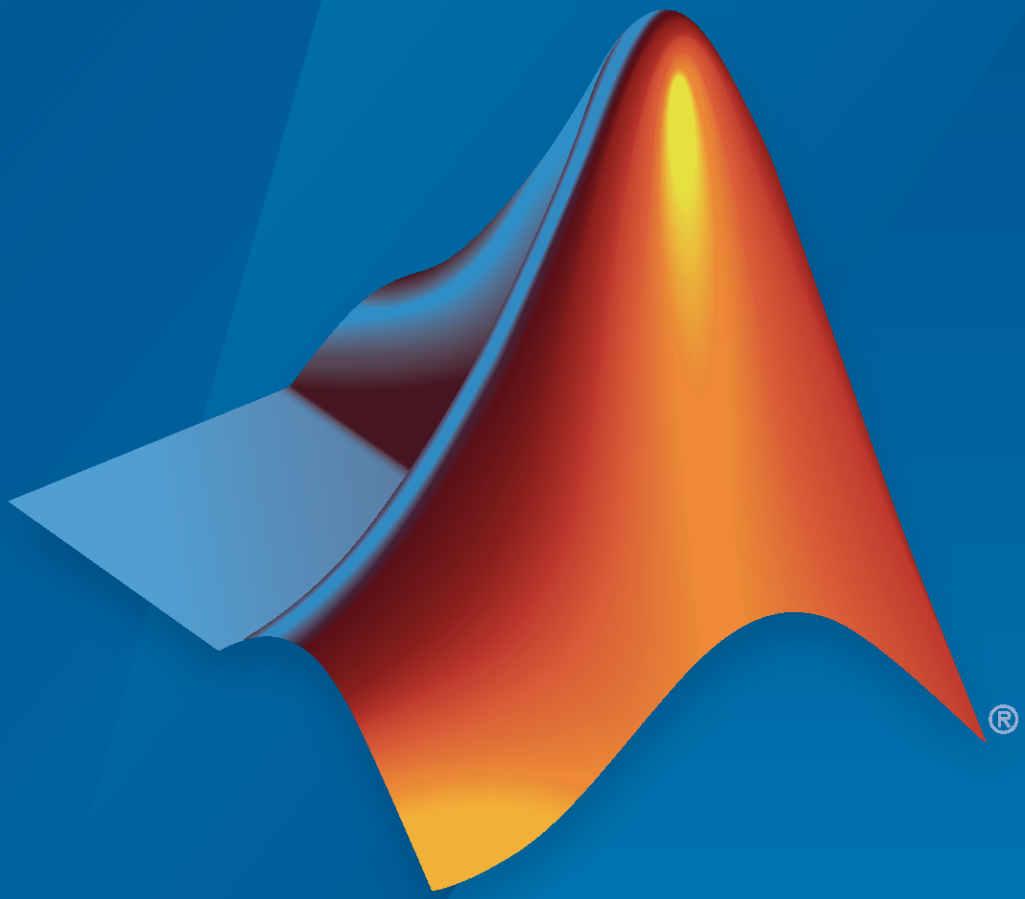
Reinforcement Learning Toolbox™ Release Notes



MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Reinforcement Learning Toolbox™ Release Notes

Contents

R2020b

R2020a

R2019b

R2019a

R2020b

Multi-Agent Reinforcement Learning: Train multiple agents in a Simulink environment

You can now train and deploy multiple agents that work in the same Simulink® environment. You can visualize the training progress of all the agents using the Episode Manager.

Create a multi-agent environment by supplying to `rlSimulinkEnv` an array of strings containing the paths of the agents, and cell arrays defining the observation and action specifications of the agent blocks.

For examples on training multiple agents, see “Train Multiple Agents to Perform Collaborative Task”, “Train Multiple Agents for Area Coverage”, and “Train Multiple Agents for Path Following Control”.

Soft Actor-Critic Agent: Train sample-efficient policies for environments with continuous-action spaces using increased exploration

You can now create soft actor-critic (SAC) agents. SAC is an improved version of DDPG that generates stochastic policies for environments with a continuous action space. It tries to maximize the entropy of the policy in addition to the cumulative long-term reward, thereby encouraging exploration.

You can create a SAC agent using the `rlSACAgent` function. You can also create a SAC-specific options object with the `rlSACAgentOptions` function.

Default Agents: Avoid manually formulating policies by creating agents with default neural network structures

You can now create a default agent based only on the observation and action specifications of a given environment. Previously, creating an agent required creating approximators for the agent actor and critic, using these approximators to create actor and critic representations, and then using these representations to create the agent.

Default agents are available for DQN, DDPG, TD3, PPO, PG, AC, and SAC agents. For each agent, you can call the agent creation function, passing in the observation and action specifications from the environment. The function creates the required actor and critic representations using deep neural network approximators.

For example, `agent = rlTD3Agent(obsInfo,actInfo)` creates a default TD3 agent using a deterministic actor network and two Q-value critic networks.

You can specify initialization options (such as the number of hidden units for each layer, or whether to use a recurrent neural network) for the default representations using an `rlAgentInitializationOptions` object.

After creating a default agent, you can then access its properties and change its actor and critic representations.

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

getModel and setModel Functions: Access computational model used by actor and critic representations

You can now access the computational model used by the actor and critic representations in a reinforcement learning agent using the following new functions.

- `getModel` — Obtain the computational model from an actor or critic representation.
- `setModel` — Set the computational model in an actor or critic representation.

Using these functions, you can modify the computational in a representation object without recreating the representation.

New Examples: Create a custom agent, use TD3 to tune a PI controller, and train agents for automatic parking and motor control

This release includes the following new reference examples.

- “Create Agent for Custom Reinforcement Learning Algorithm” — Create a custom agent for your own custom reinforcement learning algorithm.
- “Tune PI Controller using Reinforcement Learning” — Tune a PI controller using the twin-delayed deep deterministic policy gradient (TD3) reinforcement learning algorithm.
- “Train PPO Agent for Automatic Parking Valet” — Train a PPO agent to automatically search for a parking space and park.
- “Train DDPG Agent for PMSM Control” — Train a DDPG agent to control the speed of a permanent magnet synchronous motor.

Functionality being removed or changed

Default value of NumStepsToLookAhead option for AC agents is now 32
Behavior change

For AC agents, the default value of the `NumStepsToLookAhead` option is now 32.

To use the previous default value instead, create an `rlACAgentOptions` object and set the option value to `1`.

```
opt = rlACAgentOptions;
opt.NumStepsToLookAhead = 1;
```

R2020a

Version: 1.2

New Features

Bug Fixes

Compatibility Considerations

New Representation Objects: Create actors and critics with improved ease of use and flexibility

You can represent actor and critic functions using four new representation objects. These objects improve ease of use, readability, and flexibility.

- `rlValueRepresentation` — State value critic, computed based on observations from the environment.
- `rlQValueRepresentation` — State-action value critic, computed based on both actions and observations from the environment.
- `rlDeterministicActorRepresentation` — Actor with deterministic actions, based on observations from the environment.
- `rlStochasticActorRepresentation` — Actor with stochastic actions, based on observations from the environment.

These objects all you to easily implement custom training loops for your own reinforcement learning algorithms. For more information, see Train Reinforcement Learning Policy Using Custom Training Loop.

Compatibility Considerations

The `rlRepresentation` function is no longer recommended. Use one of the four new objects instead. For more information, see “rlRepresentation is not recommended” on page 2-3.

Continuous Action Spaces: Train AC, PG, and PPO agents in environments with continuous action spaces

Previously, you could train AC, PG, and PPO agents only in environments with discrete action spaces. Now, you can also train these agents in environments with continuous action spaces. For more information see `rlACAgent`, `rlPGAgent`, `rlPPOAgent`, and Create Policy and Value Function Representations.

Recurrent Neural Networks: Train DQN and PPO agents with recurrent deep neural network policies and value functions

You can now train DQN and PPO agents using recurrent neural network policy and value function representations. For more information, see `rlDQNAgent`, `rlPPOAgent`, and Create Policy and Value Function Representations.

TD3 Agent: Create twin-delayed deep deterministic policy gradient agents

The twin-delayed deep deterministic (TD3) algorithm is a state-of-the-art reinforcement learning algorithm for continuous action spaces. It often exhibits better learning speed and performance compared to deep deterministic policy gradient (DDPG) algorithms. For more information on TD3 agents, see Twin-Delayed Deep Deterministic Policy Gradient Agents. For more information on creating TD3 agents, see `rlTD3Agent` and `rlTD3AgentOptions`.

Softplus Layer: Create deep neural network layer using the softplus activation function

You can now use the new `softplusLayer` layer when creating deep neural networks. This layer implements the softplus activation function $Y = \log(1 + e^X)$, which ensures that the output is always positive. This activation function is a smooth continuous version of `reluLayer`.

Parallel Processing: Improved memory usage and performance

For experience-based parallelization, off-policy agents now flush their experience buffer before distributing them to the workers. Doing so mitigates memory issues when agents with large observation spaces are trained using many workers. Additionally, the synchronous gradient algorithm has been numerically improved, and the overhead for parallel training has been reduced.

Deep Network Designer: Scaling, quadratic, and softplus layers now supported

Reinforcement Learning Toolbox custom layers, including the `scalingLayer`, `quadraticLayer`, and `softplusLayer`, are now supported in the Deep Network Designer app.

New Examples: Train reinforcement learning agents for robotics and imitation learning applications

This release includes the following new reference examples.

- Train PPO Agent to Land Rocket — Train a PPO agent to land a rocket in an environment with a discrete action space.
- Train DDPG Agent with Pretrained Actor Network — Train a DDPG agent using an actor network that has been previously trained using supervised learning.
- Imitate Nonlinear MPC Controller for Flying Robot — Train a deep neural network to imitate a nonlinear MPC controller.

Functionality being removed or changed

`rlRepresentation` is not recommended
Still runs

`rlRepresentation` is not recommended. Depending on the type of representation being created, use one of the following objects instead:

- `rlValueRepresentation` — State value critic, computed based on observations from the environment.
- `rlQValueRepresentation` — State-action value critic, computed based on both actions and observations from the environment.
- `rlDeterministicActorRepresentation` — Actor with deterministic actions, for continuous action spaces, based on observations from the environment.
- `rlStochasticActorRepresentation` — Actor with stochastic actions, based on observations from the environment.

The following table shows some typical uses of the `rlRepresentation` function to create neural network-based critics and actors, and how to update your code with one of the new objects instead.

Network-Based Representations: Not Recommended	Network-Based Representations: Recommended
`rep = rlRepresentation(net,obsInfo,'Observation',obsName)`, with `net` having only observations as inputs, and a single scalar output.	`rep = rlValueRepresentation(net,obsInfo,'Observation',obsName)`. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `rlACAgent` or `rlPGAgent` agent.
`rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`, with `net` having both observations and action as inputs, and a single scalar output.	`rep = rlQValueRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`. Use this syntax to create a single-output state-action value representation for a critic that takes both observation and action as input, such as a critic for an `rlDQNAgent` or `rlDDPGAgent` agent.
`rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`, with `net` having observations as inputs and actions as outputs, and `actInfo` defining a continuous action space.	`rep = rlDeterministicActorRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`. Use this syntax to create a deterministic actor representation for a continuous action space.
`rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsName,'Action',actName)`, with `net` having observations as inputs and actions as outputs, and `actInfo` defining a discrete action space.	`rep = rlStochasticActorRepresentation(net,obsInfo,actInfo,'Observation',obsName)`. Use this syntax to create a stochastic actor representation for a discrete action space.

The following table shows some typical uses of the `rlRepresentation` objects to express table-based critics with discrete observation and action spaces, and how to update your code with one of the new objects instead.

Table-Based Representations: Not Recommended	Table-Based Representations: Recommended
`rep = rlRepresentation(tab)`, with `tab` containing a value table consisting in a column vector as long as the number of possible observations.	`rep = rlValueRepresentation(tab,obsInfo)`. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `rlACAgent` or `rlPGAgent` agent.

Table-Based Representations: Not Recommended	Table-Based Representations: Recommended
`rep = rlRepresentation(tab)`, with `tab` containing a Q-value table with as many rows as the possible observations and as many columns as the possible actions.	`rep = rlQValueRepresentation(tab,obsInfo,actInfo)`. Use this syntax to create a single-output state-action value representation for a critic that takes both observation and action as input, such as a critic for an `rlDQNAgent` or `rlDDPGAgent` agent.

The following table shows some typical uses of the `rlRepresentation` function to create critics and actors which use a custom basis function, and how to update your code with one of the new objects instead. In the recommended function calls, the first input argument is a two-element cell array containing both the handle to the custom basis function and the initial weight vector or matrix.

Custom Basis Function-Based Representations: Not Recommended	Custom Basis Function-Based Representations: Recommended
`rep = rlRepresentation(basisFcn,W0,obsInfo)`, where the basis function has only observations as inputs and `W0` is a column vector.	`rep = rlValueRepresentation({basisFcn,W0},obsInfo)`. Use this syntax to create a representation for a critic that does not require action inputs, such as a critic for an `rlACAgent` or `rlPGAgent` agent.
`rep = rlRepresentation(basisFcn,W0,{obsInfo,actInfo})`, where the basis function has both observations and action as inputs and `W0` is a column vector.	`rep = rlQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a single-output state-action value representation for a critic that takes both observation and action as input, such as a critic for an `rlDQNAgent` or `rlDDPGAgent` agent.
`rep = rlRepresentation(basisFcn,W0,obsInfo,actInfo)`, where the basis function has observations as inputs and actions as outputs, `W0` is a matrix, and `actInfo` defines a continuous action space.	`rep = rlDeterministicActorRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a deterministic actor representation for a continuous action space.
`rep = rlRepresentation(basisFcn,W0,obsInfo,actInfo)`, where the basis function has observations as inputs and actions as outputs, `W0` is a matrix, and `actInfo` defines a discrete action space.	`rep = rlStochasticActorRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a deterministic actor representation for a discrete action space.

Target update method settings for DQN agents have changed
Behavior change

Target update method settings for DQN agents have changed. The following changes require updates to your code:

- The `TargetUpdateMethod` option has been removed. Now, DQN agents determine the target update method based on the `TargetUpdateFrequency` and `TargetSmoothFactor` option values.
- The default value of `TargetUpdateFrequency` has changed from `4` to `1`.

To use one of the following target update methods, set the `TargetUpdateFrequency` and `TargetSmoothFactor` properties as indicated.

Update Method	`TargetUpdateFrequency`	`TargetSmoothFactor`
Smoothing	`1`	Less than `1`
Periodic	Greater than `1`	`1`
Periodic smoothing (new method in R2020a)	Greater than `1`	Less than `1`

The default target update configuration, which is a smoothing update with a `TargetSmoothFactor` value of `0.001`, remains the same.

Update Code

This table shows some typical uses of `rlDQNAgentOptions` and how to update your code to use the new option configuration.

Not Recommended	Recommended
`opt = rlDQNAgentOptions(...` `    'TargetUpdateMethod',"smoothing");`	`opt = rlDQNAgentOptions;`

`opt = rlDQNAgentOptions(...`
`    'TargetUpdateMethod',"periodic");`
	`opt = rlDQNAgentOptions;`
	`opt.TargetUpdateFrequency = 4;`
	`opt.TargetSmoothFactor = 1;`

`opt = rlDQNAgentOptions;`
`opt.TargetUpdateMethod = "periodic";`
`opt.TargetUpdateFrequency = 5;`
	`opt = rlDQNAgentOptions;`
	`opt.TargetUpdateFrequency = 5;`
	`opt.TargetSmoothFactor = 1;`

Target update method settings for DDPG agents have changed

Behavior change

Target update method settings for DDPG agents have changed. The following changes require updates to your code:

- The `TargetUpdateMethod` option has been removed. Now, DDPG agents determine the target update method based on the `TargetUpdateFrequency` and `TargetSmoothFactor` option values.
- The default value of `TargetUpdateFrequency` has changed from `4` to `1`.

To use one of the following target update methods, set the `TargetUpdateFrequency` and `TargetSmoothFactor` properties as indicated.

Update Method	`TargetUpdateFrequency`	`TargetSmoothFactor`
Smoothing	`1`	Less than `1`
Periodic	Greater than `1`	`1`

Update Method	TargetUpdateFrequency	TargetSmoothFactor
Periodic smoothing (new method in R2020a)	Greater than 1	Less than 1

The default target update configuration, which is a smoothing update with a `TargetSmoothFactor` value of `0.001`, remains the same.

Update Code

This table shows some typical uses of `rlDDPGAgentOptions` and how to update your code to use the new option configuration.

Not Recommended	Recommended
`opt = rlDDPGAgentOptions(...` `    'TargetUpdateMethod',"smoothing");`	`opt = rlDDPGAgentOptions;`

`opt = rlDDPGAgentOptions(...`
`    'TargetUpdateMethod',"periodic");`
	`opt = rlDDPGAgentOptions;`
	`opt.TargetUpdateFrequency = 4;`
	`opt.TargetSmoothFactor = 1;`

`opt = rlDDPGAgentOptions;`
`opt.TargetUpdateMethod = "periodic";`
`opt.TargetUpdateFrequency = 5;`
	`opt = rlDDPGAgentOptions;`
	`opt.TargetUpdateFrequency = 5;`
	`opt.TargetSmoothFactor = 1;`

getLearnableParameterValues is now getLearnableParameters
Behavior change

`getLearnableParameterValues` is now `getLearnableParameters`. To update your code, change the function name from `getLearnableParameterValues` to `getLearnableParameters`. The syntaxes are equivalent.

setLearnableParameterValues is now setLearnableParameters
Behavior change

`setLearnableParameterValues` is now `setLearnableParameters`. To update your code, change the function name from `setLearnableParameterValues` to `setLearnableParameters`. The syntaxes are equivalent.

R2019b

Version: 1.1

New Features

Bug Fixes

Parallel Agent Simulation: Verify trained policies by running multiple agent simulations in parallel

You can now run multiple agent simulations in parallel. If you have Parallel Computing Toolbox™ software, you can run parallel simulations on multicore computers. If you have MATLAB® Parallel Server™ software, you can run parallel simulations on computer clusters or cloud resources. For more information, see `rlSimulationOptions`.

PPO Agent: Train policies using proximal policy optimization algorithm for improved training stability

You can now train policies using proximal policy optimization (PPO). This algorithm is a type of policy gradient training that alternates between sampling data through environmental interaction and optimizing a clipped surrogate objective function using stochastic gradient descent. The clipped surrogate objective function improves training stability by limiting the size of the policy change at each step.

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

New Examples: Train reinforcement learning policies for applications such as robotics, automated driving, and control design

The following new examples show how to train policies for robotics, automated driving, and control design:

- Quadruped Robot Locomotion Using DDPG Agent
- Imitate MPC Controller for Lane Keep Assist

R2019a

Version: 1.0

New Features

Reinforcement Learning Algorithms: Train policies using DQN, DDPG, A2C, and other algorithms

Using Reinforcement Learning Toolbox™ software, you can train policies using several standard reinforcement learning algorithms. You can create agents to train policies for the following:

- Q-learning
- SARSA
- Deep Q-networks (DQN)
- Deep deterministic policy gradients (DDPG)
- Policy gradient (PG)
- Advantage actor-critic (A2C)

You can also train policies using other algorithms by creating a custom agent.

For more information on creating and training agents, see Reinforcement Learning Agents and Train Reinforcement Learning Agents.

Environment Modeling: Create MATLAB and Simulink environment models and provide observation and reward signals for training policies

In a reinforcement learning scenario, the environment models the dynamics and system behavior with which the agent interacts. To define an environment model, you specify the following:

- Action and observation signals that the agent uses to interact with the environment.
- Reward signal that the agent uses to measure its success.
- Environment dynamic behavior.

You can model your environment using MATLAB and Simulink. For more information, see Create MATLAB Environments for Reinforcement Learning and Create Simulink Environments for Reinforcement Learning

Policy and Value Function Representation: Parameterize policies using deep neural networks, linear basis functions, and look-up tables

Reinforcement Learning Toolbox software provides objects for actor and critic representations. The actor represents the policy that selects the action to take. The critic represents the value function that estimates the value of the current policy. Depending on your application and selected agent, you can define policy and value functions using deep neural networks, linear basis functions, or look-up tables. For more information, see Create Policy and Value Function Representations.

Interoperability: Import policies from Keras and the ONNX model format

You can import existing deep neural network policies and value functions from other deep learning frameworks, such as Keras and the ONNX™ format. For more information, see Import Policy and Value Function Representations.

Training Acceleration: Parallelize environment simulations and gradient calculations on GPUs and multicore CPUs for policy training

You can accelerate policy training by running parallel training simulations. If you have:

- Parallel Computing Toolbox software, you can run parallel simulations on multicore computers
- MATLAB Parallel Server software, you can run parallel simulations on computer clusters or cloud resources

You can also speed up deep neural network training and inference with high-performance NVIDIA® GPUs.

For more information, see Train Reinforcement Learning Agents.

Code Generation: Deploy trained policies to embedded devices through automatic code generation for CPUs and GPUs

Once you have trained your reinforcement learning policy, you can generate code for policy deployment. You can generate optimized CUDA® code using GPU Coder™ and C/C++ code using MATLAB Coder™.

You can deploy trained policies as C/C++ shared libraries, Microsoft® .NET Framework assemblies, Java® classes, and Python® packages.

For more information, see Deploy Trained Reinforcement Learning Policies.

Reference Examples: Implement controllers using reinforcement learning for automated driving and robotics applications

This release includes the following examples on training reinforcement learning policies for robotics and automated driving applications:

- Train DDPG Agent to Control Flying Robot
- Train Biped Robot to Walk Using DDPG Agent
- Train DQN Agent for Lane Keeping Assist
- Train DDPG Agent for Adaptive Cruise Control
- Train DDPG Agent for Path Following Control