Reinforcement Learning Toolbox™ Release Notes

MATLAB®

MathWorks®

Contents

R2023a

Learning from Data: Train agents offline using previously recorded data

You can now train off-policy agents (DQN, SAC, DDPG, and TD3) offline, using an existing dataset, instead of an environment. For more information, see `trainFromData` and `rlTrainingFromDataOptions`.

To deal with possible differences between the probability distribution of the dataset and the one generated by the environment, use the batch data regularization options provided for off-policy agents. For more information, see the new `BatchDataRegularizerOptions` property of the off-policy agents options objects, as well as the new `rlBehaviorCloningRegularizerOptions` and `rlConservativeQLearningOptions` options objects.

Data Logging: Visualize logged data in Reinforcement Learning Data Viewer

You can now visualize logged data using the **Reinforcement Learning Data Viewer**, a new interactive tool shipped with Reinforcement Learning Toolbox™. For more information, see `rlDataViewer`.

Training: Stop and resume training in the Reinforcement Learning Designer App

You can now resume agent training from the **Reinforcement Learning Designer**. When training terminates, either because a termination condition is reached or because you click **Stop Training** in the Reinforcement Learning Episode Manager, the stored training statistics and results can be now used to resume training from the exact point at which it was stopped.

Hindsight Replay Memory: Improve sample efficiency for goal-conditioned tasks with sparse rewards

You can now improve sample efficiency of off-policy agents (DQN, TD3, SAC, DDPG) using hindsight replay memory, which is a data augmentation method for goal-conditioned tasks. When the reward from the environment is sparse, hindsight replay memory can improve sample efficiency.

By default, built-in off-policy agents use an `rlReplayMemory` object as their experience buffer. Agents uniformly sample data from this buffer. To perform uniform on nonuniform hindsight replay memory, replace the default experience buffer with one of the following objects.

- `rlHindsightReplayMemory` — Replay memory with uniform sampling
- `rlHindsightPrioritizedReplayMemory` — Replay memory with prioritized sampling, which can further improve sample efficiency

Hindsight experience replay does not support agents that use recurrent neural networks.

Replay Memory: Validate experiences before adding to replay memory buffer

You can now validate experiences before adding them to a replay memory buffer using the `validateExperience` function. If the experiences are not compatible with the replay memory, `validateExperience` generates an error message in the MATLAB® command window.

R2022b

Deployment: Policy Block

The new Policy Simulink® block simplifies the process of inserting a reinforcement learning policy into a Simulink model. Use it for simulation, code generation and deployment purposes.

You can automatically generate and configure a Policy block using either the new function `generatePolicyBlock` or the new button on the RL Agent block mask.

For more information, see `generatePolicyBlock` and the Policy block. For an example, see Generate Policy Block for Deployment.

Training: Enhanced logging capability for built-in agents

You can now log data such as experience, losses and learnable parameters to disk. Also, for Simulink environments, you can log any signal value. Once collected, this data can then be loaded into memory and analyzed.

For more information, see `rlDataLogger`. For an example, see Log Training Data To Disk.

Training: Prioritized experience replay for off-policy agents

You can now improve sample efficiency of off-policy agents (DQN, TD3, SAC, DDPG) using prioritized experience replay. To do so, replace the agent experience buffer with an `rlPrioritizedReplayMemory` object.

Prioritized experience replay does not support agents that use recurrent neural networks.

R2022a

New Agent Architecture: Create memory-efficient agents with more modular and scalable actors and critics

You can now create agents using a redesigned architecture which is more modular, scalable, and memory efficient, while also facilitating parallel training. The new architecture introduces new functions and objects and changes some previously existing functionalities.

New Approximator Objects

You can represent actor and critic functions using six new approximator objects. These objects replace the previous representation objects and improve efficiency, readability, scalability, and flexibility.

- `rlValueFunction` — State value critic, computed based on an observation from the environment.
- `rlQValueFunction` — State-action value critic with a scalar output, which is the value of an action given an observation from the environment. The two inputs are a possible action and an observation from the environment.
- `rlVectorQValueFunction` — State-action value critic with a vector output, where each element of the vector is the value of one of the possible actions. The input is an observation from the environment.
- `rlContinuousDeterministicActor` — Actor with a continuous action space. The output is a deterministic action, the input is an observation from the environment.
- `rlDiscreteCategoricalActor` — Actor with a discrete action space. The output is a stochastic action from a categorical distribution, the input is an observation from the environment.
- `rlContinuousGaussianActor` — Actor with a continuous action space. The output is a stochastic action sampled from a Gaussian distribution, the input is an observation from the environment.

When creating a critic or an actor, you can now select and update optimization options using the new `rlOptimizerOptions` object, instead of using the older `rlRepresentationOptions` object.

Specifically, you can create an agent options object and set its `CriticOptimizerOptions` and `ActorOptimizerOptions` properties to suitable `rlOptimizerOptions` objects. Then you pass the agent options object to the function that creates the agent.

Alternatively, you can create the agent and then use dot notation to access the optimization options for the agent actor and critic, for example:
`agent.AgentOptions.ActorOptimizerOptions.LearnRate = 0.1;`.

New Objects for Custom Agents and Custom Training Loops

To implement a customized agent, you can instantiate a policy using the following new policy objects.

- `rlMaxQPolicy` — This object implements a policy that selects the action that maximizes a discrete state-action value function.
- `rlEpsilonGreedyPolicy` — This object implements a policy that selects the action that maximizes a discrete state-action value function with probability `1-Epsilon`, otherwise selects a random action.
- `rlDeterministicActorPolicy` — This object implements a policy that you can use to implement custom agents with a continuous action space.

- `rlAdditiveNoisePolicy` — This object is similar to `rlDeterministicActorPolicy` but noise is added to the output according to an internal noise model.
- `rlStochasticActorPolicy` — This object implements a policy that you can use to implement custom agents with a continuous action space.

For more information on these policy objects, at the MATLAB command line type, `help` followed by the policy object name.

You can use the new `rlReplayMemory` object to append, store, save, sample and replay experience data. Doing so makes it easier to implement custom training loops and your own reinforcement learning algorithms.

When creating a customized training loop or agent you can also access optimization features using the objects created by the new `rlOptimizer` function. Specifically, create an optimizer algorithm object using `rlOptimizer`, and optionally use dot notation to modify its properties. Then, create a structure and set its `CriticOptimizer` or `ActorOptimizer` field to the optimizer object. When you call `runEpisode`, pass the structure as an input parameter. The `runEpisode` function can then use the update method of the optimizer object to update the learnable parameters of your actor or critic.

For more information, see Custom Training Loop with Simulink Action Noise and Train Reinforcement Learning Policy Using Custom Training Loop.

Compatibility Considerations

The following representation objects are no longer recommended:

- `rlValueRepresentation`
- `rlQValueRepresentation`
- `rlDeterministicActorRepresentation`
- `rlStochasticActorRepresentation`

Also, the corresponding representation options object, `rlRepresentationOptions`, is no longer recommended, use an `rlOptimizerOptions` object instead.

For more information on how to update your code to use the new objects, see "Representation objects are not recommended" on page 3-5.

Neural Network Environment: Use an approximated environment model based on a deep neural network

You can now create an environment object that uses a deep neural network to calculate state transitions and rewards. Using such an environment, you can:

- Create an internal environment model for a model-based policy optimization (MBPO) agent. For more information on MBPO agents, see Model-Based Policy Optimization Agents.
- Create an environment for training other types of reinforcement learning agents. You can identify the state-transition network using experimental or simulated data. Depending on your application, using a neural network environment as an approximation of a more complex first-principle environment can speed up your simulation and training.

To create a neural network environment, use an `rlNeuralNetworkEnvironment` object.

Model Based Policy Optimization Agent: Use a model of the environment to improve sample efficiency and exploration

You can now create and train model-based policy optimization (MBPO) agents. An MBPO agent uses a neural network to internally approximate a transition model of the environment. This reusable internal model allows for a greater sample efficiency and a more effective exploration, compared to a typical model-free agent.

For more information on creating MBPO agents, see Model-Based Policy Optimization Agents.

Multi-Agent Reinforcement Learning: Train multiple agents in a centralized manner for more efficient exploration and learning

You can now group agents according to a common learning strategy and specify whether they learn in a centralized manner (that is all agents in a group share experiences) or decentralized manner (agents do not share experiences).

Centralized learning boosts exploration and facilitates learning in applications where the agents perform a collaborative (or the same) task.

For more information on creating training options set for multiple agents, see `rlMultiAgentTrainingOptions`.

Training: Stop and resume agent training

You can now resume agent training from a training result object returned by the `train` function. When training terminates, either because a termination condition is reached or because you click **Stop Training** in the Reinforcement Learning Episode Manager, the stored training statistics and results can be now used to resume training from the exact point at which it was stopped.

For more information and examples, see the `train` reference page.

Event-Based Simulation: Use RL Agent block inside conditionally executed subsystems

You can now use the RL Agent Simulink block inside a conditionally executed subsystem, such as a Triggered Subsystem (Simulink) or a Function-Call Subsystem (Simulink). To do so, you must specify the sample time of the reinforcement learning agent object specified in the RL Agent block as `-1` so that the block can inherit the sample time of its parent subsystem.

For more information, see the `SampleTime` property of any agent options object. For more information on conditionally executed subsystems, see Conditionally Executed Subsystems Overview (Simulink).

RL Agent Block: Learn from last action applied to the environment

For some applications, when training an agent in a Simulink environment, the action applied to the environment can differ from the action output by the RL Agent block. For example, the Simulink model can contain a saturation block on the action output signal.

In such cases, to improve learning results, you can now enable an input port to connect the last action signal applied to the environment.

Reinforcement Learning Designer App: Support for SAC and TRPO agents

You can now create Soft Actor-Critic Agents and Trust Region Policy Optimization Agents from **Reinforcement Learning Designer**.

For more information on creating agents using **Reinforcement Learning Designer**, see Create Agents Using Reinforcement Learning Designer.

New Examples: Train agents for robotics and automated parking applications

This release includes the following new reference examples.

- Train Reinforcement Learning Agents To Control Quanser QUBE™ Pendulum — Train a SAC agent to generate a swing-up reference trajectory for an inverted pendulum and a PPO agent as a mode-selection controller.
- Run SIL and PIL Verification for Reinforcement Learning — Perform software-in-the-loop and processor-in-the-loop verification of trained reinforcement learning agents.
- Train SAC Agent for Ball Balance Control — Control a Kinova robot arm to balance a ball on a plate using a SAC agent.
- Automatic Parking Valet with Unreal Engine Simulation — Implement a hybrid reinforcement learning and model predictive control system that searches a parking lot and parks in an open space.

Functionality being removed or changed

Representation objects are not recommended
Still runs

Functions to create representation objects are no longer recommended. Depending on the type of actor or critic being created, use one of the following objects instead.

Representation Object: Not Recommended	Approximator Object: Recommended	Usage
`rlValueRepresentation`	`rlValueFunction`	State value critic, computed based on an observation from the environment. This type of critic is used in `rlACAgent`, `rlPGAgent`, `rlPPOAgent` and `rlTRPOAgent` agents.

Representation Object: Not Recommended	Approximator Object: Recommended	Usage
`rlQValueRepresentation`	`rlQValueFunction`	State-action value critic with a scalar output, which is the value of an action given an observation from the environment. The two inputs are a possible action and an observation from the environment. It is used in `rlDQNAgent` and `rlDDPGAgent` agents.
`rlQValueRepresentation`	`rlVectorQValueFunction`	State-action value critic with a vector output, where each element of the vector is the value of one of the possible actions. The input is an observations from the environment. It is used in `rlDQNAgent` agents, (and preferred over single-output critics).
`rlDeterministicActorRepresentation`	`rlContinuousDeterministicActor`	Actor with a continuous action space. The output is a deterministic action, the input is an observation from the environment. This kind of actor is used in `rlDDPGAgent` and `rlTD3Agent` agents.
`rlStochasticActorRepresentation`	`rlDiscreteCategoricalActor`	Actor with a discrete action space. The output is a stochastic action sampled from a categorical (also known as Multinoulli) distribution, the input is an observation from the environment. it is used in `rlPGAgent`, `rlACAgent` agents, as well as in `rlPPOAgent` and `rlTRPOAgent` agents with a discrete action space.
`rlStochasticActorRepresentation`	`rlContinuousGaussianActor`	Actor with a continuous action space. The output is a stochastic action sampled from a Gaussian distribution, the input is an observation from the environment. It is used in `rlSACAgent` agents as well as in `rlPGAgent`, `rlACAgent`, `rlTRPOAgent` and `rlPPOAgent` agents with a continuous action space.

`rlRepresentationOptions` objects are no longer recommended. To specify optimization options for actors and critics, use `rlOptimizerOptions` objects instead.

Specifically, you can create an agent options object and set its `CriticOptimizerOptions` and `ActorOptimizerOptions` properties to suitable `rlOptimizerOptions` objects. Then you pass the agent options object to the function that creates the agent. This workflow is shown in the following table.

`rlRepresentationOptions`: Not Recommended	`rlOptimizerOptions`: Recommended

`crtOpts = rlRepresentationOptions(...`
`'GradientThreshold',1);`
 	
`critic = rlValueRepresentation(...`
`net,obsInfo,'Observation',{'obs'},crtOpts);`
	`criticOpts = rlOptimizerOptions(...`
	`'GradientThreshold',1);`
	
	`agentOpts = rlACAgentOptions(...`
	`'CriticOptimizerOptions',crtOpts);`
	
	`agent = rlACAgent(actor,critic,agentOpts)`

Alternatively, you can create the agent and then use dot notation to access the optimization options for the agent actor and critic, for example: `agent.AgentOptions.ActorOptimizerOptions.GradientThreshold = 1;`.

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

Network-Based Representations: Not Recommended	Network-Based Approximators: Recommended
`myCritic = rlValueRepresentation(net,obsInfo,'Observation',obsNames)`, with `net` having observations as inputs and a single scalar output.	`myCritic = rlValueFunction(net,obsInfo,'ObservationInputNames',obsNames)`. Use this syntax to create a state value function object for a critic that does not require action inputs.
`myCritic = rlQValueRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action',actNames)`, with `net` having both observations and actions as inputs and a single scalar output.	`myCritic = rlQValueFunction(net,obsInfo,actInfo,'ObservationInputNames',obsNames,'ActionInputNames',actNames)`. Use this syntax to create a single-output state-action value function object for a critic that takes both observation and action as inputs.
`myCritic = rlQValueRepresentation(net,obsInfo,actInfo,'Observation',obsNames)` with `net` having only the observations as inputs and a single output layer having as many elements as the number of possible discrete actions.	`myCritic = rlVectorQValueFunction(net,obsInfo,actInfo,'ObservationInputNames',obsNames)`. Use this syntax to create a multiple-output state-action value function object for a critic with a discrete action space. This critic takes observations as inputs, and outputs a vector in which each element is the value of one of the possible actions.
`myActor = rlDeterministicActorRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action',actNames)` with `actInfo` defining a continuous action space and `net` having observations as inputs and a single output layer with as many elements as the number of dimensions of the continuous action space.	`myActor = rlContinuousDeterministicActor(net,obsInfo,actInfo,'ObservationInputNames',obsNames)`. Use this syntax to create a deterministic actor object with a continuous action space.

Network-Based Representations: Not Recommended	**Network-Based Approximators: Recommended**
`myActor = rlStochasticActorRepresentation(net,obsInfo,actInfo,'Observation',obsNames)`, with `actInfo` defining a discrete action space and `net` having observations as inputs and a single output layer with as many elements as the number of possible discrete actions.	`myActor = rlDiscreteCategoricalActor(net,obsInfo,actInfo,'ObservationInputNames',obsNames)`. Use this syntax to create a stochastic actor object with a discrete action space. This actor samples its action from a categorical (also known as Multinoulli) distribution.
`myActor = rlStochasticActorRepresentation(net,obsInfo,actInfo,'Observation',obsNames)`, with `actInfo` defining a continuous action space and `net` having observations as inputs and a single output layer with twice as many elements as the number of dimensions of the continuous action space (representing, in sequence, all the means and all the standard deviations of every action dimension).	`myActor = rlContinuousGaussianActor(net,obsInfo,actInfo,'ObservationInputNames',obsNames,'ActionMeanOutputNames',actMeanNames,'ActionStandardDeviationOutputNames',actStdNames)`. Use this syntax to create a stochastic actor object with a continuous action space. This actor samples its action from a Gaussian distribution, and you must provide the names of the network outputs representing the mean and standard deviations for the action.

The following table shows some typical uses of the representation 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 Approximators: Recommended**
`rep = rlValueRepresentation(tab,obsInfo)` where the table `tab` contains a column vector with as many elements as the number of possible observations.	`rep = rlValueFunction(tab,obsInfo)`. Use this syntax to create a value function object for a critic that does not require action inputs.
`rep = rlQValueRepresentation(tab,obsInfo,actInfo)`, where the table `tab` contains a vector with as many elements as the number of possible observations plus the number of possible actions.	`rep = rlQValueFunction(tab,obsInfo,actInfo)`. Use this syntax to create a single-output state-action value function object for a critic that takes both observations and actions as input.
`rep = rlQValueRepresentation(tab,obsInfo,actInfo)`, where the table `tab` contains a Q-value table with as many rows as the number of possible observations and as many columns as the number of possible actions.	`rep = rlVectorQValueFunction(tab,obsInfo,actInfo)`. Use this syntax to create a multiple-output state-action value function object for a critic with a discrete action space. This critic takes observations as inputs, and outputs a vector in which each element is the value of one of the possible actions. It is good practice to use critics with vector outputs when possible.

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

Custom Basis Function-Based Representations: Not Recommended	**Custom Basis Function-Based Approximators: Recommended**
`rep = rlValueRepresentation({basisFcn,W0},obsInfo)`, where the basis function has only observations as inputs and `W0` is a column vector.	`rep = rlValueFunction({basisFcn,W0},obsInfo)`. Use this syntax to create a value function object for a critic that does not require action input.
`rep = rlQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`, where the basis function has both observations and action as inputs and `W0` is a column vector.	`rep = rlQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a single-output state-action value function object for a critic that takes both observation and action as inputs.
`rep = rlQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`, where the basis function has both observations and action as inputs and `W0` is a matrix with as many columns as the number of possible actions.	`rep = rlVectorQValueRepresentation({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a multiple-output state-action value function object for a critic with a discrete action space. This critic takes observations as inputs, and outputs a vector in which each element is the value of one of the possible actions. It is good practice to use critics with vector outputs when possible.
`rep = rlDeterministicActorRepresentation({basisFcn,W0},obsInfo,actInfo)`, where the basis function has observations as inputs and actions as outputs, `W0` is a matrix with as many columns as the number of possible actions, and `actInfo` defines a continuous action space.	`rep = rlContinuousDeterministicActor({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a deterministic actor object with a continuous action space.
`rep = rlStochasticActorRepresentation({basisFcn,W0},obsInfo,actInfo)`, where the basis function has observations as inputs and actions as outputs, `W0` is a matrix with as many columns as the number of possible actions, and `actInfo` defines a discrete action space.	`rep = rlDiscreteCategoricalActor({basisFcn,W0},obsInfo,actInfo)`. Use this syntax to create a stochastic actor object with a discrete action space which returns an action sampled from a categorical (also known as Multinoulli) distribution.

For more information on the new approximator objects, see, `rlTable`, `rlValueFunction`, `rlQValueFunction`, `rlVectorQValueFunction`, `rlContinuousDeterministicActor`, `rlDiscreteCategoricalActor`, and `rlContinuousGaussianActor`.

train now returns an object instead of a structure

Behavior change in future release

The `train` function now returns an object or an array of objects as the output. The properties of the object match the fields of the structure returned in previous versions. Therefore, the code based on dot notation works in the same way.

For example, if you train an agent using the following command:

```
trainStats = train(agent,env,trainOptions);
```

When training terminates, either because a termination condition is reached or because you click **Stop Training** in the Reinforcement Learning Episode Manager, `trainStats` is returned as an `rlTrainingResult` object.

The `rlTrainingResult` object contains the same training statistics previously returned in a structure along with data to correctly recreate the training scenario and update the episode manager.

You can use `trainStats` as third argument for another `train` call, which (when executed with the same agents and environment) will cause training to resume from the exact point at which it stopped.

For more information and examples, see `train` and "Training: Stop and resume agent training" on page 3-4. For more information on training agents, see Train Reinforcement Learning Agents.

Training Parallelization Options: DataToSendFromWorkers and StepsUntilDataIsSent properties are no longer active

Warns

The property `DataToSendFromWorkers` of the `ParallelizationOptions` object is no longer active and will be removed in a future release. The data sent from the workers to the learner is now automatically determined based on agent type.

The property `StepsUntilDataIsSent` of the `ParallelizationOptions` object is no longer active and will be removed in a future release. Data is now sent from the workers to the learner at the end each episode.

Attempting to set either of these properties will cause a warning. For more information, see `barrierPenalty`.

Code generated by generatePolicyFunction now uses policy objects

Behavior change in future release

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

R2021b

Rewards Generation: Automatically generate reward functions from controller specifications

You can now generate reinforcement learning reward functions, coded in MATLAB, from:

- Cost and constraint specifications defined in an `mpc` (Model Predictive Control Toolbox) or `nlmpc` (Model Predictive Control Toolbox) controller object. This feature requires Model Predictive Control Toolbox™ software.
- Performance constraints defined in Simulink Design Optimization™ model verification blocks.

For more information, see `generateRewardFunction` as well as `exteriorPenalty`, `hyperbolicPenalty`, and `barrierPenalty`, as well as the examples Generate Reward Function from a Model Predictive Controller for a Servomotor and Generate Reward Function from a Model Verification Block for a Water Tank System.

Episode Manager: Improved layout management for single and multiple agent training

You can now display the training progress of single and multiple agents with **Episode Manager**. The layout management is integrated with the **Reinforcement Learning Designer** app, and exhibits a more consistent plot resizing and moving behavior.

Neural Network Representations: Improved internal handling of dlnetwork objects

The built-in agents now use `dlnetwork` objects as actor and critic representations. In most cases this allows for a speedup of about 30%.

Compatibility Considerations

- `getModel` now returns a `dlnetwork` object.
- Due to numerical differences in the network calculations, previously trained agents might behave differently. If this happens, you can retrain your agents.
- To use Deep Learning Toolbox™ functions that do not support `dlnetwork`, you must convert the network to `layerGraph`. For example, to use `deepNetworkDesigner`, replace `deepNetworkDesigner(network)` with `deepNetworkDesigner(layerGraph(network))`.

Trust Region Policy Optimization Agent: Prevent significant performance drops by restricting updated policy to trust region near current policy

You can now create and train trust region policy optimization (TRPO) agents. TRPO is a policy gradient reinforcement algorithm. It prevents significant performance drops compared to standard policy gradient methods by keeping the updated policy within a trust region close to the current policy.

For more information on creating TRPO agents, see `rlTRPOAgent` and `rlTRPOAgentOptions`.

PPO Agents: Improve agent performance by normalizing advantage function

In some environments, you can improve PPO agent performance by normalizing the advantage function during training. The agent normalizes the advantage function by subtracting the mean advantage value and scaling by the standard deviation.

To enable advantage function normalization, first create an `rlPPOAgentOptions` object. Then, specify the `NormalizedAdvantageMethod` option as one of the following values.

- `"current"` — Normalize the advantage function using the advantage function mean and standard deviation for the current mini-batch of experiences.
- `"moving"` — Normalize the advantage function using the advantage function mean and standard deviation for a moving window of recent experiences. To specify the window size, set the `AdvantageNormalizingWindow` option.

For example, configure the agent options to normalize the advantage function using the mean and standard deviation from the last 500 experiences.

```
opt = rlPPOAgentOptions;
opt.NormalizedAdvantageMethod = "moving";
opt.AdvantageNormalizingWidnow = 500;
```

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

New Example: Create and train custom agent using model-based reinforcement learning

A model-based reinforcement learning agent learns a model of its environment that it can use to generate additional experiences for training. For an example that shows how to create and train such an agent, see Model-Based Reinforcement Learning Using Custom Training Loop.

Functionality being removed or changed

Built-in agents now use dlnetwork objects

Behavior change

The built-in agents now use `dlnetwork` objects as actor and critic representations. In most cases this allows for a speedup of about 30%.

- `getModel` now returns a `dlnetwork` object.
- Due to numerical differences in the network calculations, previously trained agents might behave differently. If this happens, you can retrain your agents.
- To use Deep Learning Toolbox functions that do not support `dlnetwork`, you must convert the network to `layerGraph`. For example, to use `deepNetworkDesigner`, replace `deepNetworkDesigner(network)` with `deepNetworkDesigner(layerGraph(network))`.

R2021a

Reinforcement Learning Designer App: Design, train, and simulate agents using a visual interactive workflow

The new **Reinforcement Learning Designer** app streamlines the workflow for designing, training, and simulating agents. You can now:

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

To open the **Reinforcement Learning Designer** app, at the MATLAB command line, enter the following:

```
reinforcementLearningDesigner
```

For more information, see **Reinforcement Learning Designer**.

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

You can now use recurrent neural networks (RNN) when creating representations for use with PG, DDPG, AC, SAC, and TD3 agents. Previously, only PPO and DQN agents were supported.

RNNs are deep neural networks with a `sequenceInputLayer` input layer and at least one layer that has hidden state information, such as an `lstmLayer`. These networks can be especially useful when the environment has states that are not included in the observation vector.

For more information on creating agent with RNNs, see `rlDQNAgent`, `rlPPOAgent`, and the **Recurrent Neural Networks** section in Create Policy and Value Function Representations.

For more information on creating policies and value functions, see `rlValueRepresentation`, `rlQValueRepresentation`, `rlDeterministicActorRepresentation`, and `rlStochasticActorRepresentation`.

Guided Policy Learning: Perform imitation learning in Simulink by learning policies based external actions

You can now perform imitation learning in Simulink using the RL Agent block. To do so, you pass an external signal to the RL Agent block, such as a control signal from a human expert. The block can pass this action signal to the environment and update its policy based on the resulting observations and rewards.

You can also use this input port to override the agent action for safe learning applications.

inspectTrainingResult Function: Plot training information from a previous training session

You can now plot the saved training information from a previous reinforcement learning training session using the `inspectTrainingResult` function.

By default, the `train` function shows the training progress and results in the Episode Manager. If you configure training to not show the Episode Manager or you close the Episode Manager after training, you can view the training results using the `inspectTrainingResult` function, which opens the Episode Manager.

Deterministic Exploitation: Create PG, AC, PPO, and SAC agents that use deterministic actions during simulation and in generated policy functions

PG, AC, PPO, and SAC agents generate stochastic actions during training. By default, these agents also use stochastic actions during simulation deployment. You can now configure these agents to use deterministic actions during simulations and in generated policy function code.

To enable deterministic exploitation, in the corresponding agent options object, set the `UseDeterministicExploitation` property to `true`. For more information, see `rlPGAgentOptions`, `rlACAgentOptions`, `rlPPOAgentOptions`, or `rlSACAgentOptions`.

For more information on simulating agents and generating policy functions, see `sim` and `generatePolicyFunction`, respectively.

New Examples: Train agent with constrained actions and use DQN agent for optimal scheduling

This release includes the following new reference examples.

- Water Distribution System Scheduling Using Reinforcement Learning — Train a DQN agent to learn an optimal pump scheduling policy for a water distribution system.
- Train Reinforcement Learning Agent with Constraint Enforcement — Train an agent with critical constraints enforced on its actions.

Functionality being removed or changed

Properties defining noise probability distribution in the GaussianActionNoise object have changed

Still runs

The properties defining the probability distribution of the Gaussian action noise model have changed. This noise model is used by TD3 agents for exploration and target policy smoothing.

- The `Variance` property has been replaced by the `StandardDeviation` property.
- The `VarianceDecayRate` property has been replaced by the `StandardDeviationDecayRate` property.
- The `VarianceMin` property has been replaced by the `StandardDeviationMin` property.

When a GaussianActionNoise noise object saved from a previous MATLAB release is loaded, the value of VarianceDecayRate is copied to StandardDeviationDecayRate, while the square root of the values of Variance and VarianceMin are copied to StandardDeviation and StandardDeviationMin, respectively.

The Variance, VarianceDecayRate, and VarianceMin properties still work, but they are not recommended. To define the probability distribution of the Gaussian action noise model, use the new property names instead.

Update Code

This table shows how to update your code to use the new property names for rlTD3AgentOptions object td3opt.

Not Recommended	Recommended
`td3opt.ExplorationModel.Variance = 0.5;`	`td3opt.ExplorationModel.StandardDeviation = sqrt(0.5);`
`td3opt.ExplorationModel.VarianceDecayRate = 0.1;`	`td3opt.ExplorationModel.StandardDeviationDecayRate = 0.1;`
`td3opt.ExplorationModel.VarianceMin = 0.1;`	`td3opt.ExplorationModel.StandardDeviationMin = sqrt(0.1);`
`td3opt.TargetPolicySmoothModel.Variance = 0.5;`	`td3opt.TargetPolicySmoothModel.StandardDeviation = sqrt(0.5);`
`td3opt.TargetPolicySmoothModel.VarianceDecayRate = 0.1;`	`td3opt.TargetPolicySmoothModel.StandardDeviationDecayRate = 0.1;`
`td3opt.TargetPolicySmoothModel.VarianceMin = 0.1;`	`td3opt.TargetPolicySmoothModel.StandardDeviationMin = sqrt(0.1);`

Property names defining noise probability distribution in the OrnsteinUhlenbeckActionNoise object have changed
Still runs

The properties defining the probability distribution of the Ornstein-Uhlenbeck (OU) noise model have been renamed. DDPG and TD3 agents use OU noise for exploration.

- The Variance property has been renamed StandardDeviation.
- The VarianceDecayRate property has been renamed StandardDeviationDecayRate.
- The VarianceMin property has been renamed StandardDeviationMin.

The default values of these properties remain the same. When an OrnsteinUhlenbeckActionNoise noise object saved from a previous MATLAB release is loaded, the values of Variance, VarianceDecayRate, and VarianceMin are copied in the StandardDeviation, StandardDeviationDecayRate, and StandardDeviationMin, respectively.

The Variance, VarianceDecayRate, and VarianceMin properties still work, but they are not recommended. To define the probability distribution of the OU noise model, use the new property names instead.

Update Code

This table shows how to update your code to use the new property names for `rlDDPGAgentOptions` object `ddpgopt` and `rlTD3AgentOptions` object `td3opt`.

Not Recommended	Recommended
`ddpgopt.NoiseOptions.Variance = 0.5;`	`ddpgopt.NoiseOptions.StandardDeviation = 0.5;`
`ddpgopt.NoiseOptions.VarianceDecayRate = 0.1;`	`ddpgopt.NoiseOptions.StandardDeviationDecayRate = 0.1;`
`ddpgopt.NoiseOptions.VarianceMin = 0;`	`ddpgopt.NoiseOptions.StandardDeviationMin = 0;`
`td3opt.ExplorationModel.Variance = 0.5;`	`td3opt.ExplorationModel.StandardDeviation = 0.5;`
`td3opt.ExplorationModel.VarianceDecayRate = 0.1;`	`td3opt.ExplorationModel.StandardDeviationDecayRate = 0.1;`
`td3opt.ExplorationModel.VarianceMin = 0;`	`td3opt.ExplorationModel.StandardDeviationMin = 0;`

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

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 allow 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 7-3.

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',obsNames)`, with `net` having only observations as inputs, and a single scalar output.	`rep = rlValueRepresentation(net,obsInfo,'Observation',obsNames)`. 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)`, with `net` having both observations and action as inputs, and a single scalar output.	`rep = rlQValueRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action',actNames)`. 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',obsNames,'Action',actNames)`, with `net` having observations as inputs and actions as outputs, and `actInfo` defining a continuous action space.	`rep = rlDeterministicActorRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action',actNames)`. Use this syntax to create a deterministic actor representation for a continuous action space.
`rep = rlRepresentation(net,obsInfo,actInfo,'Observation',obsNames,'Action',actNames)`, with `net` having observations as inputs and actions as outputs, and `actInfo` defining a discrete action space.	`rep = rlStochasticActorRepresentation(net,obsInfo,actInfo,'Observation',obsNames)`. 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 with as many elements 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

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