modules

The parking algorithm uses camera and lidar sensors to gather information from the environment.

Camera

The field of view of a camera mounted on the ego vehicle is represented by the area shaded in green in the following figure. The camera has a field of view φ bounded by ± 60 degrees and a maximum measurement depth $d_{\max}$ of 10 m.
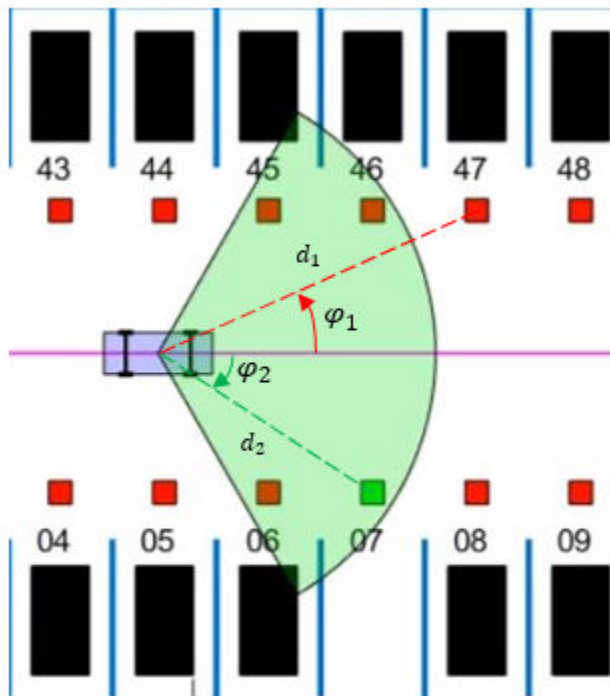
As the ego vehicle moves forward, the camera module senses the parking spots that fall within the field of view and determines whether a spot is free or occupied. For simplicity, this action is implemented using geometrical relationships between the spot locations and the current vehicle pose. A parking spot is within the camera range if $d_i \leq d_{\max}$ and $\varphi_{\min} \leq \varphi_i \leq \varphi_{\max}$, where d_i is the distance to the parking spot and φ_i is the angle to the parking spot.

Lidar

The reinforcement leangin agent uses lidar sensor readings to determine the proximity of the ego vehicle to other vehicles in the environment. The lidar sensor in this example is also modeled using geometrical relationships. Lidar distances are measured along 12 line segments that radially emerge from the center of the ego vehicle. When a lidar line intersects an obstacle, it returns the distance of the obstacle from the vehicle. The maximum measurable lidar distance along any line segment is 6 m.
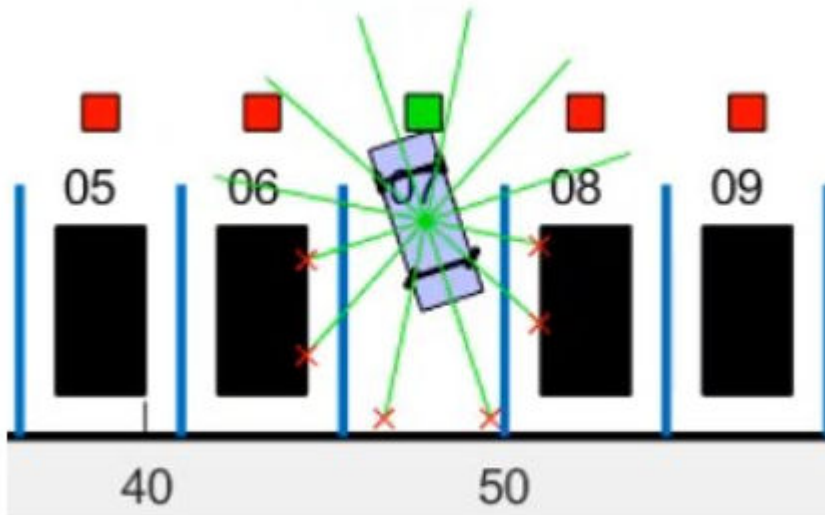
Auto Parking Valet Model

The parking valet model, including the controllers, ego vehicle, sensors, and parking lot, is implemented in Simulink®.

Load the auto parking valet parameters.

```
autoParkingValetParams
```

Open the Simulink model.

```
mdl = 'rlAutoParkingValet';
open_system(mdl)
```
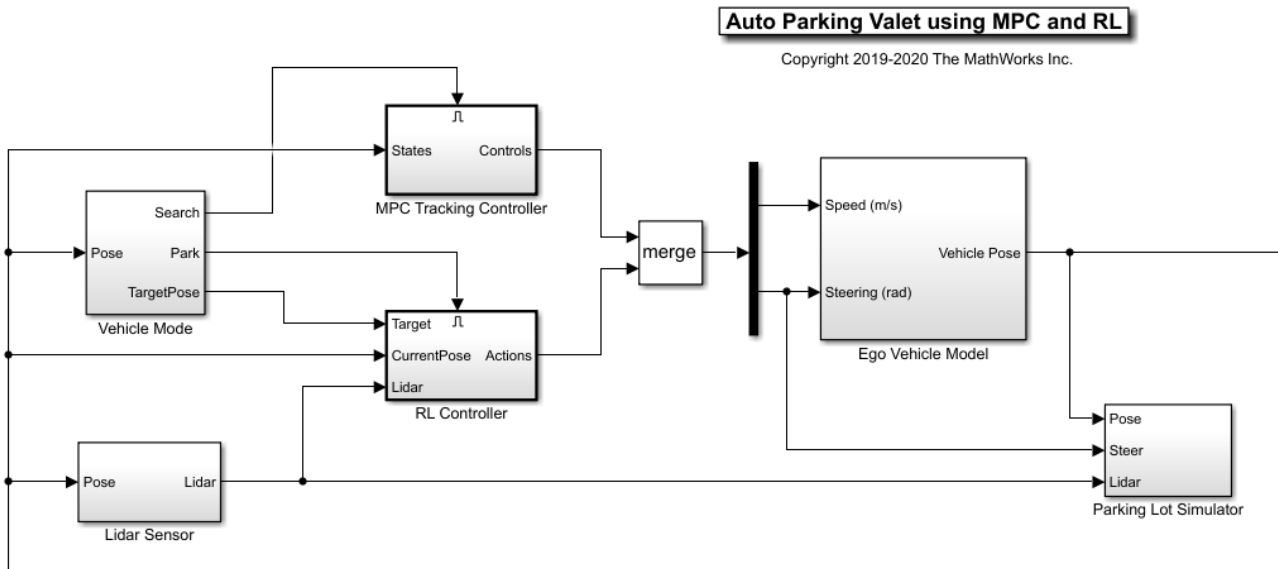
The ego vehicle dynamics in this model are represented by a single-track bicycle model with two inputs: vehicle speed v (m/s) and steering angle δ (radians). The MPC and RL controllers are placed within Enabled Subsystem blocks that are activated by signals representing whether the vehicle has to search for an empty spot or execute a parking maneuver. The enable signals are determined by the Camera algorithm within the Vehicle Mode subsystem. Initially, the vehicle is in *search* mode and the MPC controller tracks the reference path. When a free spot is found, *park* mode is activated and the RL agent executes the parking maneuver.

Adaptive Model Predictive Controller

Create the adaptive MPC controller object for reference trajectory tracking using the `createMPCForParking` script. For more information on adaptive MPC, see "Adaptive MPC" (Model Predictive Control Toolbox).

```
createMPCForParking
```

Reinforcement Learning Environment

The environment for training the RL agent is the region shaded in red in the following figure. Due to symmetry in the parking lot, training within this region is sufficient for the policy to adjust to other regions after applying appropriate coordinate transformations to the observations. Using this smaller training region significantly reduces training time compared to training over the entire parking lot.
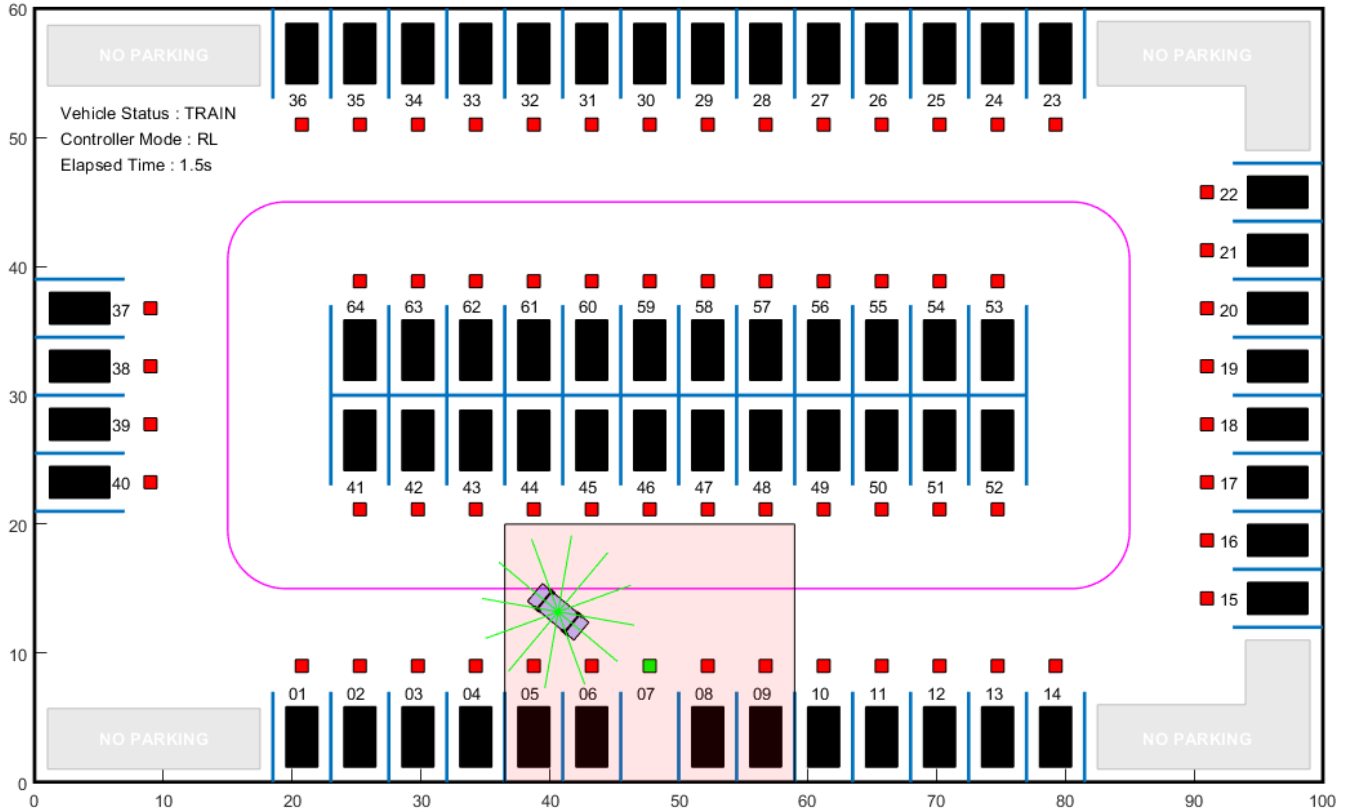
For this environment:

- The training region is a 22.5 m x 20 m space with the target spot at its horizontal center.
- The observations are the position errors X_e and Y_e of the ego vehicle with respect to the target pose, the sine and cosine of the true heading angle θ, and the lidar sensor readings.
- The vehicle speed during parking is a constant 2 m/s.
- The action signals are discrete steering angles that range between +/- 45 degrees in steps of 15 degrees.
- The vehicle is considered parked if the errors with respect to target pose are within specified tolerances of +/- 0.75 m (position) and +/-10 degrees (orientation).
- The episode terminates if the ego vehicle goes out of the bounds of the training region, collides with an obstacle, or parks successfully.
- The reward r_t provided at time t, is:

$$r_t = 2e^{-\left(0.05X_e^2 + 0.04Y_e^2\right)} + 0.5e^{-40\theta_e^2} - 0.05\delta^2 + 100f_t - 50g_t$$

Here, X_e, Y_e, and θ_e are the position and heading angle errors of the ego vehicle from the target pose, and δ is the steering angle. f_t (0 or 1) indicates whether the vehicle has parked and g_t (0 or 1) indicates if the vehicle has collided with an obstacle at time t.

The coordinate transformations on vehicle pose (X, Y, θ) observations for different parking spot locations are as follows:

- 1-14: no transformation

- 15-22: $\bar{X} = Y, \bar{Y} = -X, \bar{\theta} = \theta - \pi/2$

- 23-36: $\bar{X} = 100 - X, \bar{Y} = 60 - Y, \bar{\theta} = \theta - \pi$

- 37-40: $\bar{X} = 60 - Y, \bar{Y} = X, \bar{\theta} = \theta - 3\pi/2$

- 41-52: $\bar{X} = 100 - X, \bar{Y} = 30 - Y, \bar{\theta} = \theta + \pi$

- 53-64: $\bar{X} = X, \bar{Y} = Y - 28, \bar{\theta} = \theta$

Create the observation and action specifications for the environment.

```
numObservations = 16;
observationInfo = rlNumericSpec([numObservations 1]);
observationInfo.Name = 'observations';

steerMax = pi/4;
discreteSteerAngles = -steerMax : deg2rad(15) : steerMax;
actionInfo = rlFiniteSetSpec(num2cell(discreteSteerAngles));
actionInfo.Name = 'actions';
numActions = numel(actionInfo.Elements);
```

Create the Simulink environment interface, specifying the path to the RL Agent block.

```
blk = [mdl '/RL Controller/RL Agent'];
env = rlSimulinkEnv(mdl,blk,observationInfo,actionInfo);
```

Specify a reset function for training. The `autoParkingValetResetFcn` function resets the initial pose of the ego vehicle to random values at the start of each episode.

```
env.ResetFcn = @autoParkingValetResetFcn;
```

For more information on creating Simulink environments, see `rlSimulinkEnv`.

Create Agent

The RL agent in this example is a proximal policy optimization (PPO) agent with a discrete action space. PPO agents rely on actor and critic representations to learn the optimal policy. The agent maintains deep neural network-based function approximators for the actor and critic. To learn more about PPO agents, see "Proximal Policy Optimization Agents" on page 3-28.

Set the random seed generator for reproducibility.

```
rng(0)
```

To create the critic representations, first create a deep neural network with 16 inputs and one output. The output of the critic network is the state value function for a particular observation.

```
criticNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','observations')
    fullyConnectedLayer(128,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(128,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(128,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')];
```

Create the critic for the PPO agent using the . For more information, see `rlValueRepresentation` and `rlRepresentationOptions`.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1);
critic = rlValueRepresentation(criticNetwork,observationInfo,...
    'Observation',{'observations'},criticOptions);
```

The outputs of the actor networks are the probabilities of taking each possible steering action when the vehicle is in a certain state. Create the actor deep neural network.

```
actorNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','observations')
    fullyConnectedLayer(128,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(128,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(numActions, 'Name', 'out')
    softmaxLayer('Name','actionProb')];
```

Create a stochastic actor representation for the PPO agent. For more information, see `rlStochasticActorRepresentation`.

```
actorOptions = rlRepresentationOptions('LearnRate',2e-4,'GradientThreshold',1);
actor = rlStochasticActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'observations'},actorOptions);
```

Specify the agent options and create the PPO agent. For more information on PPO agent options, see `rlPPOAgentOptions`.

```
agentOpts = rlPPOAgentOptions(...
    'SampleTime',Ts,...
    'ExperienceHorizon',200,...
    'ClipFactor',0.2,...
    'EntropyLossWeight',0.01,...
    'MiniBatchSize',64,...
    'NumEpoch',3,...
    'AdvantageEstimateMethod',"gae",...
    'GAEFactor',0.95,...
    'DiscountFactor',0.998);
agent = rlPPOAgent(actor,critic,agentOpts);
```

During training, the agent collects experiences until it reaches experience horizon of 200 steps or the episode terminates and then trains from mini-batches of 64 experiences for three epochs. An objective function clip factor of 0.2 improves training stability and a discount factor value of 0.998 encourages long term rewards. Variance in critic the output is reduced by the generalized advantage estimate method with a GAE factor of 0.95.

Train Agent

For this example, you train the agent for a maximum of 10000 episodes, with each episode lasting a maximum of 200 time steps. The training terminates when the maximum number of episodes is reached or the average reward over 100 episodes exceeds 100.

Specify the options for training using an `rlTrainingOptions` object.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',10000,...
```

```
        'MaxStepsPerEpisode',200,...
        'ScoreAveragingWindowLength',200,...
        'Plots','training-progress',...
        'StopTrainingCriteria','AverageReward',...
        'StopTrainingValue',80);
```

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    trainingStats = train(agent,env,trainOpts);
else
    load('rlAutoParkingValetAgent.mat','agent');
end
```
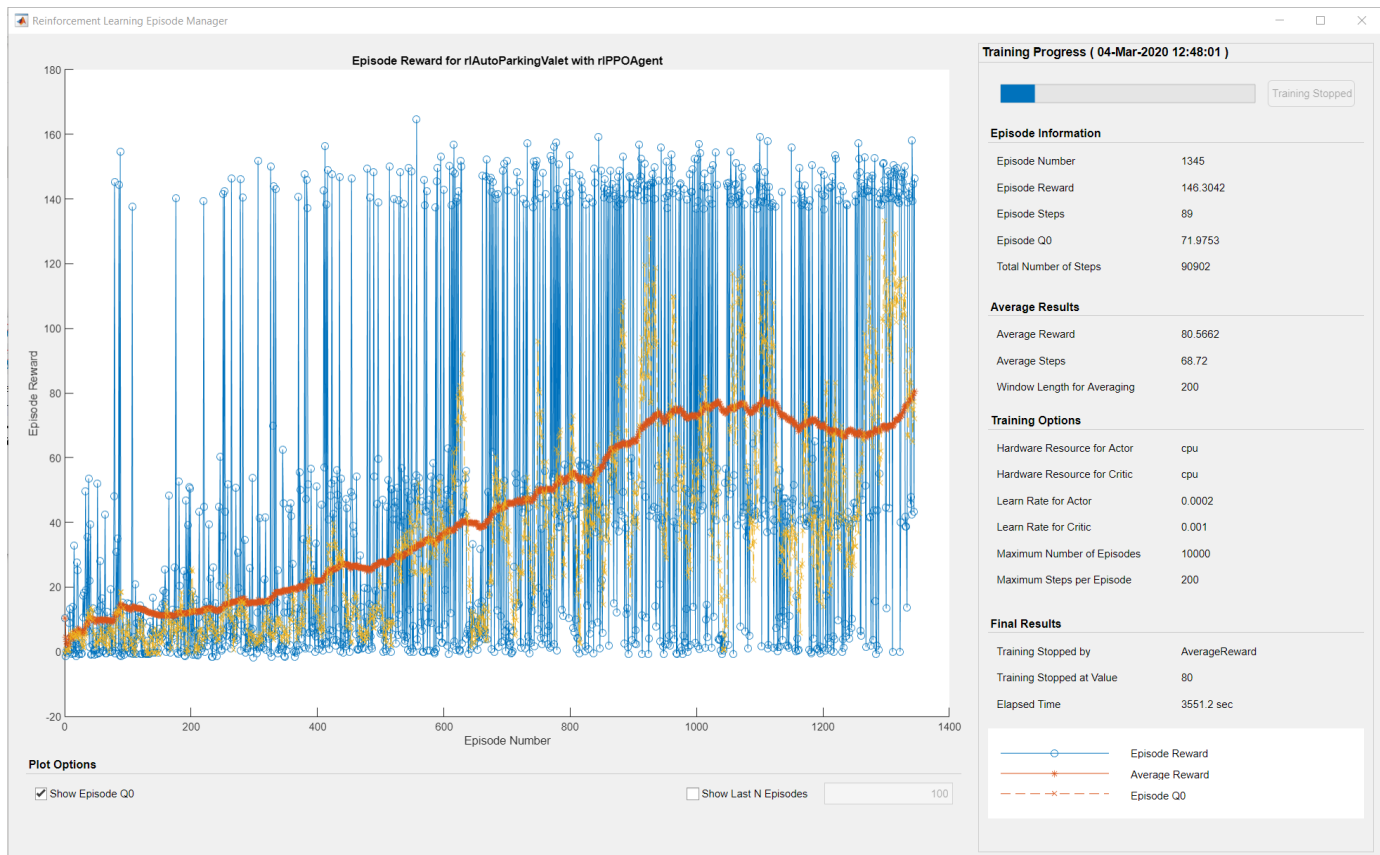

Simulate Agent

Simulate the model to park the vehicle in the free parking spot. To simulate the vehicle parking in different locations, change the free spot location in the following code.

```
freeSpotIdx = 7;  % free spot location
sim(mdl);
```
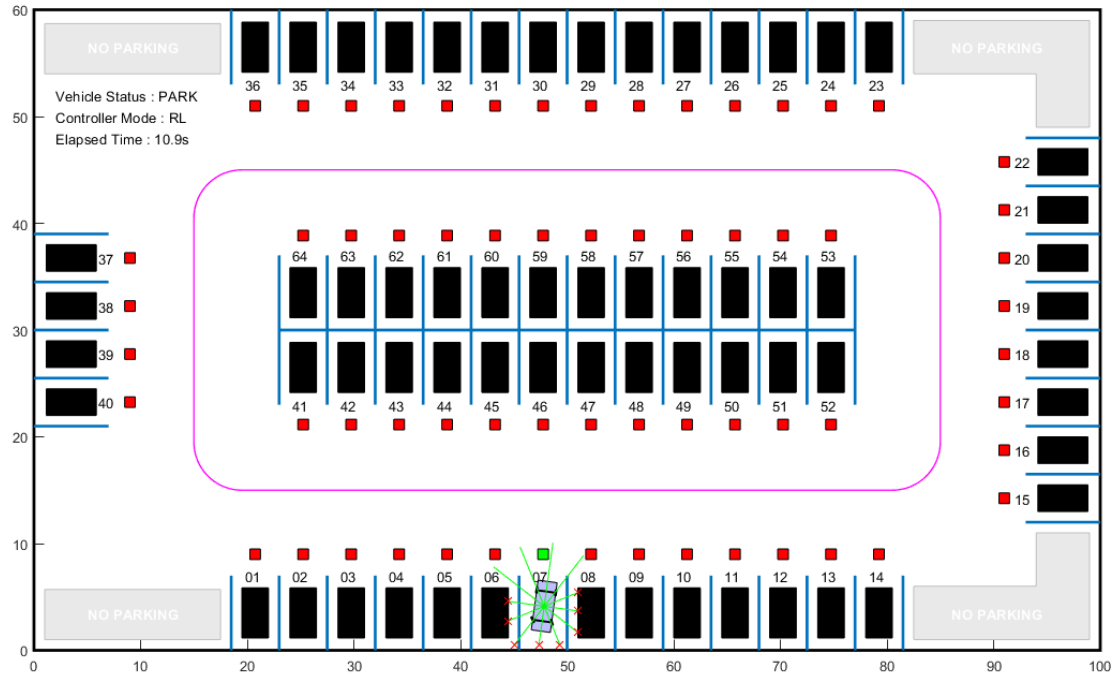
The vehicle reaches the target pose within the specified error tolerances of +/- 0.75 m (position) and +/-10 degrees (orientation).

To view the ego vehicle position and orientation, open the Ego Vehicle Pose scope.
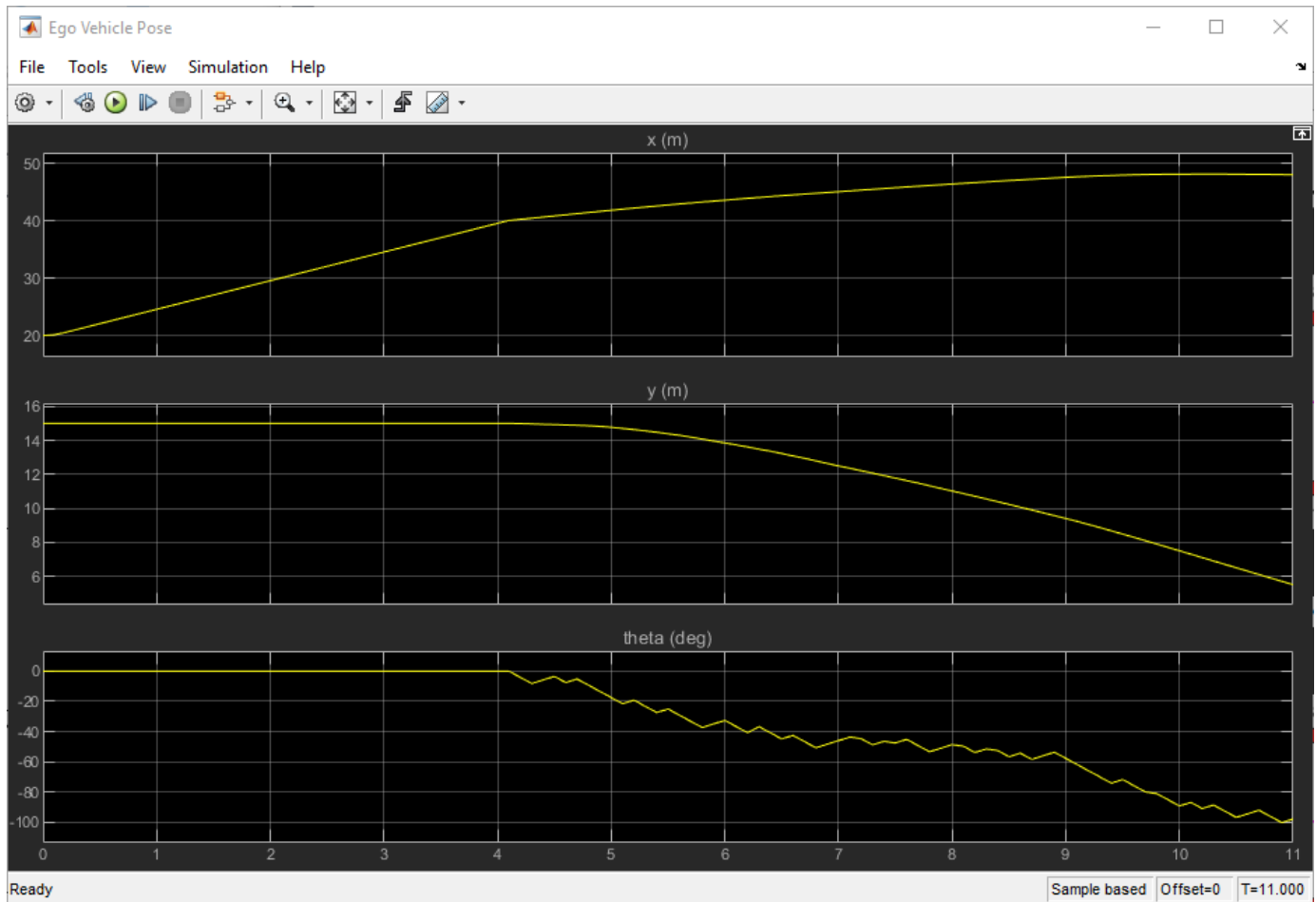
See Also
train

More About
- "Train Reinforcement Learning Agents" on page 5-2
- "Create Policy and Value Function Representations" on page 4-2

Train DDPG Agent for Path-Following Control

This example shows how to train a deep deterministic policy gradient (DDPG) agent for path-following control (PFC) in Simulink®. For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17.

Simulink Model

The reinforcement learning environment for this example is a simple bicycle model for the ego car and a simple longitudinal model for the 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, and also while keeping the ego car travelling along the centerline of its lane by controlling the front steering angle. For more information on PFC, see Path Following Control System (Model Predictive Control Toolbox). The ego car dynamics are specified by the following parameters.

```
m = 1600;    % total vehicle mass (kg)
Iz = 2875;   % yaw moment of inertia (mNs^2)
lf = 1.4;    % longitudinal distance from center of gravity to front tires (m)
lr = 1.6;    % longitudinal distance from center of gravity to rear tires (m)
Cf = 19000;  % cornering stiffness of front tires (N/rad)
Cr = 33000;  % cornering stiffness of rear tires (N/rad)
tau = 0.5;   % longitudinal time constant
```

Specify the initial position and velocity for the two vehicles.

```
x0_lead = 50;   % initial position for lead car (m)
v0_lead = 24;   % initial velocity for lead car (m/s)
x0_ego = 10;    % initial position for ego car (m)
v0_ego = 18;    % initial velocity for ego car (m/s)
```

Specify the standstill default spacing (m), time gap (s), and driver-set velocity (m/s).

```
D_default = 10;
t_gap = 1.4;
v_set = 28;
```

To simulate the physical limitations of the vehicle dynamics, constrain the acceleration to the range [−3,2] (m/s^2), and steering angle is constrained to be [−0.5,0.5] (rad).

```
amin_ego = -3;
amax_ego = 2;
umin_ego = -0.5;
umax_ego = 0.5;
```

The curvature of the road is defined by a constant 0.001 (m^{-1}). The initial value for lateral deviation is 0.2 m and the initial value for the relative yaw angle is –0.1 rad.

```
rho = 0.001;
e1_initial = 0.2;
e2_initial = -0.1;
```

Define the sample time Ts and simulation duration Tf in seconds.

```
Ts = 0.1;
Tf = 60;
```

Open the model.

```
mdl = 'rlPFCMdl';
open_system(mdl)
agentblk = [mdl '/RL Agent'];
```
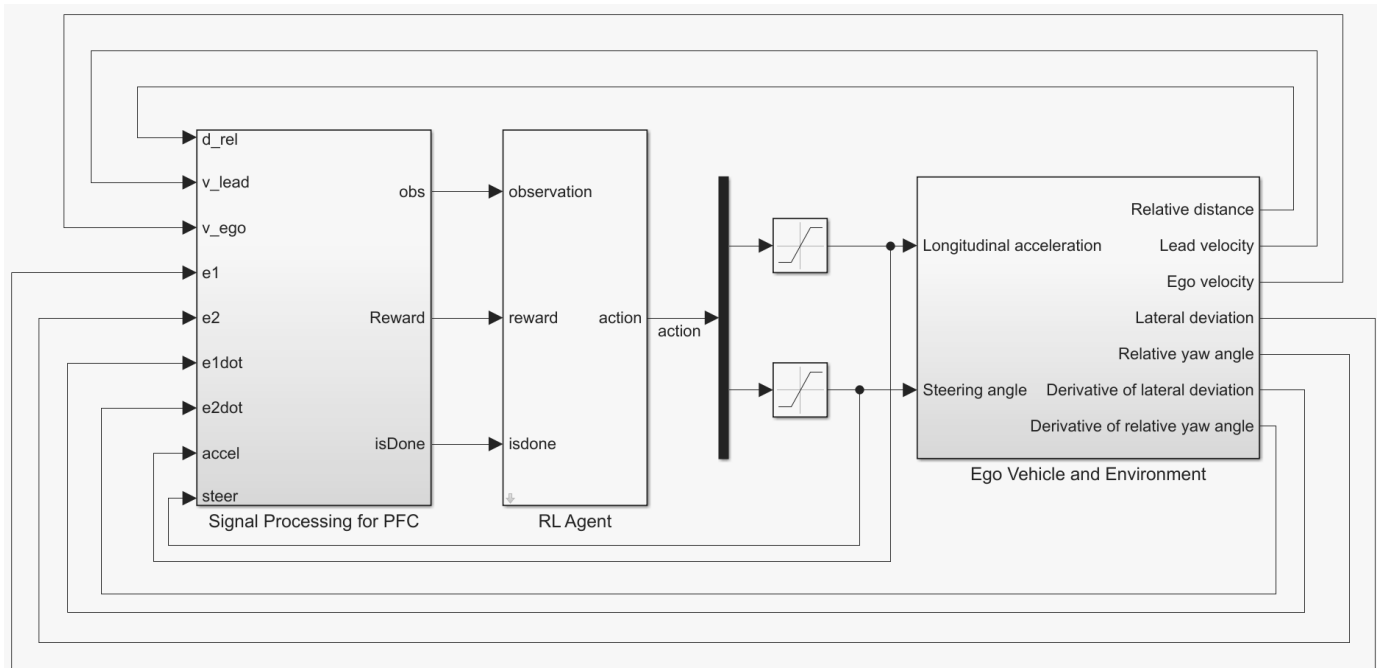

Copyright 2018 The MathWorks, Inc.

For this model:

- The action signal consists of acceleration and steering angle actions. The acceleration action signal takes value between –3 and 2 (m/s^2). The steering action signal takes a value between –15 degrees (–0.2618 rad) and 15 degrees (0.2618 rad).

- The reference velocity for the ego car V_{ref} is defined as follows. If the relative distance is less than the safe distance, the ego car tracks the minimum of the lead car velocity and driver-set velocity. In this manner, the ego car maintains some distance from the lead car. If the relative distance is greater than the safe distance, the ego car tracks the driver-set velocity. In this example, the safe distance is defined as a linear function of the ego car longitudinal velocity V, that is, $t_{gap}*V + D_{default}$. The safe distance determines the tracking velocity for the ego car.

- The observations from the environment contain the longitudinal measurements: the velocity error $e_V = V_{ref} - V_{ego}$, its integral $\int e$, and the ego car longitudinal velocity V. In addition, the observations contain the lateral measurements: the lateral deviation e_1, relative yaw angle e_2, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.

- The simulation terminates when the lateral deviation $|e_1| > 1$, when the longitudinal velocity $V_{ego} < 0.5$, or when the relative distance between the lead car and ego car $D_{rel} < 0$.

- The reward r_t, provided at every time step t, is

$$r_t = -(100e_1^2 + 500u_{t-1}^2 + 10e_V^2 + 100a_{t-1}^2) \times 1e^{-3} - 10F_t + 2H_t + M_t$$

where u_{t-1} is the steering input from the previous time step $t-1$, a_{t-1} is the acceleration input from the previous time step. The three logical values are as follows.

- $F_t = 1$ if simulation is terminated, otherwise $F_t = 0$
- $H_t = 1$ if lateral error $e_1^2 < 0.01$, otherwise $H_t = 0$
- $M_t = 1$ if velocity error $e_V^2 < 1$, otherwise $M_t = 0$

The three logical terms in the reward encourage the agent to make both lateral error and velocity error small, and in the meantime, penalize the agent if the simulation is terminated early.

Create Environment Interface

Create an environment interface for the Simulink model.

Create the observation specification.

```
observationInfo = rlNumericSpec([9 1],'LowerLimit',-inf*ones(9,1),'UpperLimit',inf*ones(9,1));
observationInfo.Name = 'observations';
```

Create the action specification.

```
actionInfo = rlNumericSpec([2 1],'LowerLimit',[-3;-0.2618],'UpperLimit',[2;0.2618]);
actionInfo.Name = 'accel;steer';
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

To define the initial conditions, specify an environment reset function using an anonymous function handle. The reset function `localResetFcn`, which is defined at the end of the example, randomizes the initial position of the lead car, the lateral deviation, and the relative yaw angle.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG Agent

A DDPG agent approximates the long-term reward given observations and actions by using a critic value function representation. To create the critic, first create a deep neural network with two inputs, the state and action, and one output. For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
L = 100; % number of neurons
statePath = [
    featureInputLayer(9,'Normalization','none','Name','observation')
    fullyConnectedLayer(L,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(L,'Name','fc2')
    additionLayer(2,'Name','add')
    reluLayer('Name','relu2')
    fullyConnectedLayer(L,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')];

actionPath = [
    featureInputLayer(2,'Normalization','none','Name','action')
```

```
    fullyConnectedLayer(L,'Name','fc5')];

criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork,actionPath);

criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');
```

View the critic network configuration.

```
figure
plot(criticNetwork)
```


Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1,'L2Regularization
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
```

A DDPG agent decides which action to take given observations by using an actor representation. To create the actor, first create a deep neural network with one input, the observation, and one output, the action.

Construct the actor similarly to the critic. For more information, see `rlDeterministicActorRepresentation`.

```
actorNetwork = [
    featureInputLayer(9,'Normalization','none','Name','observation')
    fullyConnectedLayer(L,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(L,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(L,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(2,'Name','fc4')
    tanhLayer('Name','tanh1')
    scalingLayer('Name','ActorScaling1','Scale',[2.5;0.2618],'Bias',[-0.5;0])];
actorOptions = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1,'L2RegularizationFa
actor = rlDeterministicActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'ActorScaling1'},actorOptions);
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',64);
agentOptions.NoiseOptions.Variance = [0.6;0.1];
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Then, create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent`.

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

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run each training episode for at most `10000` episodes, with each episode lasting at most `maxsteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Verbose` and `Plots` options).
- Stop training when the agent receives an cumulative episode reward greater than `1700`.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 1e4;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue',1700);
```

Train the agent using the `train` function. Training is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
```

```
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load a pretrained agent for the example.
    load('SimulinkPFCDDPG.mat','agent')
end
```
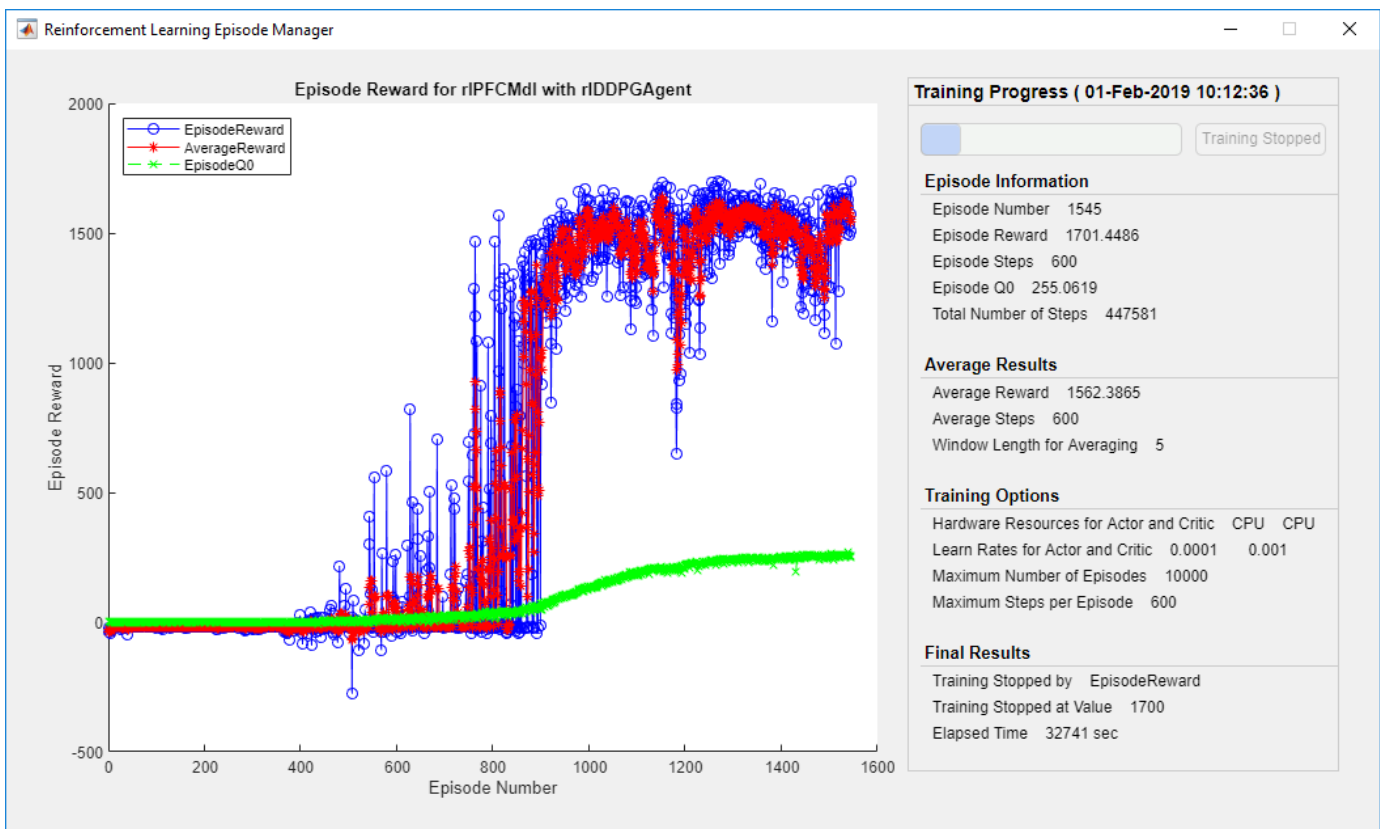

Simulate DDPG Agent

To validate the performance of the trained agent, simulate the agent within the Simulink environment by uncommenting the following commands. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.1;
```

```
x0_lead = 80;
sim(mdl)
```

The following plots show the simulation results when the lead car is 70 (m) ahead of the ego car.

- In the first 35 seconds, the relative distance is greater than the safe distance (bottom-right plot), so the ego car tracks the set velocity (top-right plot). To speed up and reach the set velocity, the acceleration is mostly nonnegative (top-left plot).

- From 35 to 42 seconds, the relative distance is mostly less than the safe distance (bottom-right plot), so the ego car tracks the minimum of the lead velocity and set velocity. Because the lead velocity is less than the set velocity (top-right plot), to track the lead velocity, the acceleration becomes nonzero (top-left plot).

- From 42 to 58 seconds, the ego car tracks the set velocity (top-right plot) and the acceleration remains zero (top-left plot).

- From 58 to 60 seconds, the relative distance becomes less than the safe distance (bottom-right plot), so the ego car slows down and tracks the lead velocity.

- The bottom-left plot shows the lateral deviation. As shown in the plot, the lateral deviation is greatly decreased within 1 second. The lateral deviation remains less than 0.05 m.

Close the Simulink model.

```
bdclose(mdl)
```
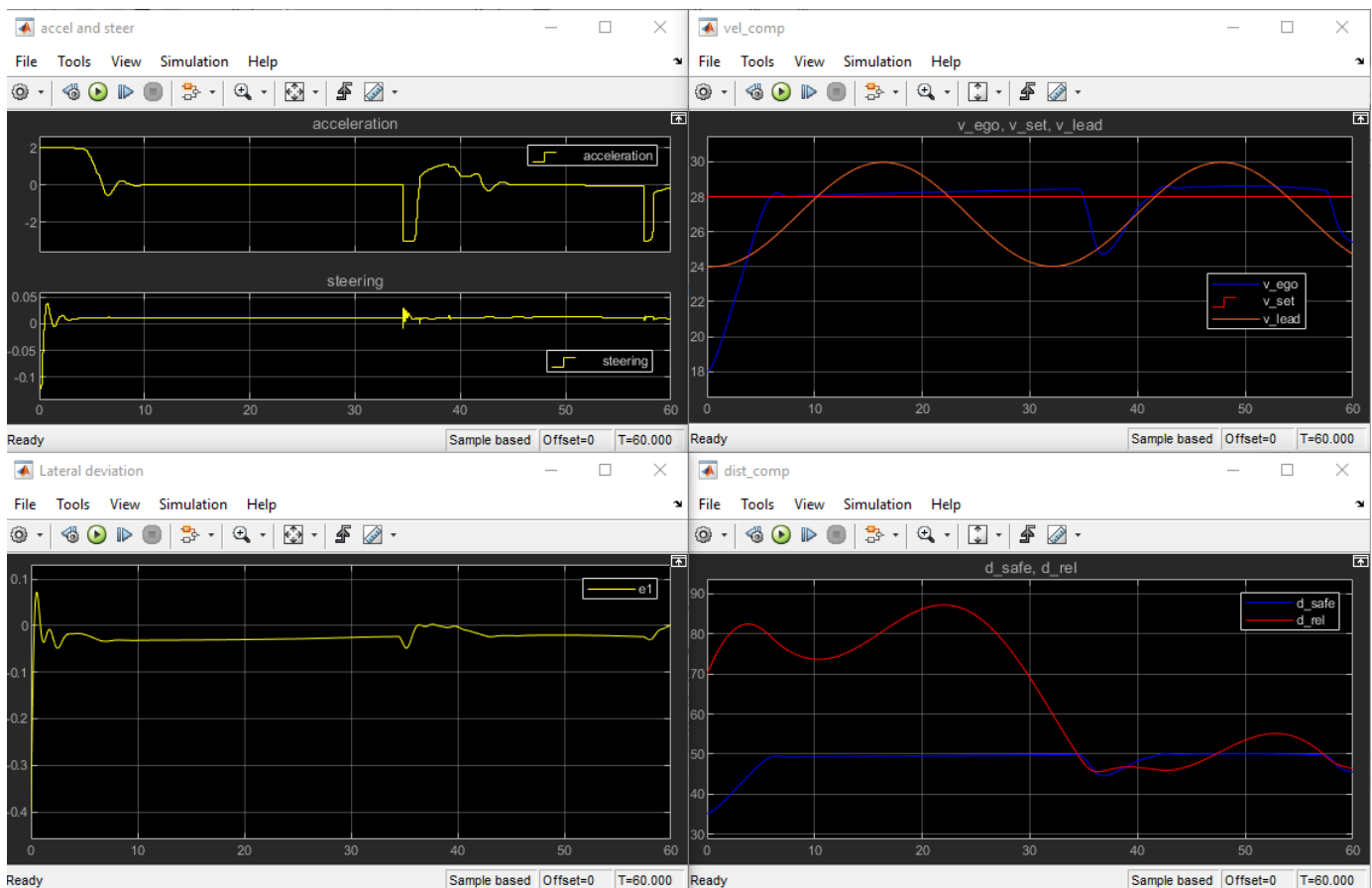
Reset Function

```
function in = localResetFcn(in)
in = setVariable(in,'x0_lead',40+randi(60,1,1));    % random value for initial position of lead
in = setVariable(in,'e1_initial', 0.5*(-1+2*rand)); % random value for lateral deviation
in = setVariable(in,'e2_initial', 0.1*(-1+2*rand)); % random value for relative yaw angle
end
```

See Also

```
train
```

More About

- "Train Reinforcement Learning Agents" on page 5-2
- "Create Policy and Value Function Representations" on page 4-2

Train DQN Agent for Lane Keeping Assist Using Parallel Computing

This example shows how to train a deep Q-learning network (DQN) agent for lane keeping assist (LKA) in Simulink® using parallel training. For an example that shows how to train the agent without using parallel training, see "Train DQN Agent for Lane Keeping Assist" on page 5-134.

For more information on DQN agents, see "Deep Q-Network Agents" on page 3-10. For an example that trains a DQN agent in MATLAB®, see "Train DQN Agent to Balance Cart-Pole System" on page 5-8.

DQN Parallel Training Overview

In a DQN agent, each worker generates new experiences from its copy of the agent and the environment. After every **N** steps, the worker sends experiences to the host agent. The host agent updates its parameters as follows.

- For asynchronous training, the host agent learns from received experiences without waiting for all workers to send experiences, and sends the updated parameters back to the worker that provided the experiences. Then, the worker continues to generate experiences from its environment using the updated parameters.

- For synchronous training, the host agent waits to receive experiences from all of the workers and learns from these experiences. The host then sends updated parameters to all the workers at the same time. Then, all workers continue to generate experiences using the updated parameters.

Simulink Model for Ego Car

The reinforcement learning environment for this example is a simple bicycle model for ego vehicle dynamics. The training goal is to keep the ego vehicle traveling along the centerline of the lanes by adjusting the front steering angle. This example uses the same vehicle model as "Train DQN Agent for Lane Keeping Assist" on page 5-134.

```
m = 1575;   % total vehicle mass (kg)
Iz = 2875;  % yaw moment of inertia (mNs^2)
lf = 1.2;   % longitudinal distance from center of gravity to front tires (m)
lr = 1.6;   % longitudinal distance from center of gravity to rear tires (m)
Cf = 19000; % cornering stiffness of front tires (N/rad)
Cr = 33000; % cornering stiffness of rear tires (N/rad)
Vx = 15;    % longitudinal velocity (m/s)
```

Define the sample time `Ts` and simulation duration `T` in seconds.

```
Ts = 0.1;
T = 15;
```

The output of the LKA system is the front steering angle of the ego car. To simulate the physical steering limits of the ego car, constrain the steering angle to the range $[-0.5, 0.5]$ rad.

```
u_min = -0.5;
u_max = 0.5;
```

The curvature of the road is defined by a constant 0.001 (m^{-1}). The initial value for the lateral deviation is `0.2` m and the initial value for the relative yaw angle is -0.1 rad.

```
rho = 0.001;
e1_initial = 0.2;
e2_initial = -0.1;
```

Open the model.

```
mdl = 'rlLKAMdl';
open_system(mdl)
agentblk = [mdl '/RL Agent'];
```


Copyright 2018 The MathWorks, Inc.

For this model:

- The steering-angle action signal from the agent to the environment is from –15 degrees to 15 degrees.

- The observations from the environment are the lateral deviation e_1, relative yaw angle e_2, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.

- The simulation is terminated when the lateral deviation $|e_1| > 1$.

- The reward r_t, provided at every time step t, is

$$r_t = -(10e_1^2 + 5e_2^2 + 2u^2 + 5\dot{e}_1^2 + 5\dot{e}_2^2)$$

where u is the control input from the previous time step $t - 1$.

Create Environment Interface

Create a reinforcement learning environment interface for the ego vehicle.

Define the observation information.

```
observationInfo = rlNumericSpec([6 1],'LowerLimit',-inf*ones(6,1),'UpperLimit',inf*ones(6,1));
observationInfo.Name = 'observations';
observationInfo.Description = 'information on lateral deviation and relative yaw angle';
```

Define the action information.

```
actionInfo = rlFiniteSetSpec((-15:15)*pi/180);
actionInfo.Name = 'steering';
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

The interface has a discrete action space where the agent can apply one of 31 possible steering angles from –15 degrees to 15 degrees. The observation is the six-dimensional vector containing lateral deviation, relative yaw angle, as well as their derivatives and integrals with respect to time.

To define the initial condition for the lateral deviation and relative yaw angle, specify an environment reset function using an anonymous function handle. `localResetFcn`, which is defined at the end of this example, randomizes the initial lateral deviation and relative yaw angle.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DQN Agent

DQN agents can use multi-output Q-value critic approximators, which are generally more efficient. A multi-output approximator has observations as inputs and state-action values as outputs. Each output element represents the expected cumulative long-term reward for taking the corresponding discrete action from the state indicated by the observation inputs.

To create the critic, first create a deep neural network with one input (the six-dimensional observed state) and one output vector with 31 elements (evenly spaced steering angles from -15 to 15 degrees). For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
nI = observationInfo.Dimension(1);  % number of inputs (6)
nL = 120;                           % number of neurons
nO = numel(actionInfo.Elements);    % number of outputs (31)

dnn = [
    featureInputLayer(nI,'Normalization','none','Name','state')
    fullyConnectedLayer(nL,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(nL,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(nO,'Name','fc3')];
```

View the network configuration.

```
figure
plot(layerGraph(dnn))
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1,'L2Regularization
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation`.

```
critic = rlQValueRepresentation(dnn,observationInfo,actionInfo,'Observation',{'state'},criticOpt
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions`.

```
agentOpts = rlDQNAgentOptions(...
    'SampleTime',Ts,...
    'UseDoubleDQN',true,...
    'TargetSmoothFactor',1e-3,...
    'DiscountFactor',0.99,...
    'ExperienceBufferLength',1e6,...
    'MiniBatchSize',256);
```

```
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 1e-4;
```

Then create the DQN agent using the specified critic representation and agent options. For more information, see `rlDQNAgent`.

```
agent = rlDQNAgent(critic,agentOpts);
```

Training Options

To train the agent, first specify the training options. For this example, use the following options.

- Run each training for at most `10000` episodes, with each episode lasting at most `ceil(T/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box only (set the `Plots` and `Verbose` options accordingly).
- Stop training when the episode reward reaches `-1`.
- Save a copy of the agent for each episode where the cumulative reward is greater than 100.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 10000;
maxsteps = ceil(T/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes, ...
    'MaxStepsPerEpisode',maxsteps, ...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue', -1,...
    'SaveAgentCriteria','EpisodeReward',...
    'SaveAgentValue',100);
```

Parallel Training Options

To train the agent in parallel, specify the following training options.

- Set the `UseParallel` option to `true`.
- Train agent in parallel asynchronously by setting the `ParallelizationOptions.Mode` option to `"async"`.
- After every 30 steps, each worker sends experiences to the host.
- DQN agent requires workers to send `"experiences"` to the host.

```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = "async";
trainOpts.ParallelizationOptions.DataToSendFromWorkers = "experiences";
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 32;
```

For more information, see `rlTrainingOptions`.

Train Agent

Train the agent using the `train` function. Training the agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`. Due to randomness of the parallel training, you can expect different training results from the plot below. The plot shows the result of training with four workers.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
```

```
else
    % Load pretrained agent for the example.
    load('SimulinkLKADQNParallel.mat','agent')
end
```


Simulate DQN Agent

To validate the performance of the trained agent, uncomment the following two lines and simulate the agent within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.2;
sim(mdl)
```

As shown below, the lateral error (middle plot) and relative yaw angle (bottom plot) are both driven to zero. The vehicle starts from off centerline (–0.4 m) and nonzero yaw angle error (0.2 rad). The LKA enables the ego car to travel along the centerline after 2.5 seconds. The steering angle (top plot) shows that the controller reaches steady state after 2 seconds.
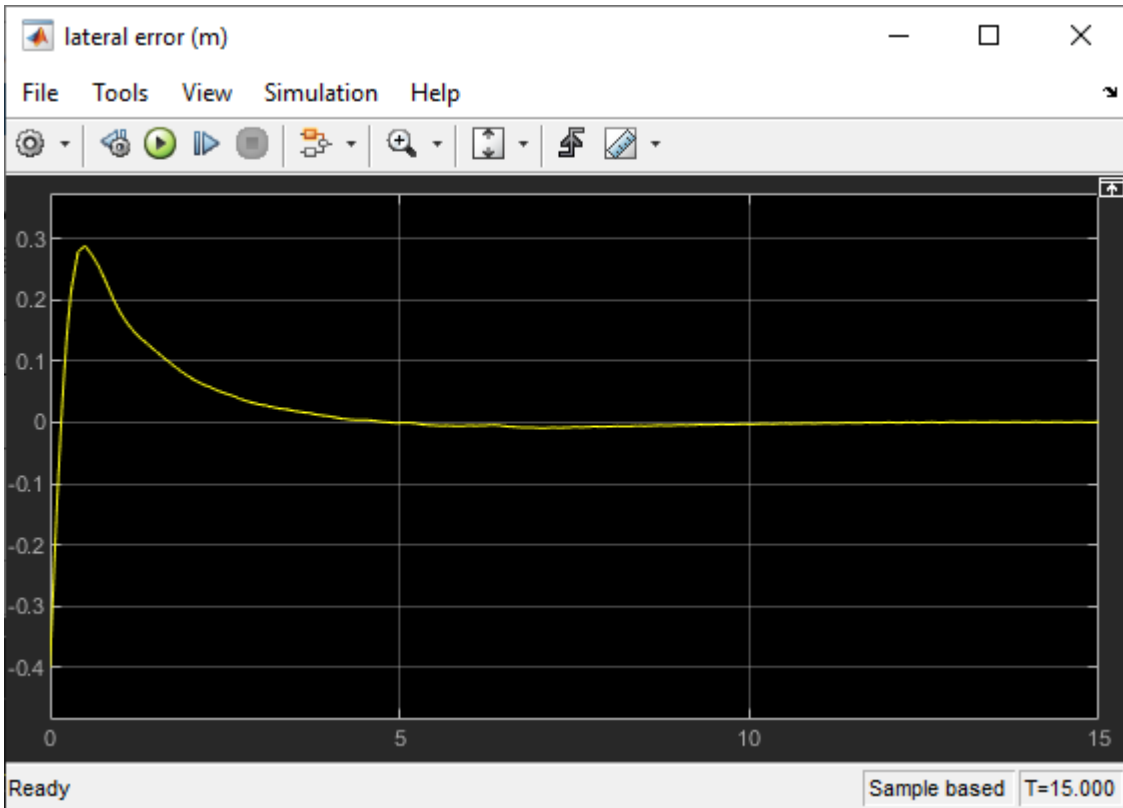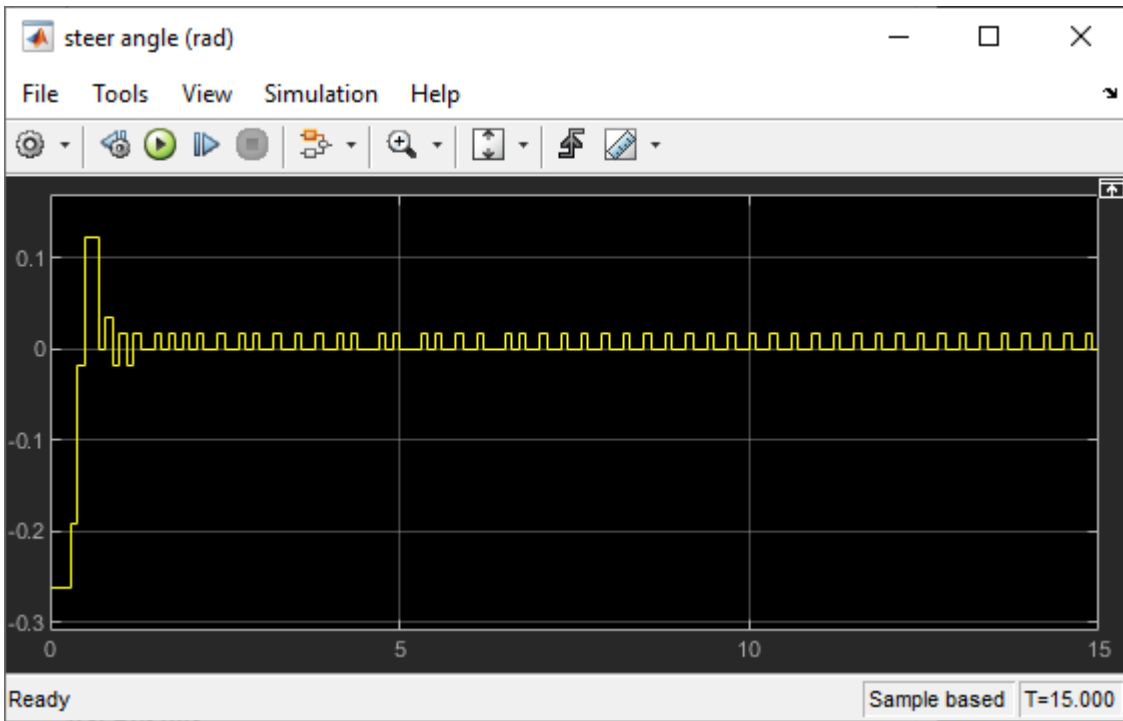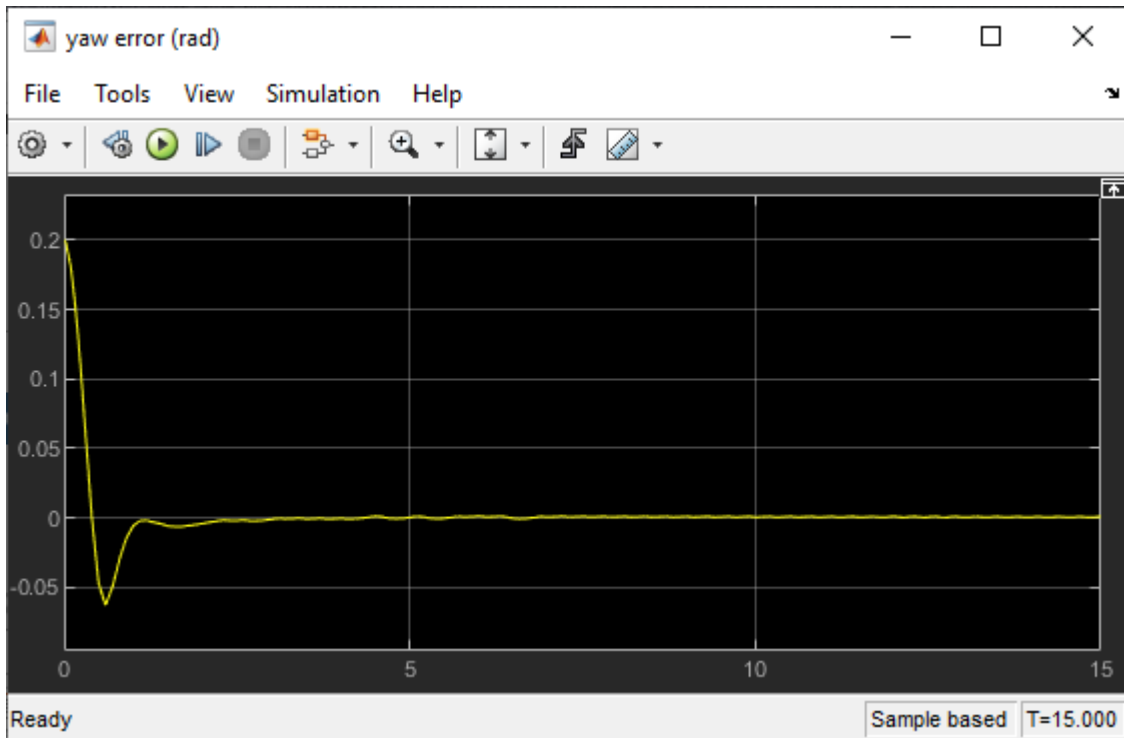
Local Function

```
function in = localResetFcn(in)
% reset
in = setVariable(in,'e1_initial', 0.5*(-1+2*rand)); % random value for lateral deviation
in = setVariable(in,'e2_initial', 0.1*(-1+2*rand)); % random value for relative yaw angle
end
```

See Also

`train`

More About

- "Train DQN Agent for Lane Keeping Assist" on page 5-134
- "Train Reinforcement Learning Agents" on page 5-2
- "Create Policy and Value Function Representations" on page 4-2

Train Biped Robot to Walk Using Reinforcement Learning Agents

This example shows how to train a biped robot to walk using both a deep deterministic policy gradient (DDPG) agent and a twin-delayed deep deterministic policy gradient (TD3) agent and compares the performance of these trained agents. The robot in this example is modeled in Simscape™ Multibody™.

For more information on these agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17 and "Twin-Delayed Deep Deterministic Policy Gradient Agents" on page 3-21.

For the purpose of comparison in this example, both agents are trained on the biped robot environment with same model parameters. The agents are also configured to have the following settings in common.

- Initial condition strategy of the biped robot
- Network structure of actor and critic, inspired by [2]
- Options for actor and critic representations
- Training options (sample time, discount factor, mini-batch size, experience buffer length, exploration noise)

Biped Robot Model

The reinforcement learning environment for this example is a biped robot. The training goal is to make the robot walk in a straight line using minimal control effort.
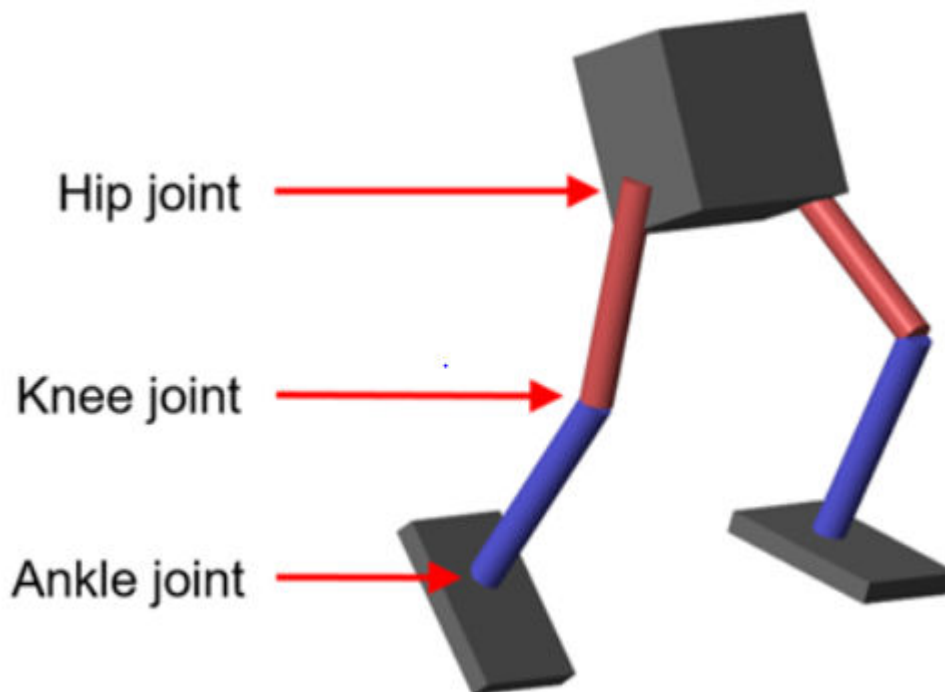
Load the parameters of the model into the MATLAB® workspace.

```
robotParametersRL
```

Open the Simulink model.

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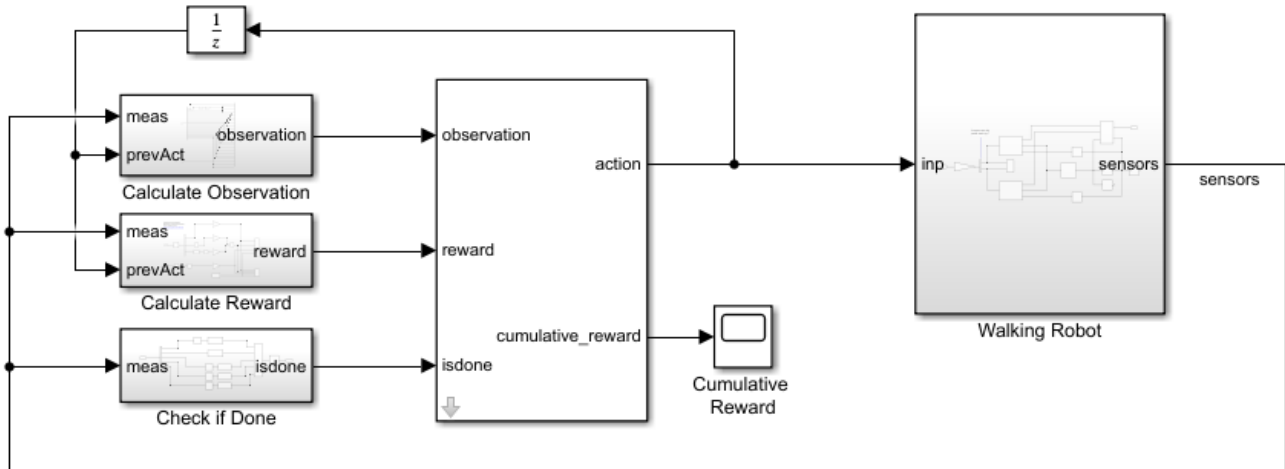

The robot is modeled using Simscape Multibody.
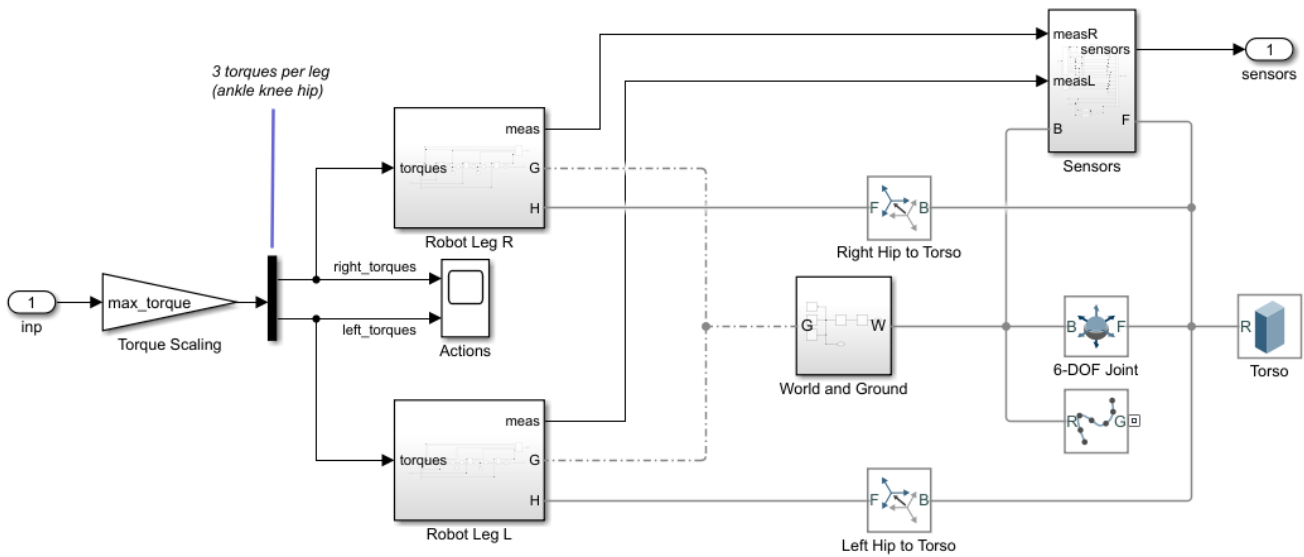
For this model:

- In the neutral 0 rad position, both of the legs are straight and the ankles are flat.

- The foot contact is modeled using the Spatial Contact Force (Simscape Multibody) block from Simscape Multibody.
- The agent can control 3 individual joints (ankle, knee, and hip) on both legs of the robot by applying torque signals from -3 to 3 N·m. The actual computed action signals are normalized between -1 and 1.

The environment provides 29 observations to the agent. The observations are:

- Y (lateral) and Z (vertical) translations of the torso center of mass. The translation in the Z direction is normalized to a similar range as the other observations.
- X (forward), Y (lateral), and Z (vertical) translation velocities.
- Yaw, pitch, and roll angles of the torso.
- Yaw, pitch, and roll angular velocities of the torso.
- Angular positions and velocities of the 3 joints (ankle, knee, hip) on both legs.
- Action values from the previous time step.

The episode terminates if either of the following conditions occur.

- The robot torso center of mass is less than 0.1 m in the Z direction (fallen) or more than 1 m in the Y direction (lateral motion).
- The absolute value of either the roll, pitch, or yaw is greater than 0.7854 rad.

The following reward function r_t, which is provided at every time step is inspired by [1]. This reward function encourages the agent to move forward by providing a positive reward for positive forward velocity. It also encourages the agent to avoid episode termination by providing a constant reward $(25\frac{Ts}{Tf})$ at every time step. The other terms in the reward function are penalties for substantial changes in lateral and vertical translations, and for the use of excess control effort.

$$r_t = v_x - 3y^2 - 50\hat{z}^2 + 25\frac{Ts}{Tf} - 0.02\sum_i u_{t-1}^i{}^2$$

Here:

- v_x is the translation velocity in the X direction (forward toward goal) of the robot.
- y is the lateral translation displacement of the robot from the target straight line trajectory.
- $\hat{z}$ is the normalized vertical translation displacement of the robot center of mass.
- u_{t-1}^i is the torque from joint i from the previous time step.
- Ts is the sample time of the environment.
- Tf is the final simulation time of the environment.

Create Environment Interface

Create the observation specification.

```
numObs = 29;
obsInfo = rlNumericSpec([numObs 1]);
obsInfo.Name = 'observations';
```

Create the action specification.

```
numAct = 6;
actInfo = rlNumericSpec([numAct 1],'LowerLimit',-1,'UpperLimit',1);
actInfo.Name = 'foot_torque';
```

Create the environment interface for the walking robot model.

```
blk = [mdl,'/RL Agent'];
env = rlSimulinkEnv(mdl,blk,obsInfo,actInfo);
env.ResetFcn = @(in) walkerResetFcn(in,upper_leg_length/100,lower_leg_length/100,h/100);
```

Select and Create Agent for Training

This example provides the option to train the robot either using either a DDPG or TD3 agent. To simulate the robot with the agent of your choice, set the `AgentSelection` flag accordingly.

```
AgentSelection = 'TD3';
switch AgentSelection
    case 'DDPG'
        agent = createDDPGAgent(numObs,obsInfo,numAct,actInfo,Ts);
    case 'TD3'
        agent = createTD3Agent(numObs,obsInfo,numAct,actInfo,Ts);
    otherwise
        disp('Enter DDPG or TD3 for AgentSelection')
end
```

The `createDDPGAgent` and `createTD3Agent` helper functions perform the following actions.

- Create actor and critic networks.
- Specify options for actor and critic representations.
- Create actor and critic representations using created networks and specified options.
- Configure agent specific options.
- Create agent.

DDPG Agent

A DDPG agent approximates the long-term reward given observations and actions using a critic value function representation. A DDPG agent decides which action to take given observations by using an actor representation. The actor and critic networks for this example are inspired by [2].

For details on the creating the DDPG agent, see the `createDDPGAgent` helper function. For information on configuring DDPG agent options, see `rlDDPGAgentOptions`.

For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2. For an example that creates neural networks for DDPG agents, see "Train DDPG Agent to Control Double Integrator System" on page 5-31.

TD3 Agent

TD3 agent approximates the long-term reward given observations and actions using 2 critic value function representations. A TD3 agent decides which action to take given observations using an actor representation. The structure of the actor and critic networks used for this agent are the same as the ones used for DDPG agent.

A DDPG agent can overestimate the Q value. Since this Q value is then used to update the policy (actor) of the agent, the resultant policy can be suboptimal and accumulating training errors can lead

to divergent behavior. The TD3 algorithm is an extension of DDPG with improvements that make it more robust by preventing overestimation of Q values [3].

- Two critic networks — TD3 agents learn two critic networks independently and use the minimum value function estimate to update the actor (policy). Doing so prevents accumulation of error in subsequent steps and overestimation of Q values.

- Addition of target policy noise — Clipped noise is added to target actions to smooth out Q function values over similar actions. Doing so prevents learning an incorrect sharp peak of noisy value estimate.

- Delayed policy and target updates — For a TD3 agent it is recommended to delay the actor network update, as it allows more time for the Q function to reduce error (get closer to the required target) before updating the policy. Doing so prevents variance in values estimates and results in a more high quality policy update.

For details on the creating the TD3 agent, see the `createTD3Agent` helper function. For information on configuring TD3 agent options, see `rlTD3AgentOptions`.

Specify Training Options and Train Agent

For this example, the training options for the DDPG and TD3 agents are the same. These options are based on the following requirements.

- Run each training session for 2000 episodes with each episode lasting at most `maxSteps` time steps.

- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option).

- Terminate the training only when it reaches the maximum number of episodes (`maxEpisodes`). Doing so allows the comparison of the learning curves for multiple agents over the entire training session.

For more information and additional options, see `rlTrainingOptions`.

```
maxEpisodes = 2000;
maxSteps = floor(Tf/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxEpisodes,...
    'MaxStepsPerEpisode',maxSteps,...
    'ScoreAveragingWindowLength',250,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeCount',...
    'StopTrainingValue',maxEpisodes,...
    'SaveAgentCriteria','EpisodeCount',...
    'SaveAgentValue',maxEpisodes);
```

To train the agent in parallel, specify the following training options. Training in parallel requires Parallel Computing Toolbox™. If you do not have Parallel Computing Toolbox software installed, set `UseParallel` to `false`.

- Set the `UseParallel` option to `true`.
- Train the agent in parallel asynchronously.
- After every 32 steps, have each worker send experiences to the host. DDPG and TD3 agents require workers to send experiences to the host.

```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = 'async';
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 32;
trainOpts.ParallelizationOptions.DataToSendFromWorkers = 'Experiences';
```

Train the agent using the `train` function. This process is computationally intensive and takes several hours to complete for each agent. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`. Due to randomness in the parallel training, you can expect different training results from the plots below. The pretrained agents were trained in parallel using four workers.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load a pretrained agent for the selected agent type.
    if strcmp(AgentSelection,'DDPG')
        load('rlWalkingBipedRobotDDPG.mat','agent')
    else
        load('rlWalkingBipedRobotTD3.mat','agent')
    end
end
```

Episode Reward for rlWalkingBipedRobot withrlTD3Agent
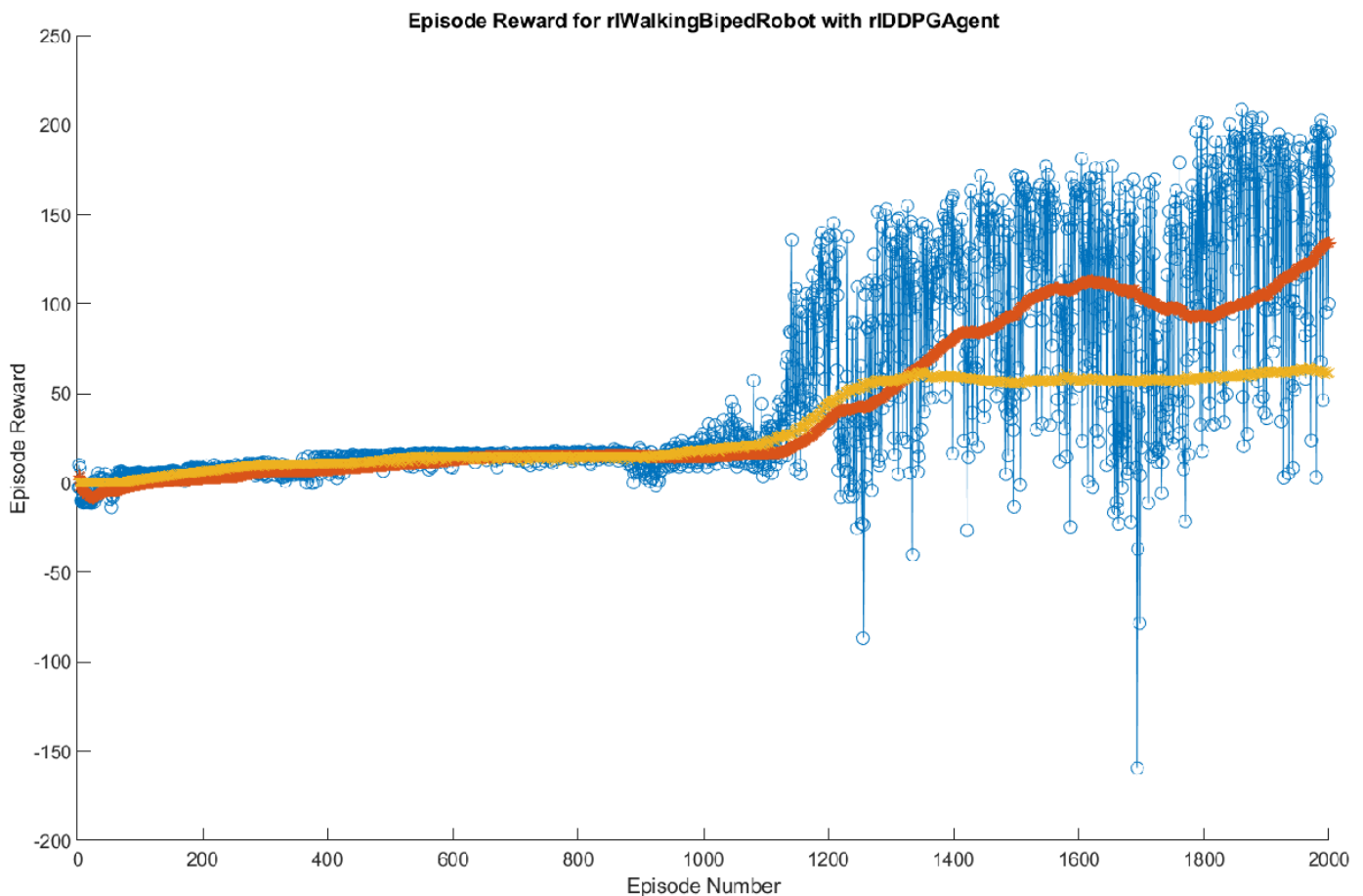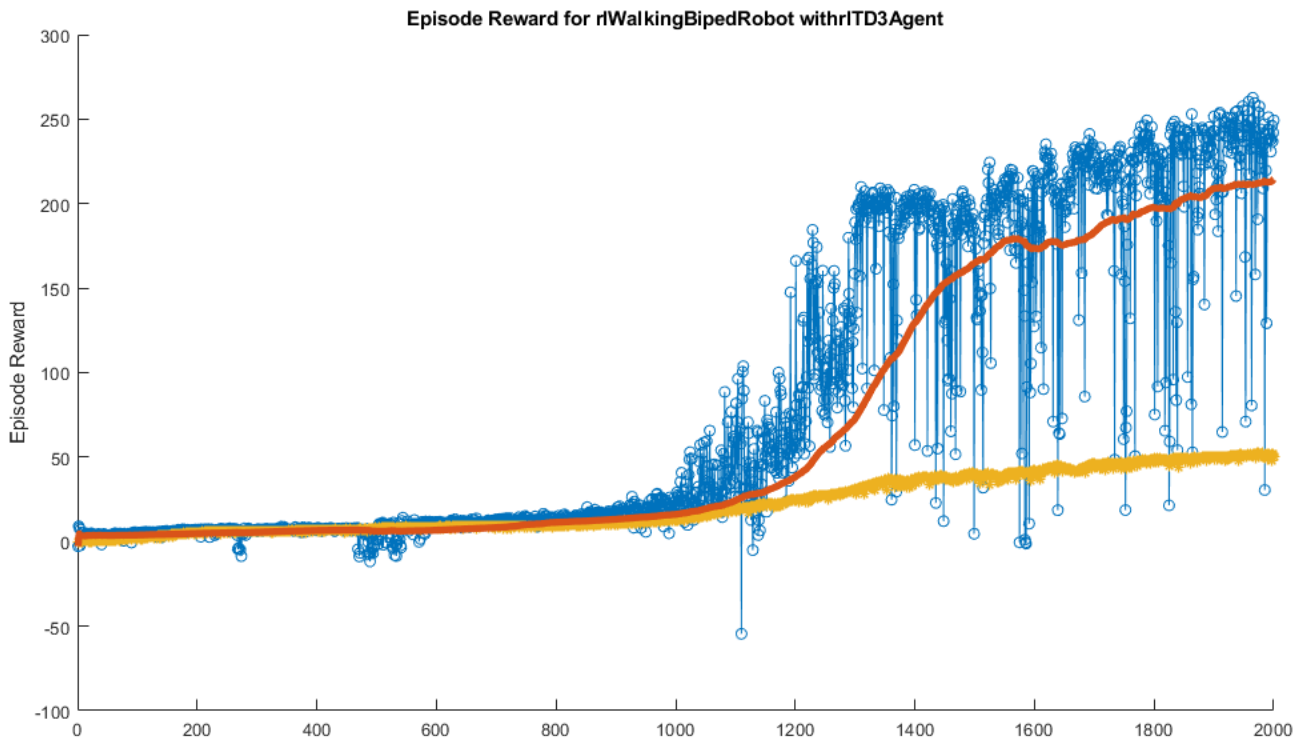
For the preceding example training curves, the average time per training step for the DDPG and TD3 agents are 0.11 and 0.12 seconds, respectively. The TD3 agent takes more training time per step because it updates two critic networks compared to the single critic used for DDPG.

Simulate Trained Agents

Fix the random generator seed for reproducibility.

```
rng(0)
```

To validate the performance of the trained agent, simulate it within the biped robot environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',maxSteps);
experience = sim(env,agent,simOptions);
```
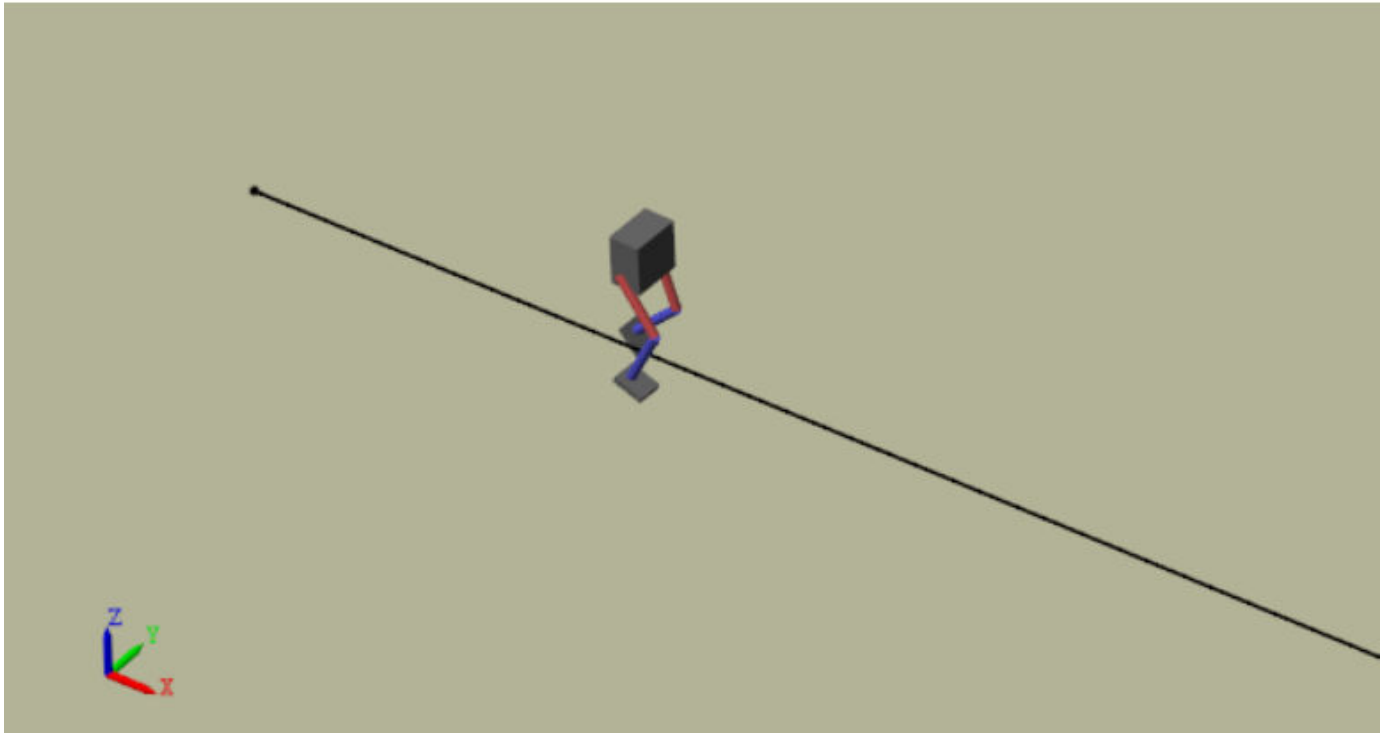
Compare Agent Performance

For the following agent comparison, each agent was trained five times using a different random seed each time. Due to the random exploration noise and the randomness in the parallel training, the learning curve for each run is different. Since the training of agents for multiple runs takes several days to complete, this comparison uses pretrained agents.

For the DDPG and TD3 agents, plot the average and standard deviation of the episode reward (top plot) and the episode Q0 value (bottom plot). The episode Q0 value is the critic estimate of the discounted long-term reward at the start of each episode given the initial observation of the environment. For a well-designed critic, the episode Q0 value approaches the true discounted long-term reward.

```
comparePerformance('DDPGAgent','TD3Agent')
```

EpisodeQ0 comparison
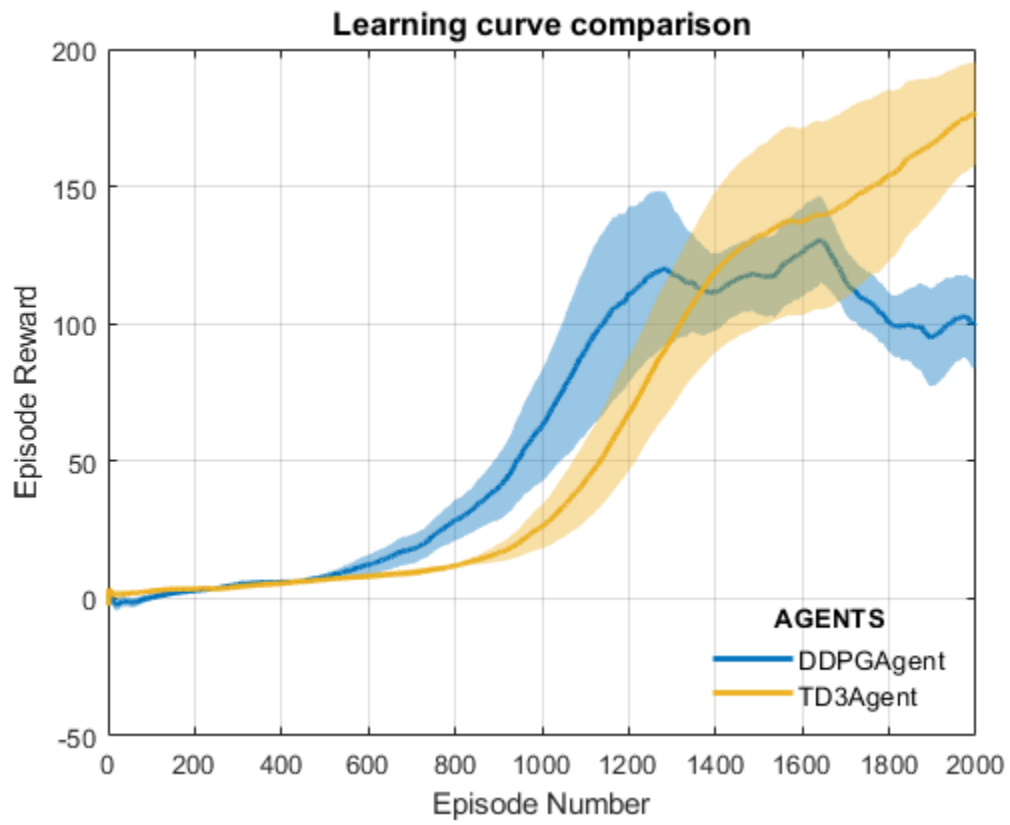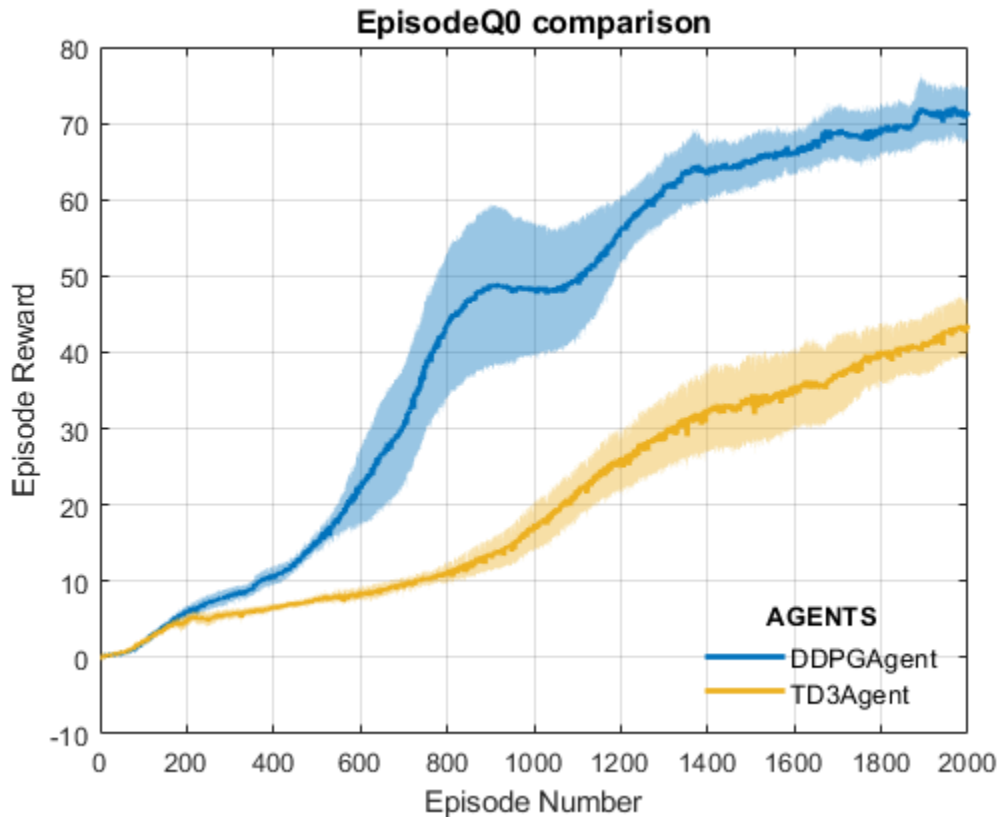
Based on the **Learning curve comparison** plot:

1   The DDPG appears to pick up learning faster (around episode number 600 on an average) but hits a local minimum. TD3 starts slower but eventually achieves higher rewards than DDPG as it avoids overestimation of Q values.

2   The TD3 Agent shows a steady improvement in its learning curve, which suggests improved stability when compared to the DDPG agent.

Based on the **Episode Q0 comparison** plot:

1   For the TD3 agent, the critic estimate of the discounted long term reward (for 2000 episodes) is lower compared to the DDPG agent. This difference is because the TD3 algorithm takes a conservative approach in updating its targets by using minimum of two Q functions. This behavior is further enhanced because of delayed updates to the targets.

2   Although the TD3 estimate for these 2000 episodes is low, the TD3 agent shows a steady increase in the episode Q0 value,s unlike DDPG agent.

In this example the training was stopped at 2000 episodes. For a larger training period, the TD3 agent with its steady increase in estimates shows the potential to converge to the true discounted long-term reward.

For another example on how to train a DDPG agent to walk an humanoid robot modeled in Simscape™ Multibody™ see "Train a Humanoid Walker" (Simscape Multibody). For an example on how to train a DDPG agent to walk a quadruped robot see "Quadruped Robot Locomotion Using DDPG Agent" on page 5-179.

References

[1] Heess, Nicolas, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, et al. 'Emergence of Locomotion Behaviours in Rich Environments'. *ArXiv:1707.02286 [Cs]*, 10 July 2017. https://arxiv.org/abs/1707.02286.

[2] Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 'Continuous Control with Deep Reinforcement Learning'. *ArXiv:1509.02971 [Cs, Stat]*, 5 July 2019. https://arxiv.org/abs/1509.02971.

[3] Fujimoto, Scott, Herke van Hoof, and David Meger. 'Addressing Function Approximation Error in Actor-Critic Methods'. *ArXiv:1802.09477 [Cs, Stat]*, 22 October 2018. https://arxiv.org/abs/1802.09477.

See Also
`train`

More About
- "Reinforcement Learning Agents" on page 3-2
- "Train Reinforcement Learning Agents" on page 5-2
- "Define Reward Signals" on page 2-7

Quadruped Robot Locomotion Using DDPG Agent

This example shows how to train a quadruped robot to walk using a deep deterministic policy gradient (DDPG) agent. The robot in this example is modeled using Simscape™ Multibody™. For more information on DDPG agents, see "Twin-Delayed Deep Deterministic Policy Gradient Agents" on page 3-21.
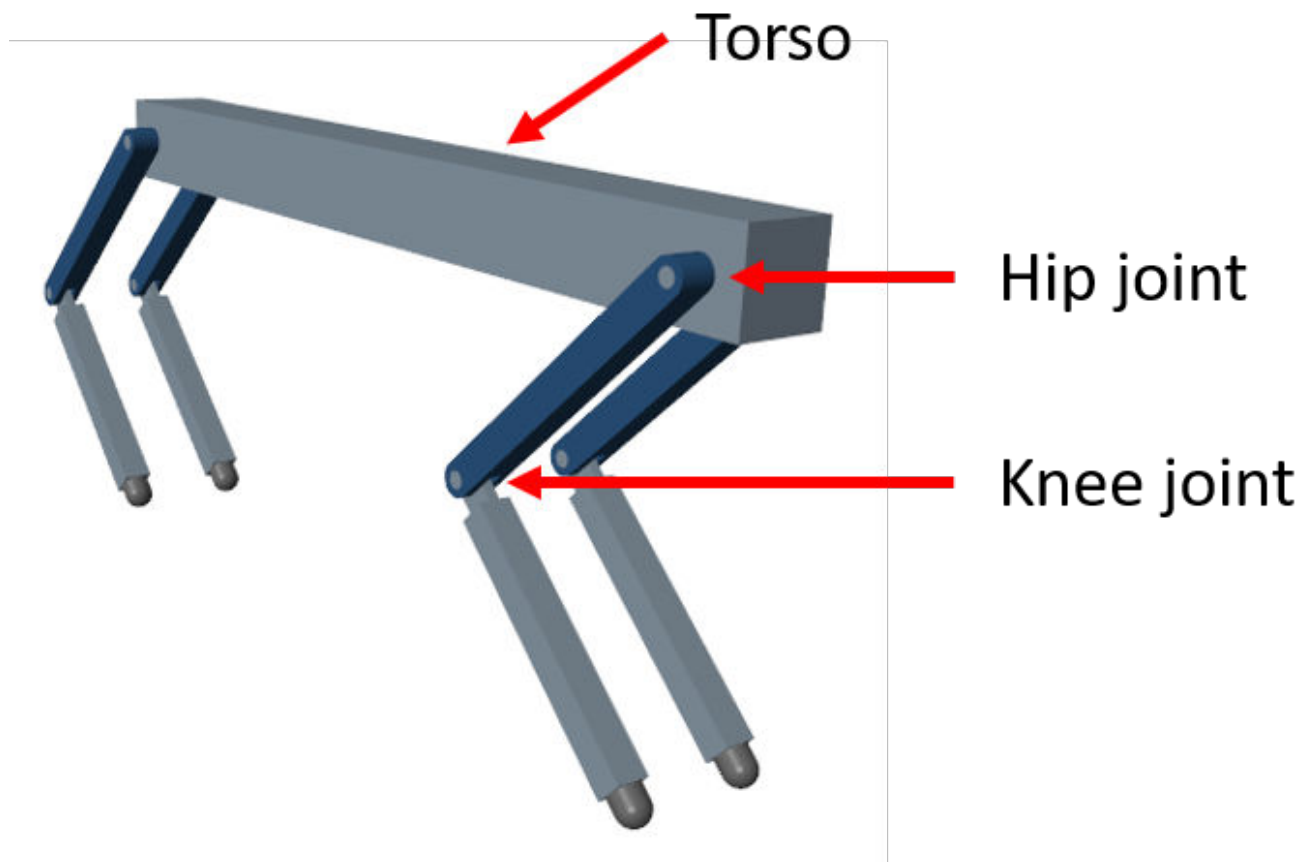
Load the necessary parameters into the base workspace in MATLAB®.

```
initializeRobotParameters
```

Quadruped Robot Model

The environment for this example is a quadruped robot, and the training goal is to make the robot walk in a straight line using minimal control effort.

The robot is modeled using Simscape Multibody and the Simscape Multibody Contact Forces Library. The main structural components are four legs and a torso. The legs are connected to the torso through revolute joints. Action values provided by the RL Agent block are scaled and converted into joint torque values. These joint torque values are used by the revolute joints to compute motion.

Open the model.

```
mdl = 'rlQuadrupedRobot';
open_system(mdl)
```
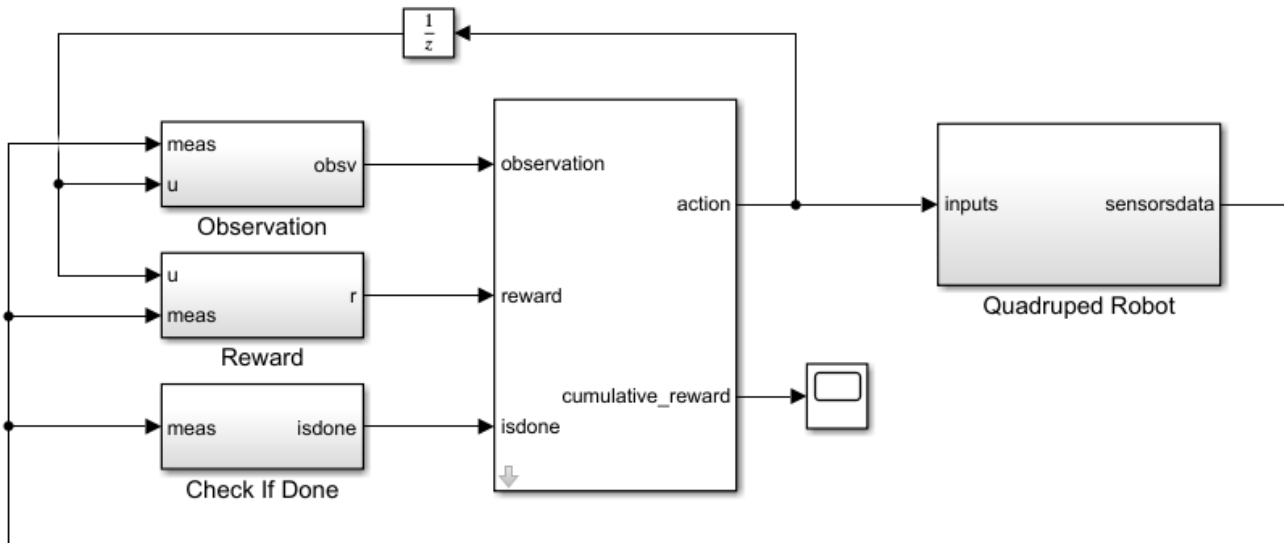
Quadruped Walking Robot Example

Copyright 2019 The MathWorks, Inc.

Observations

The robot environment provides 44 observations to the agent, each normalized between –1 and 1. These observations are:

- Y (vertical) and Y (lateral) position of the torso center of mass
- Quaternion representing the orientation of the torso
- X (forward), Y (vertical), and Z (lateral) velocities of the torso at the center of mass
- Roll, pitch, and yaw rates of the torso
- Angular positions and velocities of the hip and knee joints for each leg
- Normal and friction force due to ground contact for each leg
- Action values (torque for each joint) from the previous time step

For all four legs, the initial values for the hip and knee joint angles are set to –0.8234 and 1.6468 rad, respectively. The neutral positions of the joints are at 0 rad. The legs are in neutral position when they are stretched to their maximum and are aligned perpendicularly to the ground.

Actions

The agent generates eight actions normalized between –1 and 1. After multiplying with a scaling factor, these correspond to the eight joint torque signals for the revolute joints. The overall joint torque bounds are +/– 10 N·m for each joint.

Reward

The following reward is provided to the agent at each time step during training. This reward function encourages the agent to move forward by providing a positive reward for positive forward velocity. It also encourages the agent to avoid early termination by providing a constant reward ($25T_s/T_f$) at each time step. The remaining terms in the reward function are penalties that discourage unwanted states, such as large deviations from the desired height and orientation or the use of excessive joint torques.

$$r_t = v_x + 25\frac{T_s}{T_f} - 50\hat{y}^2 - 20\theta^2 - 0.02\sum_i {u_{t-1}^i}^2$$

where

- v_x is the velocity of the torso's center of mass in the x-direction.
- T_s and T_f are the sample time and final simulation time of the environment, respectively.
- $\hat{y}$ is the scaled height error of the torso's center of mass from the desired height of 0.75 m.
- θ is the pitch angle of the torso.
- u_{t-1}^i is the action value for joint i from the previous time step.

Episode Termination

During training or simulation, the episode terminates if any of the following situations occur.

- The height of the torso center of mass from the ground is below 0.5 m (fallen).
- The head or tail of the torso is below the ground.
- Any knee joint is below the ground.
- Roll, pitch, or yaw angles are outside bounds (+/– 0.1745, +/– 0.1745, and +/– 0.3491 rad, respectively).

Create Environment Interface

Specify the parameters for the observation set.

```
numObs = 44;
obsInfo = rlNumericSpec([numObs 1]);
obsInfo.Name = 'observations';
```

Specify the parameters for the action set.

```
numAct = 8;
actInfo = rlNumericSpec([numAct 1],'LowerLimit',-1,'UpperLimit', 1);
actInfo.Name = 'torque';
```

Create the environment using the reinforcement learning model.

```
blk = [mdl, '/RL Agent'];
env = rlSimulinkEnv(mdl,blk,obsInfo,actInfo);
```

During training, the reset function introduces random deviations into the initial joint angles and angular velocities.

```
env.ResetFcn = @quadrupedResetFcn;
```

Create DDPG agent

The DDPG agent approximates the long-term reward given observations and actions by using a critic value function representation. The agent also decides which action to take given the observations, using an actor representation. The actor and critic networks for this example are inspired by [2].

For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2. For an example that creates neural networks for DDPG agents, see "Train DDPG Agent to Control Double Integrator System" on page 5-31.

Create the networks in the MATLAB workspace using the `createNetworks` helper function.

```
createNetworks
```

You can also create your actor and critic networks interactively using the Deep Network Designer app.

View the critic network configuration.

```
plot(criticNetwork)
```

Specify the agent options using `rlDDPGAgentOptions`.

```
agentOptions = rlDDPGAgentOptions;
agentOptions.SampleTime = Ts;
agentOptions.DiscountFactor = 0.99;
agentOptions.MiniBatchSize = 250;
agentOptions.ExperienceBufferLength = 1e6;
agentOptions.TargetSmoothFactor = 1e-3;
agentOptions.NoiseOptions.MeanAttractionConstant = 0.15;
agentOptions.NoiseOptions.Variance = 0.1;
```

Create the `rlDDPGAgent` object for the agent.

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

Specify Training Options

To train the agent, first specify the following training options:

- Run each training episode for at most 10,000 episodes, with each episode lasting at most `maxSteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option).
- Stop training when the agent receives an average cumulative reward greater than 190 over 250 consecutive episodes.
- Save a copy of the agent for each episode where the cumulative reward is greater than 200.

```
maxEpisodes = 10000;
maxSteps = floor(Tf/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxEpisodes,...
    'MaxStepsPerEpisode',maxSteps,...
    'ScoreAveragingWindowLength',250,...
    'Verbose',true,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',190,...
    'SaveAgentCriteria','EpisodeReward',...
    'SaveAgentValue',200);
```

To train the agent in parallel, specify the following training options. Training in parallel requires Parallel Computing Toolbox™ software. If you do not have Parallel Computing Toolbox™ software installed, set `UseParallel` to `false`.

- Set the `UseParallel` option to `true`.
- Train the agent in parallel asynchronously.
- After every 32 steps, each worker sends experiences to the host.
- DDPG agents require workers to send `'Experiences'` to the host.

```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = 'async';
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 32;
trainOpts.ParallelizationOptions.DataToSendFromWorkers = 'Experiences';
```

Train Agent

Train the agent using the `train` function. Due to the complexity of the robot model, this process is computationally intensive and takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`. Due to the randomness of parallel training, you can expect different training results from the plot below.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load a pretrained agent for the example.
    load('rlQuadrupedAgent.mat','agent')
end
```
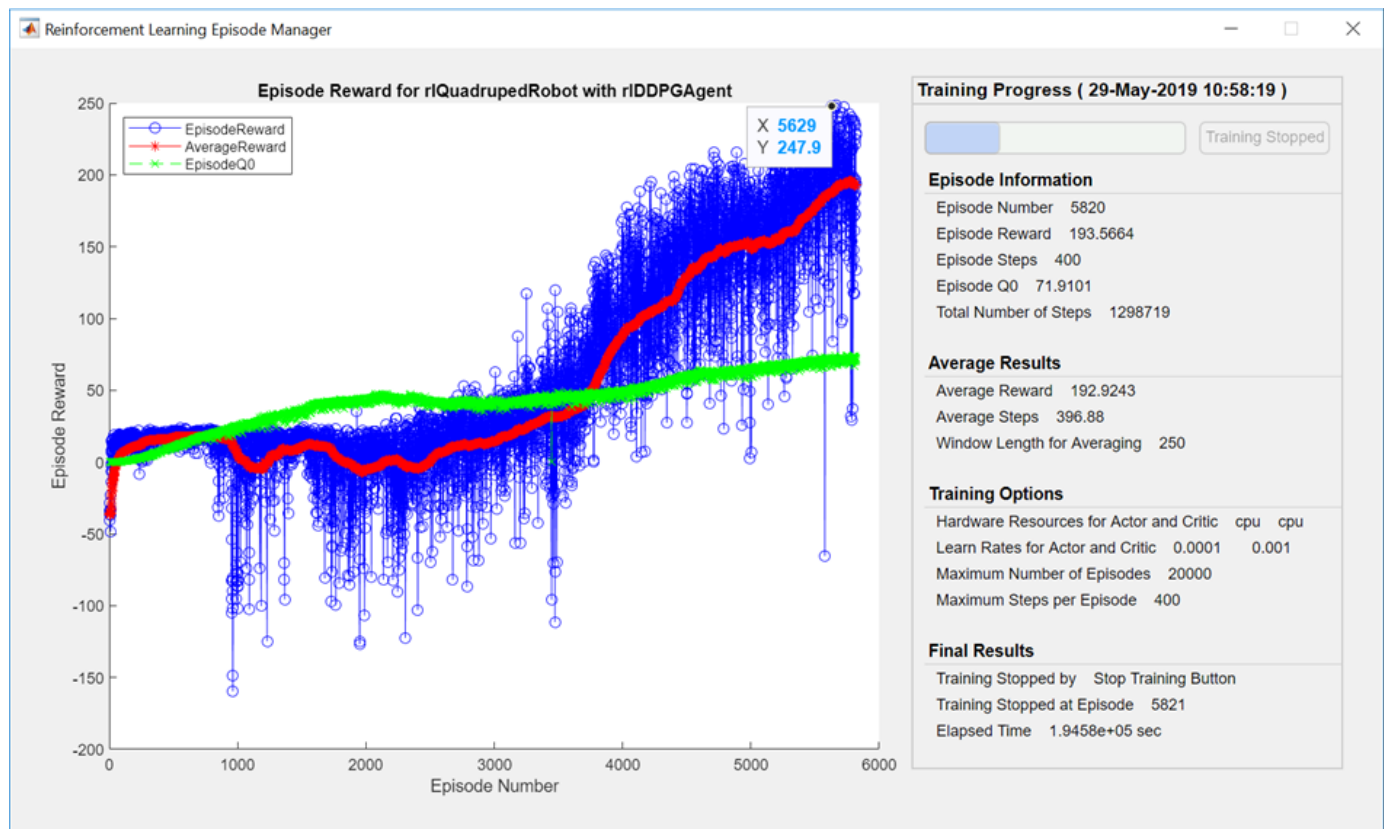
Simulate Trained Agent

Fix the random generator seed for reproducibility.

```
rng(0)
```

To validate the performance of the trained agent, simulate it within the robot environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',maxSteps);
experience = sim(env,agent,simOptions);
```
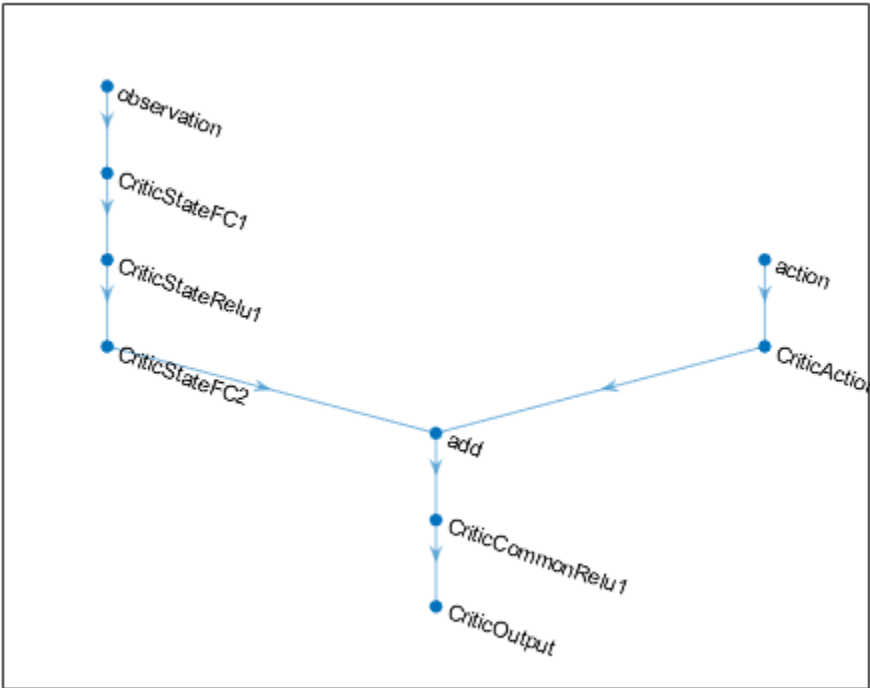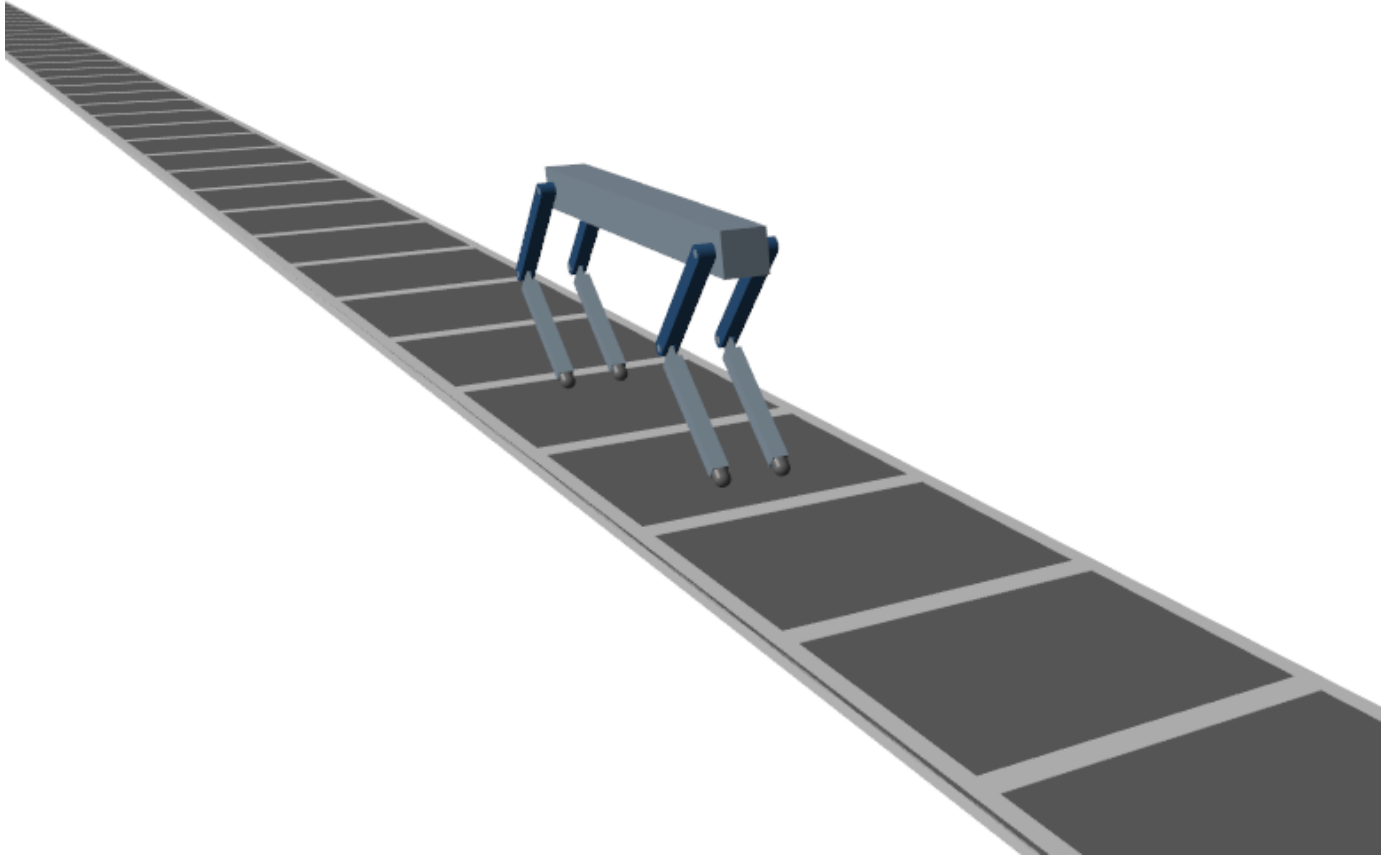
For examples on how to train a DDPG agent to walk a biped robot and a humanoid walker modeled in Simscape™ Multibody™ see "Train Biped Robot to Walk Using Reinforcement Learning Agents" on page 5-168 and "Train a Humanoid Walker" (Simscape Multibody), respectively.

References

[1] Heess, Nicolas, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, et al. 'Emergence of Locomotion Behaviours in Rich Environments'. *ArXiv:1707.02286 [Cs]*, 10 July 2017. https://arxiv.org/abs/1707.02286.

[2] Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 'Continuous Control with Deep Reinforcement Learning'. *ArXiv:1509.02971 [Cs, Stat]*, 5 July 2019. https://arxiv.org/abs/1509.02971.

See Also
train

More About
- "Reinforcement Learning Agents" on page 3-2
- "Train Reinforcement Learning Agents" on page 5-2
- "Define Reward Signals" on page 2-7

Train DDPG Agent for PMSM Control

This example demonstrates speed control of a permanent magnet synchronous motor (PMSM) using a deep deterministic policy gradient (DDPG) agent.

The goal of this example is to show that you can use reinforcement learning as an alternative to linear controllers, such as PID controllers, in speed control of PMSM systems. Outside their regions of linearity, linear controllers often do not produce good tracking performance. In such cases, reinforcement learning provides a nonlinear control alternative.

Load the parameters for this example.

`sim_data`

Open the Simulink model.

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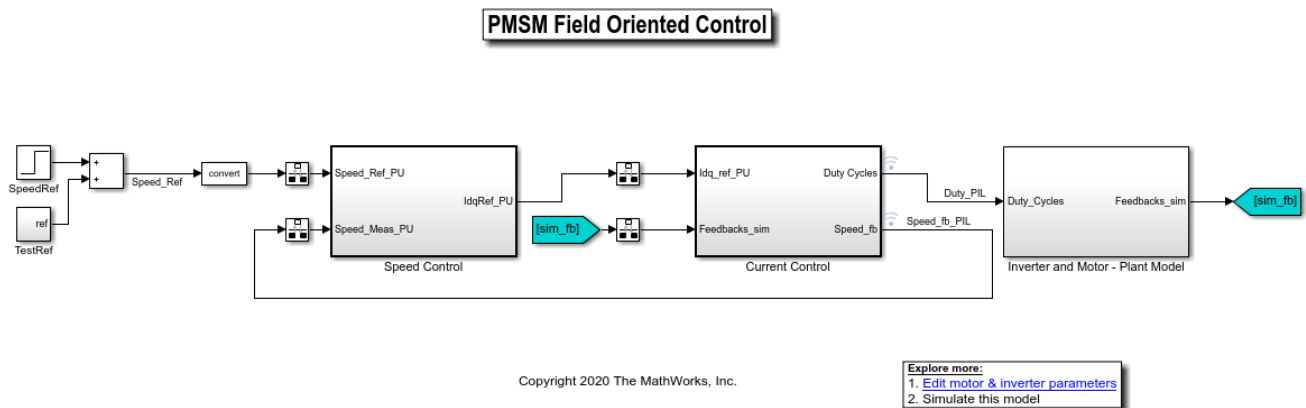

In a linear control version of this example, you can use PI controllers in both the speed and current control loops. An outer-loop PI controller can control the speed while two inner-loop PI controllers can control the d-axis and q-axis currents. The overall goal is to track the reference speeds in the `Speed_Ref` signal. This example uses a reinforcement learning agent to control the currents in the inner control loop while a PI controller controls the outer loop.

In this simulation, the reference speeds are represented by the `speed_ref_rpm` signal, which is the `Speed_Ref` signal expressed in rpm units.

Create Environment Interface

The environment in this example consists of the PMSM system, excluding the inner-loop current controller, which is the reinforcement learning agent. To view the interface between the reinforcement learning agent and the environment, open the Closed Loop Control subsystem.

```
open_system('mcb_pmsm_foc_sim_RL/Current Control/Control_System/Closed Loop Control')
```
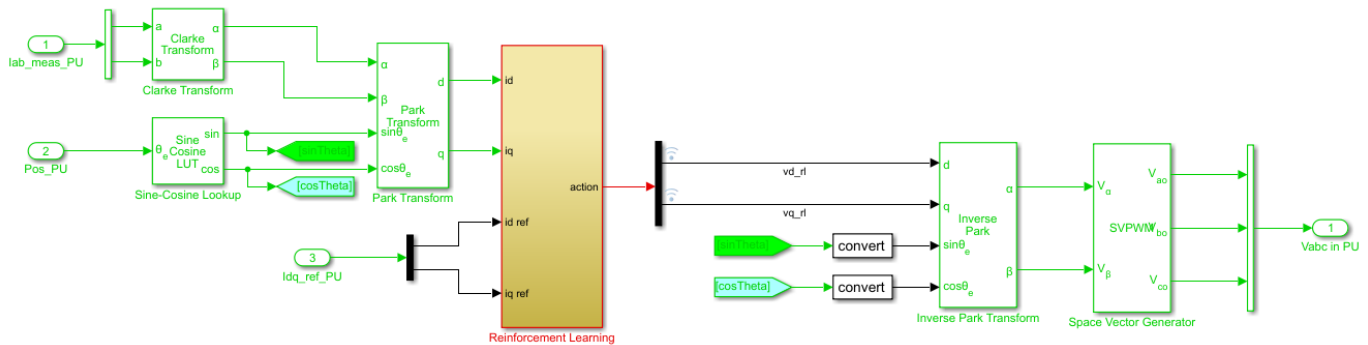
The Reinforcement Learning block contains an RL agent block, the creation of the observation vector, and the reward calculation.

For this environment:

- The observations are the d-axis and q-axis current errors id_{error} and iq_{error}, and their time derivatives and integrals.

- The actions from the agent are the voltages `vd_rl` and `vq_rl`.

- The sample time of the agent is 2e-4 seconds. The inner-loop control occurs at a different sample time than the outer loop.

- The reward at each time step is:

$$r_t = -\left(Q_1 * id_{error}^2 + Q_2 * iq_{error}^2 + R * \sum_j u^j_{t-1}^2 \right)$$

Here, $Q_1 = Q_2 = 0.1$, and $R = 0.05$ are constants, id_{error} is the d-axis current error, iq_{error} is the q-axis current error, and u^j_{t-1} are the actions from the previous time step.

Create the observation and action specifications for the environment. For information on creating continuous specifications, see `rlNumericSpec.`

```
% Create observation specifications.
numObservations = 6;
observationInfo = rlNumericSpec([numObservations 1]);
observationInfo.Name = 'observations';
observationInfo.Description = 'Information on error and reference signal';

% Create action specifications.
numActions = 2;
actionInfo = rlNumericSpec([numActions 1]);
actionInfo.Name = 'vqdRef';
```

Create the Simulink environment interface using the observation and action specifications. For more information on Simulink environments, see `rlSimulinkEnv.`

```
agentblk = 'mcb_pmsm_foc_sim_RL/Current Control/Control_System/Closed Loop Control/Reinforcement
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

Provide a reset function for this environment using the `ResetFcn` parameter. During training, the function `resetPMSM` randomly initializes the final values of step reference speeds in the `SpeedRef` block between 1119 rpm (0.3 pu), 1492 rpm (0.4 pu) and 1865 rpm (0.5 pu) at the beginning of each episode.

```
env.ResetFcn = @resetPMSM;
```

Create Agent

The agent used in this example is a deep deterministic policy gradient (DDPG) agent. A DDPG agent approximates the long-term reward given the observations and actions using a critic value function. For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" on page 3-17.

To create the critic, first create a deep neural network with two inputs (the observation and action) and one output. For more information on creating a neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
rng(0)   % fix the random seed

statePath = [featureInputLayer(numObservations,'Normalization','none','Name','State')
    fullyConnectedLayer(64,'Name','fc1')];
actionPath = [featureInputLayer(numActions, 'Normalization', 'none', 'Name','Action')
    fullyConnectedLayer(64, 'Name','fc2')];
commonPath = [additionLayer(2,'Name','add')
    reluLayer('Name','relu2')
    fullyConnectedLayer(32, 'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(16, 'Name','fc4')
    fullyConnectedLayer(1, 'Name','CriticOutput')];
criticNetwork = layerGraph();
criticNetwork = addLayers(criticNetwork,statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);
criticNetwork = connectLayers(criticNetwork,'fc1','add/in1');
criticNetwork = connectLayers(criticNetwork,'fc2','add/in2');
```

Create the critic representation using the specified neural network and options. You must also specify the action and observation specification for the critic. For more information, see `rlQValueRepresentation`.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1);
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'State'},'Action',{'Action'},criticOptions);
```

A DDPG agent decides which action to take given the observations using an actor representation. To create the actor, first create a deep neural network with one input (the observation) and one output (the action). Construct the actor in a similar manner to the critic. For more information, see `rlDeterministicActorRepresentation`.

```
actorNetwork = [featureInputLayer(numObservations,'Normalization','none','Name','State')
    fullyConnectedLayer(64, 'Name','actorFC1')
    tanhLayer('Name','tanh1')
    fullyConnectedLayer(32, 'Name','actorFC2')
```

```
    tanhLayer('Name','tanh2')
    fullyConnectedLayer(numActions,'Name','Action')
    tanhLayer('Name','tanh3')];
actorOptions = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1);
actor = rlDeterministicActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'State'},'Action',{'tanh3'},actorOptions);
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions`. The agent trains from an experience buffer of maximum capacity 1e6 by randomly selecting mini-batches of size 512. The discount factor of 0.9995 favors long-term rewards. DDPG agents maintain time-delayed copies of the actor and critic networks known as *targets*. The target networks are updated every 20 agent steps during training.

```
Ts_agent = 2e-04;
agentOptions = rlDDPGAgentOptions("SampleTime",Ts_agent, ...
    "DiscountFactor", .9995, ...
    "ExperienceBufferLength",1e6, ...
    "MiniBatchSize",512, ...
    "TargetUpdateFrequency",20);
```

During training, the agent explores the action space using the "Noise Model". Set the noise options using the `NoiseOptions` field. The noise variance decays at the rate of 1e-6, which favors exploration towards the beginning of training and exploitation in later stages.

```
agentOptions.NoiseOptions.MeanAttractionConstant = 0.5;
agentOptions.NoiseOptions.Variance = 0.15;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;
agentOptions.NoiseOptions.VarianceMin = 0.01;
```

Create the agent.

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

Train Agent

To train the agent, first specify the training options using `rlTrainingOptions`. For this example, use the following options:

- Run each training for at most 2000 episodes, with each episode lasting at most `ceil(T/Ts_agent)` time steps.

- Stop training when the agent receives an average cumulative reward greater than -80 over 100 consecutive episodes. At this point, the agent can track the reference speeds.

```
T = 1;
maxepisodes = 2000;
maxsteps = ceil(T/Ts_agent);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes, ...
    'MaxStepsPerEpisode',maxsteps, ...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-80,...
    'ScoreAveragingWindowLength',100);
```

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

During training, only train the reinforcement learning agent for a single reference speed by setting `testReferences` to 0.

```
doTraining = false;
if doTraining
    % Turn off test references during training.
    testReferences = 0;
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load pretrained agents for the example.
    load('rlPMSMAgent.mat')
end
```

A snapshot of training progress is shown in the following figure. You can expect different results due to randomness in the training process.
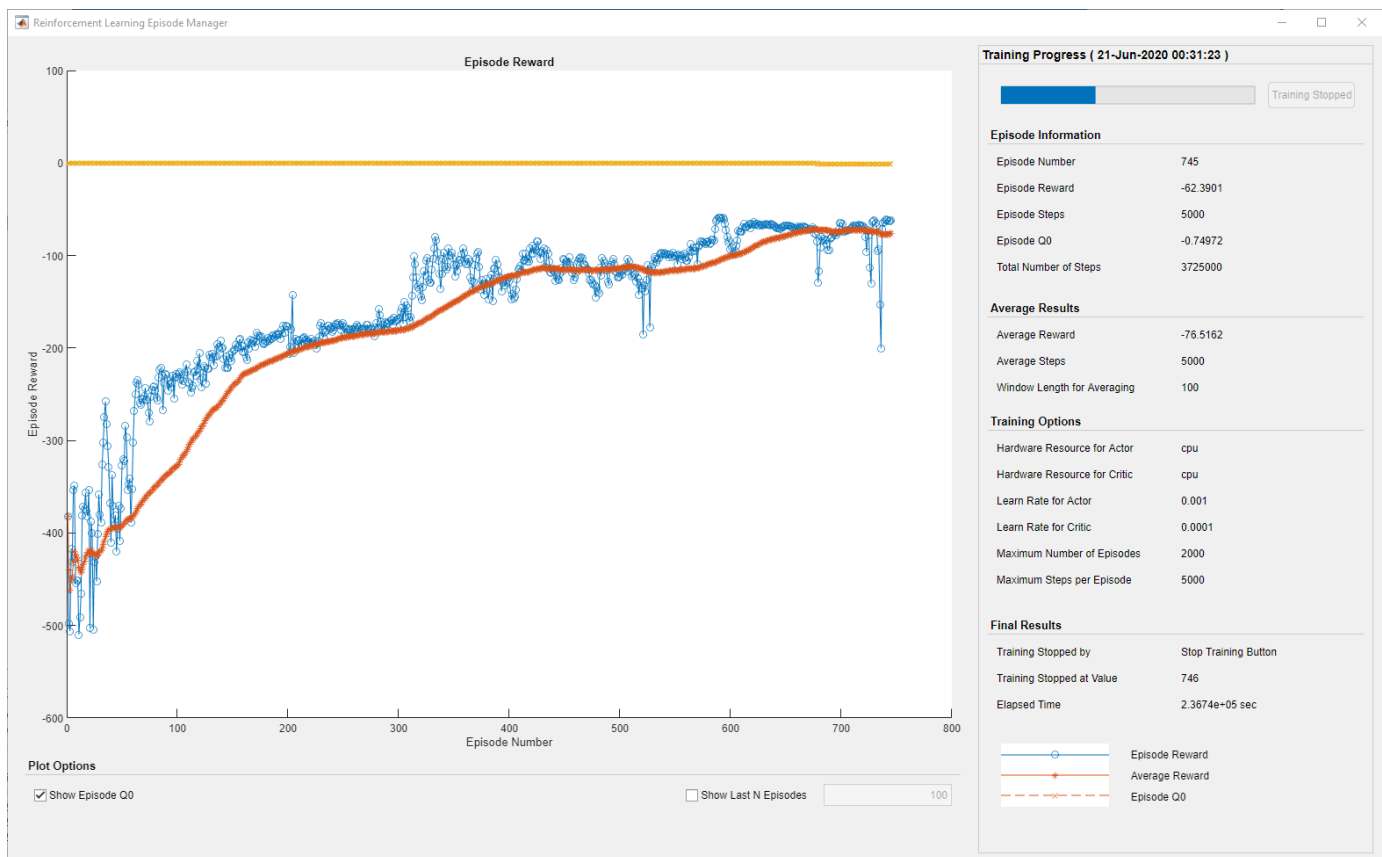
Simulate Agent

To validate the performance of the trained agent, simulate the model. To simulate the model with different reference speeds, adjust the final values of the step references under the `mcb_pmsm_foc_sim_RL/TestRef` subsystem.

For this simulation, enable the test reference step signals by setting `testReferences` to 1.

```
testReferences = 1;
sim(mdl);
```

In this simulation, the reference speed steps through values of 1119 rpm (0.3 per-unit), 3357 rpm (0.9 pu) and 2611 rpm (0.7 pu) with steps times of 0.1, 2 and 4 seconds respectively. The PI and reinforcement learning controllers track the reference signal changes within 0.5-0.8 seconds.

The reinforcement learning controller was trained to track the reference speed of 1119 rpm (0.3 pu) but is able to generalize well across other speeds.

See Also

Imitate MPC Controller for Lane Keeping Assist

This example shows how to train, validate, and test a deep neural network that imitates the behavior of a model predictive controller for an automotive lane keeping assist system. In the example, you also compare the behavior of the deep neural network with that of the original controller.

Model predictive control (MPC) solves a constrained quadratic-programming (QP) optimization problem in real time based on the current state of the plant. Because MPC solves its optimization problem in an open-loop fashion, you can potentially replace the controller with a deep neural network. Evaluating a deep neural network is more computationally efficient than solving a QP problem in real time.

If the training of the network sufficiently traverses the state-space for the application, you can create a reasonable approximation of the controller behavior. You can then deploy the network for your control application. You can also use the network as a warm starting point for training the actor network of a reinforcement learning agent. For an example, see "Train DDPG Agent with Pretrained Actor Network" on page 5-201.

Design MPC Controller

Design an MPC controller for lane keeping assist. To do so, first create a dynamic model for the vehicle.

```
[sys,Vx] = createModelForMPCImLKA;
```

Create and design the MPC controller object `mpcobj`. Also, create an `mpcstate` object for setting the initial controller state. For details on the controller design, type `edit createMPCobjImLKA`.

```
[mpcobj,initialState] = createMPCobjImLKA(sys);
```

For more information on designing model predictive controllers for lane keeping assist applications, see "Lane Keeping Assist System Using Model Predictive Control" (Model Predictive Control Toolbox) and "Lane Keeping Assist with Lane Detection" (Model Predictive Control Toolbox).

Prepare Input Data

Load the input data from `InputDataFileImLKA.mat`. The columns of the data set follow:

1. Lateral velocity V_y
2. Yaw angle rate r
3. Lateral deviation e_1
4. Relative yaw angle e_2
5. Previous steering angle (control variable) u
6. Measured disturbance (road yaw rate: longitudinal velocity * curvature (ρ))
7. Cost function value
8. MPC iterations
9. Steering angle computed by MPC controller: $u*$

The data in `InputDataFileImLKA.mat` was created by computing the MPC control action for randomly generated states, previous control actions, and measured disturbances. To generate your own training data, use the `collectDataImLKA` function.

Load the input data.

```
dataStruct = load('InputDataFileImLKA.mat');
data = dataStruct.Data;
```

Divide the input data into training, validation, and testing data. First, determine the number of validation data rows based on a given percentage.

```
totalRows = size(data,1);
validationSplitPercent = 0.1;
numValidationDataRows = floor(validationSplitPercent*totalRows);
```

Determine the number of test data rows based on a given percentage.

```
testSplitPercent = 0.05;
numTestDataRows = floor(testSplitPercent*totalRows);
```

Randomly extract validation and testing data from the input data set. To do so, first randomly extract enough rows for both data sets.

```
randomIdx = randperm(totalRows,numValidationDataRows + numTestDataRows);
randomData = data(randomIdx,:);
```

Divide the random data into validation and testing data.

```
validationData = randomData(1:numValidationDataRows,:);
testData = randomData(numValidationDataRows + 1:end,:);
```

Extract the remaining rows as training data.

```
trainDataIdx = setdiff(1:totalRows,randomIdx);
trainData = data(trainDataIdx,:);
```

Randomize the training data.

```
numTrainDataRows = size(trainData,1);
shuffleIdx = randperm(numTrainDataRows);
shuffledTrainData = trainData(shuffleIdx,:);
```

Reshape the training and validation data into 4-D matrices for use with `trainNetwork`.

```
numObservations = 6;
numActions = 1;

trainInput = shuffledTrainData(:,1:6);
trainOutput = shuffledTrainData(:,9);

validationInput = validationData(:,1:6);
validationOutput = validationData(:,9);
validationCellArray = {validationInput,validationOutput};
```

Reshape the testing data for use with `predict`.

```
testDataInput = testData(:,1:6);
testDataOutput = testData(:,9);
```

Create Deep Neural Network

The deep neural network architecture uses the following layers.

- `imageInputLayer` is the input layer of the neural network.
- `fullyConnectedLayer` multiplies the input by a weight matrix and then adds a bias vector.
- `reluLayer` is the activation function of the neural network.
- `tanhLayer` constrains the value to the range to [-1,1].
- `scalingLayer` scales the value to the range to [-1.04,1.04], implies that the steering angle is constrained to be [-60,60] degrees.
- `regressionLayer` defines the loss function of the neural network.

Create the deep neural network that will imitate the MPC controller after training.

```
imitateMPCNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','InputLayer')
    fullyConnectedLayer(45,'Name','Fc1')
    reluLayer('Name','Relu1')
    fullyConnectedLayer(45,'Name','Fc2')
    reluLayer('Name','Relu2')
    fullyConnectedLayer(45,'Name','Fc3')
    reluLayer('Name','Relu3')
    fullyConnectedLayer(numActions,'Name','OutputLayer')
    tanhLayer('Name','Tanh1')
    scalingLayer('Name','Scale1','Scale',1.04)
    regressionLayer('Name','RegressionOutput')
];
```

Plot the network.

```
plot(layerGraph(imitateMPCNetwork))
```
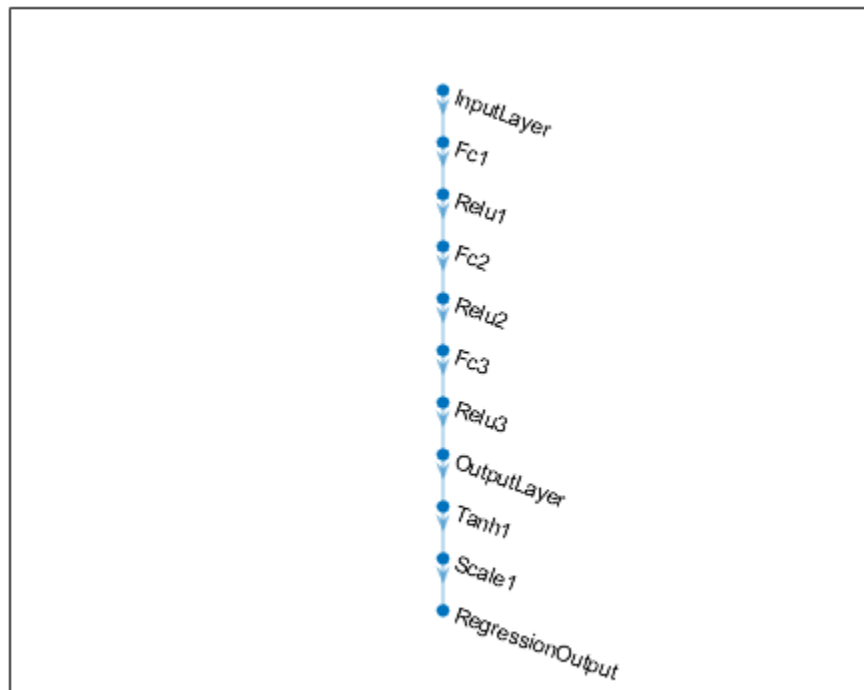
Train Deep Neural Network

Specify training options.

```
options = trainingOptions('adam', ...
    'Verbose',false, ...
    'Plots','training-progress', ...
    'Shuffle','every-epoch', ...
    'MaxEpochs', 30, ...
    'MiniBatchSize',512, ...
    'ValidationData',validationCellArray, ...
    'InitialLearnRate',1e-3, ...
    'GradientThresholdMethod','absolute-value', ...
    'ExecutionEnvironment','cpu', ...
    'GradientThreshold',10, ...
    'Epsilon',1e-8);
```

Train the deep neural network. To view detailed training information in the Command Window, set the `'Verbose'` training option to `true`.

```
imitateMPCNetObj = trainNetwork(trainInput,trainOutput,imitateMPCNetwork,options);
```

Training of the deep neural network stops after the final iteration.

The training and validation loss are nearly the same for each mini-batch, which indicates that the trained network does not overfit.

Test Trained Network

Check that the trained deep neural network returns steering angles similar to the MPC controller control actions given the test input data. Compute the network output using the `predict` function.

```
predictedTestDataOutput = predict(imitateMPCNetObj,testDataInput);
```

Calculate the root mean squared error (RMSE) between the network output and the testing data.

```
testRMSE = sqrt(mean((testDataOutput - predictedTestDataOutput).^2));
fprintf('Test Data RMSE = %d\n', testRMSE);
```

```
Test Data RMSE = 3.200035e-02
```

The small RMSE value indicates that the network outputs closely reproduce the MPC controller outputs.

Compare Trained Network with MPC Controller

To compare the performance of the MPC controller and the trained deep neural network, run closed-loop simulations using the vehicle plant model.

Generate random initial conditions for the vehicle that are not part of the original input data set, with values selected from the following ranges:

1 Lateral velocity V_y — Range (-2,2) *m/s*

2 Yaw angle rate r — Range (-60,60) *deg/s*

3 Lateral deviation e_1 — Range (-1,1) *m*

4 Relative yaw angle e_2 — Range (-45,45) *deg*

5 Last steering angle (control variable) u — Range (-60,60) *deg*

6 Measured disturbance (road yaw rate, defined as longitudinal velocity * curvature (ρ)) — Range (-0.01,0.01) with a minimum road radius of 100 *m*

```
rng(5e7)
[x0,u0,rho] = generateRandomDataImLKA(data);
```

Set the initial plant state and control action in the `mpcstate` object.

```
initialState.Plant = x0;
initialState.LastMove = u0;
```

Extract the sample time from the MPC controller. Also, set the number of simulation steps.

```
Ts = mpcobj.Ts;
Tsteps = 30;
```

Obtain the `A` and `B` state-space matrices for the vehicle model.

```
A = sys.A;
B = sys.B;
```

Initialize the state and input trajectories for the MPC controller simulation.

```
xHistoryMPC = repmat(x0',Tsteps+1,1);
uHistoryMPC = repmat(u0',Tsteps,1);
```

Run a closed-loop simulation of the MPC controller and the plant using the `mpcmove` function.

```
for k = 1:Tsteps
    % Obtain plant output measurements, which correspond to the plant outputs.
    xk = xHistoryMPC(k,:)';
    % Compute the next cotnrol action using the MPC controller.
    uk = mpcmove(mpcobj,initialState,xk,zeros(1,4),Vx*rho);
    % Store the control action.
    uHistoryMPC(k,:) = uk;
    % Update the state using the control action.
    xHistoryMPC(k+1,:) = (A*xk + B*[uk;Vx*rho])';
end
```

Initialize the state and input trajectories for the deep neural network simulation.

```
xHistoryDNN = repmat(x0',Tsteps+1,1);
uHistoryDNN = repmat(u0',Tsteps,1);
lastMV = u0;
```

Run a closed-loop simulation of the trained network and the plant. The `neuralnetLKAmove` function computes the deep neural network output using the `predict` function.

```
for k = 1:Tsteps
    % Obtain plant output measurements, which correspond to the plant outputs.
    xk = xHistoryDNN(k,:)';
    % Predict the next move using the trained deep neural network.
    uk = neuralnetLKAmove(imitateMPCNetObj,xk,lastMV,rho);
    % Store the control action and update the last MV for the next step.
    uHistoryDNN(k,:) = uk;
    lastMV = uk;
    % Update the state using the control action.
    xHistoryDNN(k+1,:) = (A*xk + B*[uk;Vx*rho])';
end
```

Plot the results, and compare the MPC controller and trained deep neural network (DNN) trajectories.

```
plotValidationResultsImLKA(Ts,xHistoryDNN,uHistoryDNN,xHistoryMPC,uHistoryMPC);
```
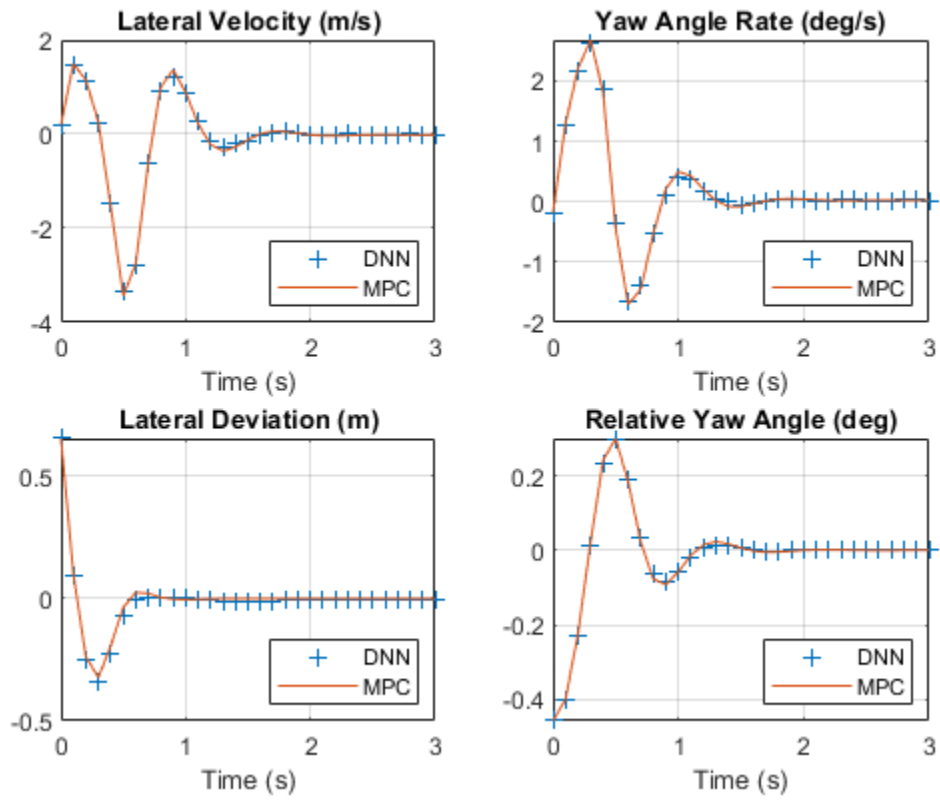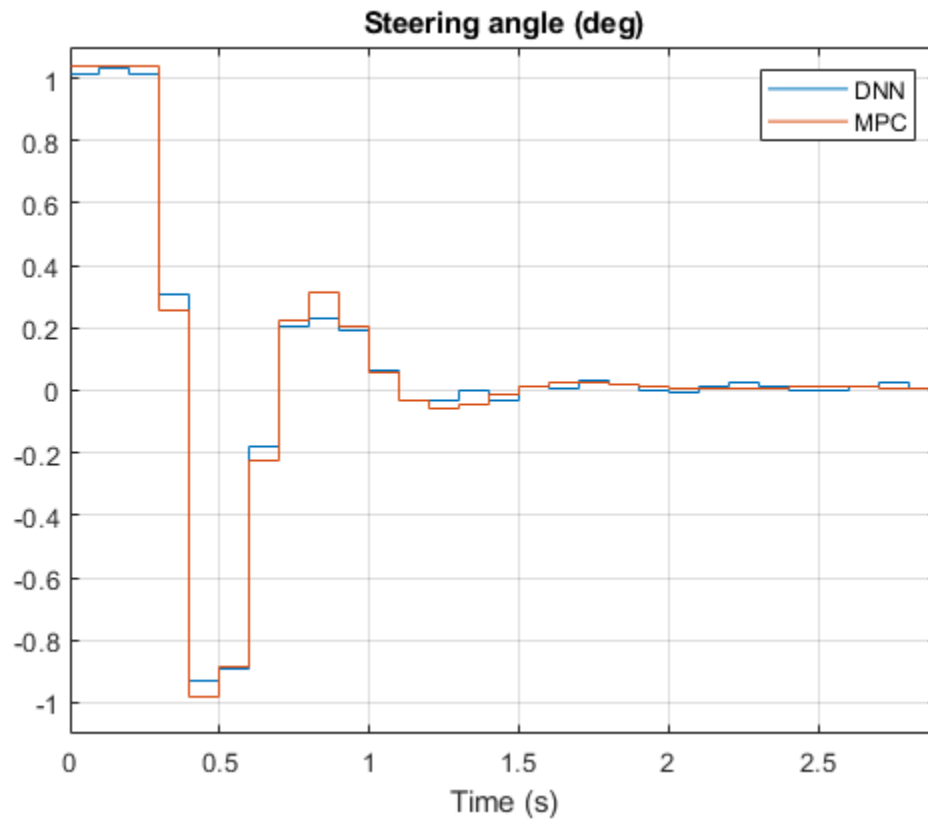
The deep neural network successfully imitates the behavior of the MPC controller. The vehicle state and control action trajectories for the controller and the deep neural network closely align.

See Also
`mpcmove` | `predict` | `trainNetwork`

More About

- "Lane Keeping Assist System Using Model Predictive Control" (Model Predictive Control Toolbox)
- "Lane Keeping Assist with Lane Detection" (Model Predictive Control Toolbox)

Train DDPG Agent with Pretrained Actor Network

This example shows how to train a deep deterministic policy gradient (DDPG) agent for lane keeping assist (LKA) in Simulink. To make training more efficient, the actor of the DDPG agent is initialized with a deep neural network that was previously trained using supervised learning. This actor trained is trained in the "Imitate MPC Controller for Lane Keeping Assist" on page 5-193 example.

For more information on DDPG agents, see Deep Deterministic Policy Gradient Agents.

Simulink Model

The training goal for the lane-keeping application is to keep the ego vehicle traveling along the centerline of the a lane by adjusting the front steering angle. This example uses the same ego vehicle dynamics and sensor dynamics as the "Train DQN Agent for Lane Keeping Assist" on page 5-134 example.

```
m = 1575;   % total vehicle mass (kg)
Iz = 2875;  % yaw moment of inertia (mNs^2)
lf = 1.2;   % longitudinal distance from center of gravity to front tires (m)
lr = 1.6;   % longitudinal distance from center of gravity to rear tires (m)
Cf = 19000; % cornering stiffness of front tires (N/rad)
Cr = 33000; % cornering stiffness of rear tires (N/rad)
Vx = 15;    % longitudinal velocity (m/s)
```

Define the sample time, `Ts`, and simulation duration, `T`, in seconds.

```
Ts = 0.1;
T = 15;
```

The output of the LKA system is the front steering angle of the ego vehicle. Considering the physical limitations of the ego vehicle, constrain its steering angle to the range [-60,60] degrees. Specify the constrains in radians.

```
u_min = -1.04;
u_max = 1.04;
```

Define the curvature of the road as a constant $0.001(m^{-1})$.

```
rho = 0.001;
```

Set initial values for the lateral deviation (`e1_initial`) and the relative yaw angle (`e2_initial`). During training, these initial conditions are set to random values for each training episode.

```
e1_initial = 0.2;
e2_initial = -0.1;
```

Open the model.

```
mdl = 'rlActorLKAMdl';
open_system(mdl)
```
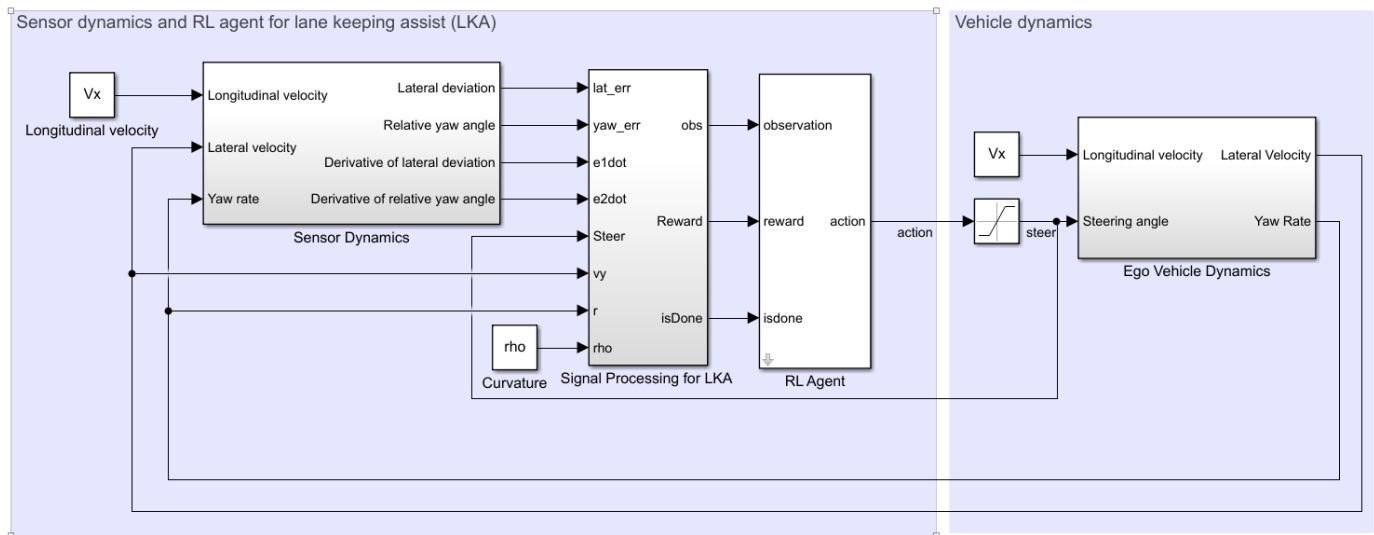
Copyright 2018 The MathWorks, Inc.

Define the path to the RL Agent block within the model.

```
agentblk = [mdl '/RL Agent'];
```

Create Environment

Create a reinforcement learning environment interface for the ego vehicle. To do so, first define the observation and action specifications. These observations and actions are the same as the features for supervised learning used in "Imitate MPC Controller for Lane Keeping Assist" on page 5-193.

The six observations for the environment are the lateral velocity v_y, yaw rate $\dot{\psi}$, lateral deviation e_1, relative yaw angle e_2, steering angle at previous step u_0, and curvature ρ.

```
observationInfo = rlNumericSpec([6 1],...
    'LowerLimit',-inf*ones(6,1),'UpperLimit',inf*ones(6,1));
observationInfo.Name = 'observations';
```

The action for the environment is the front steering angle. Specify the steering angle constraints when creating the action specification object.

```
actionInfo = rlNumericSpec([1 1],'LowerLimit',u_min,'UpperLimit',u_max);
actionInfo.Name = 'steering';
```

In the model, the Signal Processing for LKA block creates the observation vector signal, computes the reward function, and calculates the stop condition signal.

The reward r_t, provided at every time step t, is as follows, where u is the control input from the previous time step $t - 1$.

$$r_t = -(10e_1^2 + 5e_2^2 + 2u^2 + 5\dot{e}_1^2 + 5\dot{e}_2^2)$$

The simulation stops when $|e_1| > 1$.

Create the reinforcement learning environment.

```
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

To define the initial condition for lateral deviation and relative yaw angle, specify an environment reset function using an anonymous function handle. The `localResetFcn` function, which is defined at the end of the example, sets the initial lateral deviation and relative yaw angle to random values.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG Agent

A DDPG agent approximates the long-term reward given observations and actions using a critic value function representation. To create the critic, first create a deep neural network with two inputs, the state and action, and one output. For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

```
critic = createLaneKeepingCritic(observationInfo,actionInfo);
```

A DDPG agent decides which action to take given observations using an actor representation. To create the actor, first create a deep neural network with one input (the observation) and one output (the action).

```
[actor,actorOptions] = createLaneKeepingActor(observationInfo,actionInfo);
```

These initial actor and critic networks have random initial parameter values.

To create the DDPG agent, first specify the DDPG agent options.

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',64 ...
    );
agentOptions.NoiseOptions.Variance = 0.3;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent`.

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

Train Agent

As a baseline, train the agent with an actor that has random initial parameters. To train the agent, first specify the training options. For this example, use the following options.

- Run training for at most 50000 episodes, with each episode lasting at most 150 time steps.
- Display the training progress in the Episode Manager dialog box.
- Stop training when the episode reward reaches –1.
- Save a copy of the agent for each episode where the cumulative reward is greater than –`2.5`.

For more information, see `rlTrainingOptions`.

```matlab
maxepisodes = 50000;
maxsteps = T/Ts;
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue',-1,...
    'SaveAgentCriteria','EpisodeReward',...
    'SaveAgentValue',-2.5);
```

Train the agent using the `train` function. Training is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```matlab
doTraining = false;
```

```matlab
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load pretrained agent for the example.
    load('ddpgFromScratch.mat');
end
```
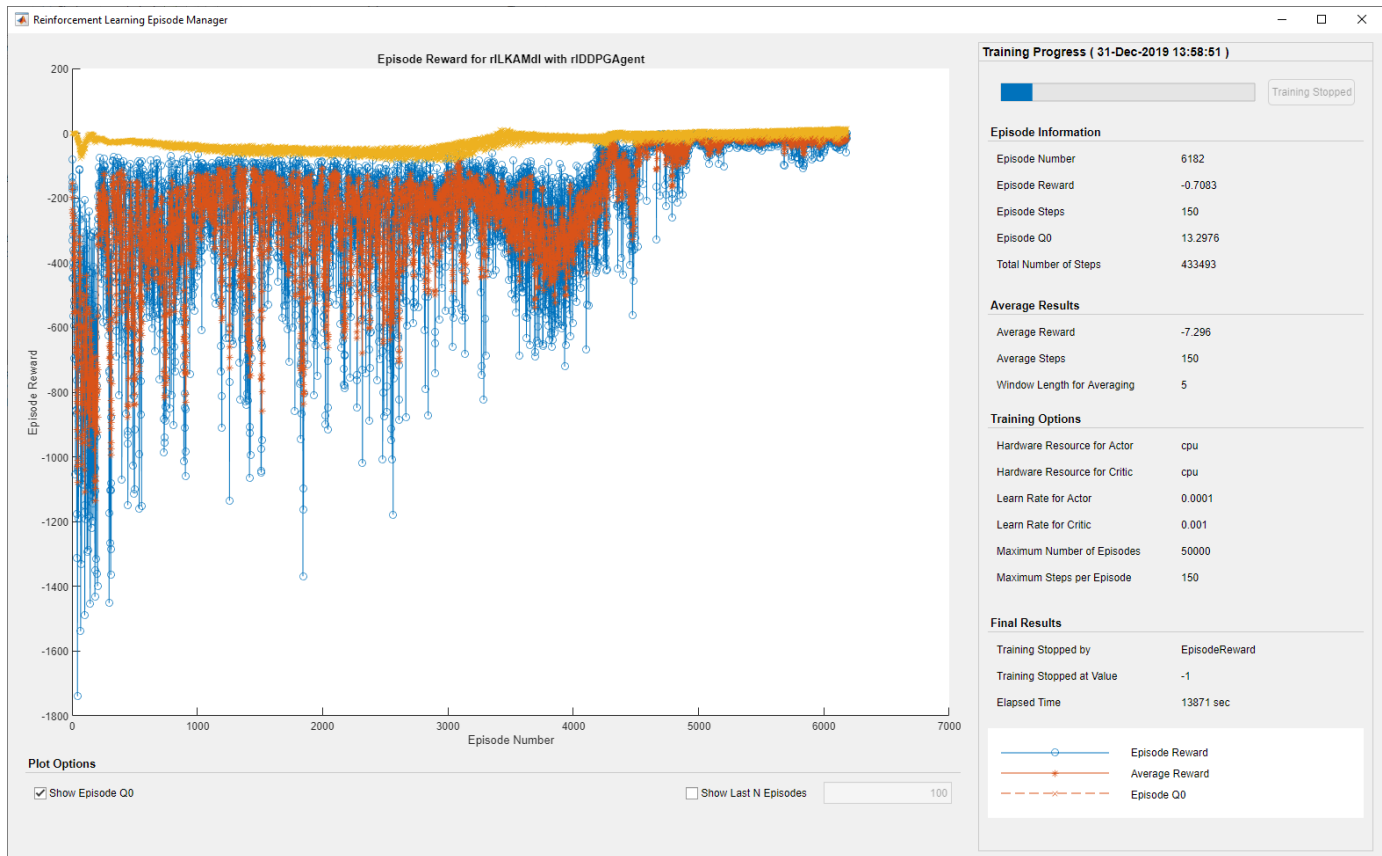
Train Agent with Pretrained Actor

You can set the actor network of your agent to a deep neural network that has been previously trained. For this example, use the deep neural network from the "Imitate MPC Controller for Lane Keeping Assist" on page 5-193 example. This network was trained to imitate a model predictive controller using supervised learning.

Load the pretrained actor network.

```
load('imitateMPCNetActorObj.mat','imitateMPCNetObj');
```

Create an actor representation using the pretrained actor.

```
supervisedActor = rlDeterministicActorRepresentation(imitateMPCNetObj,observationInfo,actionInfo
'Observation',imitateMPCNetObj.InputNames,'Action',{'Scale1'},actorOptions);
```

Check that the network used by `supervisedActor` is the same one that was loaded. To do so, evaluate both the network and the agent using the same random input observation.

```
testData = rand(6,1);
```

Evaluate the deep neural network.

```
predictImNN = predict(imitateMPCNetObj,testData');
```

Evaluate the actor.

```
evaluateRLRep = getAction(supervisedActor,{testData});
```

Compare the results.

```
error = evaluateRLRep{:} - predictImNN
```

```
error = single
    0
```

Create a DDPG agent using the pretrained actor.

```
agent = rlDDPGAgent(supervisedActor,critic,agentOptions);
```

Reduce the maximum number of training episodes and train the agent using the `train` function. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
trainingOpts.MaxEpisodes = 5000;
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load pretrained agent for the example.
    load('ddpgFromPretrained.mat');
end
```
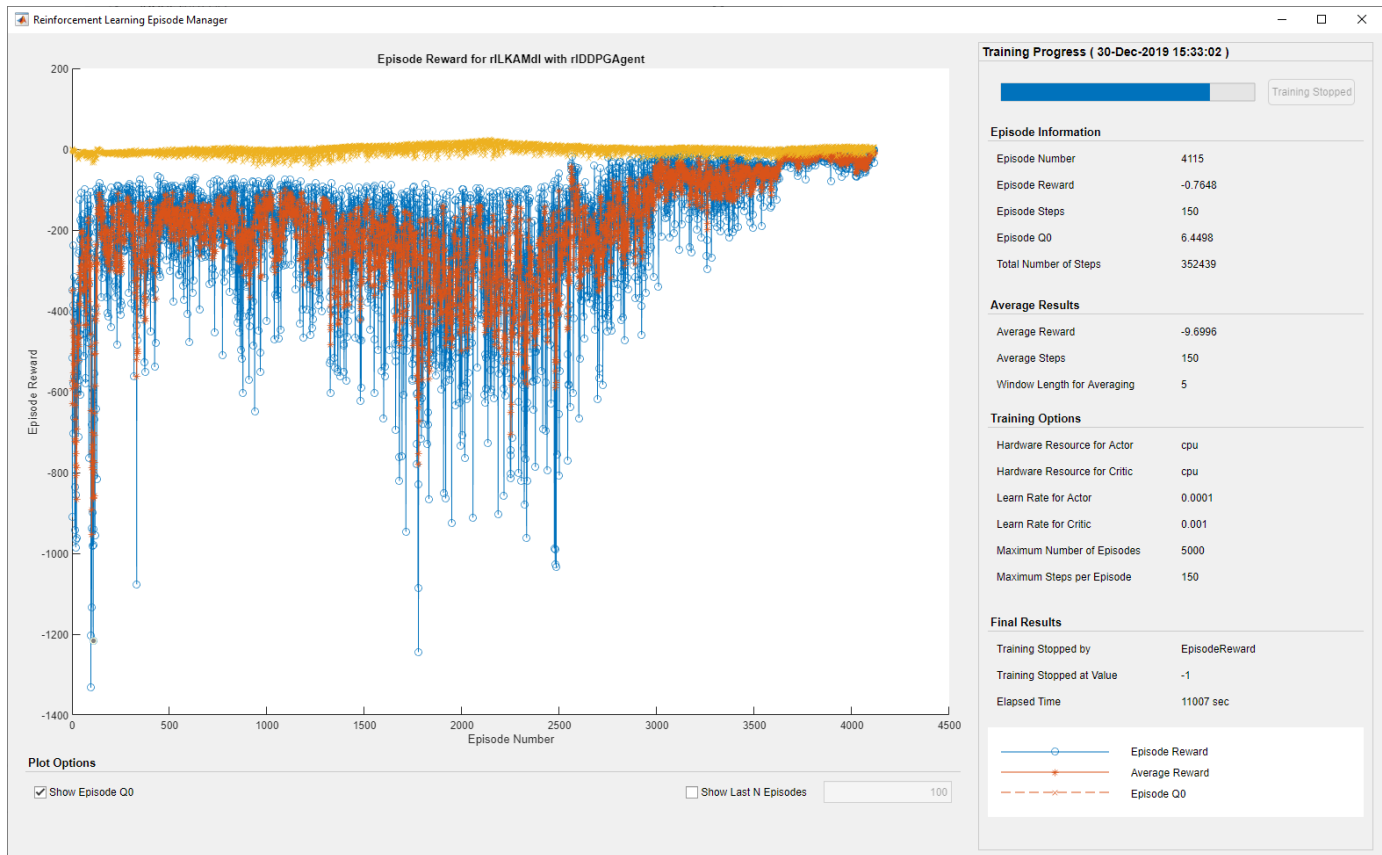
By using the pretrained actor network, the training of the DDPG agent is more efficient. Both the total training time and the total number of training steps have improved by approximately 20%. Also, the number of episodes for the training to approach the neighborhood of the optimal result decreased from approximately 4500 to approximately 3500.

Simulate DDPG Agent

To validate the performance of the trained agent, uncomment the following two lines and simulate it within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

To check the performance of the trained agent within the Simulink model, simulate the model using the previously defined initial conditions (`e1_initial = 0.2` and `e2_initial = -0.1`).

```
sim(mdl)
```

As shown below, the lateral error (middle plot) and relative yaw angle (bottom plot) are both driven to zero. The vehicle starts with a lateral deviation from the centerline (0.2 m) and a nonzero yaw angle error (-0.1 rad). The lane-keeping controller makes the ego vehicle travel along the centerline after around two seconds. The steering angle (top plot) shows that the controller reaches steady state after about two seconds.
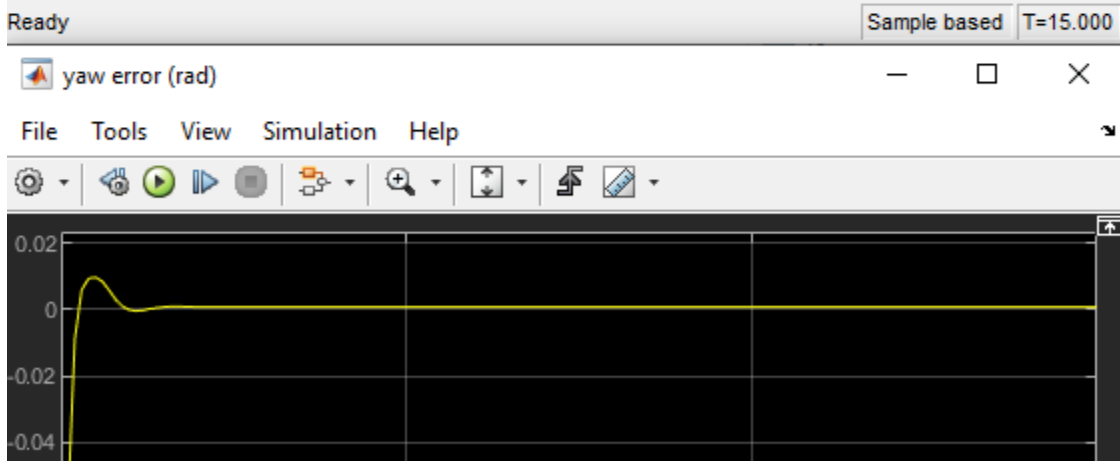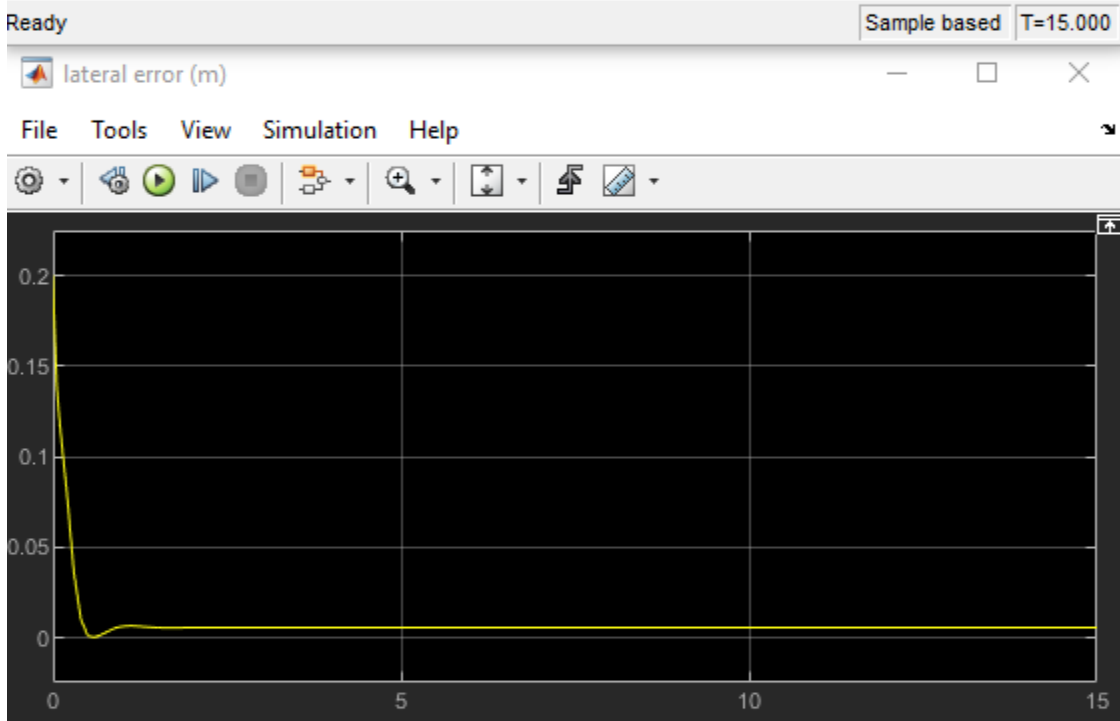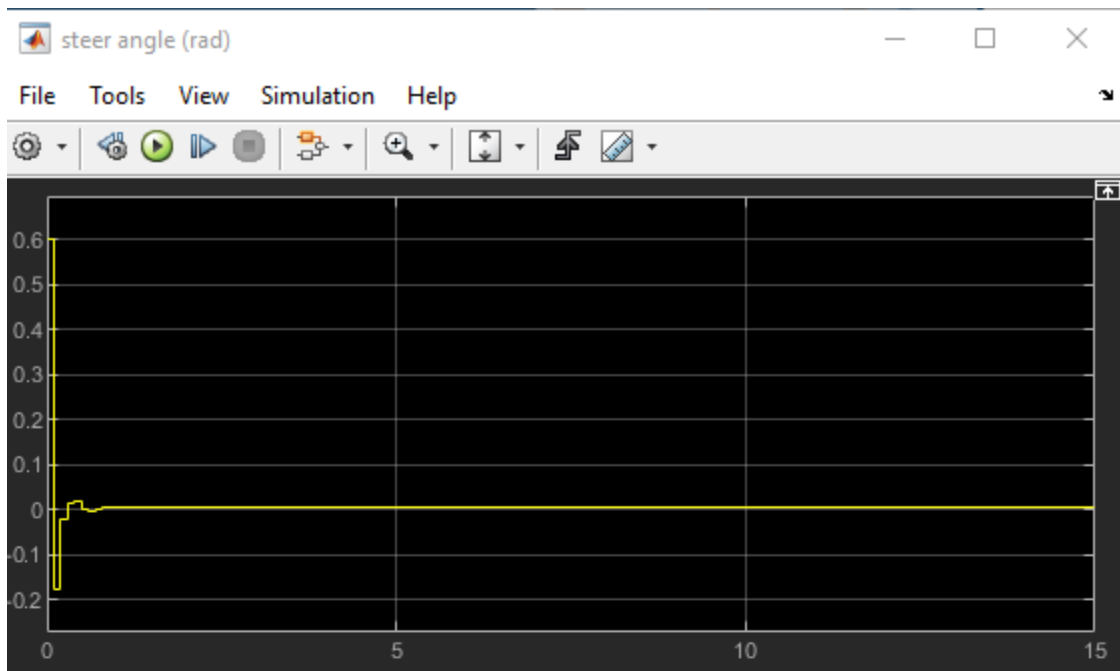
Close the Simulink model.

```
bdclose(mdl)
```

Local Functions

```matlab
function in = localResetFcn(in)
% Set random value for lateral deviation.
in = setVariable(in,'e1_initial', 0.5*(-1+2*rand));

% Set random value for relative yaw angle.
in = setVariable(in,'e2_initial', 0.1*(-1+2*rand));
end

function agent = loadPretrainedParams(agent,actorParams,criticParams)
% Set actor parameters.
actor = getActor(agent);
pretrainedActor = setLearnableParameters(actor,actorParams);

% Set critic parameters.
critic = getCritic(agent);
pretrainedCritic = setLearnableParameters(critic,criticParams);

% Set actor and critic representations in the agent.
agent = setActor(agent,pretrainedActor);
agent = setCritic(agent,pretrainedCritic);
end
```

See Also
`rlDDPGAgent` | `train`

More About
- "Imitate MPC Controller for Lane Keeping Assist" on page 5-193
- "Train DQN Agent for Lane Keeping Assist" on page 5-134
- "Deep Deterministic Policy Gradient Agents" on page 3-17

Imitate Nonlinear MPC Controller for Flying Robot

This example shows how to train, validate, and test a deep neural network (DNN) that imitates the behavior of a nonlinear model predictive controller for a flying robot. It then compares the behavior of the deep neural network with that of the original controller. To train the deep neural network, this example uses the data aggregation (DAgger) approach as in [1].

Nonlinear model predictive control (NLMPC) solves a constrained nonlinear optimization problem in real time based on the current state of the plant. Since NLMPC solves its optimization problem in an open-loop fashion, there is the potential to replace the controller with a trained DNN. Doing so is an appealing option, since evaluating a DNN can be more computationally efficient than solving a nonlinear optimization problem in real-time.

If the training of the DNN creates a reasonable approximation of the controller behavior, you can then deploy the network for your control application. You can also use the network as a warm starting point for training the actor network of a reinforcement learning agent. For an example that does so with a DNN trained for an MPC application, see "Train DDPG Agent with Pretrained Actor Network" on page 5-201.

Design Nonlinear MPC Controller

Design a nonlinear MPC controller for a flying robot. The dynamics for the flying robot are the same as in "Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC" (Model Predictive Control Toolbox) example. First, define the limit for the control variables, which are the robot thrust levels.

umax = 3;

Create the nonlinear MPC controller object `nlobj`. To reduce command-window output, disable the MPC update messages.

```
mpcverbosity off;
nlobj = createMPCobjImFlyingRobot(umax);
```

Prepare Input Data

Load the input data from `DAggerInputDataFileImFlyingRobot.mat`. The columns of the data set contain:

1 x is the position of the robot along the x-axis.
2 y is the position of the robot along the y-axis.
3 θ is the orientation of the robot.
4 $\dot{x}$ is the velocity of the robot along the x-axis.
5 $\dot{y}$ is the velocity of the robot along the y-axis.
6 $\dot{\theta}$ is the angular velocity of the robot.
7 u_l is the thrust on the left side of the flying robot
8 u_r is the thrust on the right side of the flying robot
9 u_l^* is the thrust on the left side computed by NLMPC
10 u_r^* is the thrust on the right side computed by NLMPC

The data in `DAggerInputDataFileImFlyingRobot.mat` is created by computing the NLMPC control action for randomly generated states (x, y, θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$), and previous control actions (u_l, u_r). To generate your own training data, use the `collectDataImFlyingRobot` function.

Load the input data.

```
fileName = 'DAggerInputDataFileImFlyingRobot.mat';
DAggerData = load(fileName);
data = DAggerData.data;
existingData = data;
numCol = size(data,2);
```

Create Deep Neural Network

The deep neural network architecture uses the following types of layers.

- `imageInputLayer` is the input layer of the neural network.
- `fullyConnectedLayer` multiplies the input by a weight matrix and then adds a bias vector.
- `reluLayer` is the activation function of the neural network.
- `tanhLayer` constrains the value to the range to [-1,1].
- `scalingLayer` scales the value to the range to [-3,3].
- `regressionLayer` defines the loss function of the neural network.

Create the deep neural network that will imitate the NLMPC controller after training.

```
numObservations = numCol-2;
numActions = 2;
hiddenLayerSize = 256;

imitateMPCNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','observation')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc4')
    reluLayer('Name','relu4')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc5')
    reluLayer('Name','relu5')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc6')
    reluLayer('Name','relu6')
    fullyConnectedLayer(numActions,'Name','fcLast')
    tanhLayer('Name','tanhLast')
    scalingLayer('Name','ActorScaling','Scale',umax)
    regressionLayer('Name','routput')];
```

Plot the network.

```
plot(layerGraph(imitateMPCNetwork))
```
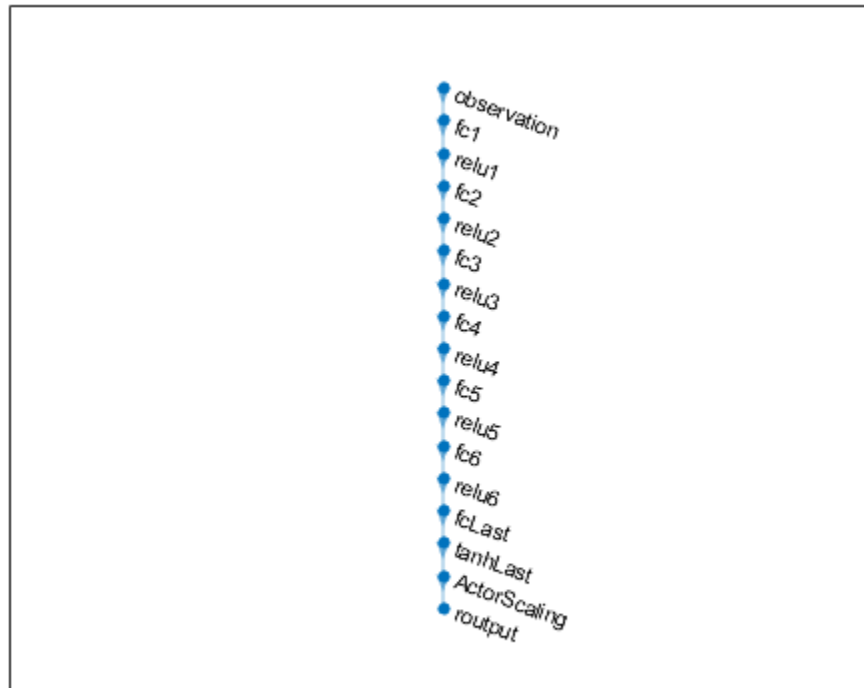
Behavior Cloning Approach

One approach to learning an expert policy using supervised learning is the behavior cloning method. This method divides the expert demonstrations (NLMPC control actions in response to observations) into state-action pairs and applies supervised learning to train the network.

Specify training options.

```
% intialize validation cell array
validationCellArray = {0,0};

options = trainingOptions('adam', ...
    'Verbose', false, ...
    'Plots', 'training-progress', ...
    'Shuffle', 'every-epoch', ...
    'MiniBatchSize', 512, ...
    'ValidationData', validationCellArray, ...
    'InitialLearnRate', 1e-3, ...
    'ExecutionEnvironment', 'cpu', ...
    'GradientThreshold', 10, ...
    'MaxEpochs', 40 ...
    );
```

You can train the behavior cloning neural network by following below steps

1. Collect data using the `collectDataImFlyingRobot` function.
2. Train the behavior cloning network using the `behaviorCloningTrainNetwork` function.

Training a DNN is a computationally intensive process. To save time, load a pretrained neural network object.

```
load('behaviorCloningMPCImDNNObject.mat');
imitateMPCNetBehaviorCloningObj = behaviorCloningNNObj.imitateMPCNetObj;
```
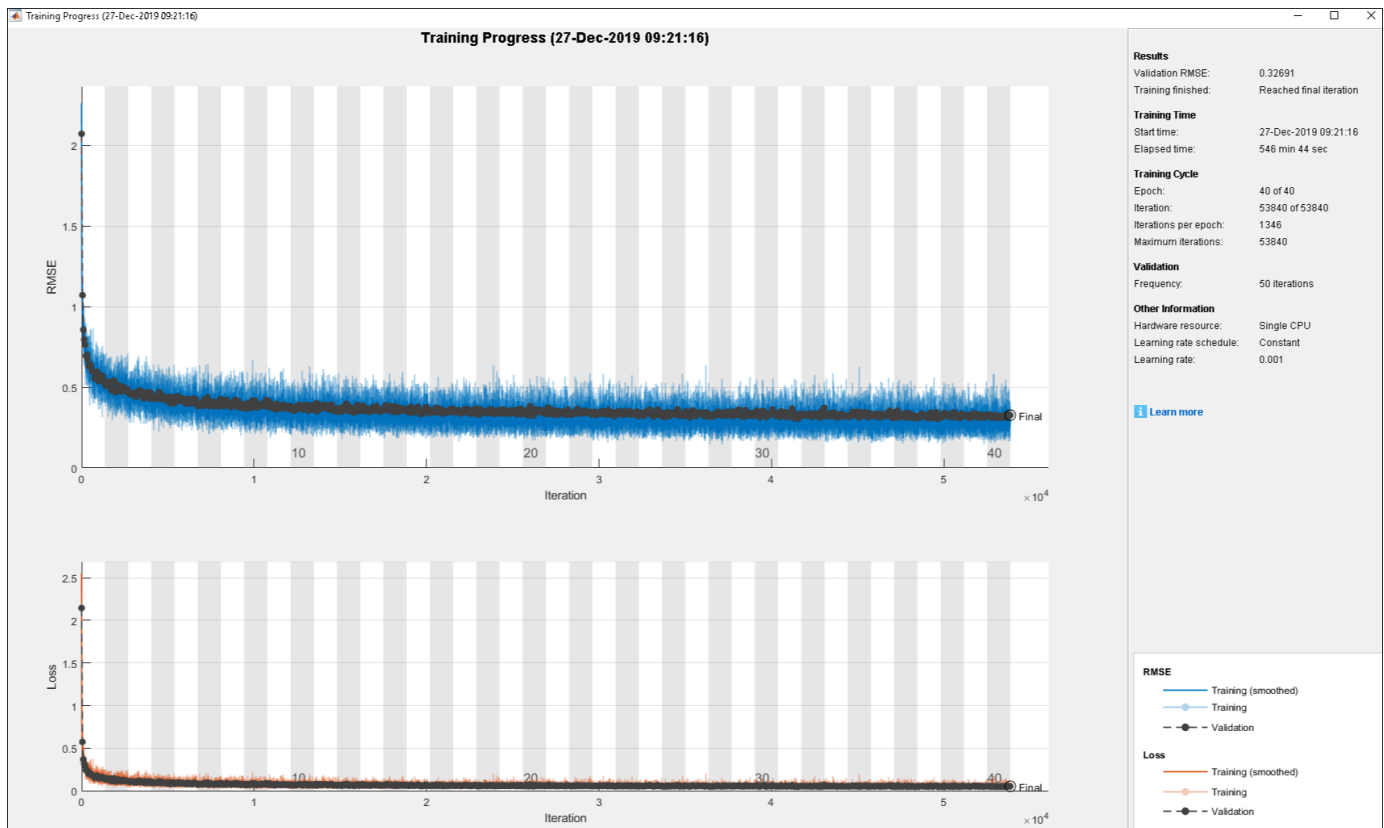

The training of the DNN using behavior cloning reduces the gap between the DNN and NLMPC performance. However, the behavior cloning neural network fails to imitate the behavior of the NLMPC controller correctly on some randomly generated data.

Data Aggregation Approach

To improve the performance of the DNN, you can learn the policy using an interactive demonstrator method. DAgger is an iterative method where the DNN is run in the closed-loop environment. The expert, in this case the NLMPC controller, outputs actions based on the states visited by the DNN. In this manner, more training data is aggregated and the DNN is retrained for improved performance. For more information, see [1].

Train the deep neural network using the `DAggerTrainNetwork` function. It creates `DAggerImFlyingRobotDNNObj.mat` file that contains the following information.

- `DatasetPath:` path where the dataset corresponding to each iteration is stored
- `policyObjs:` policies that were trained in each iteration
- `finalData:` total training data collected till final iteration
- `finalPolicy:` best policy among all the collected policies

First, create and initialize the parameters for training. Use the network trained using behavior cloning (`imitateMPCNetBehaviorCloningObj`) as the starting point for the DAgger training.

```
[dataStruct,nlmpcStruct,tuningParamsStruct,neuralNetStruct] = loadDAggerParameters(existingData,
    numCol,nlobj,umax,options,imitateMPCNetBehaviorCloningObj);
```

To save time, load a pretrained neural network by setting `doTraining` to `false`. To train the DAgger yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    DAgger = DAggerTrainNetwork(nlmpcStruct,dataStruct,neuralNetStruct,tuningParamsStruct);
else
    load('DAggerImFlyingRobotDNNObj.mat');
end
DNN = DAgger.finalPolicy;
```

As an alternative, you can train the neural network with a modified policy update rule using the `DAggerModifiedTrainNetwork` function. In this function, after every 20 training iterations, the DNN is set to the most optimal configuration from the previous 20 iterations. To run this example with a neural network object with the modified DAgger approach, use the `DAggerModifiedImFlyingRobotDNNObj.mat` file.

Compare Trained DAgger Network with NLMPC Controller

To compare the performance of the NLMPC controller and the trained DNN, run closed-loop simulations with the flying robot model.

Set initial condition for the states of the flying robot (x, y, θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$) and the control variables of flying robot (u_l, u_r).

```
x0 = [-1.8200    0.5300   -2.3500    1.1700   -1.0400    0.3100]';
u0 = [-2.1800   -2.6200]';
```

Run a closed-loop simulation of the NLMPC controller.

```
% Duration
Tf = 15;
% Sample time
Ts = nlobj.Ts;
% Simulation steps
Tsteps = Tf/Ts+1;
% Run NLMPC in closed loop.
tic
[xHistoryMPC,uHistoryMPC] = simModelMPCImFlyingRobot(x0,u0,nlobj,Tf);
toc
```

```
Elapsed time is 67.867603 seconds.
```

Run a closed-loop simulation of the trained DAgger network.

```
tic
[xHistoryDNN,uHistoryDNN] = simModelDAggerImFlyingRobot(x0,u0,DNN,Ts,Tf);
toc
```

```
Elapsed time is 2.290211 seconds.
```

Plot the results, and compare the NLMPC and trained DNN trajectories.

```
plotSimResultsImFlyingRobot(nlobj,xHistoryMPC,uHistoryMPC,xHistoryDNN,uHistoryDNN,umax,Tf)
```
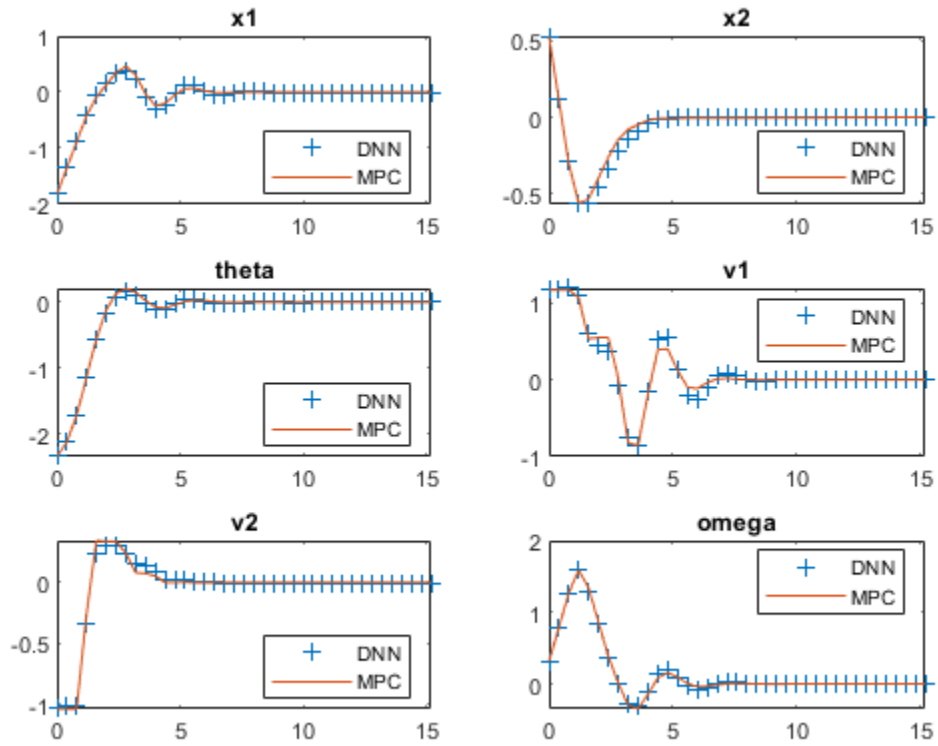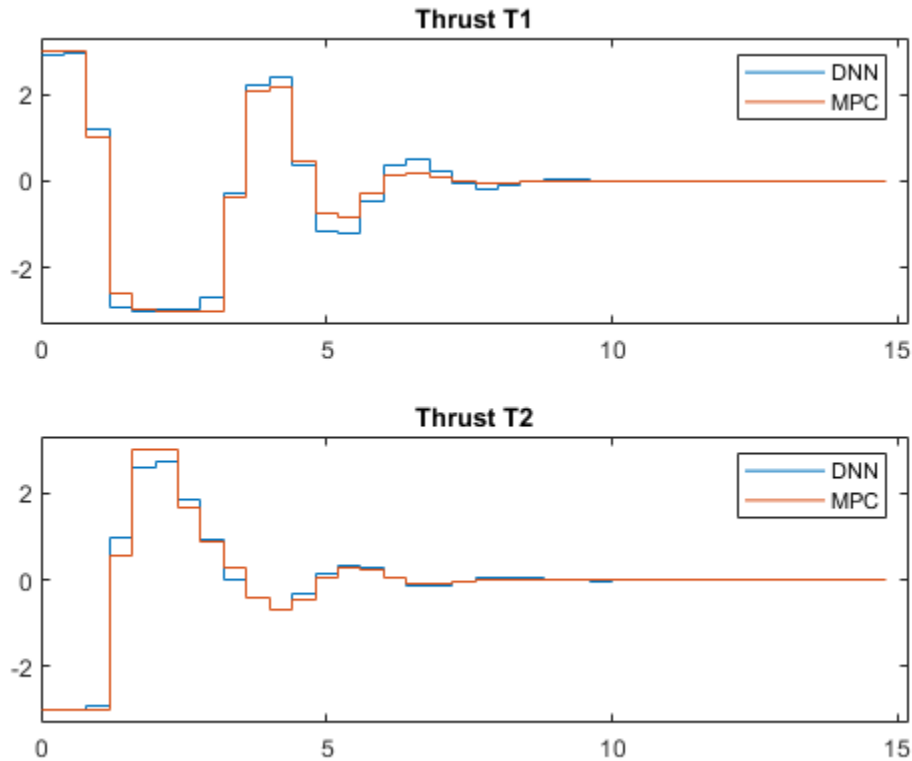
The DAgger neural network successfully imitates the behavior of the NLMPC controller. The flying robot states and control action trajectories for the controller and the DAgger deep neural network closely align. The closed-loop simulation time for the DNN is significantly less than that of the NLMPC controller.

Animate the Flying Robot with Trained DAgger Network

To validate the performance of the trained DNN, animate the flying robot with data from the DNN closed-loop simulation. The flying robot lands at the origin successfully.

```
Lx = 5;
Ly = 5;
for ct = 1:Tsteps
    x = xHistoryDNN(ct,1);
    y = xHistoryDNN(ct,2);
    theta = xHistoryDNN(ct,3);
    tL = uHistoryDNN(ct,1);
    tR = uHistoryDNN(ct,2);
    rl.env.viz.plotFlyingRobot(x,y,theta,tL,tR,Lx,Ly);
    pause(0.05);
end
```
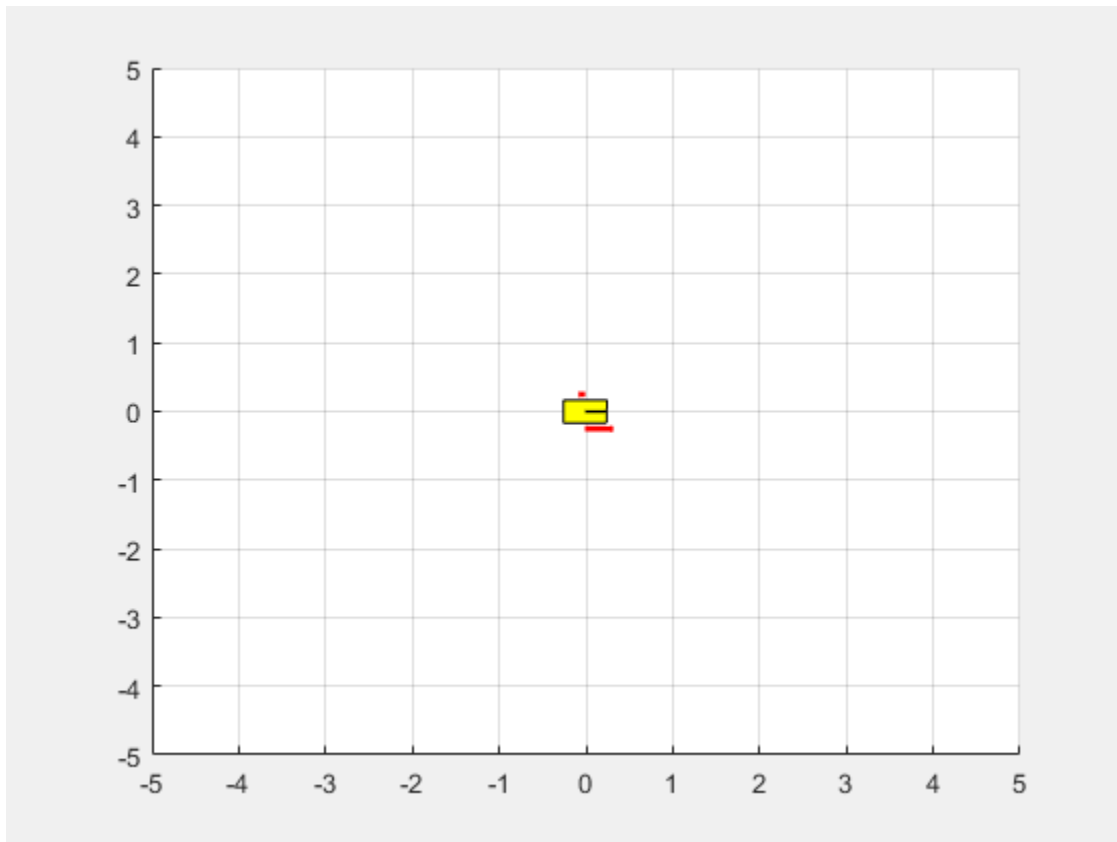
```
% Turn on MPC messages
mpcverbosity on;
```

References

[1] Osa, Takayuki, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. 'An Algorithmic Perspective on Imitation Learning'. *Foundations and Trends in Robotics* 7, no. 1–2 (2018): 1–179. https://doi.org/10.1561/2300000053.

See Also

`rlDDPGAgent` | `train`

More About

- "Imitate MPC Controller for Lane Keeping Assist" on page 5-193
- "Train DQN Agent for Lane Keeping Assist" on page 5-134
- "Deep Deterministic Policy Gradient Agents" on page 3-17

Tune PI Controller using Reinforcement Learning

This example shows how to tune a PI controller using the twin-delayed deep deterministic policy gradient (TD3) reinforcement learning algorithm. The performance of the tuned controller is compared with that of a controller tuned using the **Control System Tuner** app. Using the **Control System Tuner** app to tune controllers in Simulink® requires Simulink Control Design™ software.

For relatively simple control tasks with a small number of tunable parameters, model-based tuning techniques can get good results with a faster tuning process compared to model-free RL-based methods. However, RL methods can be more suitable for highly nonlinear systems or adaptive controller tuning.

To facilitate the controller comparison, both tuning methods use a linear quadratic Gaussian (LQG) objective function.

This example uses a reinforcement learning (RL) agent to compute the gains for a PI controller. For an example that replaces the PI controller with a neural network controller, see "Create Simulink Environment and Train Agent" on page 1-19.

Environment Model

The environment model for this example is a water tank model. The goal of this control system is to maintain the level of water in a tank to match a reference value.

```
open_system('watertankLQG')
```
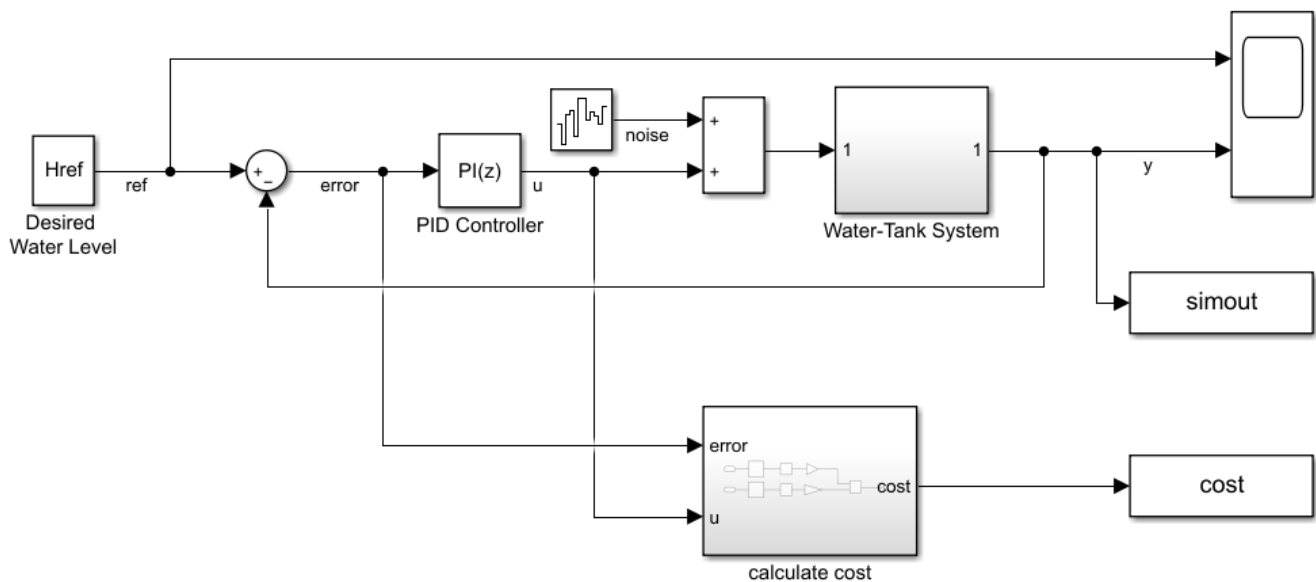

The model includes process noise with variance $E\left(n^2(t)\right) = 1$.

To maintain the water level while minimizing control effort u, the controllers in this example use the following LQG criterion.

$$J = \lim_{T \Rightarrow \infty} E\left(\frac{1}{T}\int_0^T \left((\text{ref} - y)^2(t) + 0.01u^2(t)\right)dt\right)$$

To simulate the controller in this model, you must specify the simulation time `Tf` and the controller sample time `Ts` in seconds.

```
Ts = 0.1;
Tf = 10;
```

For more information about the water tank model, see "watertank Simulink Model" (Simulink Control Design).

Tune PI Controller using Control System Tuner

To tune a controller in Simulink using **Control System Tuner**, you must specify the controller block as a tuned block and define the goals for the tuning process. For more information on using **Control System Tuner**, see "Tune a Control System Using Control System Tuner" (Simulink Control Design).

For this example, open the saved session `ControlSystemTunerSession.mat` using **Control System Tuner**. This session specifies the PID Controller block in the `watertankLQG` model as a tuned block and contains an LQG tuning goal.

```
controlSystemTuner("ControlSystemTunerSession")
```

To tune the controller, on the **Tuning** tab, click **Tune**.

The tuned proportional and integral gains are approximately 9.8 and 1e-6, respectively.

```
Kp_CST = 9.80199999804512;
Ki_CST = 1.00019996230706e-06;
```

Create Environment for Training Agent

To define the model for training the RL agent, modify the water tank model using the following steps.

1 Delete the PID Controller.
2 Insert an RL Agent block.
3 Create the observation vector $[\int e \, dt \; e \;]^T$ where $e = r - h$, h is the height of the tank, and r is the reference height. Connect the observation signal to the RL Agent block.
4 Define the reward function for the RL agent as the **negative** of the LQG cost , that is, Reward $= -\left((\text{ref} - h)^2(t) + 0.01u^2(t)\right)$. The RL agent maximizes this reward, thus minimizing the LQG cost.

The resulting model is `rlwatertankPIDTune.slx`.

```
mdl = 'rlwatertankPIDTune';
open_system(mdl)
```
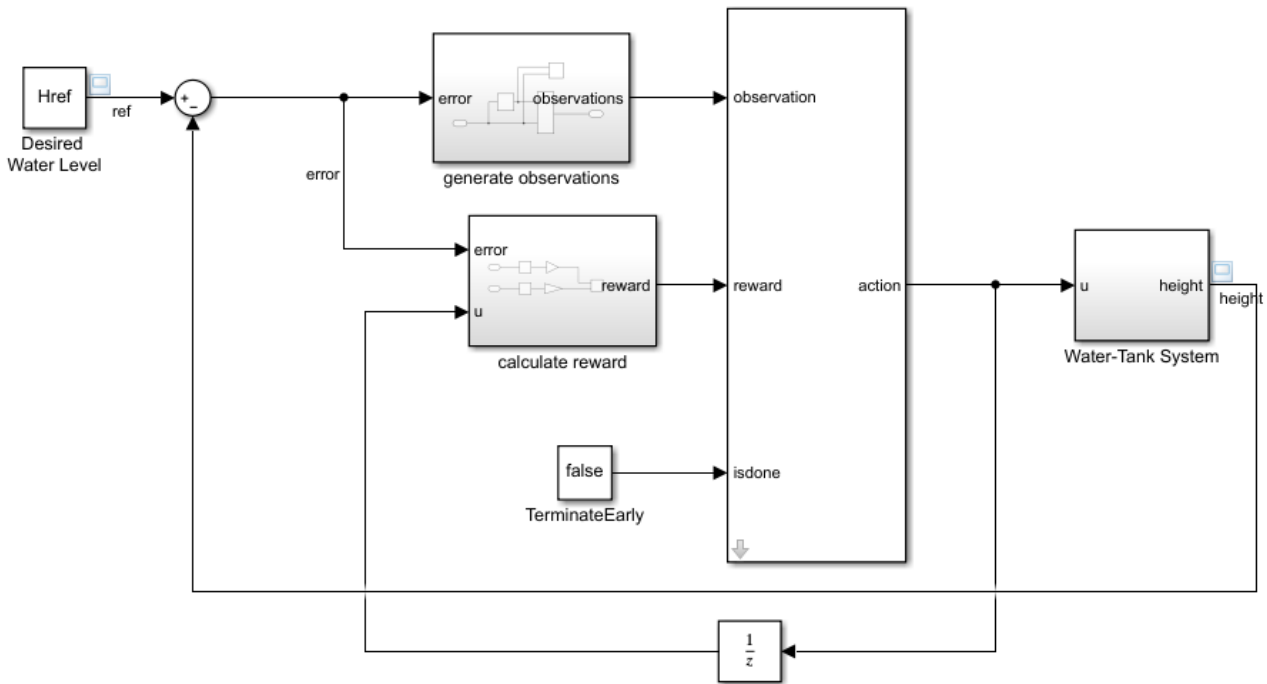
Create the environment interface object. To do so, use the `localCreatePIDEnv` function defined at the end of this example.

```
[env,obsInfo,actInfo] = localCreatePIDEnv(mdl);
```

Extract the observation and action dimensions for this environment.

```
numObservations = obsInfo.Dimension(1);
numActions = prod(actInfo.Dimension);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create TD3 Agent

Given observations, a TD3 agent decides which action to take using an actor representation. To create the actor, first create a deep neural network with the observation input and the action output. For more information, see `rlDeterministicActorRepresentation`.

You can model a PI controller as a neural network with one fully-connected layer with error and error integral observations.

$$u = \left[\int e \, \mathrm{dt} \ \ e \ \right] * [K_i \ K_p \ ]^T$$

Here:

- `u` is the output of the actor neural network.
- `Kp` and `Ki` are the absolute values of the neural network weights.

- $e = r - h$, h is the height of the tank, and r is the reference height.

Gradient descent optimization can drive the weights to negative values. To avoid negative weights, replace normal `fullyConnectedLayer` with a `fullyConnectedPILayer`. This layer ensures that the weights are positive by implementing the function $Y = \text{abs}(\text{WEIGHTS}) * X$. This layer is defined in `fullyConnectedPILayer.m`.

```
initialGain = single([1e-3 2]);
actorNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','state')
    fullyConnectedPILayer(initialGain, 'Action')];
actorOptions = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1);
actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'state'},'Action',{'Action'},actorOptions);
```

A TD3 agent approximates the long-term reward given observations and actions using two critic value-function representations. To create the critics, first create a deep neural network with two inputs, the observation and action, and one output. For more information on creating a deep neural network value function representation, see "Create Policy and Value Function Representations" on page 4-2.

To create the critics, use the `localCreateCriticNetwork` function defined at the end of this example. Use the same network structure for both critic representations.

```
criticNetwork = localCreateCriticNetwork(numObservations,numActions);
criticOpts = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1);

critic1 = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation','state','Action','action',criticOpts);
critic2 = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation','state','Action','action',criticOpts);
critic = [critic1 critic2];
```

Configure the agent using the following options.

- Set the agent to use the controller sample time `Ts`.
- Set the mini-batch size to 128 experience samples.
- Set the experience buffer length to 1e6.
- Set the exploration model and target policy smoothing model to use Gaussian noise with variance of 0.1.

Specify the TD3 agent options using `rlTD3AgentOptions`.

```
agentOpts = rlTD3AgentOptions(...
    'SampleTime',Ts,...
    'MiniBatchSize',128, ...
    'ExperienceBufferLength',1e6);
agentOpts.ExplorationModel.Variance = 0.1;
agentOpts.TargetPolicySmoothModel.Variance = 0.1;
```

Create the TD3 agent using the specified actor representation, critic representation, and agent options. For more information, see `rlTD3AgentOptions`.

```
agent = rlTD3Agent(actor,critic,agentOpts);
```

Train Agent

To train the agent, first specify the following training options.

- Run each training for at most `1000` episodes, with each episode lasting at most `100` time steps.
- Display the training progress in the Episode Manager (set the `Plots` option) and disable the command-line display (set the `Verbose` option).
- Stop training when the agent receives an average cumulative reward greater than -355 over 100 consecutive episodes. At this point, the agent can control the level of water in the tank.

For more information, see `rlTrainingOptions`.

```matlab
maxepisodes = 1000;
maxsteps = ceil(Tf/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes, ...
    'MaxStepsPerEpisode',maxsteps, ...
    'ScoreAveragingWindowLength',100, ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-355);
```

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```matlab
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load pretrained agent for the example.
    load('WaterTankPIDtd3.mat','agent')
end
```
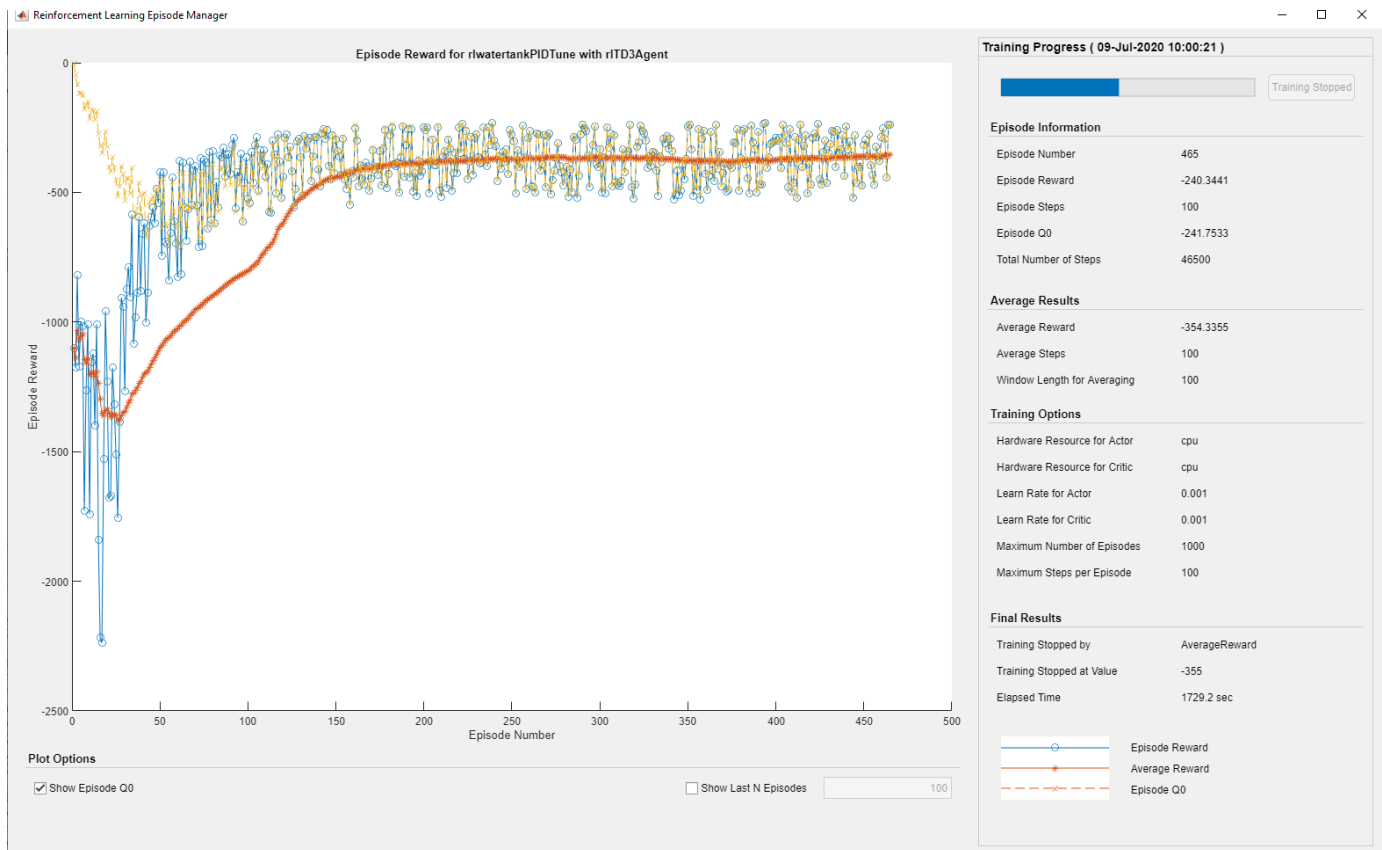
Validate Trained Agent

Validate the learned agent against the model by simulation.

```
simOpts = rlSimulationOptions('MaxSteps',maxsteps);
experiences = sim(env,agent,simOpts);
```

The integral and proportional gains of the PI controller are the absolute weights of the actor representation. To obtain the weights, first extract the learnable parameters from the actor.

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

Obtain the controller gains.

```
Ki = abs(parameters{1}(1))
```

```
Ki = single
    0.3958
```

```
Kp = abs(parameters{1}(2))
```

```
Kp = single
    8.0822
```

Apply the gains obtained from the RL agent to the original PI controller block and run a step-response simulation.

```
mdlTest = 'watertankLQG';
open_system(mdlTest);
set_param([mdlTest '/PID Controller'],'P',num2str(Kp))
set_param([mdlTest '/PID Controller'],'I',num2str(Ki))
sim(mdlTest)
```

Extract the step response information, LQG cost, and stability margin for the simulation. To compute the stability margin, use the `localStabilityAnalysis` function defined at the end of this example.

```
rlStep = simout;
rlCost = cost;
rlStabilityMargin = localStabilityAnalysis(mdlTest);
```

Apply the gains obtained using **Control System Tuner** to the original PI controller block and run a step-response simulation.

```
set_param([mdlTest '/PID Controller'],'P',num2str(Kp_CST))
set_param([mdlTest '/PID Controller'],'I',num2str(Ki_CST))
sim(mdlTest)
cstStep = simout;
cstCost = cost;
cstStabilityMargin = localStabilityAnalysis(mdlTest);
```

Compare Controller Performance

Plot the step response for each system.

```
figure
plot(cstStep)
hold on
plot(rlStep)
grid on
legend('Control System Tuner','RL','Location','southeast')
title('Step Response')
```
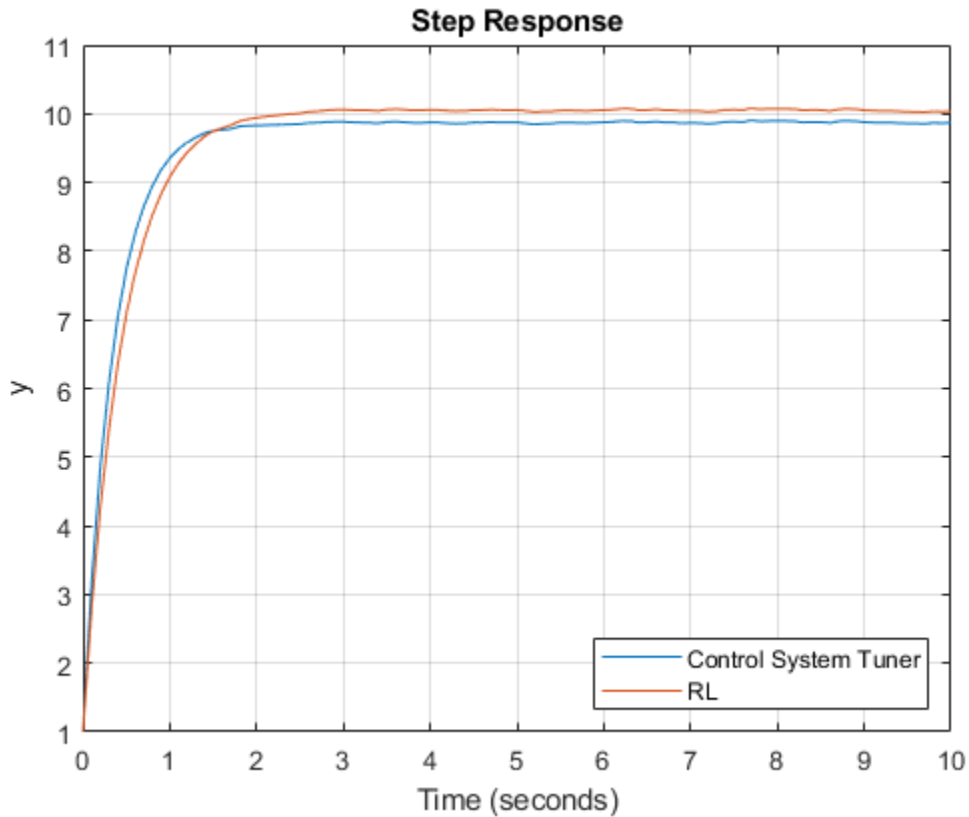
Step Response

Analyze the step response for both simulations.

```
rlStepInfo = stepinfo(rlStep.Data,rlStep.Time);
cstStepInfo = stepinfo(cstStep.Data,cstStep.Time);
stepInfoTable = struct2table([cstStepInfo rlStepInfo]);
stepInfoTable = removevars(stepInfoTable,{...
    'SettlingMin','SettlingMax','Undershoot','PeakTime'});
stepInfoTable.Properties.RowNames = {'Control System Tuner','RL'};
stepInfoTable
```

```
stepInfoTable=2×4 table
```

	RiseTime	SettlingTime	Overshoot	Peak
Control System Tuner	0.77322	1.3594	0.33125	9.9023
RL	0.97617	1.7408	0.40451	10.077

Analyze the stability for both simulations.

```
stabilityMarginTable = struct2table([cstStabilityMargin rlStabilityMargin]);
stabilityMarginTable = removevars(stabilityMarginTable,{...
    'GMFrequency','PMFrequency','DelayMargin','DMFrequency'});
stabilityMarginTable.Properties.RowNames = {'Control System Tuner','RL'};
stabilityMarginTable
```

```
stabilityMarginTable=2×3 table
```

	GainMargin	PhaseMargin	Stable

Control System Tuner	8.1616	84.122	true
RL	9.9226	84.241	true

Compare the cumulative LQG cost for the two controllers. The RL-tuned controller produces a slightly more optimal solution.

```
rlCumulativeCost = sum(rlCost.Data)

rlCumulativeCost = -375.9135

cstCumulativeCost = sum(cstCost.Data)

cstCumulativeCost = -376.9373
```

Both controllers produce stable responses, with the controller tuned using **Control System Tuner** producing a faster response. However, the RL tuning method produces a higher gain margin and a more optimal solution.

Local Functions

Function to create the water tank RL environment.

```
function [env,obsInfo,actInfo] = localCreatePIDEnv(mdl)

% Define the observation specification obsInfo and action specification actInfo.
obsInfo = rlNumericSpec([2 1]);
obsInfo.Name = 'observations';
obsInfo.Description = 'integrated error and error';

actInfo = rlNumericSpec([1 1]);
actInfo.Name = 'PID output';

% Build the environment interface object.
env = rlSimulinkEnv(mdl,[mdl '/RL Agent'],obsInfo,actInfo);

% Set a cutom reset function that randomizes the reference values for the model.
env.ResetFcn = @(in)localResetFcn(in,mdl);
end
```

Function to randomize the reference signal and initial height of the water tank at the beginning of each episode.

```
function in = localResetFcn(in,mdl)

% randomize reference signal
blk = sprintf([mdl '/Desired \nWater Level']);
hRef = 10 + 4*(rand-0.5);
in = setBlockParameter(in,blk,'Value',num2str(hRef));

% randomize initial height
hInit = 0;
blk = [mdl '/Water-Tank System/H'];
in = setBlockParameter(in,blk,'InitialCondition',num2str(hInit));

end
```

5-225

Function to linearize and compute stability margins of the SISO water tank system.

```matlab
function margin = localStabilityAnalysis(mdl)

io(1) = linio([mdl '/Sum1'],1,'input');
io(2) = linio([mdl '/Water-Tank System'],1,'openoutput');
op = operpoint(mdl);
op.Time = 5;
linsys = linearize(mdl,io,op);

margin = allmargin(linsys);
end
```

Function to create critic network.

```matlab
function criticNetwork = localCreateCriticNetwork(numObservations,numActions)
statePath = [
    featureInputLayer(numObservations,'Normalization','none','Name','state')
    fullyConnectedLayer(32,'Name','fc1')];
actionPath = [
    featureInputLayer(numActions,'Normalization','none','Name','action')
    fullyConnectedLayer(32,'Name','fc2')];
commonPath = [
    concatenationLayer(1,2,'Name','concat')
    reluLayer('Name','reluBody1')
    fullyConnectedLayer(32,'Name','fcBody')
    reluLayer('Name','reluBody2')
    fullyConnectedLayer(1,'Name','qvalue')];

criticNetwork = layerGraph();
criticNetwork = addLayers(criticNetwork,statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);

criticNetwork = connectLayers(criticNetwork,'fc1','concat/in1');
criticNetwork = connectLayers(criticNetwork,'fc2','concat/in2');
end
```

See Also
`rlTD3Agent` | `train`

More About
- "Twin-Delayed Deep Deterministic Policy Gradient Agents" on page 3-21

Train Custom LQR Agent

This example shows how to train a custom linear quadratic regulation (LQR) agent to control a discrete-time linear system modeled in MATLAB®.

Create Linear System Environment

The reinforcement learning environment for this example is a discrete-time linear system. The dynamics for the system are given by

$$x_{t+1} = Ax_t + Bu_t$$

The feedback control law is

$$u_t = -Kx_t$$

The control objective is to minimize the quadratic cost: $J = \sum_{t=0}^{\infty}(x_t'Qx_t + u_t'Ru_t)$.

In this example, the system matrices are

$$A = \begin{bmatrix} 1.05 & 0.05 & 0.05 \\ 0.05 & 1.05 & 0.05 \\ 0 & 0.05 & 1.05 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.1 & 0 & 0.2 \\ 0.1 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$$

```
A = [1.05,0.05,0.05;0.05,1.05,0.05;0,0.05,1.05];
B = [0.1,0,0.2;0.1,0.5,0;0,0,0.5];
```

The quadratic cost matrices are:

$$Q = \begin{bmatrix} 10 & 3 & 1 \\ 3 & 5 & 4 \\ 1 & 4 & 9 \end{bmatrix}$$

$$R = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$$

```
Q = [10,3,1;3,5,4;1,4,9];
R = 0.5*eye(3);
```

For this environment, the reward at time t is given by $r_t = -x_t'Qx_t - u_t'Ru_t$, which is the negative of the quadratic cost. Therefore, maximizing the reward minimizes the cost. The initial conditions are set randomly by the reset function.

Create the MATLAB environment interface for this linear system and reward. The `myDiscreteEnv` function creates an environment by defining custom `step` and `reset` functions. For more information on creating such a custom environment, see "Create MATLAB Environment Using Custom Functions" on page 2-33.

```
env = myDiscreteEnv(A,B,Q,R);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create Custom LQR Agent

For the LQR problem, the Q-function for a given control gain K can be defined as $Q_K(x, u) = \begin{bmatrix} x \\ u \end{bmatrix}' H_K \begin{bmatrix} x \\ u \end{bmatrix}$, where $H_K = \begin{bmatrix} H_{xx} & H_{xu} \\ H_{ux} & H_{uu} \end{bmatrix}$ is a symmetric, positive definite matrix.

The control law to maximize Q_K is $u = -(H_{uu})^{-1} H_{ux} x$, and the feedback gain is $K = (H_{uu})^{-1} H_{ux}$.

The matrix H_K contains $m = \frac{1}{2} n(n + 1)$ distinct element values, where n is the sum of the number of states and number of inputs. Denote θ as the vector corresponding to these m elements, where the off-diagonal elements in H_K are multiplied by two.

Represent the Q-function by θ, where θ contains the parameters to be learned.

$Q_K(x, u) = \theta'(K)\phi(x, u)$, where $\phi(x, u)$ is the quadratic basis function in terms of x and u.

The LQR agent starts with a stabilizing controller K_0. To get an initial stabilizing controller, place the poles of the closed-loop system $A - BK_0$ inside the unit circle.

```
K0 = place(A,B,[0.4,0.8,0.5]);
```

To create a custom agent, you must create a subclass of the `rl.agent.CustomAgent` abstract class. For the custom LQR agent, the defined custom subclass is `LQRCustomAgent`. For more information, see "Custom Agents" on page 3-37. Create the custom LQR agent using Q, R, and K_0. The agent does not require information on the system matrices A and B.

```
agent = LQRCustomAgent(Q,R,K0);
```

For this example, set the agent discount factor to one. To use a discounted future reward, set the discount factor to a value less than one.

```
agent.Gamma = 1;
```

Because the linear system has three states and three inputs, the total number of learnable parameters is $m = 21$. To ensure satisfactory performance of the agent, set the number of parameter estimates N_p to be greater than twice the number of learnable parameters. In this example, the value is $N_p = 45$.

```
agent.EstimateNum = 45;
```

To get good estimation results for θ, you must apply a persistently excited exploration model to the system. In this example, encourage model exploration by adding white noise to the controller output: $u_t = -Kx_t + e_t$. In general, the exploration model depends on the system models.

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

- Run each training episode for at most `10` episodes, with each episode lasting at most `50` time steps.
- Display the command line display (set the `Verbose` option) and disable the training progress in the Episode Manager dialog box (set the `Plots` option).

For more information, see `rlTrainingOptions`.

```
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',10, ...
    'MaxStepsPerEpisode',50, ...
    'Verbose',true, ...
    'Plots','none');
```

Train the agent using the `train` function.

```
trainingStats = train(agent,env,trainingOpts);
```

```
Episode:   1/ 10 | Episode Reward : -55.16 | Episode Steps:   50 | Avg Reward : -55.16 | Step Cou
Episode:   2/ 10 | Episode Reward : -12.52 | Episode Steps:   50 | Avg Reward : -33.84 | Step Cou
Episode:   3/ 10 | Episode Reward : -15.59 | Episode Steps:   50 | Avg Reward : -27.76 | Step Cou
Episode:   4/ 10 | Episode Reward : -22.22 | Episode Steps:   50 | Avg Reward : -26.37 | Step Cou
Episode:   5/ 10 | Episode Reward : -14.32 | Episode Steps:   50 | Avg Reward : -23.96 | Step Cou
Episode:   6/ 10 | Episode Reward : -19.23 | Episode Steps:   50 | Avg Reward : -16.78 | Step Cou
Episode:   7/ 10 | Episode Reward : -34.14 | Episode Steps:   50 | Avg Reward : -21.10 | Step Cou
Episode:   8/ 10 | Episode Reward : -13.95 | Episode Steps:   50 | Avg Reward : -20.77 | Step Cou
Episode:   9/ 10 | Episode Reward : -36.01 | Episode Steps:   50 | Avg Reward : -23.53 | Step Cou
Episode:  10/ 10 | Episode Reward : -12.43 | Episode Steps:   50 | Avg Reward : -23.15 | Step Cou
```

Simulate Agent and Compare with Optimal Solution

To validate the performance of the trained agent, simulate it within the MATLAB environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',20);
experience = sim(env,agent,simOptions);
totalReward = sum(experience.Reward)
```

```
totalReward = -20.1306
```

You can compute the optimal solution for the LQR problem using the `dlqr` function.

```
[Koptimal,P] = dlqr(A,B,Q,R);
```

The optimal reward is given by $J_{\mathrm{optimal}} = -x_0'\mathrm{P}x_0$.

```
x0 = experience.Observation.obs1.getdatasamples(1);
Joptimal = -x0'*P*x0;
```

Compute the error in the reward between the trained LQR agent and the optimal LQR solution.

```
rewardError = totalReward - Joptimal
```

```
rewardError = 1.5270e-06
```

View the history of the 2-norm of error in the gains between the trained LQR agent and the optimal LQR solution.

```
% number of gain updates
len = agent.KUpdate;
```

```
err = zeros(len,1);
for i = 1:len
    % norm of error in the gain
    err(i) = norm(agent.KBuffer{i}-Koptimal);
end
plot(err,'b*-')
```
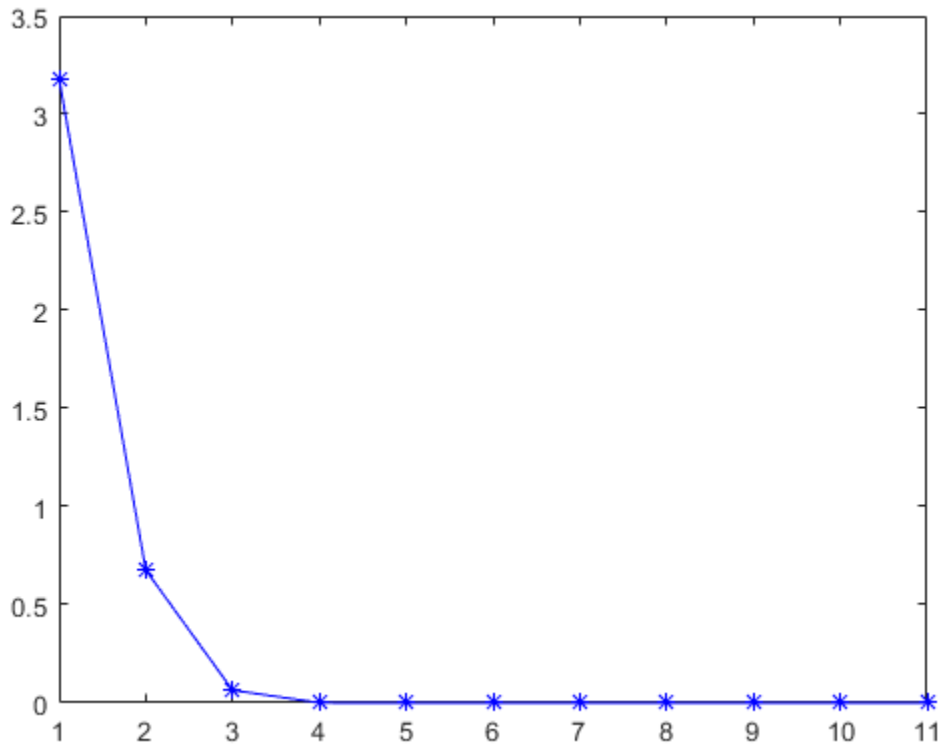

Compute the norm of final error for the feedback gain.

```
gainError = norm(agent.K - Koptimal)
```

```
gainError = 4.1670e-11
```

Overall, the trained agent finds an LQR solution that is close to the true optimal LQR solution.

See Also
```
train
```

More About
- "Custom Agents" on page 3-37
- "Train Reinforcement Learning Agents" on page 5-2

Train Reinforcement Learning Policy Using Custom Training Loop

This example shows how to define a custom training loop for a reinforcement learning policy. You can use this workflow to train reinforcement learning policies with your own custom training algorithms rather than using one of the built-in agents from the Reinforcement Learning Toolbox™ software.

Using this workflow, you can train policies that use any of the following policy and value function representations.

- `rlStochasticActorRepresentation` — Stochastic actor representation
- `rlDeterministicActorRepresentation` — Deterministic actor representation
- `rlValueRepresentation` — Value function critic representation
- `rlQValueRepresentation` — Q-Value function critic representation

In this example, a stochastic actor policy with a discrete action space is trained using the REINFORCE algorithm (with no baseline). For more information on the REINFORCE algorithm, see "Policy Gradient Agents" on page 3-13.

Fix the random generator seed for reproducibility.

```
rng(0)
```

For more information on the functions you can use for custom training, see Functions for Custom Training on page 5-0   .

Environment

For this example, a reinforcement learning policy is trained in a discrete cart-pole environment. The objective in this environment is to balance the pole by applying forces (actions) on the cart. Create the environment using the `rlPredefinedEnv` function.

```
env = rlPredefinedEnv('CartPole-Discrete');
```

Extract the observation and action specifications from the environment.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Obtain the number of observations (`numObs`) and actions (`numAct`).

```
numObs = obsInfo.Dimension(1);
numAct = actInfo.Dimension(1);
```

For more information on this environment, see "Load Predefined Control System Environments" on page 2-15.

Policy

The reinforcement learning policy in this example is a discrete-action stochastic policy. It is represented by a deep neural network that contains `fullyConnectedLayer`, `reluLayer`, and `softmaxLayer` layers. This network outputs probabilities for each discrete action given the current observations. The `softmaxLayer` ensures that the representation outputs probability values in the range [0 1] and that all probabilities sum to 1.

Create the deep neural network for the actor.

```
actorNetwork = [featureInputLayer(numObs,'Normalization','none','Name','state')
                fullyConnectedLayer(24,'Name','fc1')
                reluLayer('Name','relu1')
                fullyConnectedLayer(24,'Name','fc2')
                reluLayer('Name','relu2')
                fullyConnectedLayer(2,'Name','output')
                softmaxLayer('Name','actionProb')];
```

Create the actor representation using an `rlStochasticActorRepresentation` object.

```
actorOpts = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1);
actor = rlStochasticActorRepresentation(actorNetwork,...
    obsInfo,actInfo,'Observation','state',actorOpts);
```

For this example, the loss function for the policy is implemented in `actorLossFunction` on page 5-0 .

Set the loss function using the `setLoss` function.

```
actor = setLoss(actor,@actorLossFunction);
```

Training Setup

Configure the training to use the following options:

- Set up the training to last at most 5000 episodes, with each episode lasting at most 250 steps.
- To calculate the discounted reward, choose a discount factor of 0.995.
- Terminate the training after the maximum number of episodes is reached or when the average reward across 100 episodes reaches the value of 220.

```
numEpisodes = 5000;
maxStepsPerEpisode = 250;
discountFactor = 0.995;
aveWindowSize = 100;
trainingTerminationValue = 220;
```

Create a vector for storing the cumulative reward for each training episode.

```
episodeCumulativeRewardVector = [];
```

Create a figure for training visualization using the `hBuildFigure` on page 5-0 helper function.

```
[trainingPlot,lineReward,lineAveReward] = hBuildFigure;
```

Custom Training loop

The algorithm for the custom training loop is as follows. For each episode:

1 Reset the environment.
2 Create buffers for storing experience information: observations, actions, and rewards.
3 Generate experiences until a terminal condition occurs. To do so, evaluate the policy to get actions, apply those actions to the environment, and obtain the resulting observations and rewards. Store the actions, observations, and rewards in buffers.
4 Collect the training data as a batch of experiences.

5 Compute the episode Monte Carlo return, which is the discounted future reward.

6 Compute the gradient of the loss function with respect to the policy representation parameters.

7 Update the actor representation using the computed gradients.

8 Update the training visualization.

9 Terminate training if the policy is sufficiently trained.

```matlab
% Enable the training visualization plot.
set(trainingPlot,'Visible','on');

% Train the policy for the maximum number of episodes or until the average
% reward indicates that the policy is sufficiently trained.
for episodeCt = 1:numEpisodes

    % 1. Reset the environment at the start of the episode
    obs = reset(env);

    episodeReward = zeros(maxStepsPerEpisode,1);

    % 2. Create buffers to store experiences. The dimensions for each buffer
    % must be as follows.
    %
    % For observation buffer:
    %     numberOfObservations x numberOfObservationChannels x batchSize
    %
    % For action buffer:
    %     numberOfActions x numberOfActionChannels x batchSize
    %
    % For reward buffer:
    %     1 x batchSize
    %
    observationBuffer = zeros(numObs,1,maxStepsPerEpisode);
    actionBuffer = zeros(numAct,1,maxStepsPerEpisode);
    rewardBuffer = zeros(1,maxStepsPerEpisode);

    % 3. Generate experiences for the maximum number of steps per
    % episode or until a terminal condition is reached.
    for stepCt = 1:maxStepsPerEpisode

        % Compute an action using the policy based on the current
        % observation.
        action = getAction(actor,{obs});

        % Apply the action to the environment and obtain the resulting
        % observation and reward.
        [nextObs,reward,isdone] = step(env,action{1});

        % Store the action, observation, and reward experiences in buffers.
        observationBuffer(:,:,stepCt) = obs;
        actionBuffer(:,:,stepCt) = action{1};
        rewardBuffer(:,stepCt) = reward;

        episodeReward(stepCt) = reward;
        obs = nextObs;

        % Stop if a terminal condition is reached.
        if isdone
```

```matlab
            break;
        end

    end

    % 4. Create training data. Training is performed using batch data. The
    % batch size equal to the length of the episode.
    batchSize = min(stepCt,maxStepsPerEpisode);
    observationBatch = observationBuffer(:,:,1:batchSize);
    actionBatch = actionBuffer(:,:,1:batchSize);
    rewardBatch = rewardBuffer(:,1:batchSize);

    % Compute the discounted future reward.
    discountedReturn = zeros(1,batchSize);
    for t = 1:batchSize
        G = 0;
        for k = t:batchSize
            G = G + discountFactor ^ (k-t) * rewardBatch(k);
        end
        discountedReturn(t) = G;
    end

    % 5. Organize data to pass to the loss function.
    lossData.batchSize = batchSize;
    lossData.actInfo = actInfo;
    lossData.actionBatch = actionBatch;
    lossData.discountedReturn = discountedReturn;

    % 6. Compute the gradient of the loss with respect to the policy
    % parameters.
    actorGradient = gradient(actor,'loss-parameters',...
        {observationBatch},lossData);

    % 7. Update the actor network using the computed gradients.
    actor = optimize(actor,actorGradient);

    % 8. Update the training visualization.
    episodeCumulativeReward = sum(episodeReward);
    episodeCumulativeRewardVector = cat(2,...
        episodeCumulativeRewardVector,episodeCumulativeReward);
    movingAveReward = movmean(episodeCumulativeRewardVector,...
        aveWindowSize,2);
    addpoints(lineReward,episodeCt,episodeCumulativeReward);
    addpoints(lineAveReward,episodeCt,movingAveReward(end));
    drawnow;

    % 9. Terminate training if the network is sufficiently trained.
    if max(movingAveReward) > trainingTerminationValue
        break
    end

end
```
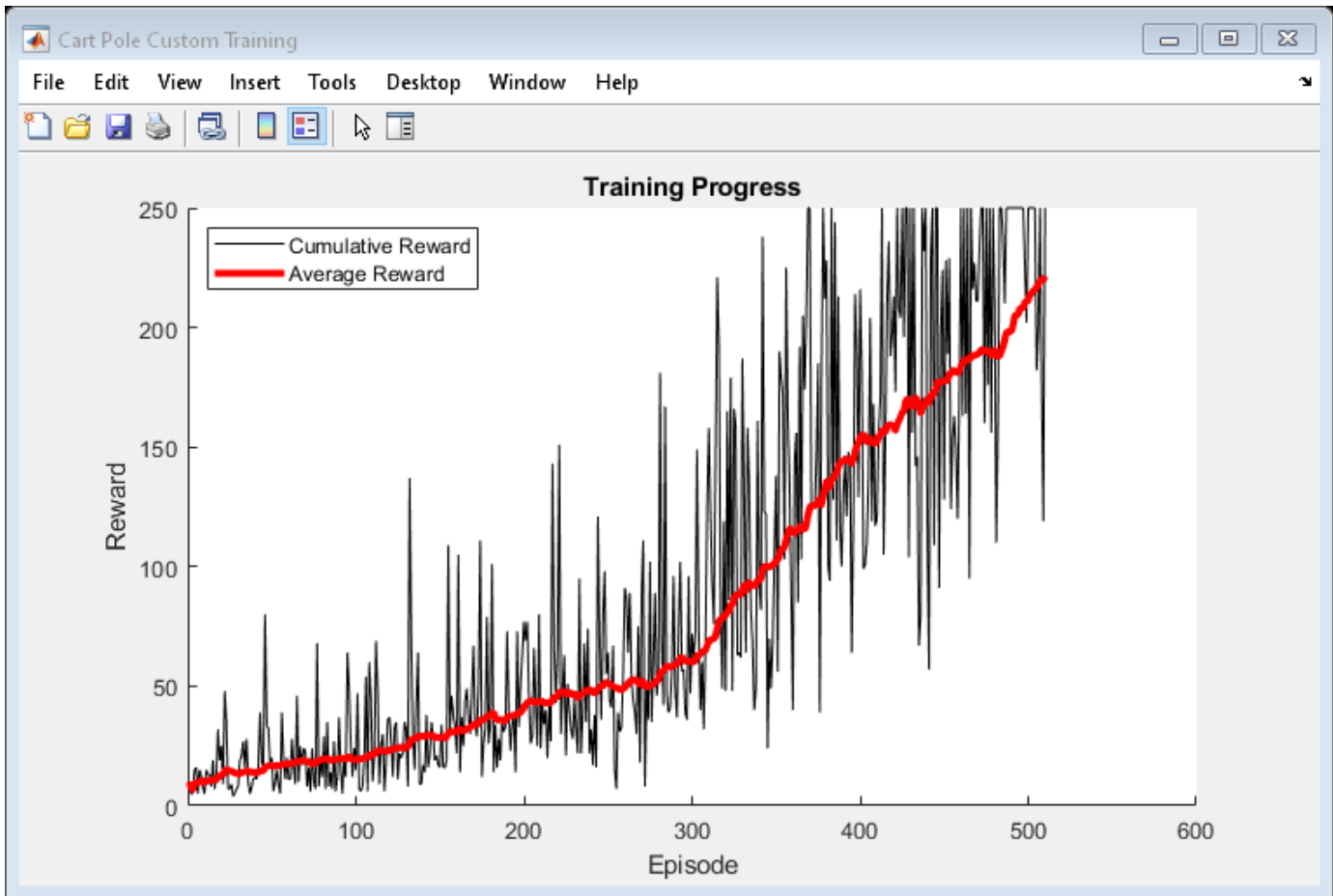
Simulation

After training, simulate the trained policy.

Before simulation, reset the environment.

```
obs = reset(env);
```

Enable the environment visualization, which is updated each time the environment step function is called.

```
plot(env)
```

For each simulation step, perform the following actions.

1   Get the action by sampling from the policy using the `getAction` function.
2   Step the environment using the obtained action value.
3   Terminate if a terminal condition is reached.

```
for stepCt = 1:maxStepsPerEpisode

    % Select action according to trained policy
    action = getAction(actor,{obs});
```

```
    % Step the environment
    [nextObs,reward,isdone] = step(env,action{1});

    % Check for terminal condition
    if isdone
        break
    end

    obs = nextObs;

end
```
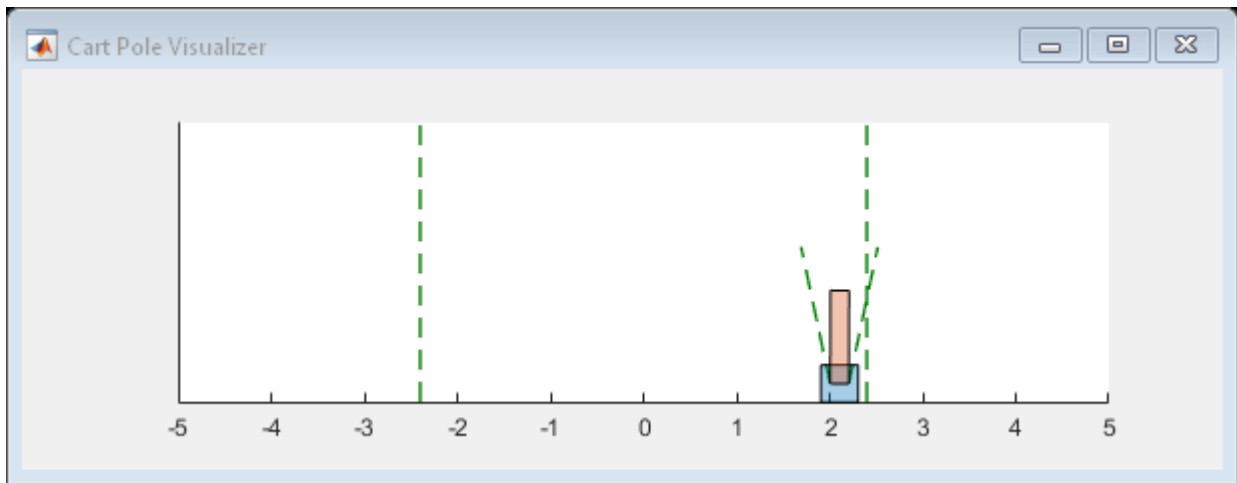

Functions for Custom Training

To obtain actions and value functions for given observations from Reinforcement Learning Toolbox policy and value function representations, you can use the following functions.

*   `getValue` — Obtain the estimated state value or state-action value function.
*   `getAction` — Obtain the action from an actor representation based on the current observation.
*   `getMaxQValue` — Obtain the estimated maximum state-action value function for a discrete Q-value representation.

If your policy or value function representation is a recurrent neural network, that is, a neural network with at least one layer that has hidden state information, the preceding functions can return the current network state. You can use the following function syntaxes to get and set the state of your representation.

*   `state = getState(rep)` — Obtain the state of representation `rep`.
*   `newRep = setState(oldRep,state)` — Set the state of representation `oldRep`, and return the result in `oldRep`.
*   `newRep = resetState(oldRep)` — Reset all state values of `oldRep` to zero and return the result in `newRep`.

You can get and set the learnable parameters of your representation using the `getLearnableParameters` and `setLearnableParameters` function, respectively.

In addition to these functions, you can use the `setLoss`, `gradient`, `optimize`, and `syncParameters` functions to set parameters and compute gradients for your policy and value function representations.

setLoss

The policy is trained in a stochastic gradient ascent manner where the gradients of a loss function is used to update the network. For custom training, you can set the loss function using the `setLoss` function. To do so, use the following syntax.

`newRep = setLoss(oldRep,lossFcn)`

Here:

- `oldRep` is a policy or value function representation object.
- `lossFcn` is the name of a custom loss function or a handle to a custom loss function.
- `newRep` is equivalent to `oldRep`, except that the loss function has been added to the representation.

gradient

The `gradient` function computes the gradients of the representation loss function. You can compute several different gradients. For example, to compute the gradient of the representation outputs with respect to its inputs, use the following syntax.

`grad = gradient(rep,"output-input",inputData)`

Here:

- `rep` is a policy or value function representation object.
- `inputData` contains values for the input channels to the representation.
- `grad` contains the computed gradients.

For more information, at the MATLAB command line, type `help rl.representation.rlAbstractRepresentation.gradient`.

optimize

The `optimize` function updates the learnable parameters of the representation based on computed gradients. To update the parameters the gradients, use the following syntax.

`newRep = optimize(oldRep,grad)`

Here, `oldRep` is a policy or value function representation object and `grad` contains gradients computed using the `gradient` function. `newRep` has the same structure as `oldRep`, but its parameters are updated.

syncParameters

The `syncParameters` function updates the learnable parameters of one policy or value function representation based on those of another representation. This function is useful for updating a target actor or critic representation, as is done for DDPG agents. To synchronize parameters values between two representations, use the following syntax.

`newTargetRep = syncParameters(oldTargetRep,sourceRep,smoothFactor)`

Here:

- `oldTargetRep` is a policy or value function representation object with parameters θ_{old}.

- `sourceRep` is a policy or value function representation object with the same structure as `oldTargetRep`, but with parameters θ_{source}.

- `smoothFactor` is a smoothing factor (τ) for the update.

- `newTargetRep` has the same structure as `oldRep`, but its parameters are $\theta_{new} = \tau\theta_{source} + (1 - \tau)\theta_{old}$.

Loss Function

The loss function in the REINFORCE algorithm is the product of the discounted reward and the log of the policy, summed across all time steps. The discounted reward calculated in the custom training loop must be resized to make it compatible for multiplication with the policy.

```
function loss = actorLossFunction(policy, lossData)

    % Create the action indication matrix.
    batchSize = lossData.batchSize;
    Z = repmat(lossData.actInfo.Elements',1,batchSize);
    actionIndicationMatrix = lossData.actionBatch(:,:) == Z;

    % Resize the discounted return to the size of policy.
    G = actionIndicationMatrix .* lossData.discountedReturn;
    G = reshape(G,size(policy));

    % Round any policy values less than eps to eps.
    policy(policy < eps) = eps;

    % Compute the loss.
    loss = -sum(G .* log(policy),'all');

end
```

Helper Function

The following helper function creates a figure for training visualization.

```
function [trainingPlot, lineReward, lineAveReward] = hBuildFigure()
    plotRatio = 16/9;
    trainingPlot = figure(...
                'Visible','off',...
                'HandleVisibility','off', ...
                'NumberTitle','off',...
                'Name','Cart Pole Custom Training');
    trainingPlot.Position(3) = plotRatio * trainingPlot.Position(4);

    ax = gca(trainingPlot);

    lineReward = animatedline(ax);
    lineAveReward = animatedline(ax,'Color','r','LineWidth',3);
    xlabel(ax,'Episode');
    ylabel(ax,'Reward');
    legend(ax,'Cumulative Reward','Average Reward','Location','northwest')
```

```
    title(ax,'Training Progress');
end
```

See Also

`train`

More About

- "Custom Agents" on page 3-37
- "Train Reinforcement Learning Agents" on page 5-2

Create Agent for Custom Reinforcement Learning Algorithm

This example shows how to create a custom agent for your own custom reinforcement learning algorithm. Doing so allows you to leverage the following built-in functionality from the Reinforcement Learning Toolbox™ software.

- Access to all agent functions, including `train` and `sim`
- Visualize training progress using the Episode Manager
- Train agents within a Simulink® environment

In this example, you convert a custom REINFORCE training loop into a custom agent class. For more information on the REINFORCE custom train loop, see "Train Reinforcement Learning Policy Using Custom Training Loop" on page 5-231. For more information on writing custom agent classes, see "Custom Agents" on page 3-37.

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create Environment

Create the same training environment used in the "Train Reinforcement Learning Policy Using Custom Training Loop" on page 5-231 example. The environment is a cart-pole balancing environment with a discrete action space. Create the environment using the `rlPredefinedEnv` function.

```
env = rlPredefinedEnv('CartPole-Discrete');
```

Extract the observation and action specifications from the environment.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Obtain the number of observations (`numObs`) and actions (`numAct`).

```
numObs = obsInfo.Dimension(1);
numAct = numel(actInfo.Elements);
```

For more information on this environment, see "Load Predefined Control System Environments" on page 2-15.

Define Policy

The reinforcement learning policy in this example is a discrete-action stochastic policy. It is represented by a deep neural network that contains `fullyConnectedLayer`, `reluLayer`, and `softmaxLayer` layers. This network outputs probabilities for each discrete action given the current observations. The `softmaxLayer` ensures that the representation outputs probability values in the range [0 1] and that all probabilities sum to 1.

Create the deep neural network for the actor.

```
actorNetwork = [featureInputLayer(numObs,'Normalization','none','Name','state')
                fullyConnectedLayer(24,'Name','fc1')
                reluLayer('Name','relu1')
                fullyConnectedLayer(24,'Name','fc2')
```

```
                    reluLayer('Name','relu2')
                    fullyConnectedLayer(2,'Name','output')
                    softmaxLayer('Name','actionProb')];
```

Create the actor representation using an `rlStochasticActorRepresentation` object.

```
actorOpts = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1);
actor = rlStochasticActorRepresentation(actorNetwork,...
    obsInfo,actInfo,'Observation','state',actorOpts);
```

Custom Agent Class

To define your custom agent, first create a class that is a subclass of the `rl.agent.CustomAgent` class. The custom agent class for this example is defined in `CustomReinforceAgent.m`.

The `CustomReinforceAgent` class has the following class definition, which indicates the agent class name and the associated abstract agent.

```
classdef CustomReinforceAgent < rl.agent.CustomAgent
```

To define your agent you must specify the following:

- Agent properties
- Constructor function
- Critic representation that estimates the discounted long-term reward (if required for learning)
- Actor representation that selects an action based on the current observation (if required for learning)
- Required agent methods
- Optional agent methods

Agent Properties

In the `properties` section of the class file, specify any parameters necessary for creating and training the agent.

The `rl.Agent.CustomAgent` class already includes properties for the agent sample time (`SampleTime`) and the action and observation specifications (`ActionInfo` and `ObservationInfo`, respectively).

The custom REINFORCE agent defines the following additional agent properties.

```
properties
    % Actor representation
    Actor

    % Agent options
    Options

    % Experience buffer
    ObservationBuffer
    ActionBuffer
    RewardBuffer
end

properties (Access = private)
```

```matlab
        % Training utilities
        Counter
        NumObservation
        NumAction
end
```

Constructor Function

To create your custom agent, you must define a constructor function. The constructor function performs the following actions.

- Defines the action and observation specifications. For more information about creating these specifications, see rlNumericSpec and rlFiniteSetSpec.
- Sets the agent properties.
- Calls the constructor of the base abstract class.
- Defines the sample time (required for training in Simulink environments).

For example, the CustomREINFORCEAgent constructor defines action and observation spaces based on the input actor representation.

```matlab
function obj = CustomReinforceAgent(Actor,Options)
    %CUSTOMREINFORCEAGENT Construct custom agent
    %    AGENT = CUSTOMREINFORCEAGENT(ACTOR,OPTIONS) creates custom
    %    REINFORCE AGENT from rlStochasticActorRepresentation ACTOR
    %    and structure OPTIONS. OPTIONS has fields:
    %        - DiscountFactor
    %        - MaxStepsPerEpisode

    % (required) Call the abstract class constructor.
    obj = obj@rl.agent.CustomAgent();
    obj.ObservationInfo = Actor.ObservationInfo;
    obj.ActionInfo = Actor.ActionInfo;

    % (required for Simulink environment) Register sample time.
    % For MATLAB environment, use -1.
    obj.SampleTime = -1;

    % (optional) Register actor and agent options.
    Actor = setLoss(Actor,@lossFunction);
    obj.Actor = Actor;
    obj.Options = Options;

    % (optional) Cache the number of observations and actions.
    obj.NumObservation = prod(obj.ObservationInfo.Dimension);
    obj.NumAction = prod(obj.ActionInfo.Dimension);

    % (optional) Initialize buffer and counter.
    reset(obj);
end
```

The constructor sets the loss function of the actor representation using a function handle to lossFunction, which is implemented as a local function in CustomREINFORCEAgent.m.

```matlab
function loss = lossFunction(policy,lossData)

    % Create the action indication matrix.
```

```
    batchSize = lossData.batchSize;
    Z = repmat(lossData.actInfo.Elements',1,batchSize);
    actionIndicationMatrix = lossData.actionBatch(:,:) == Z;

    % Resize the discounted return to the size of policy.
    G = actionIndicationMatrix .* lossData.discountedReturn;
    G = reshape(G,size(policy));

    % Round any policy values less than eps to eps.
    policy(policy < eps) = eps;

    % Compute the loss.
    loss = -sum(G .* log(policy),'all');

end
```

Required Functions

To create a custom reinforcement learning agent you must define the following implementation functions.

- `getActionImpl` — Evaluate agent policy and select an agent during simulation.
- `getActionWithExplorationImpl` — Evaluate policy and select an action with exploration during training.
- `learnImpl` — How the agent learns from the current experience

To call these functions in your own code, use the wrapper methods from the abstract base class. For example, to call `getActionImpl`, use `getAction`. The wrapper methods have the same input and output arguments as the implementation methods.

getActionImpl Function

The `getActionImpl` function is used to evaluate the policy of your agent and select an action when simulating the agent using the `sim` function. This function must have the following signature, where `obj` is the agent object, `Observation` is the current observation, and `Action` is the selected action.

```
function Action = getActionImpl(obj,Observation)
```

For the custom REINFORCE agent, you select an action by calling the `getAction` function for the actor representation. The discrete `rlStochasticActorRepresentation` generates a discrete distribution from an observation and samples an action from this distribution.

```
function Action = getActionImpl(obj,Observation)
    % Compute an action using the policy given the current
    % observation.

    Action = getAction(obj.Actor,Observation);
end
```

getActionWithExplorationImpl Function

The `getActionWithExplorationImpl` function selects an action using the exploration model of your agent when training the agent using the `train` function. Using this function you can implement exploration techniques such as epsilon-greedy exploration or the addition of Gaussian noise. This function must have the following signature, where `obj` is the agent object, `Observation` is the current observation, and `Action` is the selected action.

```
function Action = getActionWithExplorationImpl(obj,Observation)
```

For the custom REINFORCE agent, the `getActionWithExplorationImpl` function is the same as `getActionImpl`. By default, stochastic actors always explore, that is, they always select an action based on a probability distribution.

```
function Action = getActionWithExplorationImpl(obj,Observation)
    % Compute an action using the exploration policy given the
    % current observation.

    % REINFORCE: Stochastic actors always explore by default
    % (sample from a probability distribution)
    Action = getAction(obj.Actor,Observation);
end
```

`learnImpl` Function

The `learnImpl` function defines how the agent learns from the current experience. This function implements the custom learning algorithm of your agent by updating the policy parameters and selecting an action with exploration for the next state. This function must have the following signature, where `obj` is the agent object, `Experience` is the current agent experience, and `Action` is the selected action.

```
function Action = learnImpl(obj,Experience)
```

The agent experience is the cell array `Experience = {state,action,reward,nextstate,isdone}`. Here:

- `state` is the current observation.
- `action` is the current action. This is different from the output argument `Action`, which is an action for the next state.
- `reward` is the current reward.
- `nextState` is the next observation.
- `isDone` is a logical flag indicating that the training episode is complete.

For the custom REINFORCE agent, replicate steps 2 through 7 of the custom training loop in "Train Reinforcement Learning Policy Using Custom Training Loop" on page 5-231. You omit steps 1, 8, and 9 since you will use the built-in `train` function to train your agent.

```
function Action = learnImpl(obj,Experience)
    % Define how the agent learns from an Experience, which is a
    % cell array with the following format.
    %   Experience = {observation,action,reward,nextObservation,isDone}

    % Reset buffer at the beginning of the episode.
    if obj.Counter < 2
        resetBuffer(obj);
    end

    % Extract data from experience.
    Obs = Experience{1};
    Action = Experience{2};
    Reward = Experience{3};
    NextObs = Experience{4};
    IsDone = Experience{5};
```

```matlab
        % Save data to buffer.
        obj.ObservationBuffer(:,:,obj.Counter) = Obs{1};
        obj.ActionBuffer(:,:,obj.Counter) = Action{1};
        obj.RewardBuffer(:,obj.Counter) = Reward;

        if ~IsDone
            % Choose an action for the next state.

            Action = getActionWithExplorationImpl(obj, NextObs);
            obj.Counter = obj.Counter + 1;
        else
            % Learn from episodic data.

            % Collect data from the buffer.
            BatchSize = min(obj.Counter,obj.Options.MaxStepsPerEpisode);
            ObservationBatch = obj.ObservationBuffer(:,:,1:BatchSize);
            ActionBatch = obj.ActionBuffer(:,:,1:BatchSize);
            RewardBatch = obj.RewardBuffer(:,1:BatchSize);

            % Compute the discounted future reward.
            DiscountedReturn = zeros(1,BatchSize);
            for t = 1:BatchSize
                G = 0;
                for k = t:BatchSize
                    G = G + obj.Options.DiscountFactor ^ (k-t) * RewardBatch(k);
                end
                DiscountedReturn(t) = G;
            end

            % Organize data to pass to the loss function.
            LossData.batchSize = BatchSize;
            LossData.actInfo = obj.ActionInfo;
            LossData.actionBatch = ActionBatch;
            LossData.discountedReturn = DiscountedReturn;

            % Compute the gradient of the loss with respect to the
            % actor parameters.
            ActorGradient = gradient(obj.Actor,'loss-parameters',...
                {ObservationBatch},LossData);

            % Update the actor parameters using the computed gradients.
            obj.Actor = optimize(obj.Actor,ActorGradient);

            % Reset the counter.
            obj.Counter = 1;
        end
end
```

Optional Functions

Optionally, you can define how your agent is reset at the start of training by specifying a `resetImpl` function with the following function signature, where `obj` is the agent object.

```matlab
function resetImpl(obj)
```

Using this function, you can set the agent into a know or random condition before training.

```matlab
function resetImpl(obj)
    % (Optional) Define how the agent is reset before training/

    resetBuffer(obj);
    obj.Counter = 1;
end
```

Also, you can define any other helper functions in your custom agent class as required. For example, the custom REINFORCE agent defines a `resetBuffer` function for reinitializing the experience buffer at the beginning of each training episode.

```matlab
function resetBuffer(obj)
    % Reinitialize all experience buffers.

    obj.ObservationBuffer = zeros(obj.NumObservation,1,obj.Options.MaxStepsPerEpisode);
    obj.ActionBuffer = zeros(obj.NumAction,1,obj.Options.MaxStepsPerEpisode);
    obj.RewardBuffer = zeros(1,obj.Options.MaxStepsPerEpisode);
end
```

Create Custom Agent

Once you have defined your custom agent class, create an instance of it in the MATLAB workspace. To create the custom REINFORCE agent, first specify the agent options.

```matlab
options.MaxStepsPerEpisode = 250;
options.DiscountFactor = 0.995;
```

Then, using the options and the previously defined actor representation, call the custom agent constructor function.

```matlab
agent = CustomReinforceAgent(actor,options);
```

Train Custom Agent

Configure the training to use the following options.

- Set up the training to last at most 5000 episodes, with each episode lasting at most 250 steps.
- Terminate the training after the maximum number of episodes is reached or when the average reward across 100 episodes reaches a value of 240.

For more information, see `rlTrainingOptions`.

```matlab
numEpisodes = 5000;
aveWindowSize = 100;
trainingTerminationValue = 240;
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',numEpisodes,...
    'MaxStepsPerEpisode',options.MaxStepsPerEpisode,...
    'ScoreAveragingWindowLength',aveWindowSize,...
    'StopTrainingValue',trainingTerminationValue);
```

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```matlab
doTraining = false;
if doTraining
    % Train the agent.
```

```
    trainStats = train(agent,env,trainOpts);
else
    % Load pretrained agent for the example.
    load('CustomReinforce.mat','agent');
end
```
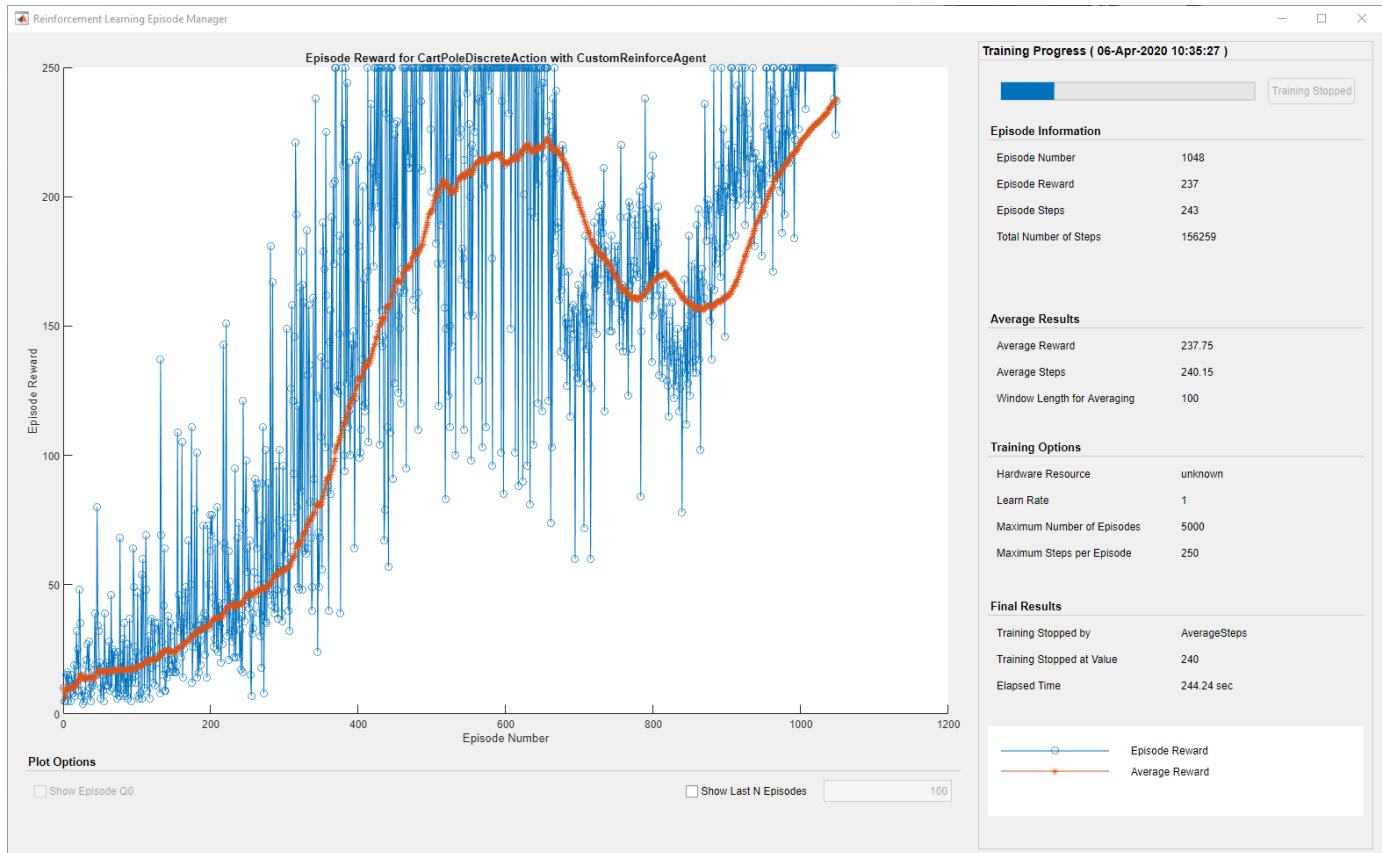

Simulate Custom Agent

Enable the environment visualization, which is updated each time the environment `step` function is called.

```
plot(env)
```

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOpts = rlSimulationOptions('MaxSteps',options.MaxStepsPerEpisode);
experience = sim(env,agent,simOpts);
```
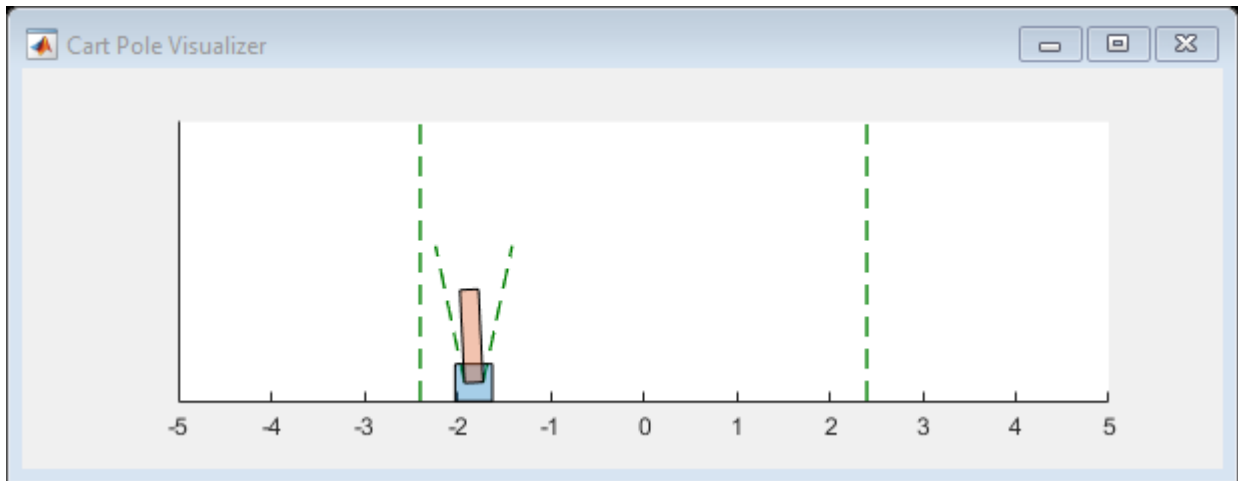
See Also

`train`

More About

- "Custom Agents" on page 3-37
- "Train Reinforcement Learning Agents" on page 5-2

Deploy Trained Policies

Deploy Trained Reinforcement Learning Policies

Once you train a reinforcement learning agent, you can generate code to deploy the optimal policy. You can generate:

- CUDA® code for deep neural network policies using GPU Coder
- C/C++ code for table, deep neural network, or linear basis function policies using MATLAB Coder

Note Code generation for deep neural network policies supports only networks with a single input layer.

For more information on training reinforcement learning agents, see "Train Reinforcement Learning Agents" on page 5-2.

To create a policy evaluation function that selects an action based on a given observation, use the `generatePolicyFunction` command. This command generates a MATLAB script, which contains the policy evaluation function, and a MAT-file, which contains the optimal policy data.

You can generate code to deploy this policy function using GPU Coder or MATLAB Coder.

Generate Code Using GPU Coder

If your trained optimal policy uses a deep neural network, you can generate CUDA code for the policy using GPU Coder. There are several required and recommended prerequisite products for generating CUDA code for deep neural networks. For more information, see "Installing Prerequisite Products" (GPU Coder) and "Setting Up the Prerequisite Products" (GPU Coder).

Not all deep neural network layers support GPU code generation. For a list of supported layers, see "Supported Networks and Layers" (GPU Coder). For more information and examples on GPU code generation, see "Deep Learning with GPU Coder" (GPU Coder).

Generate CUDA Code for Deep Neural Network Policy

As an example, generate GPU code for the policy gradient agent trained in "Train PG Agent to Balance Cart-Pole System" on page 5-14.

Load the trained agent.

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

Create a policy evaluation function for this agent.

```
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. 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 GPU Coder. For example, you can generate a CUDA compatible MEX function.

Configure the `codegen` function to create a CUDA compatible C++ MEX function.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
```

Set the dimensions of the policy evaluation input argument, which correspond to the observation specification dimensions for the agent. To find the observation dimensions, use the `getObservationInfo` function. In this case, the observations are in a four-element vector.

```
argstr = '{ones(4,1)}';
```

Generate code using the `codegen` function.

```
codegen('-config','cfg','evaluatePolicy','-args',argstr,'-report');
```

This command generates the MEX function `evaluatePolicy_mex`.

Generate Code Using MATLAB Coder

You can generate C/C++ code for table, deep neural network, or linear basis function policies using MATLAB Coder.

Using MATLAB Coder, you can generate:

- C/C++ code for policies that use Q tables, value tables, or linear basis functions. For more information on general C/C++ code generation, see "Generating Code" (MATLAB Coder).
- C++ code for policies that use deep neural networks. For more information, see "Prerequisites for Deep Learning with MATLAB Coder" (MATLAB Coder) and "Deep Learning with MATLAB Coder" (MATLAB Coder).

Generate C++ Code for Deep Neural Network Policy

As an example, generate C code for the policy gradient agent trained in "Train PG Agent to Balance Cart-Pole System" on page 5-14.

Load the trained agent.

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

Create a policy evaluation function for this agent.

```
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. 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.

Configure the `codegen` function to generate code suitable for targeting a static library.

```
cfg = coder.config('lib');
```

On the configuration object, set the target language to C++, and set `DeepLearningConfig` to the target library `'mkldnn'`. This option generates code using the Intel Math Kernel Library for Deep Neural Networks (Intel MKL-DNN).

```
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

Set the dimensions of the policy evaluation input argument, which correspond to the observation specification dimensions for the agent. To find the observation dimensions, use the `getObservationInfo` function. In this case, the observations are in a four-element vector.

```
argstr = '{ones(4,1)}';
```

Generate code using the `codegen` function.

```
codegen('-config','cfg','evaluatePolicy','-args',argstr,'-report');
```

This command generates the C++ code for the policy gradient agent containing the deep neural network actor.

Generate C Code for Q Table Policy

As an example, generate C code for the Q-learning agent trained in "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8.

Load the trained agent.

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

Create a policy evaluation function for this agent.

```
generatePolicyFunction(qAgent)
```

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

Set the dimensions of the policy evaluation input argument, which correspond to the observation specification dimensions for the agent. To find the observation dimensions, use the `getObservationInfo` function. In this case, there is a single finite observation.

```
argstr = '{[1]}';
```

Configure the `codegen` function to generate embeddable C code suitable for targeting a static library, and set the output folder to `buildFolder`.

```
cfg = coder.config('lib');
outFolder = 'buildFolder';
```

Generate C code using the `codegen` function.

```
codegen('-c','-d',outFolder,'-config','cfg',...
    'evaluatePolicy','-args',argstr,'-report');
```

See Also
generatePolicyFunction

More About

- "Reinforcement Learning Agents" on page 3-2
- "Train Reinforcement Learning Agents" on page 5-2