

# Reinforcement Learning Toolbox™

Reference



# MATLAB®

R2019a

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

## *Reinforcement Learning Toolbox™ Reference*

© COPYRIGHT 2019 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### **Revision History**

March 2019      Online only      New for Version 1.0 (Release 2019a)

**1** | Functions – Alphabetical List

**2** | Objects – Alphabetical List

**3** | Blocks – Alphabetical List



# Functions — Alphabetical List

---

## bus2RLSpec

Create reinforcement learning data specifications for elements of a Simulink bus

### Syntax

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

### Description

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

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

### Input Arguments

#### **busName** — Name of Simulink bus object

string | character vector

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

#### **Name-Value Pair Arguments**

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

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

**Model — Name of Simulink model**

string | character vector

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

**BusElementNames — Names of bus leaf elements**

string array

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

---

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

---

**DiscreteElements — Finite values for discrete bus elements**

cell array of name-value pairs

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

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

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

## Output Arguments

**specs — Data specifications**

rlnumericSpec object | rlFiniteSetSpec object | array of data specification objects

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

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

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

## See Also

### Blocks

RL Agent

### Functions

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

## Topics

“Create Simulink Environments for Reinforcement Learning”

**Introduced in R2019a**



# createGridWorld

Create a two-dimensional grid world for reinforcement learning

## Syntax

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

## Description

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

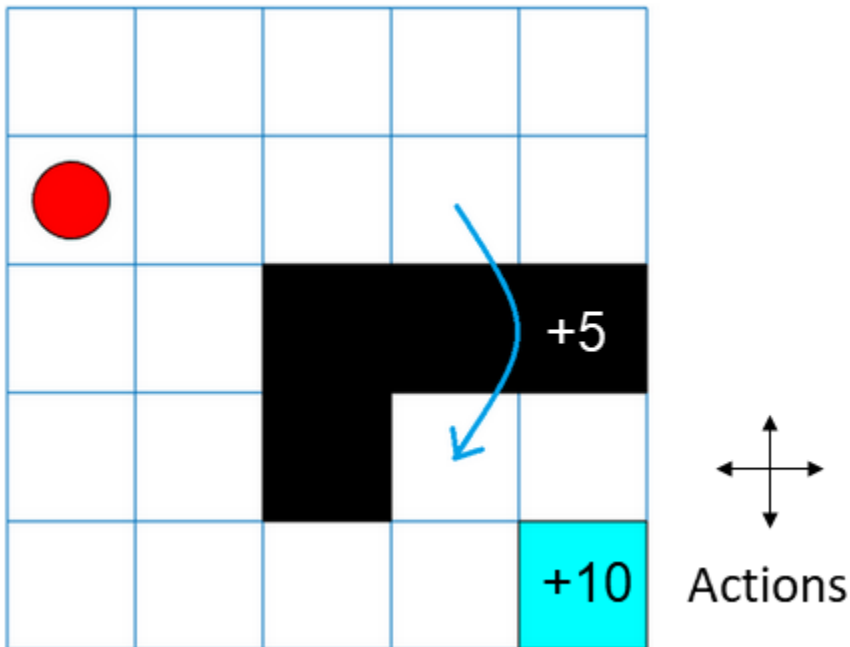
`GW = createGridWorld(m,n,moves)` creates a grid world `GW` of size `m`-by-`n` with actions specified by `moves`.

## Examples

### Create Grid World Environment

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

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



First, create a GridWorld object using the createGridWorld function.

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

```
GW =
```

```
GridWorld with properties:
```

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

Now, set the initial, terminal and obstacle states.

```

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

```

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

```

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

```

Next, define the rewards in the reward transition matrix.

```

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

```

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

```

env = rLMDPEnv(GW)

env =
  rLMDPEnv with properties:
    Model: [1x1 rl.env.GridWorld]
  ResetFcn: []

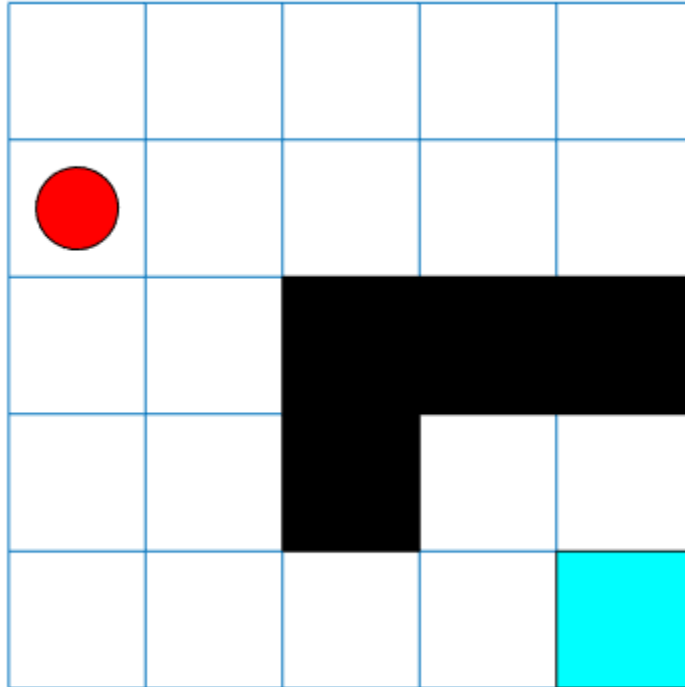
```

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

```

plot(env)

```



## Input Arguments

**m** — Number of rows of the grid world

scalar

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

**n** — Number of columns of the grid world

scalar

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

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

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

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

## Output Arguments

**GW — Two-dimensional grid world**`GridWorld` object

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

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

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

**CurrentState — Name of the current state**`string`

Name of the current state, specified as a string.

**States — State names**`string` vector

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

**Actions — Action names**`string` vector

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

`Actions` is a string vector of length:

- Four, if `moves` is specified as 'Standard'.

- Eight, moves is specified as 'Kings'.

## **T — State transition matrix**

3D array

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

$$T(s, s', a) = \text{probability}(s' | s, a).$$

T is:

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

## **R — Reward transition matrix**

3D array

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

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

R is:

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

## **ObstacleStates — State names that cannot be reached in the grid world**

string vector

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

## **TerminalStates — Terminal state names in the grid world**

string vector

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

## See Also

`rLMDPEnv` | `rLPredefinedEnv`

## Topics

“Create Custom Grid World Environments”

“Train Reinforcement Learning Agent in Basic Grid World”

**Introduced in R2019a**

## createIntegratedEnv

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

### Syntax

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

### Description

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

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

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

### Input Arguments

#### **refModel** — Reference model name

string | character vector

Reference model name, specified as a string or character vector. The new Simulink model uses this reference model as the dynamic model of the environment for reinforcement learning.



**newModel — New model name**

string | character vector

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

**Name-Value Pair Arguments**

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

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

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

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

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

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

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

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

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

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

## **ObservationBusElementNames — Names of observation bus leaf elements**

string array

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

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

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

## **ObservationDiscreteElements — Finite values for observation specifications**

cell array of name-value pairs

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

If the observation output port of the reference model is:

- A bus signal, specify the name of one of the leaf elements of the bus specified in by ObservationBusElementNames
- Nonbus signal, specify the name of the observation port, as specified by ObservationPortName

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

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

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

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

### **ActionDiscreteElements — Finite values for action specifications**

cell array of name-value pairs

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

If the action input port of the reference model is:

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

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

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

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

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

## **Output Arguments**

### **env — Reinforcement learning environment**

SimulinkEnvWithAgent object

Reinforcement learning environment interface, returned as an SimulinkEnvWithAgent object.

### **agentBlock — Block path to the agent block**

character vector

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

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

### **obsInfo — Observation data specifications**

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

Observation data specifications, returned as one of the following:

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

### **actInfo — Action data specifications**

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

Action data specifications, returned as one of the following:

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

## See Also

### **Blocks**

RL Agent

### **Functions**

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

### **Topics**

“Create Simulink Environments for Reinforcement Learning”

### **Introduced in R2019a**

# createMDP

Create Markov decision process model

## Syntax

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

## Description

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

## Examples

### Create MDP model

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

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

Specify the state transitions and their associated rewards.

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

```
MDP.T(3,6,2) = 1;
MDP.R(3,6,2) = 4;
% State 4 Transition and Reward
MDP.T(4,7,1) = 1;
MDP.R(4,7,1) = 3;
MDP.T(4,8,2) = 1;
MDP.R(4,8,2) = 2;
% State 5 Transition and Reward
MDP.T(5,7,1) = 1;
MDP.R(5,7,1) = 1;
MDP.T(5,8,2) = 1;
MDP.R(5,8,2) = 9;
% State 6 Transition and Reward
MDP.T(6,7,1) = 1;
MDP.R(6,7,1) = 5;
MDP.T(6,8,2) = 1;
MDP.R(6,8,2) = 1;
% State 7 Transition and Reward
MDP.T(7,7,1) = 1;
MDP.R(7,7,1) = 0;
MDP.T(7,7,2) = 1;
MDP.R(7,7,2) = 0;
% State 8 Transition and Reward
MDP.T(8,8,1) = 1;
MDP.R(8,8,1) = 0;
MDP.T(8,8,2) = 1;
MDP.R(8,8,2) = 0;
```

Specify the terminal states of the model.

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

## Input Arguments

### states — Model states

positive integer | string vector

Model states, specified as one of the following:

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

**actions — Model actions**

positive integer | string vector

Model actions, specified as one of the following:

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

## Output Arguments

**MDP — MDP model**

GenericMDP object

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

**CurrentState — Name of the current state**

string

Name of the current state, specified as a string.

**States — State names**

string vector

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

**Actions — Action names**

string vector

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

**T — State transition matrix**

3D array

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

$$T(s, s', a) = \text{probability}(s' | s, a).$$

T is an  $S$ -by- $S$ -by- $A$  array, where  $S$  is the number of states and  $A$  is the number of actions.

## **R — Reward transition matrix**

3D array

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

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

## **TerminalStates — Terminal state names in the grid world**

string vector

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

## **See Also**

`createGridWorld` | `rLMDPEnv`

## **Topics**

“Train Reinforcement Learning Agent in MDP Environment”

**Introduced in R2019a**



# generatePolicyFunction

**Package:** rl.agent

Create function that evaluates trained policy of reinforcement learning agent

## Syntax

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

## Description

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

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

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

## Examples

### Create Policy Evaluation Function for PG Agent

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

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

Create a policy evaluation function for this agent using default names.

```
generatePolicyFunction(agent)
```

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

To view the generated function, type:

```
type evaluatePolicy.m
```

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

Since the actor network for this PG agent has a single input layer and single output layer, you can generate code for this network using the Deep Learning Toolbox™ code generation functionality. For more information, see “Deploy Trained Reinforcement Learning Policies”.

### Create Policy Evaluation Function for Q-Learning Agent

Create and train a reinforcement learning agent. For this example, load the Q-Learning agent trained in “Train Reinforcement Learning Agent in Basic Grid World”.

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

Create a policy evaluation function for this agent, specifying the name of the agent data file.

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

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

To view the generated function, type:

```
type evaluatePolicy.m
```

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

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

### Create Policy Evaluation Function for DQN Network

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

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

Create a policy evaluation function for this agent, specifying the function and file name.

```
generatePolicyFunction(agent,'FunctionName','computeAction')
```

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

To view the generated function, type:

```
type computeAction.m
```

For a given observation, the policy function evaluates the observation-action value function for each potential discrete action, using the critic network. Then, the policy function selects the action that produces the largest predicted value function.

The Deep Learning Toolbox code generation functionality supports only networks with a single input layer. Therefore, code generation is not supported for `computeAction.m`, since the critic in a DQN agent has two input layers, one for the observation and one for the action.

## Input Arguments

### agent — Trained reinforcement learning agent

rlQAgent object | rLSARSAgent object | rLDQNAgent object | rLDDPGAgent object | rLPGAgent object | rLACAgent object

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

- rLQAgent object

- `rLSARSAgent` object
- `rLDQNAgent` object
- `rLDDPGAgent` object
- `rLACAgent` object
- `rLPGAgent` object that estimates a baseline value function using a critic

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

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

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

## Name-Value Pair Arguments

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

Example: `'FunctionName', 'computeAction'`

### **FunctionName** — Name of the generated function

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

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

### **PolicyName** — Name of the policy variable within the generated function

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

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

### **MATFileName** — Name of agent data file

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

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

## See Also

rlRepresentation | train

## Topics

“Train Reinforcement Learning Agents”

“Reinforcement Learning Agents”

“Create Policy and Value Function Representations”

“Deploy Trained Reinforcement Learning Policies”

**Introduced in R2019a**

## getActionInfo

Obtain action data specifications from reinforcement learning environment or agent

### Syntax

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

### Description

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

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

### Examples

#### Extract Action and Observation Information from Reinforcement Learning Environment

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

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

Open the model and create the reinforcement learning environment.

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

```

agentblk = [mdl '/RL Agent'];
% create the observation info
obsInfo = rlNumericSpec([3 1], 'LowerLimit', -inf*ones(3,1), 'UpperLimit', inf*ones(3,1));
obsInfo.Name = 'observations';
obsInfo.Description = 'information on velocity error and ego velocity';
% action Info
actInfo = rlNumericSpec([1 1], 'LowerLimit', -3, 'UpperLimit', 2);
actInfo.Name = 'acceleration';
% define environment
env = rlSimulinkEnv(mdl, agentblk, obsInfo, actInfo)

env =
  SimulinkEnvWithAgent with properties:
      Model: "rLACCMdl"
  AgentBlock: "rLACCMdl/RL Agent"
      ResetFcn: []
  UseFastRestart: 'on'

```

The reinforcement learning environment `env` is a `SimulinkWithAgent` object with the above properties.

Extract the action and observation information from the reinforcement learning environment `env`.

```

actInfoExt = getActionInfo(env)

actInfoExt =
  rlNumericSpec with properties:
      LowerLimit: -3
      UpperLimit: 2
      Name: "acceleration"
  Description: [0x0 string]
  Dimension: [1 1]
  DataType: "double"

obsInfoExt = getObservationInfo(env)

obsInfoExt =
  rlNumericSpec with properties:
      LowerLimit: [3x1 double]

```

```
UpperLimit: [3x1 double]
  Name: "observations"
Description: "information on velocity error and ego velocity"
Dimension: [3 1]
DataType: "double"
```

The action information contains acceleration values while the observation information contains the velocity and velocity error values of the ego vehicle.

## Input Arguments

### **env — Reinforcement learning environment**

SimulinkEnvWithAgent object

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

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

### **agent — Reinforcement learning agent**

rlQAgent object | rlSARSAAgent object | rlDQNAgent object | rlDDPGAgent object | rlPGAgent object | rlACAgent object

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

- rlQAgent
- rlSARSAAgent
- rlDQNAgent
- rlDDPGAgent
- rlPGAgent
- rlACAgent

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



## Output Arguments

### **actInfo** — Action data specifications

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

Action data specifications extracted from the reinforcement learning environment, returned as an array of one of the following:

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

## See Also

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

## Topics

“Create Simulink Environments for Reinforcement Learning”

“Reinforcement Learning Agents”

**Introduced in R2019a**

## getActor

**Package:** rl.agent

Get actor representation from reinforcement learning agent

## Syntax

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

## Description

`actor = getActor(agent)` returns the actor representation object for the specified reinforcement learning agent.

## Examples

### Modify Actor Parameter Values

Assume that you have an existing trained reinforcement learning agent, `agent`.

Obtain the actor representation from the agent.

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

Obtain the learnable parameters from the critic.

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

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,'UniformOutput',false);
```

Set the parameter values of the critic to the new modified values.

```
actor = setLearnableParameterValues(actor,modifiedParams);
```

Set the critic in the agent to the new modified critic.

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

### Add Layer to Actor Representation

Assume that you have an existing reinforcement learning agent, `agent`.

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

```
originalActor = [
    imageInputLayer([4 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(2, 'Name', 'action')];
```

Create an actor representation with an additional fully connected layer.

```
actorNetwork = [
    imageInputLayer([4 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(3, 'Name', 'x');
    fullyConnectedLayer(2, 'Name', 'action')];
actor = rlRepresentation(actorNetwork, ...
    'Observation', {'state'}, getObservationInfo(env),
    'Action', {'action'}, getActionInfo(env));
```

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

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

## Input Arguments

### **agent** — Reinforcement learning agent

`rlDDPGAgent` object | `rlPGAgent` object | `rlACAgent` object

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

- `rlDDPGAgent` object
- `rlACAgent` object

- `rlPGAgent` object

## Output Arguments

### **actor** — Actor representation

`rlLayerRepresentation` object | `rlTableRepresentation` object

Actor representation object, returned as one of the following:

- `rlLayerRepresentation` object for deep neural network representations
- `rlTableRepresentation` object for value table or Q table representations

## See Also

`getCritic` | `getLearnableParameterValues` | `rlRepresentation` | `setActor` | `setCritic` | `setLearnableParameterValues`

## Topics

“Create Policy and Value Function Representations”

“Import Policy and Value Function Representations”

**Introduced in R2019a**

# getCritic

**Package:** rl.agent

Get critic representation from reinforcement learning agent

## Syntax

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

## Description

`critic = getCritic(agent)` returns the critic representation object for the specified reinforcement learning agent.

## Examples

### Modify Critic Parameter Values

Assume that you have an existing trained reinforcement learning agent, `agent`.

Obtain the critic representation from the agent.

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

Obtain the learnable parameters from the critic.

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

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,'UniformOutput',false);
```

Set the parameter values of the critic to the new modified values.

```
critic = setLearnableParameterValues(critic,modifiedParams);
```

Set the critic in the agent to the new modified critic.

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

### Add Layer to Critic Representation

Assume that you have an existing reinforcement learning agent, `agent`.

Further, assume that this agent has a critic representation that contains the following deep neural network structure.

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

Create an actor representation with an additional fully connected layer.

```
criticNetwork = [  
    imageInputLayer([4 1 1], 'Normalization', 'none', 'Name', 'state')  
    fullyConnectedLayer(3, 'Name', 'x');  
    fullyConnectedLayer(1, 'Name', 'CriticFC')];  
critic = rlRepresentation(criticNetwork, 'Observation', {'state'}, ...  
    getObservationInfo(env));
```

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

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

## Input Arguments

### **agent** — Reinforcement learning agent

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

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

- `rlQAgent` object
- `rlSARSAAgent` object

- `rLDQNAgent` object
- `rLDDPGAgent` object
- `rLACAgent` object
- `rLPGAgent` object that estimates a baseline value function using a critic

## Output Arguments

### **critic** – Critic representation

`rLLayerRepresentation` object | `rLTableRepresentation` object

Critic representation object, returned as one of the following:

- `rLLayerRepresentation` object for deep neural network representations
- `rLTableRepresentation` object for value table or Q table representations

## See Also

`getActor` | `getLearnableParameterValues` | `rLRepresentation` | `setActor` | `setCritic` | `setLearnableParameterValues`

## Topics

“Create Policy and Value Function Representations”

“Import Policy and Value Function Representations”

**Introduced in R2019a**

## getLearnableParameterValues

**Package:** rl.util

Obtain learnable parameter values from policy or value function representation

### Syntax

```
val = getLearnableParameterValues(rep)
```

### Description

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

### Examples

#### Modify Critic Parameter Values

Assume that you have an existing trained reinforcement learning agent, `agent`.

Obtain the critic representation from the agent.

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

Obtain the learnable parameters from the critic.

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

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,'UniformOutput',false);
```

Set the parameter values of the critic to the new modified values.

```
critic = setLearnableParameterValues(critic,modifiedParams);
```



Set the critic in the agent to the new modified critic.

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

### Modify Actor Parameter Values

Assume that you have an existing trained reinforcement learning agent, `agent`.

Obtain the actor representation from the agent.

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

Obtain the learnable parameters from the critic.

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

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2, params, 'UniformOutput', false);
```

Set the parameter values of the critic to the new modified values.

```
actor = setLearnableParameterValues(actor, modifiedParams);
```

Set the critic in the agent to the new modified critic.

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

## Input Arguments

### rep — Policy or value function representation

`rLLayerRepresentation` object | `rLTableRepresentation` object

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

- `rLLayerRepresentation` object for deep neural network representations
- `rLTableRepresentation` object for value table or Q table representations

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

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

## Output Arguments

### **val** — Learnable parameter values

cell array

Learnable parameter values for the representation object, returned as a cell array. You can modify these parameter values and set them in the original agent or a different agent using the `setLearnableParameterValues` function.

## See Also

`getActor` | `getCritic` | `rlRepresentation` | `setActor` | `setCritic` | `setLearnableParameterValues`

## Topics

“Create Policy and Value Function Representations”

“Import Policy and Value Function Representations”

**Introduced in R2019a**

## getObservationInfo

Obtain observation data specifications from reinforcement learning environment or agent

### Syntax

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

### Description

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

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

### Examples

#### Extract Action and Observation Information from Reinforcement Learning Environment

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

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

Open the model and create the reinforcement learning environment.

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

```
agentblk = [mdl '/RL Agent'];
% create the observation info
obsInfo = rlNumericSpec([3 1], 'LowerLimit', -inf*ones(3,1), 'UpperLimit', inf*ones(3,1));
obsInfo.Name = 'observations';
obsInfo.Description = 'information on velocity error and ego velocity';
% action Info
actInfo = rlNumericSpec([1 1], 'LowerLimit', -3, 'UpperLimit', 2);
actInfo.Name = 'acceleration';
% define environment
env = rlSimulinkEnv(mdl, agentblk, obsInfo, actInfo)

env =
  SimulinkEnvWithAgent with properties:
      Model: "rLACCMdl"
  AgentBlock: "rLACCMdl/RL Agent"
      ResetFcn: []
  UseFastRestart: 'on'
```

The reinforcement learning environment `env` is a `SimulinkWithAgent` object with the above properties.

Extract the action and observation information from the reinforcement learning environment `env`.

```
actInfoExt = getActionInfo(env)
```

```
actInfoExt =
  rlNumericSpec with properties:
      LowerLimit: -3
      UpperLimit: 2
      Name: "acceleration"
  Description: [0x0 string]
      Dimension: [1 1]
      DataType: "double"
```

```
obsInfoExt = getObservationInfo(env)
```

```
obsInfoExt =
  rlNumericSpec with properties:
      LowerLimit: [3x1 double]
```

```
UpperLimit: [3x1 double]
  Name: "observations"
Description: "information on velocity error and ego velocity"
Dimension: [3 1]
DataType: "double"
```

The action information contains acceleration values while the observation information contains the velocity and velocity error values of the ego vehicle.

## Input Arguments

### **env — Reinforcement learning environment**

`SimulinkEnvWithAgent` object

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

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

### **agent — Reinforcement learning agent**

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

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

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

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

## Output Arguments

### **obsInfo — Observation data specifications**

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

Observation data specifications extracted from the reinforcement learning environment, returned as an array of one of the following:

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

## See Also

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

## Topics

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

**Introduced in R2019a**

# rlACAgent

Create actor-critic reinforcement learning agent

## Syntax

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

## Description

`agent = rlACAgent(actor, critic, opt)` creates an actor-critic (AC) agent with the specified actor and critic networks, using the specified AC agent options. For more information on AC agents, see “Actor-Critic Agents”.

## Examples

### Create an Actor Critic Agent

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

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

Create a critic representation.

```
criticNetwork = [
    imageInputLayer([4 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(1, 'Name', 'CriticFC')];
criticOpts = rlRepresentationOptions('LearnRate', 8e-3, 'GradientThreshold', 1);
critic = rlRepresentation(criticNetwork, obsInfo, 'Observation', {'state'}, criticOpts);
```

Create an actor representation.

```
actorNetwork = [
    imageInputLayer([4 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(2, 'Name', 'action')];
actorOpts = rlRepresentationOptions('LearnRate', 8e-3, 'GradientThreshold', 1);
```

```
actor = rlRepresentation(actorNetwork,obsInfo,actInfo,...  
    'Observation',{'state'},'Action',{'action'},actorOpts);
```

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

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

## Input Arguments

### **actor** — Actor network representation

`rlLayerRepresentation` object

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

### **critic** — Critic network representation

`rlLayerRepresentation` object

Critic network representation for estimating the state-value function, specified as an `rlLayerRepresentation` object created using `rlRepresentation`. For more information on creating critic representations, see “Create Policy and Value Function Representations”.

### **opt** — Agent options

`rlACAgentOptions` object

Agent options, specified as an `rlACAgentOptions` object.

## Output Arguments

### **agent** — AC agent

`rlACAgent` object

AC agent, returned as an `rlACAgent` object.



## See Also

r1ACAgentOptions | train

## Topics

“Actor-Critic Agents”

“Reinforcement Learning Agents”

“Train Reinforcement Learning Agents”

**Introduced in R2019a**

## rlACAgentOptions

Create options for AC agent

### Syntax

```
opt = rlACAgentOptions
opt = rlACAgentOptions(Name, Value)
```

### Description

`opt = rlACAgentOptions` creates an `rlACAgentOptions` object for use as an argument when creating an AC agent using all default settings. You can modify the object properties using dot notation.

`opt = rlACAgentOptions(Name, Value)` creates an AC options object using the specified name-value pairs to override default property values.

### Examples

#### Create AC Agent Options Object

Create an AC agent options object, specifying the discount factor.

```
opt = rlACAgentOptions('DiscountFactor', 0.95)
```

```
opt =
```

```
rlACAgentOptions with properties:
```

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

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

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

## Input Arguments

### Name-Value Pair Arguments

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

Example: "ExperienceBufferLength", 8192

#### **SampleTime** — Sample time of agent

1 (default) | numeric value

Sample time of agent, specified as the comma-separated pair consisting of 'SampleTime' and a numeric value.

#### **DiscountFactor** — Discount factor

numeric value

Discount factor applied to future rewards during training, specified as the comma-separated pair consisting of 'DiscountFactor' and a positive numeric value less than or equal to 1.

#### **NumStepsToLookAhead** — Number of steps ahead

numeric value

Number of steps to look ahead in model training, specified as the comma-separated pair consisting of 'NumStepsToLookAhead' and a numeric positive integer value. For AC agents, the number of steps to look ahead corresponds to the training episode length.

#### **EntropyLossWeight** — Entropy loss weight

0 (default) | scalar value between 0 and 1

Entropy loss weight, specified as the comma-separated pair consisting of 'EntropyLossWeight' and a scalar value between 0 and 1. A higher loss weight value promotes agent exploration by applying a penalty for being too certain about which action to take. Doing so can help the agent move out of local optima.

The entropy loss function for episode step  $t$  is:

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

Here:

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

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

## Output Arguments

### **opt — AC agent options**

`rLACAgentOptions` object

AC agent options, returned as an `rLACAgentOptions` object. The object properties are described in “Name-Value Pair Arguments” on page 1-47.

## See Also

### **Functions**

`rLACAgent`

### **Topics**

“Actor-Critic Agents”

**Introduced in R2019a**

# rlCreateEnvTemplate

Create custom reinforcement learning environment template

## Syntax

```
rlCreateEnvTemplate(className)
```

## Description

`rlCreateEnvTemplate(className)` creates and opens a MATLAB script that contains a template class representing a reinforcement learning environment. The template class contains an implementation of a simple cart-pole balancing environment. To define your custom environment, modify this template class. For more information, see “Create Custom MATLAB Environment from Template”.

## Examples

### Create Custom Environment Template File

Create and open a template file for a reinforcement learning environment. Name the class `myEnvironment`.

```
rlCreateEnvTemplate("myEnvironment")
```

This function opens a MATLAB script that contains the class. Modify this template class, and save the file as `myEnvironment.m`.

## Input Arguments

**className** — Name of environment class  
string | character vector

Name of environment class, specified as a string or character vector. This name defines the name of the class and the name of the MATLAB script.

## **See Also**

### **Topics**

“Create MATLAB Environments for Reinforcement Learning”

**Introduced in R2019a**

# rDDPGAgent

Create deep deterministic policy gradient reinforcement learning agent

## Syntax

```
agent = rDDPGAgent(actor, critic, opt)
```

## Description

`agent = rDDPGAgent(actor, critic, opt)` creates a DDPG agent with the specified actor and critic networks, using the specified DDPG agent options. For more information on DDPG agents, see “Deep Deterministic Policy Gradient Agents”.

## Examples

### Create a DDPG Agent

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

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

Create a critic representation.

```
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');
```

```
criticOpts = rlRepresentationOptions('LearnRate',5e-3,'GradientThreshold',1);
critic = rlRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation',{'state'},'Action',{'action'},criticOpts);
```

Create an actor representation.

```
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 = rlRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'state'},'Action',{'action'},actorOpts);
```

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

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

## Input Arguments

### **actor** — Actor network representation

`rlLayerRepresentation` object

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

### **critic** — Critic network representation

`rlLayerRepresentation` object

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

### **opt** — Agent options

`rlDDPGAgentOptions` object

Agent options, specified as an `rlDDPGAgentOptions` object.



## Output Arguments

### **agent** — DDPG agent

rLDDPGAgent object

DDPG agent, returned as an rLDDPGAgent object.

## See Also

rLDDPGAgentOptions | train

## Topics

“Deep Deterministic Policy Gradient Agents”

“Reinforcement Learning Agents”

“Train Reinforcement Learning Agents”

**Introduced in R2019a**

## rLDDPGAgentOptions

Create options for DDPG agent

### Syntax

```
opt = rLDDPGAgentOptions
opt = rLDDPGAgentOptions(Name, Value)
```

### Description

`opt = rLDDPGAgentOptions` creates an `rLDDPGAgentOptions` object for use as an argument when creating a DDPG agent using all default options. You can modify the object properties using dot notation.

`opt = rLDDPGAgentOptions(Name, Value)` creates a DDPG options object using the specified name-value pairs to override default property values.

### Examples

#### Create DDPG Agent Options Object

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

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

```
opt =
```

```
rLDDPGAgentOptions with properties:
```

```
    NoiseOptions: [1x1 rl.option.OrnsteinUhlenbeckActionNoise]
    TargetSmoothFactor: 1.0000e-03
    TargetUpdateFrequency: 4
    TargetUpdateMethod: "smoothing"
    ResetExperienceBufferBeforeTraining: 1
    SaveExperienceBufferWithAgent: 0
    MiniBatchSize: 48
    NumStepsToLookAhead: 1
    ExperienceBufferLength: 10000
```

```
SampleTime: 1
DiscountFactor: 0.9900
```

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

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

## Input Arguments

### Name-Value Pair Arguments

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

Example: "MiniBatchSize", 24

### NoiseOptions — Options for noise

OrnsteinUhlenbeckActionNoise object

Options for noise, specified as the comma-separated pair consisting of 'NoiseOptions' and an OrnsteinUhlenbeckActionNoise object, with the following numeric value properties.

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

At each sample time step, the noise model is updated using the following formula:

$$x(k) = x(k-1) + \text{MeanAttractionConstant} \cdot (\text{Mean} - x(k-1)) \cdot \text{SampleTime} + \text{Variance} \cdot \text{randn}(\text{size}(\text{Mean})) \cdot \text{sqrt}(\text{SampleTime})$$

To specify noise options, use dot notation after creating the `rLDDPGAgentOptions` object. For example, set the noise mean to `0.5`.

```
opt = rLDDPGAgentOptions;  
opt.NoiseModel.Mean = 0.5;
```

For continuous action signals, it is important to set the noise variance correctly. For example, when the steering angle is bounded by `[-0.26,0.26]` radians, if you set the noise variance to `0.5`, then the agent will not learn anything.

If your agent converges on local optima too quickly, promote agent exploration by increasing the amount of noise.

### **TargetSmoothFactor — Smoothing factor for target updates**

`1e-3 (default) | double`

Smoothing factor for target actor and critic updates, specified as the comma-separated pair consisting of `'TargetSmoothFactor'` and a double. The smoothing factor determines how the target properties are updated when `TargetUpdateMethod` is `"smoothing"`.

### **TargetUpdateFrequency — Number of episodes between target updates**

`4 (default) | numeric value`

Number of episodes between target actor and critic updates, specified as the comma-separated pair consisting of `'TargetUpdateFrequency'` and a numeric integer value. This option applies only when `TargetUpdateMethod` is `"periodic"`.

### **TargetUpdateMethod — Strategy for updating target actor and critic properties**

`"smoothing" (default) | "periodic"`

Strategy for updating target actor and critic properties using values from the trained actor and critic, specified as the comma-separated pair consisting of `'TargetUpdateMethod'` and one of the following:

- `"smoothing"` — Update the target actor and critic properties, `thetaTarget`, at every training episode according to the following formula, where `theta` contains the current trained network properties.

```
thetaTarget = TargetSmoothFactor*theta + (1 - TargetSmoothFactor)*thetaTarget
```

- `"periodic"` — Update the target actor and critic properties every `TargetUpdateFrequency` training episodes.

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

Flag for clearing the experience buffer before training, specified as the comma-separated pair consisting of 'ResetExperienceBufferBeforeTraining' and a logical true or false.

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

Flag for saving the experience buffer data when saving the agent, specified as the comma-separated pair consisting of 'SaveExperienceBufferWithAgent' and a logical true or false. This option applies both when saving candidate agents during training and when saving agents using the save function.

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

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

**MiniBatchSize — Size of random experience mini-batch**`64 (default) | numeric value`

Size of random experience mini-batch, specified as the comma-separated pair consisting of 'MiniBatchSize' and a positive numeric value. During each training episode, the agent randomly samples experiences from the experience buffer when computing gradients for updating the critic properties. Large mini-batches reduce the variance when computing gradients but increase the computational effort.

**NumStepsToLookAhead — Number of steps ahead**`1 (default) | numeric value`

Number of steps to look ahead during training, specified as the comma-separated pair consisting of 'NumStepsToLookAhead' and a numeric positive integer value.

**ExperienceBufferLength — Experience buffer size**`10000 (default) | numeric value`

Experience buffer size, specified as the comma-separated pair consisting of 'ExperienceBufferLength' and a numeric positive integer value. During training, the agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.

### **SampleTime — Sample time of agent**

1 (default) | numeric value

Sample time of agent, specified as the comma-separated pair consisting of 'SampleTime' and a numeric value.

### **DiscountFactor — Discount factor**

0.99 (default) | numeric value

Discount factor applied to future rewards during training, specified as the comma-separated pair consisting of 'DiscountFactor' and a positive numeric value less than or equal to 1.

## Output Arguments

### **opt — DDPG agent options**

rLDDPGAgentOptions object

DDPG agent options, returned as an rLDDPGAgentOptions object. The object properties are described in “Name-Value Pair Arguments” on page 1-55.

## See Also

### **Functions**

rLDDPGAgent

### **Topics**

“Deep Deterministic Policy Gradient Agents”

**Introduced in R2019a**

# r1DQNAgent

Create deep Q-network reinforcement learning agent

## Syntax

```
agent = r1DQNAgent(critic,opt)
```

## Description

`agent = r1DQNAgent(critic,opt)` creates a DQN agent with the specified critic network and DQN agent options. For more information on DQN agents, see “Deep Q-Network Agents”.

## Examples

### Create a DQN Agent

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

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

Create a critic representation.

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

```
criticNetwork = addLayers(criticNetwork, commonPath);
criticNetwork = connectLayers(criticNetwork, 'CriticStateFC2', 'add/in1');
criticNetwork = connectLayers(criticNetwork, 'CriticActionFC1', 'add/in2');
criticOpts = rlRepresentationOptions('LearnRate', 0.01, 'GradientThreshold', 1);
critic = rlRepresentation(criticNetwork, obsInfo, actInfo, ...
    'Observation', {'state'}, 'Action', {'action'}, criticOpts);
```

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

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

## Input Arguments

### **critic** — Critic network representation

`rlLayerRepresentation` object

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

### **opt** — Agent options

`rlDQNAgentOptions` object

Agent options, specified as an `rlDQNAgentOptions` object.

## Output Arguments

### **agent** — DQN agent

`rlDQNAgent` object

DQN agent, returned as an `rlDQNAgent` object.

## See Also

`rlDQNAgentOptions` | `train`



## **Topics**

“Deep Q-Network Agents”

“Reinforcement Learning Agents”

“Train Reinforcement Learning Agents”

**Introduced in R2019a**

## rLDQNAgentOptions

Create options for DQN agent

### Syntax

```
opt = rLDQNAgentOptions
opt = rLDQNAgentOptions(Name,Value)
```

### Description

`opt = rLDQNAgentOptions` creates an `rLDQNAgentOptions` object for use as an argument when creating a DQN agent using all default settings. You can modify the object properties using dot notation.

`opt = rLDQNAgentOptions(Name,Value)` creates a DQN options object using the specified name-value pairs to override default property values.

### Examples

#### Create DQN Agent Options Object

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

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

`opt =`

`rLDQNAgentOptions` with properties:

```
UseDoubleDQN: 1
EpsilonGreedyExploration: [1x1 rl.option.EpsilonGreedyExploration]
TargetSmoothFactor: 1.0000e-03
TargetUpdateFrequency: 4
TargetUpdateMethod: "smoothing"
ResetExperienceBufferBeforeTraining: 1
SaveExperienceBufferWithAgent: 0
MiniBatchSize: 48
NumStepsToLookAhead: 1
```

```
ExperienceBufferLength: 10000
SampleTime: 1
DiscountFactor: 0.9900
```

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

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

## Input Arguments

### Name-Value Pair Arguments

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

Example: `"UseDoubleDQN", false`

#### **UseDoubleDQN** — Flag for using double DQN

true (default) | false

Flag for using double DQN for value function target updates, specified as the comma-separated pair consisting of 'UseDoubleDQN' and a logical true or false. For most application set UseDoubleDQN to "on". For more information, see "Deep Q-Network Agents".

#### **EpsilonGreedyExploration** — Options for epsilon-greedy exploration

EpsilonGreedyExploration object

Options for epsilon-greedy exploration, specified as the comma-separated pair consisting of 'EpsilonGreedyExploration' and an EpsilonGreedyExploration object with the following numeric value properties.

Property	Description
Epsilon	Probability threshold to either randomly select an action or select the action that maximizes the state-action value function. A larger value of Epsilon means that the agent randomly explores the action space at a higher rate.
EpsilonMin	Minimum value of Epsilon
EpsilonDecay	Decay rate

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

$$\text{Epsilon} = \text{Epsilon} * (1 - \text{EpsilonDecay})$$

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

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

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

### **TargetSmoothFactor — Smoothing factor for target updates**

1e-3 (default) | double

Smoothing factor for target critic updates, specified as the comma-separated pair consisting of 'TargetSmoothFactor' and a double. The smoothing factor determines how the target critic properties are updated when TargetUpdateMethod is "smoothing".

### **TargetUpdateFrequency — Number of episodes between target updates**

4 (default) | numeric value

Number of episodes between target critic updates, specified as the comma-separated pair consisting of 'TargetUpdateFrequency' and a numeric integer value. This option applies only when TargetUpdateMethod is "periodic".

### **TargetUpdateMethod — Strategy for updating target critic properties**

"smoothing" (default) | "periodic"

Strategy for updating target critic properties using values from the trained actor and critic, specified as the comma-separated pair consisting of 'TargetUpdateMethod' and one of the following:

- "smoothing" — Update the target critic properties, `thetaTarget`, at every training episode according to the following formula, where `theta` is the current trained network properties:

$$\text{thetaTarget} = \text{TargetSmoothFactor} * \text{theta} + (1 - \text{TargetSmoothFactor}) * \text{thetaTarget}$$

- "periodic" — Update the target critic properties every `TargetUpdateFrequency` training episodes.

### **ResetExperienceBufferBeforeTraining** — Flag for clearing the experience buffer

`true` (default) | `false`

Flag for clearing the experience buffer before training, specified as the comma-separated pair consisting of 'ResetExperienceBufferBeforeTraining' and a logical `true` or `false`.

### **SaveExperienceBufferWithAgent** — Flag for saving the experience buffer

`false` (default) | `true`

Flag for saving the experience buffer data when saving the agent, specified as the comma-separated pair consisting of 'SaveExperienceBufferWithAgent' and a logical `true` or `false`. This option applies both when saving candidate agents during training and when saving agents using the `save` function.

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

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

### **MiniBatchSize** — Size of random experience mini-batch

64 (default) | numeric value

Size of random experience mini-batch, specified as the comma-separated pair consisting of 'MiniBatchSize' and a positive numeric value. During each training episode, the

agent randomly samples experiences from the experience buffer when computing gradients for updating the actor and critic properties.

### **NumStepsToLookAhead — Number of steps ahead**

1 (default) | numeric value

Number of steps to look ahead during training, specified as the comma-separated pair consisting of 'NumStepsToLookAhead' and a numeric positive integer value.

### **ExperienceBufferLength — Experience buffer size**

10000 (default) | numeric value

Experience buffer size, specified as the comma-separated pair consisting of 'ExperienceBufferLength' and a numeric positive integer value. During training, the agent updates the critic using a mini-batch of experiences randomly sampled from the buffer.

In general, agents need to learn from both good and bad experiences. Specify an experience buffer size that is able to store enough experience for learning.

### **SampleTime — Sample time of agent**

1 (default) | numeric value

Sample time of agent, specified as the comma-separated pair consisting of 'SampleTime' and a numeric value.

### **DiscountFactor — Discount factor applied to rewards**

0.99 (default) | numeric value

Discount factor applied to future rewards during training, specified as the comma-separated pair consisting of 'DiscountFactor' and a positive numeric value less than or equal to 1.

## **Output Arguments**

### **opt — DQN agent options**

rLDQNAgentOptions object

DQN agent options, returned as an rLDQNAgentOptions object. The object properties are described in “Name-Value Pair Arguments” on page 1-63.

## See Also

### Functions

r1DQNAgent

### Topics

“Deep Q-Network Agents”

**Introduced in R2019a**

## rlPGAgent

Create policy gradient reinforcement learning agent

### Syntax

```
agent = rlPGAgent(actor)
agent = rlPGAgent(actor, critic)
agent = rlPGAgent( ____, opt)
```

### Description

`agent = rlPGAgent(actor)` creates a PG agent with the specified actor network. By default, the `UseBaseline` property of the agent is `false` for this case. For more information on PG agents, see “Policy Gradient Agents”.

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

`agent = rlPGAgent( ____, opt)` creates a PG agent using the specified agent options to override the agent defaults.

### Examples

#### Create a PG Agent

Create an environment interface.

```
env = rlPredefinedEnv("DoubleIntegrator-Discrete");
obsInfo = getObservationInfo(env);
numObservations = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
numActions = numel(actInfo);
```

Create a critic representation to use as a baseline.



```

baselineNetwork = [
    imageInputLayer([numObservations 1 1], '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 = rlRepresentation(baselineNetwork,baselineOpts,'Observation',{ 'state'},obsInfo);

```

Create an actor representation.

```

actorNetwork = [
    imageInputLayer([numObservations 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(numActions, 'Name', 'action', 'BiasLearnRateFactor', 0)];
actorOpts = rlRepresentationOptions('LearnRate',5e-3,'GradientThreshold',1);
actor = rlRepresentation(actorNetwork,actorOpts,...
    'Observation',{ 'state'},obsInfo,'Action',{ 'action'},actInfo);

```

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

```

agentOpts = rlPGAOptions(...
    'UseBaseline',true, ...
    'DiscountFactor', 0.99);
agent = rlPGAAgent(actor,baseline,agentOpts);

```

## Input Arguments

### **actor** — Actor network representation

`rlLayerRepresentation` object

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

### **critic** — Critic network representation

`rlLayerRepresentation` object

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

### **opt** — Agent options

`rlPGAOptions` object

Agent options, specified as an `rlPGAOptions` object.

## Output Arguments

**agent** — PG agent

`r1PGAgent`

PG agent, returned as an `r1PGAgent` object.

## See Also

`r1PGAgentOptions` | `train`

## Topics

“Policy Gradient Agents”

“Reinforcement Learning Agents”

“Train Reinforcement Learning Agents”

**Introduced in R2019a**

# rIPGAgentOptions

Create options for PG agent

## Syntax

```
opt = rIPGAgentOptions
opt = rIPGAgentOptions(Name, Value)
```

## Description

`opt = rIPGAgentOptions` creates an `rIPGAgentOptions` object for use as an argument when creating a PG agent using all default settings. You can modify the object properties using dot notation.

`opt = rIPGAgentOptions(Name, Value)` creates a PG options object using the specified name-value pairs to override default property values.

## Examples

### Create PG Agent Options Object

Create a PG agent options object, specifying the discount factor.

```
opt = rIPGAgentOptions('DiscountFactor', 0.9)
```

```
opt =
```

```
  rIPGAgentOptions with properties:
```

```
    UseBaseline: 1
  EntropyLossWeight: 0
      SampleTime: 1
    DiscountFactor: 0.9000
```

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

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

## Input Arguments

### Name-Value Pair Arguments

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

Example: `"DiscountFactor", 0.95`

#### **UseBaseline** — Use baseline for learning

`true` (default) | `false`

Instruction to use baseline for learning, specified as the comma-separated pair consisting of `'UseBaseline'` and logical `true` or `false`. When `UseBaseline` is true, you must specify a critic network as the baseline function approximator.

In general, for simpler problems with smaller actor networks, PG agents work better without a baseline.

#### **SampleTime** — Sample time of agent

`1` (default) | numeric value

Sample time of agent, specified as the comma-separated pair consisting of `'SampleTime'` and a numeric value.

#### **DiscountFactor** — Discount factor

`0.99` | numeric value

Discount factor applied to future rewards during training, specified as the comma-separated pair consisting of `'DiscountFactor'` and a positive numeric value less than or equal to 1.

#### **EntropyLossWeight** — Entropy loss weight

`0` (default) | scalar value between 0 and 1

Entropy loss weight, specified as the comma-separated pair consisting of `'EntropyLossWeight'` and a scalar value between 0 and 1. A higher loss weight value

promotes agent exploration by applying a penalty for being too certain about which action to take. Doing so can help the agent move out of local optima.

The entropy loss function for episode step  $t$  is:

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

Here:

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

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

## Output Arguments

### **opt** — PG agent options

`rIPGAgentOptions`

PG agent options, returned as an `rIPGAgentOptions` object. The object properties are described in “Name-Value Pair Arguments” on page 1-72.

## See Also

### **Functions**

`rIPGAgent`

### **Topics**

“Policy Gradient Agents”

**Introduced in R2019a**

## rlPredefinedEnv

Create a predefined reinforcement learning environment

### Syntax

```
env = rlPredefinedEnv(keyword)
```

### Description

`env = rlPredefinedEnv(keyword)` takes a predefined keyword `keyword` representing the environment name to create a MATLAB or Simulink reinforcement learning environment `env`. The environment `env` models the dynamics with which the agent interacts, generating rewards and observations in response to agent actions.

### Examples

#### Basic Grid World Reinforcement Learning Environment

Use the predefined 'BasicGridWorld' keyword to create a basic grid world reinforcement learning environment.

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

```
env =  
    rLMDPEnv with properties:  
        Model: [1x1 rl.env.GridWorld]  
        ResetFcn: []
```

## Continuous Double Integrator Reinforcement Learning Environment

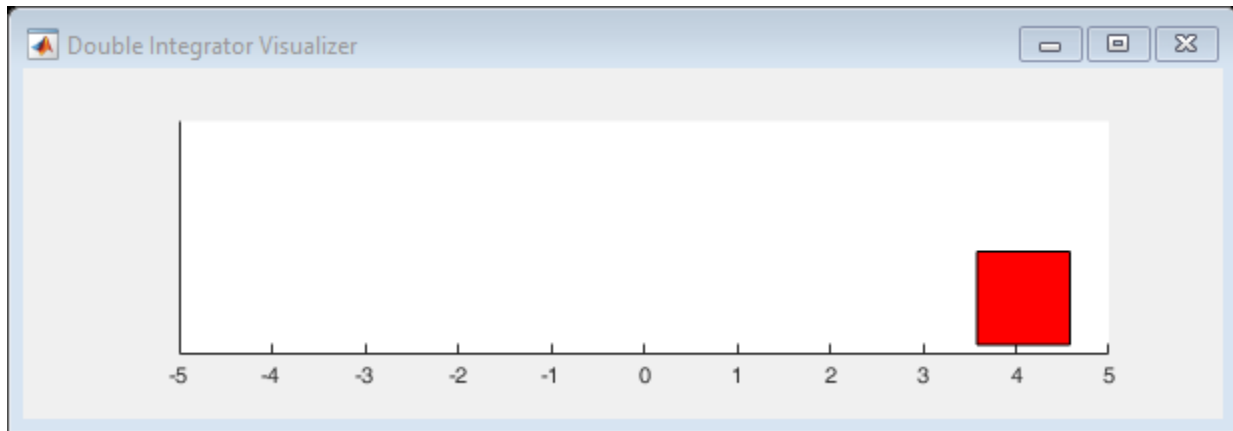
Use the predefined 'DoubleIntegrator-Continuous' keyword to create a continuous double integrator reinforcement learning environment.

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

```
env =  
  DoubleIntegratorContinuousAction with properties:  
  
      Gain: 1  
      Ts: 0.1000  
  MaxDistance: 5  
  GoalThreshold: 0.0100  
      Q: [2x2 double]  
      R: 0.0100  
  MaxForce: Inf  
  State: [2x1 double]
```

You can visualize the environment using the `plot` function and interact with it using the `reset` and `step` functions.

```
plot(env)  
observation = reset(env)  
  
observation = 2x1  
  
  4  
  0  
  
[observation, reward, isDone] = step(env,16)
```



```
observation = 2x1
```

```
    4.0800  
    1.6000
```

```
reward = -16.5559
```

```
isDone = logical  
        0
```

## Continuous Simple Pendulum Model Reinforcement Learning Environment

Use the predefined 'SimplePendulumModel-Continuous' keyword to create a continuous simple pendulum model reinforcement learning environment.

```
env = rlPredefinedEnv('SimplePendulumModel-Continuous')
```

```
env =
```

```
  SimulinkWithAgentBlock with properties:
```

```
    Model: "SimplePendulumModel"
```

```
  AgentBlock: "SimplePendulumModel/RL Agent"
```

```
    ResetFcn: []
```

```
  UseFastRestart: 'on'
```



## Input Arguments

**keyword** — Predefined keyword representing the environment name

```
'BasicGridWorld' | 'CartPole-Discrete' | 'CartPole-Continuous' |
'DoubleIntegrator-Discrete' | 'DoubleIntegrator-Continuous' |
'SimplePendulumWithImage-Discrete' | 'SimplePendulumWithImage-
Continuous' | 'WaterFallGridWorld-Deterministic' | 'WaterFallGridWorld-
Stochastic' | 'SimplePendulumModel-Discrete' | 'SimplePendulumModel-
Continuous' | 'CartPoleSimscapeModel-Discrete' |
'CartPoleSimscapeModel-Continuous'
```

Predefined keyword representing the environment name, specified as one of the following:

### MATLAB Environment

- 'BasicGridWorld'
- 'CartPole-Discrete'
- 'CartPole-Continuous'
- 'DoubleIntegrator-Discrete'
- 'DoubleIntegrator-Continuous'
- 'SimplePendulumWithImage-Discrete'
- 'SimplePendulumWithImage-Continuous'
- 'WaterFallGridWorld-Stochastic'
- 'WaterFallGridWorld-Deterministic'

### Simulink Environment

- 'SimplePendulumModel-Discrete'
- 'SimplePendulumModel-Continuous'
- 'CartPoleSimscapeModel-Discrete'
- 'CartPoleSimscapeModel-Continuous'

## Output Arguments

### **env — MATLAB or Simulink environment object**

`rLMDPEnv` object | `CartPoleDiscreteAction` object | `CartPoleContinuousAction` object | `DoubleIntegratorDiscreteAction` object | `DoubleIntegratorContinuousAction` object | `SimplePendulumWithImageDiscreteAction` object | `SimplePendulumWithImageContinuousAction` object | `SimulinkEnvWithAgent` object

MATLAB or Simulink environment object, returned as one of the following:

- `rLMDPEnv` object, when you use one of the following keywords:
  - `'BasicGridWorld'`
  - `'WaterFallGridWorld-Stochastic'`
  - `'WaterFallGridWorld-Deterministic'`
- `CartPoleDiscreteAction` object, when you use the `'CartPole-Discrete'` keyword.
- `CartPoleContinuousAction` object, when you use the `'CartPole-Continuous'` keyword.
- `DoubleIntegratorDiscreteAction` object, when you use the `'DoubleIntegrator-Discrete'` keyword.
- `DoubleIntegratorContinuousAction` object, when you use the `'DoubleIntegrator-Continuous'` keyword.
- `SimplePendulumWithImageDiscreteAction` object, when you use the `'SimplePendulumWithImage-Discrete'` keyword.
- `SimplePendulumWithImageContinuousAction` object, when you use the `'SimplePendulumWithImage-Continuous'` keyword.
- `SimulinkEnvWithAgent` object, when you use one of the following keywords:
  - `'SimplePendulumModel-Discrete'`
  - `'SimplePendulumModel-Continuous'`
  - `'CartPoleSimscapeModel-Discrete'`
  - `'CartPoleSimscapeModel-Continuous'`

## See Also

### Topics

“Create MATLAB Environments for Reinforcement Learning”

“Create Simulink Environments for Reinforcement Learning”

“Load Predefined Control System Environments”

“Load Predefined Simulink Environments”

**Introduced in R2019a**

## rlQAgent

Create Q-learning reinforcement learning agent

### Syntax

```
agent = rlQAgent(critic)
agent = rlQAgent(critic,opt)
```

### Description

`agent = rlQAgent(critic)` creates a Q-learning agent with default options and the specified critic representation. For more information on Q-learning agents, see “Q-Learning Agents”.

`agent = rlQAgent(critic,opt)` creates a Q-learning agent using the specified agent options to override the agent defaults.

### Examples

#### Create a Q-Learning Agent

Create an environment interface.

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

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

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

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

```
opt = rlQAgentOptions;  
opt.EpsilonGreedyExploration.Epsilon = 0.05;  
agent = rlQAgent(critic,opt);
```

## Input Arguments

### **critic** – Critic network representation

rlTableRepresentation object

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

### **opt** – Agent options

rlQAgentOptions object

Agent options, specified as an rlQAgentOptions object.

## Output Arguments

### **agent** – Q-learning agent

rlQAgent object

Q-learning agent, returned as an rlQAgent object.

## See Also

### **Functions**

rlQAgentOptions | train

### **Topics**

“Q-Learning Agents”

“Reinforcement Learning Agents”

“Train Reinforcement Learning Agents”

**Introduced in R2019a**

## rlQAgentOptions

Create options for Q-learning agent

### Syntax

```
opt = rlQAgentOptions
opt = rlQAgentOptions(Name,Value)
```

### Description

`opt = rlQAgentOptions` creates an `rlQAgentOptions` object for use as an argument when creating a Q-learning agent using all default settings. You can modify the object properties using dot notation.

`opt = rlQAgentOptions(Name,Value)` creates an options object using the specified name-value pairs to override default property values.

### Examples

#### Create Q-Learning Agent Options Object

Create an `rlQAgentOptions` object that specifies the agent sample time.

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

```
opt =
```

```
rlQAgentOptions with properties:
```

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

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

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

## Input Arguments

### Name-Value Pair Arguments

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

Example: `'DiscountFactor', 0.95`

### EpsilonGreedyExploration — Options for epsilon greedy exploration

`EpsilonGreedyExploration` object

Options for epsilon greedy exploration, specified as the comma-separated pair consisting of `'EpsilonGreedyExploration'` and an `EpsilonGreedyExploration` object with the following numeric value properties.

Property	Description
<code>Epsilon</code>	Probability threshold to either randomly select an action or select the action that maximizes the state-action value function. A larger value of <code>Epsilon</code> means that the agent randomly explores the action space at a higher rate.
<code>EpsilonMin</code>	Minimum value of <code>Epsilon</code>
<code>EpsilonDecay</code>	Decay rate

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

$$\text{Epsilon} = \text{Epsilon} * (1 - \text{EpsilonDecay})$$

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

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

## **SampleTime — Sample time of agent**

1 (default) | numeric value

Sample time of agent, specified as the comma-separated pair consisting of 'SampleTime' and a numeric value.

## **DiscountFactor — Discount factor applied to rewards**

numeric value

Discount factor applied to future rewards during training, specified as the comma-separated pair consisting of 'DiscountFactor' and a positive numeric value less than or equal to 1.

## **Output Arguments**

### **opt — Q-learning agent options**

rlQAgentOptions object

Q-learning agent options, returned as an `rlQAgentOptions` object. The object properties are described in “Name-Value Pair Arguments” on page 1-83.

## **See Also**

`rlQAgent`

## **Topics**

“Q-Learning Agents”

**Introduced in R2019a**



# rlRepresentation

Model representation for reinforcement learning agents

Use `rlRepresentation` to create a function approximator representation for the actor or critic of a reinforcement learning agent. To do so, you specify the observation and action signals for the training environment and options that affect the training of an agent that uses the representation. For more information on creating representations, see “Create Policy and Value Function Representations”.

## Syntax

```
rep = rlRepresentation(net,obsInfo,'Observation',obsNames)
rep = rlRepresentation(net,obsInfo,actInfo,'Observation',
obsNames,'Action',actNames)
```

```
tableCritic = rlRepresentation(tab)
```

```
critic = rlRepresentation(basisFcn,W0,obsInfo)
critic = rlRepresentation(basisFcn,W0,oaInfo)
actor = rlRepresentation(basisFcn,W0,obsInfo,actInfo)
```

```
rep = rlRepresentation( ____,repOpts)
```

## Description

`rep = rlRepresentation(net,obsInfo,'Observation',obsNames)` creates a representation for the deep neural network `net`. The observation names `obsNames` are the network input layer names. `obsInfo` contains the corresponding observation specifications for the training environment. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `rlACAgent` or `rlPGAgent` agent.

`rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action',actNames)` creates a representation with action signals specified by the names `actNames` and specification `actInfo`. Use this syntax to create a

representation for any actor, or for a critic that takes both observation and action as input, such as a critic for an `r1DQNAgent` or `r1DDPGAgent` agent.

`tableCritic = r1Representation(tab)` creates a critic representation for the value table or Q table `tab`. When you create a table representation, you specify the observation and action specifications when you create `tab`.

`critic = r1Representation(basisFcn,W0,obsInfo)` creates a linear basis function representation using the handle to a custom basis function `basisFcn` and initial weight vector `W0`. `obsInfo` contains the corresponding observation specifications for the training environment. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `r1ACAgent` or `r1PGAgent` agent.

`critic = r1Representation(basisFcn,W0,oaInfo)` creates a linear basis function representation using the specification cell array `oaInfo`, where `oaInfo = {obsInfo,actInfo}`. Use this syntax to create a representation for a critic that takes both observations and actions as inputs, such as a critic for an `r1DQNAgent` or `r1DDPGAgent` agent.

`actor = r1Representation(basisFcn,W0,obsInfo,actInfo)` creates a linear basis function representation using the specified observation and action specifications, `obsInfo` and `actInfo`, respectively. Use this syntax to create a representation for an actor that takes observations as inputs and generates actions.

`rep = r1Representation( ____, repOpts)` creates a representation using additional options that specify learning parameters for the representation when you train an agent. Available options include the optimizer used for training and the learning rate. Use `r1RepresentationOptions` to create the options set `repOpts`. You can use this syntax with any of the previous input-argument combinations.

## Examples

### Create Actor and Critic Representations

Create an actor representation and a critic representation that you can use to define a reinforcement learning agent such as an Actor Critic (AC) agent.

For this example, create actor and critic representations for an agent that can be trained against the cart-pole environment described in “Train AC Agent to Balance Cart-Pole

System". First, create the environment. Then, extract the observation and action specifications from the environment. You need these specifications to define the agent and critic representations.

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

For a state-value-function critic such as those used for AC or PG agents, the inputs are the observations and the output should be a scalar value, the state value. For this example, create the critic representation using a deep neural network with one output, and with observation signals corresponding to  $x$ ,  $\dot{x}$ ,  $\theta$ ,  $\dot{\theta}$  as described in "Train AC Agent to Balance Cart-Pole System". You can obtain the number of observations from the `obsInfo` specification. Name the network layer input 'observation'.

```
numObservation = obsInfo.Dimension(1);
criticNetwork = [
    imageInputLayer([numObservation 1 1], 'Normalization', 'none', 'Name', 'observation')
    fullyConnectedLayer(1, 'Name', 'CriticFC')];
```

Specify options for the critic representation using `rlRepresentationOptions`. These options control parameters of critic network learning, when you train an agent that incorporates the critic representation. For this example, set the learning rate to 0.05 and the gradient threshold to 1.

```
rep0pts = rlRepresentationOptions('LearnRate', 5e-2, 'GradientThreshold', 1);
```

Create the critic representation using the specified neural network and options. Also, specify the action and observation information for the critic. For more information, see `rlRepresentation`. Set the observation name to 'observation', which is the name you used when you created the network input layer for `criticNetwork`.

```
critic = rlRepresentation(criticNetwork, obsInfo, 'Observation', {'observation'}, rep0pts)
critic =
    rlLayerRepresentation with properties:
        Options: [1x1 rl.option.rlRepresentationOptions]
```

Similarly, create a network for the actor. An AC agent decides which action to take given observations using an actor representation. For an actor, the inputs are the observations, and the output depends on whether the action space is discrete or continuous. For the actor of this example, there are two possible discrete actions, -10 or 10. Thus, to create

the actor, use a deep neural network with the same observation input as the critic, that can output these two values. You can obtain the number of actions from the `actInfo` specification. Name the output `'action'`.

```
numAction = numel(actInfo.Elements);
actorNetwork = [
    imageInputLayer([4 1 1], 'Normalization','none','Name','observation')
    fullyConnectedLayer(numAction,'Name','action')];
```

Create the actor representation using the observation name and specification and the action name and specification. Use the same representation options.

```
actor = rlRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{ 'observation' }, 'Action',{ 'action' },repOpts)
```

```
actor =
    rlLayerRepresentation with properties:

    Options: [1x1 rl.option.rlRepresentationOptions]
```

You can now use the actor and critic representations to create an AC agent.

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

```
agent =
    rlACAgent with properties:

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

For additional examples showing how to create actor and critic representations for different agent types, see:

- “Train DDPG Agent to Control Double Integrator System”
- “Train DQN Agent to Balance Cart-Pole System”

### Create Q Table Representation

Create an environment interface.

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

Create a Q table using the action and observation specifications from the environment.

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

Create a representation for the Q table.

```
tableRep = rlRepresentation(qTable);
```

### Create Linear Basis Function Critic Representation

Assume that you have an environment, `env`. Obtain the observation and action specifications from the environment.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create a custom basis function. In this case, use the quadratic basis function from “Train Custom LQR Agent”.

```
function B = computeQuadraticBasis(x,u,n)
z = cat(1,x,u);
idx = 1;
for r = 1:n
    for c = r:n
        if idx == 1
            B = z(r)*z(c);
        else
            B = cat(1,B,z(r)*z(c));
        end
        idx = idx + 1;
    end
end
```

Compute any dimensions and parameters required for your basis function.

```
nQ = size(obj.Q,1);
nR = size(obj.R,1);
n = nQ+nR;
```

Set an initial weight vector.

```
w0 = 0.1*ones(0.5*(n+1)*n,1);
```

Create a representation using a handle to the custom basis function.

```
critic = rlRepresentation(@(x,u) computeQuadraticBasis(x,u,n),w0,obsInfo,actInfo);
```

## Input Arguments

### **net** — Deep neural network for actor or critic

array of layers | layer graph

Deep neural network for actor or critic, specified as an array of layers or a layer graph. For more information on creating deep neural networks for reinforcement learning, see “Create Policy and Value Function Representations”.

### **obsNames** — Observation names

cell array of character vectors

Observation names, specified as a cell array of character vectors. The observation names are the network input layer names you specify when you create `net`. The names in `obsNames` must be the same order as the observation specifications in `obsInfo`.

Example: {'observation'}

### **obsInfo** — Observation specification

spec object | array of spec objects

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

### **actNames** — Action name

single-element cell array that contains a character vector

Action name, specified as a single-element cell array that contains a character vector. The action name is the network layer name you specify when you create `net`. For `critic`

networks, this layer is the first layer of the action input path. For actors, this layer is the last layer of the action output path.

Example: `{'action'}`

### **actInfo — Action specification**

spec object

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

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

### **tab — Value table or Q table for critic**

rlTable object

Value table or Q table for critic, specified as an `rlTable` object. The learnable parameters of a table representation are the elements of `tab`.

### **basisFcn — Custom basis function**

function handle

Custom basis function, specified as a function handle to a user-defined function. For a linear basis function representation, the output of the representation is  $f = W'B$ , where  $W$  is a weight array and  $B$  is the column vector returned by the custom basis function. The learnable parameters of a linear basis function representation are the elements of  $W$ .

When creating:

- A critic representation with observation inputs only, your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here `obs1` to `obsN` are observations in the same order and with the same data type and dimensions as the observation specifications in `obsInfo`.

- A critic representation with observation and action inputs, your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN,act)
```

Here `obs1` to `obsN` are observations in the same order and with the same data type and dimensions as the observation specifications in the first element of `oaInfo`, and `act` has the same data type and dimensions as the action specification in the second element of `oaInfo`.

- An actor representation, your basis function must have the following signature.

```
B = myBasisFunction(obs1,obs2,...,obsN)
```

Here, `obs1` to `obsN` are observations in the same order and with the same data type and dimensions as the observation specifications in `obsInfo`. The data types and dimensions of the action specification in `actInfo` affect the data type and dimensions of `f`.

Example: `@(x,u) myBasisFunction(x,u)`

## **W0 — Initial value for linear basis function weight vector**

column vector | array

Initial value for linear basis function weight array, `W`, specified as one of the following:

- Column vector — When creating a critic representation or an actor representation with a continuous scalar action signal
- Array — When creating an actor representation with a column vector continuous action signal or a discrete action space.

## **oaInfo — Observation and action specifications**

cell array

Observation and action specifications for creating linear basis function critic representations, specified as the cell array `{obsInfo,actInfo}`.

## **repOpts — Representation options**

`rlRepresentationOptions` object

Representation options, specified as an option set that you create with `rlRepresentationOptions`. Available options include the optimizer used for training and the learning rate. See `rlRepresentationOptions` for details.



## Output Arguments

### **rep** — Deep neural network representation

`rllibLayerRepresentation` object

Deep neural network representation, returned as an `rllibLayerRepresentation` object. Use this representation to create an agent for reinforcement learning. For more information, see “Reinforcement Learning Agents”.

### **tableCritic** — Value or Q table critic representation

`rllibTableRepresentation` object

Value or Q table critic representation, returned as an `rllibTableRepresentation` object. Use this representation to create an agent for reinforcement learning. For more information, see “Reinforcement Learning Agents”.

### **critic** — Linear basis function critic representation

`rllibLinearBasisRepresentation` object

Linear basis function critic representation, returned as and `rllibLinearBasisRepresentation` object. Use this representation to create an agent for reinforcement learning. For more information, see “Reinforcement Learning Agents”.

### **actor** — Linear basis function actor representation

`rllibLinearBasisRepresentation` object

Linear basis function actor representation, returned as and `rllibLinearBasisRepresentation` object. Use this representation to create an agent for reinforcement learning. For more information, see “Reinforcement Learning Agents”.

## See Also

### Functions

`getActionInfo` | `getObservationInfo` | `rllibRepresentationOptions`

### Topics

“Create Policy and Value Function Representations”

“Reinforcement Learning Agents”

**Introduced in R2019a**

# rlRepresentationOptions

Create options for reinforcement learning agent representations

## Syntax

```
repOpts = rlRepresentationOptions  
repOpts = rlRepresentationOptions(Name,Value)
```

## Description

`repOpts = rlRepresentationOptions` returns the default options for defining a representation for a reinforcement learning agent.

`repOpts = rlRepresentationOptions(Name,Value)` option set using the specified name-value pairs to override default option values.

## Examples

### Configure Options for Creating Representation

Create an options set for creating a critic or actor representation for a reinforcement learning agent. Set the learning rate for the representation to 0.05, and set the gradient threshold to 1. You can set the options using Name,Value pairs when you create the options set. Any options that you do not explicitly set have their default values.

```
repOpts = rlRepresentationOptions('LearnRate',5e-2,...  
                                'GradientThreshold',1)  
  
repOpts =  
    rlRepresentationOptions with properties:  
  
        LearnRate: 0.0500  
        Optimizer: "adam"  
        OptimizerParameters: [1x1 rl.option.OptimizerParameters]
```

```
    GradientThreshold: 1
  GradientThresholdMethod: "l2norm"
  L2RegularizationFactor: 1
    UseDevice: "CPU"
  MiniBatchSize: Inf
```

Alternatively, create a default options set and use dot notation to change some of the values.

```
repOpts = rlRepresentationOptions;
repOpts.LearnRate = 5e-2;
repOpts.GradientThreshold = 1

repOpts =
  rlRepresentationOptions with properties:

    LearnRate: 0.0500
    Optimizer: "adam"
  OptimizerParameters: [1x1 rl.option.OptimizerParameters]
    GradientThreshold: 1
  GradientThresholdMethod: "l2norm"
  L2RegularizationFactor: 1
    UseDevice: "CPU"
  MiniBatchSize: Inf
```

If you want to change the properties of the `OptimizerParameters` option, use dot notation to access them.

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

ans =
  OptimizerParameters with properties:

    Momentum: "Not applicable"
    Epsilon: 1.0000e-07
  GradientDecayFactor: 0.9000
  SquaredGradientDecayFactor: 0.9990
```

# Input Arguments

## Name-Value Pair Arguments

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

Example: `'Optimizer','rmsprop'`

### **LearnRate** — Learning rate for the representation

0.01 (default) | positive scalar

Learning rate for the representation, specified as the comma-separated pair consisting of `'LearnRate'` and a positive scalar. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Example: `'LearnRate',0.025`

### **Optimizer** — Optimizer for representation

"adam" (default) | "sgdm" | "rmsprop"

Optimizer for training the network of the representation, specified as the comma-separated pair consisting of `'Optimizer'` and one of the following strings:

- `"adam"` — Use the Adam optimizer. You can specify the decay rates of the gradient and squared gradient moving averages using the `GradientDecayFactor` and `SquaredGradientDecayFactor` fields of the `OptimizerParameters` option.
- `"sgdm"` — Use the stochastic gradient descent with momentum (SGDM) optimizer. You can specify the momentum value using the `Momentum` field of the `OptimizerParameters` option.
- `"rmsprop"` — Use the RMSProp optimizer. You can specify the decay rate of the squared gradient moving average using the `SquaredGradientDecayFactor` fields of the `OptimizerParameters` option.

For more information about these optimizers, see “Stochastic Gradient Descent” (Deep Learning Toolbox) in the Algorithms section of `trainingOptions` in Deep Learning Toolbox.

Example: 'Optimizer', "sgdm"

**OptimizerParameters — Applicable parameters for optimizer**

OptimizerParameters object

Applicable parameters for the optimizer, specified as the comma-separated pair consisting of 'OptimizerParameters' and an OptimizerParameters object.

The OptimizerParameters object has the following properties.

<p>Momentum</p>	<p>Contribution of previous step, specified as a scalar from 0 to 1. A value of 0 means no contribution from the previous step. A value of 1 means maximal contribution.</p> <p>This parameter applies only when <code>Optimizer</code> is "sgdm". In that case, the default value is 0.9. This default value works well for most problems.</p>
<p>Epsilon</p>	<p>Denominator offset, specified as a positive scalar. The optimizer adds this offset to the denominator in the network parameter updates to avoid division by zero.</p> <p>This parameter applies only when <code>Optimizer</code> is "adam" or rmsprop. In that case, the default value is <math>10^{-8}</math>. This default value works well for most problems.</p>
<p>GradientDecayFactor</p>	<p>Decay rate of gradient moving average, specified as a positive scalar from 0 to 1.</p> <p>This parameter applies only when <code>Optimizer</code> is "adam". In that case, the default value is 0.9. This default value works well for most problems.</p>

<b>SquaredGradientDecayFactor</b>	Decay rate of squared gradient moving average, specified as a positive scalar from 0 to 1.  This parameter applies only when <b>Optimizer</b> is "adam" or "rmsprop". In that case, the default value is 0.999. This default value works well for most problems.
-----------------------------------	--

When a particular property of **OptimizerParameters** is not applicable to the optimizer type specified in the **Optimizer** option, that property is set to "Not applicable".

To change the default values, create an **rlRepresentationOptions** set and use dot notation to access and change the properties of **OptimizerParameters**.

```
repOpts = rlRepresentationOptions;
repOpts.OptimizerParameters.Epsilon = 1e-7;
```

### **GradientThreshold — Threshold value for gradient**

Inf (default) | positive scalar

Threshold value for the representation gradient, specified as the comma-separated pair consisting of 'GradientThreshold' and Inf or a positive scalar. If the gradient exceeds this value, the gradient is clipped as specified by the **GradientThresholdOption**. Clipping the gradient limits how much the network parameters change in a training iteration.

Example: 'GradientThreshold',1

### **GradientThresholdMethod — Gradient threshold method**

"l2norm" (default) | "global-l2norm" | "absolute-value"

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as the comma-separated pair consisting of 'GradientThresholdMethod' and one of the following strings:

- "l2norm" — If the  $L_2$  norm of the gradient of a learnable parameter is larger than **GradientThreshold**, then scale the gradient so that the  $L_2$  norm equals **GradientThreshold**.
- "global-l2norm" — If the global  $L_2$  norm,  $L$ , is larger than **GradientThreshold**, then scale all gradients by a factor of **GradientThreshold**/ $L$ . The global  $L_2$  norm considers all learnable parameters.

- "absolute-value" — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than `GradientThreshold`, then scale the partial derivative to have magnitude equal to `GradientThreshold` and retain the sign of the partial derivative.

For more information, see “Gradient Clipping” (Deep Learning Toolbox) in the Algorithms section of `trainingOptions` in Deep Learning Toolbox.

Example: `'GradientThresholdMethod','absolute-value'`

## **L2RegularizationFactor — Factor for $L_2$ regularization**

0.0001 (default) | nonnegative scalar

Factor for  $L_2$  regularization (weight decay), specified as the comma-separated pair consisting of `'L2RegularizationFactor'` and a nonnegative scalar. For more information, see “L2 Regularization” (Deep Learning Toolbox) in the Algorithms section of `trainingOptions` in Deep Learning Toolbox.

To avoid overfitting when using a representation with many parameters, consider increasing the `L2RegularizationFactor` option.

Example: `'L2RegularizationFactor',0.0005`

## **UseDevice — Computation device for training**

"cpu" (default) | "gpu"

Computation device for training an agent that uses the representation, specified as the comma-separated pair consisting of `'UseDevice'` and either "cpu" or "gpu".

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

Example: `'UseDevice','gpu'`

## **Output Arguments**

### **repOpts — Representation options**

`rlRepresentationOptions` object

Option set for defining a representation for a reinforcement learning agent., returned as an `rlRepresentationOptions` object. The property values of `repOpts` are initialized



to the default values or to the values you specify with `Name, Value` pairs. You can further modify the property values using dot notation. Use the options set as an input argument with `rlRepresentation` when you create reinforcement learning representations.

## See Also

### Functions

`rlRepresentation`

**Introduced in R2019a**

## rlSARSAgent

Create SARSA reinforcement learning agent

### Syntax

```
agent = rlSARSAgent(critic)
agent = rlQAgent(critic,opt)
```

### Description

`agent = rlSARSAgent(critic)` creates a SARSA agent with default options and the specified critic representation. For more information on SARSA agents, see “SARSA Agents”.

`agent = rlQAgent(critic,opt)` creates a SARSA agent using the specified agent options to override the agent defaults.

### Examples

#### Create a SARSA Agent

Create an environment interface.

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

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

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

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

```
opt = rLSARSAgentOptions;  
opt.EpsilonGreedyExploration.Epsilon = 0.05;  
agent = rLSARSAgent(critic,opt);
```

## Input Arguments

### **critic** — Critic network representation

rLTableRepresentation object

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

### **opt** — Agent options

rLSARSAgentOptions object

Agent options, specified as an rLSARSAgentOptions object.

## Output Arguments

### **agent** — SARSA agent

rLSARSAgent object

SARSA agent, returned as an rLSARSAgent object.

## See Also

rLSARSAgentOptions | train

## Topics

“SARSA Agents”

“Reinforcement Learning Agents”

“Train Reinforcement Learning Agents”

**Introduced in R2019a**

## rlSARSAAgentOptions

Create options for SARSA agent

### Syntax

```
opt = rlSARSAAgentOptions
opt = rlSARSAAgentOptions(Name,Value)
```

### Description

`opt = rlSARSAAgentOptions` creates an `rlSARSAAgentOptions` object for use as an argument when creating a SARSA agent using all default settings. You can modify the object properties using dot notation.

`opt = rlSARSAAgentOptions(Name,Value)` creates an options object using the specified name-value pairs to override default property values.

### Examples

#### Create SARSA Agent Options Object

Create an `rlSARSAAgentOptions` object that specifies the agent sample time.

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

```
opt =
```

```
rlSARSAAgentOptions with properties:
```

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

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

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

## Input Arguments

### Name-Value Pair Arguments

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

Example: `'DiscountFactor', 0.95`

### EpsilonGreedyExploration — Options for epsilon greedy exploration

`EpsilonGreedyExploration` object

Options for epsilon greedy exploration, specified as the comma-separated pair consisting of `'EpsilonGreedyExploration'` and an `EpsilonGreedyExploration` object with the following numeric value properties.

Property	Description
<code>Epsilon</code>	Probability threshold to either randomly select an action or select the action that maximizes the state-action value function. A larger value of <code>Epsilon</code> means that the agent randomly explores the action space at a higher rate.
<code>EpsilonMin</code>	Minimum value of <code>Epsilon</code>
<code>EpsilonDecay</code>	Decay rate

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

$$\text{Epsilon} = \text{Epsilon} * (1 - \text{EpsilonDecay})$$

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

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

## **SampleTime — Sample time of agent**

1 (default) | numeric value

Sample time of agent, specified as the comma-separated pair consisting of 'SampleTime' and a numeric value.

## **DiscountFactor — Discount factor applied to rewards**

numeric value

Discount factor applied to future rewards during training, specified as the comma-separated pair consisting of 'DiscountFactor' and a positive numeric value less than or equal to 1.

## **Output Arguments**

### **opt — SARSA agent options**

rLSARSAgentOptions object

SARSA agent options, returned as an rLSARSAgentOptions object. The object properties are described in “Name-Value Pair Arguments” on page 1-105.

## **See Also**

rLSARSAgent

## **Topics**

“SARSA Agents”

**Introduced in R2019a**

# rlSimulationOptions

Options for simulating reinforcement learning environments

## Syntax

```
simOpts = rlSimulationOptions
simOpts = rlSimulationOptions(Name,Value)
```

## Description

`simOpts = rlSimulationOptions` returns the default options for simulating a reinforcement learning environment against an agent. You use simulation options to specify parameters about the simulation such as the maximum number of steps to run per simulation and the number of simulations to run. After you configure the options, use `simOpts` as an input argument for `sim`.

`simOpts = rlSimulationOptions(Name,Value)` creates an option set for simulation using the specified name-value pairs to override default option values.

## Examples

### Configure Options for Simulation

Create an options set for simulating a reinforcement learning environment. Set the number of steps to simulate to 1000, and configure the options to run three simulations.

You can set the options using Name,Value pairs when you create the options set. Any options that you do not explicitly set have their default values.

```
simOpts = rlSimulationOptions(...
    'MaxSteps',1000,...
    'NumSimulations',3)

simOpts =
    rlSimulationOptions with properties:
```

```
        MaxSteps: 1000
    NumSimulations: 3
    StopOnError: "off"
```

Alternatively, create a default options set and use dot notation to change some of the values.

```
simOpts = rlSimulationOptions;
simOpts.MaxSteps = 1000;
simOpts.NumSimulations = 3;

simOpts
simOpts =
    rlSimulationOptions with properties:

        MaxSteps: 1000
    NumSimulations: 3
    StopOnError: "off"
```

## Input Arguments

### Name-Value Pair Arguments

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

Example: `'MaxSteps', 1000, 'StopOnError', "On"`

#### **MaxSteps** — Number of steps to run the simulation

500 (default) | positive integer

Number of steps to run the simulation, specified as the comma-separated pair consisting of `'MaxSteps'` and a positive integer. In general, you define episode termination conditions in the environment. This value is the maximum number of steps to run in the simulation if those termination conditions are not met.



Example: 'MaxSteps', 1000

### **NumSimulations — Number of simulations**

1 (default) | positive integer

Number of simulations to run, specified as the comma-separated pair consisting of 'NumSimulations' and a positive integer. At the start of each simulation, `sim` resets the environment. You specify what happens on environment reset when you create the environment. For instance, resetting the environment at the start of each episode can include randomizing initial state values, if you configure your environment to do so. In that case, running multiple simulations allows you to validate performance of a trained agent over a range of initial conditions.

Example: 'NumSimulations', 10

### **StopOnError — Stop simulation when error occurs**

"on" (default) | "off"

Stop simulation when an error occurs, specified as "off" or "on". When this option is "off", errors are captured and returned in the `SimulationInfo` output of `sim`, and simulation continues.

## **Output Arguments**

### **simOpts — Option set for simulating reinforcement learning environments**

rlSimulationOptions object

Option set for simulating reinforcement learning environments, returned as a `rlSimulationOptions` object. The property values of `simOpts` are initialized to the default values or to the values you specify with `Name, Value` pairs. You can further modify the property values using dot notation. Use the options set as an input argument with `sim` when you simulate reinforcement learning environment against an agent.

## **See Also**

`sim`

**Introduced in R2019a**

## rlSimulinkEnv

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

### Syntax

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

### Description

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

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

### Examples

#### Reinforcement Learning Environment for Simulink models

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

Open the model.

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

Assign the agent block path information, and create `rlNumericSpec` and `rlFiniteSetSpec` objects for the observation and action information. You can use `dot`

notation to assign property values of the `rINumericSpec` and `rIFiniteSetSpec` objects.

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

```
obsInfo =
  rINumericSpec with properties:

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

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

```
actInfo =
  rIFiniteSetSpec with properties:

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

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

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

```
env = rISimulinkEnv(mdl,agentBlk,obsInfo,actInfo)
```

```
env =
  SimulinkEnvWithAgent with properties:

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

You can also include a reset function using dot notation. For this example, consider randomly initializing `theta0` in the model workspace.

```
env.ResetFcn = @(in) setVariable(in, 'theta0', randn, 'Workspace', mdl)

env =
  SimulinkEnvWithAgent with properties:

        Model: "rlSimplePendulumModel"
    AgentBlock: "rlSimplePendulumModel/RL Agent"
        ResetFcn: @(in)setVariable(in, 'theta0', randn, 'Workspace', mdl)
    UseFastRestart: 'on'
```

## Input Arguments

### **mdl** — Simulink model name

string | character vector

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

### **agentBlock** — Agent block path

string | character vector

Agent block path, specified as a string or character vector.

For more information, see [RL Agent](#).

### **obsInfo** — Observation information

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

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

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

For more information, see [getObservationInfo](#).

### **actInfo** — Action information

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

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

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

For more information, see `getActionInfo`.

### **fastRestartToggle — Option to toggle fast restart**

`'on'` (default) | `'off'`

Option to toggle fast restart, specified as either `'on'` or `'off'`. Fast restart allows you to perform iterative simulations without compiling a model or terminating the simulation each time.

For more information on fast restart, see “How Fast Restart Improves Iterative Simulations” (Simulink).

## **Output Arguments**

### **env — Reinforcement learning environment**

`SimulinkEnvWithAgent` object

Reinforcement learning environment, returned as a `SimulinkEnvWithAgent` object.

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

## **See Also**

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

## **Topics**

“Train DDPG Agent to Control Double Integrator System”

“Train DDPG Agent to Swing Up and Balance Pendulum”

“Train DDPG Agent to Swing Up and Balance Cart-Pole System”

“Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal”

“Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation”

“Train DDPG Agent for Adaptive Cruise Control”  
“How Fast Restart Improves Iterative Simulations” (Simulink)

**Introduced in R2019a**

# rlTrainingOptions

Options for training reinforcement learning agents

## Syntax

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

## Description

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

`trainOpts = rlTrainingOptions(Name,Value)` creates an option set for training using the specified name-value pairs to override default option values.

## Examples

### Configure Options for Training

Create an options set for training a reinforcement learning agent. Set the maximum number of episodes and the maximum steps per episode to 1000. Configure the options to stop training when the average reward equals or exceeds 480, and turn on both the command-line display and the Reinforcement Learning Episode Manager for displaying training results. You can set the options using Name,Value pairs when you create the options set. Any options that you do not explicitly set have their default values.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',1000,...
    'MaxStepsPerEpisode',1000,...
    'StopTrainingCriteria',"AverageReward",...
```

```
'StopTrainingValue',480,...
'Verbose',true,...
'Plots',"training-progress")

trainOpts =
  rlTrainingOptions with properties:

        MaxEpisodes: 1000
        MaxStepsPerEpisode: 1000
    ScoreAveragingWindowLength: 5
        StopTrainingCriteria: "AverageReward"
        StopTrainingValue: 480
        SaveAgentCriteria: "None"
        SaveAgentValue: "None"
        SaveAgentDirectory: "savedAgents"
        Parallelization: "none"
    ParallelizationOptions: []
        StopOnError: "on"
        Verbose: 1
        Plots: "training-progress"
```

Alternatively, create a default options set and use dot notation to change some of the values.

```
trainOpts = rlTrainingOptions;
trainOpts.MaxEpisodes = 1000;
trainOpts.MaxStepsPerEpisode = 1000;
trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = 480;
trainOpts.Verbose = true;
trainOpts.Plots = "training-progress";
```

```
trainOpts
```

```
trainOpts =
  rlTrainingOptions with properties:

        MaxEpisodes: 1000
        MaxStepsPerEpisode: 1000
    ScoreAveragingWindowLength: 5
        StopTrainingCriteria: "AverageReward"
        StopTrainingValue: 480
        SaveAgentCriteria: "None"
        SaveAgentValue: "None"
```



```

    SaveAgentDirectory: "savedAgents"
      Parallelization: "none"
ParallelizationOptions: []
    StopOnError: "on"
      Verbose: 1
      Plots: "training-progress"

```

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

### Configure Parallel Computing Options for Training

To turn on parallel computing for training a reinforcement learning agent, you first create an options set with the `Parallelization` option set to a value other than "none". For this example, configure an options set for asynchronous parallel training.

```
trainOpts = rlTrainingOptions('Parallelization','async');
```

When you set `Parallelization` to this value, the software populates the `ParallelizationOptions` option with a default `ParallelTraining` object.

```

trainOpts.ParallelizationOptions
ans =
  ParallelTraining with properties:
      DataToSendFromWorkers: "Experiences"
      StepsUntilDataIsSent: -1
      WorkerRandomSeeds: -1
  TransferBaseWorkspaceVariables: "on"
      AttachedFiles: []
      SetupFcn: []
      CleanupFcn: []

```

You can further configure the parallel computing options using dot notation. For instance, configure the workers to send data to the host every 100 steps within a training episode. Further configure the workers to send gradient data, rather than experience data.

```

trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 100;
trainOpts.ParallelizationOptions.DataToSendFromWorkers = "Gradients";
trainOpts.ParallelizationOptions

```

```
ans =  
  ParallelTraining with properties:  
      DataToSendFromWorkers: "Gradients"  
      StepsUntilDataIsSent: 100  
      WorkerRandomSeeds: -1  
  TransferBaseWorkspaceVariables: "on"  
      AttachedFiles: []  
      SetupFcn: []  
      CleanupFcn: []
```

You can now use `trainOpts` as an input argument to the `train` command to perform training with parallel computing.

## Input Arguments

### Name-Value Pair Arguments

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

Example:

```
'StopTrainingCriteria', "AverageReward", 'StopTrainingValue', 100
```

#### **MaxEpisodes — Maximum number of episodes to train the agent**

500 (default) | positive integer

Maximum number of episodes to train the agent, specified as the comma-separated pair consisting of `'MaxEpisodes'` and a positive integer. Regardless of other criteria for termination, training terminates after this many episodes.

Example: `'MaxEpisodes', 1000`

#### **MaxStepsPerEpisode — Maximum number of steps to run per episode**

500 (default) | positive integer

Maximum number of steps to run per episode, specified as the comma-separated pair consisting of `'MaxStepsPerEpisode'` and a positive integer. In general, you define

episode termination conditions in the environment. This value is the maximum number of steps to run in the episode if those termination conditions are not met.

Example: `'MaxStepsPerEpisode', 1000`

### **ScoreAveragingWindowLength — Window length for averaging**

5 (default) | positive integer

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

Example: `'ScoreAveragingWindowLength', 10`

### **StopTrainingCriteria — Training termination condition**

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

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

- `"AverageSteps"` — Stop training when the running average number of steps per episode equals or exceeds the critical value specified by the option `StopTrainingValue`. The average is computed using the window `'ScoreAveragingWindowLength'`.
- `"AverageReward"` — Stop training when the running average reward equals or exceeds the critical value.
- `"EpisodeReward"` — Stop training when the reward in the current episode equals or exceeds the critical value.
- `"GlobalStepCount"` — Stop training when the total number of steps in all episodes (the total number of times the agent is invoked) equals or exceeds the critical value.
- `"EpisodeCount"` — Stop training when the number of training episodes equals or exceeds the critical value.

Example: `'StopTrainingCriteria', "AverageReward"`

### **StopTrainingValue — Critical value of training termination condition**

500 (default) | scalar

Critical value of training termination condition, specified as the comma-separated pair consisting of 'StopTrainingValue' and a scalar. Training terminates when the termination condition specified by the StopTrainingCriteria option equals or exceeds this value. For instance, if StopTrainingCriteria is "AverageReward", and StopTrainingValue is 100, then training terminates when the average reward over the number of episodes specified in 'ScoreAveragingWindowLength' equals or exceeds 100.

Example: 'StopTrainingValue',100

### **SaveAgentCriteria — Condition for saving agent during training**

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

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

- "none" — Do not save any agents during training.
- "EpisodeReward" — Save agent when the reward in the current episode equals or exceeds the critical value.
- "AverageSteps" — Save agent when the running average number of steps per episode equals or exceeds the critical value specified by the option StopTrainingValue. The average is computed using the window 'ScoreAveragingWindowLength'.
- "AverageReward" — Save agent when the running average reward over all episodes equals or exceeds the critical value.
- "GlobalStepCount" — Save agent when the total number of steps in all episodes (the total number of times the agent is invoked) equals or exceeds the critical value.
- "EpisodeCount" — Save agent when the number of training episodes equals or exceeds the critical value.

Set this option to store candidate agents that perform well according to the criteria you specify. When you set this option to a value other than "none", the software sets the SaveAgentValue option to 500. You can change that value to specify the condition for saving the agent.

For instance, suppose you want to store for further testing any agent that yields an episode reward that equals or exceeds 100. To do so, set SaveAgentCriteria to "EpisodeReward" and set the SaveAgentValue option to 100. When an episode reward equals or exceeds 100, train saves the corresponding agent in a MAT-file in the folder specified by the SaveAgentDirectory option. The MAT-file is called AgentK.mat

where K is the number of the corresponding episode. The agent is stored within that MAT-file as `saved_agent`.

Example: `'SaveAgentCriteria', "EpisodeReward"`

### **SaveAgentValue — Critical value of condition for saving agent**

`"none" (default) | 500 | scalar`

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

When you specify a condition for saving candidate agents using `SaveAgentCriteria`, the software sets this value to 500. Change the value to specify the condition for saving the agent. See the `SaveAgentValue` option for more details.

Example: `'SaveAgentValue', 100`

### **SaveAgentDirectory — Folder for saved agents**

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

Folder for saved agents, specified as the comma-separated pair consisting of `'SaveAgentDirectory'` and a string or character vector. The folder name can contain a full or relative path. When an episode occurs that satisfies the condition specified by the `SaveAgentCriteria` and `SaveAgentValue` options, the software saves the agent in a MAT-file in this folder. If the folder doesn't exist, `train` creates it. When `SaveAgentCriteria` is `"none"`, this option is ignored and `train` does not create a folder.

Example: `'SaveAgentDirectory', pwd + "run1\Agents"`

### **UseParallel — Flag for using parallel training**

`false (default) | true`

Flag for using parallel training, specified as the comma-separated pair consisting of `'UseParallel'` and either `true` or `false`. Setting this option to `true` configures training to use the parallel computing, configured using the options in `ParallelizationOptions`.

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

Using parallel computing requires Parallel Computing Toolbox software.

Example: `'Parallelization', "async"`

## **ParallelizationOptions — Parallelization options to control parallel training**

ParallelTraining object

Parallelization options to control parallel training, specified as the comma-separated pair consisting of 'ParallelizationOptions' and a ParallelTraining object. For more information about training using parallel computing, see “Train Reinforcement Learning Agents”.

The ParallelTraining object has the following properties, which you can modify using dot notation after creating the rlTrainingOptions object.

### **Mode — Parallel computing mode**

"sync" (default) | "async"

Parallel computing mode, specified as one of the following:

- "async" — Use `parpool` to run asynchronous training on the available workers. In this case, workers pause execution until all workers are finished. The host updates the actor and critic parameters based on the results from all of the workers and sends the updated parameters to all workers.
- "sync" — Use `parpool` to run synchronous training on the available workers. In this case, workers send their data back to the host as soon as they finish and receive updated parameters from the host. The workers then continue with their task.

### **DataToSendFromWorkers — Type of data that workers send to the host**

"experiences" (default) | "gradients"

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

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

---

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

---

### **StepsUntilDataIsSent — When workers send data to host**

-1 (default) | positive integer

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

---

**Note**

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

**WorkerRandomSeeds — Randomizer initialization for workers**

-1 (default) | -2 | vector

Randomizer initialization for workers, specified as one the following:

- -1 — Assign a unique random seed to each worker. The value of the seed is the worker ID.
- -2 — Do not assign a random seed to the workers.
- Vector — Manually specify the random seed for each work. The number of elements in the vector must match the number of workers.

**TransferBaseWorkspaceVariables — Send model and workspace variables to parallel workers**

"on" (default) | "off"

Send model and workspace variables to parallel workers, specified as "on" or "off". When the option is "on", the host sends variables used in models and defined in the base MATLAB workspace to the host.

**AttachedFiles — Additional files to attach to the parallel pool**

[] (default) | string | string array

Additional files to attach to the parallel pool, specified as a string or string array.

**SetupFcn — Function to run before training starts**

[] (default) | function handle

Function to run before training starts, specified as a handle to a function having no input arguments. This function is run once per worker before training begins. Write this function to perform any processing that you need prior to training.

### **CleanupFcn — Function to run after training ends**

`[]` (default) | function handle

Function to run after training ends, specified as a handle to a function having no input arguments. You can write this function to clean up the workspace or perform other processing after training terminates.

### **Verbose — Display training progress on the command line**

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

Display training progress on the command line, specified as the logical values `false` (0) or `true` (1). Set to `true` to write information from each training episode to the MATLAB command line during training.

### **StopOnError — Stop training when error occurs**

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

Stop training when an error occurs during an episode, specified as `"on"` or `"off"`. When this option is `"off"`, errors are captured and returned in the `SimulationInfo` output of `train`, and training continues to the next episode.

### **Plots — Display training progress with the Episode Manager**

`"training-progress"` (default) | `"none"`

Display training progress with the Episode Manager, specified as `"training-progress"` or `"none"`. By default, calling `train` opens the Reinforcement Learning Episode Manager, which graphically and numerically displays information about the training progress, such as the reward for each episode, average reward, number of episodes, and total number of steps. (For more information, see `train`.) To turn off this display, set this option to `"none"`.

## **Output Arguments**

### **trainOpts — Option set for training reinforcement learning agents**

`rlTrainingOptions` object



Option set for training reinforcement learning agents, returned as an `rlTrainingOptions` object. The property values of `trainOpts` are initialized to the default values or to the values you specify with `Name, Value` pairs. You can further modify the property values using dot notation. Use the options set as an input argument with `train` when you train reinforcement learning agents.

## See Also

`train`

## Topics

“Train Reinforcement Learning Agents”

**Introduced in R2019a**

## setActor

**Package:** rl.agent

Set actor representation of reinforcement learning agent

## Syntax

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

## Description

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

## Examples

### Modify Actor Parameter Values

Assume that you have an existing trained reinforcement learning agent, `agent`.

Obtain the actor representation from the agent.

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

Obtain the learnable parameters from the critic.

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

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,'UniformOutput',false);
```

Set the parameter values of the critic to the new modified values.

```
actor = setLearnableParameterValues(actor,modifiedParams);
```

Set the critic in the agent to the new modified critic.

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

## Add Layer to Actor Representation

Assume that you have an existing reinforcement learning agent, `agent`.

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

```
originalActor = [
    imageInputLayer([4 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(2, 'Name', 'action')];
```

Create an actor representation with an additional fully connected layer.

```
actorNetwork = [
    imageInputLayer([4 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(3, 'Name', 'x');
    fullyConnectedLayer(2, 'Name', 'action')];
actor = rlRepresentation(actorNetwork, ...
    'Observation', {'state'}, getObservationInfo(env),
    'Action', {'action'}, getActionInfo(env));
```

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

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

## Input Arguments

### **oldAgent** — Original reinforcement learning agent

`rlDDPGAgent` object | `rlPGAgent` object | `rlACAgent` object

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

- `rlDDPGAgent` object

- `rlACAgent` object
- `rlPGAgent` object

### **actor** — Actor representation

`rlLayerRepresentation` object | `rlTableRepresentation` object

Actor representation object, specified as one of the following:

- `rlLayerRepresentation` object for deep neural network representations
- `rlTableRepresentation` object for value table or Q table representations

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

To create an actor representation, use one of the following methods:

- Create a representation using `rlRepresentation`.
- Obtain the existing actor representation from an agent using `getActor`.

## Output Arguments

### **newAgent** — Updated reinforcement learning agent

`rlDDPGAgent` object | `rlPGAgent` object | `rlACAgent` object

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

## See Also

`getActor` | `getCritic` | `getLearnableParameterValues` | `rlRepresentation` | `setCritic` | `setLearnableParameterValues`

## Topics

“Create Policy and Value Function Representations”

“Import Policy and Value Function Representations”

**Introduced in R2019a**

## setCritic

**Package:** rl.agent

Set critic representation of reinforcement learning agent

## Syntax

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

## Description

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

## Examples

### Modify Critic Parameter Values

Assume that you have an existing trained reinforcement learning agent, `agent`.

Obtain the critic representation from the agent.

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

Obtain the learnable parameters from the critic.

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

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2, params, 'UniformOutput', false);
```

Set the parameter values of the critic to the new modified values.

```
critic = setLearnableParameterValues(critic,modifiedParams);
```

Set the critic in the agent to the new modified critic.

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

### Add Layer to Critic Representation

Assume that you have an existing reinforcement learning agent, `agent`.

Further, assume that this agent has a critic representation that contains the following deep neural network structure.

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

Create an actor representation with an additional fully connected layer.

```
criticNetwork = [
    imageInputLayer([4 1 1], 'Normalization', 'none', 'Name', 'state')
    fullyConnectedLayer(3, 'Name', 'x');
    fullyConnectedLayer(1, 'Name', 'CriticFC')];
critic = rlRepresentation(criticNetwork, 'Observation', {'state'}, ...
    getObservationInfo(env));
```

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

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

### Remove Baseline Critic from PG Agent

Assume that you have an existing PG agent, `agent`, with a baseline critic representation. You can remove the baseline critic from the agent using `setCritic`.

```
agent = setCritic(agent, []);
```

When you remove the baseline critic in this way, the `UseBaseline` option of the agent is automatically set to `false`.

## Add Baseline Critic to PG Agent

Assume that you have an existing PG agent, `agent`, without a baseline critic representation. You can add a baseline critic to the agent using `setCritic`.

First, create a critic representation, assuming you have an existing critic network, `criticNetwork`.

```
baseline = rlRepresentation(criticNetwork, 'Observation', {'state'}, ...
    getObservationInfo(env));
```

Then, set the critic in the agent.

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

When you add a baseline critic in this way, the `UseBaseline` option of the agent is automatically set to `true`.

## Input Arguments

### **oldAgent** — Original reinforcement learning agent

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

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

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

### **critic** — Critic representation

`rlLayerRepresentation` object | `rlTableRepresentation` object

Critic representation object, returned as one of the following:



- `rLLayerRepresentation` object for deep neural network representations
- `rLTableRepresentation` object for value table or Q table representations

To create a critic representation, use one of the following methods:

- Create a representation using `rLRepresentation`.
- Obtain the existing critic representation from an agent using `getCritic`

## Output Arguments

### **newAgent — Updated reinforcement learning agent**

`rLQAgent` object | `rLSARSAgent` object | `rLDQNAgent` object | `rLDDPGAgent` object | `rLPGAgent` object | `rLACAgent` object

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

## See Also

`getActor` | `getCritic` | `getLearnableParameterValues` | `rLRepresentation` | `setActor` | `setLearnableParameterValues`

## Topics

“Create Policy and Value Function Representations”

“Import Policy and Value Function Representations”

**Introduced in R2019a**

## setLearnableParameterValues

**Package:** rl.util

Set learnable parameter values of policy or value function representation

### Syntax

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

### Description

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

### Examples

#### Modify Critic Parameter Values

Assume that you have an existing trained reinforcement learning agent, `agent`.

Obtain the critic representation from the agent.

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

Obtain the learnable parameters from the critic.

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

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2, params, 'UniformOutput', false);
```

Set the parameter values of the critic to the new modified values.

```
critic = setLearnableParameterValues(critic,modifiedParams);
```

Set the critic in the agent to the new modified critic.

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

## Modify Actor Parameter Values

Assume that you have an existing trained reinforcement learning agent, `agent`.

Obtain the actor representation from the agent.

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

Obtain the learnable parameters from the critic.

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

Modify the parameter values. For this example, simply multiply all of the parameters by 2.

```
modifiedParams = cellfun(@(x) x*2,params,'UniformOutput',false);
```

Set the parameter values of the critic to the new modified values.

```
actor = setLearnableParameterValues(actor,modifiedParams);
```

Set the critic in the agent to the new modified critic.

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

## Input Arguments

### **oldRep** — Original policy or value function representation

`rLLayerRepresentation` object | `rLTableRepresentation` object

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

- `rLLayerRepresentation` object for deep neural network representations
- `rLTableRepresentation` object for value table or Q table representations

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

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

### **val** — Learnable parameter values

cell array

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

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

## Output Arguments

### **newRep** — New policy or value function representation

`rlLayerRepresentation` object | `rlTableRepresentation` object

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

## See Also

`getActor` | `getCritic` | `getLearnableParameterValues` | `rlRepresentation` | `setActor` | `setCritic`

## Topics

“Create Policy and Value Function Representations”

“Import Policy and Value Function Representations”

**Introduced in R2019a**

# sim

**Package:** rl.env

Simulate a trained reinforcement learning agent within a specified environment

## Syntax

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

## Description

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

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

## Examples

### Simulate a Reinforcement Learning Environment

Simulate a reinforcement learning environment with an agent configured for that environment. For this example, load an environment and agent that are already configured. The environment is a discrete cart-pole environment created with `rlPredefinedEnv`. The agent is a Policy Gradient (`rlPGAgent`) agent. For more information about the environment and agent used in this example, see “Train PG Agent to Balance Cart-Pole System”.

```
rng(0); % for reproducibility
load RLSimExample.mat
env
env =
    CartPoleDiscreteAction with properties:
```

```
Gravity: 9.8000
MassCart: 1
MassPole: 0.1000
Length: 0.5000
MaxForce: 10
Ts: 0.0200
ThetaThresholdRadians: 0.2094
XThreshold: 2.4000
RewardForNotFalling: 1
PenaltyForFalling: -5
State: [4x1 double]
```

**agent**

```
agent =
  rLPGAgent with properties:

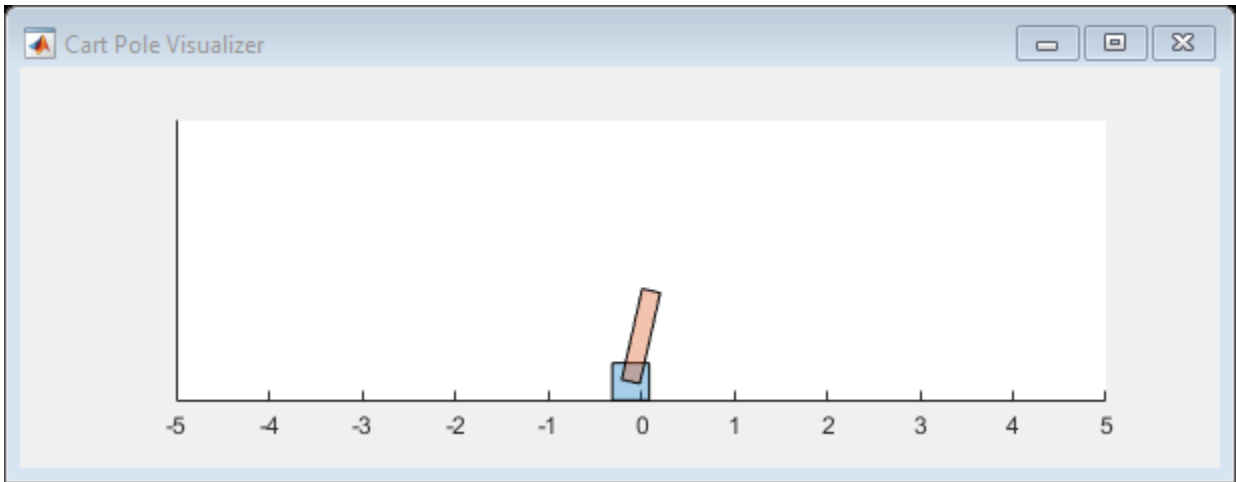
    AgentOptions: [1x1 rl.option.rLPGAgentOptions]
```

Typically, you train the agent using `train` and simulate the environment to test the performance of the trained agent. For this example, simulate the environment using the agent you loaded. Configure simulation options specifying that the simulation run for 100 steps..

```
simOpts = rlSimulationOptions('MaxSteps',100);
```

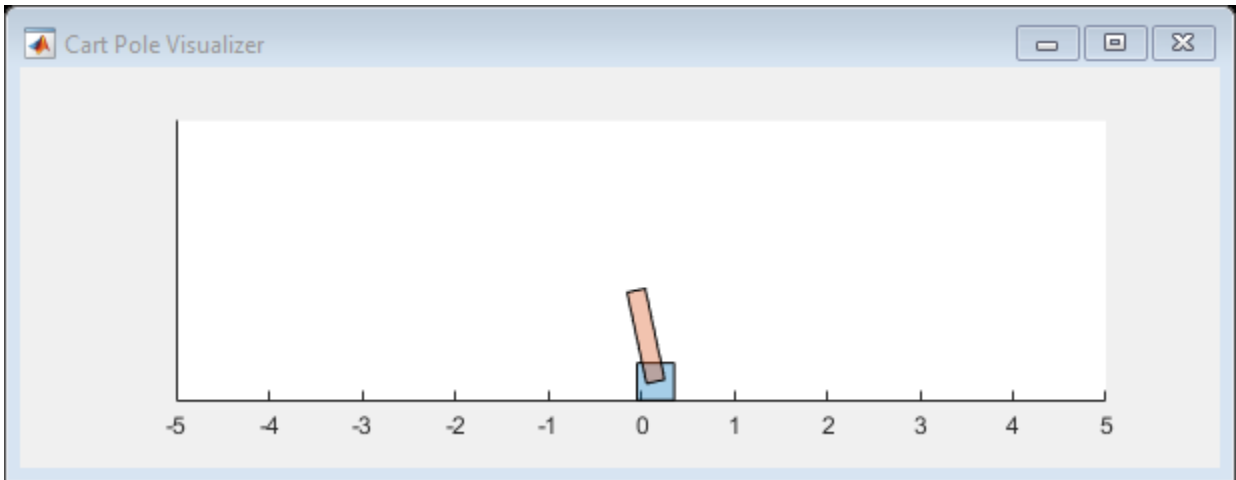
For the predefined cart-pole environment used in this example, you can use `plot` to generate a visualization of the cart-pole system. When you simulate the environment, this plot updates automatically so that you can watch the system evolve during the simulation.

```
plot(env);
```



Simulate the environment.

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



```
experience = struct with fields:  
    Observation: [1x1 struct]  
    Action: [1x1 struct]  
    Reward: [1x1 timeseries]  
    IsDone: [1x1 timeseries]
```

```
SimulationInfo: [1x1 struct]
```

The output structure `experience` records the observations collected from the environment, the action and reward, and other data collected during the simulation. Each field contains is a timeseries or a structure of timeseries data. For instance, `experience.Action` is a timeseries containing the action imposed on the cart-pole system by the agent at each step of the simulation.

```
experience.Action
```

```
ans = struct with fields:  
    CartPoleAction: [1x1 timeseries]
```

## Input Arguments

### **env — Environment**

reinforcement learning environment object

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

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

For more information about creating and configuring environments, see:

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

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

### **agent — Agent**

reinforcement learning agent object

Agent to train, specified as a reinforcement learning agent object, such as an `rlACAgent` or `rlDDPGAgent` object, or a custom agent. Before simulation, you must configure the



actor and critic representations of the agent. For more information about how to create and configure agents for reinforcement learning, see “Reinforcement Learning Agents”.

### **simOpts — Simulation options**

`rlSimulationOptions` object

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

- Number of steps per simulation
- Number of simulations to run

For details, see `rlSimulationOptions`.

## **Output Arguments**

### **experience — Simulation results**

structure | structure array

Simulation results, returned as a structure or structure array. The number of elements in the array is equal to the number of simulations specified by the `NumSimulations` option of `rlSimulationOptions`. The fields of the `experience` structure are as follows.

### **Observation — Observations**

structure

Observations collected from the environment, returned as a structure with fields corresponding to the observations specified in the environment. Each field contains a `timeseries` of length  $N + 1$ , where  $N$  is the number of simulation steps.

To obtain the current observation and the next observation for a given simulation step, use code such as the following, assuming one of the fields of `Observation` is `obs1`.

```
Obs = getSamples(experience.Observation.obs1,1:N);
NextObs = getSamples(experience.Observation.obs1,2:N+1);
```

These values can be useful if you are writing your own training algorithm using `sim` to generate experiences for training.

### **Action — Actions**

structure

Actions computed by the agent, returned as a structure with fields corresponding to the action signals specified in the environment. Each field contains a `timeseries` of length  $N$ , where  $N$  is the number of simulation steps.

## **Reward — Rewards**

`timeseries`

Reward at each step in the simulation, returned as a `timeseries` of length  $N$ , where  $N$  is the number of simulation steps.

## **IsDone — Flag indicating termination of episode**

`timeseries`

Flag indicating termination of episode, returned as a `timeseries` of a scalar logical signal. This flag is set at each step by the environment, according to conditions you specify for episode termination when you configure the environment. When the environment sets this flag to 1, simulation terminates.

## **SimulationInfo — Information collected during simulation**

structure | vector of `Simulink.SimulationOutput` objects

Information collected during simulation, returned as:

- For MATLAB environments, a structure containing the field `SimulationError`. This structure contains any errors that occurred during simulation.
- For Simulink environments, a `Simulink.SimulationOutput` object containing simulation data. Recorded data includes any signals and states that the model is configured to log, simulation metadata, and any errors that occurred.

## **See Also**

`rlSimulationOptions` | `train`

## **Topics**

“Train Reinforcement Learning Agents”

**Introduced in R2019a**

# train

**Package:** rl.agent

Train a reinforcement learning agent within a specified environment

## Syntax

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

## Description

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

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

## Examples

### Train a Reinforcement Learning Agent

Configure the training parameters and train a reinforcement learning agent. Typically, before training, you must configure your environment and agent. For this example, load an environment and agent that are already configured. The environment is a discrete cart-pole environment created with `rlPredefinedEnv`. The agent is a Policy Gradient (`rlPGAgent`) agent. For more information about the environment and agent used in this example, see “Train PG Agent to Balance Cart-Pole System”.

```
rng(0); % for reproducibility
load RLTrainExample.mat
env
```

```
env =  
    CartPoleDiscreteAction with properties:  
  
        Gravity: 9.8000  
        MassCart: 1  
        MassPole: 0.1000  
        Length: 0.5000  
        MaxForce: 10  
        Ts: 0.0200  
        ThetaThresholdRadians: 0.2094  
        XThreshold: 2.4000  
        RewardForNotFalling: 1  
        PenaltyForFalling: -5  
        State: [4×1 double]
```

**agent**

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

To train this agent, you must first specify training parameters using `rlTrainingOptions`. These parameters include the maximum number of episodes to train, the maximum steps per episode, and the conditions for terminating training. For this example, use a maximum of 1000 episodes and 500 steps per episode. Instruct the training to stop when the average reward over the previous five episodes reaches 500. Create a default options set and use dot notation to change some of the parameter values.

```
trainOpts = rlTrainingOptions;  
  
trainOpts.MaxEpisodes           = 1000;  
trainOpts.MaxStepsPerEpisode    = 500;  
trainOpts.StopTrainingCriteria  = "AverageReward";  
trainOpts.StopTrainingValue     = 500;  
trainOpts.ScoreAveragingWindowLength = 5;
```

During training, the `train` command can save candidate agents that give good results. Further configure the training options to save an agent when the episode reward exceeds 500. Save the agent to a folder called `savedAgents`.

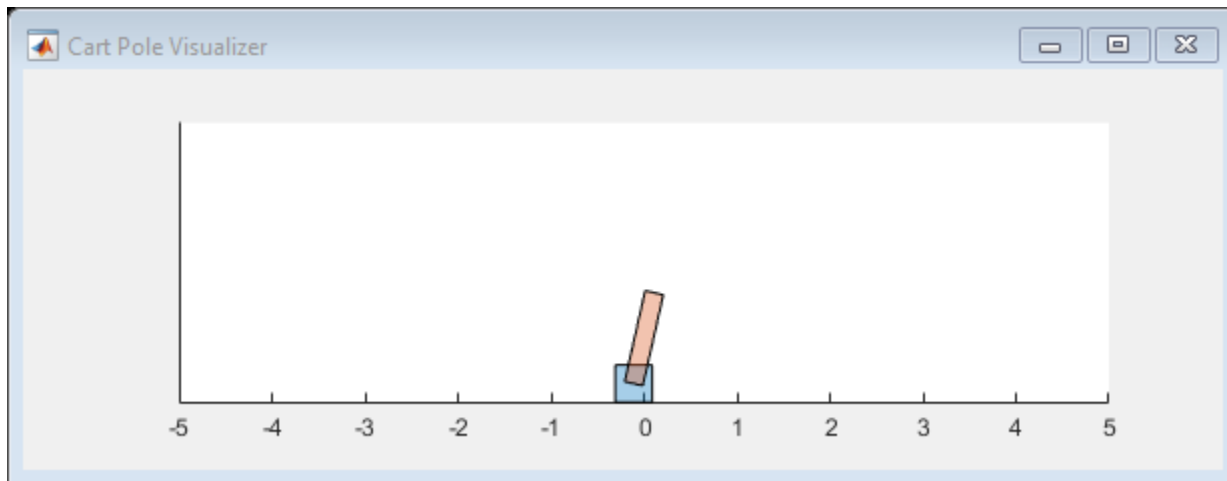
```
trainOpts.SaveAgentCriteria = "EpisodeReward";  
trainOpts.SaveAgentValue   = 500;  
trainOpts.SaveAgentDirectory = "savedAgents";
```

Finally, turn off the command-line display. Turn on the Reinforcement Learning Episode Manager so you can observe the training progress visually.

```
trainOpts.Verbose = false;  
trainOpts.Plots = "training-progress";
```

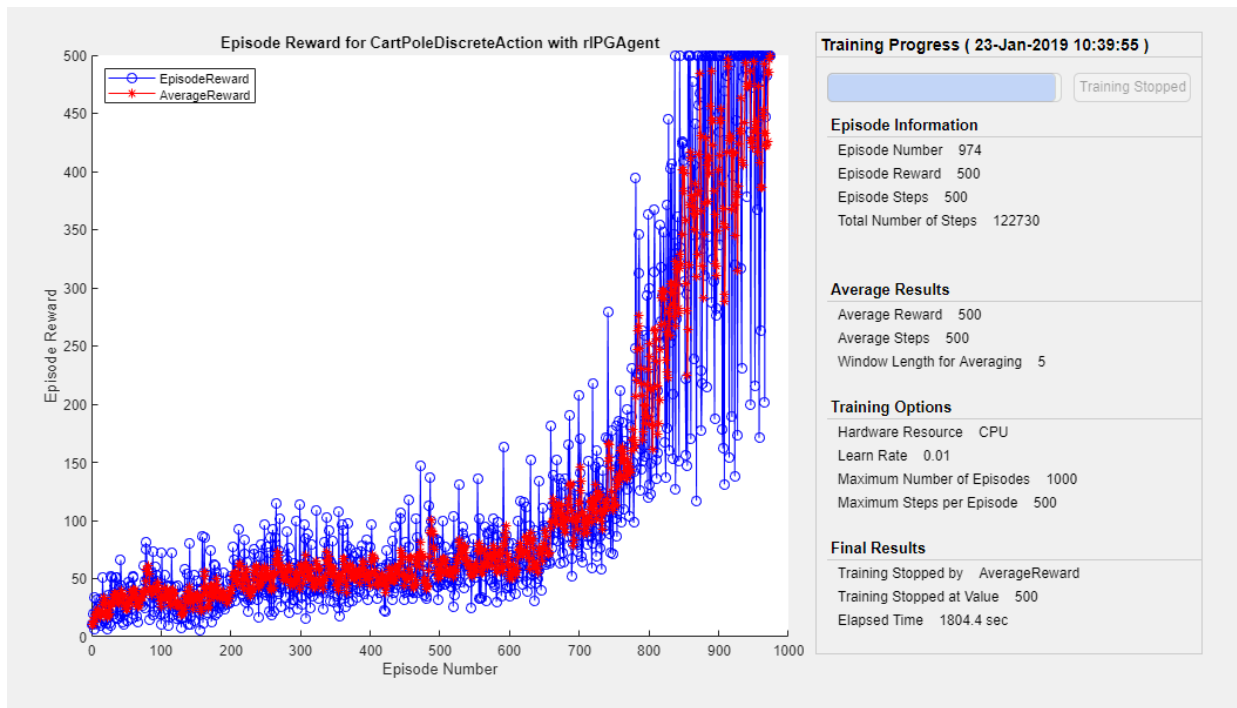
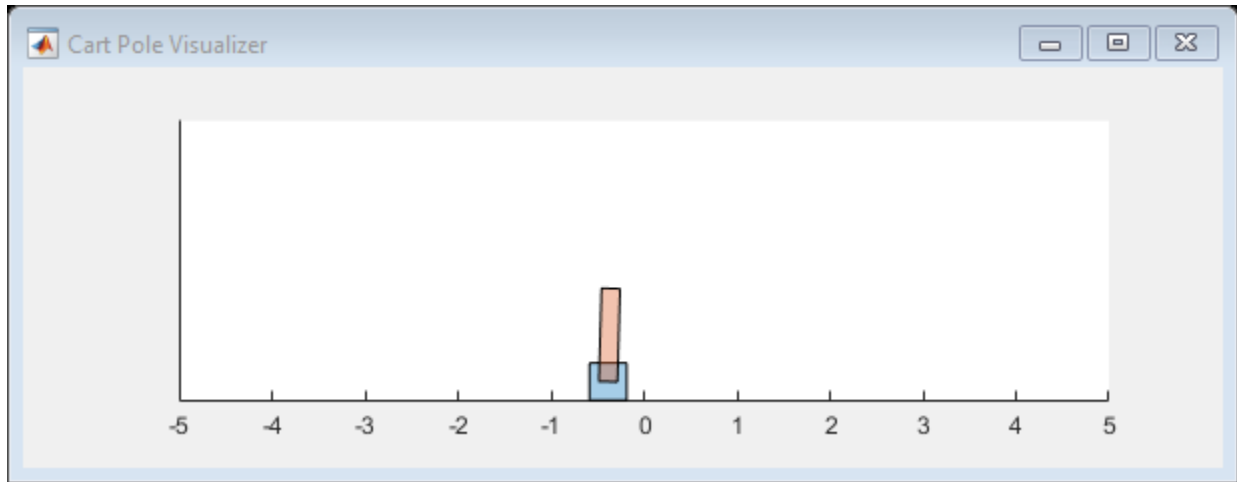
You are now ready to train the PG agent. For the predefined cart-pole environment used in this example, you can use `plot` to generate a visualization of the cart-pole system.

```
plot(env);
```



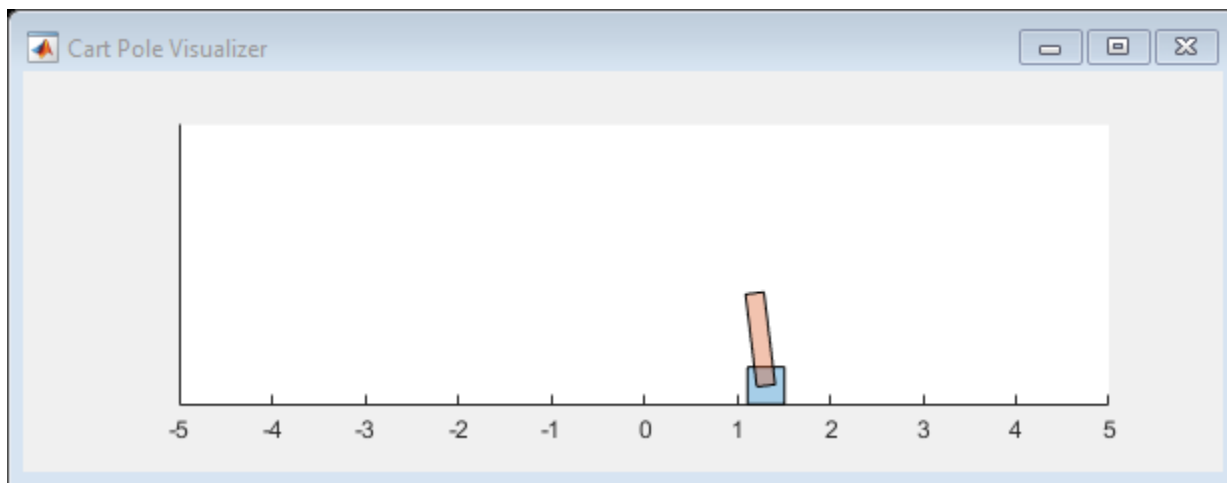
When you run this example, both this visualization and the Reinforcement Learning Episode Manager update with each training episode. Place them side by side on your screen to observe the progress, and train the agent. (This computation can take 20 minutes or more.)

```
trainingInfo = train(agent,env,trainOpts);
```



The Episode Manager shows that the training successfully reaches the termination condition of a reward of 500 averaged over the previous five episodes. At each training episode, `train` updates `agent` with the parameters learned in the previous episode. When training terminates, you can simulate the environment with the trained agent to evaluate its performance. The environment plot updates during simulation as it did during training.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```



During training, `train` saves to disk any agents that meet the condition specified with `trainOpts.SaveAgentCriteria` and `trainOpts.SaveAgentValue`. To test the performance of any of those agents, you can load the data from the data files in the folder you specified using `trainOpts.SaveAgentDirectory`, and simulate the environment with that agent.

## Input Arguments

### **agent** — Agent

reinforcement learning agent object

Agent to train, specified as a reinforcement learning agent object, such as an `rlLACAgent` or `rlDDPGAgent` object, or a custom agent. Before training, you must configure the actor

and critic representations of the agent. For more information about how to create and configure agents for reinforcement learning, see “Reinforcement Learning Agents”.

### **env — Environment**

reinforcement learning environment object

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

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

For more information about creating and configuring environments, see:

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

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

### **trainOpts — Training parameters and options**

`rlTrainingOptions` object

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

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

For details, see `rlTrainingOptions`.

## **Output Arguments**

### **trainStats — Training episode data**

structure



---

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

**EpisodeIndex — Episode numbers**

[1;2;...;N]

Episode numbers, returned as the column vector [1;2;...;N], where N is the number of episodes in the training run. This vector is useful if you want to plot the evolution of other quantities from episode to episode.

**EpisodeReward — Reward for each episode**

column vector

Reward for each episode, returned in a column vector of length N. Each entry contains the reward for the corresponding episode.

**EpisodeSteps — Number of steps in each episode**

column vector

Number of steps in each episode, returned in a column vector of length N. Each entry contains the number of steps in the corresponding episode.

**AverageReward — Average reward over the averaging window**

column vector

Average reward over the averaging window specified in `trainOpts`, returned as a column vector of length N. Each entry contains the average award computed at the end of the corresponding episode.

**TotalAgentSteps — Total number of steps**

column vector

Total number of agent steps in training, returned as a column vector of length N. Each entry contains the cumulative sum of the entries in `EpisodeSteps` up to that point.

**EpisodeQ0 — Critic estimate of long-term reward for each episode**

column vector

Critic estimate of long-term reward using the current agent and the environment initial conditions, returned as a column vector of length N. Each entry is the critic estimate ( $Q_0$ ) for the agent of the corresponding episode. This field is present only for agents that have critics, such as `rLDDPGAgent` and `rLDQNAgent`.

## **SimulationInfo** — Information collected during simulation

structure | vector of `Simulink.SimulationOutput` objects

Information collected during the simulations performed for training, returned as:

- For training in MATLAB environments, a structure containing the field `SimulationError`. This field is a column vector with one entry per episode. When the `StopOnError` option of `rlTrainingOptions` is "off", each entry contains any errors that occurred during the corresponding episode.
- For training in Simulink environments, a vector of `Simulink.SimulationOutput` objects containing simulation data recorded during the corresponding episode. Recorded data for an episode includes any signals and states that the model is configured to log, simulation metadata, and any errors that occurred during the corresponding episode.

## **Tips**

- `train` updates the agent as training progresses. To preserve the original agent parameters for later use, save the agent to a MAT-file.
- By default, calling `train` opens the Reinforcement Learning Episode Manager, which lets you visualize the progress of the training. The Episode Manager plot shows the reward for each episode, a running average reward value, and the critic estimate  $Q_0$  (for agents that have critics). The Episode Manager also displays various episode and training statistics. To turn off the Reinforcement Learning Episode Manager, set the `Plots` option of `trainOpts` to "none".
- If you use a predefined environment for which there is a visualization, you can use `plot(env)` to visualize the environment. If you call `plot(env)` before training, then the visualization updates during training to allow you to visualize the progress of each episode. (For custom environments, you must implement your own `plot` method.)
- Training terminates when the conditions specified in `trainOpts` are satisfied. To terminate training in progress, in the Reinforcement Learning Episode Manager, click **Stop Training**. Because `train` updates the agent at each episode, you can resume training by calling `train(agent,env,trainOpts)` again, without losing the trained parameters learned during the first call to `train`.
- During training, you can save candidate agents that meet conditions you specify with `trainOpts`. For instance, you can save any agent whose episode reward exceeds a certain value, even if the overall condition for terminating training is not yet satisfied. `train` stores saved agents in a MAT-file in the folder you specify with `trainOpts`.

Saved agents can be useful, for instance, to allow you to test candidate agents generated during a long-running training process. For details about saving criteria and saving location, see `rllibTrainingOptions`.

## Algorithms

In general, `train` performs the following iterative steps:

- 1** Initialize agent.
- 2** For each episode:
  - a** Reset the environment.
  - b** Get the initial observation  $s_0$  from the environment.
  - c** Compute the initial action  $a_0 = \mu(s_0)$ .
  - d** Set the current action to the initial action ( $a \leftarrow a_0$ ) and set the current observation to the initial observation ( $s \leftarrow s_0$ ).
  - e** While the episode is not finished or terminated:
    - i** Step the environment with action  $a$  to obtain the next observation  $s'$  and the reward  $r$ .
    - ii** Learn from the experience set  $(s, a, r, s')$ .
    - iii** Compute the next action  $a' = \mu(s')$ .
    - iv** Update the current action with the next action ( $a \leftarrow a'$ ) and update the current observation with the next observation ( $s \leftarrow s'$ ).
    - v** Break if the episode termination conditions defined in the environment are met.
- 3** If the training termination condition defined by `trainOpts` is met, terminate training. Otherwise, begin the next episode.

The specifics of how `train` performs these computations depends on your configuration of the agent and environment. For instance, resetting the environment at the start of each episode can include randomizing initial state values, if you configure your environment to do so.

## Extended Capabilities

### Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To train in parallel, set the `Parallelization` and `ParallelizationOptions` options in the option set `trainOpts`. For more information, see `rlTrainingOptions`.

### See Also

`rlTrainingOptions` | `sim`

### Topics

“Train Reinforcement Learning Agents”

**Introduced in R2019a**

# validateEnvironment

**Package:** rl.env

Validate custom reinforcement learning environment

## Syntax

```
validateEnvironment(env)
```

## Description

`validateEnvironment(env)` validates a reinforcement learning environment. This function is useful when:

- You are using a custom environment for which you supplied your own step and reset functions, such as an environment created using `rlCreateEnvTemplate`.
- You are using an environment you created from a Simulink model using `rlSimulinkEnv`.

`validateEnvironment` resets the environment, generates an initial observation and action, and simulates the environment for one or two steps (see “Algorithms” on page 1-155). If there are no errors during these operations, validation is successful, and `validateEnvironment` returns no result. If errors occur, these errors appear in the MATLAB command window. Use the errors to determine what to change in your observation specification, action specification, custom functions, or Simulink model.

## Examples

### Validate Simulink Environment

Create and validate an environment for the `rlwatertank` model, which represents a control system containing a reinforcement learning agent. (For details about this model, see “Create Simulink Environment and Train Agent”.)

Open the model.

```
open_system('rlwatertank')
```

Create observation and action specifications for the environment.

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

actInfo = rlNumericSpec([1 1]);
actInfo.Name = 'flow';
numActions = numel(actInfo);
```

Create an environment from the model. Then, use `validateEnvironment` to check whether the model is configured correctly.

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

```
Error using rl.env.SimulinkEnvWithAgent/validateEnvironment (line 187)
Simulink environment validation requires an agent in the MATLAB base workspace
or in a data dictionary linked to the model. Specify the agent in the Simulink model.
```

`validateEnvironment` attempts to compile the model, initialize the environment and the agent, and simulate the model. In this case, the RL Agent block is configured to use an agent called `agent`, but no such variable exists in the MATLAB workspace. Thus, the function returns an error indicating the problem.

Create an appropriate agent for this system using the commands detailed in the “Create Simulink Environment and Train Agent”. Then, run `validateEnvironment` again.

```
validateEnvironment(env)
```

This time, the initialization and two-step simulation runs without error. Therefore, `validateEnvironment` returns nothing, and validation is successful.

`validateEnvironment` leaves the model itself unchanged, restoring all model parameters to their original values.

## Input Arguments

### **env** — Environment to validate

environment object

Environment to validate, specified as a reinforcement learning environment object, such as:

- A custom MATLAB environment you create with `rlCreateEnvTemplate`. In this case, `validateEnvironment` checks that the observations and actions generated during simulation of the environment are consistent in size, data type, and value range with the observation specification and action specification. It also checks that your custom `step` and `reset` functions run without error. (When you create a custom environment using `rlFunctionEnv`, the software runs `validateEnvironment` automatically.)
- A custom Simulink environment you create using `rlSimulinkEnv`. If you use a Simulink environment, you must also have an agent defined and associated with the RL Agent block in the model. For a Simulink model, `validateEnvironment` checks that the model compiles and runs without error. The function does not dirty your model.

For more information about creating and configuring environments, see:

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

## Algorithms

`validateEnvironment` works by running a brief simulation of the environment and making sure that the generated signals match the observation and action specifications you provided when you created the environment.

## MATLAB Environments

For MATLAB environments, validation includes the following steps.

- 1 Reset the environment using the `reset` function associated with the environment.
- 2 Obtain the first observation and check whether it is consistent with the dimension, data type, and range of values in the observation specification.
- 3 Generate a test action based on the dimension, data type, and range of values in the action specification.
- 4 Simulate the environment for one step using the generated action and the `step` function associated with the environment.
- 5 Obtain the new observation signal and check whether it is consistent with the dimension, data type, and range of values in the observation specification.

If any of these operations generates an error, `validateEnvironment` returns the error. If `validateEnvironment` returns no result, then validation is successful.

### Simulink Environments

For Simulink environments, validation includes the following steps.

- 1 Reset the environment.
- 2 Simulate the model for two time steps.

If any of these operations generates an error, `validateEnvironment` returns the error. If `validateEnvironment` returns no result, then validation is successful.

`validateEnvironment` performs these steps without dirtying the model, and leaves all model parameters in the state they were in when you called the function.

### See Also

`rlCreateEnvTemplate` | `rlFunctionEnv` | `rlSimulinkEnv`

### Topics

“Create Simulink Environment and Train Agent”

“Create Custom MATLAB Environment from Template”

**Introduced in R2019a**



# Objects – Alphabetical List

---

# quadraticLayer

Quadratic layer for actor or critic network

## Description

QuadraticLayer is a deep neural network layer that takes an input vector and outputs a vector of quadratic monomials constructed from the input elements. For example, consider an input vector  $U = [u_1 \ u_2 \ u_3]$ . For this input, a quadratic layer gives the output  $Y = [u_1*u_1 \ u_1*u_2 \ u_2*u_2 \ u_1*u_3 \ u_2*u_3 \ u_3*u_3]$ .

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

## Creation

## Syntax

```
qLayer = quadraticLayer
qLayer = quadraticLayer(Name,Value)
```

## Description

`qLayer = quadraticLayer` creates a quadratic layer with default property values.

`qLayer = quadraticLayer(Name,Value)` sets properties using `Name,Value` pairs. For example, `quadraticLayer('Name','quadlayer')` creates a quadratic layer and assigns the name 'quadlayer'.

## Properties

### Name — Name of layer

'quadratic' (default) | character vector

Name of layer, specified as a character vector. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with this layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

### Description — Description of layer

'quadratic layer' (default) | character vector

This property is read-only.

Description of layer, specified when you create the quadratic layer and stored as a character vector. When you create the quadratic layer, you can use this property to give it a description that helps you identify its purpose.

## Examples

### Create Quadratic Layer

Create a quadratic layer that converts an input vector  $U$  into a vector of quadratic monomials constructed from binary combinations of the elements of  $U$ .

```
qLayer = quadraticLayer
qLayer =
  QuadraticLayer with properties:
    Name: 'quadratic'
  Show all properties
```

Confirm that the layer produces the expected output. For instance, for  $U = [u_1 \ u_2 \ u_3]$ , the expected output is  $[u_1^2 \ u_1 u_2 \ u_2^2 \ u_1 u_3 \ u_2 u_3 \ u_3^2]$ .

```
predict(qLayer,[1 2 3])
```

```
ans = 1×6
```

```
    1    2    4    3    6    9
```

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

### See Also

`scalingLayer`

### Topics

“Train DDPG Agent to Control Double Integrator System”

“Create Policy and Value Function Representations”

**Introduced in R2019a**

# rlFiniteSetSpec

Create discrete action or observation data specifications for reinforcement learning environments

## Description

Use `rlFiniteSetSpec` to create an `rlFiniteSetSpec` object that defines a finite set of actions or observations.

## Creation

## Syntax

```
spec = rlFiniteSetSpec(elements)
```

## Description

`spec = rlFiniteSetSpec(elements)` creates data specification with a discrete set defined by `elements`.

## Input Arguments

### **elements** — Set of valid values

numeric vector

Set of valid values, specified as a numeric vector.

This is equivalent to the `Elements` property.

## Properties

### **Elements — Set of valid values**

numeric vector

Set of valid values, specified as a numeric vector.

### **Name — Name of the rlFiniteSetSpec object**

string (default)

Name of the rlFiniteSetSpec object, specified as a string.

### **Description — Description of the rlFiniteSetSpec object**

string (default)

Description of the rlFiniteSetSpec object, specified as a string.

### **Dimension — Size of each element**

vector (default)

Size of each element, specified as a vector.

### **DataType — Information about the type of data**

string (default)

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

## Object Functions

rlSimulinkEnv	Create a reinforcement learning environment using a dynamic model implemented in Simulink
rlFunctionEnv	Specify custom reinforcement learning environment dynamics using functions
rlRepresentation	Model representation for reinforcement learning agents

## Examples

## Reinforcement Learning Environment for Simulink models

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

Open the model.

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

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

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

```
obsInfo =  
    rlNumericSpec with properties:  
  
        LowerLimit: -Inf  
        UpperLimit: Inf  
        Name: [0x0 string]  
        Description: [0x0 string]  
        Dimension: [3 1]  
        DataType: "double"
```

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

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

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

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

```
env = rlSimulinkEnv mdl, agentBlk, obsInfo, actInfo)
```

```
env =  
    SimulinkEnvWithAgent with properties:  
  
        Model: "rlSimplePendulumModel"  
        AgentBlock: "rlSimplePendulumModel/RL Agent"  
        ResetFcn: []  
        UseFastRestart: 'on'
```

You can also include a reset function using dot notation. For this example, consider randomly initializing `theta0` in the model workspace.

```
env.ResetFcn = @(in) setVariable(in, 'theta0', randn, 'Workspace', mdl)
```

```
env =  
    SimulinkEnvWithAgent with properties:  
  
        Model: "rlSimplePendulumModel"  
        AgentBlock: "rlSimplePendulumModel/RL Agent"  
        ResetFcn: @(in)setVariable(in, 'theta0', randn, 'Workspace', mdl)  
        UseFastRestart: 'on'
```

## See Also

[getActionInfo](#) | [getObservationInfo](#) | [rlFunctionEnv](#) | [rlNumericSpec](#) | [rlRepresentation](#) | [rlSimulinkEnv](#)

**Introduced in R2019a**



# rlFunctionEnv

Specify custom reinforcement learning environment dynamics using functions

## Description

Use `rlFunctionEnv` to define a custom reinforcement learning environment. You provide MATLAB functions that define the step and reset behavior for the environment. This object is useful when you want to customize your environment beyond the predefined environments available with `rlPredefinedEnv`.

## Creation

## Syntax

```
env = rlFunctionEnv(obsInfo,actInfo,stepfcn,resetfcn)
```

## Description

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

## Input Arguments

### **obsInfo** — Observation specification

spec object

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

### **actInfo — Action specification**

spec object

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

### **stepfcn — Step behavior for the environment**

function | function handle | anonymous function

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

### **resetfcn — Reset behavior for the environment**

function | function handle | anonymous function

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

## Properties

### **StepFcn — Step behavior for the environment**

function | function handle | anonymous function

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

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

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

Thus, the step function computes the values of the observation and reward for the given action in the environment. The required input and output arguments are:

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

- **Reward** — Reward for the current step, returned as a scalar value.
- **IsDone** — Logical value indicating whether to end the simulation episode. The step function that you define can include logic to decide whether to end the simulation based on the observation, reward, or any other values.
- **LoggedSignals** — Any data that you want to pass from one step to the next, specified as a structure.

To use additional input arguments beyond this required set, specify `StepFcn` using a function handle or an anonymous function. For an example showing multiple ways to define a step function, see “Create MATLAB Environment using Custom Functions”.

### **ResetFcn — Reset behavior for the environment**

function | function handle | anonymous function

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

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

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

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

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

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

To use input arguments with your reset function, specify `ResetFcn` using a function handle or an anonymous function. For an example showing multiple ways to define a reset function, see “Create MATLAB Environment using Custom Functions”.

### **LoggedSignals — Information to pass to next step**

structure

Information to pass to the next step, specified as a structure. When you create the environment, whatever you define as the `LoggedSignals` output of `resetfcn` initializes this property. When a step occurs, the software populates this property with data to pass to the next step, as defined in `stepfcn`.

## Object Functions

<code>getActionInfo</code>	Obtain action data specifications from reinforcement learning environment or agent
<code>getObservationInfo</code>	Obtain observation data specifications from reinforcement learning environment or agent
<code>sim</code>	Simulate a trained reinforcement learning agent within a specified environment
<code>validateEnvironment</code>	Validate custom reinforcement learning environment

## Examples

### Create Custom MATLAB Environment

Create a reinforcement learning environment by supplying custom dynamic functions in MATLAB®. Using `rlFunctionEnv`, you can create a MATLAB reinforcement learning environment from an observation specification, action specification, and `step` and `reset` functions that you define.

For this example, create an environment that represents a system for balancing a cart on a pole. The observations from the environment are the cart position, cart velocity, pendulum angle, and pendulum angle derivative. (For additional details about this environment, see “Create MATLAB Environment using Custom Functions”.) Create an observation specification for those signals.

```
oinfo = rlNumericSpec([4 1]);  
oinfo.Name = 'CartPole States';  
oinfo.Description = 'x, dx, theta, dtheta';
```

The environment has a discrete action space where the agent can apply one of two possible force values to the cart, -10 N or 10 N. Create the action specification for those actions.

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

Next, specify the custom `step` and `reset` functions. For this example, use the supplied functions `myResetFunction.m` and `myStepFunction.m`. For details about these functions and how they are constructed, see “Create MATLAB Environment using Custom Functions”.

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

```
addpath(fullfile(matlabroot,'examples','rl','main')) % Make sure the functions are c  
env = rlFunctionEnv(oinfo,ActionInfo,'myStepFunction','myResetFunction');
```

You can create agents for `env` and train them within the environment as you would for any other reinforcement learning environment.

As an alternative to using function names, you can specify the functions as function handles. For more details and an example, see “Create MATLAB Environment using Custom Functions”.

## See Also

[rlCreateEnvTemplate](#) | [rlPredefinedEnv](#)

## Topics

“Create MATLAB Environment using Custom Functions”

**Introduced in R2019a**

# rLMDPEnv

Create Markov decision process environment for reinforcement learning

## Description

A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of the decision maker. MDPs are useful for studying optimization problems solved using reinforcement learning. Use `rLMDPEnv` to create a MATLAB based Markov decision process environment object `rLMDPEnv` for reinforcement learning.

## Creation

## Syntax

```
env = rLMDPEnv(MDP)
```

## Description

`env = rLMDPEnv(MDP)` creates a reinforcement learning environment `env` with the specified MDP model.

## Input Arguments

### **MDP — Markov decision process model**

`GridWorld` object

Markov decision process model, specified as a `GridWorld` object.

For more information on creating `GridWorld` objects, see `createGridWorld`.

## Properties

### Model — Markov decision process model

GridWorld object

Markov decision process model, specified as a GridWorld object.

### ResetFcn — Reset function

function handle

Reset function, specified as a function handle.

## Object Functions

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

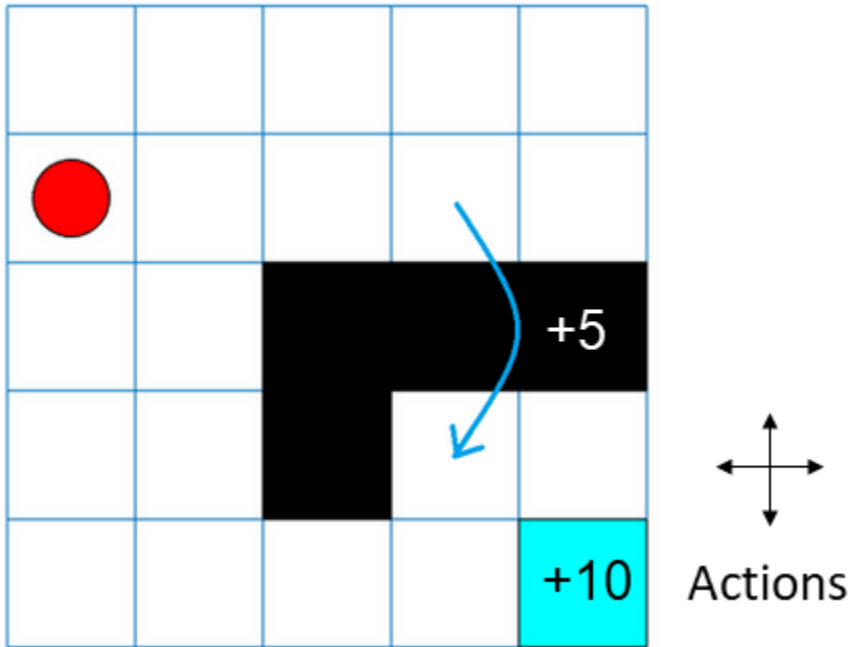
## Examples

### Create Grid World Environment

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

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

6 All other actions result in -1 reward.



First, create a GridWorld object using the createGridWorld function.

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

```
GW =  
  GridWorld with properties:  
  
    GridSize: [5 5]  
    CurrentState: "[1,1]"  
    States: [25x1 string]  
    Actions: [4x1 string]  
    T: [25x25x4 double]  
    R: [25x25x4 double]  
    ObstacleStates: [0x1 string]  
    TerminalStates: [0x1 string]
```

Now, set the initial, terminal and obstacle states.



```

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

```

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

```

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

```

Next, define the rewards in the reward transition matrix.

```

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

```

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

```

env = rLMDPEnv(GW)

env =
  rLMDPEnv with properties:
    Model: [1x1 rl.env.GridWorld]
    ResetFcn: []

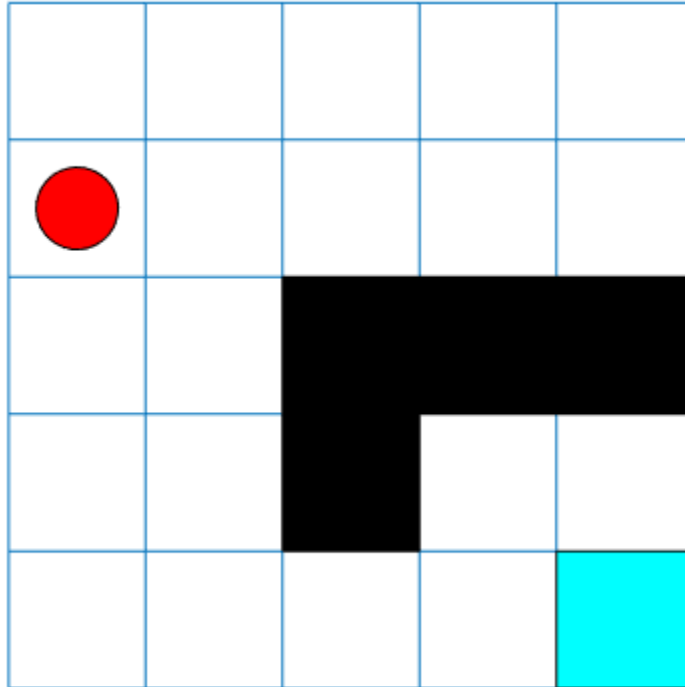
```

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

```

plot(env)

```



### See Also

`createGridWorld` | `rlPredefinedEnv`

### Topics

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

**Introduced in R2019a**

# rlNumericSpec

Create continuous action or observation data specifications for reinforcement learning environments

## Description

Use `rlNumericSpec` to create an `rlNumericSpec` object that defines a finite set of actions or observations.

## Creation

## Syntax

```
spec = rlNumericSpec(dimension)
spec = rlNumericSpec(dimension,Name,Value)
```

## Description

`spec = rlNumericSpec(dimension)` creates data specification with the shape defined by the vector `dimension`.

`spec = rlNumericSpec(dimension,Name,Value)` sets “Properties” on page 2-20 using name-value pair arguments.

## Input Arguments

### **dimension** — Dimension of the data space

numeric vector

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

This is equivalent to the `Dimension` property.

### Properties

#### **LowerLimit — Lower limit of the data space**

'-inf' (default) | scalar | matrix

Lower limit of the data space, specified as a scalar or matrix of the same size as the data space. When `LowerLimit` is specified as a scalar, `rLNumericSpec` applies it to all entries in the data space.

#### **UpperLimit — Upper limit of the data space**

'inf' (default) | scalar | matrix

Upper limit of the data space, specified as a scalar or matrix of the same size as the data space. When `UpperLimit` is specified as a scalar, `rLNumericSpec` applies it to all entries in the data space.

#### **Name — Name of the rLNumericSpec object**

string (default)

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

#### **Description — Description of the rLNumericSpec object**

string (default)

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

#### **Dimension — Dimension of the data space**

numeric vector (default)

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

#### **DataType — Information about the type of data**

string (default)

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

### Object Functions

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

rlFunctionEnv     Specify custom reinforcement learning environment dynamics using functions  
 rlRepresentation     Model representation for reinforcement learning agents

## Examples

### Reinforcement Learning Environment for Simulink models

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

Open the model.

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

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

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

obsInfo =
    rlNumericSpec with properties:
        LowerLimit: -Inf
        UpperLimit: Inf
        Name: [0x0 string]
        Description: [0x0 string]
        Dimension: [3 1]
        DataType: "double"

actInfo = rlFiniteSetSpec([2 1])

actInfo =
    rlFiniteSetSpec with properties:
        Elements: [2 1]
        Name: [0x0 string]
```

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

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

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

```
env = rlSimulinkEnv mdl, agentBlk, obsInfo, actInfo)
```

```
env =
  SimulinkEnvWithAgent with properties:
      Model: "rlSimplePendulumModel"
      AgentBlock: "rlSimplePendulumModel/RL Agent"
      ResetFcn: []
      UseFastRestart: 'on'
```

You can also include a reset function using dot notation. For this example, consider randomly initializing `theta0` in the model workspace.

```
env.ResetFcn = @(in) setVariable(in, 'theta0', randn, 'Workspace', mdl)
```

```
env =
  SimulinkEnvWithAgent with properties:
      Model: "rlSimplePendulumModel"
      AgentBlock: "rlSimplePendulumModel/RL Agent"
      ResetFcn: @(in) setVariable(in, 'theta0', randn, 'Workspace', mdl)
      UseFastRestart: 'on'
```

## See Also

[getActionInfo](#) | [getObservationInfo](#) | [rlFiniteSetSpec](#) | [rlFunctionEnv](#) | [rlRepresentation](#) | [rlSimulinkEnv](#)

## Topics

“Train DDPG Agent for Adaptive Cruise Control”

**Introduced in R2019a**

# rlTable

Value table or Q table

## Description

You can create value tables and Q tables to represent critic networks for reinforcement learning. Value tables store rewards for a finite set of observations. Q tables store rewards for corresponding finite observation-action pairs.

To create a value function representation using an `rlTable` object, use the `rlRepresentation` function.

## Creation

## Syntax

```
T = rlTable(obsinfo)
T = rlTable(obsinfo,actinfo)
```

## Description

`T = rlTable(obsinfo)` creates a value table for the given discrete observations.

`T = rlTable(obsinfo,actinfo)` creates a Q table for the given discrete observations and actions.

## Input Arguments

### **obsinfo** — Observation specification

`rlFiniteSetSpec` object

Observation specification, specified as an `rlFiniteSetSpec` object.



**actinfo — Action specification**`rLFiniteSetSpec` object

Action specification, specified as an `rLFiniteSetSpec` object.

## Properties

**Table — Reward table**

array

Reward table, returned as an array. When `Table` is a:

- Value table, it contains  $N_O$  rows, where  $N_O$  is the number of finite observation values.
- Q table, it contains  $N_O$  rows and  $N_A$  columns, where  $N_A$  is the number of possible finite actions.

## Object Functions

`rLRepresentation` Model representation for reinforcement learning agents

## See Also

### Topics

“Create Policy and Value Function Representations”

**Introduced in R2019a**

# scalingLayer

Scaling layer for actor or critic network

## Description

`ScalingLayer` is a deep neural network layer that linearly scales and biases an input array  $U$ , giving an output  $Y = \text{Scale} * U + \text{Bias}$ . You can incorporate this layer into the deep neural networks you define for actors or critics in reinforcement learning agents. This layer is useful for scaling and shifting the outputs of nonlinear layers, such as `tanhLayer` and `sigmoid`.

For instance, `tanhLayer` gives bounded output that falls between -1 and 1. If your actor network output has different bounds (as defined in the actor specification), you can include a `ScalingLayer` as an output to scale and shift the actor network output appropriately.

Scaling layer parameters are not learnable.

## Creation

## Syntax

```
sLayer = scalingLayer  
sLayer = scalingLayer(Name, Value)
```

## Description

`sLayer = scalingLayer` creates a scaling layer with default property values.

`sLayer = scalingLayer(Name, Value)` sets properties using `Name, Value` pairs. For example, `scalingLayer('Scale', 0.5)` creates a scaling layer that scales its input by 0.5. Enclose each property name in quotes.

## Properties

### Name — Name of layer

'scaling' (default) | character vector

Name of layer, specified as a character vector. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with this layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

### Description — Description of layer

'Scaling layer' (default) | character vector

This property is read-only.

Description of layer, specified when you create the scaling layer and stored as a character vector. When you create the scaling layer, you can use this property to give it a description that helps you identify its purpose.

### Scale — Element-wise scale on input

1 (default) | numeric scalar

Element-wise scale on the input to the scaling layer, specified as a numeric scalar. The scaling layer takes an input  $U$  and generates the output  $Y = \text{Scale} \cdot U + \text{Bias}$ .

### Bias — Element-wise bias on input

0 (default) | numeric scalar

Element-wise bias on the input to the scaling layer, specified as a numeric scalar. The scaling layer takes an input  $U$  and generates the output  $Y = \text{Scale} \cdot U + \text{Bias}$ .

## Examples

### Create Scaling Layer

Create a scaling layer that converts an input array  $U$  to the output array  $Y = 0.1 \cdot U - 0.4$ .

```
sLayer = scalingLayer('Scale',0.1,'Bias',-0.4)
```

```
sLayer =  
  ScalingLayer with properties:  
  
    Name: 'scaling'  
    Scale: 0.1000  
    Bias: -0.4000  
  
  Show all properties
```

Confirm that the scaling layer scales and offsets an input array as expected.

```
predict(sLayer, [10, 20, 30])  
  
ans = 1×3  
  
    0.6000    1.6000    2.6000
```

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

## See Also

`quadraticLayer`

## Topics

“Train DDPG Agent to Swing Up and Balance Pendulum”

“Create Policy and Value Function Representations”

**Introduced in R2019a**

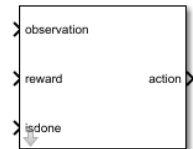
# Blocks — Alphabetical List

---

## RL Agent

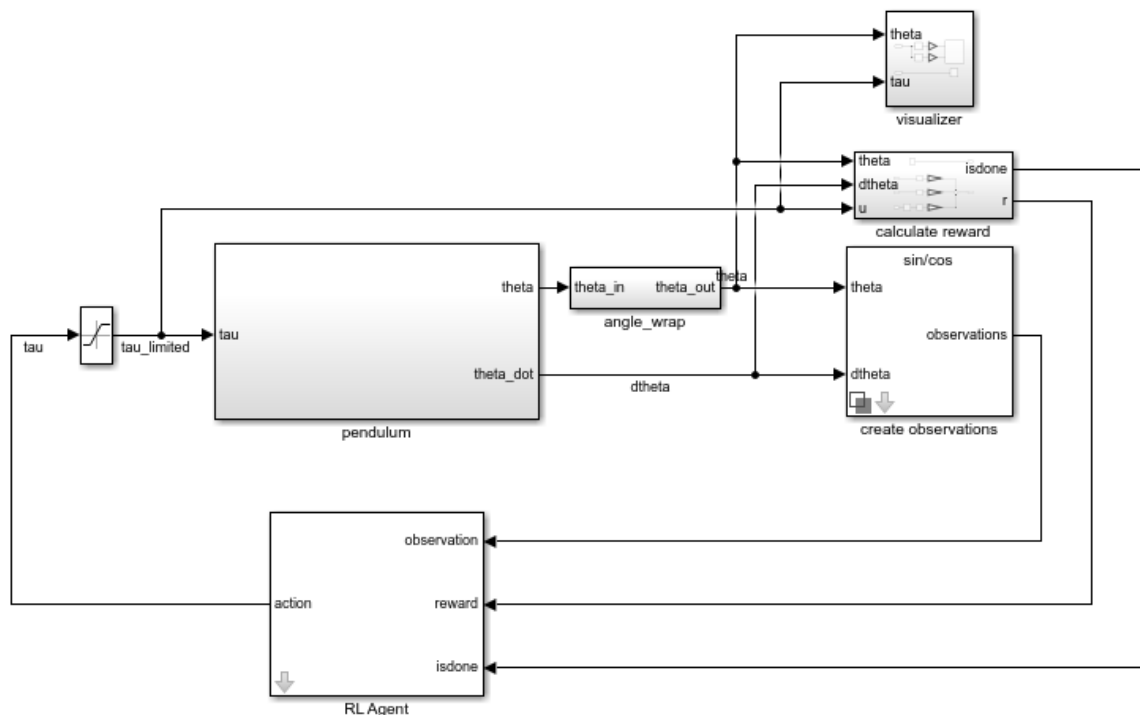
Reinforcement learning agent

**Library:** Reinforcement Learning Toolbox



## Description

Use the RL Agent block to simulate and train a reinforcement learning agent in Simulink. You associate the block with an agent stored in the MATLAB workspace or a data dictionary as an agent object such as an `rLACAgent` or `rLDDPGAgent` object. You connect the block so that it receives an observation and a computed reward. For instance, consider the following block diagram of the `SimplePendulumModel` model.



The **observation** input port of the RL Agent block receives a signal that is derived from the instantaneous angle and angular velocity of the pendulum. The reward port receives a reward calculated from the same two values and the applied action. You configure the observations and reward computations that are appropriate to your system.

The block uses the agent to generate an action based on the observation and reward you provide. Connect the **action** output port to the appropriate input for your system. For instance, in the **SimplePendulumModel**, the **action** port is a torque applied to the pendulum system. For more information about this model, see “Train DQN Agent to Swing Up and Balance Pendulum”.

To train a reinforcement learning agent in Simulink, you generate an environment from the Simulink model. You then create and configure the agent for training against that environment. For more information, see “Create Simulink Environments for

Reinforcement Learning”. When you call `train` using the environment, `train` simulates the model and updates the agent associated with the block.

## Ports

### Input

#### **observation — Environment observations**

scalar | vector | nonvirtual bus

This port receives observation signals from the environment. Observation signals represent measurements or other instantaneous system data. If you have multiple observations, you can use a Mux block to combine them into a vector signal. To use a nonvirtual bus signal, use `bus2RLSpec`.

#### **reward — Reward from environment**

scalar

This port receives the reward signal, which you compute based on the observation data. The reward signal is used during agent training to maximize the expectation of the long-term reward.

#### **isdone — Flag to terminate episode simulation**

logical

Use this signal to specify conditions under which to terminate a training episode. You must configure logic appropriate to your system to determine the conditions for episode termination. One application is to terminate an episode that is clearly going well or going poorly. For instance, you can terminate an episode if the agent reaches its goal or goes irrecoverably far from its goal.

### Output

#### **action — Agent action**

scalar | vector | nonvirtual bus

Action computed by the agent based on the observation and reward inputs. Connect this port to the inputs of your system. To use a nonvirtual bus signal, use `bus2RLSpec`.



**cumulative\_reward — Total reward**

scalar | vector

Cumulative sum of the reward signal during simulation. Observe or log this signal to track how the cumulative reward evolves over time.

**Dependencies**

To enable this port, select the **Provide cumulative reward signal** parameter.

## Parameters

**Agent object — Agent to train**

agent (default) | agent object

Enter the name of an agent object stored in the MATLAB workspace or a data dictionary, such as an `rlACAgent` or `rlDDPGAgent` object. For information about agent objects, see “Reinforcement Learning Agents”.

**Provide cumulative reward signal — Add cumulative reward output port**

off (default) | on

Enable the `cumulative_reward` block output by selecting this parameter.

## See Also

bus2RLSpec | createIntegratedEnv

## Topics

“Create Simulink Environments for Reinforcement Learning”

“Create Simulink Environment and Train Agent”

**Introduced in R2019a**

