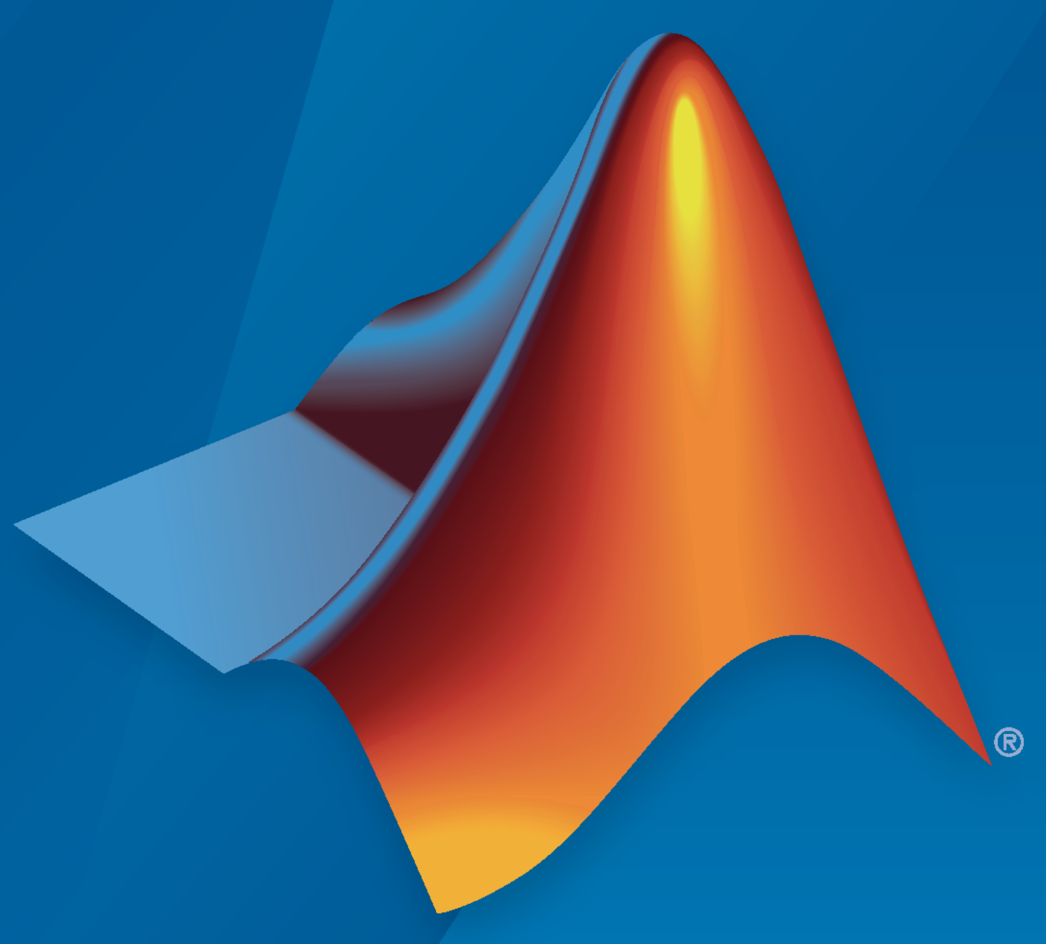
# Reinforcement Learning Toolbox™

User's Guide



MATLAB®

R2023a

# 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™ User's Guide*

© COPYRIGHT 2019–2023 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 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 for more information.

**Revision History**

| | | |
|---|---|---|
| March 2019 | Online only | New for Version 1.0 (Release 2019a) |
| September 2019 | Online only | Revised for Version 1.1 (Release 2019b) |
| March 2020 | Online only | Revised for Version 1.2 (Release 2020a) |
| September 2020 | Online only | Revised for Version 1.3 (Release 2020b) |
| March 2021 | Online only | Revised for Version 2.0 (Release 2021a) |
| September 2021 | Online only | Revised for Version 2.1 (Release 2021b) |
| March 2022 | Online only | Revised for Version 2.2 (Release 2022a) |
| September 2022 | Online only | Revised for Version 2.3 (Release 2022b) |
| March 2023 | Online only | Revised for Version 2.4 (Release 2023a) |

# Contents

## 1 Getting Started

**Reinforcement Learning Toolbox Product Description** . . . . . . . . . . . . . . . **1-2**

**What Is Reinforcement Learning?** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-3**
Reinforcement Learning Workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1-4**

**Reinforcement Learning for Control Systems Applications** . . . . . . . . . . . . **1-6**

**Train Reinforcement Learning Agent in MDP Environment** . . . . . . . . . . . **1-9**

**Train Reinforcement Learning Agent in Basic Grid World** . . . . . . . . . . . **1-14**

**Create Simulink Environment and Train Agent** . . . . . . . . . . . . . . . . . . . . **1-20**

## 2 Create Environments

**Create MATLAB Reinforcement Learning Environments** . . . . . . . . . . . . . . **2-2**
Action and Observation Signals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-2**
Predefined MATLAB Environments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-3**
Custom MATLAB Environments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-3**

**Create or Import MATLAB Environments in Reinforcement Learning Designer** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-5**

**Create Simulink Reinforcement Learning Environments** . . . . . . . . . . . . . **2-8**
Action and Observation Signals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-8**
Predefined Simulink Environments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-9**
Custom Simulink Environments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-9**

**Create or Import Simulink Environments in Reinforcement Learning Designer** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-11**

**Define Reward Signals** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-14**
Continuous Rewards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-14**
Discrete Rewards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-15**
Mixed Rewards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-15**
Reward Generation from Control Specifications . . . . . . . . . . . . . . . . . . . **2-15**

**Load Predefined Grid World Environments** . . . . . . . . . . . . . . . . . . . . . . . **2-17**
Basic Grid World . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2-17**

Deterministic Waterfall Grid Worlds . . . . . 2-18
Stochastic Waterfall Grid Worlds . . . . . 2-20

**Load Predefined Control System Environments** . . . . . **2-23**
Cart-Pole Environments . . . . . 2-23
Double Integrator Environments . . . . . 2-25
Simple Pendulum Environments with Image Observation . . . . . 2-27

**Load Predefined Simulink Environments** . . . . . **2-30**
Simple Pendulum Simulink Model . . . . . 2-30
Cart-Pole Simscape Model . . . . . 2-32

**Create Custom Grid World Environments** . . . . . **2-36**
Grid World Model . . . . . 2-36
Grid World Environment . . . . . 2-40

**Create MATLAB Environment Using Custom Functions** . . . . . **2-41**

**Create Custom MATLAB Environment from Template** . . . . . **2-48**
Create Template Class . . . . . 2-48
Environment Properties . . . . . 2-48
Required Functions . . . . . 2-49
Optional Functions . . . . . 2-51
Environment Visualization . . . . . 2-52
Create Custom Environment . . . . . 2-53

**Water Tank Reinforcement Learning Environment Model** . . . . . **2-55**

## 3 Create Agents

**Reinforcement Learning Agents** . . . . . **3-2**
Built-In Agents . . . . . 3-3
Choose Agent Type . . . . . 3-6
Model-Based Policy Optimization . . . . . 3-7
Extract Policy Objects from Agents . . . . . 3-7
Custom Agents . . . . . 3-7

**Create Agents Using Reinforcement Learning Designer** . . . . . **3-9**
Create Agent . . . . . 3-9
Import Agent . . . . . 3-11
Edit Agent Options . . . . . 3-11
Edit Actor and Critic . . . . . 3-13
Modify Deep Neural Networks . . . . . 3-14
Export Agents and Agent Components . . . . . 3-15

**Q-Learning Agents** . . . . . **3-17**
Critic Function Approximator . . . . . 3-17
Agent Creation . . . . . 3-17
Training Algorithm . . . . . 3-18

**SARSA Agents** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-20**
Critic Function Approximator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-20**
Agent Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-20**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-21**

**Deep Q-Network (DQN) Agents** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-23**
Critic Function Approximator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-23**
Agent Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-24**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-24**
Target Update Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-25**

**Policy Gradient (PG) Agents** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-27**
Actor and Critic Function Approximators . . . . . . . . . . . . . . . . . . . . . . . . **3-27**
Agent Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-28**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-28**

**Actor-Critic (AC) Agents** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-31**
Actor and Critic Function Approximators . . . . . . . . . . . . . . . . . . . . . . . . **3-31**
Agent Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-32**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-32**

**Soft Actor-Critic (SAC) Agents** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-35**
Actor and Critic Function Approximators . . . . . . . . . . . . . . . . . . . . . . . . **3-35**
Agent Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-36**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-37**
Target Update Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-38**

**Deep Deterministic Policy Gradient (DDPG) Agents** . . . . . . . . . . . . . . . . **3-40**
Actor and Critic Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-40**
Agent Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-41**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-41**
Target Update Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-42**

**Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents** . . . . . . **3-44**
Actor and Critic Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-44**
Agent Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-45**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-46**
Target Update Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-47**

**Proximal Policy Optimization (PPO) Agents** . . . . . . . . . . . . . . . . . . . . . . **3-49**
Actor and Critic Function Approximators . . . . . . . . . . . . . . . . . . . . . . . . **3-49**
Agent Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-50**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-51**
Entropy Loss . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-53**

**Trust Region Policy Optimization (TRPO) Agents** . . . . . . . . . . . . . . . . . **3-55**
Actor and Critic Function Approximators . . . . . . . . . . . . . . . . . . . . . . . . **3-56**
Agent Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-56**
Trust Region Policy Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-57**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-58**
Entropy Loss . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-60**

**Model-Based Policy Optimization (MBPO) Agents** . . . . . . . . . . . . . . . . . **3-62**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-63**
Tips . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-66**

**Create Custom Reinforcement Learning Agents** . . . . . . . . . . . . . . . . . . **3-68**
Create Template Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-68**
Agent Properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-68**
Constructor Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-69**
Actor and Critic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-70**
Required Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-70**
Optional Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-73**
Create Custom Agent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3-73**

## 4 Define Policies and Value Functions

**Create Policies and Value Functions** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4-2**
Actors and Critics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4-2**
Policy Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4-5**
Table Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4-6**
Neural Network Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4-7**
Custom Basis Function Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4-13**
Create an Agent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4-14**

**Import Neural Network Models** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **4-15**
Import Actor and Critic for Image Observation Application . . . . . . . . . . . **4-15**

## 5 Train and Validate Agents

**Train Reinforcement Learning Agents** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-3**
Training Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-4**
Episode Manager . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-4**
Save Candidate Agents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-5**
Validate Trained Policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-6**
Environment Visualization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-6**

**Train Agents Using Parallel Computing and GPUs** . . . . . . . . . . . . . . . . . . **5-8**
Using Multiple Processes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-8**
Agent-Specific Parallel Training Considerations . . . . . . . . . . . . . . . . . . . **5-9**
Using GPUs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-10**
Using both Multiple Processes and GPUs . . . . . . . . . . . . . . . . . . . . . . . **5-10**

**Design and Train Agent Using Reinforcement Learning Designer** . . . . . **5-12**

**Specify Training Options in Reinforcement Learning Designer** . . . . . . . **5-16**
Specify Basic Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-16**
Specify Additional Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-16**
Specify Parallel Training Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-17**

**Specify Simulation Options in Reinforcement Learning Designer** . . . . . **5-21**
Specify Basic Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-21**
Specify Parallel Simulation Options . . . . . . . . . . . . . . . . . . . . . . . . . . . **5-21**

Log Training Data to Disk . . . . . . . . . . 5-24

Train Reinforcement Learning Agent for Simple Contextual Bandit Problem . . . . . . . . . . 5-30

Train Agent or Tune Environment Parameters Using Parameter Sweeping . . . . . . . . . . 5-40

Train DQN Agent to Balance Cart-Pole System . . . . . . . . . . 5-50

Train PG Agent to Balance Cart-Pole System . . . . . . . . . . 5-57

Train AC Agent to Balance Cart-Pole System . . . . . . . . . . 5-63

Train PG Agent with Baseline to Control Double Integrator System . . . 5-70

Train DDPG Agent to Control Double Integrator System . . . . . . . . . . 5-77

Train DQN Agent to Swing Up and Balance Pendulum . . . . . . . . . . 5-89

Train DDPG Agent to Swing Up and Balance Pendulum . . . . . . . . . . 5-97

Train DDPG Agent to Swing Up and Balance Cart-Pole System . . . . . . 5-106

Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal . . . . . . . . . . 5-115

Train Reinforcement Learning Agents to Control Quanser QUBE Pendulum . . . . . . . . . . 5-125

Run SIL and PIL Verification for Reinforcement Learning . . . . . . . . . . 5-134

Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation . . . . . . . . . . 5-141

Create DQN Agent Using Deep Network Designer and Train Using Image Observations . . . . . . . . . . 5-152

Train AC Agent to Balance Cart-Pole System Using Parallel Computing . . . . . . . . . . 5-166

Train DDPG Agent to Control Flying Robot . . . . . . . . . . 5-172

Train PPO Agent for a Lander Vehicle . . . . . . . . . . 5-180

Train Multiple Agents to Perform Collaborative Task . . . . . . . . . . 5-190

Train Multiple Agents for Area Coverage . . . . . . . . . . 5-198

Train Multiple Agents for Path Following Control . . . . . . . . . . 5-206

Train DDPG Agent for Adaptive Cruise Control . . . . . . . . . . 5-215

Train DQN Agent for Lane Keeping Assist . . . . . . . . . . 5-226

Train PPO Agent for Automatic Parking Valet . . . . . . . . . . . . . . . . . . . . 5-235

Train DDPG Agent for Path-Following Control . . . . . . . . . . . . . . . . . . . . 5-247

Train DQN Agent for Lane Keeping Assist Using Parallel Computing . 5-257

Train Biped Robot to Walk Using Reinforcement Learning Agents . . . 5-267

Quadruped Robot Locomotion Using DDPG Agent . . . . . . . . . . . . . . . . . 5-278

Train SAC Agent for Ball Balance Control . . . . . . . . . . . . . . . . . . . . . . . . 5-286

Automatic Parking Valet with Unreal Engine Simulation . . . . . . . . . . . 5-300

Generate Reward Function from a Model Predictive Controller for a Servomotor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-315

Generate Reward Function from a Model Verification Block for a Water Tank System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-327

Train TD3 Agent for PMSM Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-341

Water Distribution System Scheduling Using Reinforcement Learning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-353

Imitate MPC Controller for Lane Keeping Assist . . . . . . . . . . . . . . . . . . 5-365

Train DDPG Agent with Pretrained Actor Network . . . . . . . . . . . . . . . . 5-373

Imitate Nonlinear MPC Controller for Flying Robot . . . . . . . . . . . . . . . 5-383

Tune PI Controller Using Reinforcement Learning . . . . . . . . . . . . . . . . 5-392

Train Reinforcement Learning Agent with Constraint Enforcement . . 5-403

Train DQN Agent with LSTM Network to Control House Heating System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-414

Generate Policy Block for Deployment . . . . . . . . . . . . . . . . . . . . . . . . . 5-424

Train Reinforcement Learning Policy Using Custom Training Loop . . . 5-433

Custom Training Loop with Simulink Action Noise . . . . . . . . . . . . . . . . 5-442

Create Agent for Custom Reinforcement Learning Algorithm . . . . . . . 5-456

Train Custom LQR Agent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-466

Train MBPO Agent to Balance Cart-Pole System . . . . . . . . . . . . . . . . . . 5-472

Model-Based Reinforcement Learning Using Custom Training Loop . . 5-484

Train DQN Agent Using Hindsight Experience Replay . . . . . . . . . . . . . 5-500

# 6 Deploy Trained Policies

**Deploy Trained Reinforcement Learning Policies** . . . . . . . . . . . . . . . . . . . . **6-2**
Generate Code Using GPU Coder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-2**
Generate Code Using MATLAB Coder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **6-3**

# Getting Started


- “Reinforcement Learning Toolbox Product Description” on page 1-2
- “What Is Reinforcement Learning?” on page 1-3
- “Reinforcement Learning for Control Systems Applications” on page 1-6
- “Train Reinforcement Learning Agent in MDP Environment” on page 1-9
- “Train Reinforcement Learning Agent in Basic Grid World” on page 1-14
- “Create Simulink Environment and Train Agent” on page 1-20

# Reinforcement Learning Toolbox Product Description

**Design and train policies using reinforcement learning**

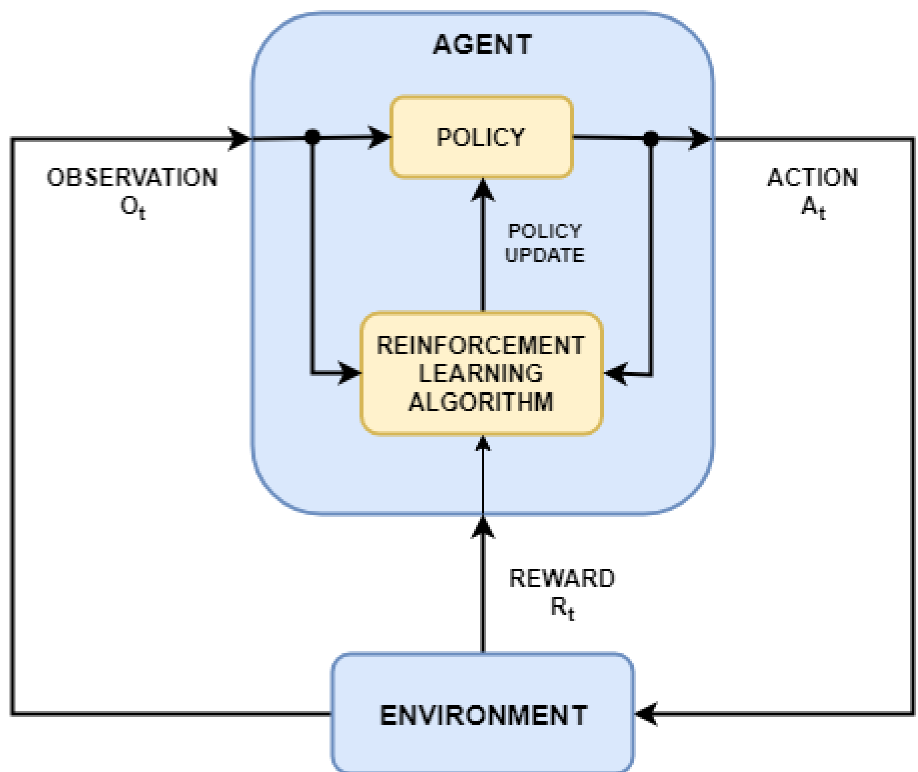
Reinforcement Learning Toolbox™ provides an app, functions, and a Simulink® block for training policies using reinforcement learning algorithms, including DQN, PPO, SAC, and DDPG. You can use these policies to implement controllers and decision-making algorithms for complex applications such as resource allocation, robotics, and autonomous systems.

The toolbox lets you represent policies and value functions using deep neural networks or look-up tables and train them through interactions with environments modeled in MATLAB® or Simulink. You can evaluate the single- or multi-agent reinforcement learning algorithms provided in the toolbox or develop your own. You can experiment with hyperparameter settings, monitor training progress, and simulate trained agents either interactively through the app or programmatically. To improve training performance, simulations can be run in parallel on multiple CPUs, GPUs, computer clusters, and the cloud (with Parallel Computing Toolbox™ and MATLAB Parallel Server™).

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

# What Is Reinforcement Learning?

Reinforcement learning is a goal-directed computational approach where a computer learns to perform a task by interacting with an unknown dynamic environment. This learning approach enables a computer to make a series of decisions to maximize the cumulative reward for the task without human intervention and without being explicitly programmed to achieve the task. The following diagram shows a general representation of a reinforcement learning scenario.




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

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

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

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

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

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

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

## Reinforcement Learning Workflow

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

1. **Formulate problem** — Define the task for the agent to learn, including how the agent interacts with the environment and any primary and secondary goals the agent must achieve.
2. **Create environment** — Define the environment within which the agent operates, including the interface between agent and environment and the environment dynamic model. For more information, see “Create MATLAB Reinforcement Learning Environments” on page 2-2 and “Create Simulink Reinforcement Learning Environments” on page 2-8.
3. **Define reward** — Specify the reward signal that the agent uses to measure its performance against the task goals and how to calculate this signal from the environment. For more information, see “Define Reward Signals” on page 2-14.
4. **Create agent** — Create the agent, which includes defining a policy approximator (actor) an value function approximator (critic) and configuring the agent learning algorithm. For more information, see “Create Policies and Value Functions” on page 4-2 and “Reinforcement Learning Agents” on page 3-2.
5. **Train agent** — Train the agent approximators using the defined environment, reward, and agent learning algorithm. For more information, see “Train Reinforcement Learning Agents” on page 5-3.
6. **Validate agent** — Evaluate the performance of the trained agent by simulating the agent and environment together. For more information, see “Train Reinforcement Learning Agents” on page 5-3.

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

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

- Training settings
- Learning algorithm configuration
- Policy and value function (actor and critic) approximators
- Reward signal definition
- Action and observation signals
- Environment dynamics

## See Also

### Related Examples

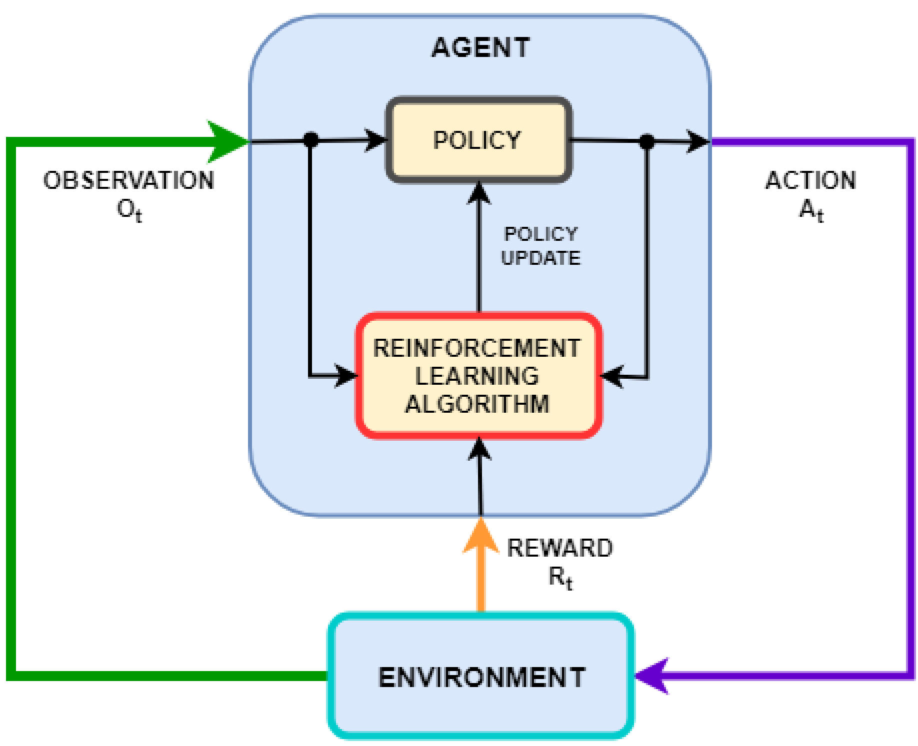
- “Create Simulink Environment and Train Agent” on page 1-20

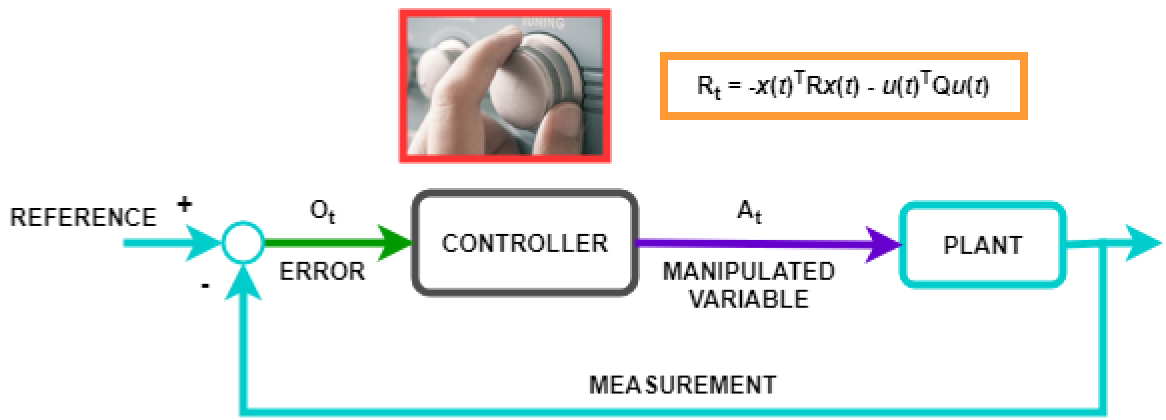
### More About

- “Reinforcement Learning for Control Systems Applications” on page 1-6

# Reinforcement Learning for Control Systems Applications

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




| Reinforcement Learning | Control Systems |
| --- | --- |
| Policy | Controller |

<table>
<tr><th>Reinforcement Learning</th><th>Control Systems</th></tr>
<tr><td>Environment</td><td>Everything that is not the controller — In the preceding diagram, the environment includes the plant, the reference signal, and the calculation of the error. In general, the environment can also include additional elements, such as:<ul><li>Measurement noise</li><li>Disturbance signals</li><li>Filters</li><li>Analog-to-digital and digital-to-analog converters</li></ul></td></tr>
<tr><td>Observation</td><td>Any measurable value from the environment that is visible to the agent — In the preceding diagram, the controller can see the error signal from the environment. You can also create agents that observe, for example, the reference signal, measurement signal, and measurement signal rate of change.</td></tr>
<tr><td>Action</td><td>Manipulated variables or control actions</td></tr>
<tr><td>Reward</td><td>Function of the measurement, error signal, or some other performance metric — For example, you can implement reward functions that minimize the steady-state error while minimizing control effort. When control specifications such as cost and constraint functions are available, you can use <code>generateRewardFunction</code> to generate a reward function from an MPC object or model verification blocks. You can then use the generated reward function as a starting point for reward design, for example by changing the weights or penalty functions.</td></tr>
<tr><td>Learning Algorithm</td><td>Adaptation mechanism of an adaptive controller</td></tr>
</table>

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

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

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

## See Also

## More About

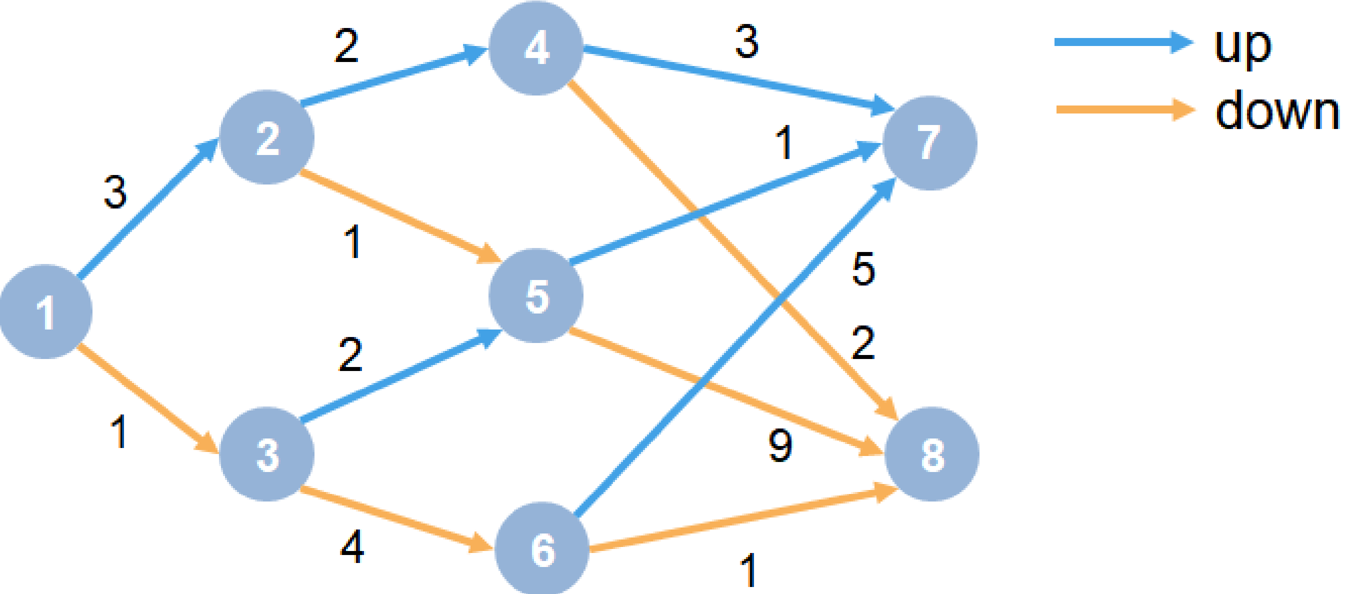
- “What Is Reinforcement Learning?” on page 1-3

- “Create MATLAB Reinforcement Learning Environments” on page 2-2
- “Create Simulink Reinforcement Learning Environments” on page 2-8
- “Define Reward Signals” on page 2-14

# Train Reinforcement Learning Agent in MDP Environment

This example shows how to train a Q-learning agent to solve a generic Markov decision process (MDP) environment. For more information on these agents, see "Q-Learning Agents" on page 3-17.

The MDP environment has the following graph.




Here:

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

### Create MDP Environment

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

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

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

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

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

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

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

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

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

Specify states "s7" and "s8" as terminal states of the MDP.

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

Create the reinforcement learning MDP environment for this process model.

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

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

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create Q-Learning Agent

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

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
qTable = rlTable(obsInfo, actInfo);
qFunction = rlQValueFunction(qTable, obsInfo, actInfo);
qOptions = rlOptimizerOptions(LearnRate=1);
```

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

```
agentOpts = rlQAgentOptions;
agentOpts.DiscountFactor = 1;
agentOpts.EpsilonGreedyExploration.Epsilon = 0.9;
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 0.01;
agentOpts.CriticOptimizerOptions = qOptions;
qAgent = rlQAgent(qFunction,agentOpts); %#ok<NASGU>
```

### Train Q-Learning Agent

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

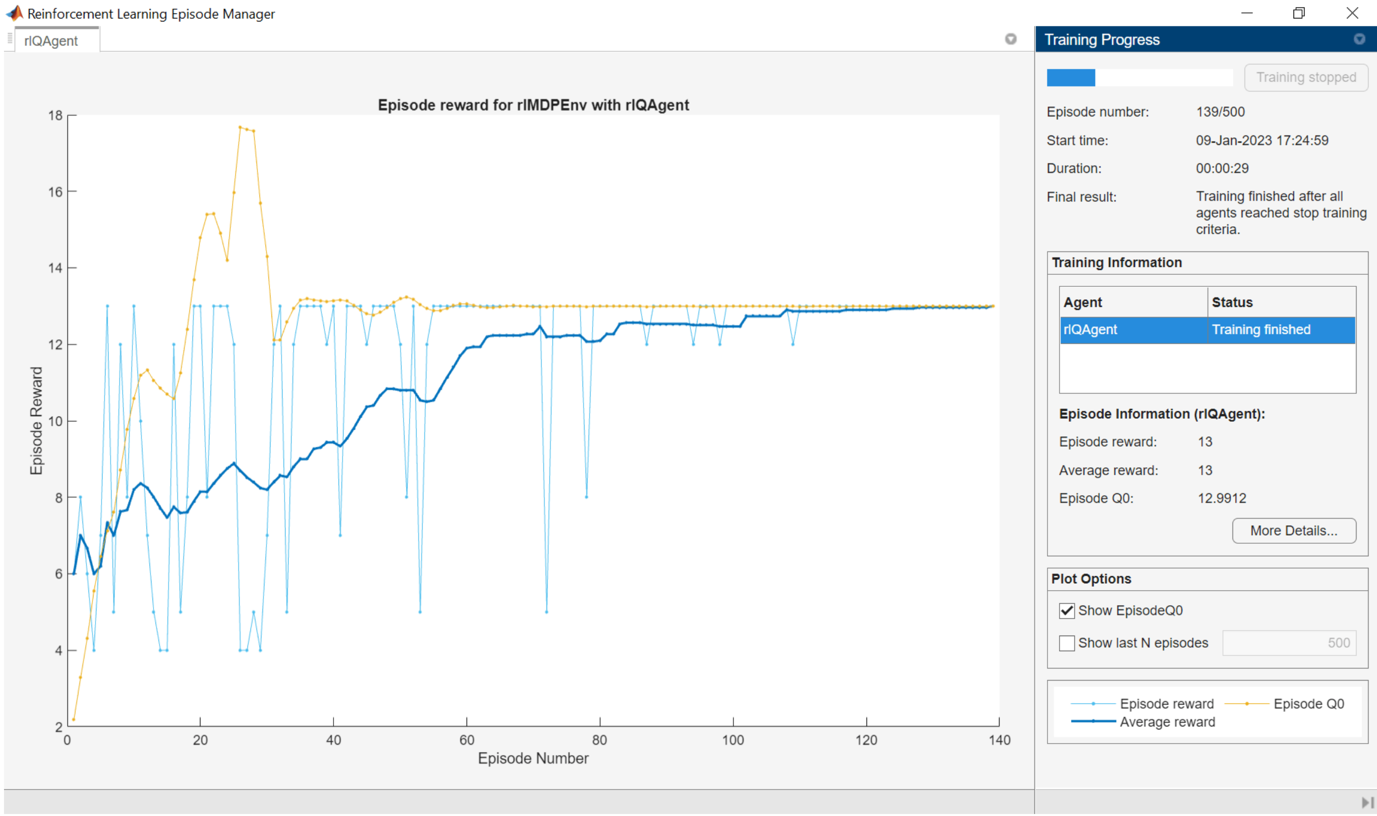
- Train for at most 500 episodes, with each episode lasting at most 50 time steps.
- Stop training when the agent receives an average cumulative reward greater than 10 over 30 consecutive episodes.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions;
trainOpts.MaxStepsPerEpisode = 50;
trainOpts.MaxEpisodes = 500;
trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = 13;
trainOpts.ScoreAveragingWindowLength = 30;
```

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

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(qAgent,env,trainOpts); %#ok<UNRCH>
else
    % Load pretrained agent for the example.
    load("genericMDPQAgent.mat","qAgent");
end
```




### Validate Q-Learning Results

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

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

```
cumulativeReward = 13
```

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

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

```
ans = 8x2 single matrix

   12.9912   11.6621
    8.2141    9.9950
   10.8645    4.0414
    4.8017   -1.6150
    5.1975    8.9975
    5.8058   -0.2353
         0         0
         0         0
```

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

```
TrueTableValues = 8×2

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

## See Also

**Functions**
`createMDP`

**Objects**
`rlMDPEnv`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About

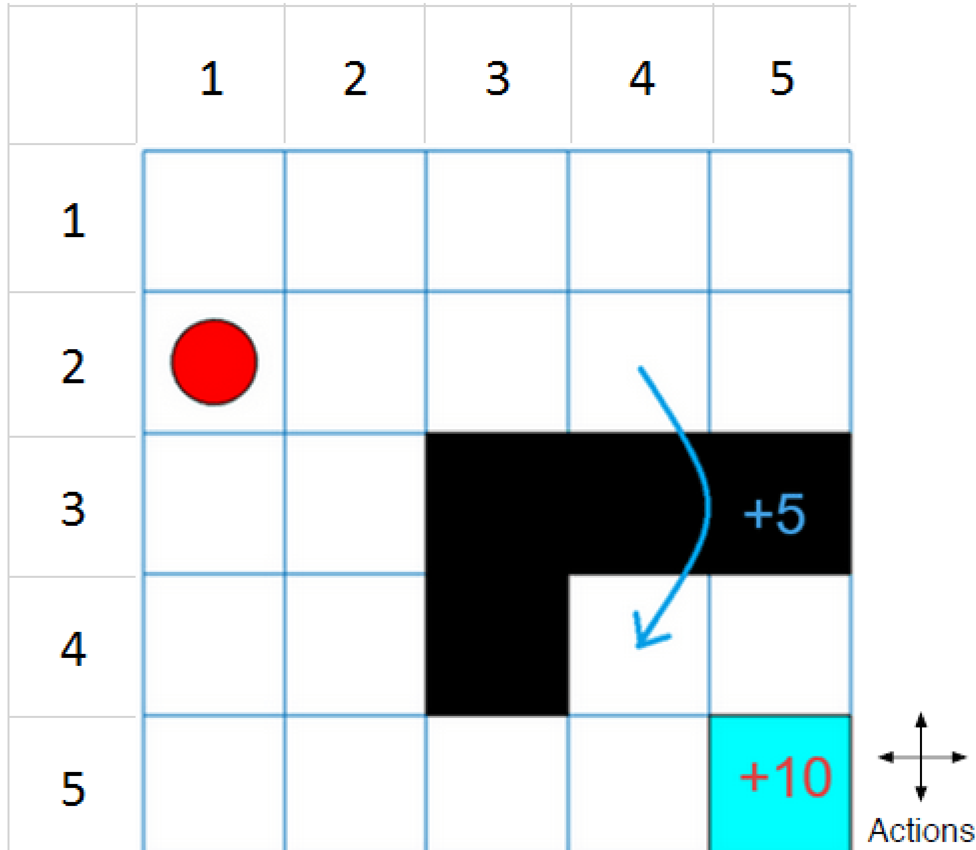
- “Reinforcement Learning Agents” on page 3-2

# Train Reinforcement Learning Agent in Basic Grid World

This example shows how to solve a grid world environment using reinforcement learning by training Q-learning and SARSA agents. For more information on these agents, see “Q-Learning Agents” on page 3-17 and “SARSA Agents” on page 3-20.

This grid world environment has the following configuration and rules:

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




### Create Grid World Environment

Create the basic grid world environment.

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

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

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

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create Q-Learning Agent

To create a Q-learning agent, first create a Q table using the observation and action specifications from the grid world environment. Set the learning rate of the optimizer to `0.01`.

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

To approximate the Q-value function within the agent, create a `rlQValueFunction` approximator object, using the table and the environment information.

```
qFcnAppx = rlQValueFunction(qTable, ...
    getObservationInfo(env), ...
    getActionInfo(env));
```

Next, create a Q-learning agent using the Q-value function.

```
qAgent = rlQAgent(qFcnAppx);
```

Configure agent options such as the epsilon-greedy exploration and the learning rate for the function approximator.

```
qAgent.AgentOptions.EpsilonGreedyExploration.Epsilon = .04;
qAgent.AgentOptions.CriticOptimizerOptions.LearnRate = 0.01;
```

For more information on creating Q-learning agents, see `rlQAgent` and `rlQAgentOptions`.

### Train Q-Learning Agent

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

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

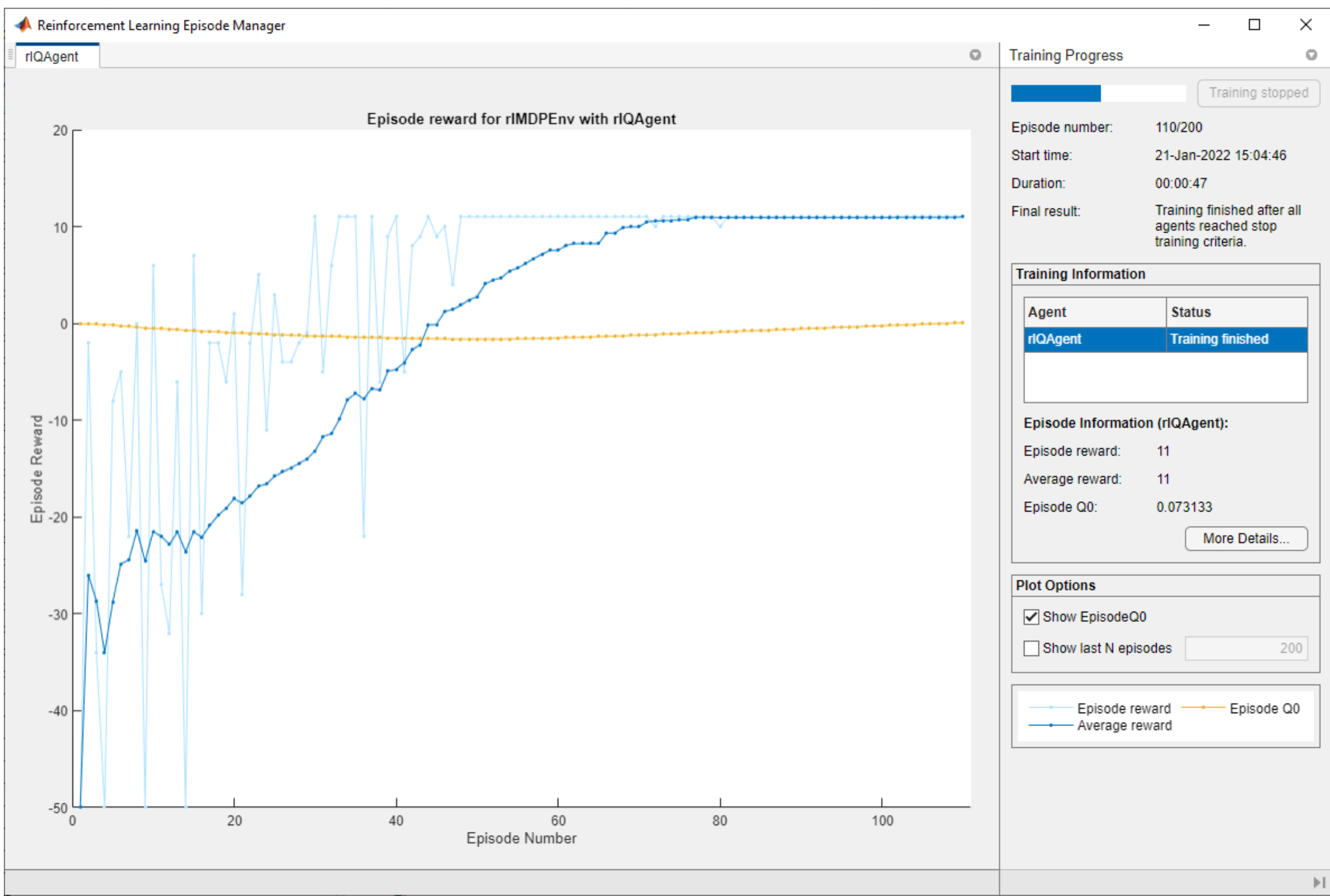
For more information, see `rlTrainingOptions`.

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

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

```
doTraining = false;

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

The **Episode Manager** window opens and displays the training progress.




## Validate Q-Learning Results

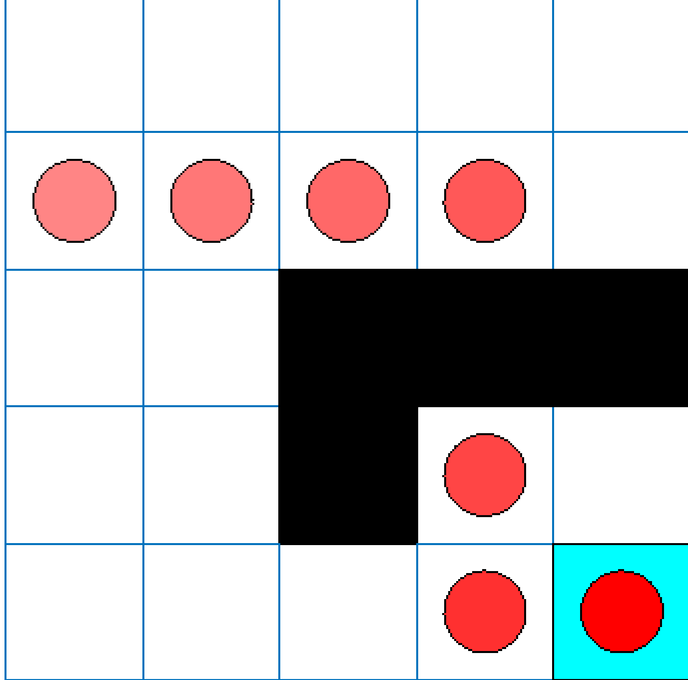
To validate the training results, simulate the agent in the training environment.

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

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

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

```
sim(qAgent,env)
```



The agent trace shows that the agent successfully finds the jump from cell [2,4] to cell [4,4].

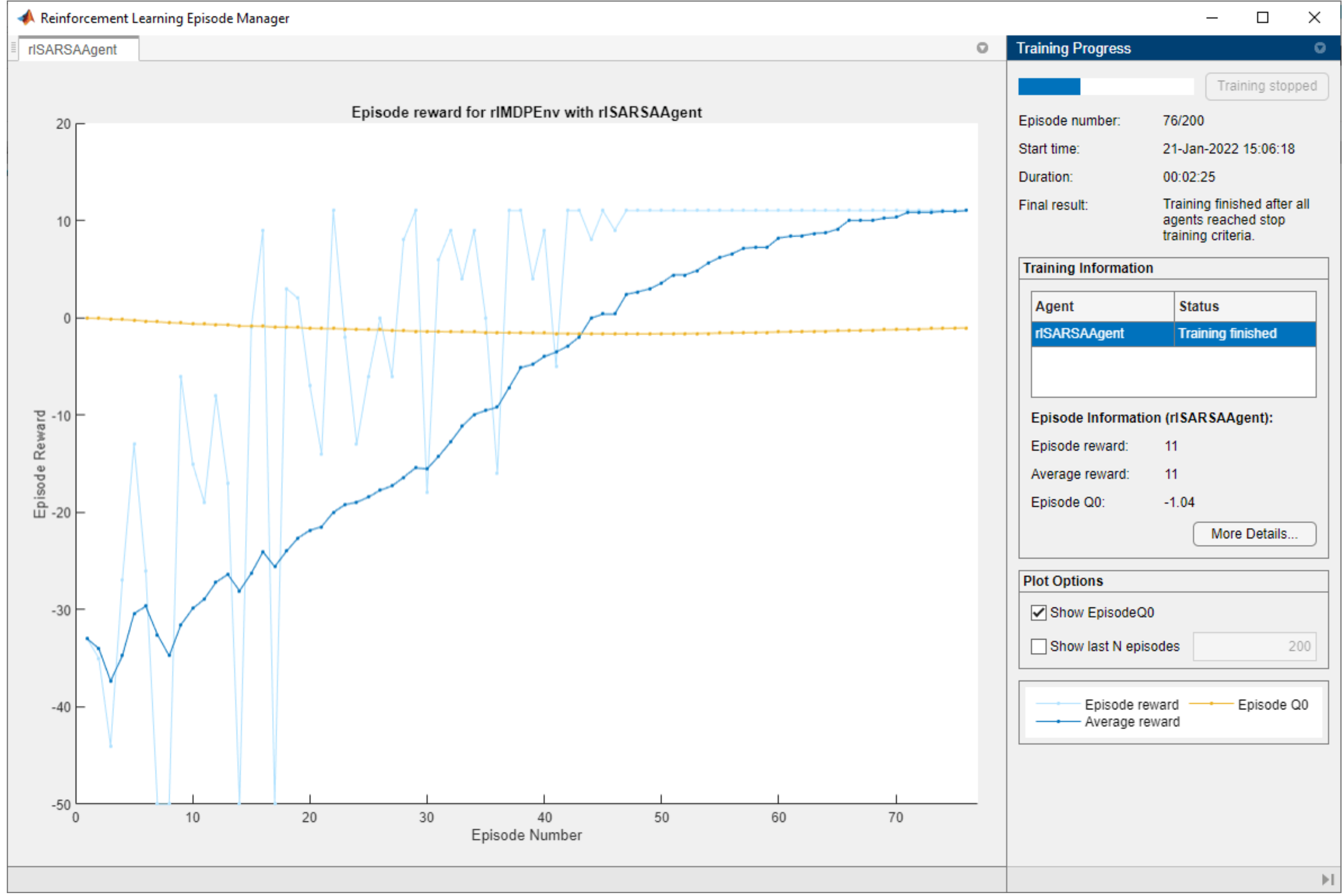
### Create and Train SARSA Agent

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

```
sarsaAgent = rlSARSAAgent(qFcnAppx);
sarsaAgent.AgentOptions.EpsilonGreedyExploration.Epsilon = .04;
sarsaAgent.AgentOptions.CriticOptimizerOptions.LearnRate = 0.01;
```

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

```
doTraining = false;

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



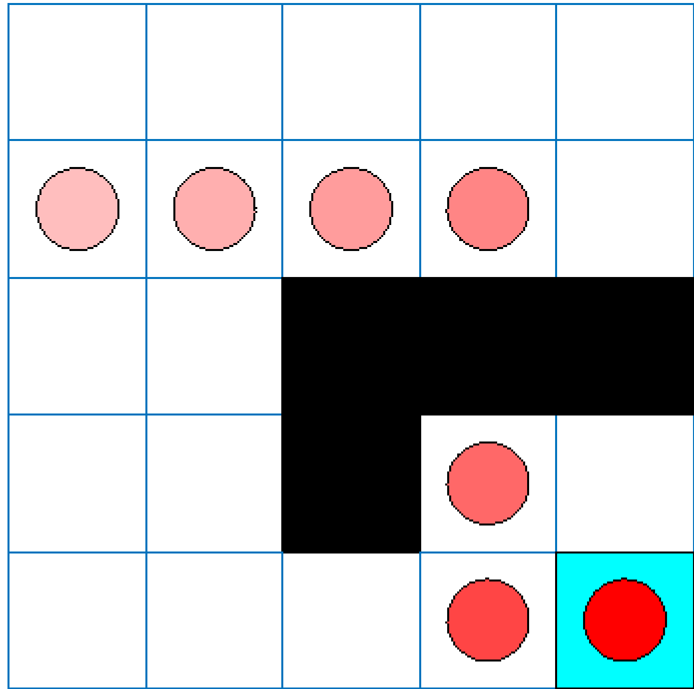

### Validate SARSA Training

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

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

Simulate the agent in the environment.

```
sim(sarsaAgent,env)
```



The SARSA agent finds the same grid world solution as the Q-learning agent.

## See Also

**Functions**
`createGridWorld`

**Objects**
`rlMDPEnv`

## Related Examples


- “Train Reinforcement Learning Agents” on page 5-3


## More About

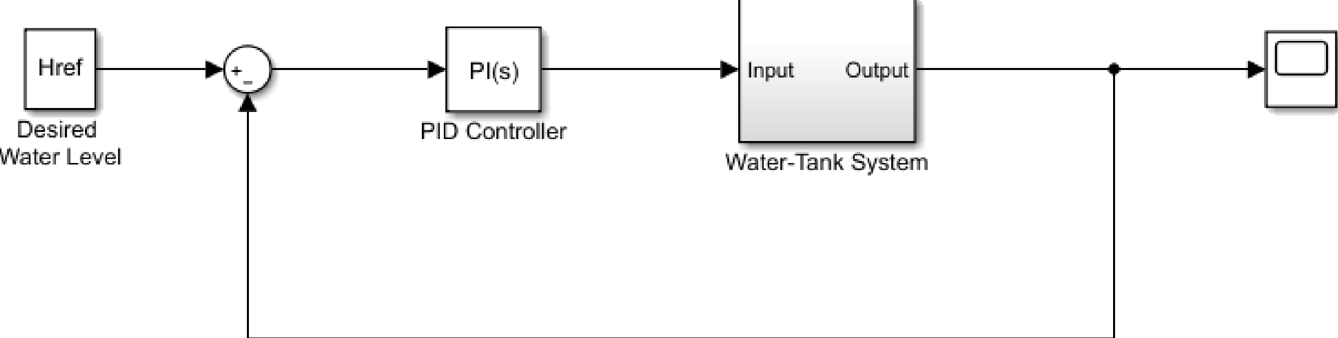

- “Reinforcement Learning Agents” on page 3-2

# Create Simulink Environment and Train Agent

This example shows how to convert the PI controller in the `watertank` Simulink® model to a reinforcement learning deep deterministic policy gradient (DDPG) agent. For an example that trains a DDPG agent in MATLAB®, see “Train DDPG Agent to Control Double Integrator System” on page 5-77.

### Water Tank Model

The original model for this example is the water tank model. The goal is to control the level of the water in the tank. For more information about the water tank model, see “watertank Simulink Model” (Simulink Control Design).

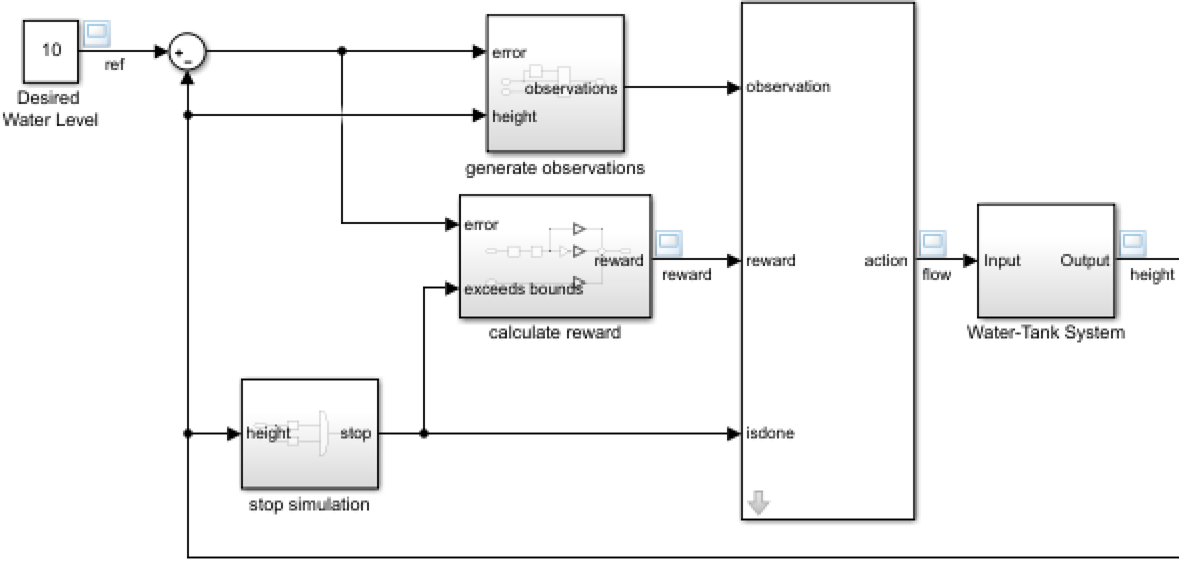



Modify the original model by making the following changes:

1. Delete the PID Controller.
2. Insert the RL Agent block.
3. Connect the observation vector $[\int e\,\mathrm{dt} \;\; e \;\; h]^T$, where $h$ is the height of the tank, $e = r - h$, and $r$ is the reference height.
4. Set up the reward $reward = 10(|e| < 0.1) - 1(|e| \geq 0.1) - 100(h \leq 0 || h \geq 20)$.
5. Configure the termination signal such that the simulation stops if $h \leq 0$ or $h \geq 20$.

The resulting model is `rlwatertank.slx`. For more information on this model and the changes, see “Create Simulink Reinforcement Learning Environments” on page 2-8.

```
open_system("rlwatertank")
```




### Create the Environment Interface

Creating an environment model includes defining the following:

- Action and observation signals that the agent uses to interact with the environment. For more information, see `rlNumericSpec` and `rlFiniteSetSpec`.
- Reward signal that the agent uses to measure its success. For more information, see “Define Reward Signals” on page 2-14.

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

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

actInfo = rlNumericSpec([1 1]);
actInfo.Name = "flow";
```

Build the environment interface object.

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

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

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

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

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create the Critic

DDPG agents use a parametrized Q-value function approximator to estimate the value of the policy. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for which receives the action from the state corresponding to the current observation, and following the policy thereafter).

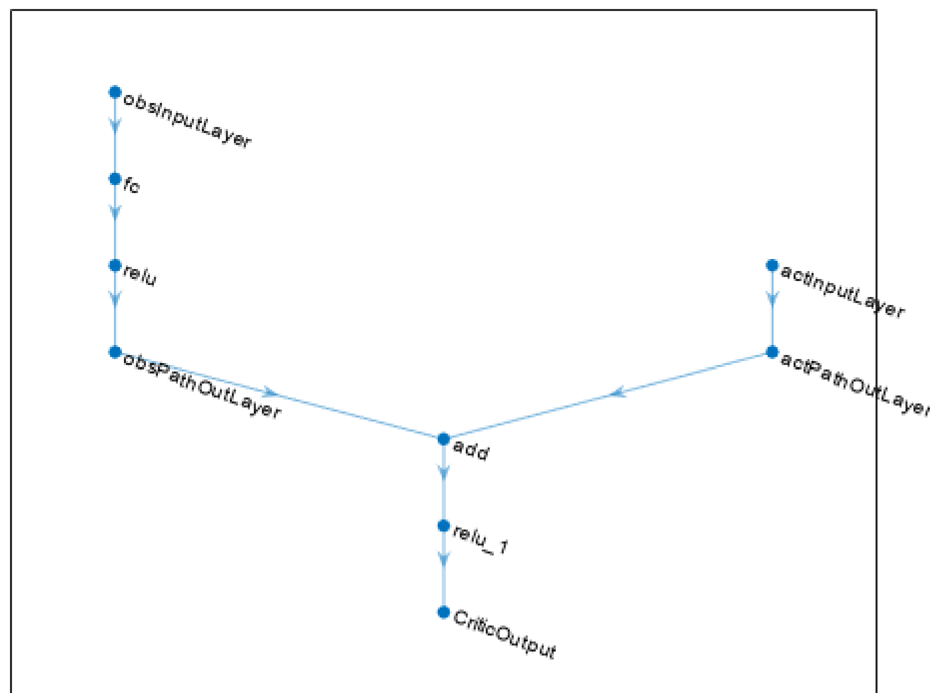
To model the parametrized Q-value function within the critic, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value).

Define each network path as an array of layer objects. Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel. Obtain the dimension of the observation and action spaces from the `obsInfo` and `actInfo` specifications.

```
% Observation path
obsPath = [
    featureInputLayer(obsInfo.Dimension(1),Name="obsInputLayer")
    fullyConnectedLayer(50)
    reluLayer
    fullyConnectedLayer(25,Name="obsPathOutLayer")];

% Action path
actPath = [
    featureInputLayer(actInfo.Dimension(1),Name="actInputLayer")
    fullyConnectedLayer(25,Name="actPathOutLayer")];

% Common path
commonPath = [
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(1,Name="CriticOutput")];

criticNetwork = layerGraph();
criticNetwork = addLayers(criticNetwork,obsPath);
criticNetwork = addLayers(criticNetwork,actPath);
criticNetwork = addLayers(criticNetwork,commonPath);

criticNetwork = connectLayers(criticNetwork, ...
    "obsPathOutLayer","add/in1");
criticNetwork = connectLayers(criticNetwork, ...
    "actPathOutLayer","add/in2");
```

View the critic network configuration.

```
figure
plot(criticNetwork)
```




Convert the network to a `dlnetwork` object and summarize its properties.

```
criticNetwork = dlnetwork(criticNetwork);
summary(criticNetwork)
```

```
   Initialized: true

   Number of learnables: 1.5k

   Inputs:
      1   'obsInputLayer'   3 features
      2   'actInputLayer'   1 features
```

Create the critic approximator object using the specified deep neural network, the environment specification objects, and the names if the network inputs to be associated with the observation and action channels.

```
critic = rlQValueFunction(criticNetwork, ...
    obsInfo,actInfo, ...
    ObservationInputNames="obsInputLayer", ...
    ActionInputNames="actInputLayer");
```

For more information on Q-value function objects, see `rlQValueFunction`.

Check the critic with a random input observation and action.

```
getValue(critic, ...
    {rand(obsInfo.Dimension)}, ...
    {rand(actInfo.Dimension)})
```

```
ans = single
    -0.1631
```

For more information on creating critics, see “Create Policies and Value Functions” on page 4-2.

### Create the Actor

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor.

A continuous deterministic actor implements a parametrized deterministic policy for a continuous action space. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects.

```
actorNetwork = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(3)
    tanhLayer
    fullyConnectedLayer(actInfo.Dimension(1))
    ];
```

Convert the network to a `dlnetwork` object and summarize its properties.

```
actorNetwork = dlnetwork(actorNetwork);
summary(actorNetwork)
```

```
   Initialized: true

   Number of learnables: 16

   Inputs:
      1   'input'   3 features
```

Create the actor approximator object using the specified deep neural network, the environment specification objects, and the name if the network input to be associated with the observation channel.

```
actor = rlContinuousDeterministicActor(actorNetwork,obsInfo,actInfo);
```

For more information, see `rlContinuousDeterministicActor`.

Check the actor with a random input observation.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-0.3408]}
```

For more information on creating critics, see "Create Policies and Value Functions" on page 4-2.

### Create the DDPG Agent

Create the DDPG agent using the specified actor and critic approximator objects.

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

For more information, see `rlDDPGAgent`.

Specify options for the agent, the actor, and the critic using dot notation.

```
agent.SampleTime = Ts;

agent.AgentOptions.TargetSmoothFactor = 1e-3;
agent.AgentOptions.DiscountFactor = 1.0;
agent.AgentOptions.MiniBatchSize = 64;
agent.AgentOptions.ExperienceBufferLength = 1e6;

agent.AgentOptions.NoiseOptions.Variance = 0.3;
agent.AgentOptions.NoiseOptions.VarianceDecayRate = 1e-5;

agent.AgentOptions.CriticOptimizerOptions.LearnRate = 1e-03;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
agent.AgentOptions.ActorOptimizerOptions.LearnRate = 1e-04;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold = 1;
```

Alternatively, you can specify the agent options using an `rlDDPGAgentOptions` object.

Check the agent with a random input observation.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-0.7926]}
```

### Train Agent

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

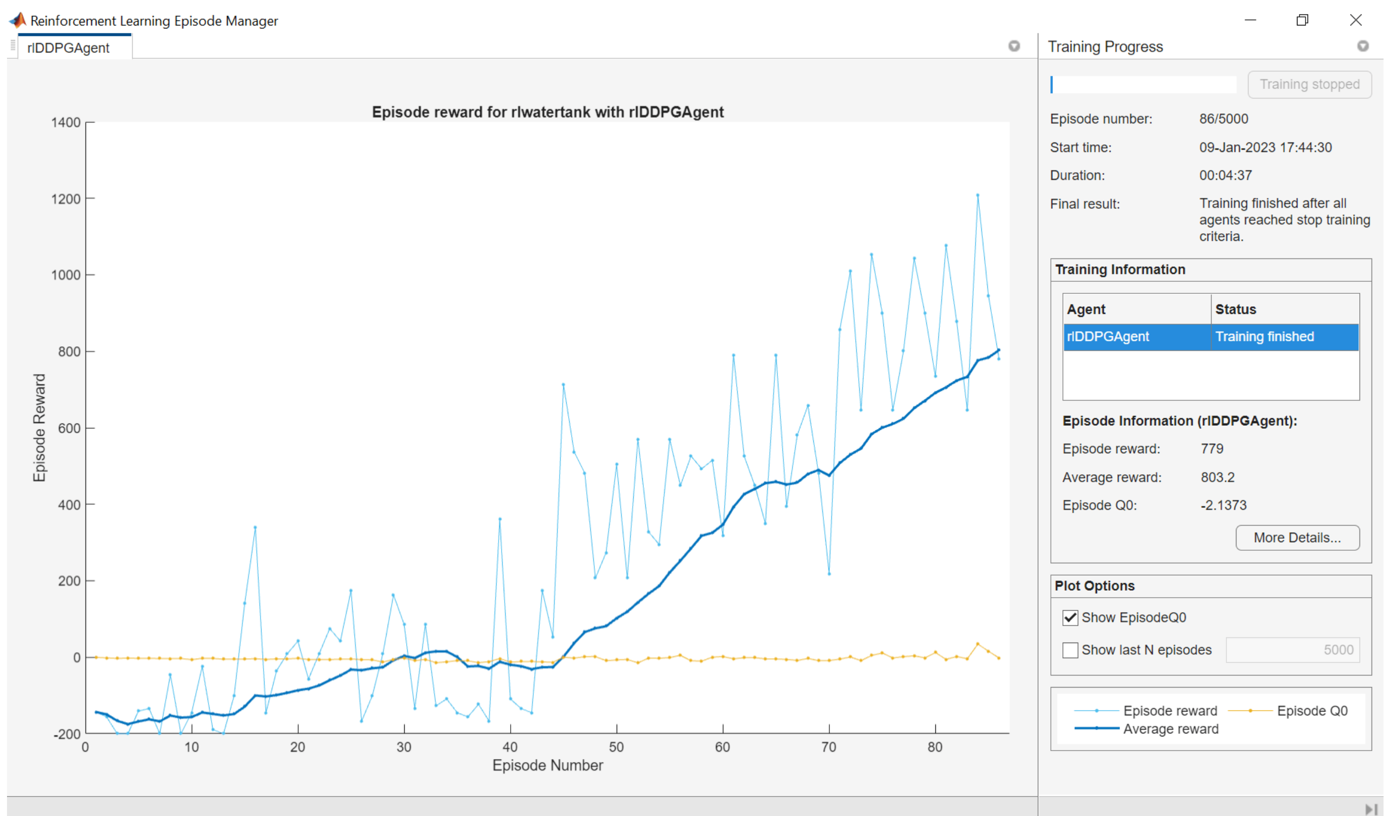
- Run each training for at most `5000` episodes. Specify that each episode lasts for at most `ceil(Tf/Ts)` (that is `200`) time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).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

For more information, see `rlTrainingOptions`.

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

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

```
doTraining = false;

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



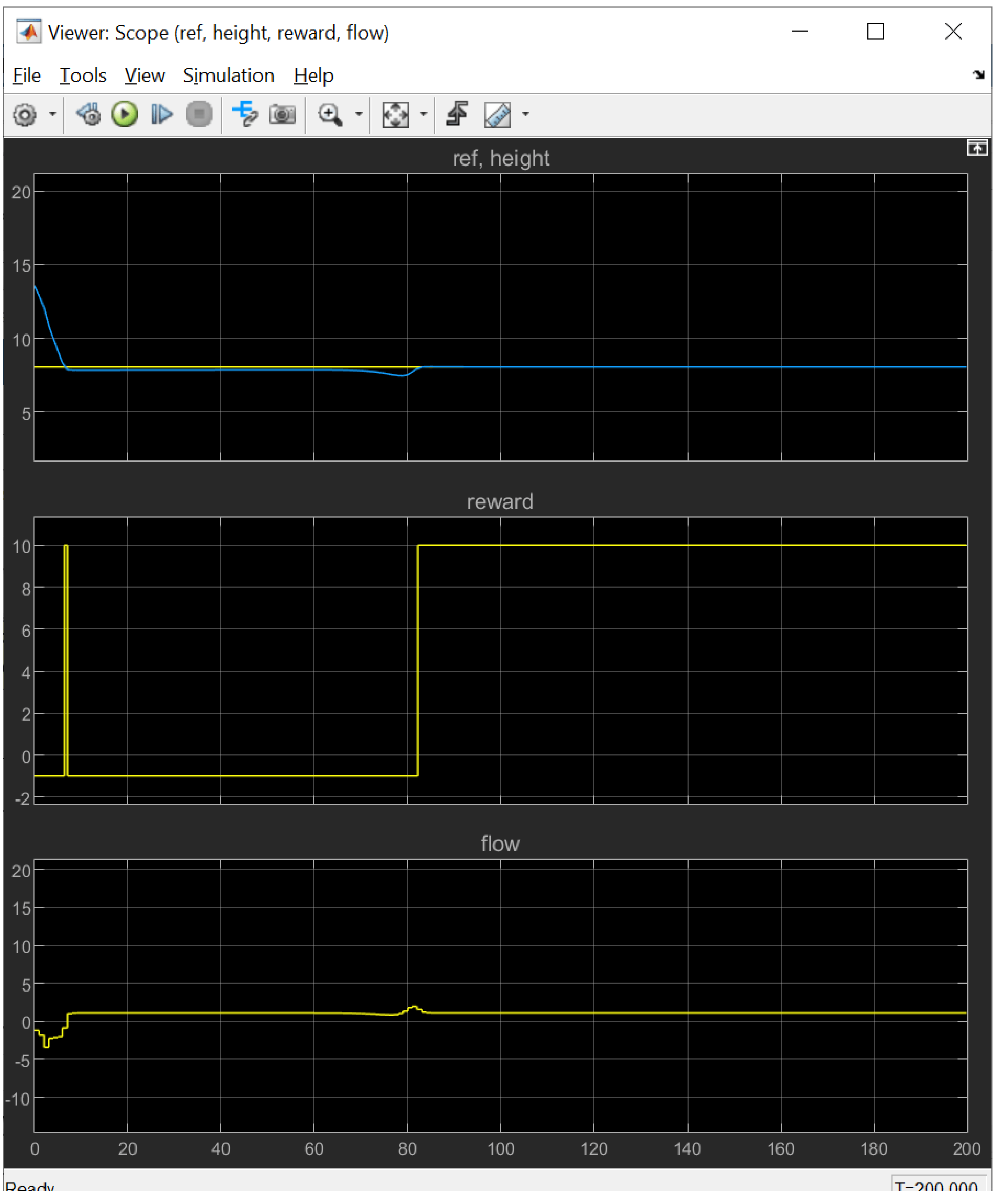

## Validate Trained Agent

Validate the learned agent against the model by simulation. Since the reset function randomizes the reference values, fix the random generator seed to ensure simulation reproducibility.

```
rng(1)
```

Simulate the agent within the environment, and return the experiences as output.

```
simOpts = rlSimulationOptions(MaxSteps=ceil(Tf/Ts),StopOnError="on");
experiences = sim(env,agent,simOpts);
```


Viewer: Scope (ref, height, reward, flow)
File Tools View Simulation Help
ref, height
reward
flow
0 20 40 60 80 100 120 140 160 180 200
Ready
T=200.000

#### Local Function

```
function in = localResetFcn(in)

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

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

end
```

## See Also

#### Functions
`train`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “Create Simulink Reinforcement Learning Environments” on page 2-8

# Create Environments

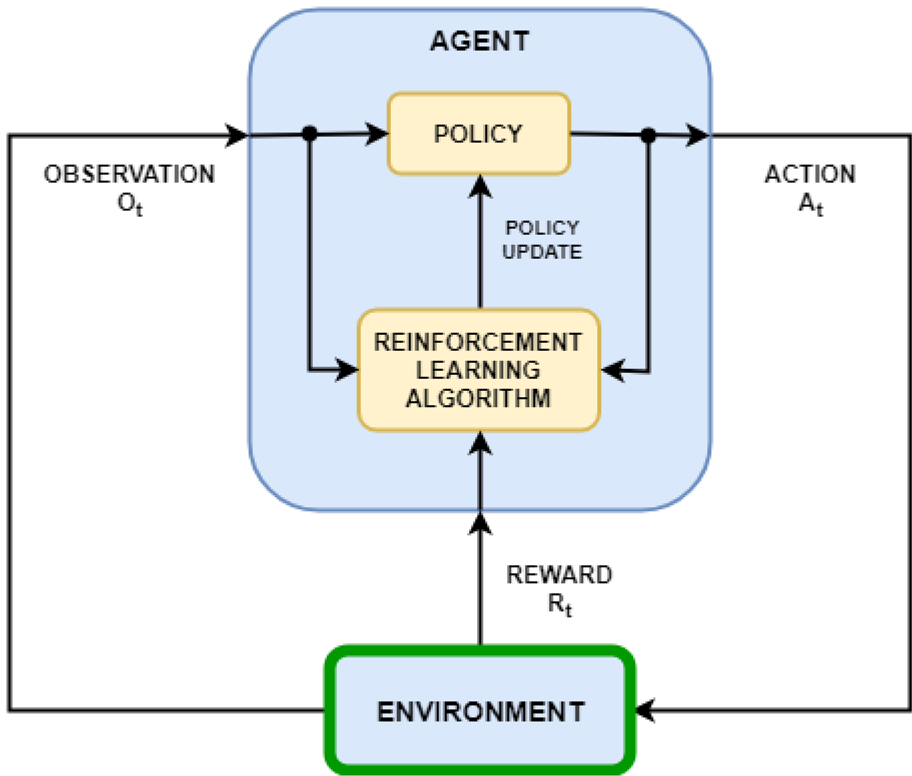

- “Create MATLAB Reinforcement Learning Environments” on page 2-2
- “Create or Import MATLAB Environments in Reinforcement Learning Designer” on page 2-5
- “Create Simulink Reinforcement Learning Environments” on page 2-8
- “Create or Import Simulink Environments in Reinforcement Learning Designer” on page 2-11
- “Define Reward Signals” on page 2-14
- “Load Predefined Grid World Environments” on page 2-17
- “Load Predefined Control System Environments” on page 2-23
- “Load Predefined Simulink Environments” on page 2-30
- “Create Custom Grid World Environments” on page 2-36
- “Create MATLAB Environment Using Custom Functions” on page 2-41
- “Create Custom MATLAB Environment from Template” on page 2-48
- “Water Tank Reinforcement Learning Environment Model” on page 2-55

# Create MATLAB Reinforcement Learning Environments

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

As shown in the following figure, the environment:

1. Receives actions from the agent.
2. Returns observations in response to the actions.
3. Generates a reward measuring how well the action contributes to achieving the task.




Creating an environment model involves defining:

- Action and observation signals that the agent uses to interact with the environment.
- A reward signal that the agent uses to measure its success. For more information, see "Define Reward Signals" on page 2-14.
- The environment initial condition and its dynamic behavior.

## Action and Observation Signals

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

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

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

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

## Predefined MATLAB Environments

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

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

For more information, see “Load Predefined Grid World Environments” on page 2-17 and “Load Predefined Control System Environments” on page 2-23.

## Custom MATLAB Environments

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

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

Once you create a custom environment object, you can train an agent in the same manner as in a predefined environment. For more information on training agents, see “Train Reinforcement Learning Agents” on page 5-3.

### Custom Grid Worlds

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

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

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

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

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

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

For more information on custom grid worlds, see "Create Custom Grid World Environments" on page 2-36.

### Specify Custom Functions

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

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

For more information, see "Create MATLAB Environment Using Custom Functions" on page 2-41.

### Create and Modify Template Environment

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

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

For more information, see "Create Custom MATLAB Environment from Template" on page 2-48.

## See Also

**Functions**
`rlPredefinedEnv` | `rlCreateEnvTemplate`

**Objects**
`rlFunctionEnv`

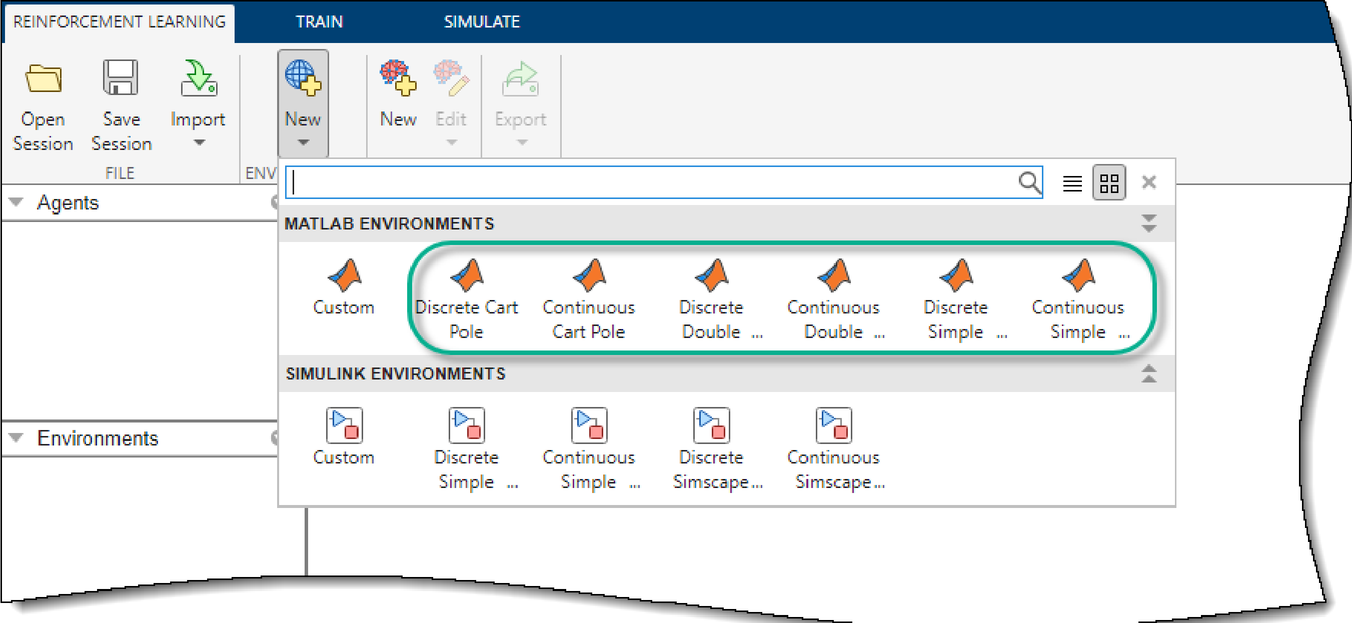
## More About

- "What Is Reinforcement Learning?" on page 1-3
- "Create or Import MATLAB Environments in Reinforcement Learning Designer" on page 2-5
- "Create Simulink Reinforcement Learning Environments" on page 2-8

# Create or Import MATLAB Environments in Reinforcement Learning Designer

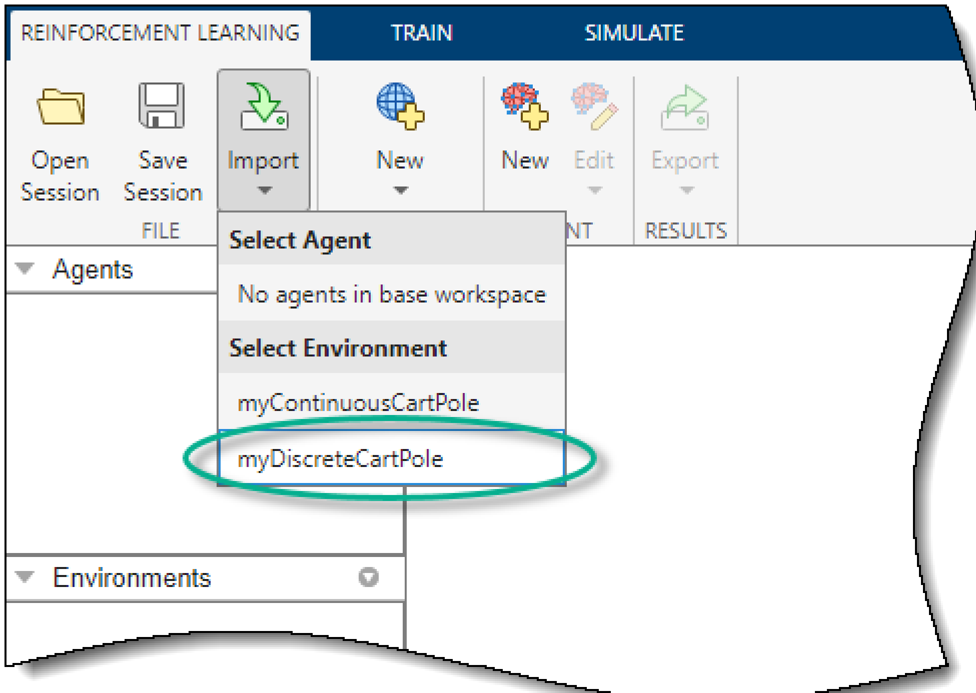
When training an agent using the **Reinforcement Learning Designer** app, you can create a predefined MATLAB environment from within the app or import a custom environment.

To create a predefined environment, on the **Reinforcement Learning** tab, in the **Environment** section, click **New**. Then, under **MATLAB Environments**, select one of the predefined environments.

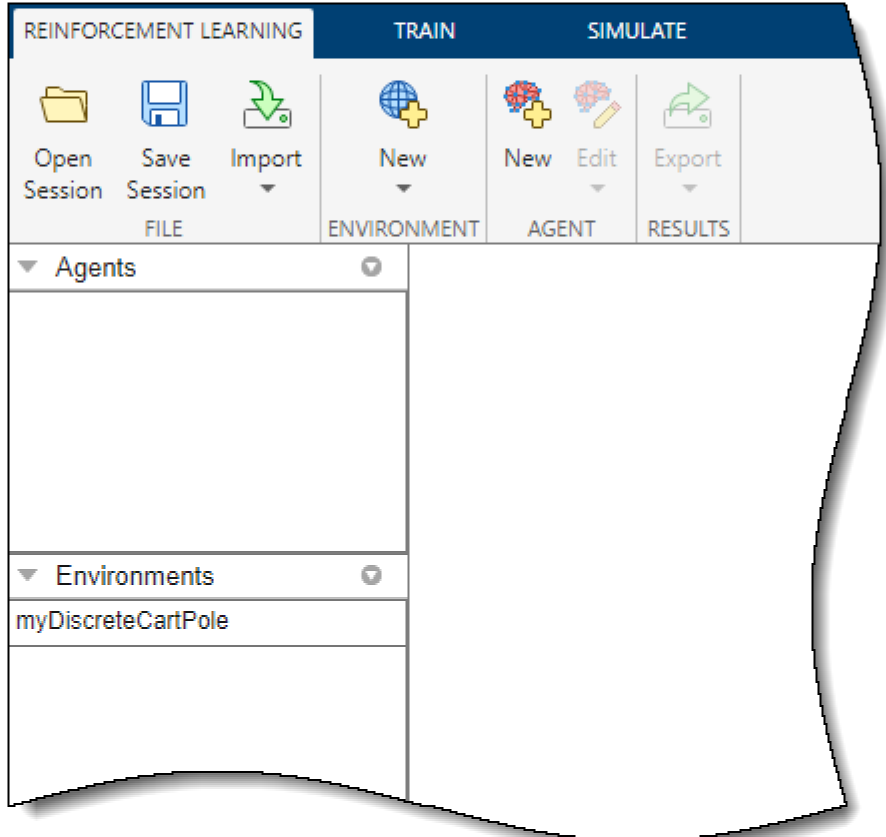



To use a custom environment, you must first create the environment at the MATLAB command line and then import the environment into **Reinforcement Learning Designer**. For more information on creating such an environment, see “Create MATLAB Reinforcement Learning Environments” on page 2-2.

Once you create a custom environment using one of the methods described in the preceding section, import the environment into **Reinforcement Learning Designer**. On the **Reinforcement Learning** tab, click **Import**. Then, under **Select Environment**, select the environment.




Once you have created or imported an environment, the app adds the environment to the **Environments** pane.




Once you have created an environment, you can create an agent to train in that environment. For more information, see “Create Agents Using Reinforcement Learning Designer” on page 3-9.

## See Also

**Apps**
**Reinforcement Learning Designer**

## Related Examples

- “Design and Train Agent Using Reinforcement Learning Designer” on page 5-12

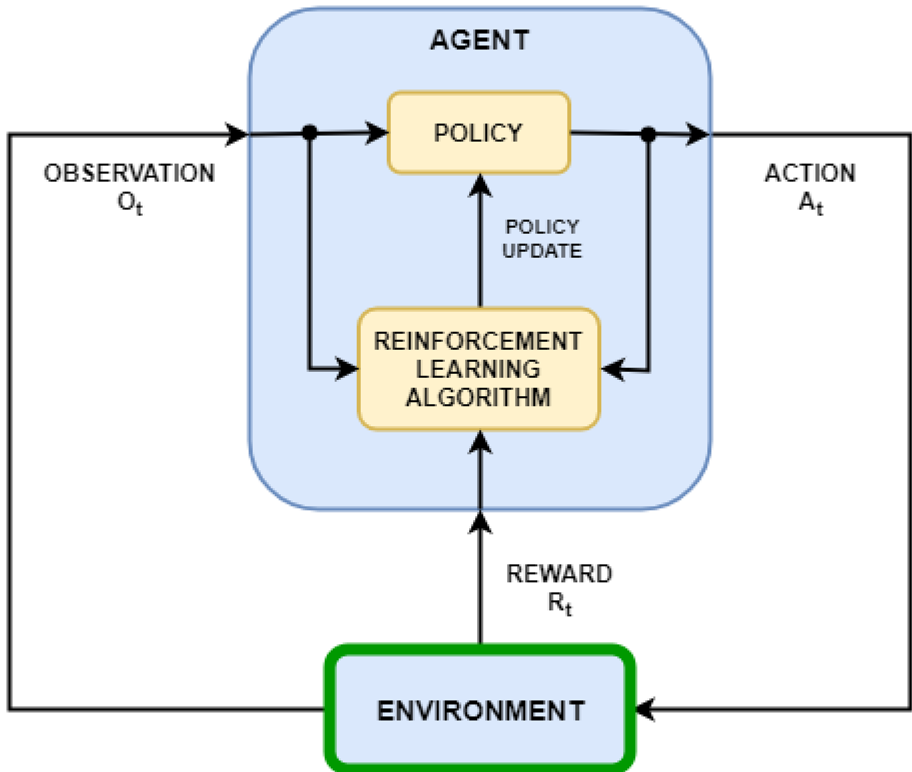
## More About

- “Create MATLAB Reinforcement Learning Environments” on page 2-2
- “Create or Import Simulink Environments in Reinforcement Learning Designer” on page 2-11
- “Create Agents Using Reinforcement Learning Designer” on page 3-9

# Create Simulink Reinforcement Learning Environments

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

1. Receives actions from the agent.
2. Outputs observations in response to the actions.
3. Generates a reward measuring how well the action contributes to achieving the task.




Creating an environment model includes defining the following:

- Action and observation signals that the agent uses to interact with the environment.
- Reward signal that the agent uses to measure its success. For more information, see “Define Reward Signals” on page 2-14.
- Environment dynamic behavior.

## Action and Observation Signals

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

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

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

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

## Predefined Simulink Environments

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

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

For more information, see “Load Predefined Simulink Environments” on page 2-30.

## Custom Simulink Environments

To specify your own custom reinforcement learning environment, create a Simulink model with an RL Agent block. In this model, connect the action, observation, and reward signals to the RL Agent block. For an example, see “Water Tank Reinforcement Learning Environment Model” on page 2-55.

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

For the reward signal, construct a scalar signal in the model and connect this signal to the RL Agent block. For more information, see “Define Reward Signals” on page 2-14.

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

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

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

## See Also

**Functions**
`rlPredefinedEnv` | `rlSimulinkEnv` | `createIntegratedEnv`

## More About

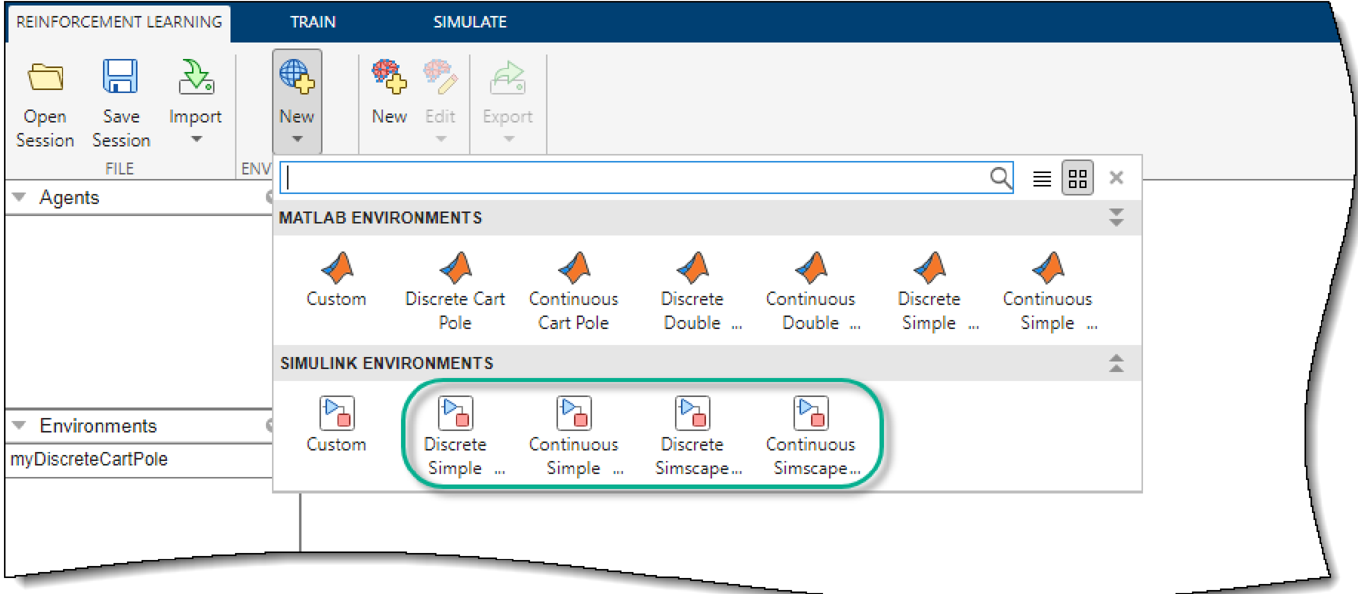
- “What Is Reinforcement Learning?” on page 1-3

- “Create or Import Simulink Environments in Reinforcement Learning Designer” on page 2-11
- “Create MATLAB Reinforcement Learning Environments” on page 2-2

# Create or Import Simulink Environments in Reinforcement Learning Designer

When training an agent using the **Reinforcement Learning Designer** app, you can create a predefined Simulink environment from within the app or import a custom environment.

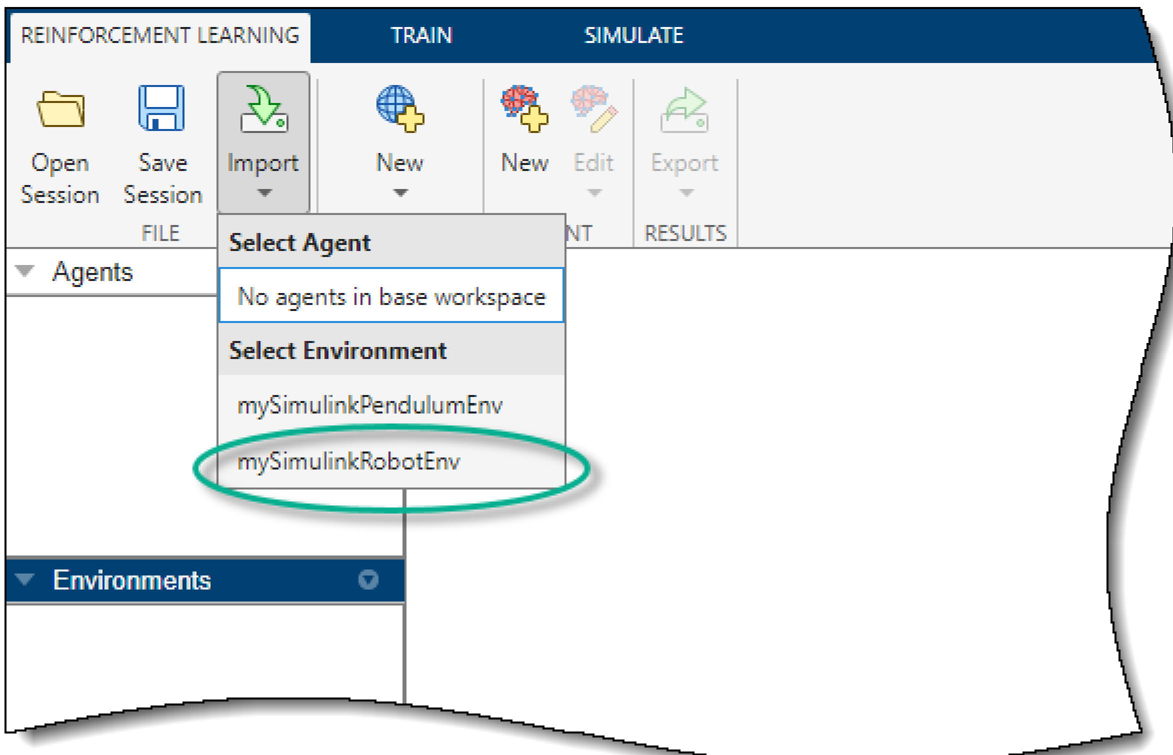
To create a predefined environment, on the **Reinforcement Learning** tab, in the **Environment** section, click **New**. Then, under **Simulink Environments**, select one of the predefined environments.



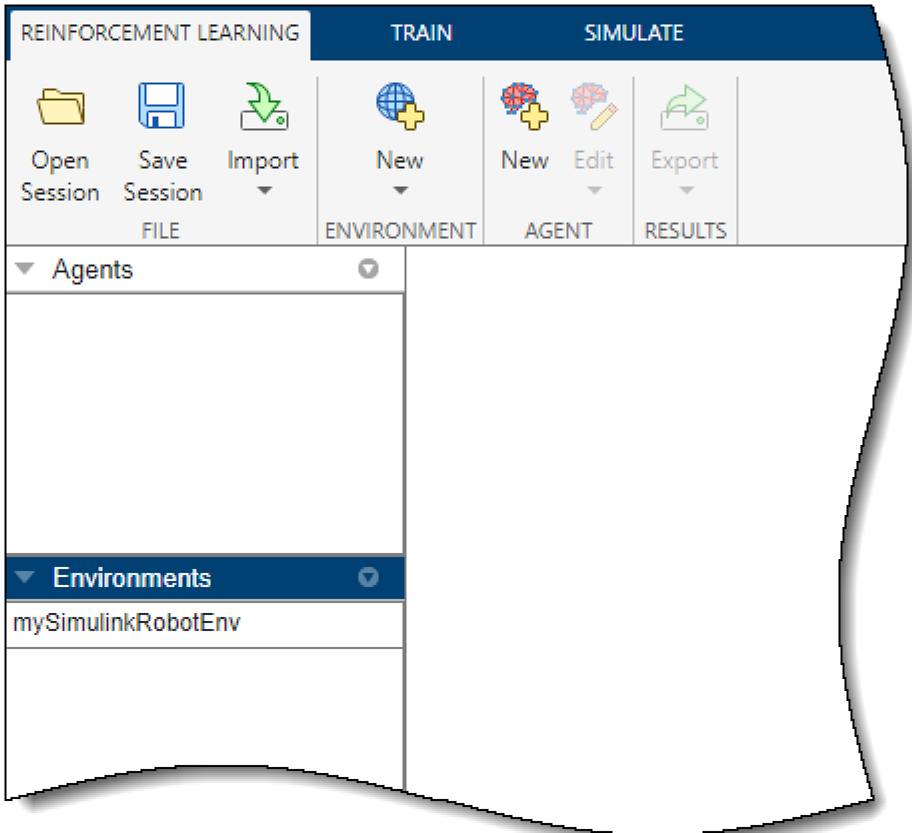

To use a custom environment, you must first create the environment at the MATLAB command line and then import the environment into **Reinforcement Learning Designer**. For more information on creating a Simulink environment, see “Create Simulink Reinforcement Learning Environments” on page 2-8.

For training and simulating Simulink environments, you must define all variables necessary for running the Simulink mode in the MATLAB workspace.

Once you create a custom environment using one of the methods described in the preceding section, import the environment into **Reinforcement Learning Designer**. On the **Reinforcement Learning** tab, click **Import**. Then, under **Select Environment**, select the environment.




Once you have created or imported an environment, the app adds the environment to the **Environments** pane.




Once you have created an environment, you can create an agent to train in that environment. For more information, see "Create Agents Using Reinforcement Learning Designer" on page 3-9.

## See Also

**Apps**
**Reinforcement Learning Designer**

## Related Examples

- "Design and Train Agent Using Reinforcement Learning Designer" on page 5-12

## More About

- "Create Simulink Reinforcement Learning Environments" on page 2-8
- "Create or Import MATLAB Environments in Reinforcement Learning Designer" on page 2-5
- "Create Agents Using Reinforcement Learning Designer" on page 3-9

# Define Reward Signals

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

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

For example, when an agent must perform a task for as long as possible, a common strategy is to provide a small positive reward for each time step that the agent successfully performs the task and a large penalty when the agent fails. This approach encourages longer training episodes while heavily discouraging episodes that fail. For an example that uses this approach, see “Train DQN Agent to Balance Cart-Pole System” on page 5-50.

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

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

For applications where control system specifications like cost functions and constraints are already available, you can also use generate rewards functions from such specifications.

## Continuous Rewards

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

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

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

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

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

This QR reward structure encourages driving $s$ to zero with minimal action effort. A QR-based reward structure is a good reward to choose for regulation or stationary point problems, such as pendulum swing-up or regulating the position of the double integrator. For training examples that use a QR reward, see “Train DQN Agent to Swing Up and Balance Pendulum” on page 5-89 and “Train DDPG Agent to Control Double Integrator System” on page 5-77.

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

## Discrete Rewards

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

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

## Mixed Rewards

In many cases, providing a mixed reward signal that has a combination of continuous and discrete reward components is beneficial. The discrete reward signal can be used to drive the system away from bad states, and the continuous reward signal can improve convergence by providing a smooth reward near target states. For example, in “Train DDPG Agent to Control Flying Robot” on page 5-172, the reward function has three components: $r_1$, $r_2$, and $r_3$.

$$r_1 = 10\left(\left(x_t^2 + y_t^2 + \theta_t^2\right) < 0.5\right)$$
$$r_2 = -100(|x_t| \geq 20 \,||\, |y_t| \geq 20)$$
$$r_3 = -\left(0.2(R_{t-1} + L_{t-1})^2 + 0.3(R_{t-1} - L_{t-1})^2 + 0.03x_t^2 + 0.03y_t^2 + 0.02\theta_t^2\right)$$
$$r = r_1 + r_2 + r_3$$

Here:

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

## Reward Generation from Control Specifications

For applications where a working control system already exists, specifications such as cost functions or constraints might already be available. In these cases, you can use `generateRewardFunction` to generate a reward function, coded in MATLAB, that can be used as a starting point for reward design. This function allows you to generate rewards from:

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

In both cases, when constraints are violated, a negative reward is calculated using penalty functions such as `exteriorPenalty` (default), `hyperbolicPenalty` or `barrierPenalty` functions.

Starting from the generated reward function, you can tune the cost and penalty weights, use a different penalty function, and then use the resulting reward function within an environment to train an agent.

## See Also

**Functions**
`generateRewardFunction` | `exteriorPenalty` | `hyperbolicPenalty` | `barrierPenalty`

## Related Examples

- "Generate Reward Function from a Model Predictive Controller for a Servomotor" on page 5-315
- "Generate Reward Function from a Model Verification Block for a Water Tank System" on page 5-327

## More About

- "Create MATLAB Reinforcement Learning Environments" on page 2-2
- "Create Simulink Reinforcement Learning Environments" on page 2-8

# Load Predefined Grid World Environments

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

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

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

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

For more information on the properties of grid world environments, see “Create Custom Grid World Environments” on page 2-36.

You can also load predefined MATLAB control system environments. For more information, see “Load Predefined Control System Environments” on page 2-23.

## Basic Grid World

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

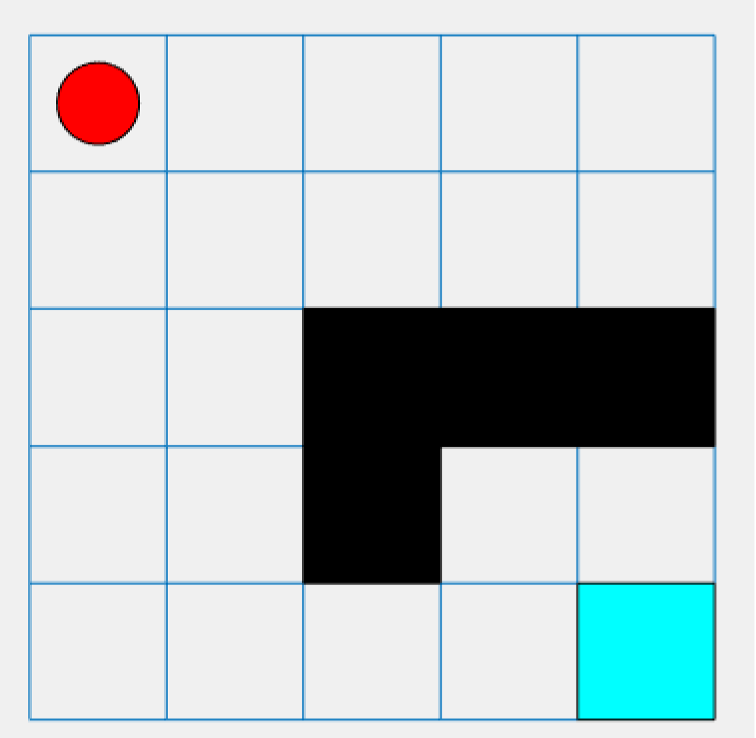
To create a basic grid world environment, use the `rlPredefinedEnv` function. This function creates an `rlMDPEnv` object representing the grid world.

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

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

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

```
plot(env)
```



#### Actions

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

#### Rewards

The agent receives the following rewards or penalties:

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

## Deterministic Waterfall Grid Worlds

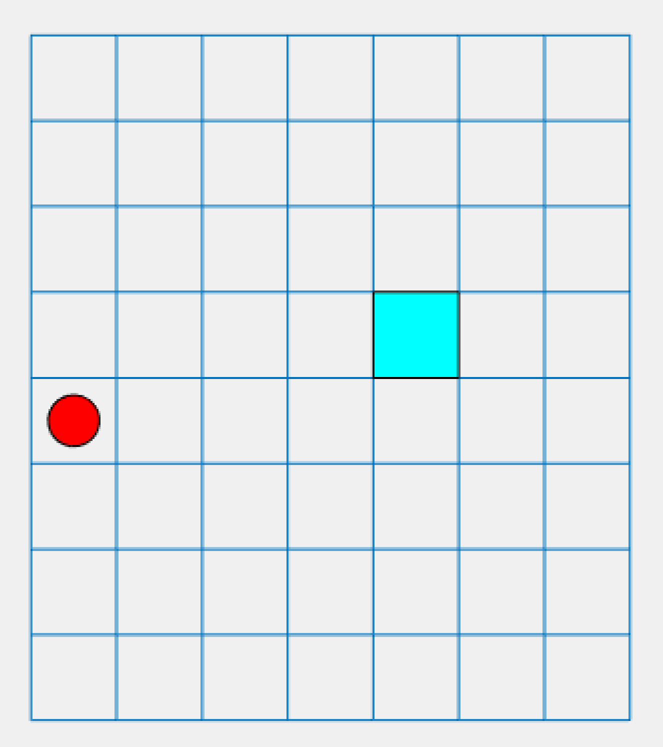
The deterministic waterfall grid world environment is a two-dimensional 8-by-7 grid with a starting location and terminal location. The environment includes a waterfall that pushes the agent toward the bottom of the grid. The goal of the agent is to move from the starting location to the terminal location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

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

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

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

```
plot(env)
```



**Actions**

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

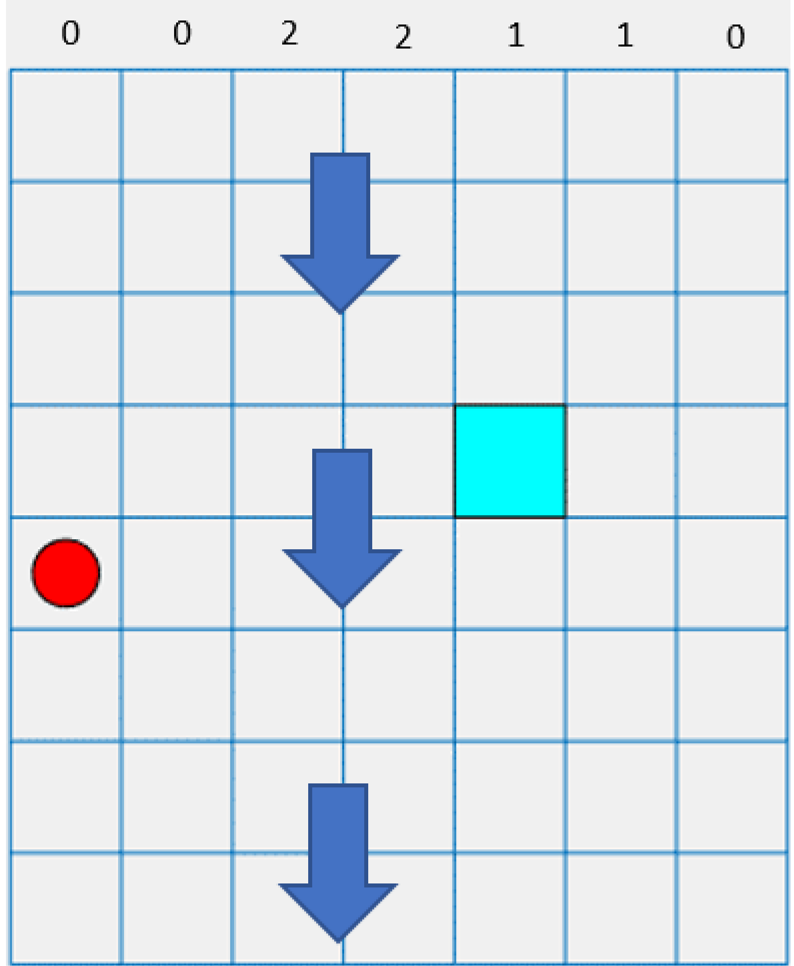
**Rewards**

The agent receives the following rewards or penalties:

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

**Waterfall Dynamics**

In this environment, a waterfall pushes the agent toward the bottom of the grid.




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

## Stochastic Waterfall Grid Worlds

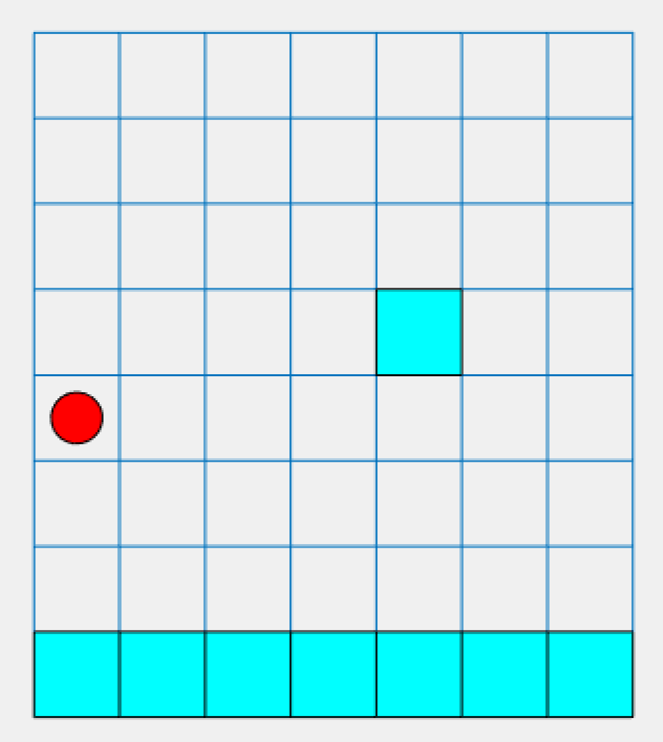
The stochastic waterfall grid world environment is a two-dimensional 8-by-7 grid with a starting location and terminal locations. The environment includes a waterfall that pushes the agent towards the bottom of the grid with a stochastic intensity. The goal of the agent is to move from the starting location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

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

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

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

```
plot(env)
```



#### Actions

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

#### Rewards

The agent receives the following rewards or penalties:

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

#### Waterfall Dynamics

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

## See Also

**Functions**
`rlPredefinedEnv` | `train`

**Objects**
`rlMDPEnv`

## Related Examples

- "Train Reinforcement Learning Agents" on page 5-3

## More About

- "Create MATLAB Reinforcement Learning Environments" on page 2-2
- "Load Predefined Control System Environments" on page 2-23

# Load Predefined Control System Environments

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

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

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

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

You can also load predefined MATLAB grid world environments. For more information, see “Load Predefined Grid World Environments” on page 2-17.

## Cart-Pole Environments

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

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

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

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

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

- Discrete action space

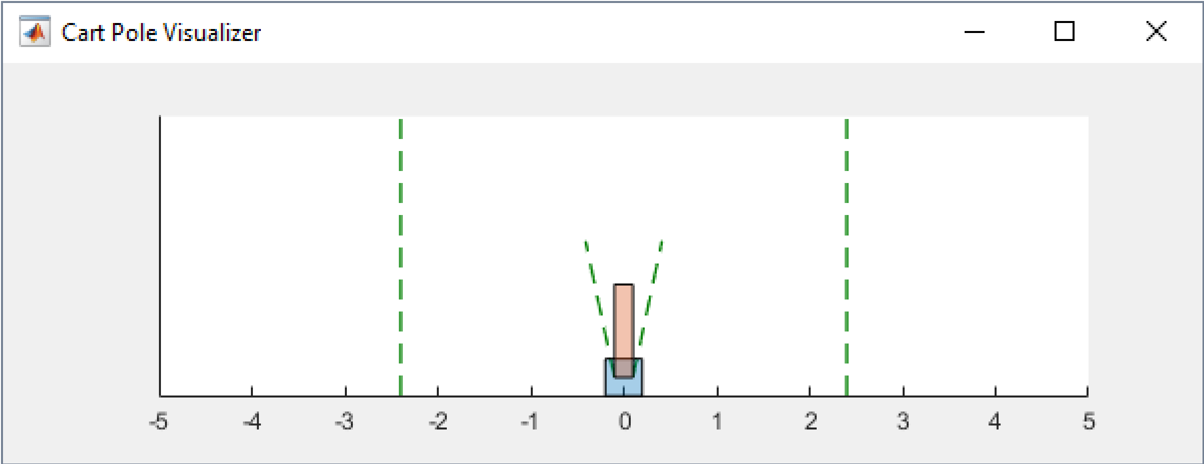
```
env = rlPredefinedEnv('CartPole-Discrete');
```

- Continuous action space

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

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

```
plot(env)
```




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

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

- "Train DQN Agent to Balance Cart-Pole System" on page 5-50
- "Train PG Agent to Balance Cart-Pole System" on page 5-57
- "Train AC Agent to Balance Cart-Pole System" on page 5-63

### Environment Properties

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

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

#### Actions

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

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

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

#### Observations

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

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

#### Reward

The reward signal for this environment consists of two components.

- A positive reward for each time step that the pole is balanced, that is, the cart and pole both remain within their specified threshold ranges. This reward accumulates over the entire training episode. To control the size of this reward, use the `RewardForNotFalling` property of the environment.
- A one-time negative penalty if either the pole or cart moves outside of their threshold range. At this point, the training episode stops. To control the size of this penalty, use the `PenaltyForFalling` property of the environment.

## Double Integrator Environments

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

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

- The mass moves beyond a given threshold from the origin.
- The norm of the state vector is less than a given threshold.

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

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

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

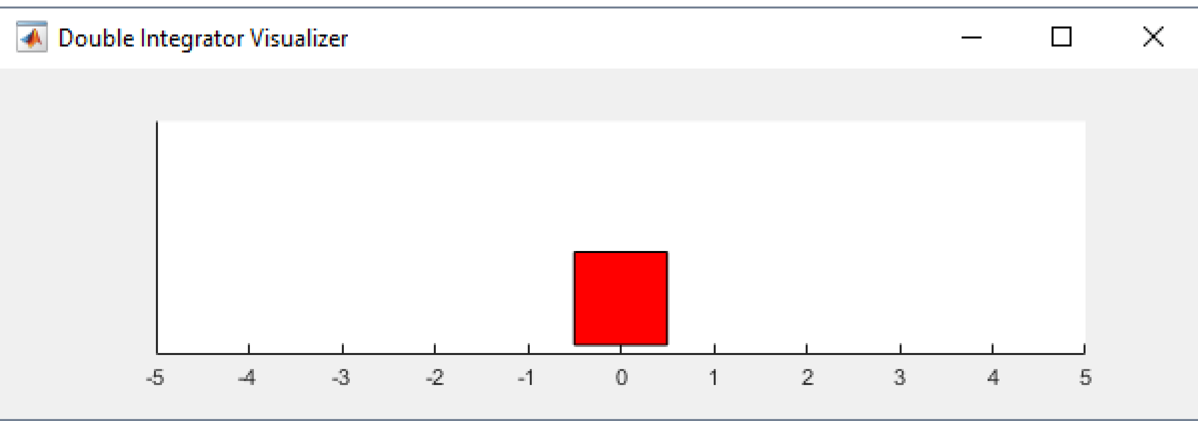
- Discrete action space

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

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

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

```
plot(env)
```




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

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

- “Train DDPG Agent to Control Double Integrator System” on page 5-77
- “Train PG Agent with Baseline to Control Double Integrator System” on page 5-70

**Environment Properties**

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

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

### Actions

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

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

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

### Observations

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

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

### Reward

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

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

Here:

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

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

## Simple Pendulum Environments with Image Observation

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

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

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

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

- Discrete action space

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

- Continuous action space

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

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

- “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation” on page 5-141
- “Create DQN Agent Using Deep Network Designer and Train Using Image Observations” on page 5-152

#### Environment Properties

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

#### Actions

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

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

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

#### Observations

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

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

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

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

#### Reward

The reward signal for this environment is

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

Here:

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

## See Also

#### Functions
`rlPredefinedEnv` | `train`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “Create MATLAB Reinforcement Learning Environments” on page 2-2
- “Load Predefined Grid World Environments” on page 2-17

# Load Predefined Simulink Environments

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

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

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

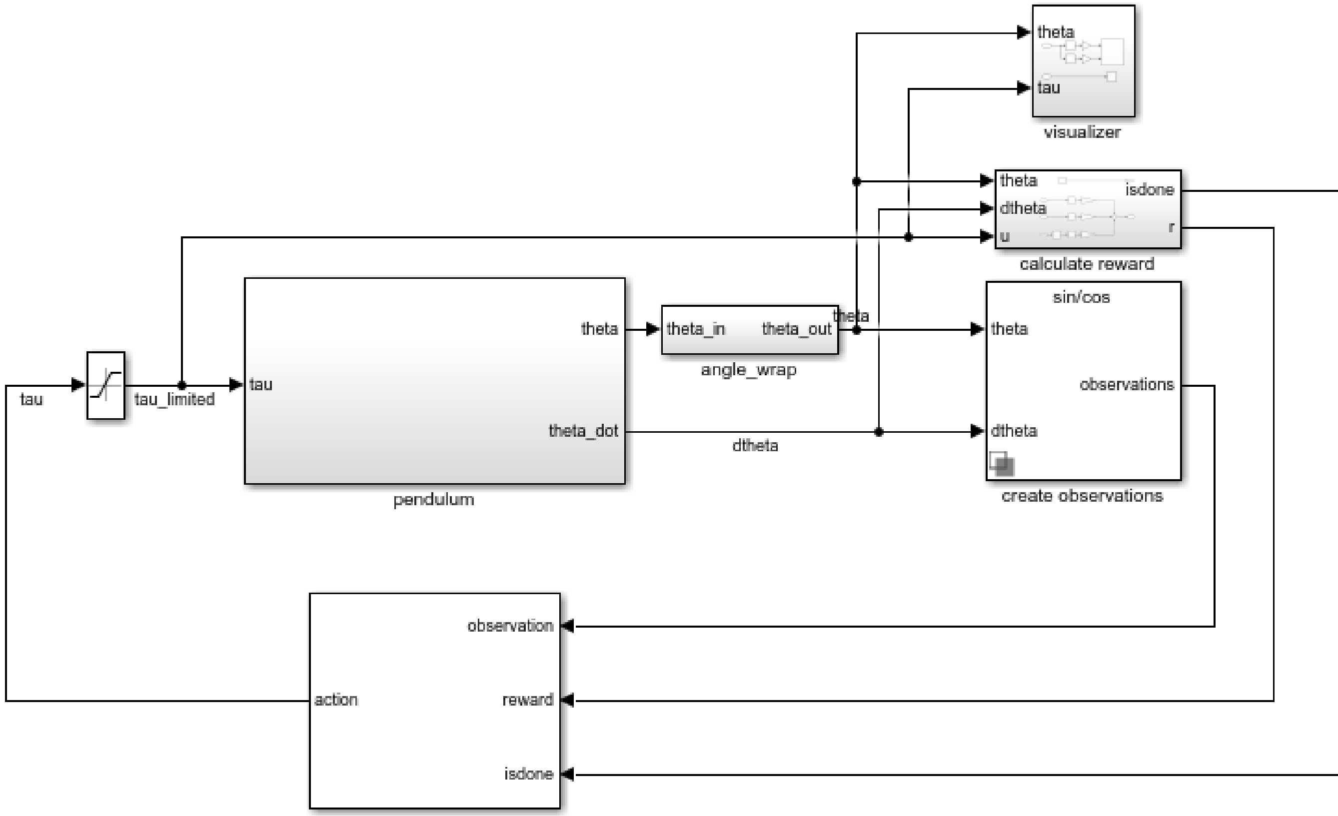
| Environment | Agent Task |
| --- | --- |
| Simple pendulum Simulink model | Swing up and balance a simple pendulum using either a discrete or continuous action space. |
| Cart-pole Simscape™ model | Balance a pole on a moving cart by applying forces to the cart using either a discrete or continuous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

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

## Simple Pendulum Simulink Model

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

```
open_system('rlSimplePendulumModel')
```




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

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

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

- Discrete action space

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

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

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

- “Train DQN Agent to Swing Up and Balance Pendulum” on page 5-89
- “Train DDPG Agent to Swing Up and Balance Pendulum” on page 5-97

### Actions

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

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

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

#### Observations

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

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

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

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

#### Reward

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

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

Here:

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

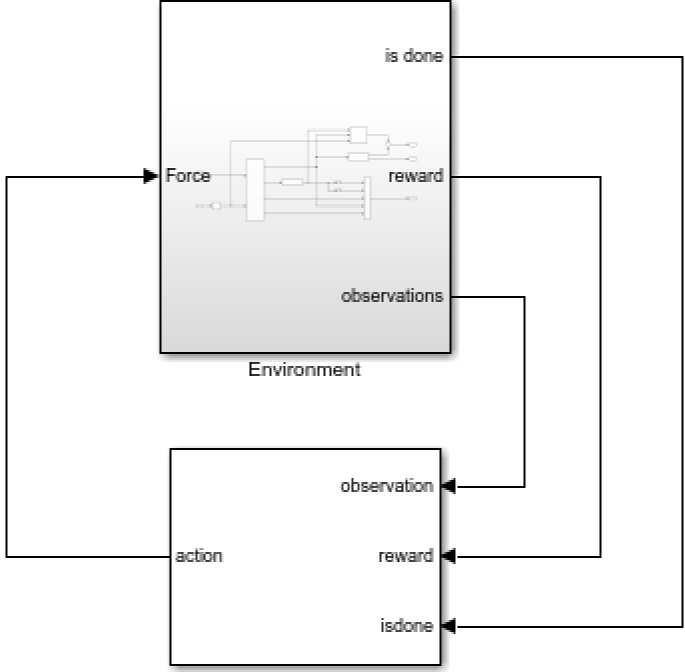
## Cart-Pole Simscape Model

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

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

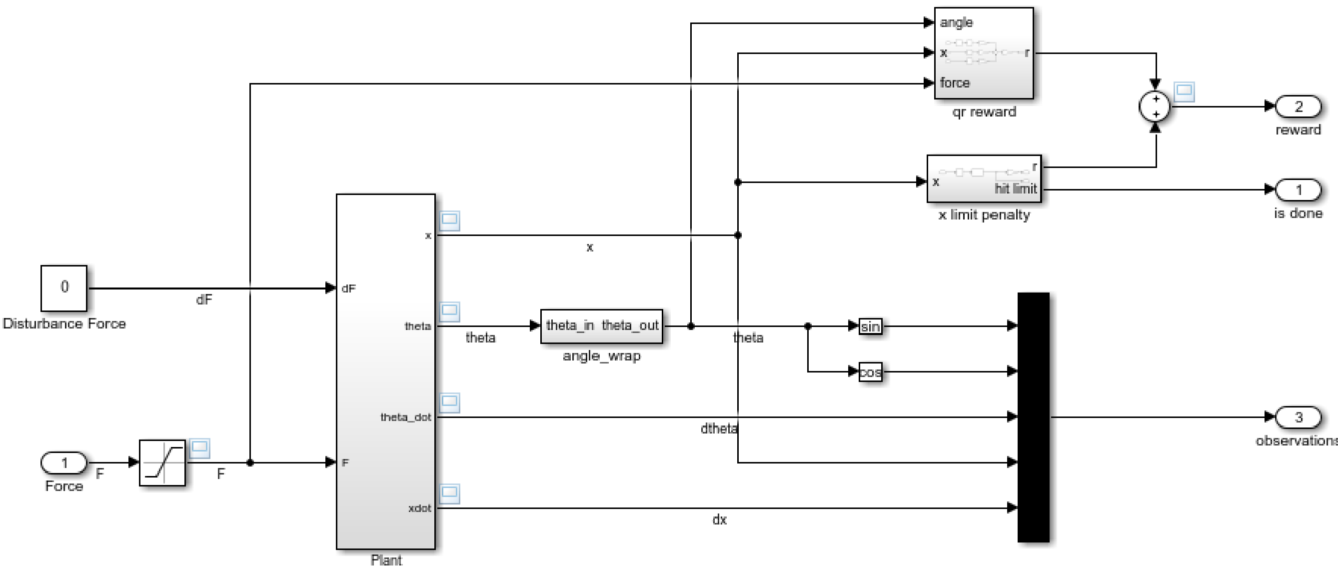
The model for this environment is defined in the `rlCartPoleSimscapeModel` Simulink model. The dynamics of this model are defined using Simscape Multibody™.

```
open_system('rlCartPoleSimscapeModel')
```




In the Environment subsystem, the model dynamics are defined using Simscape components and the reward and observation are constructed using Simulink blocks.

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




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

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

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

- Discrete action space

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

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

For an example that trains an agent in this cart-pole environment, see “Train DDPG Agent to Swing Up and Balance Cart-Pole System” on page 5-106.

#### Actions

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

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

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

#### Observations

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

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

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

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

#### Reward

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

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

  $$r_{qr} = -\left(0.1 * x^2 + 0.5 * \theta^2 + 0.005 * u_{t-1}^2\right)$$
- A cart limit penalty, constructed in the `Environment/x limit penalty` subsystem. This subsystem generates a negative reward when the magnitude of the cart position exceeds a given threshold.

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

Here:

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

## See Also

**Functions**
`rlPredefinedEnv` | `train`

**Blocks**
RL Agent

## Related Examples

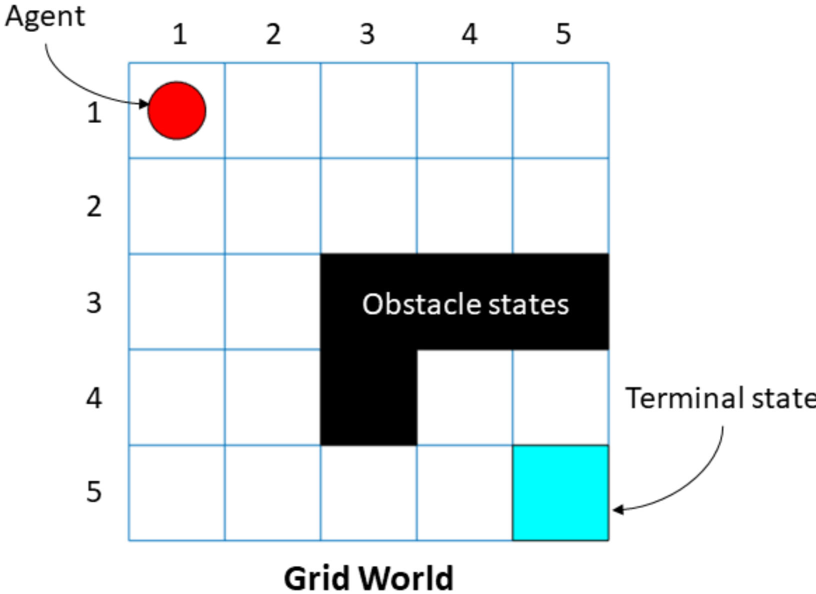
- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “Create Simulink Reinforcement Learning Environments” on page 2-8

# Create Custom Grid World Environments

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




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

1. Create the grid world model.
2. Configure the grid world model.
3. Use the grid world model to create your own grid world environment.

## Grid World Model

You can create your own grid world model using the `createGridWorld` function. Specify the grid size when creating the `GridWorld` model object.

The `GridWorld` object has the following properties.

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

<table>
<tr><th>Property</th><th>Read-Only</th><th>Description</th></tr>
<tr><td>CurrentState</td><td>No</td><td>Name of the current state of the agent, specified as a string. You can use this property to set the initial state of the agent. The agent always starts from cell [1,1] by default.<br><br>The agent starts from the CurrentState once you use the reset function in the rlMDPEnv environment object.</td></tr>
<tr><td>States</td><td>Yes</td><td>A string vector containing the state names of the grid world. For instance, for a 2-by-2 grid world model GW, specify the following:<br><br><pre>GW.States = ["[1,1]";
             "[2,1]";
             "[1,2]";
             "[2,2]"];</pre></td></tr>
<tr><td>Actions</td><td>Yes</td><td>A string vector containing the list of possible actions that the agent can use. You can set the actions when you create the grid world model by using the moves argument:<br><br><pre>GW = createGridWorld(m,n,moves)</pre>Specify moves as either 'Standard' or 'Kings'.<br><br><table>
<tr><th>moves</th><th>Gw.Actions</th></tr>
<tr><td>'Standard'</td><td>['N';'S';'E';'W']</td></tr>
<tr><td>'Kings'</td><td>['N';'S';'E';'W';'NE';'NW';'SE';'SW']</td></tr>
</table></td></tr>
</table>

| Property | Read-Only | Description |
| --- | --- | --- |
| T | No | State transition matrix, specified as a 3-D array. `T` is a probability matrix that indicates the likelihood of the agent moving from the current state `s` to any possible next state `s'` by performing action `a`.<br><br>`T` can be denoted as<br><br>$T(s, s', a) = probability(s' \| s, a)$.<br><br>For instance, consider a 5-by-5 deterministic grid world object `GW` with the agent in cell `[3,1]`. View the state transition matrix for the north direction.<br><br>`northStateTransition = GW.T(:,:,1)`<br><br>From the above figure, the value of `northStateTransition(3,2)` is 1 since the agent moves from cell `[3,1]` to cell `[2,1]` with action `'N'`. A probability of 1 indicates that from a given state, if the agent goes north, it has a 100% chance of moving one cell north on the grid. For an example showing how to set up the state transition matrix, see "Train Reinforcement Learning Agent in Basic Grid World" on page 1-14. |

| Property | Read-Only | Description |
|---|---|---|
| R | No | Reward transition matrix, specified as a 3-D array. `R` determines how much reward the agent receives after performing an action in the environment. `R` has the same shape and size as the state transition matrix `T`.<br><br>The reward transition matrix `R` can be denoted as<br><br>$r = R(s, s', a)$.<br><br>Set up `R` such that there is a reward to the agent after every action. For instance, you can set up a positive reward if the agent transitions over obstacle states and when it reaches the terminal state. You can also set up a default reward of -11 for all actions the agent takes, independent of the current state and next state. For an example that show how to set up the reward transition matrix, see "Train Reinforcement Learning Agent in Basic Grid World" on page 1-14. |
| `ObstacleStates` | No | `ObstacleStates` are states that cannot be reached in the grid world, specified as a string vector. Consider the following 5-by-5 grid world model `GW`.<br><br>The black cells are obstacle states, and you can specify them using the following syntax:<br><br>`GW.ObstacleStates = ["[3,3]";"[3,4]";"[3,5]";"[4,3]"];`<br><br>For a workflow example, see "Train Reinforcement Learning Agent in Basic Grid World" on page 1-14. |

| Property | Read-Only | Description |
|---|---|---|
| TerminalStates | No | TerminalStates are the final states in the grid world, specified as a string vector. Consider the previous 5-by-5 grid world model GW. The blue cell is the terminal state and you can specify it by:<br><br>`GW.TerminalStates = "[5,5]";`<br><br>For a workflow example, see "Train Reinforcement Learning Agent in Basic Grid World" on page 1-14. |

## Grid World Environment

You can create a Markov decision process (MDP) environment using rlMDPEnv from the grid world model from the previous step. MDP is a discrete-time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of the decision maker. The agent uses the grid world environment object rlMDPEnv to interact with the grid world model object GridWorld.

For more information, see rlMDPEnv and "Train Reinforcement Learning Agent in Basic Grid World" on page 1-14.

## See Also

**Functions**
createGridWorld | rlPredefinedEnv

**Objects**
rlMDPEnv

## Related Examples

- "Train Reinforcement Learning Agent in Basic Grid World" on page 1-14

# Create MATLAB Environment Using Custom Functions

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

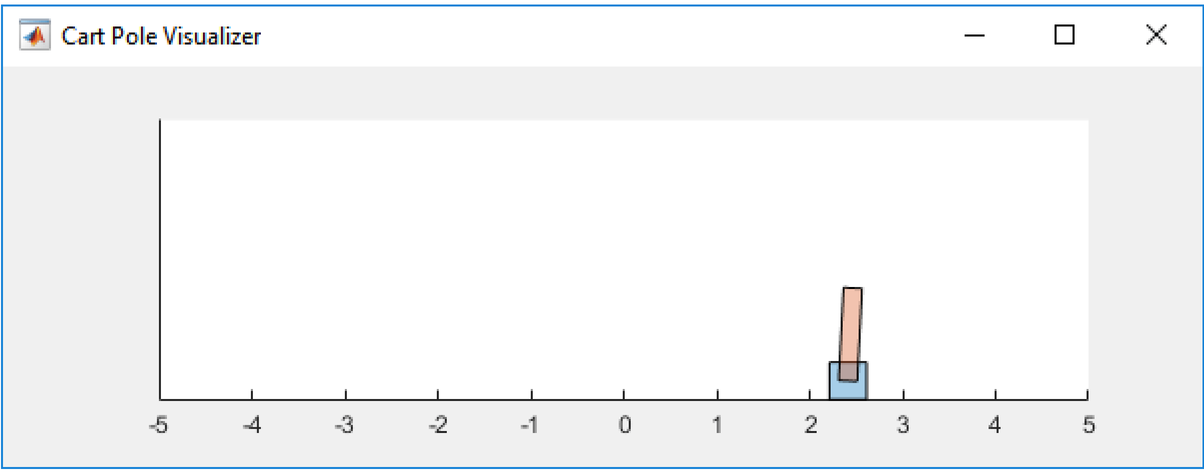
Using the `rlFunctionEnv` function, you can create a MATLAB reinforcement learning environment from an observation specification, an action specification, and user-defined `step` and `reset` functions. You can then train a reinforcement learning agent in this environment. The necessary `step` and `reset`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

Creating an environment using custom functions is useful for environments with less complex dynamics, environments with no special visualization requirements, or environments with interfaces to third-party libraries. For more complex environments, you can create an environment object using a template class. For more information, see "Create Custom MATLAB Environment from Template" on page 2-48.

For more information on creating reinforcement learning environments, see "Create MATLAB Reinforcement Learning Environments" on page 2-2 and "Create Simulink Reinforcement Learning Environments" on page 2-8.

### Cart-Pole MATLAB Environment

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




For this environment:

- The upward balanced pendulum position is `0` radians, and the downward hanging position is `pi` radians.
- The pendulum starts upright with an initial angle that is between –0.05 and 0.05.
- The force action signal from the agent to the environment is from –10 to 10 N.
- The observations from the environment are the cart position, cart velocity, pendulum angle, and pendulum angle derivative.
- The episode terminates if the pole is more than 12 degrees from vertical, or if the cart moves more than 2.4 m from the original position.

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

For more information on this model, see "Load Predefined Control System Environments" on page 2-23.

**Observation and Action Specifications**

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

```
ObservationInfo = rlNumericSpec([4 1]);
ObservationInfo.Name = "CartPole States";
ObservationInfo.Description = 'x, dx, theta, dtheta';
```

The environment has a discrete action space where the agent can apply one of two possible force values to the cart: `-10` or `10` N.

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

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

**Create Environment Using Function Names**

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

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

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

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

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

```
type myResetFunction.m
```

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

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

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

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

end
```

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

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

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

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

```
type myStepFunction.m
```

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

% Define the environment constants.

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

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

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

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

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

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

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

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

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

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

end
```

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

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

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

### Create Environment Using Function Handles

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

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

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

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

For more information, see “Anonymous Functions”.

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

```
type myStepFunction2.m
```

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

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

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

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

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

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

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

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

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

end
```

Create the structure that contains the environment constants.

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

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

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

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

```
ResetHandle = @() myResetFunction;
```

Create the environment using the custom function handles.

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

**Validate Custom Functions**

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

Validate the environment created using function names.

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

```
InitialObs = 4×1

         0
         0
    0.0315
         0
```

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

```
NextObs = 4×1

         0
    0.1947
    0.0315
   -0.2826
```

Validate the environment created using function handles.

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

```
InitialObs2 = 4×1

         0
         0
    0.0315
         0
```

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

```
NextObs2 = 4×1

         0
    0.1947
    0.0315
   -0.2826
```

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

## See Also

**Objects**
`rlFunctionEnv`

## More About

- “Create MATLAB Reinforcement Learning Environments” on page 2-2
- “Create Custom MATLAB Environment from Template” on page 2-48

# Create Custom MATLAB Environment from Template

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

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

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

You can create less complex custom reinforcement learning environments using custom functions, as described in “Create MATLAB Environment Using Custom Functions” on page 2-41.

## Create Template Class

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

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

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

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

By default, the template class implements a simple cart-pole balancing model similar to the cart-pole predefined environments described in “Load Predefined Control System Environments” on page 2-23.

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

- Environment properties
- Required environment methods
- Optional environment methods

## Environment Properties

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

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

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

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

    % Mass of the cart
    CartMass = 1.0

    % Mass of the pole
    PoleMass = 0.1

    % Half the length of the pole
    HalfPoleLength = 0.5

    % Max force the input can apply
    MaxForce = 10

    % Sample time
    Ts = 0.02

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

    % Distance at which to fail the episode
    DisplacementThreshold = 2.4

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

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

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

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

## Required Functions

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

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

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

#### Sample Constructor Function

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

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

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

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

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

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

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

#### Sample reset Function

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

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

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

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

### Sample step Function

The sample cart-pole `step` function:

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

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

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

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

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

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

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

    % Update system states
    this.State = Observation;

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

    % Get reward
    Reward = getReward(this);

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

## Optional Functions

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

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

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

## Environment Visualization

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

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

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

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

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

    % Handle to figure
    Figure
end
```

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

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

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

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

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

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

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

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

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

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

## Create Custom Environment

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

```
env = MyEnvironment;
```

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

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

```
validateEnvironment(env)
```

After validating the environment object, you can use it to train a reinforcement learning agent. For more information on training agents, see “Train Reinforcement Learning Agents” on page 5-3.

## See Also

**Functions**
`rlCreateEnvTemplate` | `train`

## More About

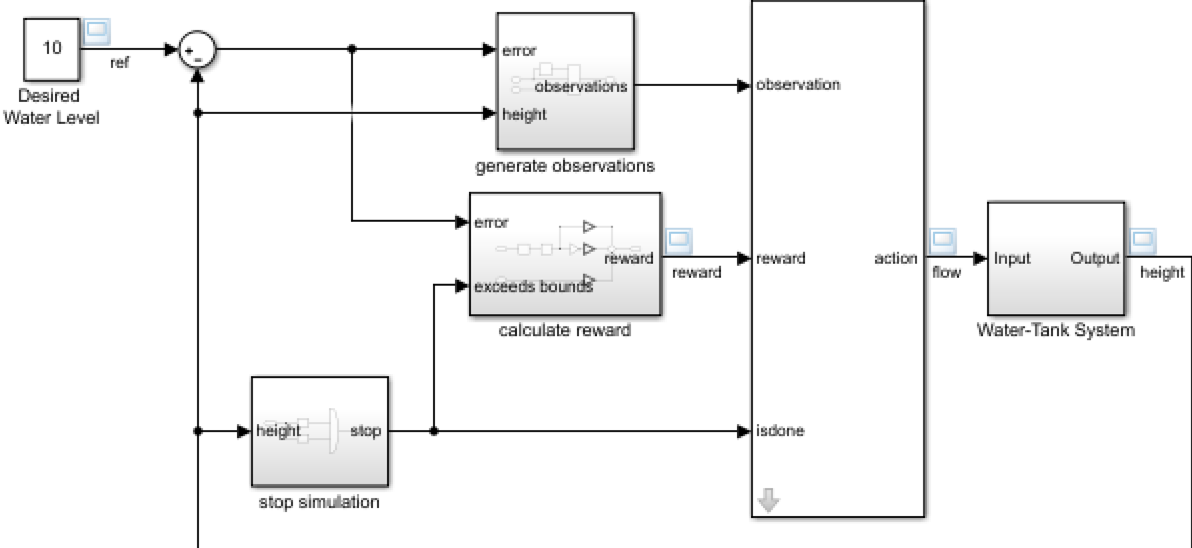
- “Create MATLAB Reinforcement Learning Environments” on page 2-2
- “Create MATLAB Environment Using Custom Functions” on page 2-41

- "Define Reward Signals" on page 2-14

# Water Tank Reinforcement Learning Environment Model

This example shows how to create a water tank reinforcement learning Simulink® environment that contains an RL Agent block in the place of a controller for the water level in a tank. To simulate this environment, you must create an agent and specify that agent in the RL Agent block. For an example that trains an agent using this environment, see "Create Simulink Environment and Train Agent" on page 1-20.

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




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

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

### Actions and Observations

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

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

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

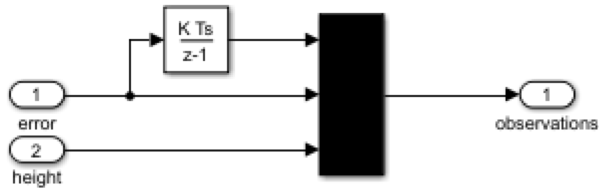
If the action signal takes one of a discrete set of possible values, create the specification using the `rlFiniteSetSpec` function.

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

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

Compute the observation signals in the generate observations subsystem.

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




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

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

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

#### Reward Signal

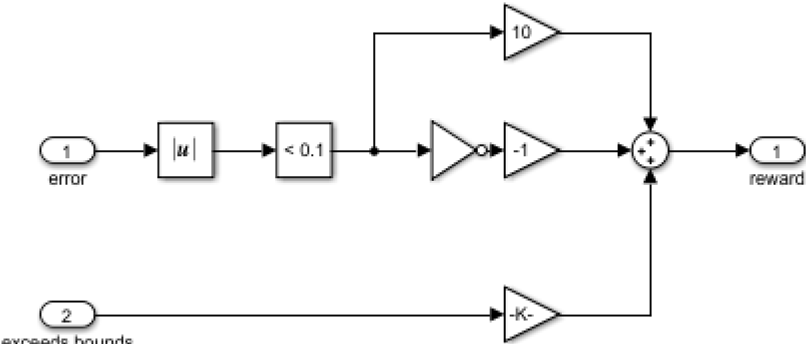
Construct a scalar reward signal. For this example, specify the following reward.

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

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

Construct this reward in the calculate reward subsystem.

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

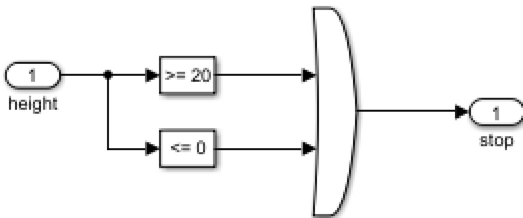



### Stop Signal

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

Compute this signal in the stop simulation subsystem.

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




### Create Environment Object

Create an environment object for the Simulink model.

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

### Reset Function

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

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

### Local Function

```
function in = localResetFcn(in)

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

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

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

end
```

## See Also

**Functions**
rlSimulinkEnv

## More About

- "Create Simulink Reinforcement Learning Environments" on page 2-8

# Create Agents

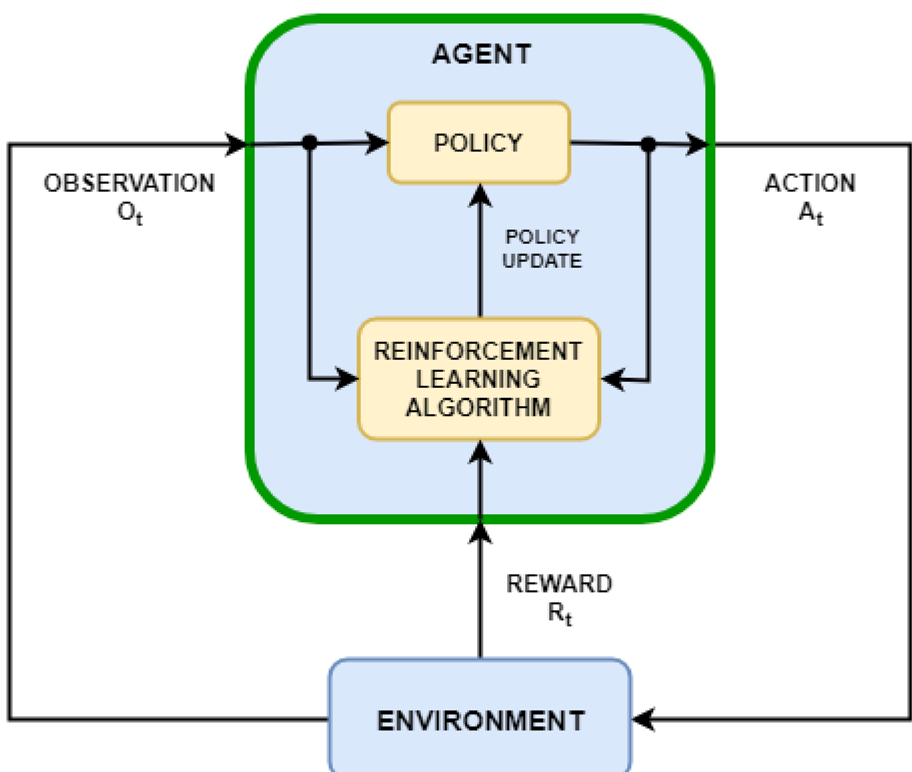

- “Reinforcement Learning Agents” on page 3-2
- “Create Agents Using Reinforcement Learning Designer” on page 3-9
- “Q-Learning Agents” on page 3-17
- “SARSA Agents” on page 3-20
- “Deep Q-Network (DQN) Agents” on page 3-23
- “Policy Gradient (PG) Agents” on page 3-27
- “Actor-Critic (AC) Agents” on page 3-31
- “Soft Actor-Critic (SAC) Agents” on page 3-35
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents” on page 3-44
- “Proximal Policy Optimization (PPO) Agents” on page 3-49
- “Trust Region Policy Optimization (TRPO) Agents” on page 3-55
- “Model-Based Policy Optimization (MBPO) Agents” on page 3-62
- “Create Custom Reinforcement Learning Agents” on page 3-68

# Reinforcement Learning Agents

The goal of reinforcement learning is to train an agent to complete a task within an uncertain environment. At each time interval, the agent receives observations and a reward from the environment and sends an action to the environment. The reward is a measure of how successful the previous action (taken from the previous state) was with respect to completing the task goal.

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

- The policy is a mapping from the current environment observation to a probability distribution of the actions to be taken. Within an agent, the policy is implemented by a function approximator with tunable parameters and a specific approximation model, such as a deep neural network.
- The learning algorithm continuously updates the policy parameters based on the actions, observations, and rewards. The goal of the learning algorithm is to find an optimal policy that maximizes the expected cumulative long-term reward received during the task.




Depending on the learning algorithm, an agent maintains one or more parameterized function approximators for training the policy. Approximators can be used in two ways.

- **Critics** — For a given observation and action, a critic returns the predicted discounted value of the cumulative long-term reward.
- **Actor** — For a given observation, an actor returns as output the action that (often) maximizes the predicted discounted cumulative long-term reward.

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

function (value as a function of the observation) or Q-value function (value as a function of observation and action). In general, these agents work better with discrete action spaces but can become computationally expensive for continuous action spaces.

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

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

## Built-In Agents

The following tables summarize the types, action spaces, and used approximators for all the built-in agents provided with Reinforcement Learning Toolbox software.

On-policy agents attempt to evaluate or improve the policy that they are using to make decisions, whereas off-policy agents evaluate or improve a policy that can be different from the one that they are using to make decisions, (or from the one that has been used to generate data). For each agent, the observation space can be discrete, continuous or mixed. For more information, see [1].

**On-Policy Built-In Agents: Type and Action Space**

| Agent | Type | Action Space |
|---|---|---|
| “SARSA Agents” on page 3-20 | Value-Based | Discrete |
| “Policy Gradient (PG) Agents” on page 3-27 (PG) | Policy-Based | Discrete or continuous |
| “Actor-Critic (AC) Agents” on page 3-31 (AC) | Actor-Critic | Discrete or continuous |
| “Proximal Policy Optimization (PPO) Agents” on page 3-49 (PPO) | Actor-Critic | Discrete or continuous |
| “Trust Region Policy Optimization (TRPO) Agents” on page 3-55 (TRPO) | Actor-Critic | Discrete or continuous |

**Off-Policy Built-In Agents: Type and Action Space**

| Agent | Type | Action Space |
|---|---|---|
| "Q-Learning Agents" on page 3-17 (Q) | Value-Based | Discrete |
| "Deep Q-Network (DQN) Agents" on page 3-23 | Value-Based | Discrete |
| "Soft Actor-Critic (SAC) Agents" on page 3-35 (SAC) | Actor-Critic | Continuous |
| "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40 | Actor-Critic | Continuous |
| "Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents" on page 3-44 (TD3) | Actor-Critic | Continuous |
| "Model-Based Policy Optimization (MBPO) Agents" on page 3-62 (MBPO) | Actor-Critic | Discrete or continuous |

**Built-In Agents: Approximators Used by Each Agent**

| Approximator | Q, DQN, SARSA | PG | AC, PPO, TRPO | SAC | DDPG, TD3 |
|---|---|---|---|---|---|
| Value function critic $V(S)$, which you can create using<br><br>`rlValueFunction` | | X (if baseline is used) | X | | |
| Q-value function critic $Q(S,A)$, which you can create using<br><br>`rlQValueFunction` | X | | | X | X |
| Multi-output Q-value function critic $Q(S)$, for discrete action spaces, which you can create using<br><br>`rlVectorQValueFunction` | X | | | | |
| Deterministic policy actor $\pi(S)$, which you can create using<br><br>`rlContinuousDeterministicActor` | | | | | X |
| Stochastic (Multinoulli) policy actor $\pi(S)$, for discrete action spaces, which you can create using<br><br>`rlDiscreteCategoricalActor` | | X | X | | |
| Stochastic (Gaussian) policy actor $\pi(S)$, for continuous action spaces, which you can create using<br><br>`rlContinuousGaussianActor` | | X | X | X | |

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

1. Create observation specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getObservationInfo`.
2. Create action specifications for your environment. If you already have an environment interface object, you can obtain these specifications using `getActionInfo`.
3. If needed, specify the number of neurons in each learnable layer or whether to use an LSTM layer. To do so, create an agent initialization option object using `rlAgentInitializationOptions`.
4. If needed, specify agent options by creating an options object set for the specific agent. This option object in turn includes `rlOptimizerOptions` objects that specify optimization objects for the agent actor or critic.

5 Create the agent using the corresponding agent creation function. The resulting agent contains the appropriate actor and critics listed in the table above. The actor and critic use default agent-specific deep neural networks as internal approximators.

For more information on creating actor and critic function approximators, see “Create Policies and Value Functions” on page 4-2.

You can use the **Reinforcement Learning Designer** app to import an existing environment and interactively design DQN, DDPG, PPO, or TD3 agents. The app allows you to train and simulate the agent within your environment, analyze the simulation results, refine the agent parameters, and export the agent to the MATLAB workspace for further use and deployment. For more information, see “Create Agents Using Reinforcement Learning Designer” on page 3-9.

## Choose Agent Type

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

- **Discrete action space** — For environments with a discrete action spaces, the Q-learning and SARSA agents are the simplest compatible agent, followed by DQN, PPO, and TRPO. If your observation space is continuous you cannot use a table as approximation model.

- **Continuous action space** — For environments with a continuous action space, DDPG is the simplest compatible agent, followed by TD3, PPO, and SAC, which are then followed by TRPO. For such environments, try DDPG first. In general:
  - TD3 is an improved, more complex version of DDPG.
  - PPO has more stable updates but requires more training.
  - SAC is an improved, more complex version of DDPG that generates stochastic policies.
  - TRPO is a more complex version of PPO that is more robust for deterministic environments with fewer observations.

## Model-Based Policy Optimization

If you are using an off-policy agent (DQN, DDPG, TD3, SAC), you can consider using model-based policy optimization (MBPO) agent. to improve your training sample efficiency. An MBPO agent contains an internal model of the environment, which it uses to generate additional experiences without interacting with the environment.

During training, the MBPO agent generates real experiences by interacting with the environment. These experiences are used to train the internal environment model, which is used to generate additional experiences. The training algorithm then uses both the real and generated experiences to update the agent policy.

An MBPO agent can be more sample efficient than model-free agents because the model can generate large sets of diverse experiences. However, MBPO agents require much more computational time than model-free agents, because they must train the environment model and generate samples in addition to training the base agent.

For more information, see “Model-Based Policy Optimization (MBPO) Agents” on page 3-62.

## Extract Policy Objects from Agents

You can extract a policy object from an agent and then use `getAction` to generate deterministic or stochastic actions from the policy, given an input observation. Working with policy objects can be useful for application deployment or custom training purposes. For more information, see “Create Policies and Value Functions” on page 4-2.

## Custom Agents

You can also train policies using other learning algorithms by creating a custom agent. To do so, you create a subclass of a custom agent class, and define the agent behavior using a set of required and optional methods. For more information, see “Create Custom Reinforcement Learning Agents” on page 3-68. For more information about custom training loops, see “Train Reinforcement Learning Policy Using Custom Training Loop” on page 5-433.

## References

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning. Cambridge, Mass: The MIT Press, 2018.

## See Also

**Objects**
`rlQAgent` | `rlSARSAAgent` | `rlDQNAgent` | `rlPGAgent` | `rlDDPGAgent` | `rlTD3Agent` | `rlACAgent` | `rlSACAgent` | `rlPPOAgent` | `rlTRPOAgent` | `rlMBPOAgent`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “What Is Reinforcement Learning?” on page 1-3
- “Create Agents Using Reinforcement Learning Designer” on page 3-9

# Create Agents Using Reinforcement Learning Designer

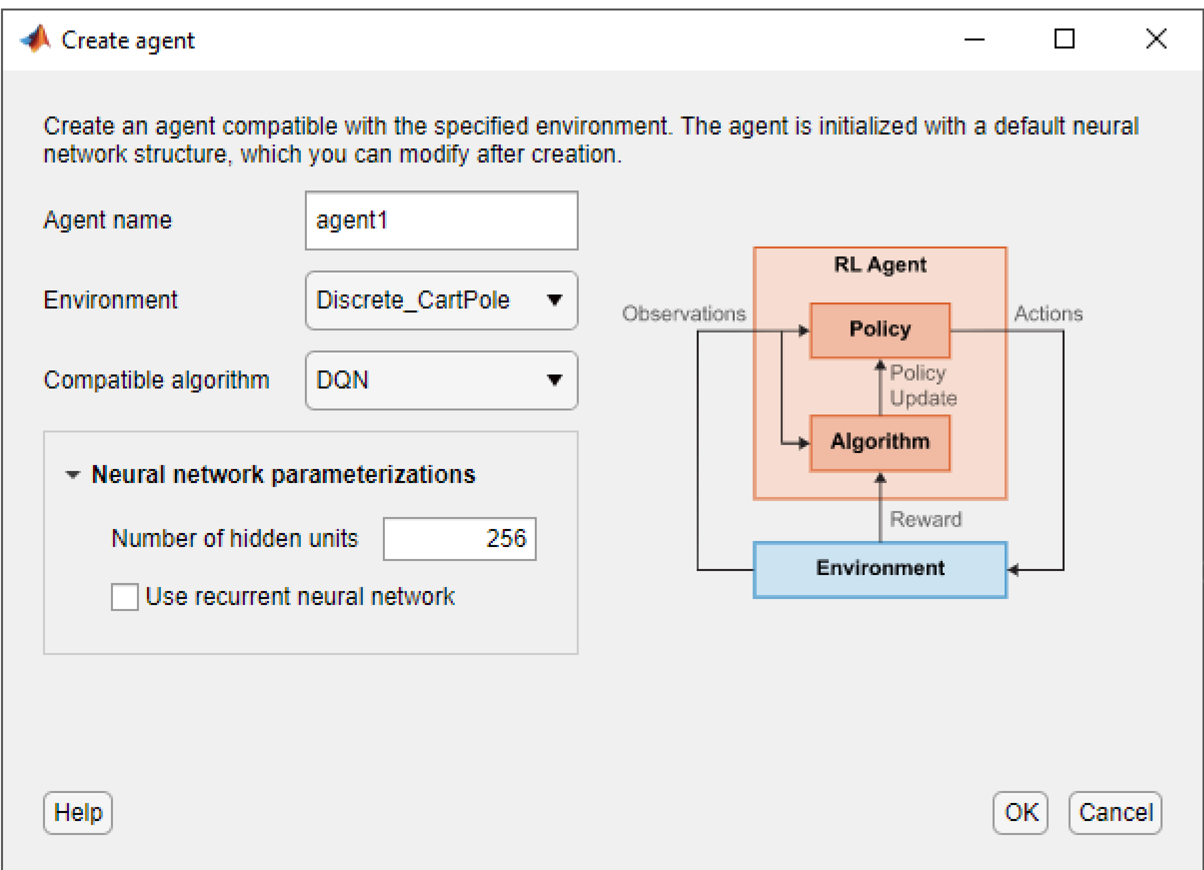
The **Reinforcement Learning Designer** app supports the following types of agents.

- "Deep Q-Network (DQN) Agents" on page 3-23 (DQN)
- "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40 (DDPG)
- "Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents" on page 3-44 (TD3)
- "Proximal Policy Optimization (PPO) Agents" on page 3-49 (PPO)
- "Trust Region Policy Optimization (TRPO) Agents" on page 3-55 (TRPO)
- "Soft Actor-Critic (SAC) Agents" on page 3-35 (SAC)

To train an agent using **Reinforcement Learning Designer**, you must first create or import an environment. For more information, see "Create or Import MATLAB Environments in Reinforcement Learning Designer" on page 2-5 and "Create or Import Simulink Environments in Reinforcement Learning Designer" on page 2-11.

## Create Agent

To create an agent, on the **Reinforcement Learning** tab, in the **Agent** section, click **New**.




In the Create agent dialog box, specify the following information.

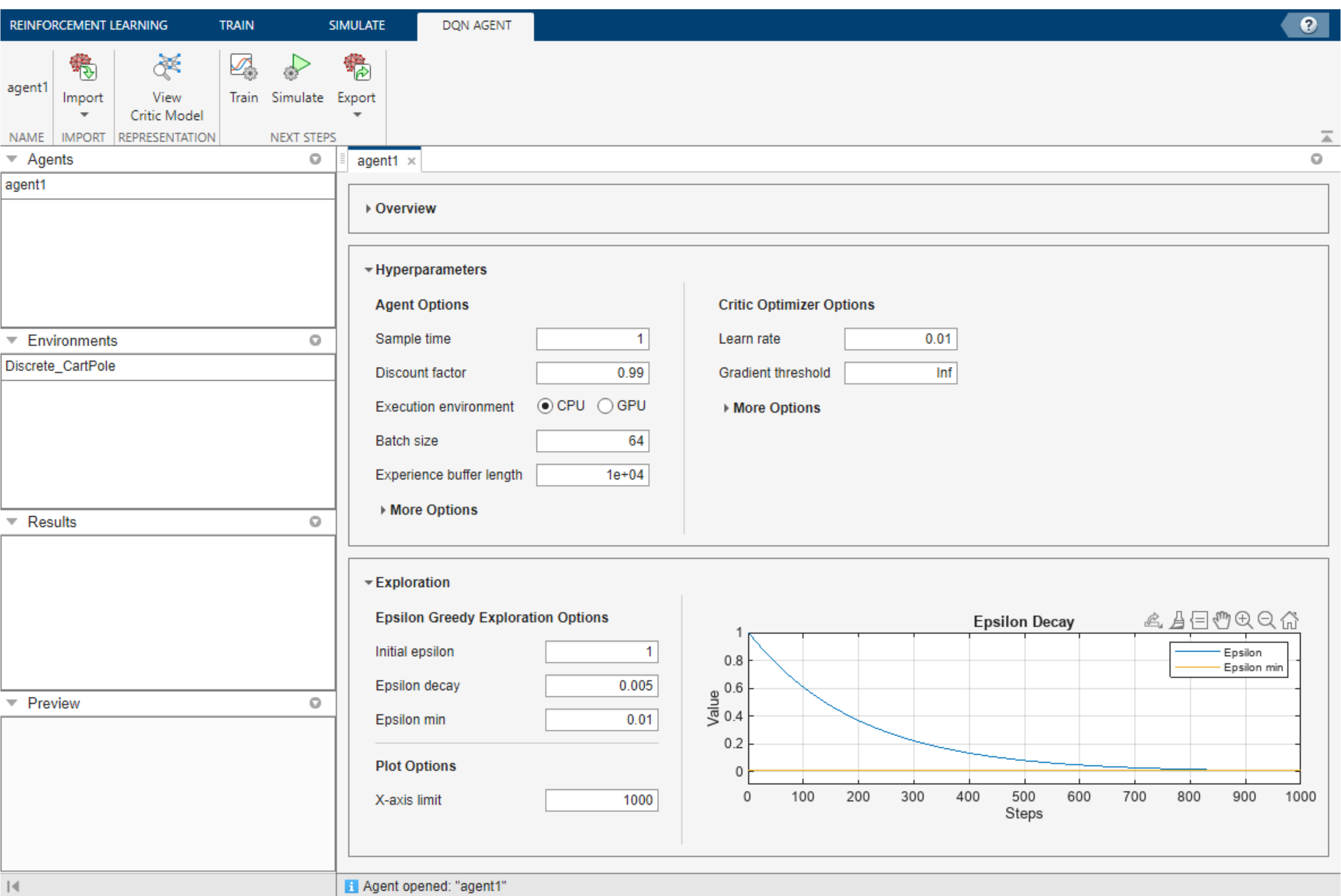
- **Agent name** — Specify the name of your agent.
- **Environment** — Select an environment that you previously created or imported.
- **Compatible algorithm** — Select an agent training algorithm. This list contains only algorithms that are compatible with the environment you select.

The **Reinforcement Learning Designer** app creates agents with actors and critics based on default deep neural network. You can specify the following options for the default networks.

- **Number of hidden units** — Specify number of units in each fully-connected or LSTM layer of the actor and critic networks.
- **Use recurrent neural network** — Select this option to create actor and critic with recurrent neural networks that contain an LSTM layer.

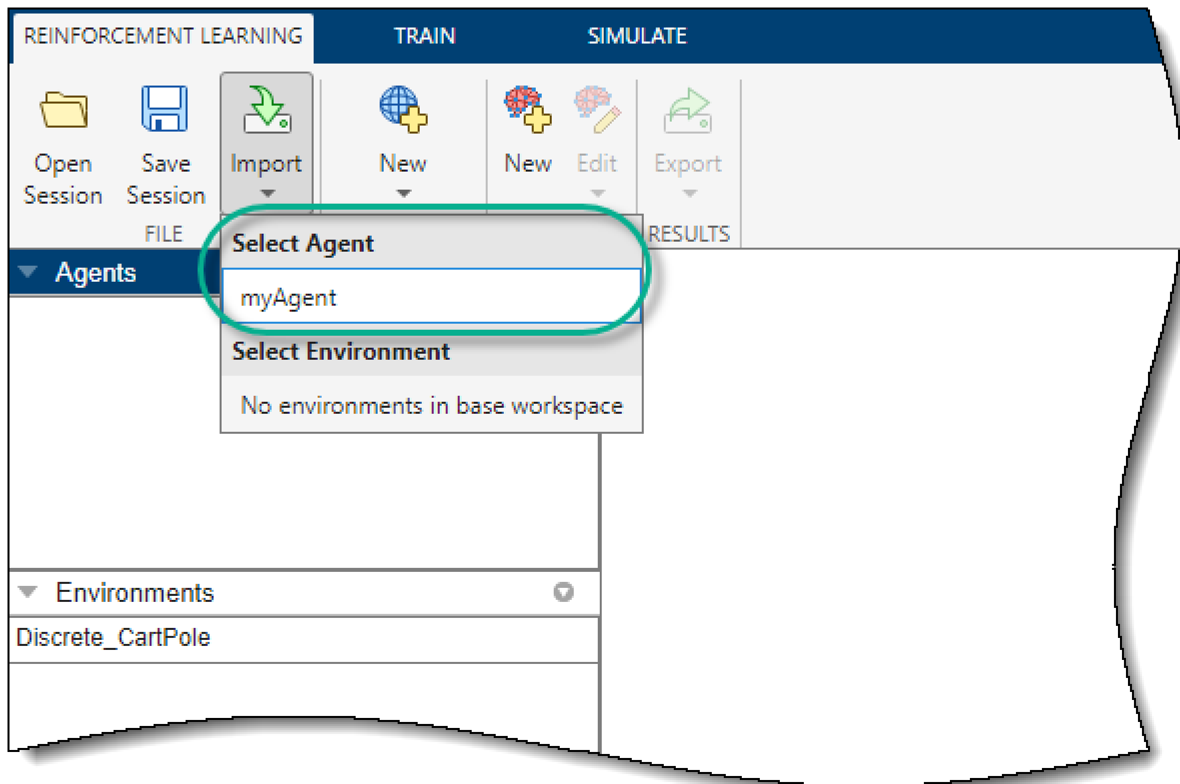
To create the agent, click **OK**.

The app adds the new default agent to the **Agents** pane and opens a document for editing the agent options.



## Import Agent

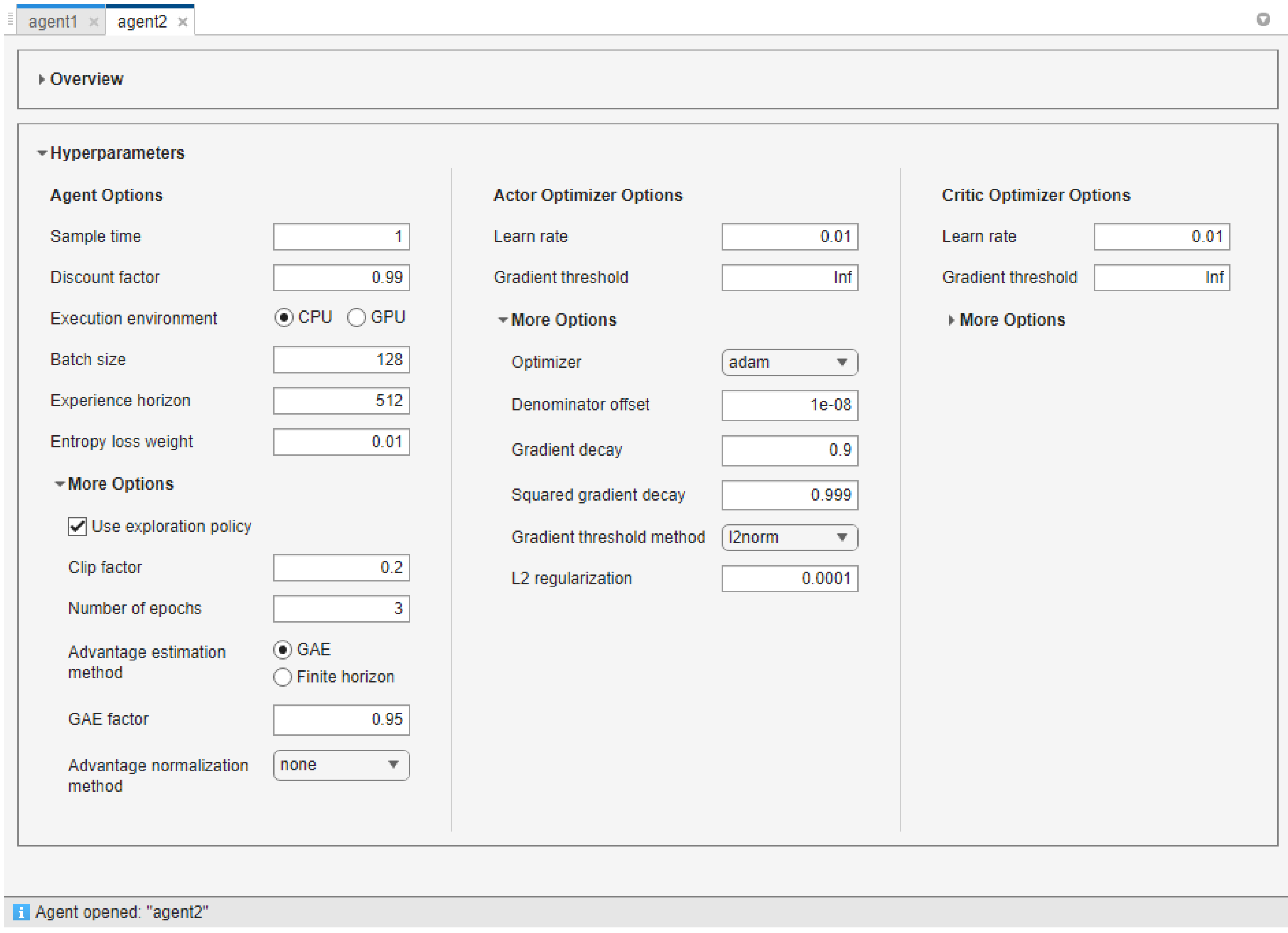
You can also import an agent from the MATLAB workspace into **Reinforcement Learning Designer**. To do so, on the **Reinforcement Learning** tab, click **Import**. Then, under **Select Agent**, select the agent to import.




The app adds the new imported agent to the **Agents** pane and opens a document for editing the agent options.

## Edit Agent Options

In **Reinforcement Learning Designer**, you can edit agent options in the corresponding agent document.




You can edit the following options for each agent.

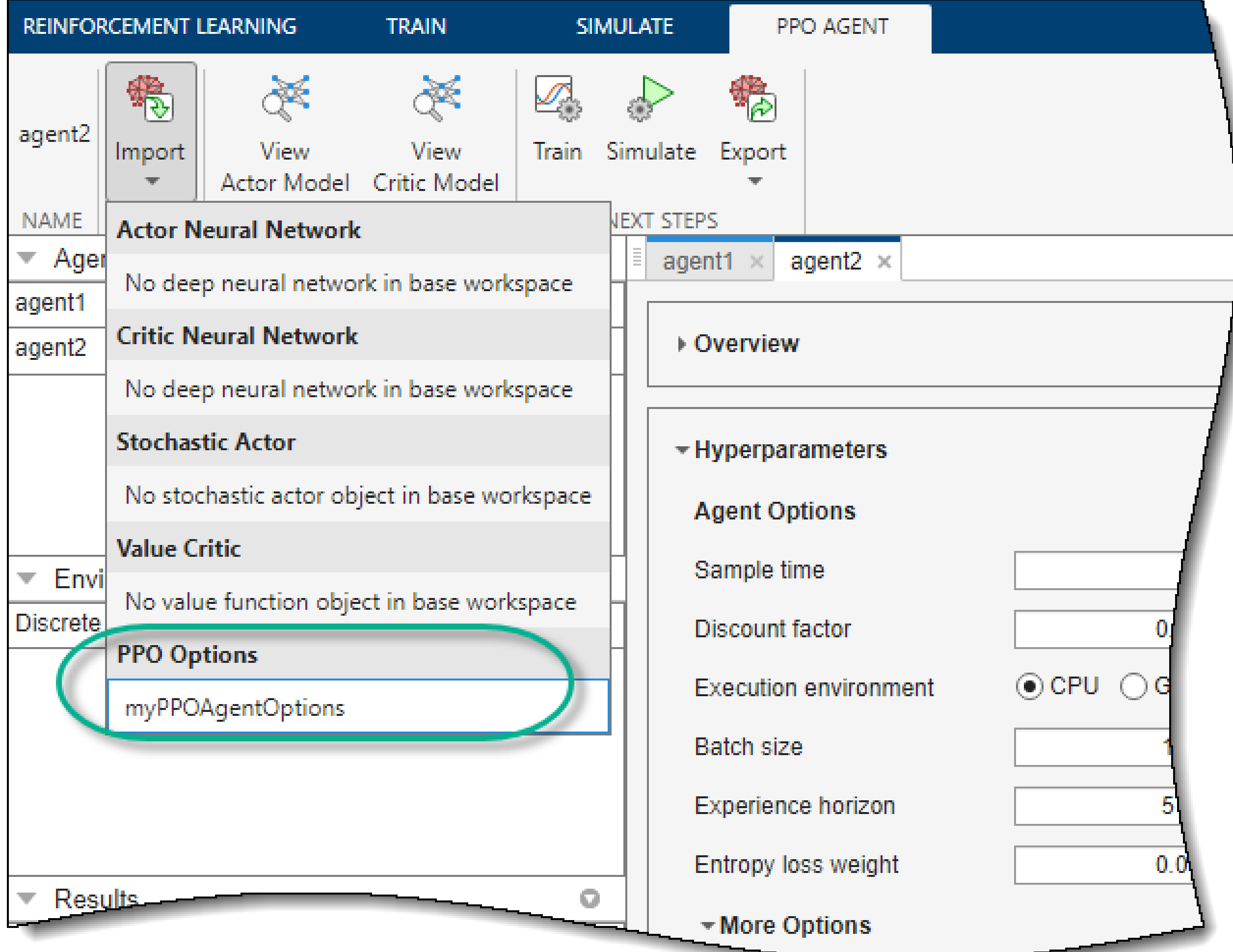
- **Agent Options** — Agent options, such as the sample time and discount factor. Specify these options for all supported agent types.
- **Exploration Model** — Exploration model options. PPO agents do not have an exploration model.
- **Target Policy Smoothing Model** — Options for target policy smoothing, which is supported for only TD3 agents.

For more information on these options, see the corresponding agent options object.

- `rlDQNAgentOptions` — DQN agent options
- `rlDDPGAgentOptions` — DDPG agent options
- `rlTD3AgentOptions` — TD3 agent options
- `rlPPOAgentOptions` — PPO agent options

You can import agent options from the MATLAB workspace. To create options for each type of agent, use one of the preceding objects. You can also import options that you previously exported from the **Reinforcement Learning Designer** app

To import the options, on the corresponding **Agent** tab, click **Import**. Then, under **Options**, select an options object. The app lists only compatible options objects from the MATLAB workspace.




The app configures the agent options to match those In the selected options object.

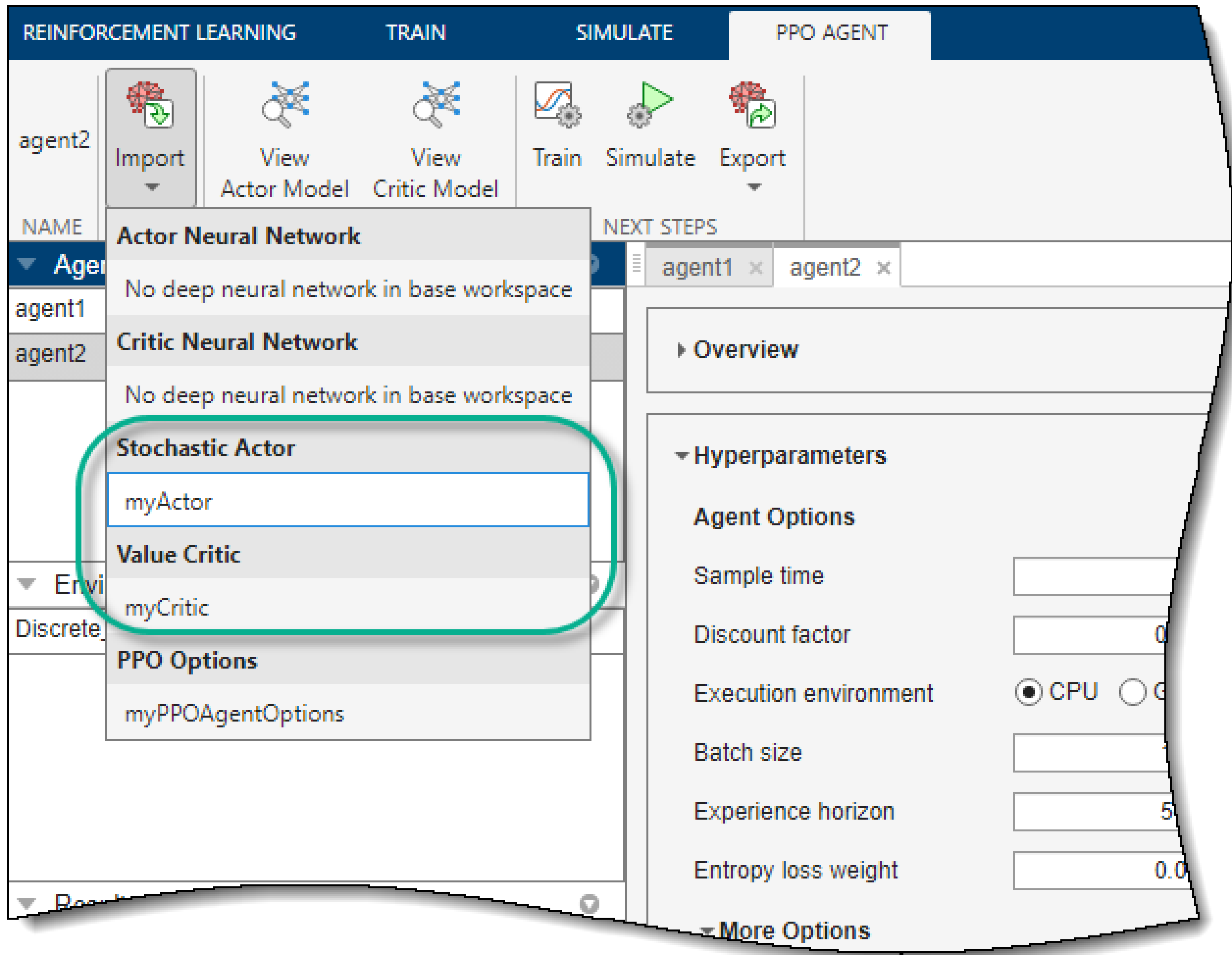
## Edit Actor and Critic

You can edit the properties of the actor and critic of each agent.

- DQN agents have just a critic network.
- DDPG and PPO agents have an actor and a critic.
- TD3 agents have an actor and two critics. When you modify the critic options for a TD3 agent, the changes apply to both critics.

You can also import actors and critics from the MATLAB workspace. For more information on creating actors and critics, see “Create Policies and Value Functions” on page 4-2. You can also import actors and critics that you previously exported from the **Reinforcement Learning Designer** app.

To import an actor or critic, on the corresponding **Agent** tab, click **Import**. Then, under either **Actor** or **Critic**, select an actor or critic object with action and observation specifications that are compatible with the specifications of the agent.

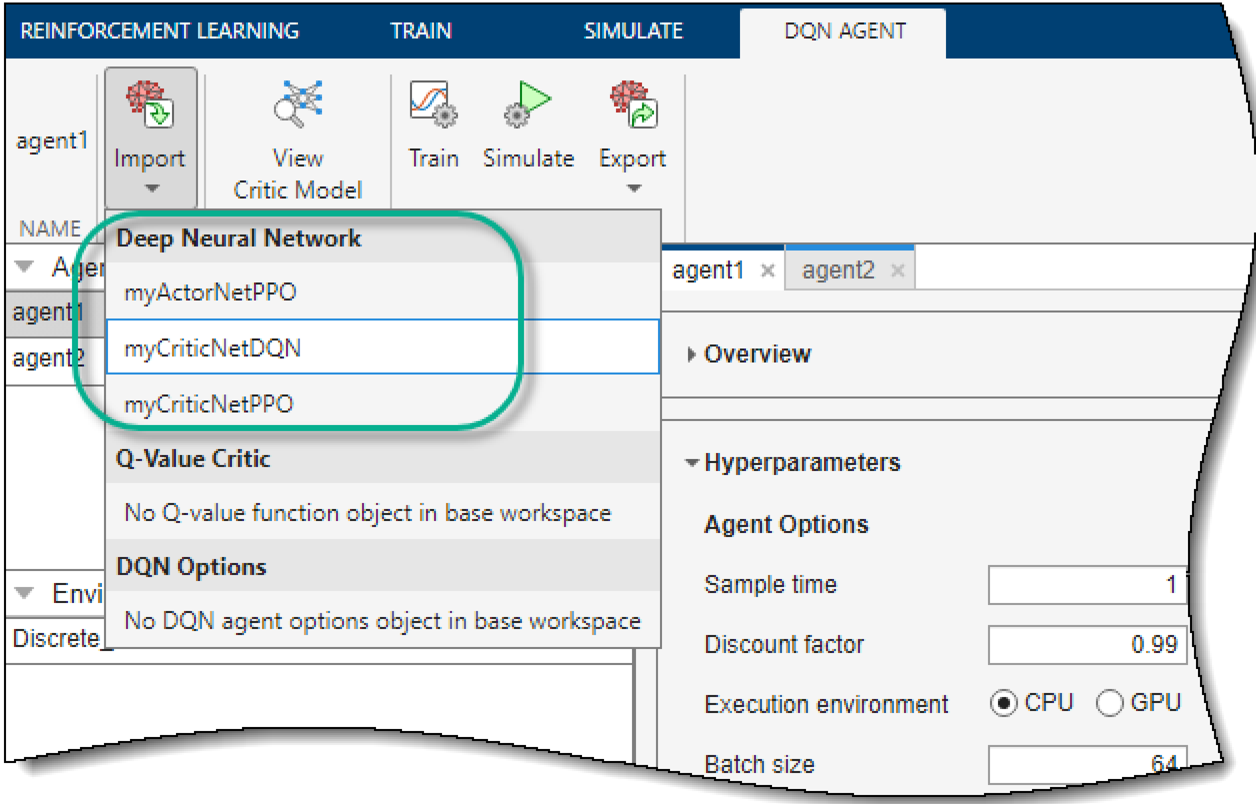



The app replaces the existing actor or critic in the agent with the selected one. If you import a critic for a TD3 agent, the app replaces the network for both critics.

## Modify Deep Neural Networks

To use a nondefault deep neural network for an actor or critic, you must import the network from the MATLAB workspace. One common strategy is to export the default deep neural network, modify it using the **Deep Network Designer** app, and then import it back into **Reinforcement Learning Designer**. For more information on creating deep neural networks for actors and critics, see “Create Policies and Value Functions” on page 4-2.

To import a deep neural network, on the corresponding **Agent** tab, click **Import**. Then, under either **Actor Neural Network** or **Critic Neural Network**, select a network with input and output layers that are compatible with the observation and action specifications of the agent.



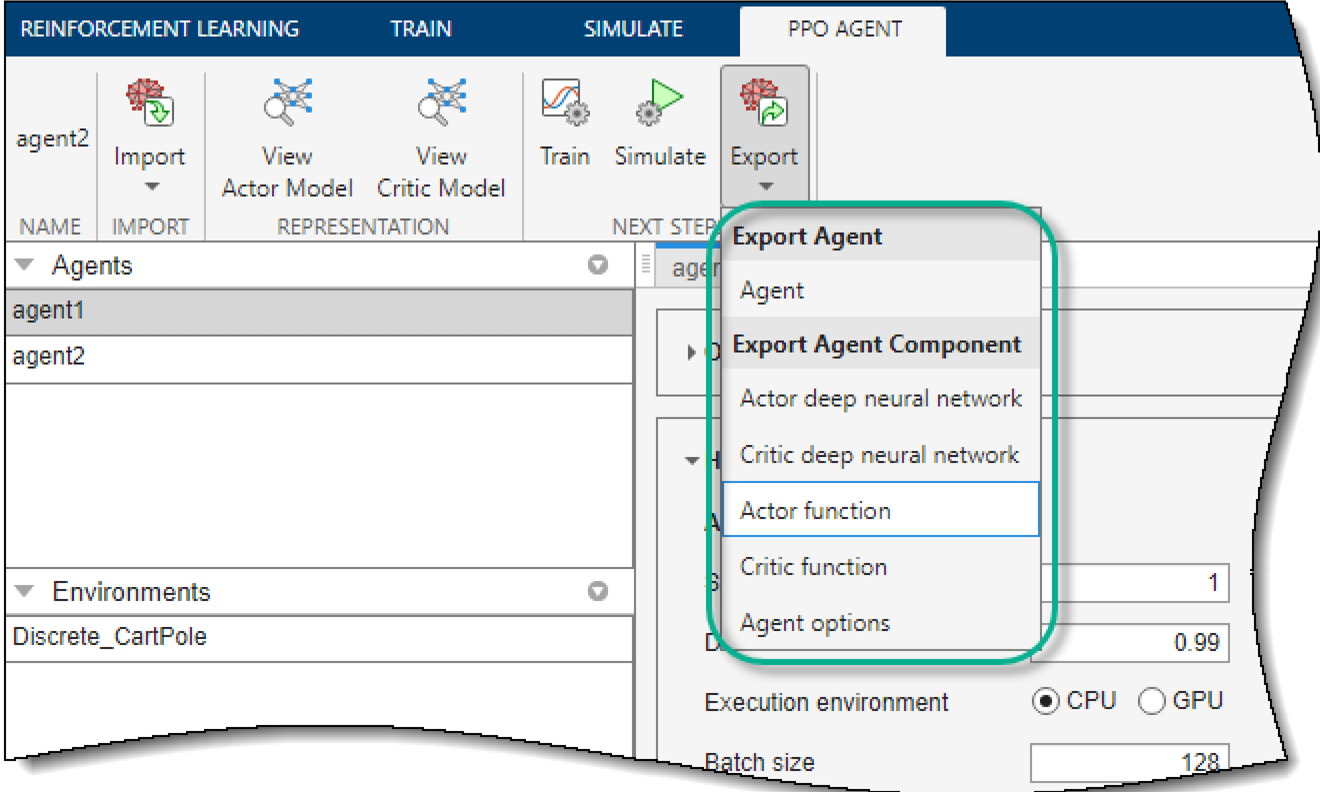

The app replaces the deep neural network in the corresponding actor or agent. If you import a critic network for a TD3 agent, the app replaces the network for both critics.

## Export Agents and Agent Components

For a given agent, you can export any of the following to the MATLAB workspace.

- Agent
- Agent options
- Actor or critic
- Deep neural network in the actor or critic

To export an agent or agent component, on the corresponding **Agent** tab, click **Export**. Then, select the item to export.




The app saves a copy of the agent or agent component in the MATLAB workspace.

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`analyzeNetwork`

## Related Examples

- “Design and Train Agent Using Reinforcement Learning Designer” on page 5-12

## More About

- “Reinforcement Learning Agents” on page 3-2
- “Create or Import MATLAB Environments in Reinforcement Learning Designer” on page 2-5
- “Create or Import Simulink Environments in Reinforcement Learning Designer” on page 2-11
- “Deep Q-Network (DQN) Agents” on page 3-23
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents” on page 3-44
- “Proximal Policy Optimization (PPO) Agents” on page 3-49

# Q-Learning Agents

The Q-learning algorithm is a model-free, online, off-policy reinforcement learning method. A Q-learning agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. For a given observation, the agent selects and outputs the action for which the estimated return is greatest.

**Note** Q-learning agents do not support recurrent networks.

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

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

| Observation Space | Action Space |
|---|---|
| Continuous or discrete | Discrete |

Q agents use the following critic.

| Critic | Actor |
|---|---|
| Q-value function critic $Q(S,A)$, which you create using `rlQValueFunction` or `rlVectorQValueFunction` | Q agents do not use an actor |

During training, the agent explores the action space using epsilon-greedy exploration. During each control interval the agent selects a random action with probability $\epsilon$, otherwise it selects the action for which the value function greatest with probability $1-\epsilon$.

## Critic Function Approximator

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

For critics that use table-based value functions, the parameters in $\phi$ are the actual $Q(S,A)$ values in the table.

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

During training, the agent tunes the parameter values in $\phi$. After training, the parameters remain at their tuned value and the trained value function approximator is stored in critic $Q(S,A)$.

## Agent Creation

To create a Q-learning agent:

1. Create a critic using an `rlQValueFunction` object.
2. Specify agent options using an `rlQAgentOptions` object.

3. Create the agent using an `rlQAgent` object.

## Training Algorithm

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

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

  1. Get the initial observation $S$ from the environment.
  2. Repeat the following for each step of the episode until $S$ is a terminal state.

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

$$A = \underset{A}{\operatorname{argmax}} Q(S, A; \phi)$$

     To specify $\epsilon$ and its decay rate, use the `EpsilonGreedyExploration` option.

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

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

$$y = R + \gamma \max_{A} Q(S', A; \phi)$$

     To set the discount factor $\gamma$, use the `DiscountFactor` option.

     d. Compute the difference $\Delta Q$ between the value function target and the current $Q(S,A;\phi)$ value.

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

     e. Update the critic using the learning rate $\alpha$. Specify the learning rate when you create the critic by setting the `LearnRate` option in the `rlCriticOptimizerOptions` property within the agent options object.

        - For table-based critics, update the corresponding $Q(S,A)$ value in the table.

$$Q(S, A) = Q(S, A; \phi) + \alpha \cdot \Delta Q$$

        - For all other types of critics, compute the gradients $\Delta\phi$ of the loss function with respect to the parameters $\phi$. Then, update the parameters based on the computed gradients. In this case, the loss function is the square of $\Delta Q$.

$$\Delta\phi = \frac{1}{2}\nabla_{\phi}(\Delta Q)^2$$

$$\phi = \phi + \alpha \cdot \Delta\phi$$

     f. Set the observation $S$ to $S'$.

## See Also

**Objects**
`rlQAgent` | `rlQAgentOptions`

## Related Examples


- "Train Reinforcement Learning Agents" on page 5-3
- "Train Reinforcement Learning Agent in Basic Grid World" on page 1-14


## More About


- "Reinforcement Learning Agents" on page 3-2
- "Create Policies and Value Functions" on page 4-2

# SARSA Agents

The SARSA algorithm is a model-free, online, on-policy reinforcement learning method. A SARSA agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. For a given observation, the agent selects and outputs the action for which the estimated return is greatest.

**Note** SARSA agents do not support recurrent networks.

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

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

| Observation Space | Action Space |
| --- | --- |
| Continuous or discrete | Discrete |

SARSA agents use the following critic.

| Critic | Actor |
| --- | --- |
| Q-value function critic $Q(S,A)$, which you create using `rlQValueFunction` or `rlVectorQValueFunction` | SARSA agents do not use an actor. |

During training, the agent explores the action space using epsilon-greedy exploration. During each control interval the agent selects a random action with probability $\epsilon$, otherwise it selects the action for which the value function greatest with probability $1–\epsilon$.

## Critic Function Approximator

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

For critics that use table-based value functions, the parameters in $\phi$ are the actual $Q(S,A)$ values in the table.

For more information on creating critics for value function approximation, see “Create Policies and Value Functions” on page 4-2.

During training, the agent tunes the parameter values in $\phi$. After training, the parameters remain at their tuned value and the trained value function approximator is stored in critic $Q(S,A)$.

## Agent Creation

To create a SARSA agent:

1. Create a critic using an `rlQValueFunction` object.
2. Specify agent options using an `rlSARSAAgentOptions` object.

3 Create the agent using an `rlSARSAAgent` object.

## Training Algorithm

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

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

1. Get the initial observation $S$ from the environment.
2. For the current observation $S$, select a random action $A$ with probability $\epsilon$. Otherwise, select the action for which the critic value function is greatest.

$$A = \underset{A}{\operatorname{argmax}} Q(S, A; \phi)$$

   To specify $\epsilon$ and its decay rate, use the `EpsilonGreedyExploration` option.

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

   a. Execute action $A_0$. Observe the reward $R$ and next observation $S'$.
   b. For the current observation $S'$, select a random action $A'$ with probability $\epsilon$. Otherwise, select the action for which the critic value function is greatest.

$$A' = \underset{A'}{\operatorname{argmax}} Q(S', A'; \phi)$$

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

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

   To set the discount factor $\gamma$, use the `DiscountFactor` option.

   d. Compute the difference $\Delta Q$ between the value function target and the current $Q(S,A;\phi)$ value.

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

   e. Update the critic using the learning rate $\alpha$. Specify the learning rate when you create the critic by setting the `LearnRate` option in the `rlCriticOptimizerOptions` property within the agent options object.

      - For table-based critics, update the corresponding $Q(S,A)$ value in the table.

$$Q(S, A) = Q(S, A; \phi) + \alpha \cdot \Delta Q$$

      - For all other types of critics, compute the gradients $\Delta\phi$ of the loss function with respect to the parameters $\phi$. Then, update the parameters based on the computed gradients. In this case, the loss function is the square of $\Delta Q$.

$$\Delta\phi = \frac{1}{2} \nabla_{\phi} (\Delta Q)^2$$

$$\phi = \phi + \alpha \cdot \Delta\phi$$

   f. Set the observation $S$ to $S'$.
   g. Set the action $A$ to $A'$.

## See Also

**Objects**
`rlSARSAAgent` | `rlSARSAAgentOptions`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3
- “Train Reinforcement Learning Agent in Basic Grid World” on page 1-14

## More About

- “Reinforcement Learning Agents” on page 3-2
- “Create Policies and Value Functions” on page 4-2

# Deep Q-Network (DQN) Agents

The deep Q-network (DQN) algorithm is a model-free, online, off-policy reinforcement learning method. A DQN agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. DQN is a variant of Q-learning. For more information on Q-learning, see “Q-Learning Agents” on page 3-17.

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

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

| Observation Space | Action Space |
| --- | --- |
| Continuous or discrete | Discrete |

DQN agents use the following critic.

| Critic | Actor |
| --- | --- |
| Q-value function critic $Q(S,A)$, which you create using `rlQValueFunction` or `rlVectorQValueFunction` | DQN agents do not use an actor. |

During training, the agent:

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

## Critic Function Approximator

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

- Critic $Q(S,A;\phi)$ — The critic, with parameters $\phi$, takes observation $S$ and action $A$ as inputs and returns the corresponding expectation of the long-term reward.
- Target critic $Q_t(S,A;\phi_t)$ — To improve the stability of the optimization, the agent periodically updates the target critic parameters $\phi_t$ using the latest critic parameter values.

Both $Q(S,A;\phi)$ and $Q_t(S,A;\phi_t)$ have the same structure and parameterization.

For more information on creating critics for value function approximation, see “Create Policies and Value Functions” on page 4-2.

During training, the agent tunes the parameter values in $\phi$. After training, the parameters remain at their tuned value and the trained value function approximator is stored in critic $Q(S,A)$.

## Agent Creation

You can create and train DQN agents at the MATLAB command line or using the **Reinforcement Learning Designer** app. For more information on creating agents using **Reinforcement Learning Designer**, see "Create Agents Using Reinforcement Learning Designer" on page 3-9.

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

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

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

1. Create a critic using an `rlQValueFunction` object.
2. Specify agent options using an `rlDQNAgentOptions` object.
3. Create the agent using an `rlDQNAgent` object.

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

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

## Training Algorithm

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

- Initialize the critic $Q(s,a;\phi)$ with random parameter values $\phi$, and initialize the target critic parameters $\phi_t$ with the same values. $\phi_t = \phi$.
- For each training time step:

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

  $$A = \underset{A}{\operatorname{argmax}} Q(S, A; \phi)$$

     To specify $\epsilon$ and its decay rate, use the `EpsilonGreedyExploration` option.
  2. Execute action $A$. Observe the reward $R$ and next observation $S'$.
  3. Store the experience $(S,A,R,S')$ in the experience buffer.

4. Sample a random mini-batch of $M$ experiences $(S_i,A_i,R_i,S'_i)$ from the experience buffer. To specify $M$, use the `MiniBatchSize` option.
5. If $S'_i$ is a terminal state, set the value function target $y_i$ to $R_i$. Otherwise, set it to

$$\begin{aligned} A_{\max} &= \underset{A'}{\arg\max}\, Q(S_i', A'; \phi) \\ y_i &= R_i + \gamma Q_t(S_i', A_{\max}; \phi_t) \end{aligned} \quad \text{(double DQN)}$$

$$y_i = R_i + \gamma \max_{A'} Q_t(S_i', A'; \phi_t) \quad \text{(DQN)}$$

   To set the discount factor $\gamma$, use the `DiscountFactor` option. To use double DQN, set the `UseDoubleDQN` option to `true`.
6. Update the critic parameters by one-step minimization of the loss $L$ across all sampled experiences.

$$L = \frac{1}{2M} \sum_{i=1}^{M} \left(y_i - Q(S_i, A_i; \phi)\right)^2$$

7. Update the target critic parameters depending on the target update method. For more information, see “Target Update Methods” on page 3-25.
8. Update the probability threshold $\epsilon$ for selecting a random action based on the decay rate you specify in the `EpsilonGreedyExploration` option.

## Target Update Methods

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

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

$$\phi_t = \tau\phi + (1 - \tau)\phi_t$$

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

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

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

## References

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

## See Also

**Objects**
`rlDQNAgent` | `rlDQNAgentOptions`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “Reinforcement Learning Agents” on page 3-2
- “Create Policies and Value Functions” on page 4-2

# Policy Gradient (PG) Agents

The policy gradient (PG) algorithm is a model-free, online, on-policy reinforcement learning method. A policy gradient agent is a policy-based reinforcement learning agent that uses the REINFORCE algorithm to search for an optimal policy that maximizes the expected cumulative long-term reward.

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

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

| Observation Space | Action Space |
| --- | --- |
| Discrete or continuous | Discrete or continuous |

Policy gradient agents use the following actor and critic.

| Critic (if a baseline is used) | Actor |
| --- | --- |
| Value function critic $V(S)$, which you create using `rlValueFunction` | Stochastic policy actor $\pi(S)$, which you create using `rlDiscreteCategoricalActor` (for a for discrete action space) or `rlContinuousGaussianActor` (for a continuous action space). |

During training, a PG agent:

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

If the `UseExplorationPolicy` option of the agent is set to `false`, the action with maximum likelihood is always used in `sim` and `generatePolicyFunction`. As a result, the simulated agent and generated policy behave deterministically.

If the `UseExplorationPolicy` is set to `true` the agent selects its actions by sampling its probability distribution. As a result the policy is stochastic and the agent explores its observation space.

This option affects only simulation and deployment; it does not affect training.

## Actor and Critic Function Approximators

Policy gradient agents represent the policy using an actor function approximator $\pi(A|S;\theta)$ with parameters $\theta$. The actor outputs the conditional probability of taking each action $A$ when in state $S$ as one of the following:

- Discrete action space — The probability of taking each discrete action. The sum of these probabilities across all actions is 1.
- Continuous action space — The mean and standard deviation of the Gaussian probability distribution for each continuous action.

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

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

During training, the agent tunes the parameter values in $\theta$. After training, the parameters remain at their tuned value and the trained actor function approximator is stored in $\pi(A|S)$.

## Agent Creation

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

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

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

1. Create an actor using an `rlDiscreteCategoricalActor` (for a for discrete action space) or `rlContinuousGaussianActor` (for a continuous action space) object.
2. If you are using a baseline function, create a critic using an `rlValueFunction` object.
3. Specify agent options using the `rlPGAgentOptions` object.
4. Create the agent using an `rlPGAgent` object.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

## Training Algorithm

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

### REINFORCE Algorithm

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

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

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

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

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

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

$$d\theta = \sum_{t=1}^{T-1} G_t \nabla_\theta \ln\pi(S_t; \theta)$$

5. Update the actor parameters by applying the gradients.

$$\theta = \theta + \alpha d\theta$$

Here, $\alpha$ is the learning rate of the actor. Specify the learning rate when you create the actor by setting the `LearnRate` option in the `rlActorOptimizerOptions` property within the agent options object. For simplicity, this step shows a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer you specify using in the `rlOptimizerOptions` object assigned to the `rlActorOptimizerOptions` property.

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

#### REINFORCE with Baseline Algorithm

1. Initialize the actor $\pi(S;\theta)$ with random parameter values in $\theta$.
2. Initialize the critic $V(S;\phi)$ with random parameter values in $\phi$.
3. For each training episode, generate the episode experience by following the actor policy $\pi(S)$. The episode experience consists of the sequence

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

4. For $t$ = 1, 2, …, $T$:

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

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

   - Compute the advantage function $\delta_t$ using the baseline value function estimate from the critic.

$$\delta_t = G_t - V(S_t; \phi)$$

5. Accumulate the gradients for the critic network.

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

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

$$d\theta = \sum_{t=1}^{T-1} \delta_t \nabla_\theta \ln\pi(S_t;\theta)$$

7. Update the critic parameters $\phi$.

   $$\phi = \phi + \beta d\phi$$

   Here, $\beta$ is the learning rate of the critic. Specify the learning rate when you create the critic by setting the `LearnRate` option in the `rlCriticOptimizerOptions` property within the agent options object.
8. Update the actor parameters $\theta$.

   $$\theta = \theta + \alpha d\theta$$
9. Repeat steps 3 through 8 for each training episode until training is complete.

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

## References

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

## See Also

**Objects**
`rlPGAgent` | `rlPGAgentOptions`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “Reinforcement Learning Agents” on page 3-2
- “Create Policies and Value Functions” on page 4-2

# Actor-Critic (AC) Agents

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

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

Actor-critic agents can be trained in environments with the following observation and action spaces.

| Observation Space | Action Space |
| --- | --- |
| Discrete or continuous | Discrete or continuous |

Actor-critic agents use the following actor and critics.

| Critic | Actor |
| --- | --- |
| Value function critic $V(S)$, which you create using `rlValueFunction` | Stochastic policy actor $\pi(S)$, which you create using `rlDiscreteCategoricalActor` (for discrete action spaces) or `rlContinuousGaussianActor` (for continuous action spaces) |

During training, an actor-critic agent:

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

If the `UseExplorationPolicy` option of the agent is set to `false` the action with maximum likelihood is always used in `sim` and `generatePolicyFunction`. As a result, the simulated agent and generated policy behave deterministically.

If the `UseExplorationPolicy` is set to `true` the agent selects its actions by sampling its probability distribution. As a result the policy is stochastic and the agent explores its observation space.

This option affects only simulation and deployment; it does not affect training.

## Actor and Critic Function Approximators

To estimate the policy and value function, an actor-critic agent maintains two function approximators.

- Actor $\pi(A|S;\theta)$ — The actor, with parameters $\theta$, outputs the conditional probability of taking each action $A$ when in state $S$ as one of the following:
  - Discrete action space — The probability of taking each discrete action. The sum of these probabilities across all actions is 1.
  - Continuous action space — The mean and standard deviation of the Gaussian probability distribution for each continuous action.

- Critic $V(S;\phi)$ — The critic, with parameters $\phi$, takes observation $S$ and returns the corresponding expectation of the discounted long-term reward.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

During training, the agent tunes the parameter values in $\theta$. After training, the parameters remain at their tuned value and the trained actor function approximator is stored in $\pi(A|S)$.

## Agent Creation

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

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

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

1. Create an actor using an `rlDiscreteCategoricalActor` (for discrete action spaces) or an `rlContinuousGaussianActor` (for continuous action spaces) object.
2. Create a critic using an `rlValueFunction` object.
3. Specify agent options using an `rlACAgentOptions` object.
4. Create the agent using an `rlACAgent` object.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

## Training Algorithm

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

1. Initialize the actor $\pi(A|S;\theta)$ with random parameter values $\theta$.
2. Initialize the critic $V(S;\phi)$ with random parameter values $\phi$.
3. Generate $N$ experiences by following the current policy. The episode experience sequence is

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

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

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

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

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

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

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

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

To specify the discount factor $\gamma$, use the `DiscountFactor` option.

5. Compute the advantage function $D_t$.

$$D_t = G_t - V(S_t; \phi)$$

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

$$d\theta = \sum_{t=1}^{N} \nabla_{\theta_\mu} \ln\pi(A|S_t; \theta) \cdot D_t$$

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

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

8. Update the actor parameters by applying the gradients.

$$\theta = \theta + \alpha d\theta$$

Here, $\alpha$ is the learning rate of the actor. Specify the learning rate when you create the actor by setting the `LearnRate` option in the `rlActorOptimizerOptions` property within the agent options object.

9. Update the critic parameters by applying the gradients.

$$\phi = \phi + \beta d\phi$$

Here, $\beta$ is the learning rate of the critic. Specify the learning rate when you create the critic by setting the `LearnRate` option in the `rlCriticOptimizerOptions` property within the agent options object.

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

For simplicity, the actor and critic updates in this algorithm description show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer you specify using in the `rlOptimizerOptions` object assigned to the `rlCriticOptimizerOptions` property.

## References

[1] Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning." *ArXiv:1602.01783 [Cs]*, February 4, 2016. https://arxiv.org/abs/1602.01783.

## See Also

**Objects**
`rlACAgent` | `rlACAgentOptions`

## Related Examples

- "Train Reinforcement Learning Agents" on page 5-3

## More About

- "Reinforcement Learning Agents" on page 3-2
- "Create Policies and Value Functions" on page 4-2

# Soft Actor-Critic (SAC) Agents

The soft actor-critic (SAC) algorithm is a model-free, online, off-policy, actor-critic reinforcement learning method. The SAC algorithm computes an optimal policy that maximizes both the long-term expected reward and the entropy of the policy. The policy entropy is a measure of policy uncertainty given the state. A higher entropy value promotes more exploration. Maximizing both the expected cumulative long term reward and the entropy helps to balance between exploitation and exploration of the environment.

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

The implementation of the SAC agent in Reinforcement Learning Toolbox software uses two Q-value function critics, which prevents overestimation of the value function. Other implementations of the SAC algorithm use an additional value function critic.

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

| Observation Space | Action Space |
|---|---|
| Discrete or continuous | Continuous |

SAC agents use the following actor and critic.

| Critics | Actor |
|---|---|
| Q-value function critics $Q(S,A)$, which you create using `rlQValueFunction` | Stochastic policy actor $\pi(S)$, which you create using `rlContinuousGaussianActor` |

During training, a SAC agent:

- Updates the actor and critic properties at regular intervals during learning.
- Estimates the mean and standard deviation of a Gaussian probability distribution for the continuous action space, then randomly selects actions based on the distribution.
- Updates an entropy weight term that balances the expected return and the entropy of the policy.
- Stores past experience using a circular experience buffer. The agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.

If the `UseExplorationPolicy` option of the agent is set to `false` the action with maximum likelihood is always used in `sim` and `generatePolicyFunction`. As a result, the simulated agent and generated policy behave deterministically.

If the `UseExplorationPolicy` is set to `true` the agent selects its actions by sampling its probability distribution. As a result the policy is stochastic and the agent explores its observation space.

This option affects only simulation and deployment; it does not affect training.

## Actor and Critic Function Approximators

To estimate the policy and value function, a SAC agent maintains the following function approximators.

- Stochastic actor $\pi(A|S;\theta)$ — The actor, with parameters $\theta$, outputs the mean ans standard deviation of conditional Gaussian probability of taking each continuous action $A$ when in state $S$.
- One or two Q-value critics $Q_k(S,A;\phi_k)$ — The critics, each with parameters $\phi_k$, take observation $S$ and action $A$ as inputs and return the corresponding expectation of the value function, which includes both the long-term reward and entropy.
- One or two target critics $Q_{tk}(S,A;\phi_{tk})$ — To improve the stability of the optimization, the agent periodically sets the target critic parameters $\phi_{tk}$ to the latest corresponding critic parameter values. The number of target critics matches the number of critics.

When you use two critics, $Q_1(S,A;\phi_1)$ and $Q_2(S,A;\phi_2)$, each critic can have different structures. When the critics have the same structure, they must have different initial parameter values.

Each critic $Q_k(S,A;\phi_k)$ and corresponding target critic $Q_{tk}(S,A;\phi_{tk})$ must have the same structure and parameterization.

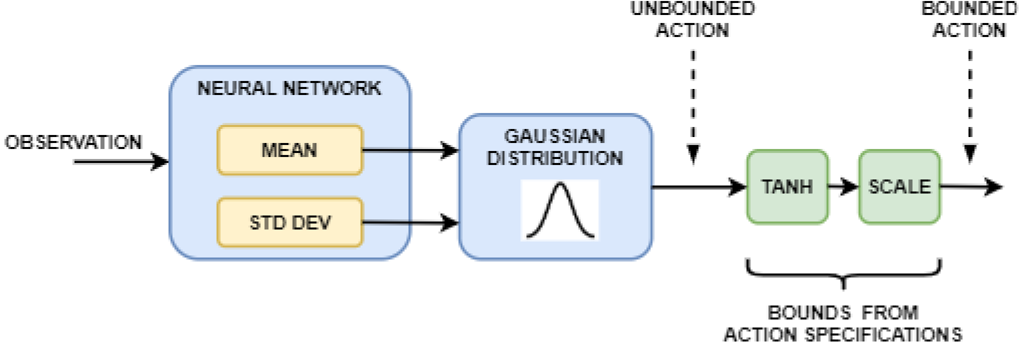
For more information on creating actors and critics for function approximation, see "Create Policies and Value Functions" on page 4-2.

During training, the agent tunes the parameter values in $\theta$. After training, the parameters remain at their tuned value and the trained actor function approximator is stored in $\pi(A|S)$.

### Action Generation

The actor in a SAC agent generates mean and standard deviation outputs. To select an action, the actor first randomly selects an unbounded action from a Gaussian distribution with these parameters. During training, the SAC agent uses the unbounded probability distribution to compute the entropy of the policy for the given observation.

If the action space of the SAC agent is bounded, the actor generates bounded actions by applying *tanh* and *scaling* operations to the unbounded action.




## Agent Creation

You can create and train SAC agents at the MATLAB command line or using the **Reinforcement Learning Designer** app. For more information on creating agents using **Reinforcement Learning Designer**, see "Create Agents Using Reinforcement Learning Designer" on page 3-9.

At the command line, you can create a SAC agent with default actor and critic based on the observation and action specifications from the environment. To do so, perform the following steps.

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

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

1. Create a stochastic actor using an `rlContinuousGaussianActor` object. For SAC agents, in order to properly scale the mean values to the desired action range, the actor network must not contain a `tanhLayer` and `scalingLayer` as last two layers in the output path for the mean values. However, in order to ensure non-negativity of the standard deviation values, the actor network must contain a `reluLayer` as a last layer in the output path for the standard deviation values.
2. Create one or two critics using `rlQValueFunction` objects.
3. Specify agent options using an `rlSACAgentOptions` object.
4. Create the agent using an `rlSACAgent` object.

SAC agents do not support actors and critics that use recurrent deep neural networks as function approximators.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

## Training Algorithm

SAC agents use the following training algorithm, in which they periodically update their actor and critic models and entropy weight. To configure the training algorithm, specify options using an `rlSACAgentOptions` object. Here, $K = 2$ is the number of critics and $k$ is the critic index.

- Initialize each critic $Q_k(S,A;\phi_k)$ with random parameter values $\phi_k$, and initialize each target critic with the same random parameter values: $\phi_{tk} = \phi_k$.
- Initialize the actor $\pi(S;\theta)$ with random parameter values $\theta$.
- Perform a warm start by taking a sequence of actions following the initial random policy in $\pi(S)$. For each action, store the experience in the experience buffer. To specify the number of warm up actions, use the `NumWarmStartSteps` option.
- For each training time step:

  1. For the current observation $S$, select action $A$ using the policy in $\pi(S;\theta)$.
  2. Execute action $A$. Observe the reward $R$ and next observation $S'$.
  3. Store the experience $(S,A,R,S')$ in the experience buffer.
  4. Sample a random mini-batch of $M$ experiences $(S_i,A_i,R_i,S'_i)$ from the experience buffer. To specify $M$, use the `MiniBatchSize` option.

5. Every $D_C$ time steps, update the parameters of each critic by minimizing the loss $L_k$ across all sampled experiences. To specify $D_C$, use the `CriticUpdateFrequency` option.

$$L_k = \frac{1}{2M} \sum_{i=1}^{M} (y_i - Q_k(S_i, A_i; \phi_k))^2$$

   If $S'_i$ is a terminal state, the value function target $y_i$ is equal to the experience reward $R_i$. Otherwise, the value function target is the sum of $R_i$, the minimum discounted future reward from the critics, and the weighted entropy.

$$y_i = R_i + \gamma * \min_k(Q_{tk}(S_i', A_i'; \phi_{tk})) - \alpha \ln\pi(S_i'; \theta)$$

   Here:

   - $A'_i$ is the bounded action derived from the unbounded output of the actor $\pi(S'_i)$.
   - $\gamma$ is the discount factor, which you specify using the `DiscountFactor` option.
   - $-\alpha \ln\pi(S; \theta)$ is the weighted policy entropy for the bounded output of the actor when in state $S$. $\alpha$ is the entropy loss weight, which you specify using the `EntropyLossWeight` option.

6. Every $D_A$ time steps, update the actor parameters by minimizing the following objective function. To set $D_A$, use the `PolicyUpdateFrequency` option.

$$J_\pi = \frac{1}{M} \sum_{i=1}^{M} \left(-\min_k(Q_k(S_i, A_i; \phi_k)) + \alpha \ln\pi(S_i; \theta)\right)$$

7. Every $D_A$ time steps, also update the entropy weight by minimizing the following loss function.

$$L_\alpha = \frac{1}{M} \sum_{i=1}^{M} (-\alpha \ln\pi(S_i; \theta) - \alpha \mathscr{H})$$

   Here, $\mathscr{H}$ is the target entropy, which you specify using the `EntropyWeightOptions.TargetEntropy` option.

8. Every $D_T$ steps, update the target critics depending on the target update method. To specify $D_T$, use the `TargetUpdateFrequency` option. For more information, see “Target Update Methods” on page 3-47.
9. Repeat steps 4 through 8 $N_G$ times, where $N_G$ is the number of gradient steps, which you specify using the `NumGradientStepsPerUpdate` option.

## Target Update Methods

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

- Smoothing — Update the target critic parameters at every time step using smoothing factor $\tau$. To specify the smoothing factor, use the `TargetSmoothFactor` option.

$$\phi_{tk} = \tau\phi_k + (1 - \tau)\phi_{tk}$$

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

$$\phi_{tk} = \phi_k$$

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

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

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

## References

[1] Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, et al. "Soft Actor-Critic Algorithms and Application." Preprint, submitted January 29, 2019. https://arxiv.org/abs/1812.05905.

## See Also

**Objects**
`rlSACAgent` | `rlSACAgentOptions`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “Reinforcement Learning Agents” on page 3-2
- “Create Policies and Value Functions” on page 4-2

# Deep Deterministic Policy Gradient (DDPG) Agents

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

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

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

| Observation Space | Action Space |
| --- | --- |
| Continuous or discrete | Continuous |

DDPG agents use the following actor and critic.

| Critic | Actor |
| --- | --- |
| Q-value function critic $Q(S,A)$, which you create using `rlQValueFunction` | Deterministic policy actor $\pi(S)$, which you create using `rlContinuousDeterministicActor` |

During training, a DDPG agent:

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

## Actor and Critic Functions

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

- Actor $\pi(S;\theta)$— The actor, with parameters $\theta$, takes observation $S$ and returns the corresponding action that maximizes the long-term reward.
- Target actor $\pi_t(S;\theta_t)$ — To improve the stability of the optimization, the agent periodically updates the target actor parameters $\theta_t$ using the latest actor parameter values.
- Critic $Q(S,A;\phi)$ — The critic, with parameters $\phi$, takes observation $S$ and action $A$ as inputs and returns the corresponding expectation of the long-term reward.
- Target critic $Q_t(S,A;\phi_t)$ — To improve the stability of the optimization, the agent periodically updates the target critic parameters $\phi_t$ using the latest critic parameter values.

Both $Q(S,A;\phi)$ and $Q_t(S,A;\phi_t)$ have the same structure and parameterization, and both $\pi(S;\theta)$ and $\pi_t(S;\theta_t)$ have the same structure and parameterization.

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

During training, the agent tunes the parameter values in $\theta$. After training, the parameters remain at their tuned value and the trained actor function approximator is stored in $\pi(S)$.

## Agent Creation

You can create and train DDPG agents at the MATLAB command line or using the **Reinforcement Learning Designer** app. For more information on creating agents using **Reinforcement Learning Designer**, see “Create Agents Using Reinforcement Learning Designer” on page 3-9.

At the command line, you can create a DDPG agent with default actor and critics based on the observation and action specifications from the environment. To do so, perform the following steps.

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

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

1. Create an actor using an `rlContinuousDeterministicActor` object.
2. Create a critic using an `rlQValueFunction` object.
3. Specify agent options using an `rlDDPGAgentOptions` object.
4. Create the agent using an `rlDDPGAgent` object.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

## Training Algorithm

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

- Initialize the critic $Q(S,A;\phi)$ with random parameter values $\phi$, and initialize the target critic parameters $\phi_t$ with the same values: $\phi_t = \phi$.
- Initialize the actor $\pi(S;\theta)$ with random parameter values $\theta$, and initialize the target actor parameters $\theta_t$ with the same values: $\theta_t = \theta$.
- For each training time step:

  1. For the current observation $S$, select action $A = \pi(S;\theta) + N$, where $N$ is stochastic noise from the noise model. To configure the noise model, use the `NoiseOptions` option.
  2. Execute action $A$. Observe the reward $R$ and next observation $S'$.
  3. Store the experience $(S,A,R,S')$ in the experience buffer. The length of the experience buffer is specified in the `ExperienceBufferLength` property of the `rlDDPGAgentOptions` object.

4. Sample a random mini-batch of $M$ experiences ($S_i$,$A_i$,$R_i$,$S'_i$) from the experience buffer. To specify $M$, use the `MiniBatchSize` property of the `rlDDPGAgentOptions` object.
5. If $S'_i$ is a terminal state, set the value function target $y_i$ to $R_i$. Otherwise, set it to

$$y_i = R_i + \gamma Q_t(S_i', \pi_t(S_i'; \theta_t); \phi_t)$$

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

   To compute the cumulative reward, the agent first computes a next action by passing the next observation $S'_i$ from the sampled experience to the target actor. The agent finds the cumulative reward by passing the next action to the target critic.
6. Update the critic parameters by minimizing the loss $L$ across all sampled experiences.

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

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

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

$$G_{ai} = \nabla_A Q(S_i, A; \phi) \quad \text{where } A = \pi(S_i; \theta)$$

$$G_{\pi i} = \nabla_\theta \pi(S_i; \theta)$$

   Here, $G_{ai}$ is the gradient of the critic output with respect to the action computed by the actor network, and $G_{\pi i}$ is the gradient of the actor output with respect to the actor parameters. Both gradients are evaluated for observation $S_i$.
8. Update the target actor and critic parameters depending on the target update method. For more information see “Target Update Methods” on page 3-42.

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

## Target Update Methods

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

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

$$\phi_t = \tau\phi + (1 - \tau)\phi_t \quad \text{(critic parameters)}$$

$$\theta_t = \tau\theta + (1 - \tau)\theta_t \quad \text{(actor parameters)}$$

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

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

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

## References

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

## See Also

**Objects**
rlDDPGAgent | rlDDPGAgentOptions

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “Reinforcement Learning Agents” on page 3-2
- “Create Policies and Value Functions” on page 4-2

# Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents

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

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

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

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

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

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

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

| Observation Space | Action Space |
| --- | --- |
| Continuous or discrete | Continuous |

TD3 agents use the following actor and critics.

| Critics | Actor |
| --- | --- |
| One or more Q-value function critics $Q(S,A)$, which you create using `rlQValueFunction` | Deterministic policy actor $\pi(S)$, which you create using `rlContinuousDeterministicActor` |

During training, a TD3 agent:

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

## Actor and Critic Functions

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

- Deterministic actor $\pi(S;\theta)$ — The actor, with parameters $\theta$, takes observation $S$ and returns the corresponding action that maximizes the long-term reward.
- Target actor $\pi_t(S;\theta_t)$ — To improve the stability of the optimization, the agent periodically updates the target actor parameters $\theta_t$ using the latest actor parameter values.
- One or two Q-value critics $Q_k(S,A;\phi_k)$ — The critics, each with different parameters $\phi_k$, take observation $S$ and action $A$ as inputs and returns the corresponding expectation of the long-term reward.
- One or two target critics $Q_{tk}(S,A;\phi_{tk})$ — To improve the stability of the optimization, the agent periodically updates the target critic parameters $\phi_{tk}$ using the latest corresponding critic parameter values. The number of target critics matches the number of critics.

Both $\pi(S;\theta)$ and $\pi_t(S;\theta_t)$ have the same structure and parameterization.

For each critic, $Q_k(S,A;\phi_k)$ and $Q_{tk}(S,A;\phi_{tk})$ have the same structure and parameterization.

When using two critics, $Q_1(S,A;\phi_1)$ and $Q_2(S,A;\phi_2)$, each critic can have a different structure, though TD3 works best when the critics have the same structure. When the critics have the same structure, they must have different initial parameter values.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

During training, the agent tunes the parameter values in $\theta$. After training, the parameters remain at their tuned value and the trained actor function approximator is stored in $\pi(S)$.

## Agent Creation

You can create and train TD3 agents at the MATLAB command line or using the **Reinforcement Learning Designer** app. For more information on creating agents using **Reinforcement Learning Designer**, see “Create Agents Using Reinforcement Learning Designer” on page 3-9.

At the command line, you can create a TD3 agent with default actor and critics based on the observation and action specifications from the environment. To do so, perform the following steps.

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

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

1. Create an actor using an `rlContinuousDeterministicActor` object.
2. Create one or two critics using `rlQValueFunction` objects.

3. Specify agent options using an `rlTD3AgentOptions` object.
4. Create the agent using an `rlTD3Agent` object.

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

## Training Algorithm

TD3 agents use the following training algorithm, in which they update their actor and critic models at each time step. To configure the training algorithm, specify options using an `rlTD3AgentOptions` object. Here, $K = 2$ is the number of critics and $k$ is the critic index.

- Initialize each critic $Q_k(S,A;\phi_k)$ with random parameter values $\phi_k$, and initialize each target critic with the same random parameter values: $\phi_{tk} = \phi_k$.
- Initialize the actor $\pi(S;\theta)$ with random parameter values $\theta$, and initialize the target actor with the same parameter values: $\theta_t = \theta$.
- For each training time step:

  1. For the current observation $S$, select action $A = \pi(S;\theta) + N$, where $N$ is stochastic noise from the noise model. To configure the noise model, use the `ExplorationModel` option.
  2. Execute action $A$. Observe the reward $R$ and next observation $S'$.
  3. Store the experience $(S,A,R,S')$ in the experience buffer.
  4. Sample a random mini-batch of $M$ experiences $(S_i,A_i,R_i,S'_i)$ from the experience buffer. To specify $M$, use the `MiniBatchSize` option.
  5. If $S'_i$ is a terminal state, set the value function target $y_i$ to $R_i$. Otherwise, set it to

$$y_i = R_i + \gamma * \min_k(Q_{tk}(S_i', \text{clip}(\pi_t(S_i'; \theta_t) + \varepsilon); \phi_{tk}))$$

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

     To compute the cumulative reward, the agent first computes a next action by passing the next observation $S'_i$ from the sampled experience to the target actor. Then, the agent adds noise $\varepsilon$ to the computed action using the `TargetPolicySmoothModel`, and clips the action based on the upper and lower noise limits. The agent finds the cumulative rewards by passing the next action to the target critics.
  6. At every time training step, update the parameters of each critic by minimizing the loss $L_k$ across all sampled experiences.

$$L_k = \frac{1}{2M} \sum_{i=1}^{M} (y_i - Q_k(S_i, A_i; \phi_k))^2$$

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

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

$$G_{ai} = \nabla_{A} \min_{k}(Q_k(S_i, A; \phi)) \quad \text{where } A = \pi(S_i; \theta)$$

$$G_{\pi i} = \nabla_{\theta} \pi(S_i; \theta)$$

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

8. Every $D_2$ steps, update the target actor and critics depending on the target update method. To specify $D_2$, use the `TargetUpdateFrequency` option. For more information, see "Target Update Methods" on page 3-47.

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

## Target Update Methods

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

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

  $$\phi_{tk} = \tau\phi_k + (1 - \tau)\phi_{tk} \quad \text{(critic parameters)}$$

  $$\theta_t = \tau\theta + (1 - \tau)\theta_t \quad \text{(actor parameters)}$$

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

  $$\phi_{tk} = \phi_k$$

  $$\theta_t = \theta$$

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

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

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

## References

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

## See Also

**Objects**
`rlTD3Agent` | `rlTD3AgentOptions`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3
- “Train Biped Robot to Walk Using Reinforcement Learning Agents” on page 5-267

## More About

- “Reinforcement Learning Agents” on page 3-2
- “Create Policies and Value Functions” on page 4-2

# Proximal Policy Optimization (PPO) Agents

Proximal policy optimization (PPO) is a model-free, online, on-policy, policy gradient reinforcement learning method. This algorithm is a type of policy gradient training that alternates between sampling data through environmental interaction and optimizing a clipped surrogate objective function using stochastic gradient descent. The clipped surrogate objective function improves training stability by limiting the size of the policy change at each step [1].

PPO is a simplified version of TRPO. TRPO is more computationally expensive than PPO, but TRPO tends to be more robust than PPO if the environment dynamics are deterministic and the observation is low dimensional. For more information on TRPO agents, see “Trust Region Policy Optimization (TRPO) Agents” on page 3-55.

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

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

| Observation Space | Action Space |
| --- | --- |
| Discrete or continuous | Discrete or continuous |

PPO agents use the following actor and critics.

| Critic | Actor |
| --- | --- |
| Value function critic $V(S)$, which you create using `rlValueFunction` | Stochastic policy actor $\pi(S)$, which you create using `rlDiscreteCategoricalActor` (for discrete action spaces) or `rlContinuousGaussianActor` (for continuous action spaces) |

During training, a PPO agent:

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

If the `UseExplorationPolicy` option of the agent is set to `false` the action with maximum likelihood is always used in `sim` and `generatePolicyFunction`. As a result, the simulated agent and generated policy behave deterministically.

If the `UseExplorationPolicy` is set to `true` the agent selects its actions by sampling its probability distribution. As a result the policy is stochastic and the agent explores its observation space.

This option affects only simulation and deployment; it does not affect training.

## Actor and Critic Function Approximators

To estimate the policy and value function, a PPO agent maintains two function approximators.

- Actor $\pi(A|S;\theta)$ — The actor, with parameters $\theta$, outputs the conditional probability of taking each action $A$ when in state $S$ as one of the following:
  - Discrete action space — The probability of taking each discrete action. The sum of these probabilities across all actions is 1.
  - Continuous action space — The mean and standard deviation of the Gaussian probability distribution for each continuous action.
- Critic $V(S;\phi)$ — The critic, with parameters $\phi$, takes observation $S$ and returns the corresponding expectation of the discounted long-term reward.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

During training, the agent tunes the parameter values in $\theta$. After training, the parameters remain at their tuned value and the trained actor function approximator is stored in $\pi(A|S)$.

## Agent Creation

You can create and train PPO agents at the MATLAB command line or using the **Reinforcement Learning Designer** app. For more information on creating agents using **Reinforcement Learning Designer**, see “Create Agents Using Reinforcement Learning Designer” on page 3-9.

At the command line, you can create a PPO agent with default actor and critic based on the observation and action specifications from the environment. To do so, perform the following steps.

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

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

1. Create an actor using an `rlDiscreteCategoricalActor` object (for discrete action spaces) or an `rlContinuousGaussianActor` object (for continuous action spaces).
2. Create a critic using an `rlValueFunction` object.
3. If needed, specify agent options using an `rlPPOAgentOptions` object.
4. Create the agent using the `rlPPOAgent` function.

PPO agents support actors and critics that use recurrent deep neural networks as function approximators.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

## Training Algorithm

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

1. Initialize the actor $\pi(A|S;\theta)$ with random parameter values $\theta$.
2. Initialize the critic $V(S;\phi)$ with random parameter values $\phi$.
3. Generate $N$ experiences by following the current policy. The experience sequence is

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

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

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

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

   For each experience sequence that does not contain a terminal state, $N$ is equal to the `ExperienceHorizon` option value. Otherwise, $N$ is less than `ExperienceHorizon` and $S_N$ is the terminal state.
4. For each episode step $t = ts+1, ts+2, \ldots, ts+N$, compute the return and advantage function using the method specified by the `AdvantageEstimateMethod` option.

   - **Finite Horizon** (`AdvantageEstimateMethod = "finite-horizon"`) — Compute the return $G_t$, which is the sum of the reward for that step and the discounted future reward [2].

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

     Here, $b$ is `0` if $S_{ts+N}$ is a terminal state and `1` otherwise. That is, if $S_{ts+N}$ is not a terminal state, the discounted future reward includes the discounted state value function, computed using the critic network $V$.

     Compute the advantage function $D_t$.

   $$D_t = G_t - V(S_t;\phi)$$

   - **Generalized Advantage Estimator** (`AdvantageEstimateMethod = "gae"`) — Compute the advantage function $D_t$, which is the discounted sum of temporal difference errors [3].

   $$D_t = \sum_{k=t}^{ts+N-1} (\gamma\lambda)^{k-t} \delta_k$$

   $$\delta_k = R_t + b\gamma V(S_t;\phi)$$

     Here, $b$ is `0` if $S_{ts+N}$ is a terminal state and `1` otherwise. $\lambda$ is a smoothing factor specified using the `GAEFactor` option.

     Compute the return $G_t$.

$$G_t = D_t + V(S_t;\phi)$$

To specify the discount factor $\gamma$ for either method, use the DiscountFactor option.

5 Learn from mini-batches of experiences over $K$ epochs. To specify $K$, use the NumEpoch option. For each learning epoch:

   a Sample a random mini-batch data set of size $M$ from the current set of experiences. To specify $M$, use the MiniBatchSize option. Each element of the mini-batch data set contains a current experience and the corresponding return and advantage function values.

   b Update the critic parameters by minimizing the loss $L_{critic}$ across all sampled mini-batch data.

$$L_{critic}(\phi) = \frac{1}{2M}\sum_{i=1}^{M}(G_i - V(S_i;\phi))^2$$

   c Normalize the advantage values $D_i$ based on recent unnormalized advantage values.

   - If the NormalizedAdvantageMethod option is 'none', do not normalize the advantage values.

$$\widehat{D}_i \leftarrow D_i$$

   - If the NormalizedAdvantageMethod option is 'current', normalize the advantage values based on the unnormalized advantages in the current mini-batch.

$$\widehat{D}_i \leftarrow \frac{D_i - mean(D_1, D_2, \ldots, D_M)}{std(D_1, D_2, \ldots, D_M)}$$

   - If the NormalizedAdvantageMethod option is 'moving', normalize the advantage values based on the unnormalized advantages for the $N$ most recent advantages, including the current advantage value. To specify the window size $N$, use the AdvantageNormalizingWindow option.

$$\widehat{D}_i \leftarrow \frac{D_i - mean(D_1, D_2, \ldots, D_N)}{std(D_1, D_2, \ldots, D_N)}$$

   d Update the actor parameters by minimizing the actor loss function $L_{actor}$ across all sampled mini-batch data.

$$L_{actor}(\theta) = \frac{1}{M}\sum_{i=1}^{M}(-\min(r_i(\theta)\cdot D_i, c_i(\theta)\cdot D_i) + w\mathscr{H}_i(\theta, S_i))$$

$$r_i(\theta) = \frac{\pi(A_i|S_i;\theta)}{\pi(A_i|S_i;\theta_{old})}$$

$$c_i(\theta) = \max(\min(r_i(\theta), 1+\varepsilon), 1-\varepsilon)$$

   Here:

   - $D_i$ and $G_i$ are the advantage function and return value for the $i$th element of the mini-batch, respectively.
   - $\pi(A_i|S_i;\theta)$ is the probability of taking action $A_i$ when in state $S_i$, given the updated policy parameters $\theta$.
   - $\pi(A_i|S_i;\theta_{old})$ is the probability of taking action $A_i$ when in state $S_i$, given the previous policy parameters $\theta_{old}$ from before the current learning epoch.

- $\varepsilon$ is the clip factor specified using the `ClipFactor` option.
- $\mathscr{H}_i(\theta)$ is the entropy loss and $w$ is the entropy loss weight factor, specified using the `EntropyLossWeight` option. For more information on entropy loss, see “Entropy Loss” on page 3-53.

6. Repeat steps 3 through 5 until the training episode reaches a terminal state.

## Entropy Loss

To promote agent exploration, you can add an entropy loss term $w\mathscr{H}_i(\theta,S_i)$ to the actor loss function, where $w$ is the entropy loss weight and $\mathscr{H}_i(\theta,S_i)$ is the entropy.

The entropy value is higher when the agent is more uncertain about which action to take next. Therefore, maximizing the entropy loss term (minimizing the negative entropy loss) increases the agent uncertainty, thus encouraging exploration. To promote additional exploration, which can help the agent move out of local optima, you can specify a larger entropy loss weight.

For a discrete action space, the agent uses the following entropy value. In this case, the actor outputs the probability of taking each possible discrete action.

$$\mathscr{H}_i(\theta, S_i) = -\sum_{k=1}^{P} \pi(A_k|S_i;\theta)\ln\pi(A_k|S_i;\theta)$$

Here:

- $P$ is the number of possible discrete actions.
- $\pi(A_k|S_i;\theta)$ is the probability of taking action $A_k$ when in state $S_i$ following the current policy.

For a continuous action space, the agent uses the following entropy value. In this case, the actor outputs the mean and standard deviation of the Gaussian distribution for each continuous action.

$$\mathscr{H}_i(\theta, S_i) = \frac{1}{2}\sum_{k=1}^{C} \ln\left(2\pi \cdot e \cdot \sigma_{k,i}^2\right)$$

Here:

- $C$ is the number of continuous actions output by the actor.
- $\sigma_{k,i}$ is the standard deviation for action $k$ when in state $S_i$ following the current policy.

## References

[1] Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Proximal Policy Optimization Algorithms.” *ArXiv:1707.06347 [Cs]*, July 19, 2017. https://arxiv.org/abs/1707.06347.

[2] Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. “Asynchronous Methods for Deep Reinforcement Learning.” *ArXiv:1602.01783 [Cs]*, February 4, 2016. https://arxiv.org/abs/1602.01783.

[3] Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. “High-Dimensional Continuous Control Using Generalized Advantage Estimation.” *ArXiv:1506.02438 [Cs]*, October 20, 2018. https://arxiv.org/abs/1506.02438.

## See Also

**Objects**
`rlPPOAgent` | `rlPPOAgentOptions`

## Related Examples

- "Train Reinforcement Learning Agents" on page 5-3

## More About

- "Reinforcement Learning Agents" on page 3-2
- "Create Policies and Value Functions" on page 4-2

# Trust Region Policy Optimization (TRPO) Agents

Trust Region Policy Optimization (TRPO) is a model-free, online, on-policy, policy gradient reinforcement learning algorithm. TRPO alternates between sampling data through environmental interaction and updating the policy parameters by solving a constrained optimization problem. The KL-divergence between the old policy and the new policy is used as a constraint during optimization. As a result, this algorithm prevents significant performance drops compared to standard policy gradient methods by keeping the updated policy within a trust region close to the current policy [1].

**Note** TRPO agents do not support recurrent networks.

PPO is a simplified version of TRPO. TRPO is more computationally expensive than PPO, but TRPO tends to be more robust than PPO if the environment dynamics are deterministic and the number of observations is low. For more information on PPO agents, see "Proximal Policy Optimization (PPO) Agents" on page 3-49.

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

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

| Observation Space | Action Space |
|---|---|
| Discrete or continuous | Discrete or continuous |

TRPO agents use the following actor and critic.

| Critic | Actor |
|---|---|
| Value function critic $V(S)$, which you create using `rlValueFunction` | Stochastic policy actor $\pi(S)$, which you create using `rlDiscreteCategoricalActor` (for discrete action spaces) or `rlContinuousGaussianActor` (for continuous action spaces) |

During training, a TRPO agent:

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

If the `UseExplorationPolicy` option of the agent is set to `false` the action with maximum likelihood is always used in `sim` and `generatePolicyFunction`. As a result, the simulated agent and generated policy behave deterministically.

If the `UseExplorationPolicy` is set to `true` the agent selects its actions by sampling its probability distribution. As a result the policy is stochastic and the agent explores its observation space.

This option affects only simulation and deployment; it does not affect training.

## Actor and Critic Function Approximators

To estimate the policy and value function, a TRPO agent maintains two function approximators.

- Actor $\pi(A|S;\theta)$ — The actor, with parameters $\theta$, outputs the conditional probability of taking each action $A$ when in state $S$ as one of the following:
  - Discrete action space — The probability of taking each discrete action. The sum of these probabilities across all actions is 1.
  - Continuous action space — The mean and standard deviation of the Gaussian probability distribution for each continuous action.
- Critic $V(S;\phi)$ — The critic, with parameters $\phi$, takes observation $S$ and returns the corresponding expectation of the discounted long-term reward.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

During training, the agent tunes the parameter values in $\theta$. After training, the parameters remain at their tuned value and the trained actor function approximator is stored in $\pi(A|S)$.

## Agent Creation

You can create and train TRPO agents at the MATLAB command line or using the **Reinforcement Learning Designer** app. For more information on creating agents using **Reinforcement Learning Designer**, see “Create Agents Using Reinforcement Learning Designer” on page 3-9.

At the command line, you can create a TRPO agent with default actor and critic based on the observation and action specifications from the environment. To do so, perform the following steps.

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

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

1. Create an actor using `rlDiscreteCategoricalActor` object (for discrete action spaces) or `rlContinuousGaussianActor` object (for continuous action spaces).
2. Create a critic using an `rlValueFunction` object.
3. If needed, specify agent options using an `rlTRPOAgentOptions` object.
4. Create the agent using the `rlTRPOAgent` function.

TRPO agents do not support actors and critics that use recurrent deep neural networks as function approximators. TRPO agents also do not support deep neural networks that use a `quadraticLayer`.

For more information on creating actors and critics for function approximation, see “Create Policies and Value Functions” on page 4-2.

## Trust Region Policy Optimization

Trust region policy optimization finds the actor parameters that minimize the following actor loss function.

$$L_{actor}(\theta) = -\frac{1}{M}\sum_{i=1}^{M}\left(\frac{\pi(A_i|S_i;\theta)}{\pi(A_i|S_i;\theta_{old})}D_i + w\mathcal{H}_i(\theta, S_i)\right)$$

Here:

- $M$ is the mini-batch size.
- $D_i$ is an advantage function.
- $\pi_i(A_i|S_i;\theta)$ is the probability of taking action $A_i$ following the current policy. This value is a specific value of the probability (discrete action) or of the probability density function (continuous action).
- $\pi(A_i|S_i;\theta_{old})$ is the probability of taking action $A_i$ following the old policy.
- $w\mathcal{H}_i(\theta,S_i)$ is an entropy loss term, where $w$ is the entropy loss weight and $\mathcal{H}_i(\theta,S_i)$ is the entropy. For more information, see “Entropy Loss” on page 3-60.

This minimization is subject to the following constraint.

$$\frac{1}{M}\sum_{i=1}^{M}D_{KL}(\theta_{old}, \theta, S_i) \leq \delta$$

Here:

- $D_{KL}(\theta_{old},\theta,S_i)$ is the Kullback-Leibler (KL) divergence between the old policy $\pi(A|S_i;\theta_{old})$ and current policy $\pi(A|S_i;\theta)$. $D_{KL}$ measures how much the probability distributions of the old and new policies differ. $D_{KL}$ is zero when the two distributions are identical.
- $\delta$ is the limit for $D_{KL}$ and controls how much the new policy can deviate from the old policy.

For agents with discrete action spaces, $D_{KL}$ is computed as follows, where $P$ is the number of actions.

$$D_{KL}(\theta_{old}, \theta, S_i) = \sum_{k=1}^{P}\pi(A_k|S_i;\theta_{old})\ln\left(\frac{\pi(A_k|S_i;\theta_{old})}{\pi(A_k|S_i;\theta)}\right)$$

For agents with continuous action spaces, $D_{KL}$ is computed as follows.

$$D_{KL}(\theta_{old}, \theta, S_i) = \frac{1}{P}\sum_{k=1}^{P}\left(\ln(\sigma_{\theta,k}) - \ln(\sigma_{\theta old,k}) + \frac{\sigma_{\theta old,k}^2 + (\mu_{\theta old,k} - \mu_{\theta,k})^2}{2\sigma_{\theta,k}^2} - 0.5\right)$$

Here:

- $\mu_{\theta,k}$ and $\sigma_{\theta,k}$ are the mean and standard deviation for the $k$th action output by the current actor policy $\pi(A_k|S_i;\theta)$.
- $\mu_{\theta old,k}$ and $\sigma_{\theta old,k}$ are the mean and standard deviation for the $k$th action output by the old policy $\pi(A_k|S_i;\theta_{old})$.

To approximate this optimization problem, the TRPO agent uses a linear approximation of $L_{actor}(\theta)$ and a quadratic approximation of $D_{KL}(\theta_{old},\theta,S_i)$. The approximations are computed by taking the Taylor series expansion around $\theta$.

$$\min_{\theta} L_{actor}(\theta) \approx g(\theta - \theta_{old}) = \nabla_{\theta} L_{actor}(\theta)|_{\theta = \theta_{old}} \cdot (\theta - \theta_{old})$$

$$subject\ to \quad \frac{1}{2}(\theta_{old} - \theta)^T H(\theta_{old} - \theta) \le \delta$$

$$where \quad H = \nabla_{\theta}^2 \frac{1}{M} \sum_{i=1}^{M} D_{KL}(\theta_{old}, \theta, S_i) \Big|_{\theta = \theta_{old}}$$

The analytical solution to this approximate optimization problem is as follows.

$$\theta = \theta_{old} + \alpha \sqrt{\frac{2\delta}{x^T H^{-1} x}} x$$

Here, $x = H^{-1} g$ and $\alpha$ is a coefficient for ensuring that the policy improves and satisfies the constraint.

## Training Algorithm

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

1. Initialize the actor $\pi(A|S;\theta)$ with random parameter values $\theta$.
2. Initialize the critic $V(S;\phi)$ with random parameter values $\phi$.
3. Generate $N$ experiences by following the current policy. The experience sequence is

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

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

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

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

   For each experience sequence that does not contain a terminal state, $N$ is equal to the `ExperienceHorizon` option value. Otherwise, $N$ is less than `ExperienceHorizon` and $S_N$ is the terminal state.
4. For each episode step $t = ts+1, ts+2, \ldots, ts+N$, compute the return and advantage function using the method specified by the `AdvantageEstimateMethod` option.

   - **Finite Horizon** (`AdvantageEstimateMethod = "finite-horizon"`) — Compute the return $G_t$, which is the sum of the reward for that step and the discounted future reward [2].

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

Here, $b$ is `0` if $S_{ts+N}$ is a terminal state and `1` otherwise. That is, if $S_{ts+N}$ is not a terminal state, the discounted future reward includes the discounted state value function, computed using the critic network $V$.

Compute the advantage function $D_t$.

$$D_t = G_t - V(S_t; \phi)$$

- **Generalized Advantage Estimator** (`AdvantageEstimateMethod = "gae"`) — Compute the advantage function $D_t$, which is the discounted sum of temporal difference errors [3].

$$D_t = \sum_{k=t}^{ts+N-1} (\gamma\lambda)^{k-t} \delta_k$$

$$\delta_k = R_t + b\gamma V(S_t; \phi)$$

Here, $b$ is `0` if $S_{ts+N}$ is a terminal state and `1` otherwise. $\lambda$ is a smoothing factor specified using the `GAEFactor` option.

Compute the return $G_t$.

$$G_t = D_t + V(S_t; \phi)$$

To specify the discount factor $\gamma$ for either method, use the `DiscountFactor` option.

5. Learn from mini-batches of experiences over $K$ epochs. To specify $K$, use the `NumEpoch` option. For each learning epoch:

   a. Sample a random mini-batch data set of size $M$ from the current set of experiences. To specify $M$, use the `MiniBatchSize` option. Each element of the mini-batch data set contains a current experience and the corresponding return and advantage function values.

   b. Update the critic parameters by minimizing the loss $L_{critic}$ across all sampled mini-batch data.

$$L_{critic}(\phi) = \frac{1}{2M} \sum_{i=1}^{M} (G_i - V(S_i; \phi))^2$$

   c. Normalize the advantage values $D_i$ based on recent unnormalized advantage values.

   - If the `NormalizedAdvantageMethod` option is `'none'`, do not normalize the advantage values.

$$\widehat{D}_i \leftarrow D_i$$

   - If the `NormalizedAdvantageMethod` option is `'current'`, normalize the advantage values based on the unnormalized advantages in the current mini-batch.

$$\widehat{D}_i \leftarrow \frac{D_i - mean(D_1, D_2, \ldots, D_M)}{std(D_1, D_2, \ldots, D_M)}$$

   - If the `NormalizedAdvantageMethod` option is `'moving'`, normalize the advantage values based on the unnormalized advantages for the $N$ most recent advantages, including the current advantage value. To specify the window size $N$, use the `AdvantageNormalizingWindow` option.

$$\widehat{D}_i \leftarrow \frac{D_i - mean(D_1, D_2, \ldots, D_N)}{std(D_1, D_2, \ldots, D_N)}$$

**d** Update the actor parameters by solving the constrained optimization problem.

**i** Compute the policy gradient.

$$g = \nabla_\theta L_{actor}(\theta) = \nabla_\theta - \frac{1}{M}\sum_{i=1}^{M}\left(\frac{\pi(A_i|S_i;\theta)}{\pi(A_i|S_i;\theta_{old})}\widehat{D}_i + w\mathscr{H}_i(\theta, S_i)\right)$$

**ii** Apply the conjugate gradient (CG) method to find an approximate solution to the following equation, where $H$ is the Hessian of the KL-divergence between the old and new policies.

$$x \approx H^{-1}g$$

To configure the termination conditions for the CG algorithm, use the `NumIterationsConjugateGradient` and `ConjugateGradientResidualTolerance` options. To stabilize the numerical computation for the CG algorithm, use the `ConjugateGradientDamping` option.

**iii** Using a line search algorithm, find the largest $\alpha$ that satisfies the following constraints.

$$\theta = \theta_{old} + \alpha\sqrt{\frac{2\delta}{x^T H^{-1} x}}x$$

$$L_{actor}(\theta) - L_{actor}(\theta_{old}) < 0$$

$$\frac{1}{M}\sum_{i=1}^{M} D_{KL}(\theta_{old}, \theta, S_i) \le \delta$$

$$\alpha \in \left\{1, \frac{1}{2}, \frac{1}{2^2}, \ldots, \frac{1}{2^{n-1}}\right\}$$

Here, $\delta$ is the KL-divergence limit, which you set using the `KLDivergenceLimit` option. $n$ is the number of line search iterations, which you set using the `NumIterationsLineSearch` option.

**iv** If a valid value of $\alpha$ exists, update the parameters of the actor network to $\theta$. If a valid value of $\alpha$ does not exist, do not update the actor parameters.

**6** Repeat steps 3 through 5 until the training episode reaches a terminal state.

## Entropy Loss

To promote agent exploration, you can add an entropy loss term $w\mathscr{H}_i(\theta,S_i)$ to the actor loss function, where $w$ is the entropy loss weight and $\mathscr{H}_i(\theta,S_i)$ is the entropy.

The entropy value is higher when the agent is more uncertain about which action to take next. Therefore, maximizing the entropy loss term (minimizing the negative entropy loss) increases the agent uncertainty, thus encouraging exploration. To promote additional exploration, which can help the agent move out of local optima, you can specify a larger entropy loss weight.

For a discrete action space, the agent uses the following entropy value. In this case, the actor outputs the probability of taking each possible discrete action.

$$\mathscr{H}_i(\theta, S_i) = -\sum_{k=1}^{P} \pi(A_k|S_i;\theta)\ln\pi(A_k|S_i;\theta)$$

Here:

- $P$ is the number of possible discrete actions.
- $\pi(A_k|S_i;\theta)$ is the probability of taking action $A_k$ when in state $S_i$ following the current policy.

For a continuous action space, the agent uses the following entropy value. In this case, the actor outputs the mean and standard deviation of the Gaussian distribution for each continuous action.

$$\mathscr{H}_i(\theta, S_i) = \frac{1}{2} \sum_{k=1}^{C} \ln\left(2\pi \cdot e \cdot \sigma_{k,i}^2\right)$$

Here:

- $C$ is the number of continuous actions output by the actor.
- $\sigma_{k,i}$ is the standard deviation for action $k$ when in state $S_i$ following the current policy.

## References

[1] Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. "Trust Region Policy Optimization." *Proceedings of the 32nd International Conference on Machine Learning*, pp. 1889-1897. 2015.

[2] Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. “Asynchronous Methods for Deep Reinforcement Learning.” *ArXiv:1602.01783 [Cs]*, February 4, 2016. https://arxiv.org/abs/1602.01783.

[3] Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. “High-Dimensional Continuous Control Using Generalized Advantage Estimation.” *ArXiv:1506.02438 [Cs]*, October 20, 2018. https://arxiv.org/abs/1506.02438.

## See Also

**Objects**

`rlTRPOAgent` | `rlTRPOAgentOptions`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

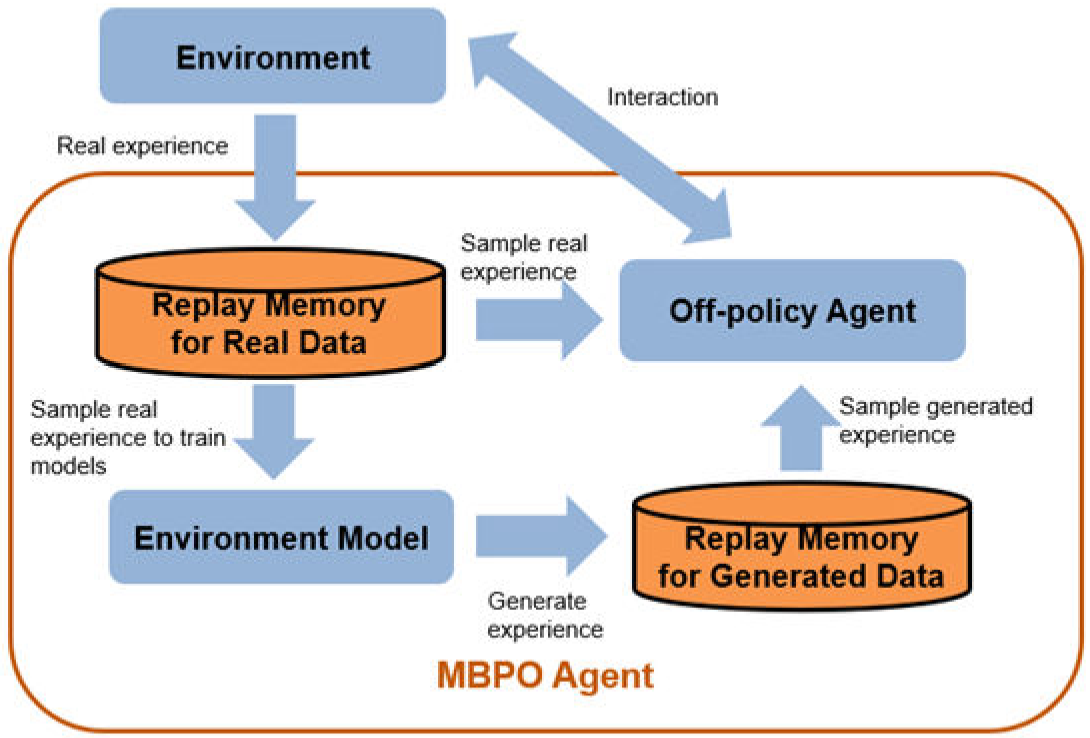
## More About

- “Reinforcement Learning Agents” on page 3-2
- “Create Policies and Value Functions” on page 4-2

# Model-Based Policy Optimization (MBPO) Agents

Model-based policy optimization (MBPO) is a model-based, online, off-policy reinforcement learning algorithm. For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents” on page 3-2.

The following figure shows the components and behavior of an MBPO agent. The agent samples real experience data through environmental interaction and trains a model of the environment using this experience. Then, the agent updates the policy parameters of its base agent using the real experience data and experience generated from the environment model.




**Note** MBPO agents do not support recurrent networks.

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

| Observation Space | Action Space |
| --- | --- |
| Continuous | Discrete or continuous |

You can use the following off-policy agents as the base agent in an MBPO agent.

| Action Space | Base Off-Policy Agent |
| --- | --- |
| Discrete | • DQN agent — “Deep Q-Network (DQN) Agents” on page 3-23 |

| Action Space | Base Off-Policy Agent |
| --- | --- |
| Continuous | • DDPG agent — "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40<br>• TD3 agent — "Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents" on page 3-44<br>• SAC agent — "Soft Actor-Critic (SAC) Agents" on page 3-35 |

MBPO agents use an environment model that you define using an `rlNeuralNetworkEnvironment` object, which contains the following components. In general, these components use a deep neural network to learn the environment behavior during training.

- One or more transition functions that predict the next observation based on the current observation and action. You can define deterministic transition functions using `rlContinuousDeterministicTransitionFunction` objects or stochastic transition functions using `rlContinuousGaussianTransitionFunction` objects.
- A reward function that predicts the reward from the environment based on a combination of the current observation, current action, and next observation. You can define a deterministic reward function using an `rlContinuousDeterministicRewardFunction` object or a stochastic reward function using an `rlContinuousGaussianRewardFunction` object. You can also define a known reward function using a custom function.
- An is-done function that predicts the termination signal based on a combination of the current observation, current action, and next observation. You can also define a known termination signal using a custom function.

During training, an MBPO agent:

- Updates the environment model at the beginning of each episode by training the transition functions, reward function, and is-done function
- Generates samples using the trained environment model and stores the samples in a circular experience buffer
- Stores real samples from the interaction between the agent and the environment using a separate circular experience buffer within the base agent
- Updates the actor and critic of the base agent using a mini-batch of experiences randomly sampled from both the generated experience buffer and the real experience buffer

## Training Algorithm

MBPO agents use the following training algorithm, in which they periodically update the environment model and the base off-policy agent. To configure the training algorithm, specify options using an `rlMBPOAgentOptions` object.

1. Initialize the actor and critics of the base agent.
2. Initialize the transition functions, reward function, and is-done function in the environment model.
3. At the beginning of each training episode:

   a. For each model-training epoch, perform the following steps. To specify the number of epochs, use the `NumEpochForTrainingModel` option.

**i** Train the transition functions. If the corresponding `LearnRate` optimizer option is `0`, skip this step.

- Use a half-mean loss for an `rlContinuousDeterministicTransitionFunction` object and a maximum likelihood loss for an `rlContinuousStochasticTransitionFunction` object.
- To make each observation channel equally important, first compute the loss for each observation channel. Then, divide each loss by the number of elements in its corresponding observation specification.

$$Loss = \sum_{i=1}^{N_O} \frac{1}{M_{oi}} Loss_{oi}$$

For example, if the observation specification for the environment is defined by `[rlNumericSpec([10.1]) rlNumericSpec([4,1])]`, then $N_o$ is 2, $M_{o1}$ is 10, and $M_{o2}$ is 4.

**ii** Train the reward function. If the corresponding `LearnRate` optimizer option is `0` or a ground-truth custom reward function is defined, skip this step.

- Use a half-mean loss for an `rlContinuousDeterministicRewardFunction` object and a maximum likelihood loss for an `rlContinuousStochasticRewardFunction` object.

**iii** Train the is-done function. If the corresponding `LearnRate` optimizer option is `0` or a ground-truth custom is-done function is defined, skip this step.

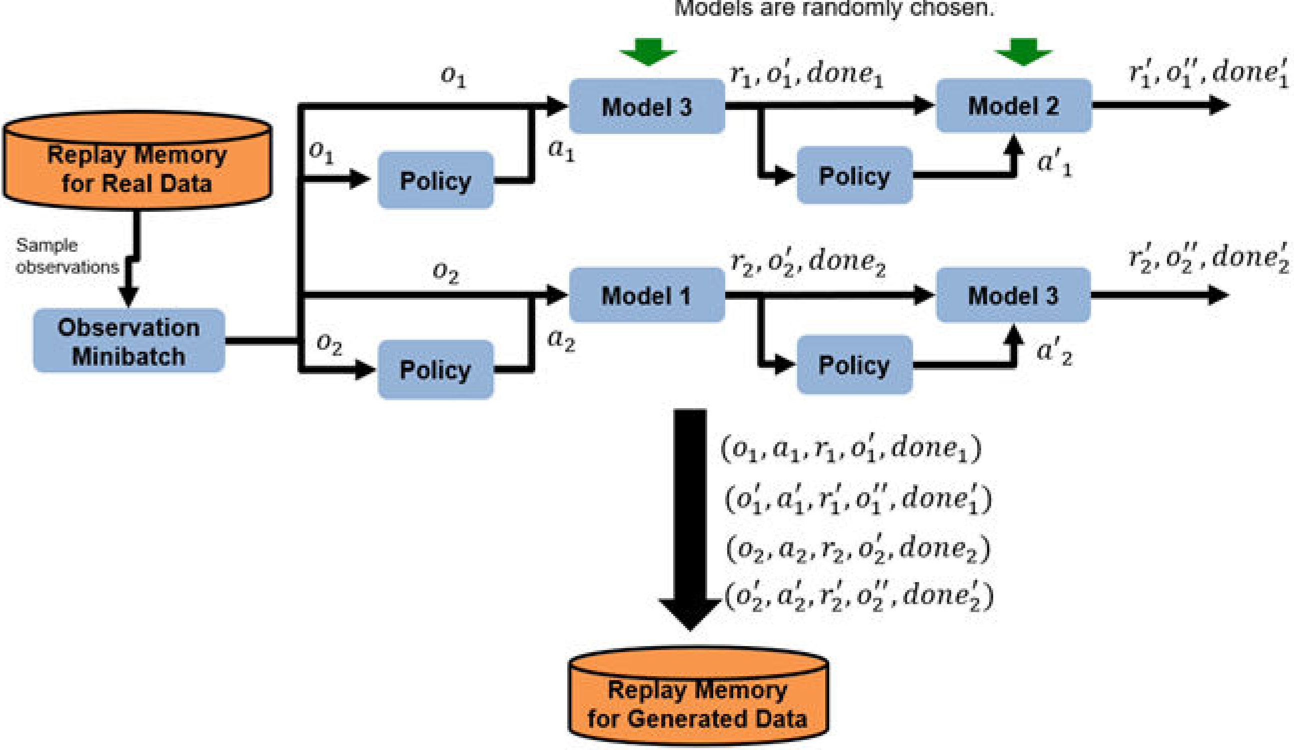
- Use a weighted cross-entropy loss function. In general, the terminal conditions (`isdone = 1`) occur much less frequently than nonterminal conditions (`isdone = 0`). To deal with the heavily imbalanced data, use the following weights and loss function.

$$w_0 = \frac{1}{\sum_{i=1}^{M} (1 - T_i)}, \quad w_1 = \frac{1}{\sum_{i=1}^{M} T_i}$$

$$Loss = \frac{-1}{M} \sum_{i=1}^{M} (w_0 T_i \ln Y_i + w_1 (1 - T_i) \ln(1 - Y_i))$$

Here, $M$ is the mini-batch size, $T_i$ is a target, and $Y_i$ is the output from the reward network for the $i$th sample in the batch. $T_i = 1$ when `isdone` is 1 and $T_i = 0$ when `isdone` is 0.

**b** Generate samples using the trained environment model. The following figure shows an example of two roll-out trajectories with a horizon of two.




**i** Increase the horizon based on the horizon update settings defined in the `ModelRolloutOptions` object.

**ii** Randomly sample a batch of $N_R$ observations from the real experience buffer. To specify $N_R$, use the `ModelRolloutOptions.NumRollout` option.

**iii** For each horizon step:

- Randomly divide the observations into $N_M$ groups, where $N_M$ is the number of transition models, and assign each group to a transition model.
- For each observation $o_i$, generate an action $a_i$ using the exploration policy defined by the `ModelRolloutOptions.NoiseOptions` object. If `ModelRolloutOptions.NoiseOptions` is empty, use the exploration policy of the base agent.
- For each observation-action pair, predict the next observation $o'_2$ using the corresponding transition model.
- Using the environment model reward function, predict the reward value $r_i$ based on the observation, action, and next observation.
- Using the environment model is-done function, predict the termination signal $done_i$ based on the observation, action, and next observation.
- Add the experience $(o_i,a_i,r_i,o'_i,done_i)$ to the generated experience buffer.
- For the next horizon step, substitute each observation with the predicted next observation.

**4** For each step in each training episode:

**a** Sample a mini-batch of $M$ total experiences from the real experience buffer and the generated experience buffer. To specify $M$, use the `MiniBatchSize` option.

- Sample $N_{real} = \lceil M \cdot R \rceil$ samples from the real experience buffer. To specify $R$, use the `RealRatio` option.
- $N_{model} = M - N_{real}$ samples from the generated experience buffer.

**b** Train the base agent using the sampled mini-batch of data by following the update rule of the base agent. For more information, see the corresponding SAC on page 3-35, TD3 on page 3-44, DDPG on page 3-40, or DQN on page 3-23 training algorithm.

## Tips

- MBPO agents can be more sample-efficient than model-free agents because the model can generate large sets of diverse experiences. However, MBPO agents require much more computational time than model-free agents, because they must train the environment model and generate samples in addition to training the base agent.
- To overcome modeling uncertainty, best practice is to use multiple environment transition models.
- If they are available, it is best to use known ground-truth reward and is-done functions.
- It is better to generate a large number of trajectories (thousands or tens of thousands). Doing so generates many samples, which reduces the likelihood of selecting the same sample multiple times in a training episode.
- Since modeling errors can accumulate, it is better to use a shorter horizon when generating samples. A shorter horizon is usually enough to generate diverse experiences.
- In general, an agent created using `rlMBPOAgent` is not suitable for environments with image observations.
- When using a SAC base agent, taking more gradient steps (defined by the `NumGradientStepsPerUpdate` SAC agent option) makes the MBPO agent more sample-efficient. However, doing so increases the computational time.
- The MBPO implementation in `rlMBPOAgent` is based on the algorithm in the original MBPO paper [1] but with the differences shown in the following table.

| Original Paper | `rlMBPOAgent` |
|---|---|
| Generates samples at each environment step | Generates samples at the beginning of each training episode |
| Trains actor and critic using only generated samples | Trains actor and critic using both real data and generated data |
| Uses stochastic environment models | Uses either stochastic or deterministic environment models |
| Uses SAC agents | Can use SAC, DQN, DDPG, and TD3 agents |

## References

[1] Janner, Michael, Justin Fu, Marvin Zhang, and Sergey Levine. "When to Trust Your Model: Model-Based Policy Optimization." In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 12519–30. 1122. Red Hook, NY, USA: Curran Associates Inc., 2019.

## See Also

**Objects**
`rlMBPOAgent` | `rlMBPOAgentOptions` | `rlNeuralNetworkEnvironment`

## Related Examples

- "Train MBPO Agent to Balance Cart-Pole System" on page 5-472

# Create Custom Reinforcement Learning Agents

To implement your own custom reinforcement learning algorithms, you can create a custom agent by creating a subclass of a custom agent class. You can then train and simulate this agent in MATLAB and Simulink environments. For more information about creating classes in MATLAB, see "User-Defined Classes".

## Create Template Class

To define your custom agent, first create a class that is a subclass of the `rl.agent.CustomAgent` class. As an example, this topic describes the custom LQR agent trained in "Train Custom LQR Agent" on page 5-466. As a starting point for your own agent, you can open and modify this custom agent class. To download the example files in a local folder and open the main example live script, at the MATLAB command line, type the following code.

```
openExample('rl/TrainCustomLQRAgentExample')
```

Close the `TrainCustomLQRAgentExample.mlx` file and open the `LQRCustomAgent.m` class file.

```
edit LQRCustomAgent.m
```

After saving the class to your own working folder, you can remove the example files and the local folder in which they were downloaded.

The class defined in `LQRCustomAgent.m` has the following class definition, which indicates the agent class name and the associated abstract agent.

```
classdef LQRCustomAgent < rl.agent.CustomAgent
```

To define your agent, you must specify the following:

- Agent properties
- Constructor function
- A critic that estimates the discounted long-term reward (if required for learning)
- An actor that selects an action based on the current observation (if required for learning)
- Required agent methods
- Optional agent methods

## Agent Properties

In the `properties` section of the class file, specify any parameters necessary for creating and training the agent. These parameters can include:

- Discount factor for discounting future rewards
- Configuration parameters for exploration models, such as noise models or epsilon-greedy exploration
- Experience buffers for using replay memory
- Mini-batch sizes for sampling from the experience buffer
- Number of steps to look ahead during training

For more information on potential agent properties, see the option objects for the built-in Reinforcement Learning Toolbox agents.

The `rl.Agent.CustomAgent` class already includes properties for the agent sample time (`SampleTime`) and the action and observation specifications (`ActionInfo` and `ObservationInfo`, respectively).

The custom LQR agent defines the following agent properties.

```
properties
    % Q
    Q

    % R
    R

    % Feedback gain
    K

    % Discount factor
    Gamma = 0.95

    % Critic
    Critic

    % Buffer for K
    KBuffer
    % Number of updates for K
    KUpdate = 1

    % Number for estimator update
    EstimateNum = 10
end

properties (Access = private)
    Counter = 1
    YBuffer
    HBuffer
end
```

## Constructor Function

To create your custom agent, you must define a constructor function that:

- Defines the action and observation specifications. For more information about creating these specifications, see `rlNumericSpec` and `rlFiniteSetSpec`.
- Creates actor and critic as required by your training algorithm. For more information, see “Create Policies and Value Functions” on page 4-2.
- Configures agent properties.
- Calls the constructor of the base abstract class.

For example, the `LQRCustomAgent` constructor defines continuous action and observation spaces and creates a critic. The `createCritic` function is an optional helper function that defines the critic.

```
function obj = LQRCustomAgent(Q,R,InitialK)
    % Check the number of input arguments
```

```
    narginchk(3,3);

    % Call the abstract class constructor
    obj = obj@rl.agent.CustomAgent();

    % Set the Q and R matrices
    obj.Q = Q;
    obj.R = R;

    % Define the observation and action spaces
    obj.ObservationInfo = rlNumericSpec([size(Q,1),1]);
    obj.ActionInfo = rlNumericSpec([size(R,1),1]);

    % Create the critic
    obj.Critic = createCritic(obj);

    % Initialize the gain matrix
    obj.K = InitialK;

    % Initialize the experience buffers
    obj.YBuffer = zeros(obj.EstimateNum,1);
    num = size(Q,1) + size(R,1);
    obj.HBuffer = zeros(obj.EstimateNum,0.5*num*(num+1));
    obj.KBuffer = cell(1,1000);
    obj.KBuffer{1} = obj.K;
end
```

## Actor and Critic

If your learning algorithm uses a critic to estimate the long-term reward, an actor for selecting an action, or both, you must add these as agent properties. You must then create these objects when you create your agent; that is, in the constructor function. For more information on creating actors and critics, see "Create Policies and Value Functions" on page 4-2.

For example, the custom LQR agent uses a critic, stored in its `Critic` property, and no actor. The critic creation is implemented in the `createCritic` helper function, which is called from the `LQRCustomAgent` constructor.

```
function critic = createCritic(obj)
    nQ = size(obj.Q,1);
    nR = size(obj.R,1);
    n = nQ+nR;
    w0 = 0.1*ones(0.5*(n+1)*n,1);
    critic = rlQValueFunction({@(x,u) computeQuadraticBasis(x,u,n),w0},...
        getObservationInfo(obj),getActionInfo(obj));
    critic.Options.GradientThreshold = 1;
end
```

In this case, the critic is an `rlQValueFunction` object. To create this object, you must specify the handle to a custom basis function, in this case the `computeQuadraticBasis` function. For more information, see "Train Custom LQR Agent" on page 5-466.

## Required Functions

To create a custom reinforcement learning agent you must define the following implementation functions. To call these functions in your own code, use the wrapper methods from the abstract base class. For example, to call `getActionImpl`, use `getAction`. The wrapper methods have the same input and output arguments as the implementation methods.

| Function | Description |
| --- | --- |
| getActionImpl | Selects an action by evaluating the agent policy for a given observation |
| getActionWithExplorationImpl | Selects an action using the exploration model of the agent |
| learnImpl | Learns from the current experiences and returns an action with exploration |

Within your implementation functions, to evaluate your actor and critic, you can use the getValue, getAction, and getMaxQValue functions.

- To evaluate an rlValueFunction critic, you need only the observation input, and you can obtain the value of the current observation V using the following syntax.

  ```
  V = getValue(Critic,Observation);
  ```
- To evaluate an rlQValueFunction critic you need both observation and action inputs, and you can obtain the value of the current state-action Q using the following syntax.

  ```
  Q = getValue(Critic,[Observation,Action]);
  ```
- To evaluate an rlVectorQValueFunction critic you need only the observation input, and you can obtain the value of the current observation Q for all possible discrete actions using the following syntax.

  ```
  Q = getValue(Critic,Observation);
  ```
- For a discrete action space rlQValueFunction critic, obtain the maximum Q state-action value function Q for all possible discrete actions using the following syntax.

  ```
  [MaxQ,MaxActionIndex] = getMaxQValue(Critic,Observation);
  ```
- To evaluate an actor, obtain the action A using the following syntax.

  ```
  A = getAction(Actor,Observation);
  ```

For each of these cases, if your actor or critic network uses a recurrent neural network, the functions can also return the current values of the network state after obtaining the corresponding network output.

#### getActionImpl Function

The getActionImpl function evaluates the policy of your agent and selects an action. This function must have the following signature, where obj is the agent object, Observation is the current observation, and action is the selected action.

```
function action = getActionImpl(obj,Observation)
```

For the custom LQR agent, you select an action by applying the $u$=-$Kx$ control law.

```
function action = getActionImpl(obj,Observation)
    % Given the current state of the system, return an action
    action = -obj.K*Observation{:};
end
```

#### getActionWithExplorationImpl Function

The getActionWithExplorationImpl function selects an action using the exploration model of your agent. Using this function you can implement algorithms such as epsilon-greedy exploration.

This function must have the following signature, where `obj` is the agent object, `Observation` is the current observation, and `action` is the selected action.

```
function action = getActionWithExplorationImpl(obj,Observation)
```

For the custom LQR agent, the `getActionWithExplorationImpl` function adds random white noise to an action selected using the current agent policy.

```
function action = getActionWithExplorationImpl(obj,Observation)
    % Given the current observation, select an action
    action = getAction(obj,Observation);

    % Add random noise to the action
    num = size(obj.R,1);
    action = action + 0.1*randn(num,1);
end
```

#### learnImpl Function

The `learnImpl` function defines how the agent learns from the current experience. This function implements the custom learning algorithm of your agent by updating the policy parameters and selecting an action with exploration. This function must have the following signature, where `obj` is the agent object, `exp` is the current agent experience, and `action` is the selected action.

```
function action = learnImpl(obj,exp)
```

The agent experience is the cell array `exp = {state,action,reward,nextstate,isdone}`.

- `state` is the current observation.
- `action` is the current action.
- `reward` is the current reward.
- `nextState` is the next observation.
- `isDone` is a logical flag indicating that the training episode is complete.

For the custom LQR agent, the critic parameters are updated every `N` steps.

```
function action = learnImpl(obj,exp)
    % Parse the experience input
    x = exp{1}{1};
    u = exp{2}{1};
    dx = exp{4}{1};
    y = (x'*obj.Q*x + u'*obj.R*u);
    num = size(obj.Q,1) + size(obj.R,1);

    % Wait N steps before updating the critic parameters
    N = obj.EstimateNum;
    h1 = computeQuadraticBasis(x,u,num);
    h2 = computeQuadraticBasis(dx,-obj.K*dx,num);
    H = h1 - obj.Gamma* h2;
    if obj.Counter<=N
        obj.YBuffer(obj.Counter) = y;
        obj.HBuffer(obj.Counter,:) = H;
        obj.Counter = obj.Counter + 1;
    else
        % Update the critic parameters based on the batch of
        % experiences
```

```
        H_buf = obj.HBuffer;
        y_buf = obj.YBuffer;
        theta = (H_buf'*H_buf)\H_buf'*y_buf;
        obj.Critic = setLearnableParameters(obj.Critic,{theta});

        % Derive a new gain matrix based on the new critic parameters
        obj.K = getNewK(obj);

        % Reset the experience buffers
        obj.Counter = 1;
        obj.YBuffer = zeros(N,1);
        obj.HBuffer = zeros(N,0.5*num*(num+1));
        obj.KUpdate = obj.KUpdate + 1;
        obj.KBuffer{obj.KUpdate} = obj.K;
    end

    % Find and return an action with exploration
    action = getActionWithExploration(obj,exp{4});
end
```

## Optional Functions

Optionally, you can define how your agent is reset at the start of training by specifying a `resetImpl` function with the following function signature, where `obj` is the agent object. Using this function, you can set the agent into a known or random condition before training.

```
function resetImpl(ob)
```

Also, you can define any other helper functions in your custom agent class as required. For example, the custom LQR agent defines a `createCritic` function for creating the critic and a `getNewK` function that derives the feedback gain matrix from the trained critic parameters.

## Create Custom Agent

After you define your custom agent class, create an instance of it in the MATLAB workspace. For example, to create the custom LQR agent, define the `Q`, `R`, and `InitialK` values and call the constructor function.

```
Q = [10,3,1;3,5,4;1,4,9];
R = 0.5*eye(3);
K0 = place(A,B,[0.4,0.8,0.5]);
agent = LQRCustomAgent(Q,R,K0);
```

After validating the environment object, you can use it to train a reinforcement learning agent. For an example that trains the custom LQR agent, see “Train Custom LQR Agent” on page 5-466.

## See Also

**Functions**
`train`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About


- "Reinforcement Learning Agents" on page 3-2
- "Create Policies and Value Functions" on page 4-2

# Define Policies and Value Functions


- "Create Policies and Value Functions" on page 4-2
- "Import Neural Network Models" on page 4-15

# Create Policies and Value Functions

A reinforcement learning policy is a mapping from an environment observation to a probability distribution of the actions to be taken (starting from the state corresponding to the observation). A value function is a mapping from an environment observation (or observation-action pair) to the value (the expected cumulative long-term reward) of a policy.

Reinforcement learning agents use *parametrized* policies and value functions, which are implemented by function approximators called actors and critics, respectively. During training, an agent updates the parameters of its actor and critic to maximize the expected cumulative long-term reward.

Before creating a non-default agent, you must create the actor and critic using approximation models such as deep neural networks, linear basis functions, or lookup tables. The type of function approximator and model you can use depends on the type of agent that you want to create.

You can also create policy objects from agents, actors, or critics. You can train these objects using custom loops and deploy them in applications.

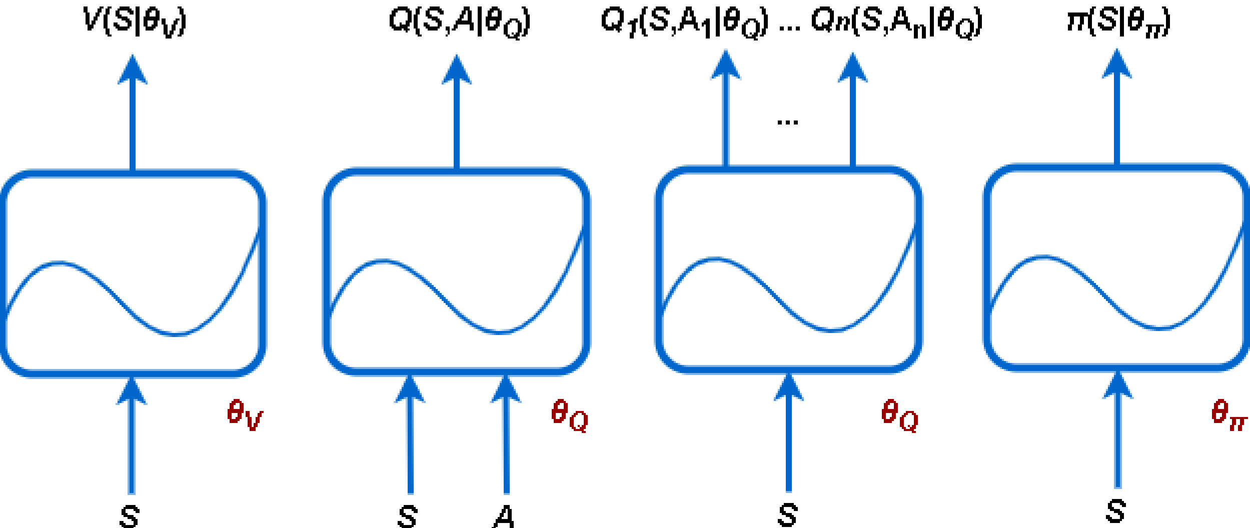
For more information on agents, see “Reinforcement Learning Agents” on page 3-2.

## Actors and Critics

Reinforcement Learning Toolbox software supports the following types of actors and critics:

- $V(S|\theta_V)$ — Critics that estimate the expected cumulative long-term reward of a policy based on a given observation $S$. You can create these critics using `rlValueFunction`.
- $Q(S,A|\theta_Q)$ — Critics that estimate the expected cumulative long-term reward of a policy for a given discrete action $A$ and a given observation $S$. You can create these critics using `rlQValueFunction`.
- $Q_i(S,A_i|\theta_Q)$ — Multi-output critics that estimate the expected cumulative long-term reward of a policy for all possible discrete actions $A_i$ given the observation $S$. You can create these critics using `rlVectorQValueFunction`.
- $\pi(S|\theta_\pi)$ — Actors with a continuous action space that select an action deterministically based on a given observation $S$, thereby implementing a deterministic policy. You can create these actors using `rlContinuousDeterministicActor`.
- $\pi(S|\theta_\pi)$ — Actors that select an action stochastically (the action is sampled from a probability distribution) based on a given observation $S$, thereby implementing a stochastic policy. You can create these actors using either `rlDiscreteCategoricalActor` (for discrete action spaces) or `rlContinuousGaussianActor` (for continuous action spaces).

Each approximator uses a set of parameters ($\theta_V$, $\theta_Q$, $\theta_\pi$), which are computed during the learning process.




For systems with a limited number of discrete observations and discrete actions, you can store value functions in a lookup table. For systems that have many discrete observations and actions and for observation and action spaces that are continuous, storing the observations and actions is impractical. For such systems, you can represent your actors and critics using deep neural networks or custom (linear in the parameters) basis functions.

The following table summarizes the way in which you can use the six approximator objects available with Reinforcement Learning Toolbox software, depending on the action and observation spaces of your environment, and on the approximation model and agent that you want to use.

**How Function Approximators (Actors or Critics) are Used in Agents**

| Approximator (Actor or Critic) | Supported Model | Observation Space | Action Space | Supported Agents |
|---|---|---|---|---|
| Value function critic $V(S)$, which you create using `rlValueFunction` | Table | Discrete | Not applicable | PG, AC, PPO |
| | Deep neural network or custom basis function | Discrete or continuous | Not applicable | PG, AC, PPO |
| | Deep neural network | Discrete or continuous | Not applicable | TRPO |
| Q-value function critic, $Q(S,A)$, which you create using `rlQValueFunction` | Table | Discrete | Discrete | Q, DQN, SARSA |
| | Deep neural network or custom basis function | Discrete or continuous | Discrete | Q, DQN, SARSA |
| | | | Continuous | DDPG, TD3, SAC |
| Multi-output Q-value function critic with a discrete action space $Q(S,A)$, which you create using `rlVectorQValueFunction` | Deep neural network or custom basis function | Discrete or continuous | Discrete | Q, DQN, SARSA |
| Deterministic policy actor with a continuous action space $\pi(S)$, which you create using `rlContinuousDeterministicActor` | Deep neural network or custom basis function | Discrete or continuous | Continuous | DDPG, TD3 |
| Stochastic policy actor with a discrete action space $\pi(S)$, which you create using `rlDiscreteCategoricalActor` | Deep neural network or custom basis function | Discrete or continuous | Discrete | PG, AC, PPO |
| | Deep neural network | Discrete or continuous | Discrete | TRPO |
| Stochastic policy actor with a continuous action space $\pi(S)$, which you create using `rlContinuousGaussianActor` | Deep neural network | Discrete or continuous | Continuous | PG, AC, PPO, SAC, TRPO |

You can configure the actor and critic optimization options using the `rlOptimizerOptions` object within an agent option object.

Specifically, you can create an agent options object and set its `CriticOptimizerOptions` and `ActorOptimizerOptions` properties to appropriate `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;`.

For more information on agents, see "Reinforcement Learning Agents" on page 3-2.

## Policy Objects

You can extract a policy object from an agent using `getGreedyPolicy` or `getExplorationPolicy`, or you can create a policy object from an actor or critic.

Once you have the policy object, you can then use `getAction` to generate deterministic or stochastic actions from it, given an input observation. Differently from function approximator objects like actors and critics, policy objects do not have functions that you can use to easily calculate gradients with respect to parameters. Therefore, policy objects are more tailored toward application deployment, rather than training. The following table describes the available policy objects.

**Policy Objects**

| Policy Object and getAction Behavior | Distribution and Exploration | Action Space | Approximator Objects Used for Creation | Agents Needed for Extraction |
|---|---|---|---|---|
| `rlMaxQPolicy`<br><br>Generates actions that maximize a discrete action-space Q-value function | Deterministic (no exploration) and greedy. | Discrete | `rlQValueFunction` or `rlVectorQValueFunction` | Q, DQN, SARSA |
| `rlEpsilonGreedyPolicy`<br><br>Generates either actions that maximize a discrete action-space Q-value function with probability `1-Epsilon` or random actions otherwise | Default: Stochastic (random actions help exploration) | Discrete | `rlQValueFunction` or `rlVectorQValueFunction` | Q, DQN, SARSA |
| `rlDeterministicActorPolicy`<br><br>Generates continuous deterministic actions | Deterministic (no exploration) and greedy. | Continuous | `rlContinuousDeterministicActor` | DDPG, TD3 |
| `rlAdditiveNoisePolicy`<br><br>Generates continuous deterministic actions with added noise according to an internal noise model | Default: Stochastic (noise helps exploration) | Continuous | `rlContinuousDeterministicActor` | DDPG, TD3 |
| `rlStochasticActorPolicy`<br><br>Generates stochastic actions according to a probability distribution | Default: Stochastic (random actions help exploration) | Discrete | `rlDiscreteCategoricalActor` | PG, AC, PPO, TRPO |
| | | Continuous | `rlContinuousGaussianActor` | PG, AC, PPO, TRPO, SAC |

Each one of the stochastic policy objects has an option to enable deterministic behavior, thereby disabling exploration. Except for `rlEpsilonGreedyPolicy` and `rlAdditiveNoisePolicy`, you can use `generatePolicyBlock` and `generatePolicyFunction` to generate a Simulink block or a function that evaluates the policy, returning an action, for a given observation input. You can then use the generated function or block to generate code for application deployment. For more information, see “Deploy Trained Reinforcement Learning Policies” on page 6-2.

## Table Models

Value function approximators (critics) based on lookup tables models are appropriate for environments with a limited number of *discrete* observations and actions. You can create two types of lookup tables:

- Value tables, which store rewards for corresponding observations
- Q-tables, which store rewards for corresponding observation-action pairs

To create a table based critic, first create a value table or Q-table using the `rlTable` function. Then use the table object as input argument for either `rlValueFunction` or `rlQValueFunction` to create the approximator object.

## Neural Network Models

You can create actor and critic function approximators using deep neural networks models. Doing so uses Deep Learning Toolbox software features.

### Network Input and Output Dimensions

The dimensions of the network input and output layers for your actor and critic must match the dimension of the corresponding environment observation and action channels, respectively. To obtain the action and observation specifications from the environment `env`, use the `getActionInfo` and `getObservationInfo` functions, respectively.

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

Access the `Dimensions` property of each channel. For example, get the size of the first environment and action channel:

```
actSize = actInfo(1).Dimensions;
obsSize = obsInfo(1).Dimensions;
```

In general `actSize` and `obsSize` are row vectors whose elements are the lengths of the corresponding dimensions. For example, if the first observation channel is a 256-by-256 RGB image, `actSize` is the vector `[256 256 3]`. To calculate the total number of dimension of the channel, use `prod`.For example, assuming the environment has only one observation channel:

```
obsDimensions = prod(obsInfo.Dimensions);
```

For `rlVectorQValueFunction` critics and `rlDiscreteCategoricalActor` actors, you need to obtain the number of possible elements of the action set. You can do so by accessing the `Elements` property of the action channel. For example, assuming the environment has only one action channel:

```
actNumElements = numel(actInfo.Elements);
```

Networks for value function critics (such as the ones used in AC, PG, PPO or TRPO agents) must take only observations as inputs and must have a single scalar output. For these networks, the dimensions of the input layers must match the dimensions of the environment observation channels. For more information, see `rlValueFunction`.

Networks for single-output Q-value function critics (such as the ones used in Q, DQN, SARSA, DDPG, TD3, and SAC agents) must take both observations and actions as inputs, and must have a single scalar output. For these networks, the dimensions of the input layers must match the dimensions of the environment channels for both observations and actions. For more information, see `rlQValueFunction`.

Networks for multi-output Q-value function critics (such as those used in Q, DQN, and SARSA agents) take only observations as inputs and must have a single output layer with output size equal to the number of possible discrete actions. For these networks the dimensions of the input layers must

match the dimensions of the environment observations channels. For more information, see rlVectorQValueFunction.

For actor networks, the dimensions of the input layers must match the dimensions of the environment observation channels and the dimension of the output layer must be as follows.

- Networks used in actors with a discrete action space (such as the ones in PG, AC, and PPO agents) must have a single output layer with an output size equal to the number of possible discrete actions. For more information, see rlDiscreteCategoricalActor.
- Networks used in deterministic actors with a continuous action space (such as the ones in DDPG and TD3 agents) must have a single output layer with an output size matching the dimension of the action space defined in the environment action specification. For more information, see rlContinuousDeterministicActor.
- Networks used in stochastic actors with a continuous action space (such as the ones in PG, AC, PPO, and SAC agents) must have a two output layers each with as many elements as the dimension of the action space, as defined in the environment specification. One output layer must produce the mean values (which must be scaled to the output range of the action), and the other must produce the standard deviations of the actions (which must be non-negative). For more information, see rlContinuousGaussianActor.

### Deep Neural Networks

Deep neural networks consist of a series of interconnected layers. You can specify a deep neural network as one of the following:

- Array of Layer objects
- layerGraph object
- DAGNetwork object
- SeriesNetwork object
- dlnetwork object

---

**Note** Among the different network objects, dlnetwork is preferred, since it has built-in validation checks and supports automatic differentiation. If you pass another network object as an input argument, it is internally converted to a dlnetwork object. However, best practice is to convert other network objects to dlnetwork explicitly *before* using it to create a critic or an actor for a reinforcement learning agent. You can do so using dlnet=dlnetwork(net), where net is any neural network object from the Deep Learning Toolbox. The resulting dlnet is the dlnetwork object that you use for your critic or actor. This practice allows a greater level of insight and control for cases in which the conversion is not straightforward and might require additional specifications.

---

Typically, you build your neural network by stacking together a number of layers in an array of Layer objects, possibly adding these arrays to a layerGraph object, and then converting the final result to a dlnetwork object.

For agents that need multiple input or output layers, you create an array of Layer objects for each input path (observations or actions) and for each output path (estimated rewards or actions). You then add these arrays to a layerGraph object and connect them paths together using the connectLayers function.

You can also create your deep neural network using the **Deep Network Designer** app. For an example, see “Create DQN Agent Using Deep Network Designer and Train Using Image Observations” on page 5-152.

The following table lists some common deep learning layers used in reinforcement learning applications. For a full list of available layers, see “List of Deep Learning Layers”.

| Layer | Description |
| --- | --- |
| `featureInputLayer` | Inputs feature data and applies normalization |
| `imageInputLayer` | Inputs vectors and 2-D images and applies normalization. |
| `sigmoidLayer` | Applies a sigmoid function to the input such that the output is bounded in the interval (0,1). |
| `tanhLayer` | Applies a hyperbolic tangent activation layer to the input. |
| `reluLayer` | Sets any input values that are less than zero to zero. |
| `fullyConnectedLayer` | Multiplies the input vector by a weight matrix, and add a bias vector. |
| `softmaxLayer` | Applies a softmax function layer to the input, normalizing it to a probability distribution. |
| `convolution2dLayer` | Applies sliding convolutional filters to the input. |
| `additionLayer` | Adds the outputs of multiple layers together. |
| `concatenationLayer` | Concatenates inputs along a specified dimension. |
| `sequenceInputLayer` | Provides inputs sequence data to a network. |
| `lstmLayer` | Applies a Long Short-Term Memory layer to the input. Supported for DQN and PPO agents. |

The `bilstmLayer` and `batchNormalizationLayer` layers are not supported for reinforcement learning.

The Reinforcement Learning Toolbox software provides the following layers, which contain no tunable parameters (that is, parameters that change during training).

| Layer | Description |
| --- | --- |
| `scalingLayer` | Applies a linear scale and bias to an input array. This layer is useful for scaling and shifting the outputs of nonlinear layers, such as `tanhLayer` and `sigmoidLayer`. |
| `quadraticLayer` | Creates a vector of quadratic monomials constructed from the elements of the input array. This layer is useful when you need an output that is some quadratic function of its inputs, such as for an LQR controller. |
| `softplusLayer` | Implements the softplus activation $Y = \log(1 + e^X)$, which ensures that the output is always positive. This function is a smoothed version of the rectified linear unit (ReLU). |

You can also create your own custom layers. For more information, see “Define Custom Deep Learning Layers”.

When you create a deep neural network, it is good practice to specify names for the first layer of each input path and the final layer of the output path. These names allow you to connect network paths and then later explicitly associate each network input layer with its appropriate environment channel.

The following code creates and connects the following input and output paths:

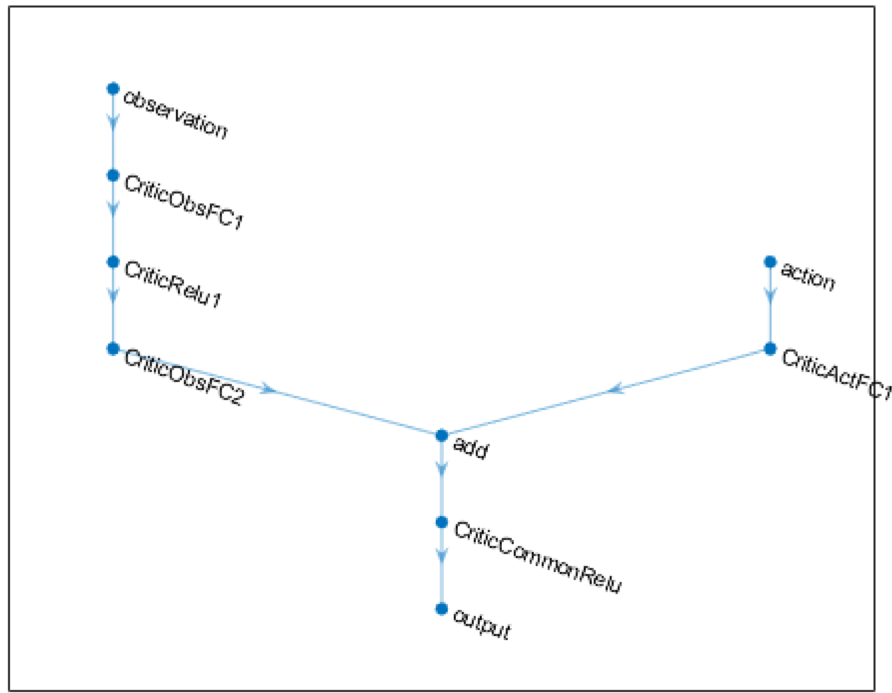
- An observation input path, `observationPath`, with the first layer named `"obsInputLayer"`.
- An action input path, `actionPath`, with the first layer named `"actInputLayer"`.
- An estimated value function output path, `commonPath`, which takes the outputs of `observationPath` and `actionPath` as inputs. The final layer of this path is named `"QValueOutputLayer"`.

```
% Observation path: array of layer objects
observationPath = [
    featureInputLayer(4,Name="obsInputLayer")
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(24,Name="ObsFC2")
    ];

% Action path: array of layer objects
actionPath = [
    featureInputLayer(1,Name="actInputLayer")
    fullyConnectedLayer(24,Name="ActFC1")
    ];

% Common path: array of layer objects
commonPath = [
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(1,Name="QValueOutputLayer")];

% Assemble layerGraph object
criticNetwork = layerGraph(observationPath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);

% Connect layers
criticNetwork = connectLayers(criticNetwork,"ObsFC2","add/in1");
criticNetwork = connectLayers(criticNetwork,"ActFC1","add/in2");

% Convert to a dlnetwork object
criticNetwork = dlnetwork(criticNetwork);

% Display the number of learnable parameters
summary(criticNetwork)
```

For all observation and action input paths, you must specify a `featureInputLayer` as the first layer in the path, with a number of input neurons equal to the number of dimensions of the corresponding environment channel.

You can view the structure of your deep neural network using the `plot` function.

```
plot(layerGraph(criticNetwork))
```




Since the output of a network in an `rlDiscreteCategoricalActor` actors must represent the probability of executing each possible action, the software automatically adds a `softmaxLayer` as a final output layer if you do not specify it explicitly. When computing the action, the actor then randomly samples the distribution to return an action.

Determining the number, type, and size of layers for your deep neural network can be difficult and is application dependent. However, the most critical component in deciding the characteristics of the function approximator is whether it is able to approximate the optimal policy or discounted value function for your application, that is, whether it has layers that can correctly learn the features of your observation, action, and reward signals.

Consider the following tips when constructing your network.

- For continuous action spaces, bound actions with a `tanhLayer` followed by a `ScalingLayer` to scale the action to desired values, if necessary.
- Deep dense networks with `reluLayer` layers can be fairly good at approximating many different functions. Therefore, they are often a good first choice.
- Start with the smallest possible network that you think can approximate the optimal policy or value function.
- When you approximate strong nonlinearities or systems with algebraic constraints, adding more layers is often better than increasing the number of outputs per layer. In general, the ability of the approximator to represent more complex (compositional) functions grows only polynomially in the size of the layers, but grows exponentially with the number of layers. In other words, more layers allow approximating more complex and nonlinear compositional functions, although this generally requires more data and longer training times. Given a total number of neurons and comparable approximation tasks, networks with fewer layers can require exponentially more units to

successfully approximate the same class of functions, and might fail to learn and generalize correctly.

- For on-policy agents (the ones that learn only from experience collected while following the current policy), such as AC and PG agents, parallel training works better if your networks are large (for example, a network with two hidden layers with 32 nodes each, which has a few hundred parameters). On-policy parallel updates assume each worker updates a different part of the network, such as when they explore different areas of the observation space. If the network is small, the worker updates can correlate with each other and make training unstable.

### Create and Configure Actors and Critics from a Neural Network

To create a critic from your deep neural network, use an `rlValueFunction`, `rlQValueFunction` or (whenever possible) an `rlVectorQValueFunction` object. To create a deterministic actor for a continuous action space from your deep neural network, use an `rlContinuousDeterministicActor` object. To create a stochastic actor from your deep neural network use either an `rlDiscreteCategoricalActor` or an `rlContinuousGaussianActor` object. To configure the learning rate and optimization used by the actor or critic, use an optimizer object within an agent option object.

For example, create a Q-value function critic using the neural network `criticNetwork` and the environment action and observation specifications. Pass as additional arguments also the names of the network input layers to be connected with the observation and action channels, respectively.

```
critic = rlQValueFunction(criticNetwork,obsInfo,actInfo,...
        ObservationInputNames={"obsInputLayer"}, ...
        ActionInputNames="actInputLayer");
```

To specify training options for the critic, use `rlOptimizerOptions` to create the critic optimizer object `criticOpts`, specifying a learning rate of `0.02` and a gradient threshold of `1`.

```
criticOpts = rlOptimizerOptions(LearnRate=0.02,...
                                GradientThreshold=1);
```

Then create an agent option object, and set the `CriticOptimizerOptions` property of the agent option object to `criticOpts`. When finally you create the agent, pass the agent option object as a last input argument to the agent constructor function. Alternatively, you can create the agent first, and then access its option object, and modify the options, using dot notation.

When you create your deep neural network and configure your actor or critic, consider using the following approach as a starting point.

1. Start with the smallest possible network and a high learning rate (`0.01`). Train this initial network to see if the agent converges quickly to a poor policy or acts in a random manner. If either of these issues occur, rescale the network by adding more layers or more outputs on each layer. Your goal is to find a network structure that is just big enough, does not learn too fast, and shows signs of learning (an improving trajectory of the reward graph) after an initial training period.
2. Once you settle on a good network architecture, a low initial learning rate can allow you to see if the agent is on the right track, and help you check that your network architecture is satisfactory for the problem. A low learning rate makes tuning parameters easier, especially for difficult problems.

Also, consider the following tips when configuring your deep neural network agent.

- Be patient with DDPG and DQN agents, since they might not learn anything for some time during the early episodes, and they typically show a dip in cumulative reward early in the training process. Eventually, they can show signs of learning after the first few thousand episodes.
- For DDPG and DQN agents, promoting exploration of the agent is critical.
- For agents with both actor and critic networks, set the initial learning rates of both actor and critic to the same value. However, for some problems, setting the critic learning rate to a higher value than that of the actor can improve learning results.

#### Recurrent Neural Networks

When creating actors or critics for use with any agent except Q, SARSA, TRPO and MBPO, you can use recurrent neural networks (RNN). These networks are deep neural networks with a `sequenceInputLayer` input layer and at least one layer that has hidden state information, such as an `lstmLayer`. They can be especially useful when the environment has states that cannot be included in the observation vector.

For agents that have both actor and critic, you must either use an RNN for both of them, or not use an RNN for any of them. You cannot use an RNN only for the critic or only for the actor.

When using PG agents, the learning trajectory length for the RNN is the whole episode. For an AC agent, the `NumStepsToLookAhead` property of its options object is treated as the training trajectory length. For a PPO agent, the trajectory length is the `MiniBatchSize` property of its options object.

For DQN, DDPG, SAC and TD3 agents, you must specify the length of the trajectory training as an integer greater than one in the `SequenceLength` property of their options object.

Note that code generation is not supported for continuous action space PG, AC, PPO and TRPO agents, and SAC agents using a recurrent neural network (RNN), or for any agent having multiple input paths and containing an RNN in any of the paths.

For more information and examples on policies and value functions, see `rlValueFunction`, `rlQValueFunction`, `rlVectorQValueFunction`, `rlContinuousDeterministicActor`, `rlDiscreteCategoricalActor`, and `rlContinuousGaussianActor`.

## Custom Basis Function Models

Custom (linear in the parameters) basis function approximation models have the form `f = W'B`, where `W` is a weight array and `B` is the column vector output of a custom basis function that you must create. The learnable parameters of a linear basis function are the elements of `W`.

For value function critics, (such as the ones used in AC, PG or PPO agents), `f` is a scalar value, so `W` must be a column vector with the same length as `B`, and `B` must be a function of the observation. For more information and examples, see `rlValueFunction`.

For single-output Q-value function critics, (such as the ones used in Q, DQN, SARSA, DDPG, TD3, and SAC agents), `f` is a scalar value, so `W` must be a column vector with the same length as `B`, and `B` must be a function of both the observation and action. For more information and examples, see `rlQValueFunction`.

For multi-output Q-value function critics with discrete action spaces, (such as those used in Q, DQN, and SARSA agents), `f` is a vector with as many elements as the number of possible actions. Therefore `W` must be a matrix with as many columns as the number of possible actions and as many rows as the

length of B. B must be only a function of the observation. For more information and examples, see rlVectorQValueFunction.

- For deterministic actors with a continuous action space (such as the ones in DDPG, and TD3 agents), the dimensions of f must match the dimensions of the agent action specification, which is either a scalar or a column vector. For more information and examples, see rlContinuousDeterministicActor.
- For stochastic actors with a discrete action space (such as the ones in PG, AC, and PPO agents), f must be column vector with length equal to the number of possible discrete actions. The output of the actor is softmax(f), which represents the probability of selecting each possible action. For more information and examples, see rlDiscreteCategoricalActor.
- For stochastic actors with continuous action spaces cannot rely on custom basis functions (they can only use neural network approximators, due to the need to enforce positivity for the standard deviations). For more information and examples, see rlContinuousGaussianActor.

For any actor, W must have as many columns as the number of elements in f, and as many rows as the number of elements in B. B must be only a function of the observation.

For an example that trains a custom agent that uses a linear basis function, see "Train Custom LQR Agent" on page 5-466.

## Create an Agent

Once you create your actor and critic, you can create a reinforcement learning agent that uses them. For example, create a PG agent using a given actor and critic (baseline) network.

```
agentOpts = rlPGAgentOptions(UseBaseline=true);
agent = rlPGAgent(actor,baseline,agentOpts);
```

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

You can obtain the actor and critic from an existing agent using getActor and getCritic, respectively.

You can also set the actor and critic of an existing agent using setActor and setCritic, respectively. The input and output layers of the actor and critic must match the observation and action specifications of the original agent.

## See Also

## More About


- "Reinforcement Learning Agents" on page 3-2
- "Import Neural Network Models" on page 4-15

# Import Neural Network Models

To create function approximators for reinforcement learning, you can import pre-trained deep neural networks or deep neural network layer architectures using the Deep Learning Toolbox network import functionality. You can import:

- Open Neural Network Exchange (ONNX) models, which require the Deep Learning Toolbox Converter for ONNX Model Format support package software. For more information, `importONNXLayers`.
- TensorFlow-Keras networks, which require Deep Learning Toolbox Converter for TensorFlow Models support package software. For more information, see `importKerasLayers`.
- Caffe convolutional networks, which require Deep Learning Toolbox Importer for Caffe Models support package software. For more information, see `importCaffeLayers`.

After you import a deep neural network, you can create an actor or critic object, such as `rlQValueFunction` or `rlDiscreteCategoricalActor`.

When you import deep neural network architectures, consider the following.

- The dimensions of the imported network architecture input and output layers must match the dimensions of the corresponding action, observation, or reward dimensions for your environment.
- After importing the network architecture, you must set the names of the input and output layers to match the names of the corresponding action and observation specifications.

For more information on the deep neural network architectures supported for reinforcement learning, see “Create Policies and Value Functions” on page 4-2.

## Import Actor and Critic for Image Observation Application

As an example, assume that you have an environment with a 50-by-50 grayscale image observation signal and a continuous action space. To train a policy gradient agent, you require the following function approximators, both of which must have a single 50-by-50 image input observation layer and a single scalar output value.

- **Actor** — Selects an action value based on the current observation
- **Critic** — Estimates the expected long-term reward based on the current observation

Also, assume that you have the following network architectures to import:

- A deep neural network architecture for the actor with a 50-by-50 image input layer and a scalar output layer, which is saved in the ONNX format (`criticNetwork.onnx`).
- A deep neural network architecture for the critic with a 50-by-50 image input layer and a scalar output layer, which is saved in the ONNX format (`actorNetwork.onnx`).

To import the critic and actor networks, use the `importONNXLayers` function without specifying an output layer.

```
criticNetwork = importONNXLayers("criticNetwork.onnx");
actorNetwork = importONNXLayers("actorNetwork.onnx");
```

These commands generate a warning, which states that the network is trainable until an output layer is added. When you use an imported network to create an actor or critic, Reinforcement Learning Toolbox software automatically adds an output layer for you.

After you import the networks, create the actor and critic function approximators. To do so, first obtain the observation and action specifications from the environment.

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

Create the critic, specifying the name of the input layer of the critic network as the observation name. Since the critic network has a single observation input and a single action output, use a value-function.

```
critic = rlValueFunction(criticNetwork,obsInfo,...
             ObservationInputNames={criticNetwork.Layers(1).Name});
```

Create the actor, specifying the name of the input layer of the actor network as the observation name and the output layer of the actor network as the observation name. Since the actor network has a single scalar output, use a continuous deterministic actor.

```
actor = rlContinuousDeterministicActor(actorNetwork,obsInfo,actInfo,...
             ObservationInputNames={actorNetwork.Layers(1).Name});
```

You can then:

- Create an agent using this actor and critic. For more information, see “Reinforcement Learning Agents” on page 3-2.
- Set the actor and critic in an existing agent using `setActor` and `setCritic`, respectively.

## See Also

## More About

- “Create Policies and Value Functions” on page 4-2
- “Reinforcement Learning Agents” on page 3-2

# Train and Validate Agents


- “Train Reinforcement Learning Agents” on page 5-3
- “Train Agents Using Parallel Computing and GPUs” on page 5-8
- “Design and Train Agent Using Reinforcement Learning Designer” on page 5-12
- “Specify Training Options in Reinforcement Learning Designer” on page 5-16
- “Specify Simulation Options in Reinforcement Learning Designer” on page 5-21
- “Log Training Data to Disk” on page 5-24
- “Train Reinforcement Learning Agent for Simple Contextual Bandit Problem” on page 5-30
- “Train Agent or Tune Environment Parameters Using Parameter Sweeping” on page 5-40
- “Train DQN Agent to Balance Cart-Pole System” on page 5-50
- “Train PG Agent to Balance Cart-Pole System” on page 5-57
- “Train AC Agent to Balance Cart-Pole System” on page 5-63
- “Train PG Agent with Baseline to Control Double Integrator System” on page 5-70
- “Train DDPG Agent to Control Double Integrator System” on page 5-77
- “Train DQN Agent to Swing Up and Balance Pendulum” on page 5-89
- “Train DDPG Agent to Swing Up and Balance Pendulum” on page 5-97
- “Train DDPG Agent to Swing Up and Balance Cart-Pole System” on page 5-106
- “Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal” on page 5-115
- “Train Reinforcement Learning Agents to Control Quanser QUBE Pendulum” on page 5-125
- “Run SIL and PIL Verification for Reinforcement Learning” on page 5-134
- “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation” on page 5-141
- “Create DQN Agent Using Deep Network Designer and Train Using Image Observations” on page 5-152
- “Train AC Agent to Balance Cart-Pole System Using Parallel Computing” on page 5-166
- “Train DDPG Agent to Control Flying Robot” on page 5-172
- “Train PPO Agent for a Lander Vehicle” on page 5-180
- “Train Multiple Agents to Perform Collaborative Task” on page 5-190
- “Train Multiple Agents for Area Coverage” on page 5-198
- “Train Multiple Agents for Path Following Control” on page 5-206
- “Train DDPG Agent for Adaptive Cruise Control” on page 5-215
- “Train DQN Agent for Lane Keeping Assist” on page 5-226
- “Train PPO Agent for Automatic Parking Valet” on page 5-235
- “Train DDPG Agent for Path-Following Control” on page 5-247
- “Train DQN Agent for Lane Keeping Assist Using Parallel Computing” on page 5-257
- “Train Biped Robot to Walk Using Reinforcement Learning Agents” on page 5-267
- “Quadruped Robot Locomotion Using DDPG Agent” on page 5-278

- “Train SAC Agent for Ball Balance Control” on page 5-286
- “Automatic Parking Valet with Unreal Engine Simulation” on page 5-300
- “Generate Reward Function from a Model Predictive Controller for a Servomotor” on page 5-315
- “Generate Reward Function from a Model Verification Block for a Water Tank System” on page 5-327
- “Train TD3 Agent for PMSM Control” on page 5-341
- “Water Distribution System Scheduling Using Reinforcement Learning” on page 5-353
- “Imitate MPC Controller for Lane Keeping Assist” on page 5-365
- “Train DDPG Agent with Pretrained Actor Network” on page 5-373
- “Imitate Nonlinear MPC Controller for Flying Robot” on page 5-383
- “Tune PI Controller Using Reinforcement Learning” on page 5-392
- “Train Reinforcement Learning Agent with Constraint Enforcement” on page 5-403
- “Train DQN Agent with LSTM Network to Control House Heating System” on page 5-414
- “Generate Policy Block for Deployment” on page 5-424
- “Train Reinforcement Learning Policy Using Custom Training Loop” on page 5-433
- “Custom Training Loop with Simulink Action Noise” on page 5-442
- “Create Agent for Custom Reinforcement Learning Algorithm” on page 5-456
- “Train Custom LQR Agent” on page 5-466
- “Train MBPO Agent to Balance Cart-Pole System” on page 5-472
- “Model-Based Reinforcement Learning Using Custom Training Loop” on page 5-484
- “Train DQN Agent Using Hindsight Experience Replay” on page 5-500

# Train Reinforcement Learning Agents

Once you have created an environment and reinforcement learning agent, you can train the agent in the environment using the `train` function. To configure your training, use an `rlTrainingOptions` object. For example, create a training option set `opt`, and train agent `agent` in environment `env`.

```
opt = rlTrainingOptions(...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=1000,...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=480);
trainResults = train(agent,env,opt);
```

If `env` is a multi-agent environment created with `rlSimulinkEnv`, specify the agent argument as an array. The order of the agents in the array must match the agent order used to create `env`. Multi-agent training is not supported for MATLAB environments.

For more information on creating agents, see “Reinforcement Learning Agents” on page 3-2. For more information on creating environments, see “Create MATLAB Reinforcement Learning Environments” on page 2-2 and “Create Simulink Reinforcement Learning Environments” on page 2-8.

---

**Note** `train` updates the agent as training progresses. This is possible because each agent is an handle object. To preserve the original agent parameters for later use, save the agent to a MAT-file:

```
save("initialAgent.mat","agent")
```

If you copy the agent into a new variable, the new variable will also always point to the most recent agent version with updated parameters. For more information about handle objects, see “Handle Object Behavior”.

---

Training terminates automatically when the conditions you specify in the `StopTrainingCriteria` and `StopTrainingValue` options of your `rlTrainingOptions` object are satisfied. You can also terminate training before any termination condition is reached by clicking **Stop Training** in the Reinforcement Learning Episode Manager.

When training terminates the training statistics and results are stored in the `trainResults` object.

Because `train` updates the agent at the end of each episode, and because `trainResults` stores the last training results along with data to correctly recreate the training scenario and update the episode manager, you can later resume training from the exact point at which it stopped. To do so, at the command line, type:

```
trainResults = train(agent,env,trainResults);
```

This starts the training from the last values of the agent parameters and training results object obtained after the previous `train` call.

The `trainResults` object contains, as one of its properties, the `rlTrainingOptions` object `opt` specifying the training option set. Therefore, to restart the training with updated training options, first change the training options in `trainResults` using dot notation. If the maximum number of episodes was already reached in the previous training session, you must increase the maximum number of episodes.

For example, disable displaying the training progress on Episode Manager, enable the `Verbose` option to display training progress at the command line, change the maximum number of episodes to 2000, and then restart the training, returning a new `trainResults` object as output.

```
trainResults.TrainingOptions.MaxEpisodes = 2000;
trainResults.TrainingOptions.Plots = "none";
trainResults.TrainingOptions.Verbose = 1;
trainResultsNew = train(agent,env,trainResults);
```

**Note** When training terminates, `agents` reflects the state of each agent at the end of the final training episode. The rewards obtained by the final agents are not necessarily the highest achieved during the training process, due to continuous exploration. To save agents during training, create an `rlTrainingOptions` object specifying the SaveAgentCriteria and SaveAgentValue properties and pass it to `train` as a `trainOpts` argument.

## Training Algorithm

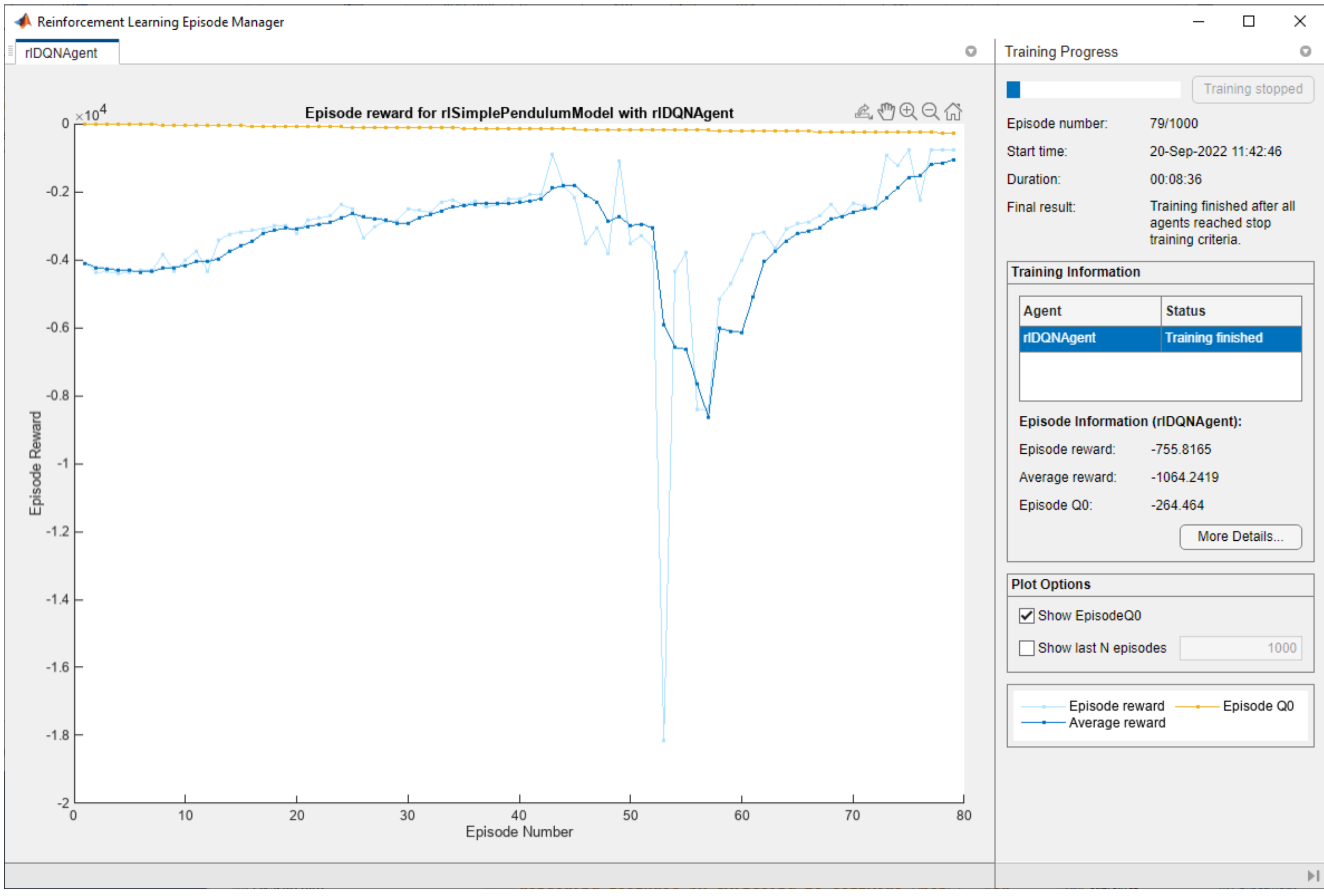
In general, training performs the following steps.

1. Initialize the agent.
2. For each episode:
   - a. Reset the environment.
   - b. Get the initial observation $s_0$ from the environment.
   - c. Compute the initial action $a_0 = \mu(s_0)$, where $\mu(s)$ is the current policy.
   - d. Set the current action to the initial action ($a \leftarrow a_0$), and set the current observation to the initial observation ($s \leftarrow s_0$).
   - e. While the episode is not finished or terminated, perform the following steps.
     - i. Apply action $a$ to the environment and obtain the next observation $s''$and the reward $r$.
     - ii. Learn from the experience set ($s,a,r,s'$).
     - iii. Compute the next action $a' = \mu(s')$.
     - iv. Update the current action with the next action ($a \leftarrow a'$) and update the current observation with the next observation ($s \leftarrow s'$).
     - v. Terminate the episode if the termination conditions defined in the environment are met.
3. If the training termination condition is met, terminate training. Otherwise, begin the next episode.

The specifics of how the software performs these steps depend on the configuration of the agent and environment. For instance, resetting the environment at the start of each episode can include randomizing initial state values, if you configure your environment to do so. For more information on agents and their training algorithms, see “Reinforcement Learning Agents” on page 3-2. To use parallel processing and GPUs to speed up training, see “Train Agents Using Parallel Computing and GPUs” on page 5-8.

## Episode Manager

By default, calling the `train` function opens the Reinforcement Learning Episode Manager, which lets you visualize the training progress.




The Episode Manager plot shows the reward for each episode (**EpisodeReward**) and a running average reward value (**AverageReward**).

For agents with a critic, **Episode Q0** is the estimate of the discounted long-term reward at the start of each episode, given the initial observation of the environment. As training progresses, if the critic is well designed and learns successfully, **Episode Q0** approaches in average the true discounted long-term reward, which may be offset from the **EpisodeReward** value because of discounting. For a well designed critic using an undiscounted reward (`DiscountFactor` is equal to `1`), then on average **Episode Q0** approaches the true episode reward, as shown in the preceding figure.

The Episode Manager also displays various episode and training statistics. You can also use the `train` function to return episode and training information. To turn off the Reinforcement Learning Episode Manager, set the `Plots` option of `rlTrainingOptions` to `"none"`.

## Save Candidate Agents

During training, you can save candidate agents that meet conditions you specify in the `SaveAgentCriteria` and `SaveAgentValue` options of your `rlTrainingOptions` object. For instance, you can save any agent whose episode reward exceeds a certain value, even if the overall condition for terminating training is not yet satisfied. For example, save agents when the episode reward is greater than `100`.

```
opt = rlTrainingOptions(SaveAgentCriteria="EpisodeReward",SaveAgentValue=100);
```

train stores saved agents in a MAT-file in the folder you specify using the SaveAgentDirectory option of rlTrainingOptions. Saved agents can be useful, for instance, to test candidate agents generated during a long-running training process. For details about saving criteria and saving location, see rlTrainingOptions.

After training is complete, you can save the final trained agent from the MATLAB workspace using the save function. For example, save the agent myAgent to the file finalAgent.mat in the current working directory.

```
save(opt.SaveAgentDirectory + "/finalAgent.mat",'agent')
```

By default, when DDPG and DQN agents are saved, the experience buffer data is not saved. If you plan to further train your saved agent, you can start training with the previous experience buffer as a starting point. In this case, set the SaveExperienceBufferWithAgent option to true. For some agents, such as those with large experience buffers and image-based observations, the memory required for saving the experience buffer is large. In these cases, you must ensure that enough memory is available for the saved agents.

## Validate Trained Policy

To validate your trained agent, you can simulate the agent within the training environment using the sim function. To configure the simulation, use rlSimulationOptions.

When validating your agent, consider checking how your agent handles the following:

- Changes to simulation initial conditions — To change the model initial conditions, modify the reset function for the environment. For example reset functions, see “Create MATLAB Environment Using Custom Functions” on page 2-41, “Create Custom MATLAB Environment from Template” on page 2-48, and “Create Simulink Reinforcement Learning Environments” on page 2-8.
- Mismatches between the training and simulation environment dynamics — To check such mismatches, create test environments in the same way that you created the training environment, modifying the environment behavior.

As with parallel training, if you have Parallel Computing Toolbox software, you can run multiple parallel simulations on multicore computers. If you have MATLAB Parallel Server software, you can run multiple parallel simulations on computer clusters or cloud resources. For more information on configuring your simulation to use parallel computing, see UseParallel and ParallelizationOptions in rlSimulationOptions.

## Environment Visualization

If your training environment implements the plot method, you can visualize the environment behavior during training and simulation. If you call plot(env) before training or simulation, where env is your environment object, then the visualization updates during training to allow you to visualize the progress of each episode or simulation.

Environment visualization is not supported when training or simulating your agent using parallel computing.

For custom environments, you must implement your own plot method. For more information on creating a custom environments with a plot function, see “Create Custom MATLAB Environment from Template” on page 2-48.

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train`

**Objects**
`rlTrainingOptions`

## Related Examples

- "Design and Train Agent Using Reinforcement Learning Designer" on page 5-12
- "Train Agents Using Parallel Computing and GPUs" on page 5-8

## More About

- "Reinforcement Learning Agents" on page 3-2

# Train Agents Using Parallel Computing and GPUs

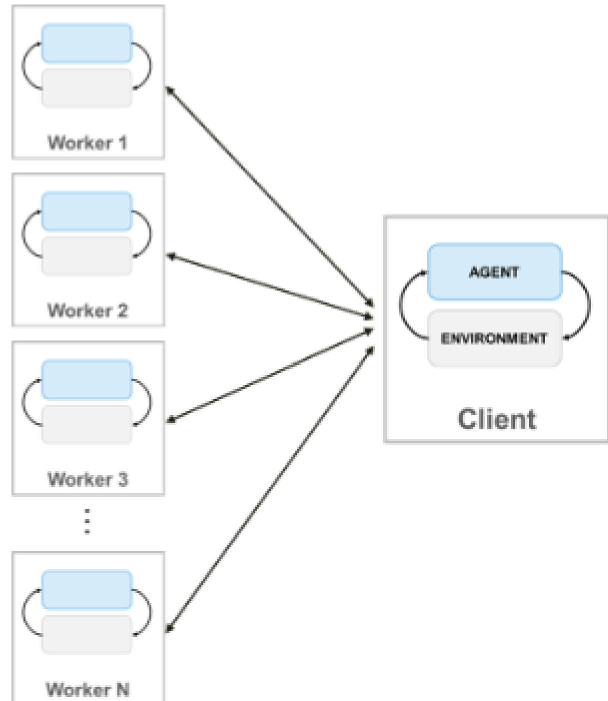
If you have Parallel Computing Toolbox software, you can run parallel simulations on multicore processors or GPUs. If you additionally have MATLAB Parallel Server software, you can run parallel simulations on computer clusters or cloud resources.

Note that parallel training and simulation of agents using recurrent neural networks, or agents within multi-agent environments, is not supported.

Independently on which devices you use to simulate or train the agent, once the agent has been trained, you can generate code to deploy the optimal policy on a CPU or GPU. This is explained in more detail in “Deploy Trained Reinforcement Learning Policies” on page 6-2.

## Using Multiple Processes

When you train agents using parallel computing, the parallel pool client (the MATLAB process that starts the training) sends copies of both its agent and environment to each parallel worker. Each worker simulates the agent within the environment and sends their simulation data back to the client. The client agent learns from the data sent by the workers and sends the updated policy parameters back to the workers.




To create a parallel pool of `N` workers, use the following syntax.

```
pool = parpool(N);
```

If you do not create a parallel pool using `parpool`, the `train` function automatically creates one using your default parallel pool preferences. For more information on specifying these preferences,

see “Specify Your Parallel Preferences” (Parallel Computing Toolbox). Note that using a parallel pool of thread workers, such as `pool = parpool("threads")`, is not supported.

To train an agent using multiple processes you must pass to the `train` function an `rlTrainingOptions` object in which the `UseParallel` property is set to `true`.

For more information on configuring your training to use parallel computing, see the `UseParallel` and `ParallelizationOptions` options in `rlTrainingOptions`. For an example on how to configure options for asynchronous advantage actor-critic (A3C) agent training, see the last example in `rlTrainingOptions`.

For an example that trains an agent using parallel computing in MATLAB, see “Train AC Agent to Balance Cart-Pole System Using Parallel Computing” on page 5-166. For an example that trains an agent using parallel computing in Simulink, see “Train DQN Agent for Lane Keeping Assist Using Parallel Computing” on page 5-257 and “Train Biped Robot to Walk Using Reinforcement Learning Agents” on page 5-267.

## Agent-Specific Parallel Training Considerations

Reinforcement learning agents can be trained in parallel in two main ways, experience-based parallelization, in which the workers only calculate experiences, and gradient-based parallelization, in which the workers also calculate the gradients that allow the agent approximators to learn.

### Experience-Based Parallelization (DQN, DDPG, TD3, SAC, PPO, TRPO)

When training an DQN, DDPG, TD3, SAC, PPO or TRPO agent in parallel, the environment simulation is done by the workers and the gradient computation is done by the client. Specifically, the workers simulate (their copy of) the agent within (their copy of) the environment, and send experience data (observation, action, reward, next observation, and a termination signal) to the client. The client then computes the gradients from experiences, updates the agent parameters and sends the updated parameters back to the workers, which then continue to perform simulations using their copy of the updated agent.

This type of parallel training is also known as experience-based parallelization, and can run using *asynchronous* training (that is the `Mode` property of the `rlTrainingOptions` object that you pass to the `train` function can be set to `"async"`).

Experience-based parallelization can reduce training time only when the computational cost of simulating the environment is high compared to the cost of optimizing network parameters. Otherwise, when the environment simulation is fast enough, the workers lie idle waiting for the client to learn and send back the updated parameters.

In other words, experience-based parallelization can improve sample efficiency (intended as the number of samples an agent can process within a given time) only when the ratio $R$ between the environment step complexity and the learning complexity is large. If both environment simulation and gradient computation (that is, learning) are similarly computationally expensive, experience-based parallelization is unlikely to improve sample efficiency. In this case, for off-policy agents that are supported in parallel (DQN, DDPG, TD3, and SAC) you can reduce the mini-batch size to make $R$ larger, thereby improving sample efficiency.

---

**Note** For experience-based parallelization, do not use all of your processor cores for parallel training. For example, if your CPU has six cores, train with four workers. Doing so provides more

resources for the parallel pool client to compute gradients based on the experiences sent back from the workers.

For and example of experience-based parallel training, see “Train DQN Agent for Lane Keeping Assist Using Parallel Computing” on page 5-257.

#### Gradient-Based Parallelization (AC and PG)

When training an AC or PG agent in parallel, both the environment simulation and gradient computations are done by the workers. Specifically, workers simulate (their copy of) the agent within (their copy of) the environment, obtain the experiences, compute the gradients from the experiences, and send the gradients to the client. The client averages the gradients, updates the agent parameters and sends the updated parameters back to the workers so they can continue to perform simulations using an updated copy of the agent.

This type of parallel training is also known as gradient-based parallelization, and allows you to achieve, in principle, a speed improvement which is nearly linear in the number of workers. However, this option requires *synchronous* training (that is the `Mode` property of the `rlTrainingOptions` object that you pass to the `train` function must be set to `"sync"`). This means that workers must pause execution until all workers are finished, and as a result the training only advances as fast as the slowest worker allows.

In general, limiting the number of workers in order to leave some processor cores for the client is not necessary when using gradient-based parallelization, because the gradients are not computed on the client. Therefore, for gradient-based parallelization, it might be beneficial to use all your processor cores for parallel training.

For and example of gradient-based parallel training, see “Train AC Agent to Balance Cart-Pole System Using Parallel Computing” on page 5-166.

## Using GPUs

You can speed up training by performing actor and critic operations (such as gradient computation and prediction), on a local GPU rather than a CPU. To do so, when creating a critic or actor, set its `UseDevice` option to `"gpu"` instead of `"cpu"`.

The `"gpu"` option requires both Parallel Computing Toolbox software and a CUDA enabled NVIDIA® GPU. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox).

You can use `gpuDevice` to query or select a local GPU device to be used with MATLAB.

Using GPUs is likely to be beneficial when you have a deep neural network in the actor or critic which has large batch sizes or needs to perform operations such as multiple convolutional layers on input images.

For an example on how to train an agent using the GPU, see “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation” on page 5-141.

## Using both Multiple Processes and GPUs

You can also train agents using both multiple processes and a local GPU (previously selected using `gpuDevice`) at the same time. To do so, first create a critic or actor approximator object in which the

UseDevice option is set to "gpu". You can then use the critic and actor to create an agent, and then train the agent using multiple processes. This is done by creating an rlTrainingOptions object in which UseParallel is set to true and passing it to the train function.

For gradient-based parallelization, (which must run in synchronous mode) the environment simulation is done by the workers, which also use their local GPU to calculate the gradients and perform a prediction step. The gradients are then sent back to the parallel pool client process which calculates the averages, updates the network parameters and sends them back to the workers so they continue to simulate the agent, with the new parameters, against the environment.

For experience-based parallelization, (which can run in asynchronous mode), the workers simulate the agent against the environment, and send experiences data back to the parallel pool client. The client then uses its local GPU to compute the gradients from the experiences, then updates the network parameters and sends the updated parameters back to the workers, which continue to simulate the agent, with the new parameters, against the environment.

Note that when using both parallel processing and GPU to train PPO agents, the workers use their local GPU to compute the advantages, and then send processed experience trajectories (which include advantages, targets and action probabilities) back to the client.

## See Also

**Functions**
train

**Objects**
rlTrainingOptions | gpuDevice

## Related Examples

- “Train AC Agent to Balance Cart-Pole System Using Parallel Computing” on page 5-166
- “Train DQN Agent for Lane Keeping Assist Using Parallel Computing” on page 5-257
- “Train Biped Robot to Walk Using Reinforcement Learning Agents” on page 5-267
- “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation” on page 5-141
- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “GPU Computing Requirements” (Parallel Computing Toolbox)
- “Reinforcement Learning Agents” on page 3-2
- “Create Policies and Value Functions” on page 4-2
- “Train Reinforcement Learning Agents” on page 5-3

# Design and Train Agent Using Reinforcement Learning Designer

This example shows how to design and train a DQN agent for an environment with a discrete action space using **Reinforcement Learning Designer**.

### Open the Reinforcement Learning Designer App

Open the **Reinforcement Learning Designer** app.

```
reinforcementLearningDesigner
```

Initially, no agents or environments are loaded in the app.

### Import Cart-Pole Environment

When using the **Reinforcement Learning Designer**, you can import an environment from the MATLAB workspace or create a predefined environment. For more information, see “Create or Import MATLAB Environments in Reinforcement Learning Designer” on page 2-5 and “Create or Import Simulink Environments in Reinforcement Learning Designer” on page 2-11.

For this example, use the predefined discrete cart-pole MATLAB environment. To import this environment, on the **Reinforcement Learning** tab, in the **Environments** section, select **New > Discrete Cart-Pole**.

In the **Environments** pane, the app adds the imported `Discrete CartPole` environment. To rename the environment, click the environment text. You can also import multiple environments in the session.

To view the dimensions of the observation and action space, click the environment text. The app shows the dimensions in the **Preview** pane.

This environment has a continuous four-dimensional observation space (the positions and velocities of both the cart and pole) and a discrete one-dimensional action space consisting of two possible forces, –10N or 10N. This environment is used in the “Train DQN Agent to Balance Cart-Pole System” on page 5-50 example. For more information on predefined control system environments, see “Load Predefined Control System Environments” on page 2-23.

### Create DQN Agent for Imported Environment

To create an agent, on the **Reinforcement Learning** tab, in the **Agent** section, click **New**. In the Create agent dialog box, specify the agent name, the environment, and the training algorithm. The default agent configuration uses the imported environment and the DQN algorithm. For this example, change the number of hidden units from 256 to 20. For more information on creating agents, see “Create Agents Using Reinforcement Learning Designer” on page 3-9.

Click **OK**.

The app adds the new agent to the **Agents** pane and opens a corresponding **agent1** document.

In the **Hyperparameter** section, under **Critic Optimizer Options** set **Learn rate** to `0.0001`.

For a brief summary of DQN agent features and to view the observation and action specifications for the agent, click **Overview**.

When you create a DQN agent in **Reinforcement Learning Designer**, the agent uses a default deep neural network structure for its critic. To view the critic network, on the **DQN Agent** tab, click **View Critic Model**.

The **Deep Learning Network Analyzer** opens and displays the critic structure.

Close the **Deep Learning Network Analyzer**.

### Train Agent

To train your agent, on the **Train** tab, first specify options for training the agent. For information on specifying training options, see "Specify Simulation Options in Reinforcement Learning Designer" on page 5-21.

For this example, specify the maximum number of training episodes by setting **Max Episodes** to `1000`. For the other training options, use their default values. The default criteria for stopping is when the average number of steps per episode (over the last `5` episodes) is greater than `500`.

To start training, click **Train**.

During training, the app opens the **Training Session** tab and displays the training progress in the **Training Results** document.

At any time during training, you can click on the **Stop** or **Stop Training** buttons to interrupt training and perform other operations on the command line.

At this point the **Resume**, **Accept**, and **Cancel** buttons in the **Training Session** tab give you the option to resume the training, accept the training results (which stores the training results and the trained agent in the app) or cancel the training altogether, respectively.

To resume training click **Resume**.

Here, the training stops when the average number of steps per episode is 500.

To accept the training results click **Accept**. In the **Agents** pane, the app adds the trained agent, `agent1_Trained`.

### Simulate Agent and Inspect Simulation Results

To simulate the trained agent, on the **Simulate** tab, first select `agent1_Trained` in the **Agent** drop-down list, then configure the simulation options. For this example, use the default number of episodes (`10`) and maximum episode length (`500`). For more information on specifying simulation options, see "Specify Simulation Options in Reinforcement Learning Designer" on page 5-21.

To simulate the agent, click **Simulate**.

The app opens the **Simulation Session** tab. After the simulation is completed, the **Simulation Results** document shows the reward for each episode as well as the reward mean and standard deviation.

For three episodes the agent was not able to reach the maximum reward of 500. This suggests that the robustness of the trained agent to different initial conditions might be improved. In this case, training the agent longer, for example by selecting an **Average Window Length** of `10` instead of `5`, yields better robustness. You can also modify some DQN agent options such as `BatchSize` and `TargetUpdateFrequency` to promote faster and more robust learning.

To analyze the simulation results, click **Inspect Simulation Data**. This opens the **Simulation Data Inspector**. For more information, see Simulation Data Inspector (Simulink).

You also have the option to preemptively clear from the **Simulation Data Inspector** any data that you might have loaded in a previous session. To do so, under **Inspect Simulation Data**, select **Clear and Inspect Simulation Data**.

In the **Simulation Data Inspector** you can view the saved signals for each simulation episode.

By default, the upper plot area is selected. To show the first state (the cart position), during the first episode, under **Run 1: Simulation Result**, open the `CartPoleStates` variable, and select `CartPoleStates(1,1)`. The cart goes outside the boundary after about 390 seconds, causing the simulation to terminate.

To also show the reward in the upper plot area, select the `Reward` variable. Note that the units on the vertical axis change accordingly.

Click the middle plot area, and select the third state (pole angle). Then click the bottom area and select the second and fourth state (cart velocity and pole angle derivative).

For a related example, in which a DQN agent is trained on the same environment, see “Train DQN Agent to Balance Cart-Pole System” on page 5-50.

Close the **Simulation Data Inspector**.

To accept the simulation results, on the **Simulation Session** tab, click **Accept**.

In the **Results** pane, the app adds the simulation results structure, `experience1`.

### Export Agent and Save Session

To select the trained agent and open the corresponding **agent1_Trained** document, under the **Agents** pane, double click on `agent1_Trained`.

Then, to export the trained agent to the MATLAB workspace, on the **Reinforcement Learning** tab, under **Export**, select the trained agent.

To save the app session, on the **Reinforcement Learning** tab, click **Save Session**. In the future, to resume your work where you left off, you can open the session in **Reinforcement Learning Designer**.

#### Simulate Agent at the Command Line

To simulate the agent at the MATLAB command line, first load the cart-pole environment.

```
env = rlPredefinedEnv("CartPole-Discrete");
```

The cart-pole environment has an environment visualizer that allows you to see how the system behaves during simulation and training.

Plot the environment and perform a simulation using the trained agent that you previously exported from the app.

```
plot(env)
xpr2 = sim(env,agent1_Trained);
```

During the simulation, the visualizer shows the movement of the cart and pole. In this simulation, the trained agent is able to stabilize the system.

Finally, display the cumulative reward for the simulation.

```
sum(xpr2.Reward)
```

```
env =
   500
```

As expected, the cumulative reward is 500.

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `analyzeNetwork`

## Related Examples

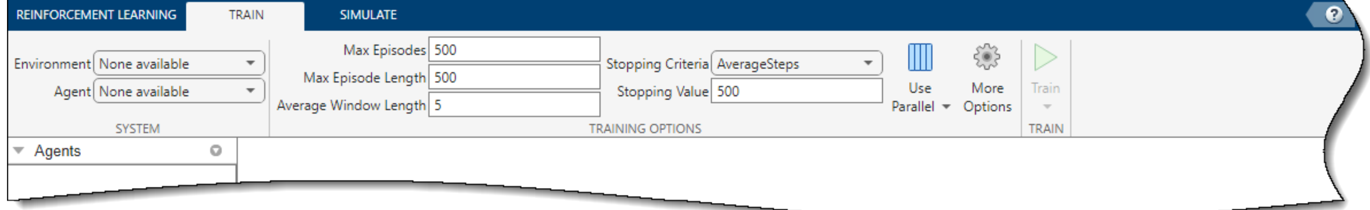
- “Train DQN Agent to Balance Cart-Pole System” on page 5-50

## More About

- “Load Predefined Control System Environments” on page 2-23
- “Create or Import MATLAB Environments in Reinforcement Learning Designer” on page 2-5
- “Create or Import Simulink Environments in Reinforcement Learning Designer” on page 2-11
- “Create Agents Using Reinforcement Learning Designer” on page 3-9
- “Reinforcement Learning Agents” on page 3-2
- “Deep Q-Network (DQN) Agents” on page 3-23
- “Create Policies and Value Functions” on page 4-2
- “Train Reinforcement Learning Agents” on page 5-3

# Specify Training Options in Reinforcement Learning Designer

To configure the training of an agent in the **Reinforcement Learning Designer** app, specify training options on the **Train** tab.




## Specify Basic Options

On the **Train** tab, you can specify the following basic training options.

| Option | Description |
| --- | --- |
| **Max Episodes** | Maximum number of episodes to train the agent, specified as a positive integer. |
| **Max Episode Length** | Maximum number of steps to run per episode, specified as a positive integer. |
| **Stopping Criteria** | Training termination condition, specified as one of the following values.<br>• `AverageSteps` — Stop training when the running average number of steps per episode equals or exceeds the critical value specified by **Stopping Value**.<br>• `AverageReward` — Stop training when the running average reward equals or exceeds the critical value.<br>• `EpisodeReward` — Stop training when the reward in the current episode equals or exceeds the critical value.<br>• `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.<br>• `EpisodeCount` — Stop training when the number of training episodes equals or exceeds the critical value. |
| **Stopping Value** | Critical value of the training termination condition in **Stopping Criteria**, specified as a scalar. |
| **Average Window Length** | Window length for averaging the scores, rewards, and number of steps for the agent when either **Stopping Criteria** or **Save agent criteria** specify an averaging condition. |

## Specify Additional Options

To specify additional training options, on the **Train** tab, click **More Options**.

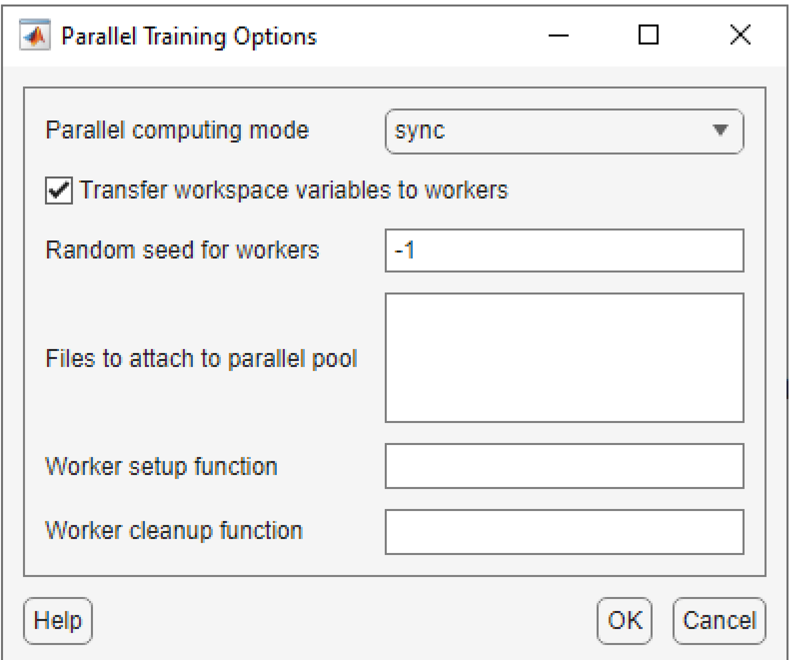
In the More Training Options dialog box, you can specify the following options.

| Option | Description |
| --- | --- |
| **Save agent criteria** | Condition for saving agents during training, specified as one of the following values.<br>• `none` — Do not save any agents during training.<br>• `AverageSteps` — Save the agent when the running average number of steps per episode equals or exceeds the critical value specified by **Save agent value**.<br>• `AverageReward` — Save the agent when the running average reward equals or exceeds the critical value.<br>• `EpisodeReward` — Save the agent when the reward in the current episode equals or exceeds the critical value.<br>• `GlobalStepCount` — Save the 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.<br>• `EpisodeCount` — Save the agent when the number of training episodes equals or exceeds the critical value. |
| **Save agent value** | Critical value of the save agent condition in **Save agent criteria**, specified as a scalar or `"none"`. |
| **Save directory** | Folder for saved agents. If you specify a name and the folder does not exist, the app creates the folder in the current working directory.<br>To interactively select a folder, click **Browse**. |
| **Show verbose output** | Select this option to display training progress at the command line. |
| **Stop on Error** | Select this option to stop training when an error occurs during an episode. |
| **Training plot** | Option to graphically display the training progress in the app, specified as one of the following values. `"training-progress"` or `"none"`.<br>• `training-progress` — Show training progress<br>• `none` — Do not show training progress |

## Specify Parallel Training Options

To train your agent using parallel computing, on the **Train** tab, click . Training agents using parallel computing requires Parallel Computing Toolbox software. For more information, see "Train Agents Using Parallel Computing and GPUs" on page 5-8.

To specify options for parallel training, select **Use Parallel > Parallel training options**.



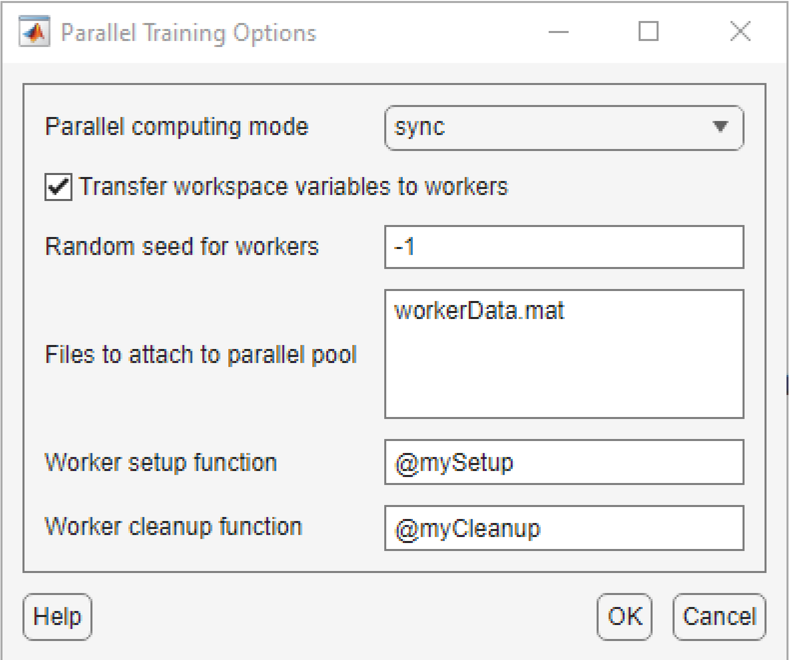

In the Parallel Training Options dialog box, you can specify the following training options.

| Option | Description |
|---|---|
| **Parallel computing mode** | Parallel computing mode, specified as one of the following values.<br>• `sync` — Use `parpool` to run synchronous training on the available workers. The parallel pool client (the process that starts the training) updates the parameters of its actor and critic, based on the results from all the workers, and sends the updated parameters to all workers. In this case, workers must pause execution until all workers are finished, and as a result the training only advances as fast as the slowest worker allows.<br>• `async` — Use `parpool` to run asynchronous training on the available workers. In this case, workers send their data back to the client as soon as they finish and receive updated parameters from the client. The workers then continue with their task. |
| **Transfer workspace variables to workers** | Select this option to send model and workspace variables to parallel workers. When you select this option, the parallel pool client (the process that starts the training) sends variables used in models and defined in the MATLAB workspace to the workers. |
| **Random seed for workers** | Randomizer initialization for workers, specified as one of the following values.<br>• –1 — Assign a unique random seed to each worker. The value of the seed is the worker ID.<br>• –2 — Do not assign a random seed to the workers.<br>• Vector — Manually specify the random seed for each worker. The number of elements in the vector must match the number of workers. |

| Option | Description |
| --- | --- |
| **Files to attach to parallel pool** | Additional files to attach to the parallel pool. Specify names of files in the current working directory, with one name on each line. |
| **Worker setup function** | 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. |
| **Worker cleanup function** | 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. |

The following figure shows an example parallel training configuration the following files and functions.

- Data file attached to the parallel pool — `workerData.mat`
- Worker setup function — `mySetup.m`
- Worker cleanup function — `myCleanup.m`




## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train`

**Objects**
`rlTrainingOptions`

## Related Examples

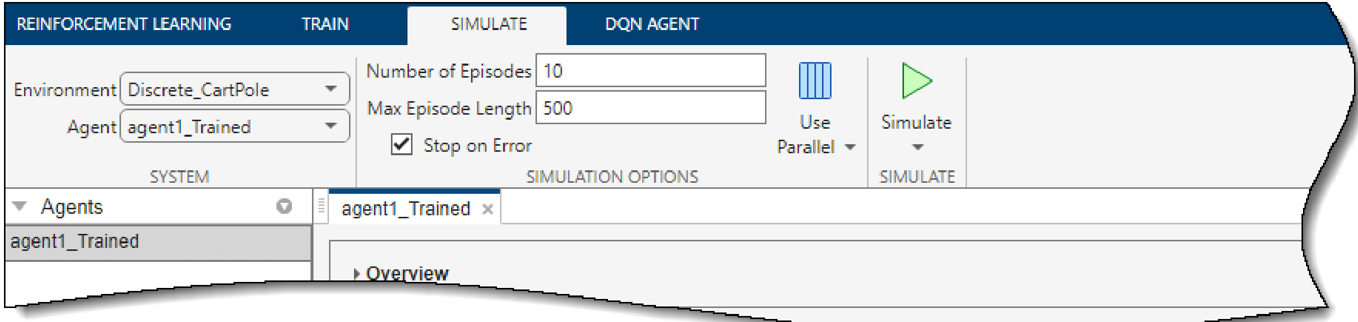

- “Design and Train Agent Using Reinforcement Learning Designer” on page 5-12
- “Specify Simulation Options in Reinforcement Learning Designer” on page 5-21


## More About


- “Create Agents Using Reinforcement Learning Designer” on page 3-9
- “Reinforcement Learning Agents” on page 3-2
- “Create Policies and Value Functions” on page 4-2
- “Train Reinforcement Learning Agents” on page 5-3

# Specify Simulation Options in Reinforcement Learning Designer

To configure the simulation of an agent in the **Reinforcement Learning Designer** app, specify simulation options on the **Simulate** tab.




## Specify Basic Options

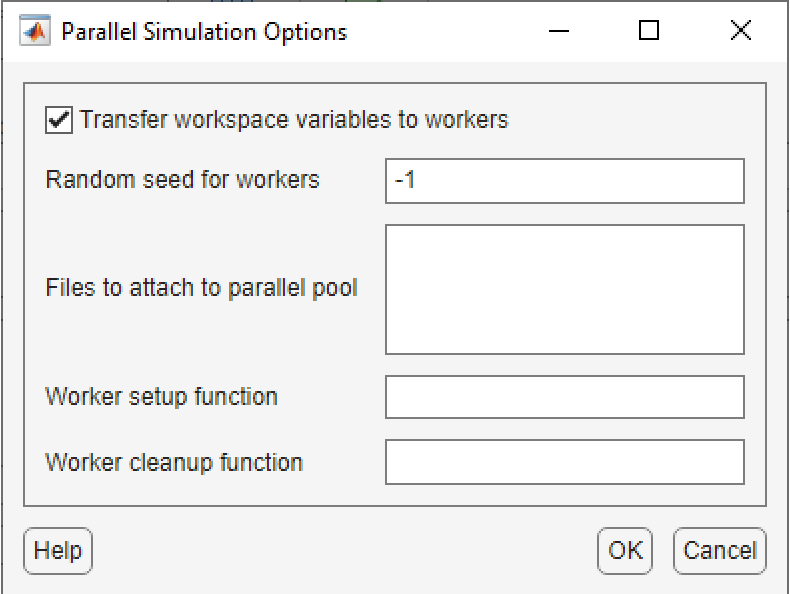
On the **Simulate** tab, you can specify the following basic simulation options.

| Option | Description |
|---|---|
| **Number of Episodes** | Number of episodes to simulate the agent, specified as a positive integer. At the start of each simulation episode, the app resets the environment. |
| **Max Episode Length** | Number of steps to run the simulation, specified as 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. |
| **Stop on Error** | Select this option to stop simulation when an error occurs during an episode. |

## Specify Parallel Simulation Options

To simulate your agent using parallel computing, on the **Simulate** tab, click . Simulating agents using parallel computing requires Parallel Computing Toolbox software. For more information, see “Train Agents Using Parallel Computing and GPUs” on page 5-8.

To specify options for parallel simulation, select **Use Parallel** > **Parallel training options**.



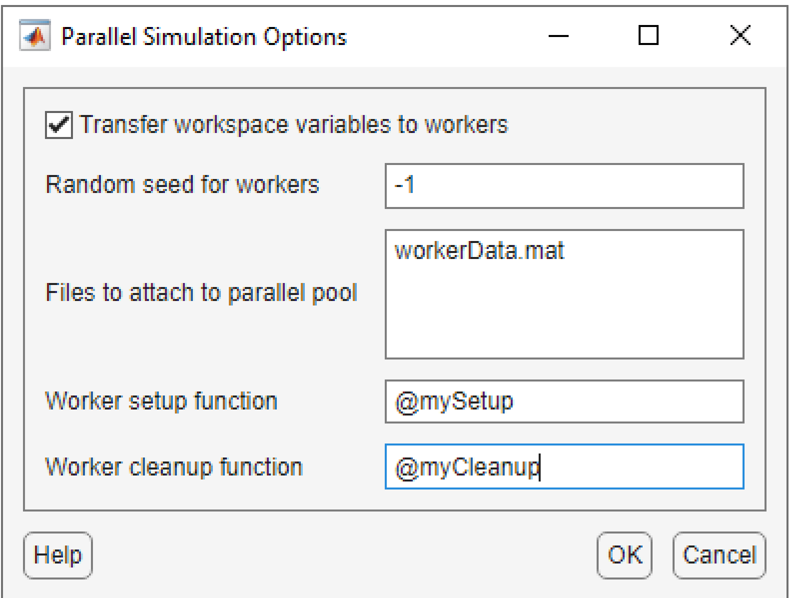

In the Parallel Simulation Options dialog box, you can specify the following training options.

| Option | Description |
| --- | --- |
| **Transfer workspace variables to workers** | Select this option to send model and workspace variables to parallel workers. When you select this option, the parallel pool client (the process that starts the training) sends variables used in models and defined in the MATLAB workspace to the workers. |
| **Random seed for workers** | Randomizer initialization for workers, specified as one of the following values.<br>• `–1` — Assign a unique random seed to each worker. The value of the seed is the worker ID.<br>• `–2` — Do not assign a random seed to the workers.<br>• Vector — Manually specify the random seed for each worker. The number of elements in the vector must match the number of workers. |
| **Files to attach to parallel pool** | Additional files to attach to the parallel pool. Specify names of files in the current working directory, with one name on each line. |
| **Worker setup function** | Function to run before simulation starts, specified as the name of a function having no input arguments. This function is run once per worker before simulation begins. Write this function to perform any processing that you need prior to training. |
| **Worker cleanup function** | Function to run after simulation ends, specified as the name of a function having no input arguments. You can write this function to clean up the workspace or perform other processing after training terminates. |

The following figure shows an example parallel training configuration the following files and functions.

- Data file attached to the parallel pool — `workerData.mat`
- Worker setup function — `mySetup.m`

- Worker cleanup function — `myCleanup.m`




## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`sim`

**Objects**
`rlSimulationOptions`

## Related Examples

- "Design and Train Agent Using Reinforcement Learning Designer" on page 5-12
- "Specify Training Options in Reinforcement Learning Designer" on page 5-16

## More About

- "Create Agents Using Reinforcement Learning Designer" on page 3-9
- "Reinforcement Learning Agents" on page 3-2
- "Train Reinforcement Learning Agents" on page 5-3
- "Train Agents Using Parallel Computing and GPUs" on page 5-8

# Log Training Data to Disk

This example shows how to log custom data to disk when training agents using the Reinforcement Learning Toolbox™ train function.

### Overview

The general steps for data logging are:

1. Create a data logger object using the `rlDataLogger` function.
2. Configure the data logger object with callback functions to specify the data to log at different stages of the training process.
3. Specify the logger object as a name-value input argument in the `train` function.

### Create Data Logger

Create a file data logger object using the `rlDataLogger` function.

```
fileLogger = rlDataLogger()
```

```
fileLogger = 
  FileLogger with properties:

           LoggingOptions: [1x1 rl.logging.option.MATFileLoggingOptions]
       EpisodeFinishedFcn: []
     AgentStepFinishedFcn: []
    AgentLearnFinishedFcn: []
```

Specify options to log data such as the logging directory and the frequency (in number of episodes) at which the data logger writes data to disk. This step is optional.

```
% Specify a logging directory. You must have write
% access for this directory.
logDir = fullfile(pwd,"myDataLog");
fileLogger.LoggingOptions.LoggingDirectory = logDir;

% Specify a naming rule for files. The naming rule episode<id>
% saves files as episode001.mat, episode002.mat and so on.
fileLogger.LoggingOptions.FileNameRule = "episode<id>";

% Set the frequency (in number of episodes) at which
% the data logger writes data to disk
fileLogger.LoggingOptions.DataWriteFrequency = 1;
```

### Configure Data Logging

Training data of interest is generated at different stages of the training loop; for example, experience data is available after the completion of an episode. You can configure the logger object with callback functions to log data at these stages. The functions must return either a structure containing the data to log, or an empty array if no data needs to be logged at that stage.

The callback functions are:

- `EpisodeFinishedFcn` - Callback function to log data such as experiences, logged Simulink signals, or initial observations. The trainng loop executes this function after the completion of a training episode. The following is an example for the function.

```
function dataToLog = myEpisodeLoggingFcn(data)
% data is a structure that contains the following fields:
%
% EpisodeCount: The current episode number.
% Environment: Environment object.
% Agent: Agent object.
% Experience: A structure containing the experiences
%             from the current episode.
% EpisodeInfo: A structure containing the fields
%              CumulativeReward, StepsTaken, and
%              InitialObservation.
% SimulationInfo: Contains simulation information for the
%                 current episode.
%                 For MATLAB environments this is a structure
%                 with the field "SimulationError".
%                 For Simulink environments this is a
%                 Simulink.SimulationOutput object.
%
% dataToLog is a structure containing the data to be logged
% to disk.

% Write your code to log data to disk. For example,
% dataToLog.Experience = data.Experience;

dataToLog.Experience = data.Experience;
dataToLog.EpisodeReward = data.EpisodeInfo.CumulativeReward;
if data.EpisodeInfo.StepsTaken > 0
    dataToLog.EpisodeQ0 = evaluateQ0(data.Agent, ...
        data.EpisodeInfo.InitialObservation);
else
    dataToLog.EpisodeQ0 = 0;
end
```

- `AgentStepFinishedFcn` - Callback function to log data such as the state of exploration. The trainng loop executes this function after the completion of an agent step within an episode. The following is a example for the function.

```
function dataToLog = myAgentStepLoggingFcn(data)
% data is a structure that contains the following fields:
%
% EpisodeCount:   The current episode number.
% AgentStepCount: The cumulative number of steps taken by
%                 the agent.
% SimulationTime: The current simulation time in the
%                 environment.
% Agent:          Agent object.
%
% dataToLog is a structure containing the data to be logged
% to disk.

% Write your code to log data to disk. For example,
% noiseState = getState(getExplorationPolicy(data.Agent));
% dataToLog.noiseState = noiseState;
```

```
policy = getExplorationPolicy(data.Agent);
if hasprop(policy,"NoiseType")
    state = getState(policy);
    if strcmp(policy.NoiseType,"ou")
        dataToLog.OUNoise = state.Noise{1};
        dataToLog.StandardDeviation = state.StandardDeviation{1};
    elseif strcmp(policy.NoiseType,"gaussian")
        dataToLog.StandardDeviation = state.StandardDeviation{1};
    end
else
    dataToLog = [];
end
```

- `AgentLearnFinishedFcn` - Callback function to log data such as the actor and critic training losses. The trainng loop executes this function after the completion of the learning subroutine. The following is a example for the function.

```
function dataToLog = myAgentLearnLoggingFcn(data)
% data is a structure that contains the following fields:
%
% EpisodeCount:   The current episode number.
% AgentStepCount: The cumulative number of steps taken by
%                 the agent.
% AgentLearnCount: The cumulative number of learning steps
%                  taken by the agent.
% EnvModelTrainingInfo: A structure containing the fields:
%                       a. TransitionFcnLoss
%                       b. RewardFcnLoss
%                       c. IsDoneFcnLoss.
%                       This is applicable for model-based
%                       agent training.
% Agent: Agent object.
% ActorLoss: Training loss of the actor.
% CriticLoss: Training loss of the critic.
%
% dataToLog is a structure containing the data to be logged
% to disk.

% Write your code to log data to disk. For example,
% dataToLog.ActorLoss = data.ActorLoss;

dataToLog.ActorLoss  = data.ActorLoss;
dataToLog.CriticLoss = data.CriticLoss;
```

For this example, configure only the `AgentLearnFinishedFcn` callback. The function `logTrainingLoss` logs the actor and critic training losses and is provided at the end of this example.

```
fileLogger.AgentLearnFinishedFcn = @logTrainingLoss;
```

### Run Training

Create a predefined `CartPole-continuous` environment and a deep deterministic policy gradient (DDPG) agent for training.

```
% Set the random seed to facilitate reproducibility
rng(0);
```

```
% Create a CartPole-continuous environment
env = rlPredefinedEnv("CartPole-continuous");

% Create a DDPG agent
agent = rlDDPGAgent(getObservationInfo(env), getActionInfo(env));
agent.AgentOptions.NoiseOptions.StandardDeviationDecayRate = 0.001;
```

Specify training options to train the agent for 100 episodes without visualization in the Episode Manager.

Note that you can still use the `SaveAgentCriteria`, `SaveAgentValue` and `SaveAgentDirectory` options of the `rlTrainingOptions` object to save the agent duing training. Such options do not affect (and are not affected by) any usage of `FileLogger` or `MonitorLogger` objects.

```
trainOpts = rlTrainingOptions( ...
    MaxEpisodes=100, ...
    Plots="training-progress");
```

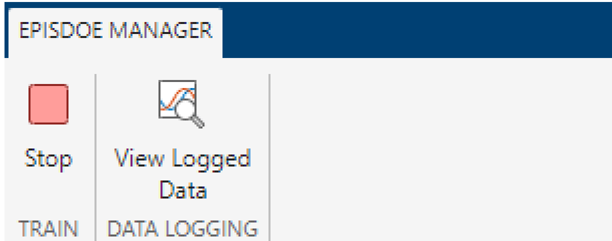
Train the agent using the `train` function. Specify the file logger object in the `Logger` name-value option.

```
result = train(agent, env, trainOpts, Logger=fileLogger);
```

The logged data is saved within the directory specified by `logDir`.

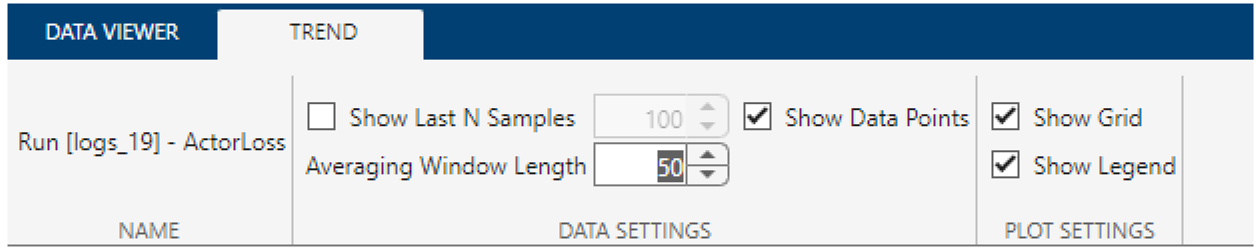
#### Visualize Logged Data

You can visualize data logged to disk using the interactive **Reinforcement Learning Data Viewer** graphical user interface. To open the visualization, click **View Logged Data** in the **Reinforcement Learning Episode Manager** window.




To create plots in the **Reinforcement Learning Data Viewer**, select a data from the **Data** panel and a choice of plot from the toolstrip. The following image shows a plot of the ActorLoss data generated using the Trend plot type. The plot shows logged data points and a moving average line.

On the toolstrip, navigate to the **Trend** tab to configure plot options. Set the window length for averaging data to `50`. The plot updates with the new configuration.




### Local Functions

```
function dataToLog = logTrainingLoss(data)

% Function to log the actor and critic training losses
dataToLog.ActorLoss = data.ActorLoss;
```

```
dataToLog.CriticLoss = data.CriticLoss;
end
```

## See Also

**Functions**
rlDataLogger | train | sim

**Objects**
FileLogger | MonitorLogger | trainingProgressMonitor | rlTrainingOptions

## Related Examples


- "Train DDPG Agent to Swing Up and Balance Cart-Pole System" on page 5-106
- "Train Agent or Tune Environment Parameters Using Parameter Sweeping" on page 5-40


## More About


- "Load Predefined Control System Environments" on page 2-23
- "Monitor Custom Training Loop Progress"
- "Train Reinforcement Learning Agents" on page 5-3
- "Train Agents Using Parallel Computing and GPUs" on page 5-8

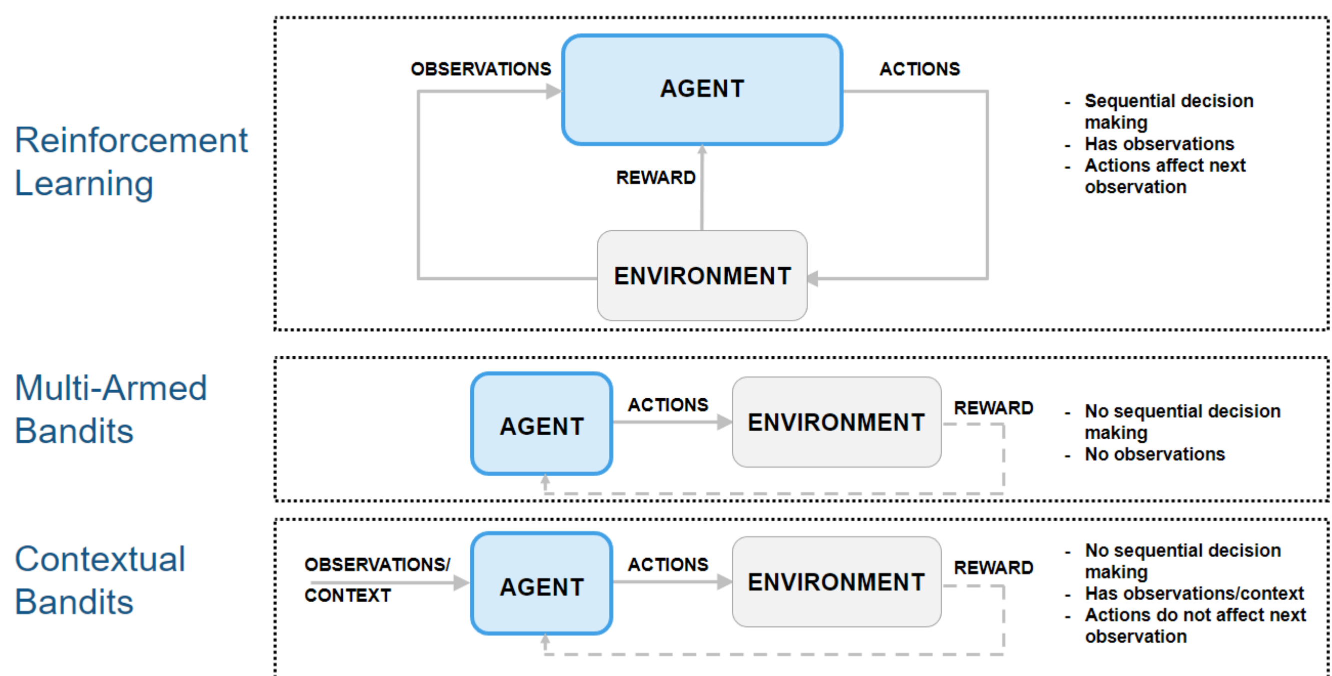
# Train Reinforcement Learning Agent for Simple Contextual Bandit Problem

This example shows how to solve a contextual bandit problem [1] using reinforcement learning by training DQN and Q agents. For more information on these agents, see “Deep Q-Network (DQN) Agents” on page 3-23 and “Q-Learning Agents” on page 3-17.

In contextual bandit problems, an agent selects an action given the initial observation (context), it receives a reward, and the episode terminates. Hence, the agent action does not affect the next observation.

Contextual bandits can be used for various applications such as hyperparameter tuning, recommender systems, medical treatment, and 5G communication.

The following figure show how multi-armed bandits and contextual bandits are special cases of reinforcement learning.




In bandit problems, the environment has no dynamics, so the reward is only influenced by the current action and (for contextual bandits) the current observation (in these problems the observation is also referred to as context).

Neither rewards nor observations are influenced by any environment state (or by previous actions or observations), so the environment does not evolve along the time dimension, and there is no sequential decision making. The problem then become one of finding the action that maximizes the current reward (given a context, if present). Single-armed bandit problems are just special cases of multi-armed bandit problems in which the action is a scalar instead of a vector.

### Environment

The contextual bandit environment in this example is defined as follows:

- **Observation** (discrete): {1, 2}

The context (initial observation) is sampled randomly.

$$\Pr(s = 1) = 0.5$$
$$\Pr(s = 2) = 0.5$$

- **Action** (discrete): {1, 2, 3}
- **Reward**:

Rewards in this environment are stochastic. The probability of each observation and action pair is defined as follows.

1. $s = 1,\ a = 1$

$$\Pr(r = 5 \mid s = 1, a = 1) = 0.3$$
$$\Pr(r = 2 \mid s = 1, a = 1) = 0.7$$

2. $s = 1,\ a = 2$

$$\Pr(r = 10 \mid s = 1, a = 2) = 0.1$$
$$\Pr(r = 1 \mid s = 1, a = 2) = 0.9$$

3. $s = 1,\ a = 3$

$$\Pr(r = 3.5 \mid s = 1, a = 3) = 1$$

4. $s = 2,\ a = 1$

$$\Pr(r = 10 \mid s = 2, a = 1) = 0.2$$
$$\Pr(r = 2 \mid s = 2, a = 1) = 0.8$$

5. $s = 2,\ a = 2$

$$\Pr(r = 3 \mid s = 2, a = 2) = 1$$

6. $s = 2,\ a = 3$

$$\Pr(r = 5 \mid s = 2, a = 3) = 0.5$$
$$\Pr(r = 0.5 \mid s = 2, a = 3) = 0.5$$

Note that the agent does not know these distributions.

- **Is-Done signal**: Since this is a contextual bandit problem, each episode has only one step. Hence, the Is-Done signal is always 1.

### Create Environment Interface

Create the contextual bandit environment using `ToyContextualBanditEnvironment`, located in this example folder.

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

Fix the random generator seed for reproducibility.

```
rng(1)
```

### Create DQN Agent

Create a DQN agent with a default network structure using `rlAgentInitializationOptions`.

```
agentOpts = rlDQNAgentOptions(...
    UseDoubleDQN = false, ...
    TargetSmoothFactor = 1, ...
    TargetUpdateFrequency = 4, ...
    MiniBatchSize = 64);
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 0.0005;

initOpts = rlAgentInitializationOptions(NumHiddenUnit = 16);

DQNagent = rlDQNAgent(obsInfo, actInfo, initOpts, agentOpts);
```

### Train Agent

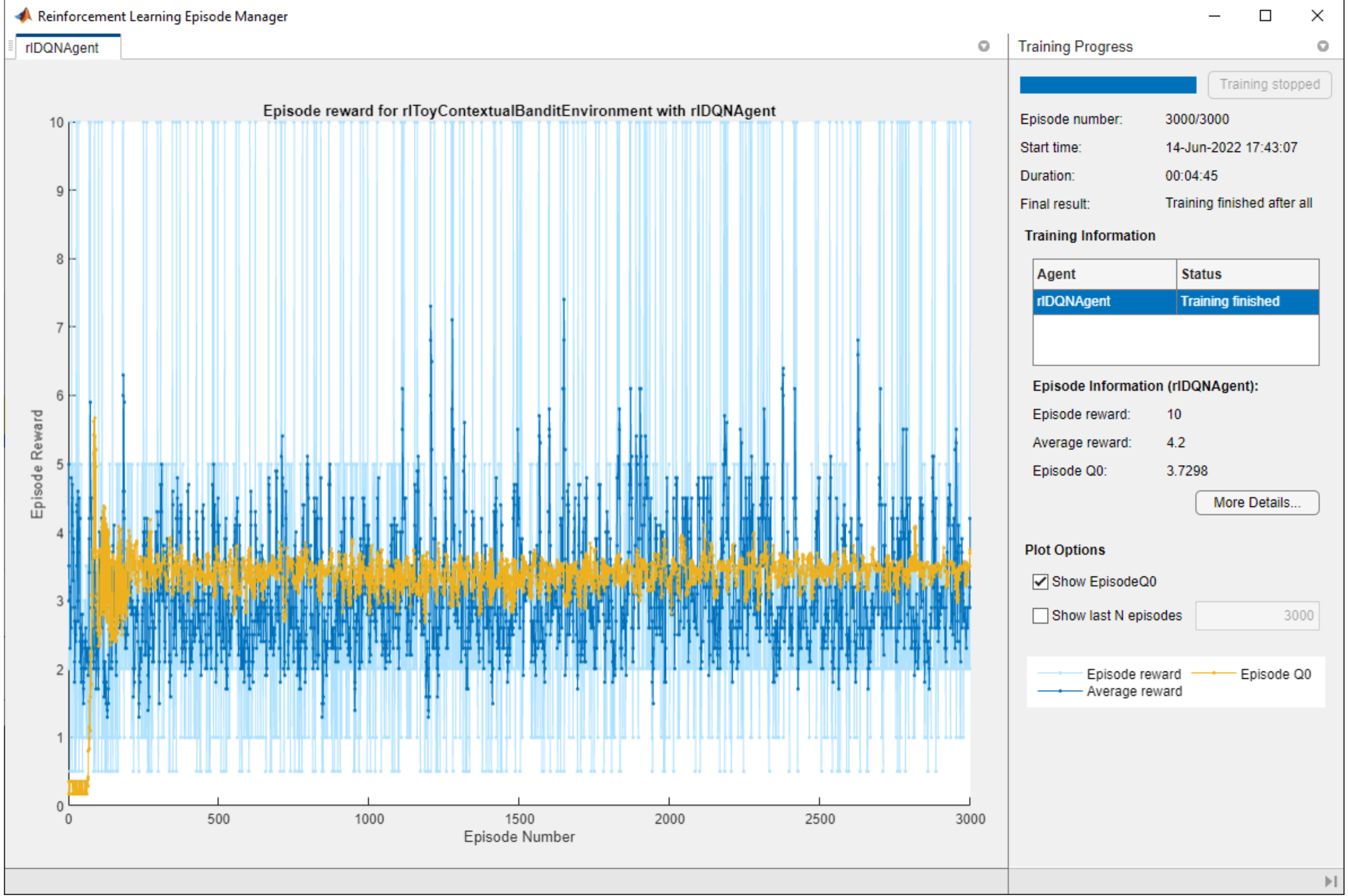
To train the agent, first specify the training options. For this example, use the following options:

- Train for 3000 episodes.
- Since this is a contextual bandit problem, and each episode has only one step, set `MaxStepsPerEpisode` to `1`.

For more information, see `rlTrainingOptions`.

Train the agent using the `train` function. To save time while running this example, load a pre-trained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to be `true`.

```
MaxEpisodes = 3000;
trainOpts = rlTrainingOptions(...
    MaxEpisodes = MaxEpisodes, ...
    MaxStepsPerEpisode = 1, ...
    Verbose = false, ...
    Plots = "training-progress",...
    StopTrainingCriteria = "EpisodeCount",...
    StopTrainingValue = MaxEpisodes);

doTraining = false;
if doTraining
    % Train the agent
    trainingStats = train(DQNagent,env,trainOpts);
else
    % Load the pre-trained agent for the example
    load("ToyContextualBanditDQNAgent.mat","DQNagent")
end
```




### Validate DQN Agent

Assume that you know the distribution of the rewards, and you can compute the optimal actions. Validate the agent's performance by comparing these optimal actions with the actions selected by the agent. First, compute the true expected rewards with the true distributions.

1. The expected reward of each action at s=1 is as follows.

If $a = 1$

$E[R] = 0.3*5 + 0.7*2 = 2.9$

If $a = 2$

$E[R] = 0.1*10 + 0.9*1 = 1.9$

If $a = 3$

$E[R] = 3.5$

Hence, the optimal action is 3 when s=1.

2. The expected reward of each action at s=2 is as follows.

If $a = 1$

$E[R] = 0.2*10 + 0.8*2 = 3.6$

If $a = 2$

$E[R] = 3.0$

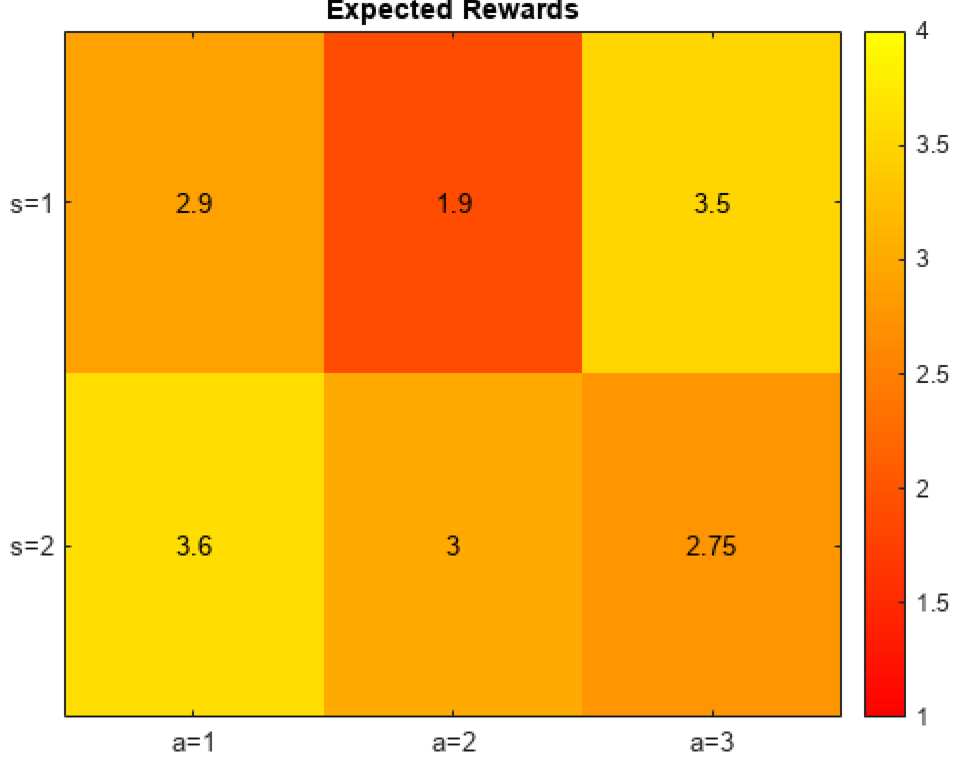
If $a = 3$

$E[R] = 0.5*5 + 0.5*0.5 = 2.75$

Hence, the optimal action is 1 when s=2.

With enough sampling, the Q-values should be closer to the true expected reward. Visualize the true expected rewards.

```
ExpectedRewards = zeros(2,3);
ExpectedRewards(1,1) = 0.3*5 + 0.7*2;
ExpectedRewards(1,2) = 0.1*10 + 0.9*1;
ExpectedRewards(1,3) = 3.5;
ExpectedRewards(2,1) = 0.2*10 + 0.8*2;
ExpectedRewards(2,2) = 3.0;
ExpectedRewards(2,3) = 0.5*5 + 0.5*0.5;

localPlotQvalues(ExpectedRewards, "Expected Rewards")
```




Now, validate whether the DQN agent learns the optimal behavior.

If the state is 1, the optimal action is 3.

```
observation = 1;
getAction(DQNagent,observation)
```

```
ans = 1x1 cell array
    {[3]}
```

The agent selects the optimal action.

If the state is 2, the optimal action is 1.

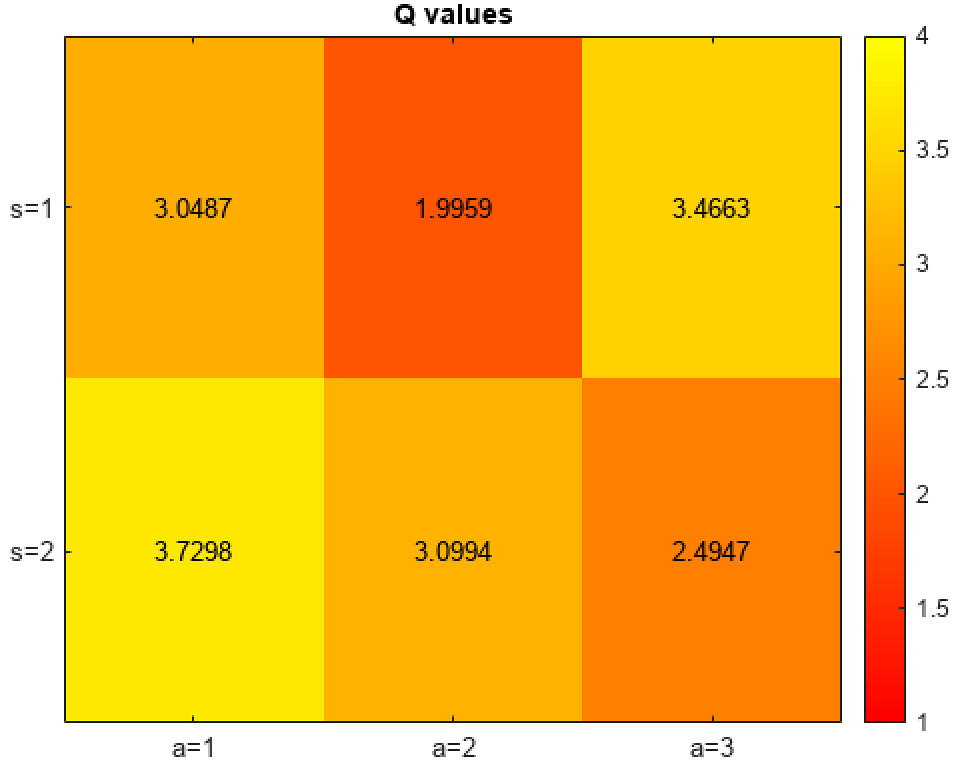
```
observation = 2;
getAction(DQNagent,observation)
```

```
ans = 1x1 cell array
    {[1]}
```

The agent selects the optimal action. Thus, the DQN agent has learned the optimal behavior.

Next, compare the Q-Value function to the true expected reward when selecting the optimal action.

```
% Get critic
figure(1)
DQNcritic = getCritic(DQNagent);
QValues = zeros(2,3);
for s = 1:2
    QValues(s,:) = getValue(DQNcritic, {s});
end

% Visualize Q values
localPlotQvalues(QValues, "Q values")
```




The learned Q-values are close to the true expected rewards computed above.

### Create Q-learning Agent

Next, train a Q-learning agent. To create a Q-learning agent, first create a table using the observation and action specifications from the environment.

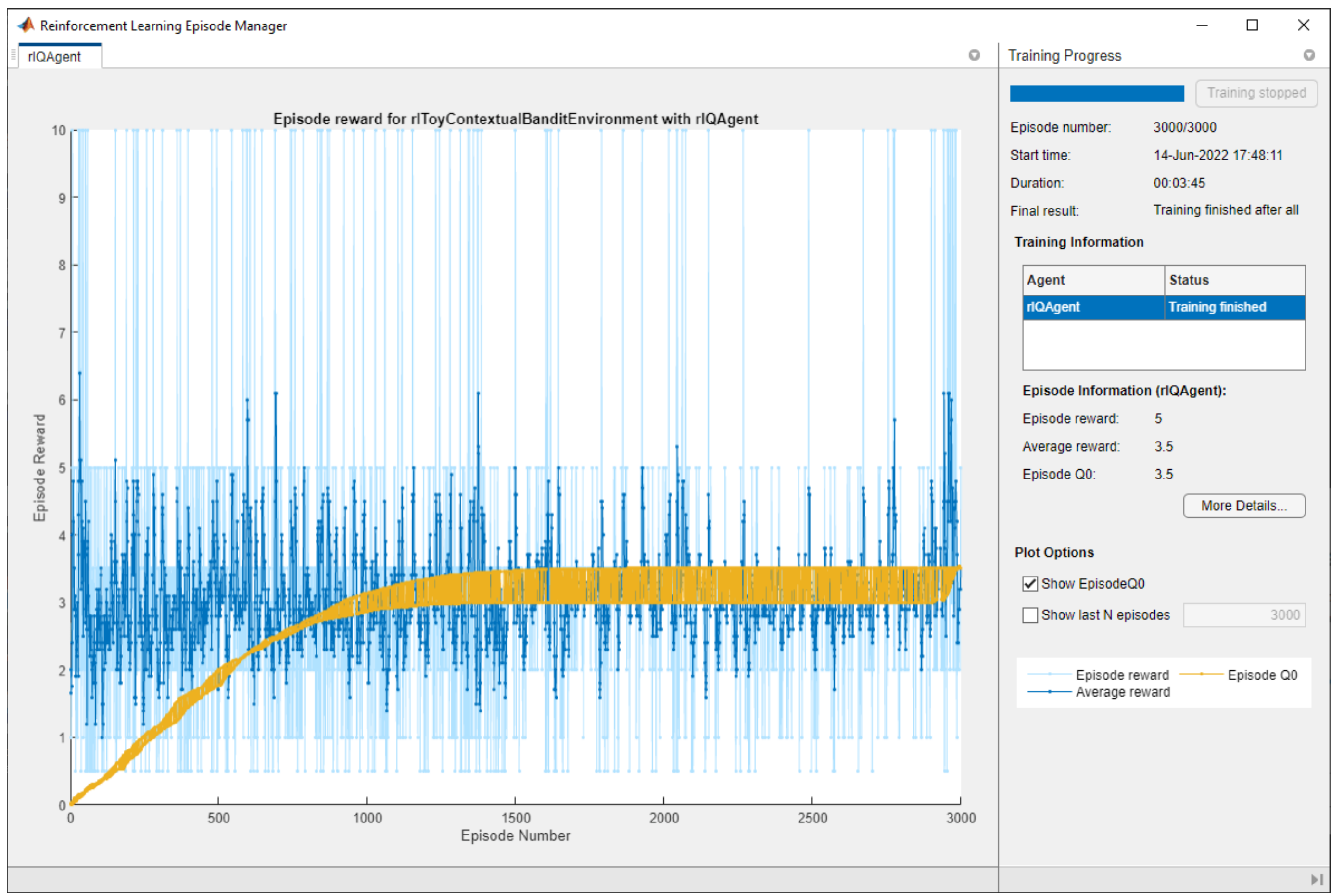
```
rng(1); % For reproducibility

qTable = rlTable(obsInfo, actInfo);
critic = rlQValueFunction(qTable, obsInfo, actInfo);

opt = rlQAgentOptions;
opt.EpsilonGreedyExploration.Epsilon = 1;
opt.EpsilonGreedyExploration.EpsilonDecay = 0.0005;

Qagent = rlQAgent(critic,opt);
```

### Train Q-Learning Agent

To save time while running this example, load a pre-trained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(Qagent,env,trainOpts);
else
```

```
    % Load the pre-trained agent for the example.
    load("ToyContextualBanditQAgent.mat","Qagent")
end
```




### Validate Q-Learning Agent

When the state is 1, the optimal action is 3.

```
observation = 1;
getAction(Qagent,observation)
```

```
ans = 1x1 cell array
    {[3]}
```

The agent selects the optimal action.

When the state is 2, the optimal action is 1.

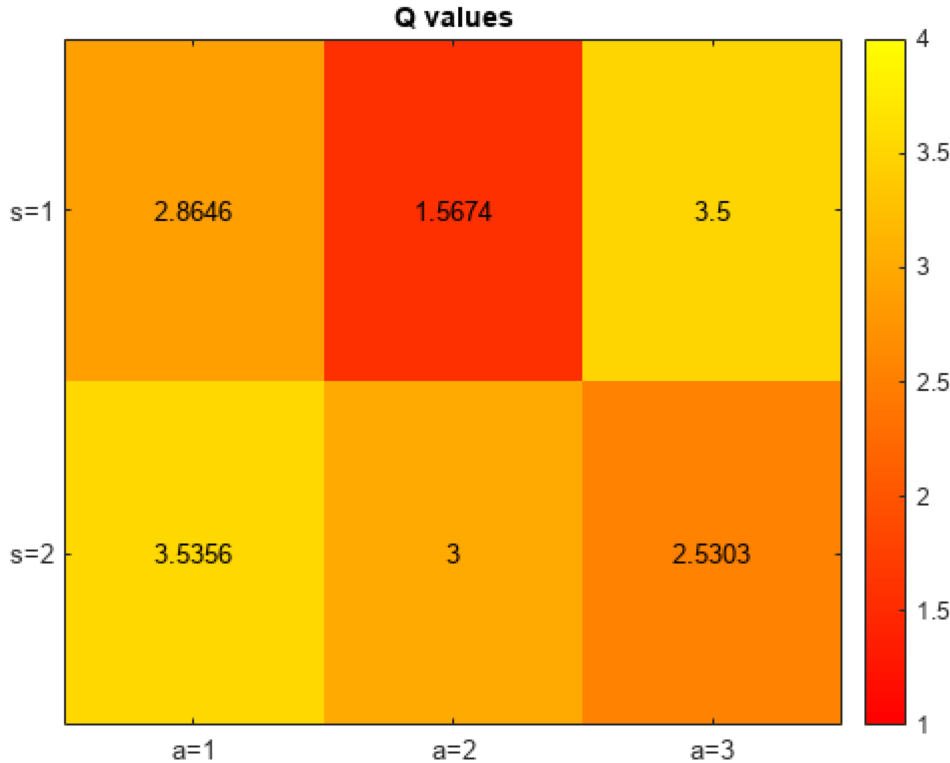
```
observation = 2;
getAction(Qagent,observation)
```

```
ans = 1x1 cell array
    {[1]}
```

The agent selects the optimal action. Hence, the Q-learning agent has learned the optimal behavior.

Next, compare the Q-Value function to the true expected reward when selecting the optimal action.

```
% Get critic
figure(2)
Qcritic = getCritic(Qagent);
QValues = zeros(2,3);
for s = 1:2
    for a = 1:3
        QValues(s,a) = getValue(Qcritic, {s}, {a});
    end
end

% Visualize Q values
localPlotQvalues(QValues, "Q values")
```




Again, the learned Q-values are close to the true expected rewards. The Q-values for deterministic rewards, Q(s=1, a=3) and Q(s=2, a=2), are the same as the true expected rewards. Note that the corresponding Q-values learned by the DQN network, while close, are not identical to the true values. This happens because the DQN uses a neural network instead of a table as the internal function approximator.

### Local Function

```
function localPlotQvalues(QValues, titleText)
    % Visualize Q values
```

```
    figure;
    imagesc(QValues,[1,4])
    colormap("autumn")
    title(titleText)
    colorbar
    set(gca,"Xtick",1:3,"XTickLabel",{"a=1", "a=2", "a=3"})
    set(gca,"Ytick",1:2,"YTickLabel",{"s=1", "s=2"})

    % Plot values on the image
    x = repmat(1:size(QValues,2), size(QValues,1), 1);
    y = repmat(1:size(QValues,1), size(QValues,2), 1)';
    QValuesStr = num2cell(QValues);
    QValuesStr = cellfun(@num2str, QValuesStr, UniformOutput=false);
    text(x(:), y(:), QValuesStr, HorizontalAlignment = "Center")
end
```

**Reference**

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press, 2018.

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim`

**Objects**
`rlDQNAgent` | `rlDQNAgentOptions` | `rlTrainingOptions`

## Related Examples

- “Train DQN Agent to Balance Cart-Pole System” on page 5-50

## More About

- “Deep Q-Network (DQN) Agents” on page 3-23
- “Load Predefined Grid World Environments” on page 2-17
- “Train Reinforcement Learning Agents” on page 5-3
- “Create Policies and Value Functions” on page 4-2

# Train Agent or Tune Environment Parameters Using Parameter Sweeping

This example shows how to train a reinforcement learning agent with the water tank reinforcement learning Simulink® environment by sweeping parameters. You can use this example as a template for tuning parameters when training reinforcement learning agents.

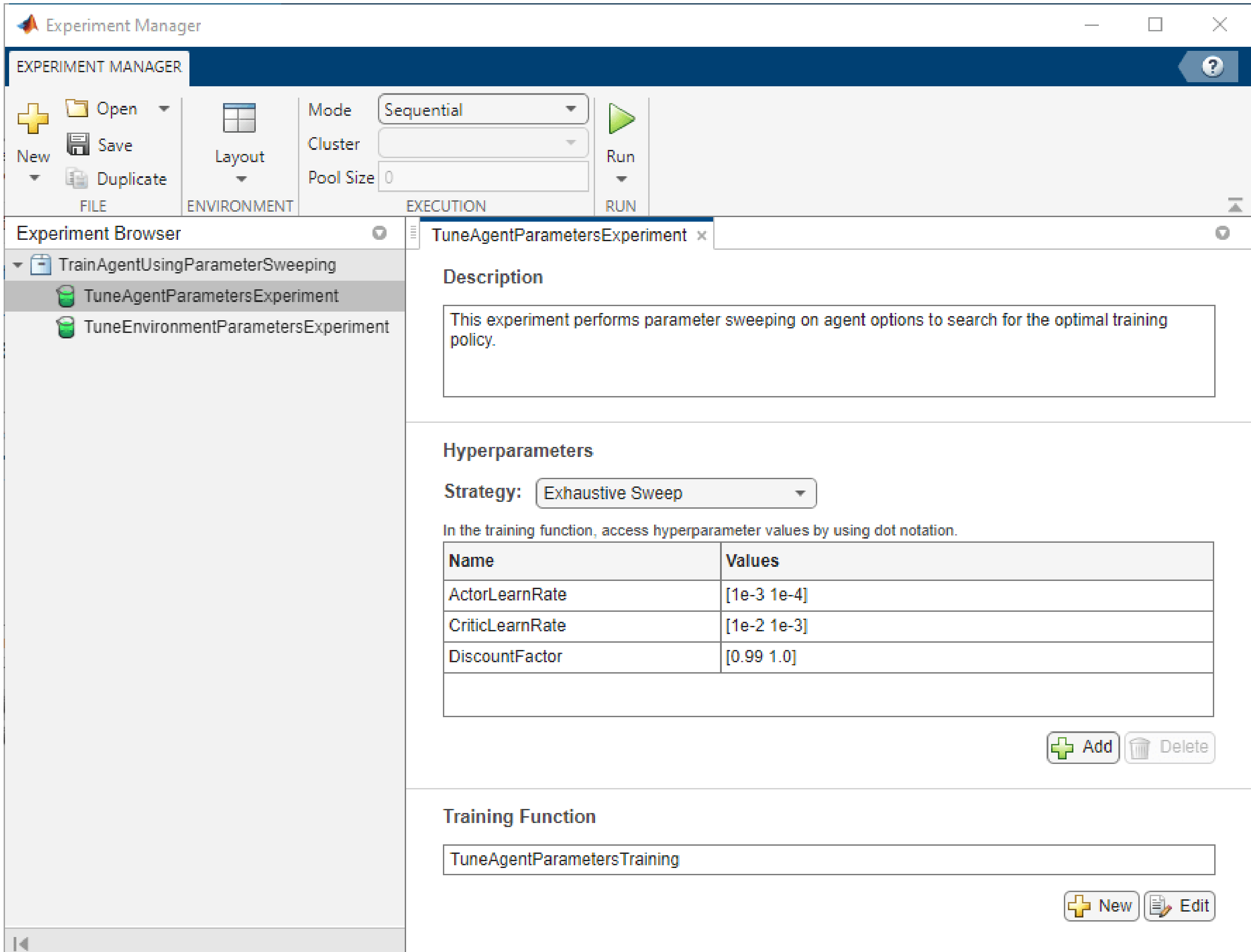
Open a preconfigured project, which has all required files added as project dependencies. Opening the project also launches the Experiment Manager app.

```
TrainAgentUsingParameterSweepingStart
```

Note that it is best practice to add any Simulink models and supporting files as dependencies to your project.

### Tune Agent Parameters Using Parameter Sweeping

In this section you tune the agent parameters to search for an optimal training policy.



#### Open Experiment

- In the **Experiment Browser** pane double-click the name of the experiment (`TuneAgentParametersExperiment`). This opens a tab for the experiment.
- The **Hyperparameters** section contains the hyperparameters to tune for this experiment. A set of hyperparameters has been added for this experiment. To add a new parameter, click **Add** and specify a name and array of values for the hyperparameter. When you run the experiment, Experiment Manager runs the training using every combination of parameter values specified in the hyperparameter table.
- Verify that **Strategy** is set to `Exhaustive Sweep`.
- Under **Training Function**, click **Edit.** The MATLAB Editor opens to show code for the training function `TuneAgentParametersTraining`. The training function creates the environment and agent objects and runs the training using one combination of the specified hyperparameters.

```
function output = TuneAgentParametersTraining(params,monitor)

% Set the random seed generator
rng(0);

% Load the Simulink model
mdl = "rlwatertank";
load_system(mdl);

% Create variables in the base workspace. When running on a parallel
% worker this will also create variables in worker's base workspace.
evalin("base", "loadWaterTankParams");
Ts = evalin("base","Ts");
Tf = evalin("base","Tf");

% Create a reinforcement learning environment
actionInfo = rlNumericSpec([1 1]);
observationInfo = rlNumericSpec([3 1],...
    LowerLimit=[-inf -inf 0  ]',...
    UpperLimit=[ inf  inf inf]');
blk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl, blk, observationInfo, actionInfo);

% Specify a reset function for the environment
env.ResetFcn = @localResetFcn;

% Create options for the reinforcement learning agent. You can assign
% values from the params structure for sweeping parameters.
agentOpts = rlDDPGAgentOptions();
agentOpts.MiniBatchSize = 64;
agentOpts.TargetSmoothFactor = 1e-3;
agentOpts.SampleTime = Ts;
agentOpts.DiscountFactor = params.DiscountFactor;
agentOpts.ActorOptimizerOptions.LearnRate = params.ActorLearnRate;
agentOpts.CriticOptimizerOptions.LearnRate = params.CriticLearnRate;
agentOpts.ActorOptimizerOptions.GradientThreshold = 1;
agentOpts.CriticOptimizerOptions.GradientThreshold = 1;
agentOpts.NoiseOptions.Variance = 0.3;
agentOpts.NoiseOptions.VarianceDecayRate = 1e-5;

% Create the reinforcement learning agent. You can modify the
% localCreateActorAndCritic function to edit the agent model.
```

```
[actor, critic] = localCreateActorAndCritic(observationInfo, actionInfo);
agent = rlDDPGAgent(actor, critic, agentOpts);

maxepisodes = 200;
maxsteps = ceil(Tf/Ts);
trainOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes, ...
    MaxStepsPerEpisode=maxsteps, ...
    ScoreAveragingWindowLength=20, ...
    Verbose=false, ...
    Plots="none",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=800);

% Create a data logger for logging data to the monitor object
logger = rlDataLogger(monitor);

% Run the training
result = train(agent, env, trainOpts, Logger=logger);

% Export experiment results
output.Agent = agent;
output.Environment = env;
output.TrainingResult = result;
output.Parameters = params;

end
```

**Run Experiment**

When you run the experiment, Experiment Manager executes the training function multiple times. Each trial uses one combination of hyperparameter values. By default, Experiment Manager runs one trial at a time. If you have the Parallel Computing Toolbox ™, you can run multiple trials at the same time or offload your experiment as a batch job in a cluster.

Under **Mode**, select `Sequential`, and click **Run** to run the experiment one trial at a time.

- To run multiple trials simultaneously, under **Mode**, select `Simultaneous`, and click **Run**. This requires a Parallel Computing Toolbox license.
- To offload the experiment as a batch job under **Mode**, select `Batch Sequential` or `Batch Simultaneous`, specify your **Cluster** and **Pool Size**, and click **Run**. Note that you will need to configure the cluster with the files necessary for this example. This mode also requires a Parallel Computing Toolbox license.

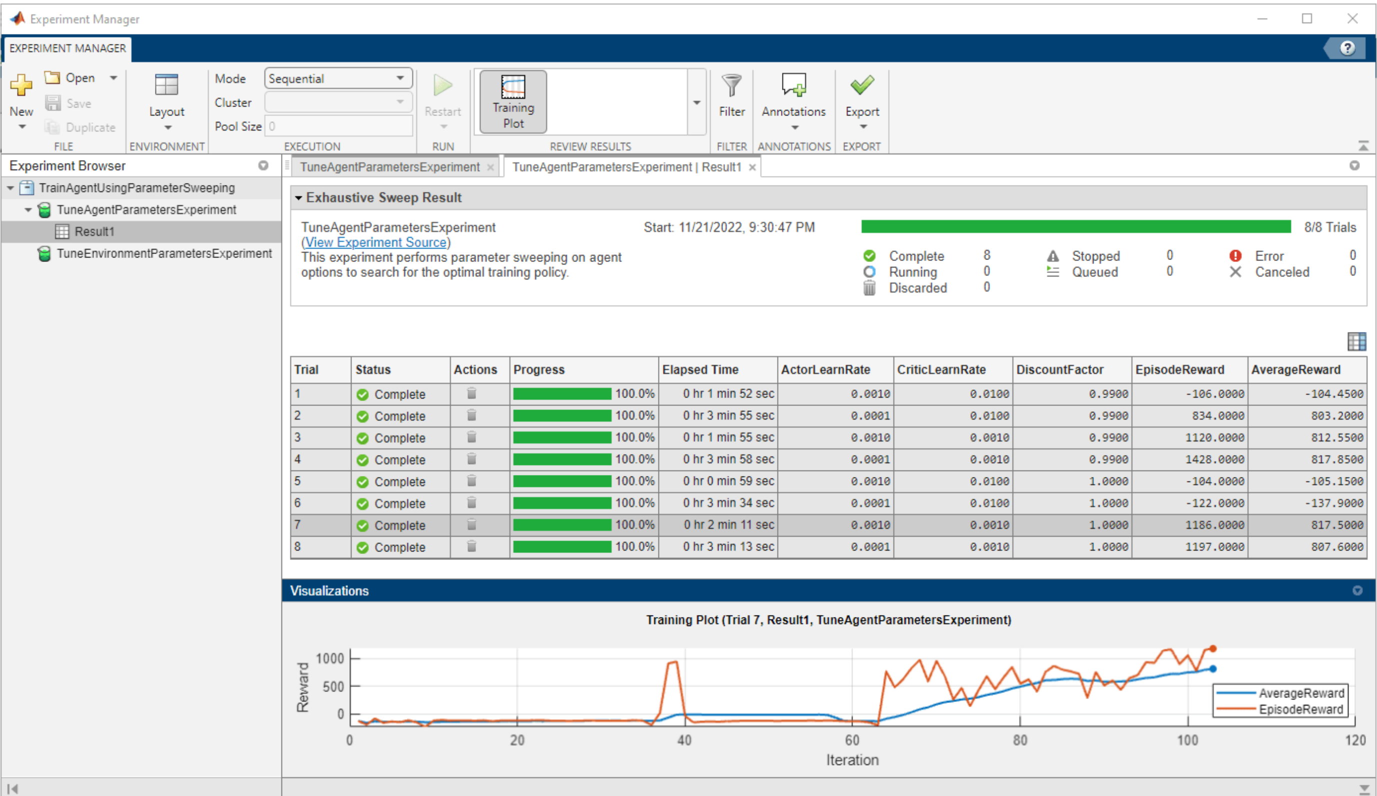
Note that your cluster needs to be configured with files necessary for this experiment when running in the `Batch Sequential` or `Batch Simultaneous` modes. For more information on the Cluster Profile Manager, see "Discover Clusters and Use Cluster Profiles" (Parallel Computing Toolbox). To configure your cluster:

- Open the Cluster Profile Manager and under **Properties**, click **Edit**.
- Under the `AttachedFiles` option, click **Add** and specify the files `rlwatertank.slx` and `loadWaterTankParams.m`.
- Click **Done**.

When the experiment is running, select a trial row from the table of results, and under the toolstrip, click **Training Plot**. This shows the episode and average reward plots for that trial.

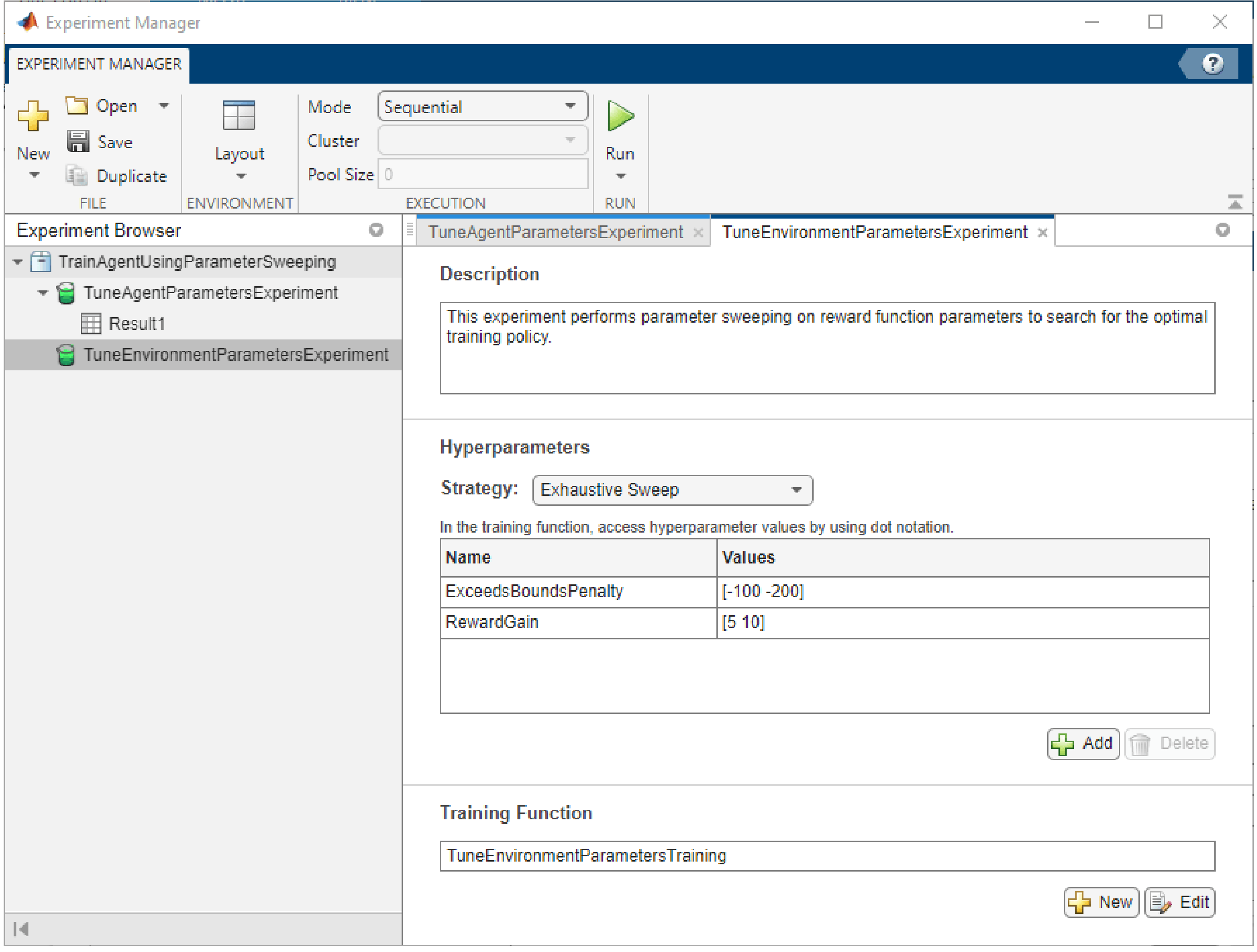
After the experiment is finished:

- Select the row corresponding to trial 7 which has the average reward `817.5`, and under the toolstrip, click **Export**. This action exports the results of the trial to a base workspace variable.
- Name the variable `agentParamSweepTrainingOutput`.




## Tune Environment Parameters Using Parameter Sweeping

In this section you tune the environment's reward function parameters to search for an optimal training policy.




### Open Experiment

In the **Experiment Browser** pane, open `TuneEnvironmentParametersExperiment`. Verify, as with the agent tuning, that **Strategy** is set to `Exhaustive Sweep`. View code for the training function `TuneEnvironmentParametersTraining` as before.

```
function output = TuneEnvironmentParametersTraining(params,monitor)

% Set the random seed generator
rng(0);

% Load the Simulink model
mdl = "rlwatertank";
load_system(mdl);

% Create variables in the base workspace. When running on a parallel
% worker this will also create variables in the worker's base workspace.
evalin("base", "loadWaterTankParams");
Ts = evalin("base","Ts");
Tf = evalin("base","Tf");
```

```
% Create a reinforcement learning environment
actionInfo = rlNumericSpec([1 1]);
observationInfo = rlNumericSpec([3 1],...
    LowerLimit=[-inf -inf 0  ]',...
    UpperLimit=[ inf  inf inf]');
blk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl, blk, observationInfo, actionInfo);

% Specify a reset function for the environment. You can tune environment
% parameters such as reward or initial condition within this function.
env.ResetFcn = @(in) localResetFcn(in, params);

% Create options for the reinforcement learning agent. You can assign
% values from the params structure for sweeping parameters.
agentOpts = rlDDPGAgentOptions();
agentOpts.MiniBatchSize                             = 64;
agentOpts.TargetSmoothFactor                        = 1e-3;
agentOpts.SampleTime                                = Ts;
agentOpts.DiscountFactor                            = 0.99;
agentOpts.ActorOptimizerOptions.LearnRate           = 1e-3;
agentOpts.CriticOptimizerOptions.LearnRate          = 1e-3;
agentOpts.ActorOptimizerOptions.GradientThreshold   = 1;
agentOpts.CriticOptimizerOptions.GradientThreshold  = 1;
agentOpts.NoiseOptions.Variance                     = 0.3;
agentOpts.NoiseOptions.VarianceDecayRate            = 1e-5;

% Create the reinforcement learning agent. You can modify the
% localCreateActorAndCritic function to edit the agent model.
[actor, critic] = localCreateActorAndCritic(observationInfo, actionInfo);
agent = rlDDPGAgent(actor, critic, agentOpts);

maxepisodes = 200;
maxsteps = ceil(Tf/Ts);
trainOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes, ...
    MaxStepsPerEpisode=maxsteps, ...
    ScoreAveragingWindowLength=20, ...
    Verbose=false, ...
    Plots="none",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=800);

% Create a data logger for logging data to the monitor object
logger = rlDataLogger(monitor);

% Run the training
result = train(agent, env, trainOpts, Logger=logger);

% Export experiment results
output.Agent = agent;
output.Environment = env;
output.TrainingResult = result;
output.Parameters = params;

end

%% Environment reset function
```

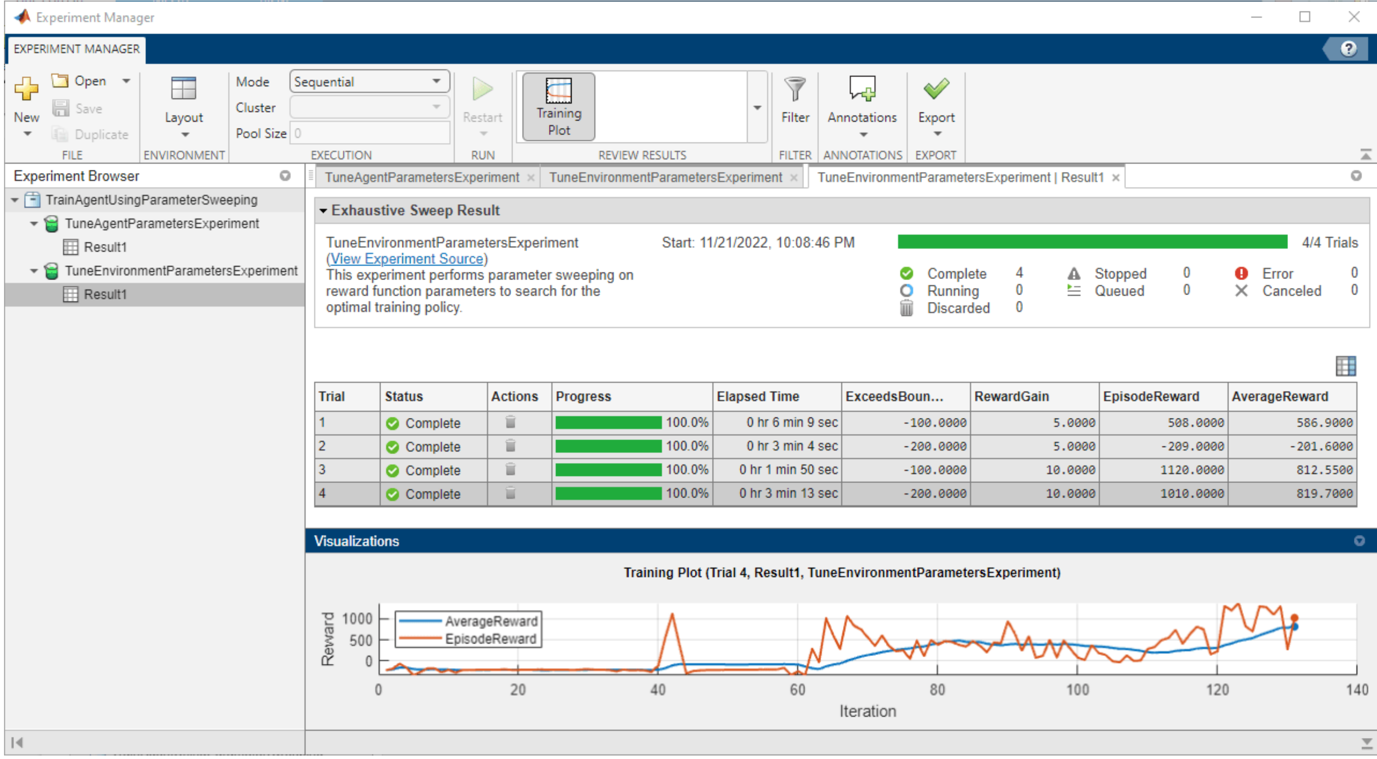
```
function in = localResetFcn(in, params)

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

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

% Tune the reward parameters
in = setBlockParameter(in, ...
       "rlwatertank/calculate reward/Gain", ...
       "Gain",num2str(params.RewardGain));
in = setBlockParameter(in, ...
       "rlwatertank/calculate reward/Gain2", ...
       "Gain",num2str(params.ExceedsBoundsPenalty));

end
```

#### Run Experiment

Run the experiment using the same settings you use for agent tuning.

After the experiment is finished:

- Select the row corresponding to trial 4, which has the maximum average reward, and export the result to a base workspace variable.
- Name the variable as `envParamSweepTrainingOutput`.

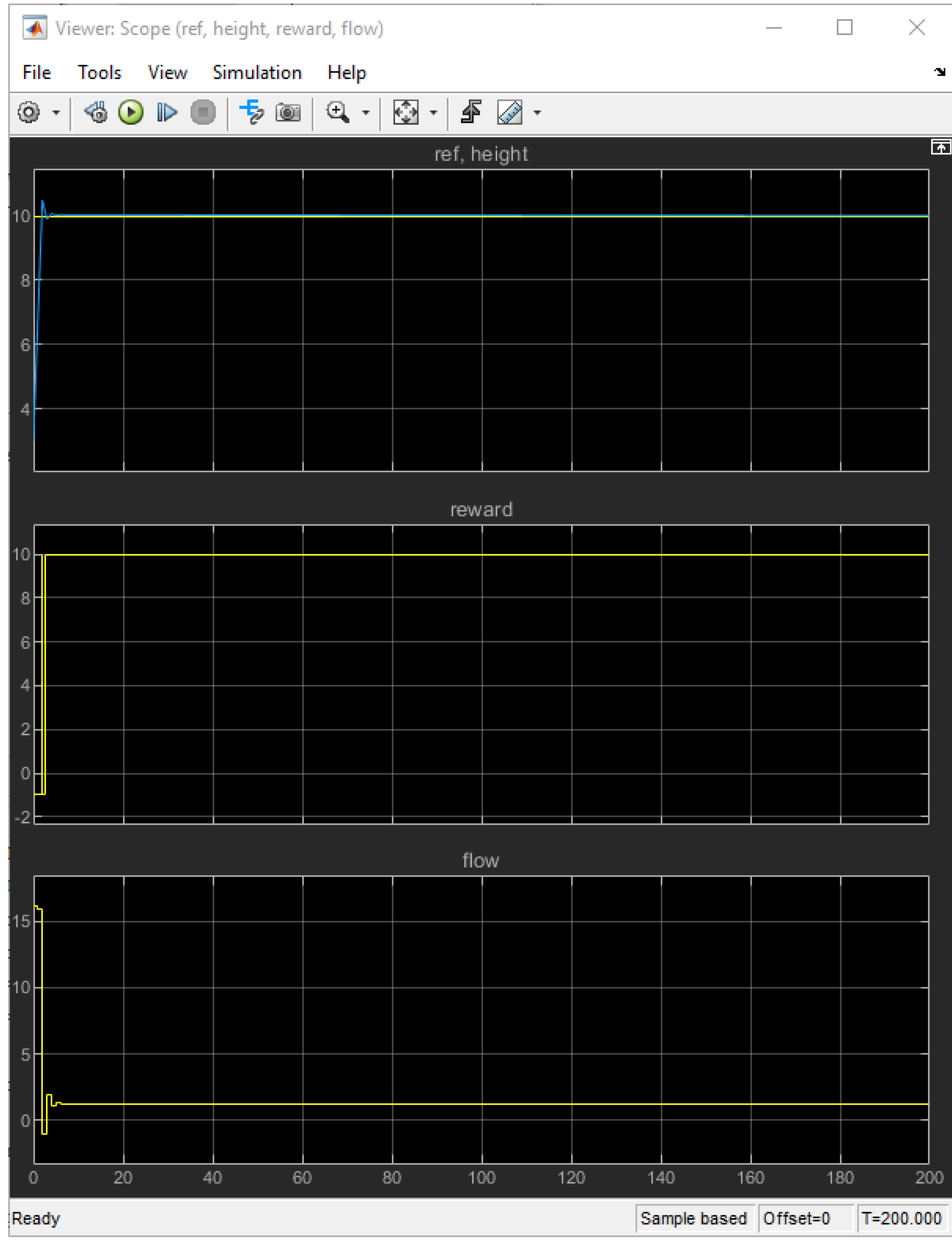



## Evaluate Agent Performance

Execute the following code in MATLAB after exporting the agents from the above experiments. This code simulates the agent with the environment and displays the performance in the Scope blocks.

```
open_system("rlwatertank");
simOpts = rlSimulationOptions(MaxSteps=200);

% evaluate the agent exported from
% TuneAgentParametersExperiment
experience = sim(agentParamSweepTrainingOutput.Agent, ...
                 agentParamSweepTrainingOutput.Environment, ...
                 simOpts);

% evaluate the agent exported from
% TuneEnvironmentParametersExperiment
experience = sim(envParamSweepTrainingOutput.Agent, ...
                 envParamSweepTrainingOutput.Environment,
                 simOpts);
```

The agent is able to track the desired water level.




Close the project.

```
close(prj);
```

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim`

**Objects**
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlTrainingOptions`

## Related Examples

- “Log Training Data to Disk” on page 5-24

## More About

- “Train Reinforcement Learning Agents” on page 5-3
- “Water Tank Reinforcement Learning Environment Model” on page 2-55
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40

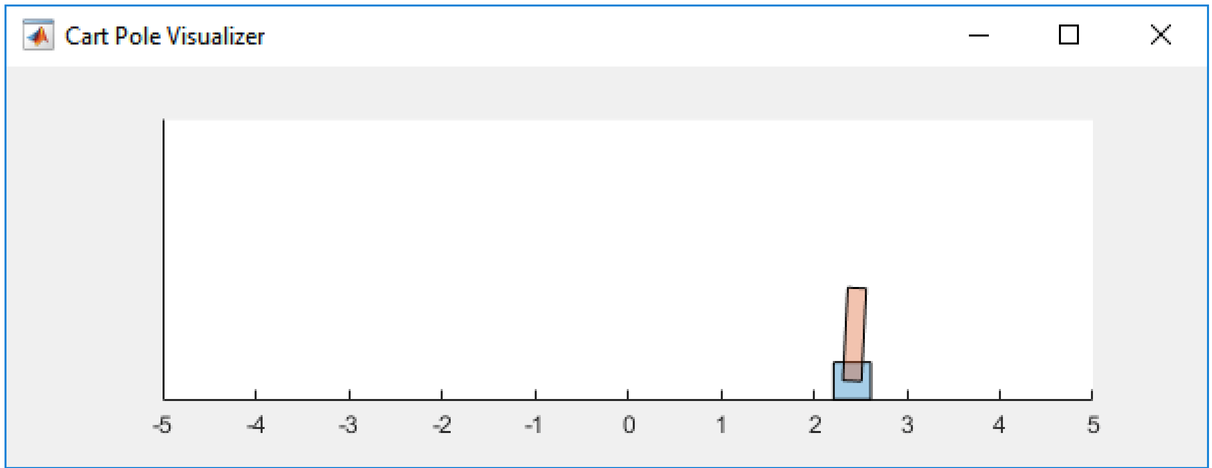
# Train DQN Agent to Balance Cart-Pole System

This example shows how to train a deep Q-learning network (DQN) agent to balance a cart-pole system modeled in MATLAB®.

For more information on DQN agents, see “Deep Q-Network (DQN) Agents” on page 3-23. For an example that trains a DQN agent in Simulink®, see “Train DQN Agent to Swing Up and Balance Pendulum” on page 5-89.

### Cart-Pole MATLAB Environment

The reinforcement learning environment for this example is a pole attached to an unactuated joint on a cart, which moves along a frictionless track. The training goal is to make the pole stand upright without falling over.




For this environment:

- The upward balanced pole position is 0 radians, and the downward hanging position is `pi` radians.
- The pole starts upright with an initial angle between –0.05 and 0.05 radians.
- The force action signal from the agent to the environment is either –10 or 10 N.
- The observations from the environment are the position and velocity of the cart, the pole angle, and the pole angle derivative.
- The episode terminates if the pole is more than 12 degrees from vertical or if the cart moves more than 2.4 m from the original position.
- A reward of +1 is provided for every time step that the pole remains upright. A penalty of –5 is applied when the pole falls.

For more information on this model, see “Load Predefined Control System Environments” on page 2-23.

### Create Environment Interface

Create a predefined environment interface for the system.

```
env = rlPredefinedEnv("CartPole-Discrete")
```

```
env = 
  CartPoleDiscreteAction with properties:

                  Gravity: 9.8000
                 MassCart: 1
                 MassPole: 0.1000
                   Length: 0.5000
                 MaxForce: 10
                       Ts: 0.0200
    ThetaThresholdRadians: 0.2094
               XThreshold: 2.4000
      RewardForNotFalling: 1
        PenaltyForFalling: -5
                    State: [4x1 double]
```

The interface has a discrete action space where the agent can apply one of two possible force values to the cart, –10 or 10 N.

Get the observation and action specification information.

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

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "CartPole States"
    Description: "x, dx, theta, dtheta"
      Dimension: [4 1]
       DataType: "double"
```

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

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [-10 10]
           Name: "CartPole Action"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create DQN Agent

DQN agents can use vector Q-value functions critics, which are generally more efficient than comparable single-output critics. A vector Q-value function critic has observations as inputs and state-action values as outputs. Each output element represents the expected cumulative long-term reward for taking the corresponding discrete action from the state indicated by the observation inputs. For more information on creating value-functions, see "Create Policies and Value Functions" on page 4-2.

To approximate the Q-value function within the critic, use a neural network with one input channel (the 4-dimensional observed state vector) and one output channel with two elements (one for the 10 N action, another for the –10 N action). Define the network as an array of layer objects, and get the dimension of the observation space and the number of possible actions from the environment specification objects.

```
net = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(20)
    reluLayer
    fullyConnectedLayer(length(actInfo.Elements))
    ];
```

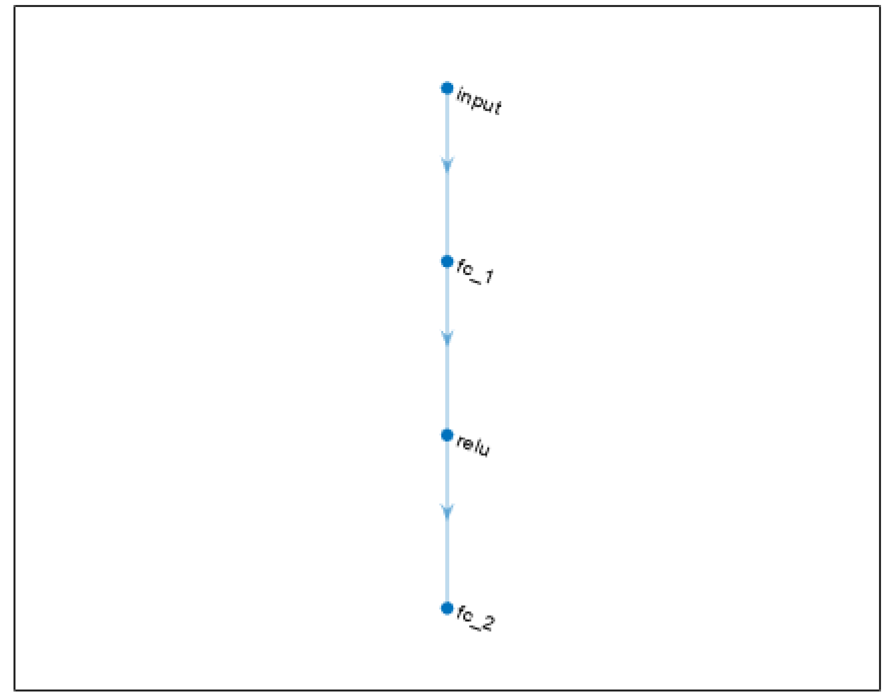
Convert to `dlnetwork` and display the number of weights.

```
net = dlnetwork(net);
summary(net)
```

```
   Initialized: true

   Number of learnables: 142

   Inputs:
      1   'input'   4 features
```

View the network configuration.

```
plot(net)
```



Create the critic approximator using `net` and the environment specifications. For more information, see `rlVectorQValueFunction`.

```
critic = rlVectorQValueFunction(net,obsInfo,actInfo);
```

Check the critic with a random observation input.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = 2x1 single column vector

   -0.2257
    0.4299
```

Create the DQN agent using `critic`. For more information, see `rlDQNAgent`.

```
agent = rlDQNAgent(critic);
```

Check the agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[10]}
```

Specify the DQN agent options, including training options for the critic. Alternatively, you can use `rlDQNAgentOptions` and `rlOptimizerOptions` objects.

```
agent.AgentOptions.UseDoubleDQN = false;
agent.AgentOptions.TargetSmoothFactor = 1;
agent.AgentOptions.TargetUpdateFrequency = 4;
agent.AgentOptions.ExperienceBufferLength = 1e5;
agent.AgentOptions.MiniBatchSize = 256;
agent.AgentOptions.CriticOptimizerOptions.LearnRate = 1e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
```

### Train Agent

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

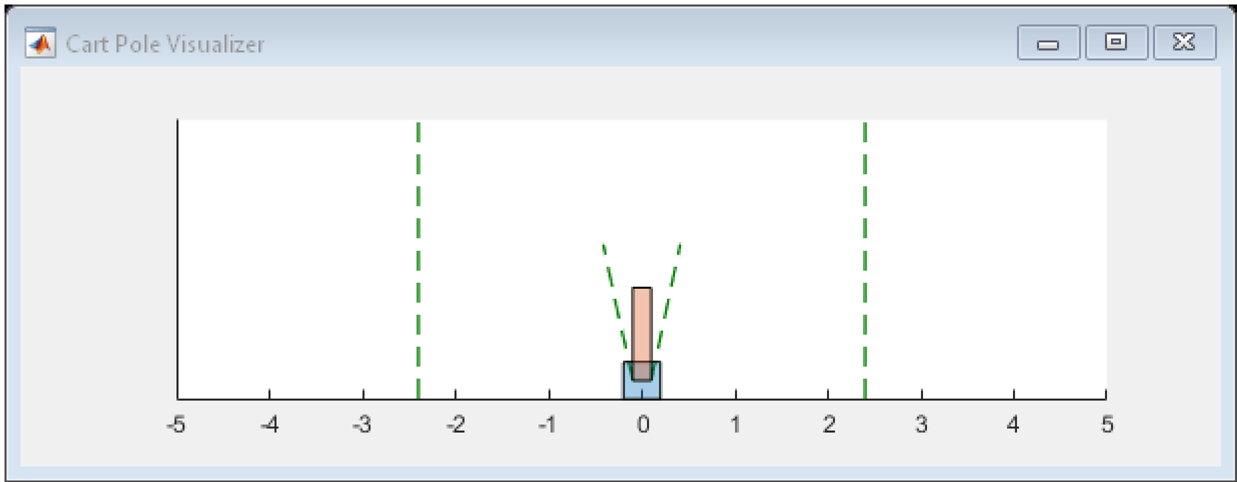
- Run one training session containing at most 1000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an moving average cumulative reward greater than 480. At this point, the agent can balance the cart-pole system in the upright position.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=1000, ...
    MaxStepsPerEpisode=500, ...
    Verbose=false, ...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=480);
```

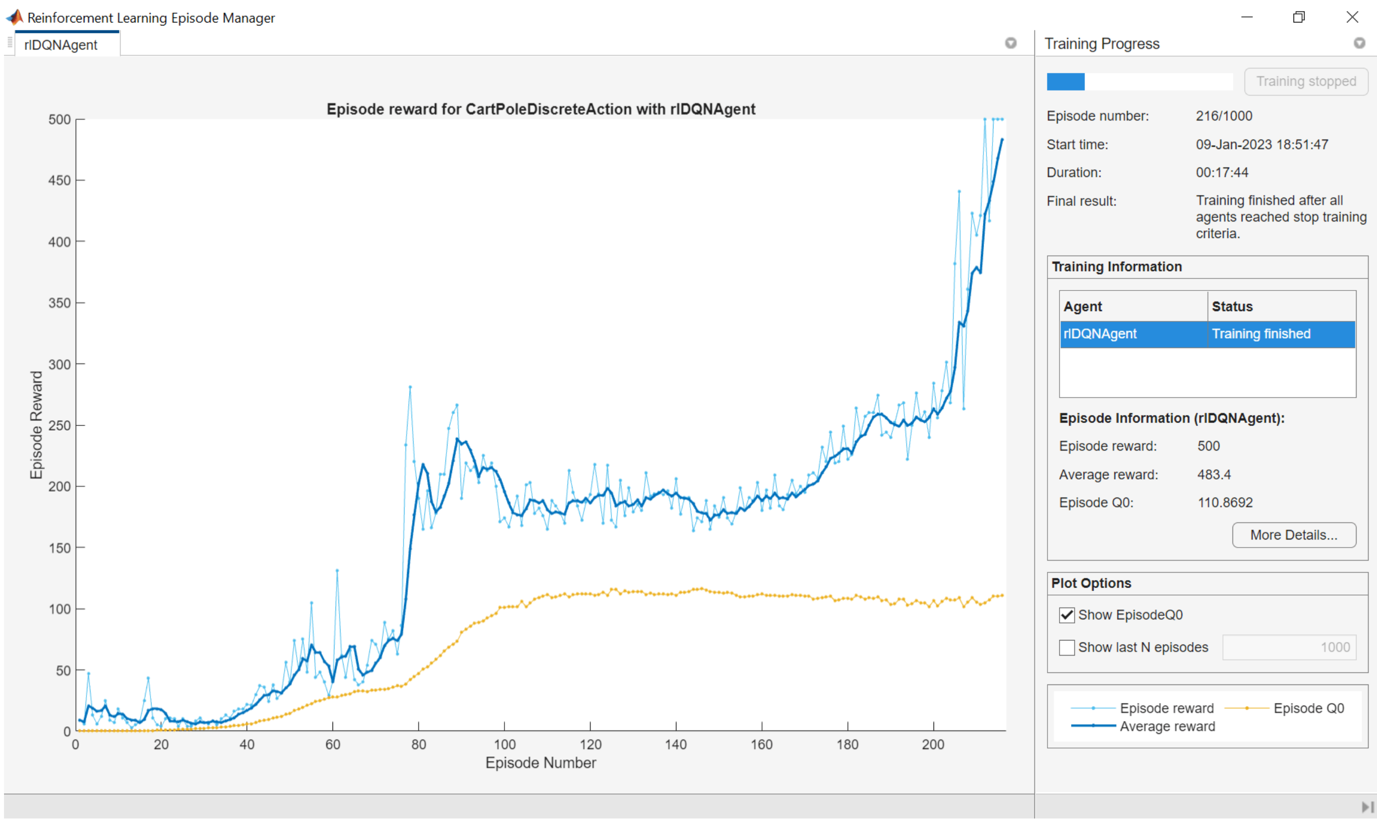
You can visualize the cart-pole system by using the `plot` function during training or simulation.

```
plot(env)
```




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

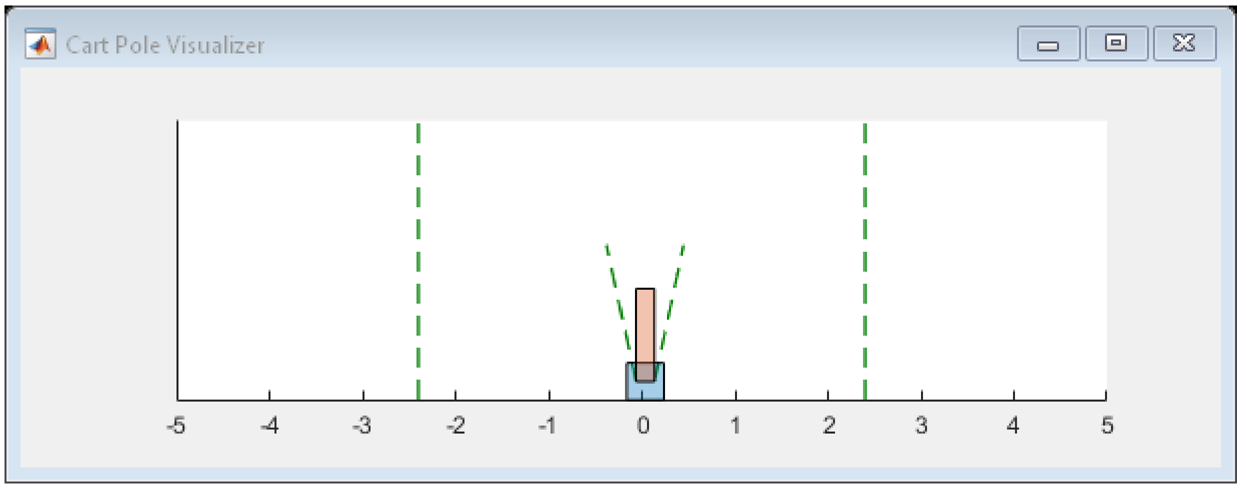
```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load("MATLABCartpoleDQNMulti.mat","agent")
end
```




## Simulate DQN Agent

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`. The agent can balance the cart-pole even when the simulation time increases to 500 steps.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




```
totalReward = sum(experience.Reward)
```

```
totalReward = 500
```

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim`

**Objects**
`rlDQNAgent` | `rlDQNAgentOptions` | `rlTrainingOptions`

## Related Examples


- "Train DQN Agent to Swing Up and Balance Pendulum" on page 5-89
- "Train PG Agent to Balance Cart-Pole System" on page 5-57
- "Train Reinforcement Learning Agents" on page 5-3


## More About


- "Load Predefined Control System Environments" on page 2-23
- "Create Policies and Value Functions" on page 4-2
- "Deep Q-Network (DQN) Agents" on page 3-23
- "Train Reinforcement Learning Agents" on page 5-3

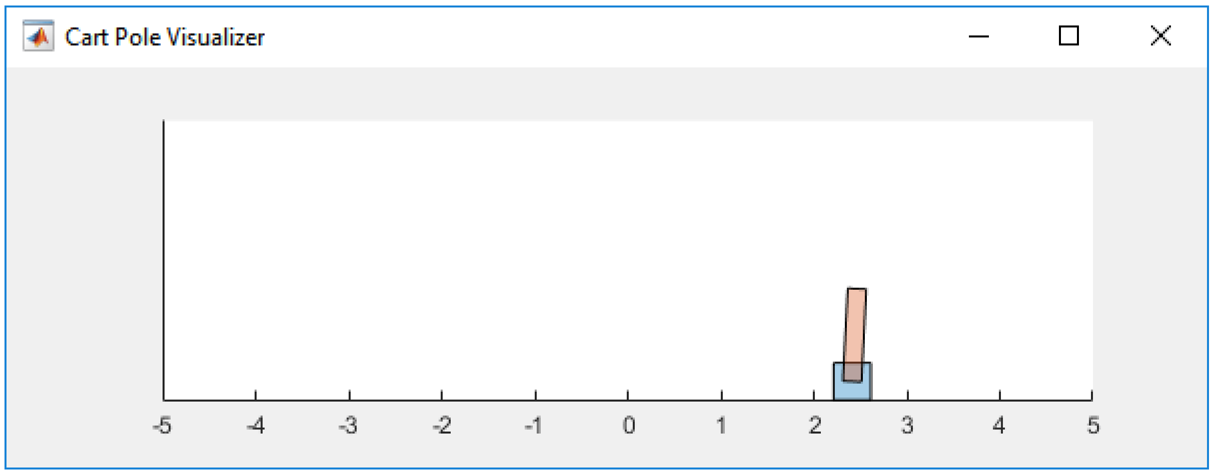
# Train PG Agent to Balance Cart-Pole System

This example shows how to train a policy gradient (PG) agent to balance a cart-pole system modeled in MATLAB®. For more information on PG agents, see “Policy Gradient (PG) Agents” on page 3-27.

For an example that trains a PG agent with a baseline, see “Train PG Agent with Baseline to Control Double Integrator System” on page 5-70.

### Cart-Pole MATLAB Environment

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




For this environment:

- The upward balanced pendulum position is `0` radians, and the downward hanging position is `pi` radians.
- The pendulum starts upright with an initial angle between –0.05 and 0.05 radians.
- The force action signal from the agent to the environment is either –10 or 10 N.
- The observations from the environment are the position and velocity of the cart, the pendulum angle, and the pendulum angle derivative.
- The episode terminates if the pole is more than 12 degrees from vertical or if the cart moves more than 2.4 m from the original position.
- A reward of +1 is provided for every time step that the pole remains upright. A penalty of –5 is applied when the pendulum falls.

For more information on this model, see “Load Predefined Control System Environments” on page 2-23.

### Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv("CartPole-Discrete")
```

```
env = 
  CartPoleDiscreteAction with properties:

                  Gravity: 9.8000
                 MassCart: 1
                 MassPole: 0.1000
                   Length: 0.5000
                 MaxForce: 10
                       Ts: 0.0200
    ThetaThresholdRadians: 0.2094
               XThreshold: 2.4000
      RewardForNotFalling: 1
        PenaltyForFalling: -5
                    State: [4x1 double]
```

The interface has a discrete action space where the agent can apply one of two possible force values to the cart, –10 or 10 N.

Obtain the observation and action information from the environment interface.

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create PG Agent

For policy gradient agents, the actor executes a stochastic policy, which for discrete action spaces is approximated by a discrete categorical actor. This actor must take the observation signal as input and return a probability for each action.

To approximate the policy within the actor, use a deep neural network. Define the network as an array of layer objects, and get the dimension of the observation space and the number of possible actions from the environment specification objects. For more information on creating a deep neural network policy representation, see “Create Policies and Value Functions” on page 4-2.

```
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(10)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    softmaxLayer
    ];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 72

   Inputs:
      1   'input'   4 features
```

Create the actor representation using the specified deep neural network and the environment specification objects. For more information, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

To return the probability distribution of the possible actions as a function of a random observation, and given the current network weights, use `evaluate`.

```
prb = evaluate(actor,{rand(obsInfo.Dimension)});
prb{1}
```

```
ans = 2x1 single column vector

    0.7229
    0.2771
```

Create the agent using the actor. For more information, see `rlPGAgent`.

```
agent = rlPGAgent(actor);
```

Check the agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-10]}
```

Specify training options for the actor. Alternatively, you can use `rlPGAgentOptions` and `rlOptimizerOptions` objects.

```
agent.AgentOptions.CriticOptimizerOptions.LearnRate = 5e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
```

### Train Agent

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

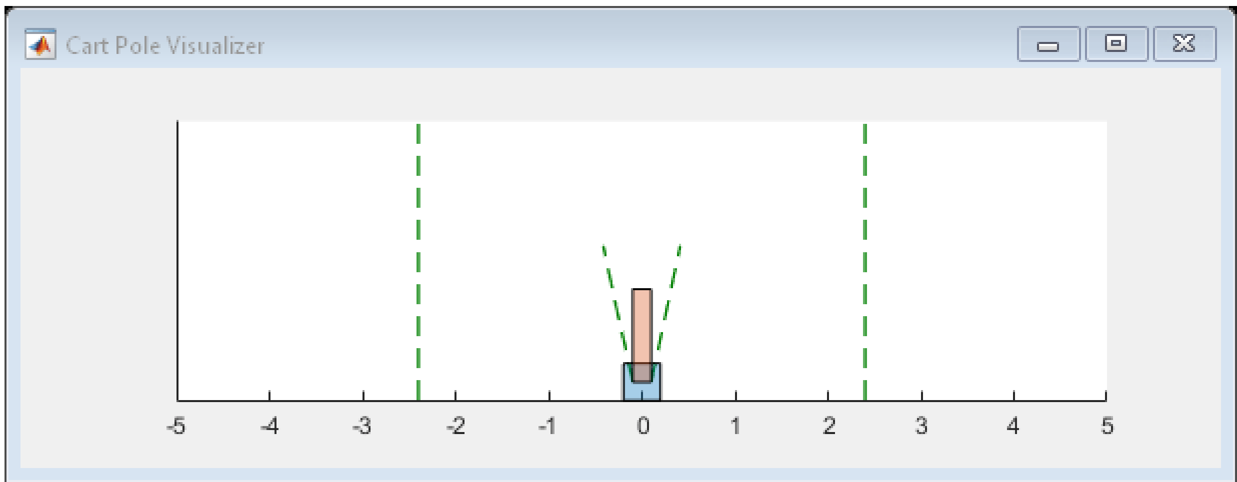
- Run each training episode for at most 1000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than 480 over 100 consecutive episodes. At this point, the agent can balance the pendulum in the upright position.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=1000, ...
    MaxStepsPerEpisode=500, ...
    Verbose=false, ...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=480,...
    ScoreAveragingWindowLength=100);
```

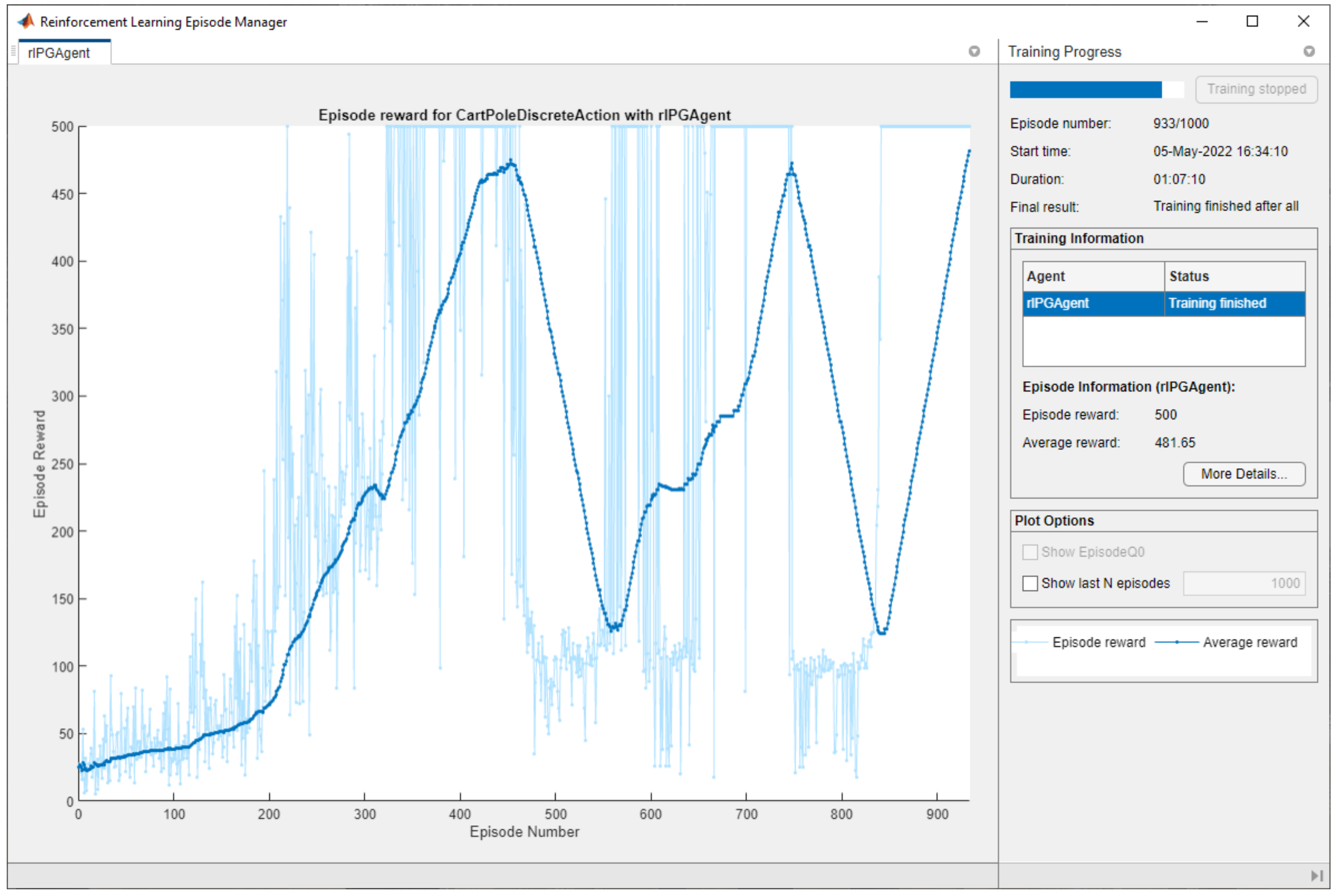
You can visualize the cart-pole system by using the `plot` function during training or simulation.

```
plot(env)
```




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

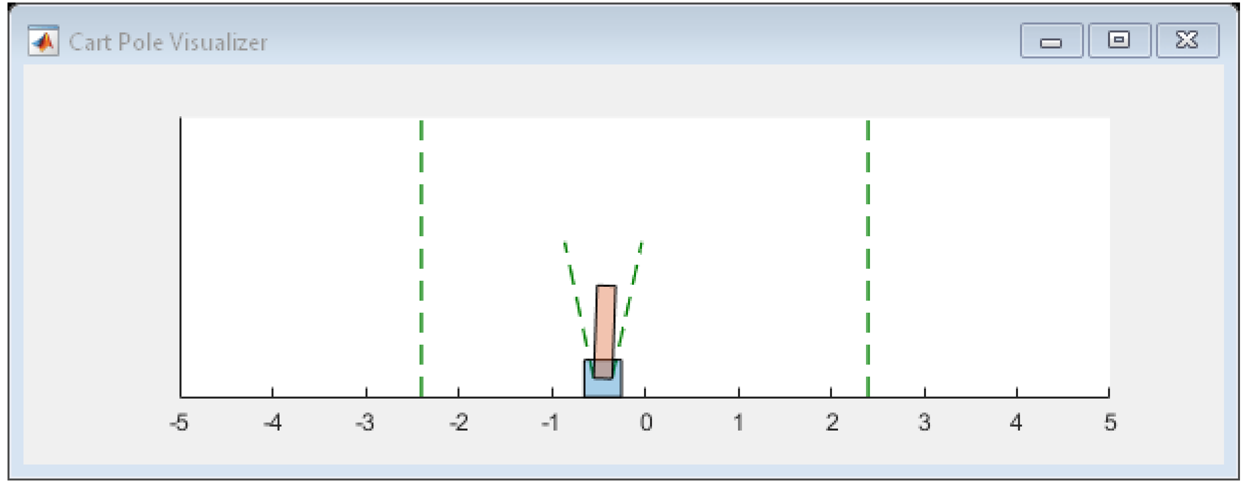
```
doTraining = false;

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




### Simulate PG Agent

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`. The agent can balance the cart-pole system even when the simulation time increases to 500 steps.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




```
totalReward = sum(experience.Reward)
```

```
totalReward = 500
```

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim`

**Objects**
`rlTrainingOptions` | `rlPGAgent` | `rlPGAgentOptions`

## Related Examples

- “Train AC Agent to Balance Cart-Pole System” on page 5-63
- “Train PG Agent with Baseline to Control Double Integrator System” on page 5-70
- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “Load Predefined Control System Environments” on page 2-23
- “Policy Gradient (PG) Agents” on page 3-27
- “Create Policies and Value Functions” on page 4-2
- “Train Reinforcement Learning Agents” on page 5-3

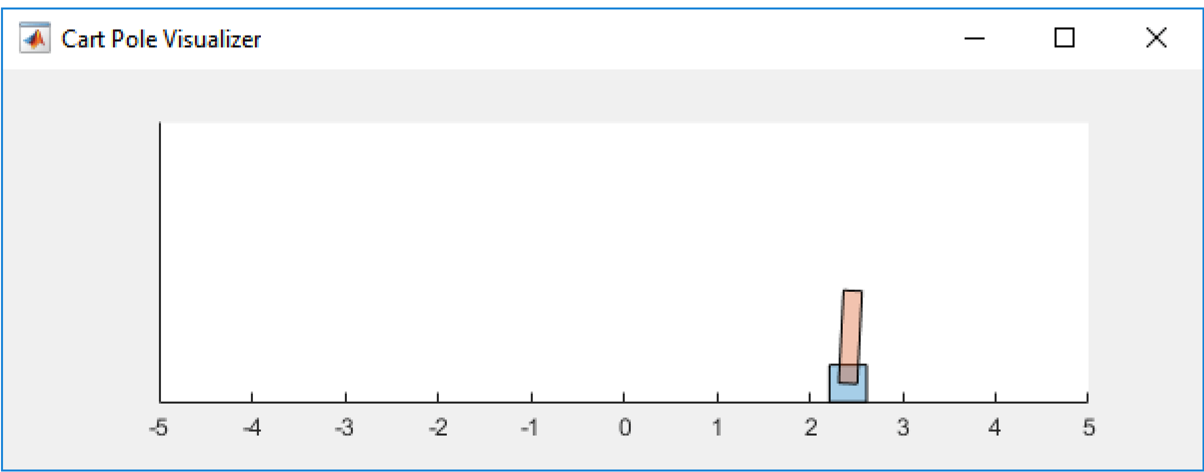
# Train AC Agent to Balance Cart-Pole System

This example shows how to train an actor-critic (AC) agent to balance a cart-pole system modeled in MATLAB®.

For more information on AC agents, see “Actor-Critic (AC) Agents” on page 3-31. For an example showing how to train an AC agent using parallel computing, see “Train AC Agent to Balance Cart-Pole System Using Parallel Computing” on page 5-166.

### Cart-Pole MATLAB Environment

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




For this environment:

- The upward balanced pendulum position is `0` radians, and the downward hanging position is `pi` radians.
- The pendulum starts upright with an initial angle between –0.05 and 0.05 rad.
- The force action signal from the agent to the environment is either –10 or 10 N.
- The observations from the environment are the position and velocity of the cart, the pendulum angle, and the pendulum angle derivative.
- The episode terminates if the pole is more than 12 degrees from vertical or if the cart moves more than 2.4 m from the original position.
- A reward of +1 is provided for every time step that the pole remains upright. A penalty of –5 is applied when the pendulum falls.

For more information on this model, see “Load Predefined Control System Environments” on page 2-23.

### Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv("CartPole-Discrete")
```

```
env = 
  CartPoleDiscreteAction with properties:

                  Gravity: 9.8000
                 MassCart: 1
                 MassPole: 0.1000
                   Length: 0.5000
                 MaxForce: 10
                       Ts: 0.0200
    ThetaThresholdRadians: 0.2094
               XThreshold: 2.4000
      RewardForNotFalling: 1
        PenaltyForFalling: -5
                    State: [4x1 double]
```

```
env.PenaltyForFalling = -10;
```

The interface has a discrete action space where the agent can apply one of two possible force values to the cart, –10 or 10 N.

Obtain the observation and action information from the environment interface.

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create AC Agent

An AC agent approximates the discounted cumulative long-term reward using a value-function critic. A value-function critic must accept an observation as input and return a single scalar (the estimated discounted cumulative long-term reward) as output.

To approximate the value function within the critic, use a neural network. Define the network as an array of layer objects, and get the dimension of the observation space and the number of possible actions from the environment specification objects. For more information on creating a deep neural network value function representation, see “Create Policies and Value Functions” on page 4-2.

```
criticNet = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(1)];
```

Convert to `dlnetwork` and display the number of weights.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 193

   Inputs:
      1   'input'   4 features
```

Create the critic approximator object using `criticNet`, and the observation specification. For more information, see `rlValueFunction`.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Check the critic with a random observation input.

```
getValue(critic,{rand(obsInfo.Dimension)})
```

```
ans = single
    -0.3590
```

An AC agent decides which action to take using a stochastic policy, which for discrete action spaces is approximated by a discrete categorical actor. This actor must take the observation signal as input and return a probability for each action.

To approximate the policy function within the actor, use a deep neural network. Define the network as an array of layer objects, and get the dimension of the observation space and the number of possible actions from the environment specification objects.

```
actorNet = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))
    softmaxLayer];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 226

   Inputs:
      1   'input'   4 features
```

Create the actor approximator object using `actorNet` and the observation and action specifications. For more information, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

To return the probability distribution of the possible actions as a function of a random observation, and given the current network weights, use evaluate.

```
prb = evaluate(actor,{rand(obsInfo.Dimension)})
```

```
prb = 1x1 cell array
    {2x1 single}
```

```
prb{1}
```

```
ans = 2x1 single column vector

    0.4414
```

```
    0.5586
```

Create the agent using the actor and critic. For more information, see `rlACAgent`.

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

Check the agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-10]}
```

Specify agent options, including training options for the actor and critic, using dot notation. Alternatively, you can use `rlACAgentOptions` and `rlOptimizerOptions` objects before creating the agent.

```
agent.AgentOptions.EntropyLossWeight = 0.01;

agent.AgentOptions.ActorOptimizerOptions.LearnRate = 1e-2;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold = 1;
agent.AgentOptions.CriticOptimizerOptions.LearnRate = 1e-2;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
```

### Train Agent

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

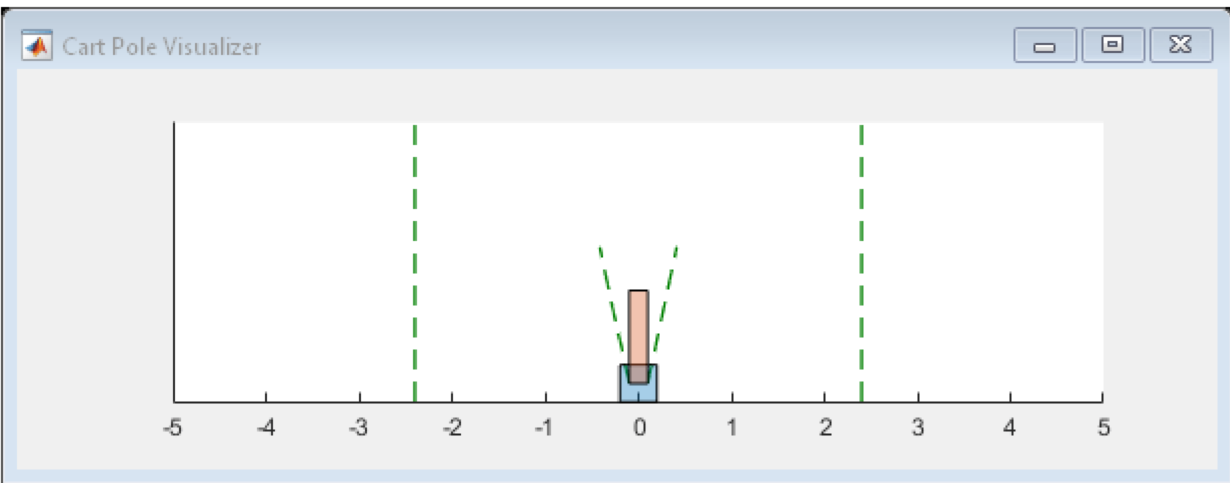
- Run each training episode for at most 1000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than 480 over 10 consecutive episodes. At this point, the agent can balance the pendulum in the upright position.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=500,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=480,...
    ScoreAveragingWindowLength=10);
```

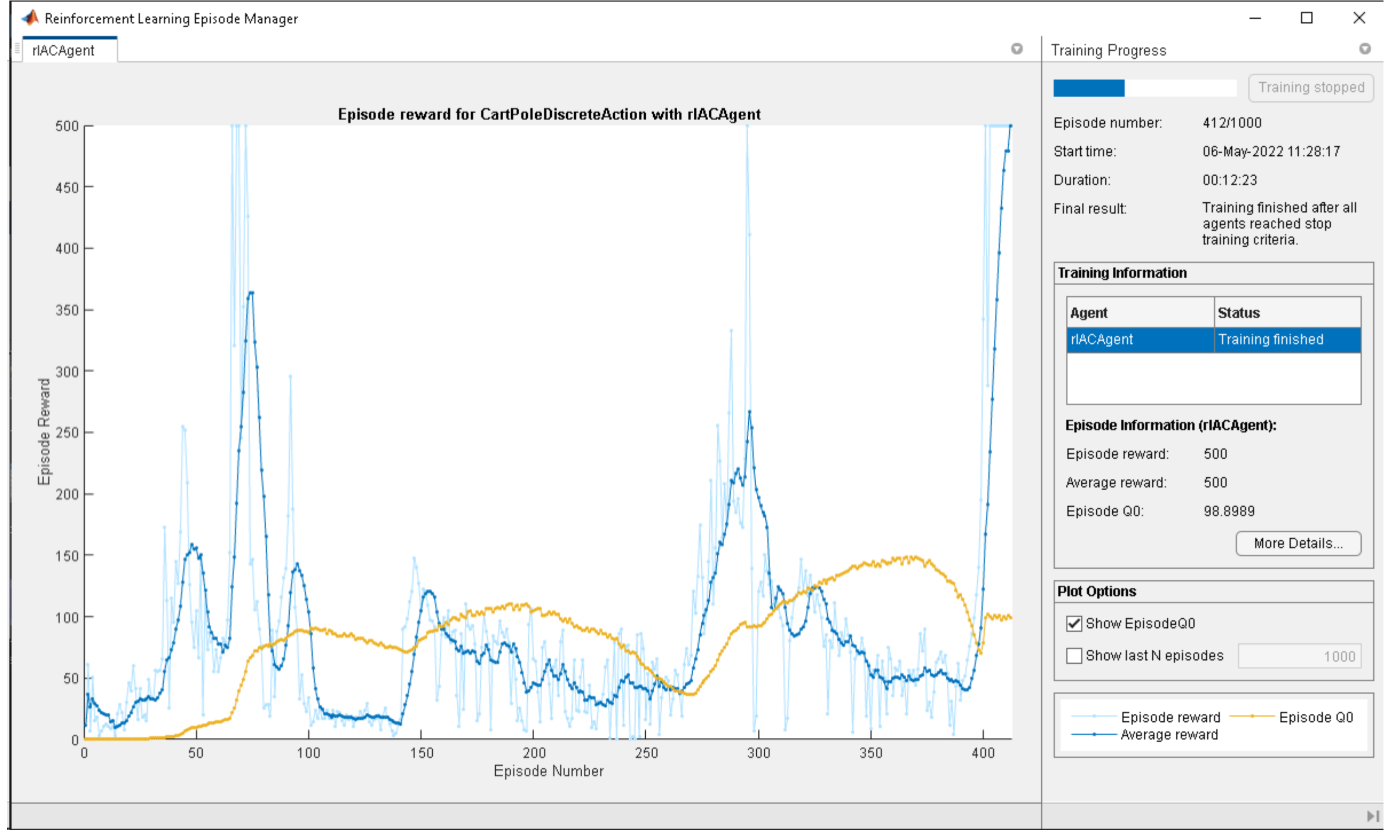
You can visualize the cart-pole system during training or simulation using the `plot` function.

```
plot(env)
```




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

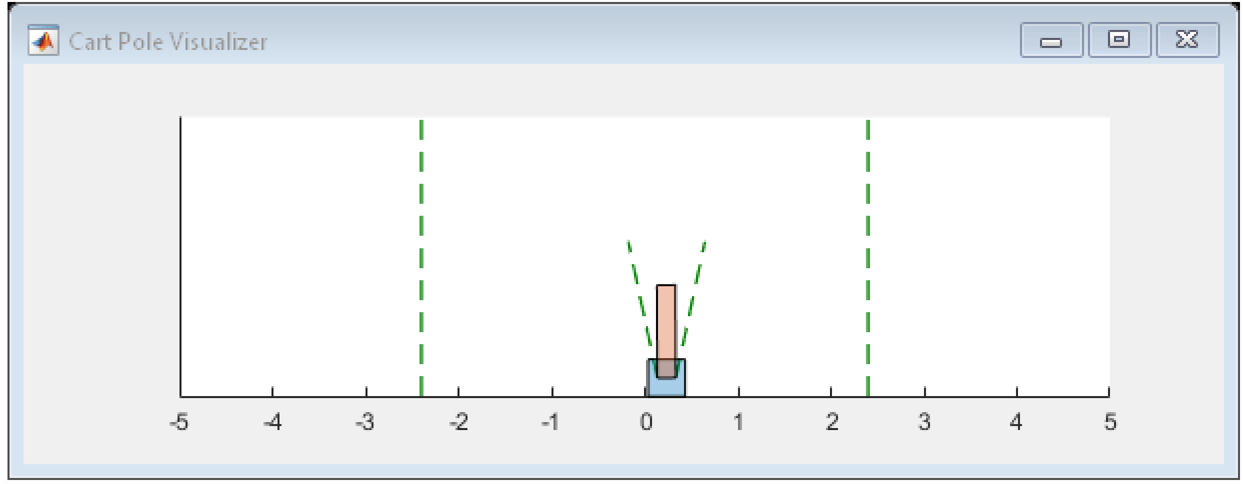
```
doTraining = false;

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




## Simulate AC Agent

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




```
totalReward = sum(experience.Reward)
```

```
totalReward = 500
```

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim`

**Objects**
`rlACAgent` | `rlACAgentOptions` | `rlTrainingOptions`

## Related Examples


- “Train PG Agent to Balance Cart-Pole System” on page 5-57
- “Train AC Agent to Balance Cart-Pole System Using Parallel Computing” on page 5-166


## More About


- “Load Predefined Control System Environments” on page 2-23
- “Actor-Critic (AC) Agents” on page 3-31
- “Create Policies and Value Functions” on page 4-2
- “Train Reinforcement Learning Agents” on page 5-3

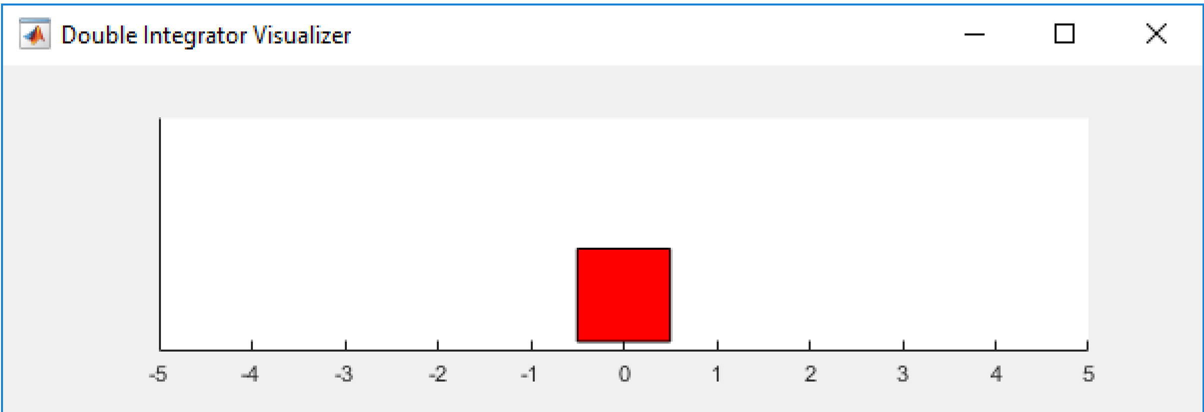
# Train PG Agent with Baseline to Control Double Integrator System

This example shows how to train a policy gradient (PG) agent with baseline to control a second-order dynamic system modeled in MATLAB®.

For more information on the basic PG agent with no baseline, see the example “Train PG Agent to Balance Cart-Pole System” on page 5-57.

### Double Integrator MATLAB Environment

The reinforcement learning environment for this example is a second-order double integrator system with a gain. The training goal is to control the position of a mass in the second-order system by applying a force input.




For this environment:

- The mass starts at an initial position between –2 and 2 units.
- The force action signal from the agent to the environment is from –2 to 2 N.
- The observations from the environment are the position and velocity of the mass.
- The episode terminates if the mass moves more than 5 m from the original position or if $|x| < 0.01$.
- The reward $r_t$, provided at every time step, is a discretization of $r(t)$:

$$r(t) = -(x(t)'Qx(t) + u(t)'Ru(t))$$

Here:

- $x$ is the state vector of the mass.
- $u$ is the force applied to the mass.
- $Q$ is the weights on the control performance; $Q = [10\ 0; 0\ 1]$.
- $R$ is the weight on the control effort; $R = 0.01$.

For more information on this model, see “Load Predefined Control System Environments” on page 2-23.

### Create Double Integrator MATLAB Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv("DoubleIntegrator-Discrete")
```

```
env = 
  DoubleIntegratorDiscreteAction with properties:

             Gain: 1
               Ts: 0.1000
      MaxDistance: 5
    GoalThreshold: 0.0100
                Q: [2x2 double]
                R: 0.0100
         MaxForce: 2
            State: [2x1 double]
```

The interface has a discrete action space where the agent can apply one of three possible force values to the mass: -2, 0, or 2 N.

Obtain the observation and action information from the environment interface.

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create PG Agent Actor

For policy gradient agents, the actor executes a stochastic policy, which for discrete action spaces is approximated by a discrete categorical actor. This actor must take the observation signal as input and return a probability for each action.

To approximate the policy within the actor, use a neural network. Define the network as an array of layer objects with one input (the observation) and one output (the action), and get the dimension of the observation space and the number of possible actions from the environment specification objects. For more information on creating a deep neural network value function representation, see “Create Policies and Value Functions” on page 4-2.

```
actorNet = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(numel(actInfo.Elements))];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 9

   Inputs:
      1   'input'   2 features
```

Specify training options for the actor. For more information, see `rlOptimizerOptions`. Alternatively, you can change agent (including actor and critic) options using dot notation after the agent is created.

```
actorOpts = rlOptimizerOptions( ...
    LearnRate=5e-3, ...
    GradientThreshold=1);
```

Create the actor representation using the neural network and the environment specification objects. For more information, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

To return the probability distribution of the possible actions as a function of a random observation, and given the current network weights, use `evaluate`.

```
prb = evaluate(actor,{rand(obsInfo.Dimension)})
```

```
prb = 1x1 cell array
    {3x1 single}
```

```
prb{1}
```

```
ans = 3x1 single column vector

    0.4994
    0.3770
    0.1235
```

### Create PG Agent Baseline

The PG Agent algorithm, (also known as REINFORCE) returns can be compared to a baseline that depends on the state. This can reduce the variance of the expected value of the update and thus improve the speed of learning. A possible choice for the baseline is an estimate of the state value function [1].

A value-function approximator object must accept an observation as input and return a single scalar (the estimated discounted cumulative long-term reward) as output. Use a neural network as approximation model. Define the network as an array of layer objects, and get the dimension of the observation space and the number of possible actions from the environment specification objects.

```
baselineNet = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(8)
    reluLayer
    fullyConnectedLayer(1)];
```

Convert to `dlnetwork` and display the number of weights.

```
baselineNet = dlnetwork(baselineNet);
```

Create the baseline value function approximator using `baselineNet`, and the observation specification. For more information, see `rlValueFunction`.

```
baseline = rlValueFunction(baselineNet,obsInfo);
```

Check the baseline with a random observation input.

```
getValue(baseline,{rand(obsInfo.Dimension)})
```

```
ans = single
    0.2152
```

Specify some training option for the baseline.

```
baselineOpts = rlOptimizerOptions( ...
    LearnRate=5e-3, ...
    GradientThreshold=1);
```

To create the PG agent with baseline, specify the PG agent options using `rlPGAgentOptions` and set the `UseBaseline` option set to `true`.

```
agentOpts = rlPGAgentOptions(...
    UseBaseline=true, ...
    ActorOptimizerOptions=actorOpts, ...
    CriticOptimizerOptions=baselineOpts);
```

Then create the agent using the specified actor representation, baseline representation, and agent options. For more information, see `rlPGAgent`.

```
agent = rlPGAgent(actor,baseline,agentOpts);
```

Check the agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0]}
```

### Train Agent

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

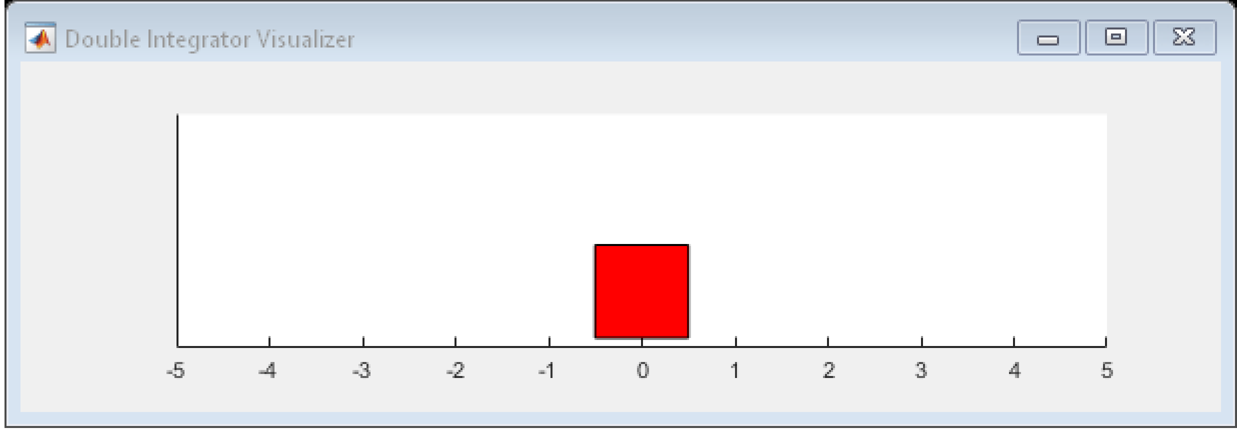
- Run at most 1000 episodes, with each episode lasting at most 200 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option).
- Stop training when the agent receives a moving average cumulative reward greater than –45. At this point, the agent can control the position of the mass using minimal control effort.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=1000, ...
    MaxStepsPerEpisode=200, ...
    Verbose=false, ...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-43);
```

You can visualize the double integrator system using the `plot` function during training or simulation.

```
plot(env)
```



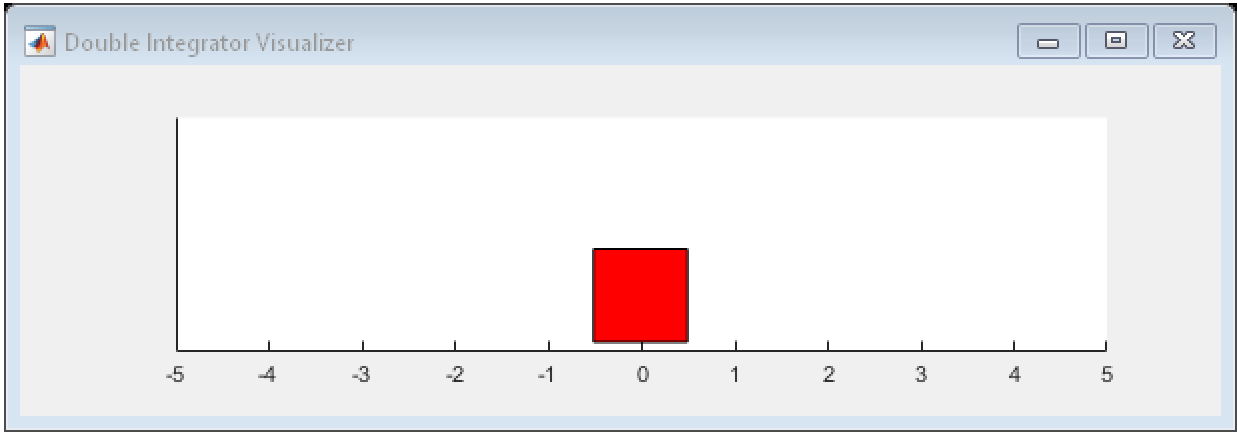

Train the agent using the `train` function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

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

### Simulate PG Agent

To validate the performance of the trained agent, simulate it within the double integrator environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




```
totalReward = sum(experience.Reward)
```

```
totalReward = -39.9140
```

**References**

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, MA: The MIT Press, 2018.

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim`

**Objects**
`rlPGAgent` | `rlPGAgentOptions` | `rlTrainingOptions`

## Related Examples


- "Train PG Agent to Balance Cart-Pole System" on page 5-57
- "Train DDPG Agent to Control Double Integrator System" on page 5-77


## More About


- "Load Predefined Control System Environments" on page 2-23
- "Policy Gradient (PG) Agents" on page 3-27
- "Create Policies and Value Functions" on page 4-2
- "Train Reinforcement Learning Agents" on page 5-3

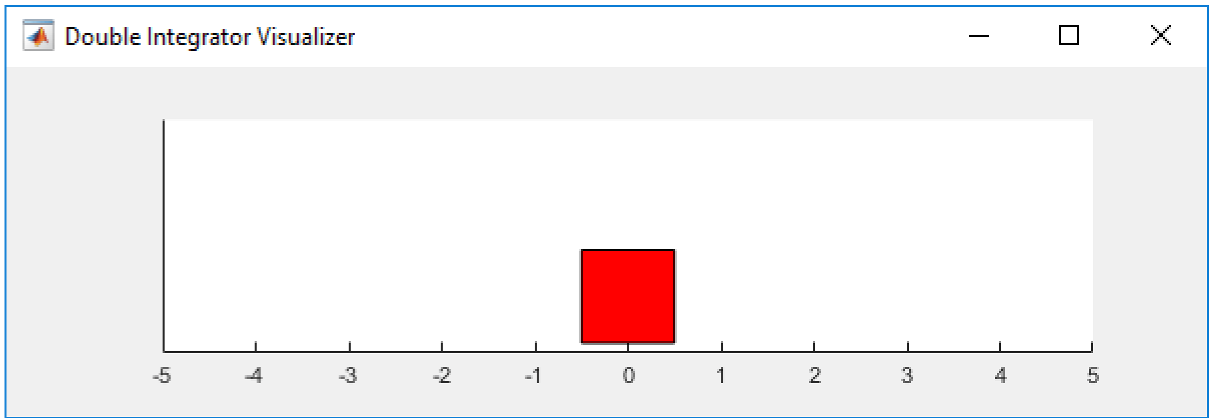
# Train DDPG Agent to Control Double Integrator System

This example shows how to train a deep deterministic policy gradient (DDPG) agent to control a second-order linear dynamic system modeled in MATLAB®. The example also compares the DDPG agent to an LQR controller.

For more information on DDPG agents, see “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40. For an example showing how to train a DDPG agent in Simulink®, see “Train DDPG Agent to Swing Up and Balance Pendulum” on page 5-97.

### Double Integrator MATLAB Environment

The reinforcement learning environment for this example is a second-order double-integrator system with a gain. The training goal is to control the position of a mass in the second-order system by applying a force input.




For this environment:

- The mass starts at an initial position between –4 and 4 units.
- The observations from the environment are the position and velocity of the mass.
- The episode terminates if the mass moves more than 5 m from the original position or if $|x| < 0.01$.
- The reward $r_t$, provided at every time step, is a discretization of $r(t)$:
- $r(t) = -(x(t)' Q x(t) + u(t)' R u(t))$

Here:

- $x$ is the state vector of the mass.
- $u$ is the force applied to the mass.
- $Q$ is the matrix of weights on the control performance; $Q = [10\ 0; 0\ 1]$.
- $R$ is the weight on the control effort; $R = 0.01$.

For more information on this model, see “Load Predefined Control System Environments” on page 2-23.

For this example the environment is a linear dynamical system, the environment state is observed directly, and the reward is a quadratic function of the observation and action. Therefore the problem

of finding the sequence of actions that minimizes the cumulative long-term reward is a discrete-time linear-quadratic optimal control problem, for which the optimal action is known to be a linear function of the system states. This problem can also be solved using Linear-Quadratic Regulator (LQR) design, and in the last part of the example you can compare the agent to an LQR controller.

### Create Environment Interface

Create a predefined environment interface for the double integrator system.

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

```
env = 
  DoubleIntegratorContinuousAction with properties:

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

The interface has a continuous action space where the agent can apply force values from `-Inf` to `Inf` to the mass. The sample time is stored in `env.Ts`, while the continuous time cost function matrices are stored in `env.Q` and `env.R` respectively.

Obtain the observation and action information from the environment interface.

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

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "states"
    Description: "x, dx"
      Dimension: [2 1]
       DataType: "double"
```

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

```
actInfo = 
  rlNumericSpec with properties:

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

Reset the environment and get its initial state.

```
x0 = reset(env)
```

```
x0 = 2×1

     4
     0
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create DDPG Agent

A DDPG agent approximates the discounted cumulative long-term reward using a Q-value-function critic. A Q-value function critic must accept an observation and an action as inputs and return a scalar (the estimated discounted cumulative long-term reward) as output. To approximate the Q-value function within the critic, use a neural network. The value function of the optimal policy is known to be quadratic, use a network with a quadratic layer (which outputs a vector of quadratic monomials, as described in `quadraticLayer`) and a fully connected layer (which provides a linear combination of its inputs).

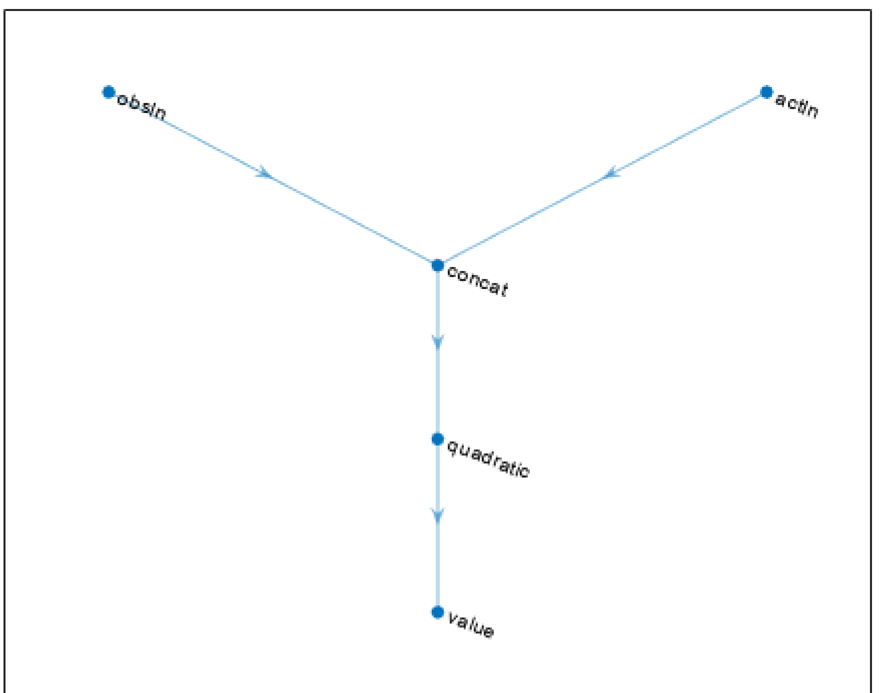
Define each network path as an array of layer objects and get the dimension of the observation and action spaces from the environment specification objects. Assign names to the network input layers, so you can connect them to the output path and later explicitly associate them with the appropriate environment channel. Since there is no need for a bias term, set the bias term to zero (`Bias=0`) and prevent it from changing (`BiasLearnRateFactor=0`).

For more information on creating value function approximators, see “Create Policies and Value Functions” on page 4-2.

```
% Observation and action paths
obsPath = featureInputLayer(obsInfo.Dimension(1),Name="obsIn");
actPath = featureInputLayer(actInfo.Dimension(1),Name="actIn");

% Common path
commonPath = [
    concatenationLayer(1,2,Name="concat")
    quadraticLayer
    fullyConnectedLayer(1,Name="value", ...
        BiasLearnRateFactor=0,Bias=0)
    ];

% Add layers to layerGraph object
criticNet = layerGraph(obsPath);
criticNet = addLayers(criticNet,actPath);
criticNet = addLayers(criticNet,commonPath);

% Connect layers
criticNet = connectLayers(criticNet,"obsIn","concat/in1");
criticNet = connectLayers(criticNet,"actIn","concat/in2");
```

View the critic network configuration.

```
figure
plot(criticNet)
```




Convert to `dlnetwork` and display the number of weights.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 7

   Inputs:
      1   'obsIn'   2 features
      2   'actIn'   1 features
```

Create the critic approximator object using `criticNet`, the environment observation and action specifications, and the names of the network input layers to be connected with the environment observation and action channels, respectively. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(criticNet, ...
    obsInfo,actInfo, ...
    ObservationInputNames="obsIn",ActionInputNames="actIn");
```

Check the critic with a random observation and action input.

```
getValue(critic,{rand(obsInfo.Dimension)},{rand(actInfo.Dimension)})
```

```
ans = single
    -0.3977
```

DDPG agents use a parametrized continuous deterministic policy, which is implemented by a continuous deterministic actor. This actor must accept an observation as input and return an action as output. To approximate the policy function within the actor, use a neural network. Since for this example the optimal policy is known to be linear in the state, use a shallow network with a fully connected layer to provide a linear combination of the two network inputs.

Define the network as an array of layer objects, and get the dimension of the observation and action spaces from the environment specification objects. Since there is no need for a bias term, as done for the critic, set the bias term to zero (`Bias=0`) and prevent it from changing (`BiasLearnRateFactor=0`). For more information on actors, see “Create Policies and Value Functions” on page 4-2.

```
actorNet = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(actInfo.Dimension(1), ...
        BiasLearnRateFactor=0,Bias=0)
    ];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 3

   Inputs:
      1   'input'   2 features
```

Create the actor using `actorNet` and the observation and action specifications. For more information, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNet,obsInfo,actInfo);
```

Check the actor with a random observation input.

```
getAction(actor,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[0.3493]}
```

Create the DDPG agent using the actor and critic. For more information, see `rlDDPGAgent`.

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

Specify options for the agent, including training options for the critic, using dot notation. Alternatively, you can use `rlDDPGAgentOptions`, and `rlOptimizerOptions` objects before creating the agent.

```
agent.AgentOptions.SampleTime = env.Ts;
agent.AgentOptions.ExperienceBufferLength = 1e6;
agent.AgentOptions.MiniBatchSize = 32;
agent.AgentOptions.NoiseOptions.StandardDeviation = 0.3;
agent.AgentOptions.NoiseOptions.StandardDeviationDecayRate = 1e-7;
agent.AgentOptions.ActorOptimizerOptions.LearnRate = 1e-4;
agent.AgentOptions.ActorOptimizerOptions.GradientThreshold = 1;
```

```
agent.AgentOptions.CriticOptimizerOptions.LearnRate = 5e-3;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
```

**Initialize Agent Parameters**

The policy implemented by the actor is $u = K_1 x_1 + K_2 x_2 = Kx$, where the feedback gains $K_1$ and $K_2$ are the two weights of the actor network. It can be shown that the closed loop system is stable if these gains are negative, therefore, initializing them to negative values can speed up convergence.

The Q-value function has the following structure:

$$Q(x, u) = W_1 x_1^2 + W_2 x_1 x_2 + W_3 x_2^2 + W_4 x_1 u + W_5 x_2 u + W_6 u^2$$

Where$W_i$ are the weights of the fully connected layer. Alternatively, in matrix form:

$$Q(x, u) = \begin{bmatrix} x_1 & x_2 & u \end{bmatrix} \begin{bmatrix} W_1 & \frac{W_2}{2} & \frac{W_4}{2} \\ \frac{W_2}{2} & W_3 & \frac{W_5}{2} \\ \frac{W_4}{2} & \frac{W_5}{2} & W_6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ u \end{bmatrix} = \begin{bmatrix} x^T & u \end{bmatrix} W \begin{bmatrix} x \\ u \end{bmatrix}$$

For a fixed policy $u = Kx$, the cumulative long-term reward (that is the value of the policy) becomes:

$$V(x) = Q(x, Kx) =$$
$$x^T \begin{bmatrix} I & K^T \end{bmatrix} W \begin{bmatrix} I \\ K \end{bmatrix} x =$$
$$x^T P x$$

Since the rewards are always negative, to properly approximate the cumulative reward both $P$ and $W$ must be negative definite. Therefore, to speed up convergence, initialize the critic network weights$W_i$ so that $W$ is negative definite.

```
% Create diagonal matrix with negative eigenvalues
W = -single(diag([1 1 1])+0.1)

W = 3x3 single matrix

   -1.1000   -0.1000   -0.1000
   -0.1000   -1.1000   -0.1000
   -0.1000   -0.1000   -1.1000


% Extract indexes of upper triangular part of a 3 by 3 matrix
idx = triu(true(3))

idx = 3x3 logical array

   1   1   1
   0   1   1
   0   0   1


% Update parameters in the actor and critic
par = getLearnableParameters(agent);
```

```
par.Actor{1} = -single([1 1]);
par.Critic{1} = W(idx)';
setLearnableParameters(agent,par);
```

Check the agent with a random observation input.

```
getAction(agent,{rand(obsInfo.Dimension)})
```

```
ans = 1x1 cell array
    {[-1.2857]}
```

### Train Agent

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

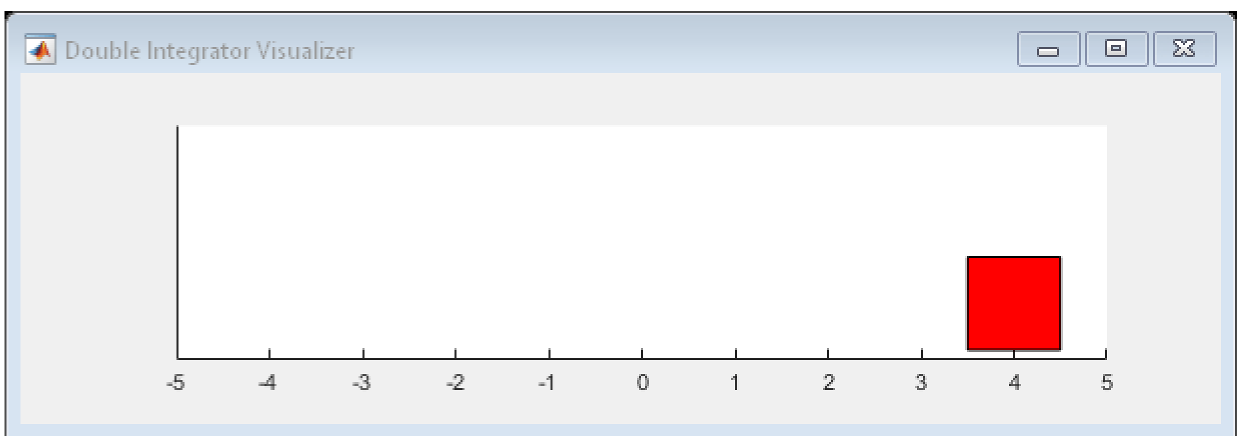
- Run at most `5000` episodes in the training session, with each episode lasting at most `200` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (`Verbose` option).
- Stop training when the agent receives a moving average cumulative reward greater than –`66`. At this point, the agent can control the position of the mass using minimal control effort.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=5000, ...
    MaxStepsPerEpisode=200, ...
    Verbose=false, ...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-66);
```

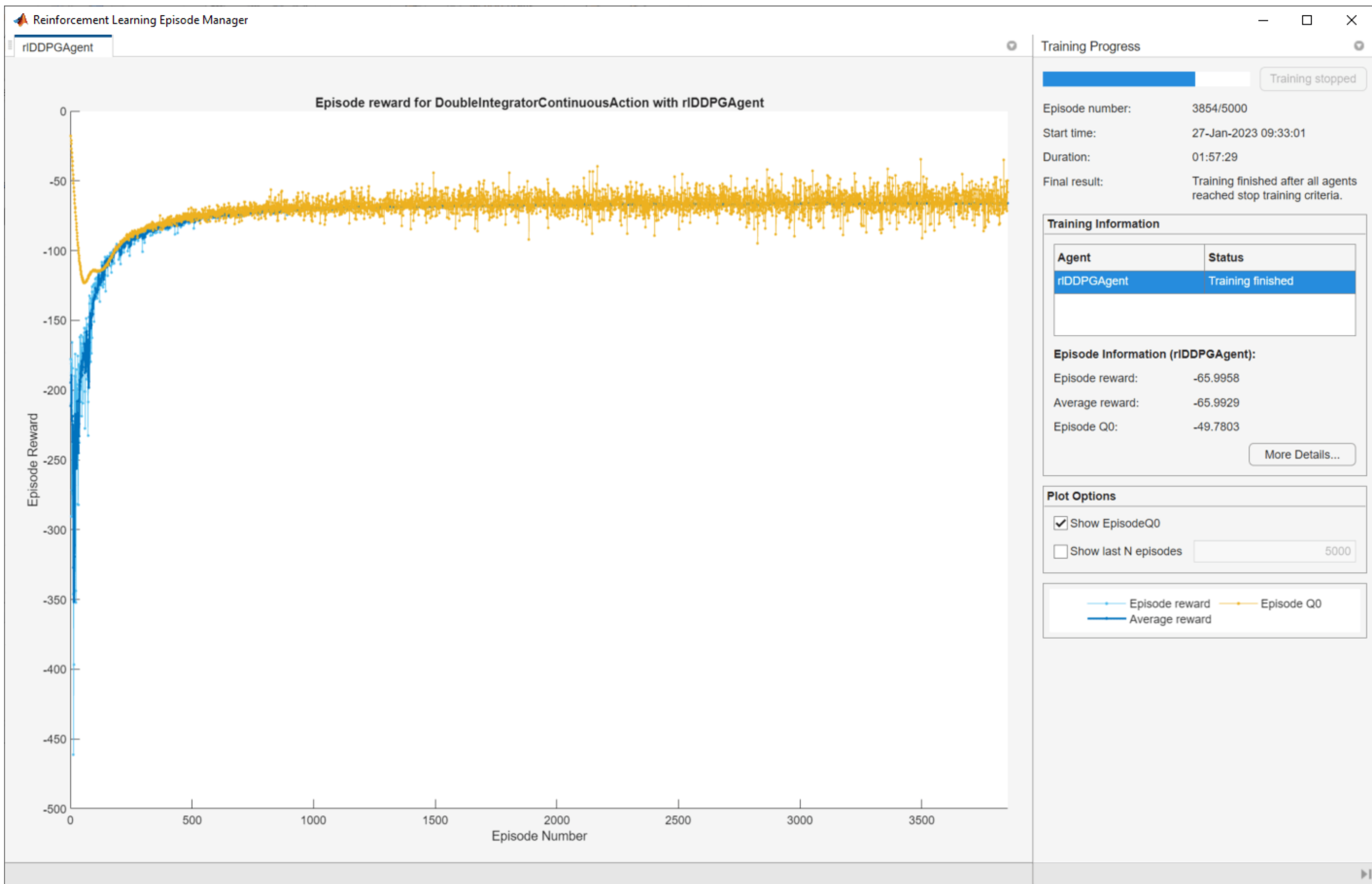
You can visualize the double integrator environment by using the `plot` function during training or simulation.

```
plot(env)
```




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

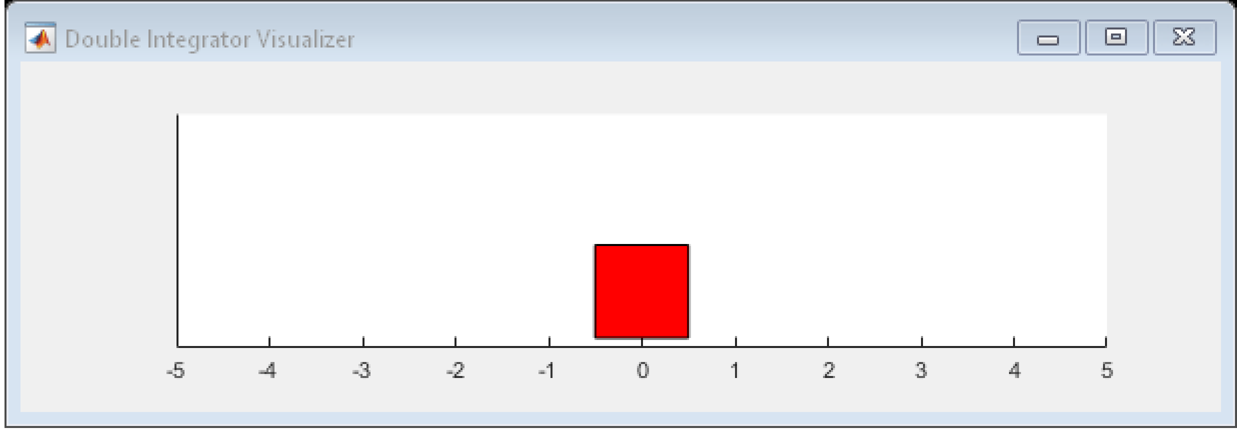
```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load("DoubleIntegDDPG.mat","agent");
end
```




### Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the double integrator environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




```
totalReward = sum(experience.Reward)
```

```
totalReward = -65.9875
```

### Solve LQR Problem

The function `lqrd` (Control System Toolbox) solves a discretized LQR problem, like the one presented in this example. This function calculates the optimal discrete-time gain matrix `Klqr`, together with the solution of the Riccati equation `Plqr`. When `Klqr` is connected via negative state feedback to the plant input (force), the discrete-time equivalent of the cost function specified by `env.Q` and `env.R` is minimized going forward. Furthermore, the cumulative cost from initial time to infinity, starting from an initial state `x0`, is equal to `x0'*Plqr*x0`.

Use `lqrd` to solve the discretized LQR problem.

```
[Klqr,Plqr] = lqrd([0 1;0 0],[0;env.Gain],env.Q,env.R,env.Ts);
```

Here, `[0 1;0 0]` and `[0;env.Gain]` are the continuous-time transition and input gain matrices, respectively, of the double integrator system.

If Control System Toolbox™ is not installed, use the solution for the default example values.

```
Klqr = [17.8756 8.2283];
Plqr = [4.1031 0.3376; 0.3376 0.1351];
```

If the actor policy $u = Kx$ successfully approximates the optimal policy, then the resulting $K$must be close to $-K_{\text{lqr}}$ (the minus sign is due to the fact that $K_{\text{lqr}}$ is calculated assuming a negative feedback connection).

If the critic learns a good approximation of the optimal value function, then the resulting $P$, as defined before, must be close to $-P_{\text{lqr}}$ (the minus sign is due to the fact that the reward is defined as the negative of the cost).

### Compare DDPG Agent to Optimal Controller

Extract the parameters (weighs) of the actor and critic within the agent.

```
par = getLearnableParameters(agent);
```

Display the actor weights.

```
K = par.Actor{1}
```

```
K = 1x2 single row vector

  -15.4601   -7.2076
```

Note that the gains are close to the ones of the optimal solution `-Klqr`:

```
-Klqr
```

```
ans = 1×2

  -17.8756   -8.2283
```

Recreate the matrices $W$ and $P$ defining the Q-value and value functions, respectively. First, re-initialize $W$ to zero.

```
W = zeros(3);
```

Place the critic weights in the upper triangular portion of $W_c$.

```
W(idx) = par.Critic{1};
```

Recreate $W_c$ as defined before.

```
W = (W + W')/2
```

```
W = 3×3

   -4.9869   -0.7788   -0.0548
   -0.7788   -0.3351   -0.0222
   -0.0548   -0.0222    0.0008
```

Using $W$ and $K$, calculate $P$ as defined before.

```
P = [eye(2) K']*W*[eye(2);K]
```

```
P = 2x2 single matrix

   -3.1113    0.0436
    0.0436    0.0241
```

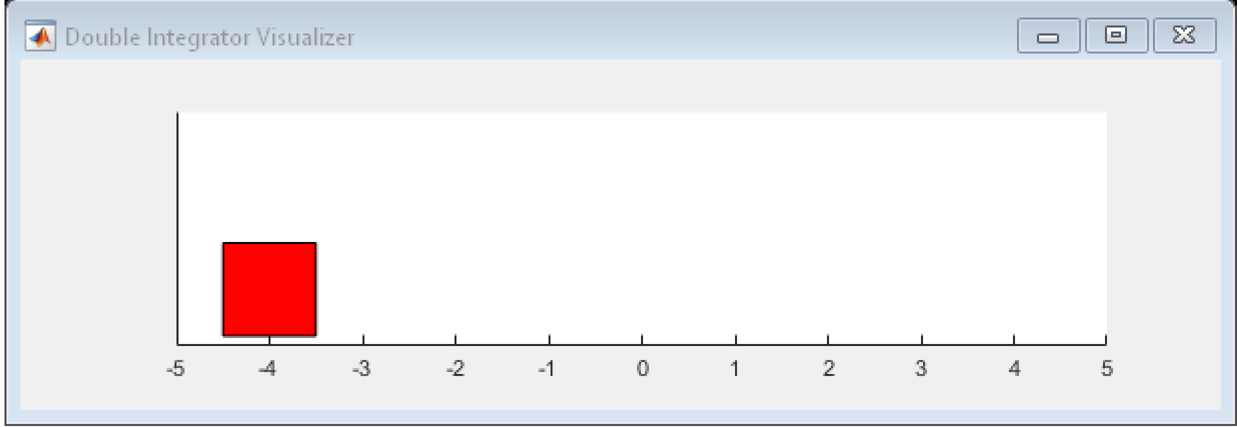
Note that the gains are close to the solution of the Riccati equation `-Plqr`.

```
-Plqr
```

```
ans = 2×2

   -4.1031   -0.3376
   -0.3376   -0.1351
```

Get the environment initial state.

```
x0=reset(env);
```




The value function is the estimate of future cumulative long-term reward when using the policy enacted by the actor. Calculate the value function at the initial state, according to the critic weights. This is the same value displayed in the training window as **Episode Q0**.

```
q0 = x0'*P*x0
```

```
q0 = single
    -49.7803
```

Note that the value is close to the actual reward obtained in the validation simulation, `totalReward`, suggesting that the critic learns a good approximation of the value function for the policy enacted by the actor.

Calculate the value of the initial state, following the true optimal policy enacted by the LQR controller.

```
-x0'*Plqr*x0
```

```
ans = -65.6494
```

This value is also very close to the value obtained in the validation simulation, confirming that the policy learned and enacted by the actor is a good approximation of the true optimal policy.

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim` | `lqr`

**Objects**
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlTrainingOptions`

## Related Examples

- “Train PG Agent with Baseline to Control Double Integrator System” on page 5-70
- “Train DDPG Agent to Swing Up and Balance Pendulum” on page 5-97

- “Tune PI Controller Using Reinforcement Learning” on page 5-392
- “Train Custom LQR Agent” on page 5-466

## More About

- “Load Predefined Control System Environments” on page 2-23
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Create Policies and Value Functions” on page 4-2
- “Train Reinforcement Learning Agents” on page 5-3

# Train DQN Agent to Swing Up and Balance Pendulum

This example shows how to train a deep Q-learning network (DQN) agent to swing up and balance a pendulum modeled in Simulink®.

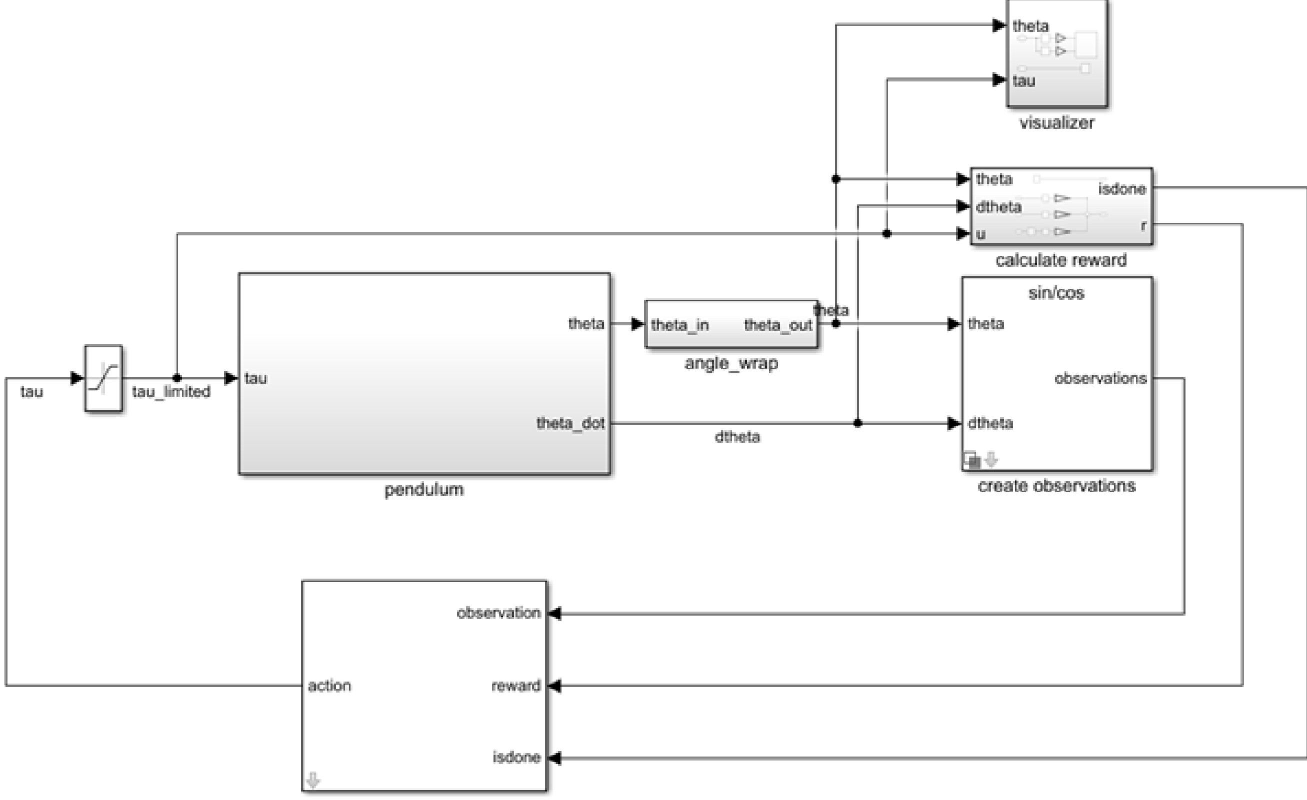
For more information on DQN agents, see “Deep Q-Network (DQN) Agents” on page 3-23. For an example that trains a DQN agent in MATLAB®, see “Train DQN Agent to Balance Cart-Pole System” on page 5-50.

### Pendulum Swing-up Model

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

Open the model.

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




For this model:

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

- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are the sine of the pendulum angle, the cosine of the pendulum angle, and the pendulum angle derivative.
- The reward $r_t$, provided at every timestep, is

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

Here:

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

For more information on this model, see “Load Predefined Simulink Environments” on page 2-30.

### Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv("SimplePendulumModel-Discrete")
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

The interface has a discrete action space where the agent can apply one of three possible torque values to the pendulum: –2, 0, or 2 N·m.

To define the initial condition of the pendulum as hanging downward, specify an environment reset function using an anonymous function handle. This reset function sets the model workspace variable `theta0` to `pi`.

```
env.ResetFcn = @(in)setVariable(in,"theta0",pi,"Workspace",mdl);
```

Get the observation and action specification information from the environment

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

```
obsInfo = 
  rlNumericSpec with properties:

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

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

```
actInfo = 
  rlFiniteSetSpec with properties:

       Elements: [3x1 double]
           Name: "torque"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

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

```
Ts = 0.05;
Tf = 20;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

#### Create DQN Agent

A DQN agent approximates the long-term reward, given observations and actions, using a parametrized Q-value function critic.

For DQN agents with a discrete action space, you have the option to create a vector (that is a multi-output) Q-value function critic, which is generally more efficient than a comparable single-output critic. A vector Q-value function is a mapping from an environment observation to a vector in which each element represents the expected discounted cumulative long-term reward when an agent starts from the state corresponding to the given observation and executes the action corresponding to the element number (and follows a given policy afterwards).

To model the Q-value function within the critic, use a deep neural network. The network must have one input layer (which receives the content of the observation channel, as specified by `obsInfo`) and one output layer (which returns the vector of values for all the possible actions). Note that `prod(obsInfo.Dimension)` returns the number of dimensions of the observation space (regardless of whether they are arranged as a row vector, column vector, or matrix, while `numel(actInfo.Elements)` returns the number of elements of the discrete action space.

Define the network as an array of layer objects.

```
criticNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(48)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements))];
```

Convert to `dlnetwork` and display the number of weights.

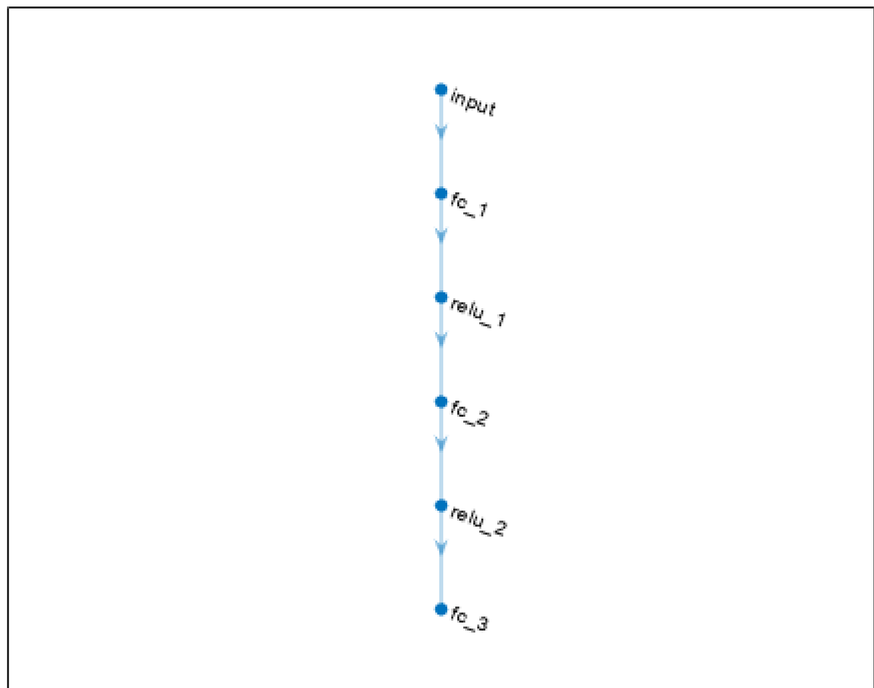
```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 1.4k
```

```
Inputs:
   1   'input'   3 features
```

View the critic network configuration.

```
plot(criticNet)
```




For more information on creating value functions that use a deep neural network model, see “Create Policies and Value Functions” on page 4-2.

Create the critic using `criticNet`, as well as observation and action specifications. For more information, see `rlVectorQValueFunction`.

```
critic = rlVectorQValueFunction(criticNet,obsInfo,actInfo);
```

Specify options for the critic optimizer using `rlOptimizerOptions`.

```
criticOpts = rlOptimizerOptions(LearnRate=0.001,GradientThreshold=1);
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions`.

```
agentOptions = rlDQNAgentOptions(...
    SampleTime=Ts,...
    CriticOptimizerOptions=criticOpts,...
    ExperienceBufferLength=3000,...
    UseDoubleDQN=false);
```

Then, create the DQN agent using the specified critic and agent options. For more information, see `rlDQNAgent`.

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

#### Train Agent

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

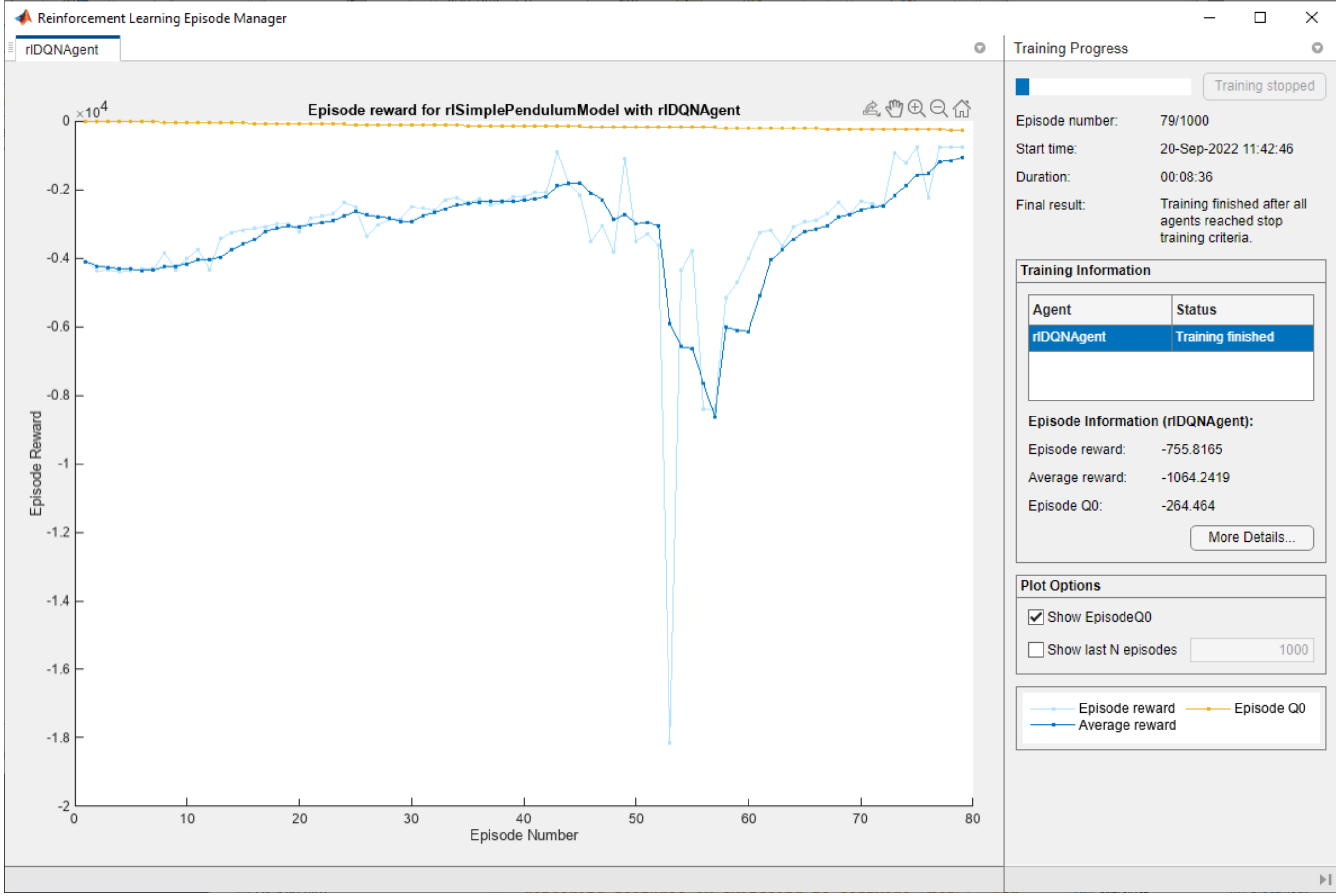
- Run each training for at most 1000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –1100 over five consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.
- Save a copy of the agent for each episode where the cumulative reward is greater than –1100.

For more information, see `rlTrainingOptions`.

```
trainingOptions = rlTrainingOptions(...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=500,...
    ScoreAveragingWindowLength=5,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-1100,...
    SaveAgentCriteria="EpisodeReward",...
    SaveAgentValue=-1100);
```

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

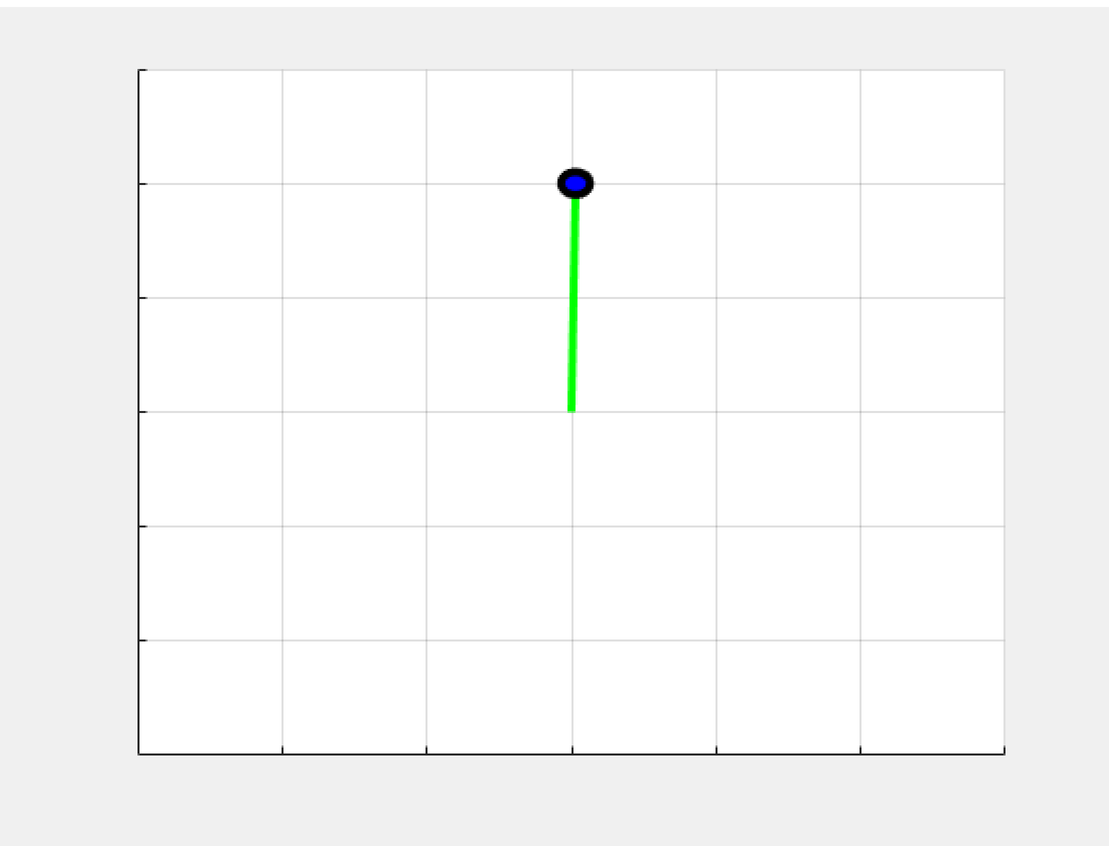
```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
    % Load the pretrained agent for the example.
    load("SimulinkPendulumDQNMulti.mat","agent");
end
```




### Simulate DQN Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```



## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim`

**Objects**
`rlDQNAgent` | `rlDQNAgentOptions` | `rlVectorQValueFunction` | `rlTrainingOptions` | `rlSimulationOptions` | `rlOptimizerOptions`

**Blocks**
RL Agent

## Related Examples

- “Train DDPG Agent to Swing Up and Balance Pendulum” on page 5-97
- “Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal” on page 5-115
- “Create DQN Agent Using Deep Network Designer and Train Using Image Observations” on page 5-152

## More About


- “Load Predefined Control System Environments” on page 2-23
- “Deep Q-Network (DQN) Agents” on page 3-23
- “Create Simulink Reinforcement Learning Environments” on page 2-8
- “Train Reinforcement Learning Agents” on page 5-3

# Train DDPG Agent to Swing Up and Balance Pendulum

This example shows how to train a deep deterministic policy gradient (DDPG) agent to swing up and balance a pendulum modeled in Simulink®.

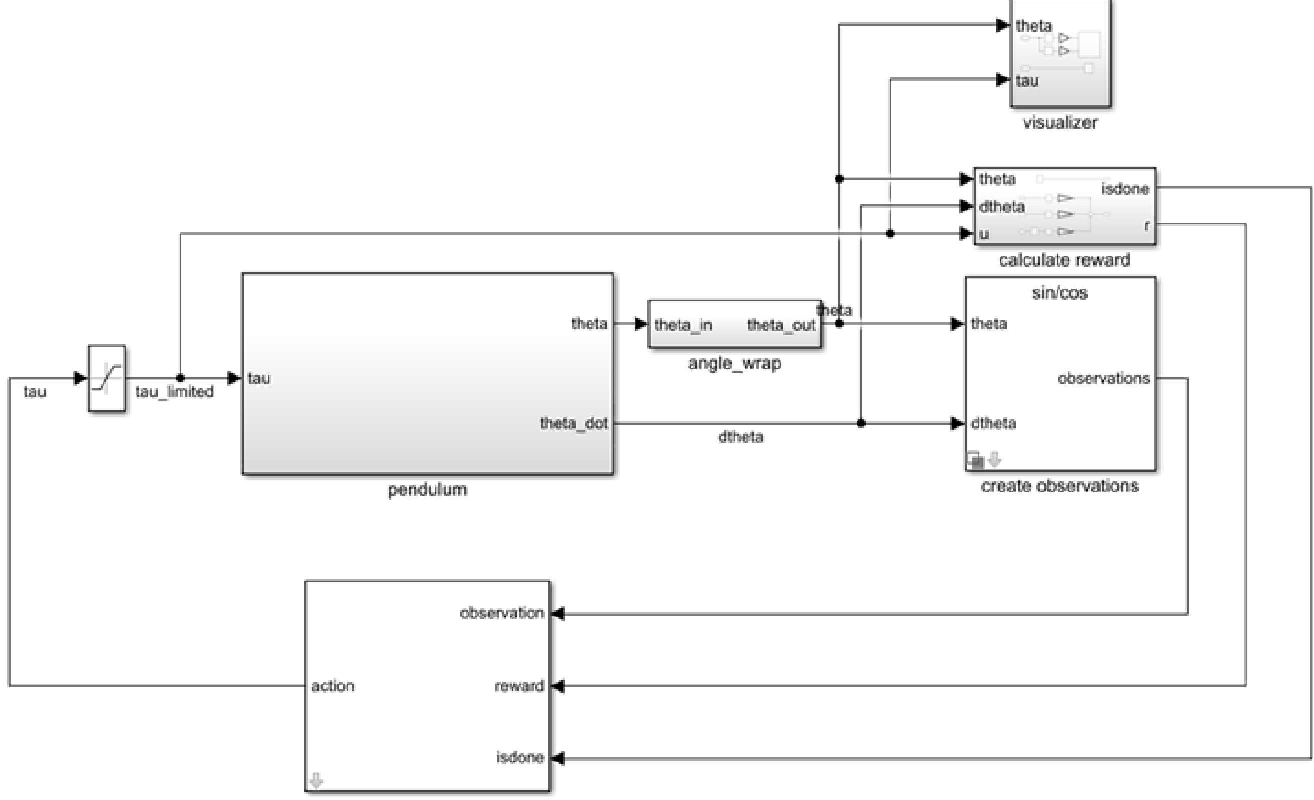
For more information on DDPG agents, see “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40. For an example that trains a DDPG agent in MATLAB®, see “Train DDPG Agent to Control Double Integrator System” on page 5-77.

### Pendulum Swing-Up Model

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

Open the model.

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




For this model:

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

- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are the sine of the pendulum angle, the cosine of the pendulum angle, and the pendulum angle derivative.
- The reward $r_t$, provided at every time step, is

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

Here:

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

For more information on this model, see “Load Predefined Simulink Environments” on page 2-30.

#### Create Environment Interface

Create a predefined environment interface for the pendulum.

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

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlSimplePendulumModel
      AgentBlock : rlSimplePendulumModel/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

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

The interface has a continuous action space where the agent can apply torque values between –2 to 2 N·m to the pendulum.

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

```
actInfo = 
  rlNumericSpec with properties:

     LowerLimit: -2
     UpperLimit: 2
           Name: "torque"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

Set the observations of the environment to be the sine of the pendulum angle, the cosine of the pendulum angle, and the pendulum angle derivative.

```
set_param( ...
    "rlSimplePendulumModel/create observations", ...
    "ThetaObservationHandling","sincos");
```

To define the initial condition of the pendulum as hanging downward, specify an environment reset function using an anonymous function handle. This reset function sets the model workspace variable `theta0` to `pi`.

```
env.ResetFcn = @(in)setVariable(in,"theta0",pi,"Workspace",mdl);
```

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

```
Ts = 0.05;
Tf = 20;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

#### Create DDPG Agent

DDPG agents use a parametrized Q-value function approximator to estimate the value of the policy. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for which receives the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value).

Define each network path as an array of layer objects and assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

For more information on creating a deep neural network value function representation, see “Create Policies and Value Functions” on page 4-2.

```
% Define state path
statePath = [
    featureInputLayer( ...
        obsInfo.Dimension(1), ...
        Name="obsPathInputLayer")
    fullyConnectedLayer(400)
    reluLayer
    fullyConnectedLayer(300,Name="spOutLayer")
    ];

% Define action path
actionPath = [
    featureInputLayer( ...
        actInfo.Dimension(1), ...
        Name="actPathInputLayer")
    fullyConnectedLayer(300, ...
        Name="apOutLayer", ...
        BiasLearnRateFactor=0)
    ];

% Define common path
commonPath = [
    additionLayer(2,Name="add")
    reluLayer
```

```
    fullyConnectedLayer(1)
    ];

% Create layergraph, add layers and connect them
criticNetwork = layerGraph();
criticNetwork = addLayers(criticNetwork,statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);
criticNetwork = connectLayers(criticNetwork,"spOutLayer","add/in1");
criticNetwork = connectLayers(criticNetwork,"apOutLayer","add/in2");
```

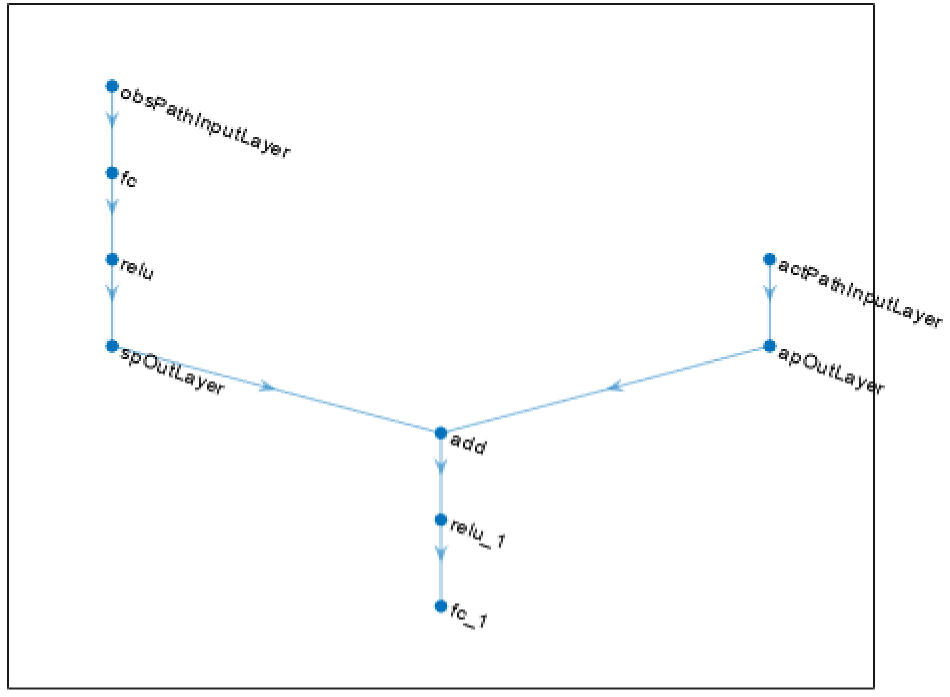
Convert to `dlnetwork` and display the number of weights.

```
criticNetwork = dlnetwork(criticNetwork);
summary(criticNetwork)
```

```
   Initialized: true

   Number of learnables: 122.8k

   Inputs:
      1   'obsPathInputLayer'   3 features
      2   'actPathInputLayer'   1 features
```

View the critic network configuration.

```
plot(criticNetwork)
```



Create the critic approximator object using `criticNet`, the environment observation and action specifications, and the names of the network input layers to be connected with the environment observation and action channels. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(criticNetwork, ...
    obsInfo,actInfo, ...
    ObservationInputNames="obsPathInputLayer", ...
    ActionInputNames="actPathInputLayer");
```

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is implemented by a continuous deterministic actor.

A continuous deterministic actor implements a parametrized deterministic policy for a continuous action space. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects. Since the output of the hyperbolic tangent layer is always between -1 and 1, use a scaling layer to scale it to the actual range of the action, as specified by `actInfo.UpperLimit`.

```
actorNetwork = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(400)
    reluLayer
    fullyConnectedLayer(300)
    reluLayer
    fullyConnectedLayer(1)
    tanhLayer
    scalingLayer(Scale=max(actInfo.UpperLimit))
    ];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNetwork = dlnetwork(actorNetwork);
summary(actorNetwork)
```

```
   Initialized: true

   Number of learnables: 122.2k

   Inputs:
      1   'input'   3 features
```

Create the actor using `actorNet` and the observation and action specifications. For more information on continuous deterministic actors, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNetwork,obsInfo,actInfo);
```

Specify options for the critic and actor using `rlOptimizerOptions`.

```
criticOpts = rlOptimizerOptions(LearnRate=1e-03,GradientThreshold=1);
actorOpts = rlOptimizerOptions(LearnRate=1e-04,GradientThreshold=1);
```

Specify the DDPG agent options using `rlDDPGAgentOptions`, include the training options for the actor and critic.

```
agentOpts = rlDDPGAgentOptions(...
    SampleTime=Ts,...
    CriticOptimizerOptions=criticOpts,...
    ActorOptimizerOptions=actorOpts,...
    ExperienceBufferLength=1e6,...
    DiscountFactor=0.99,...
    MiniBatchSize=128);
```

You can also modify the agent options using dot notation.

```
agentOpts.NoiseOptions.Variance = 0.6;
agentOpts.NoiseOptions.VarianceDecayRate = 1e-5;
```

Alternatively, you can create the agent first, and then access its option object and modify the options using dot notation.

Create the DDPG agent using the specified actor, critic, and agent options objects. For more information, see `rlDDPGAgent`.

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

### Train Agent

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

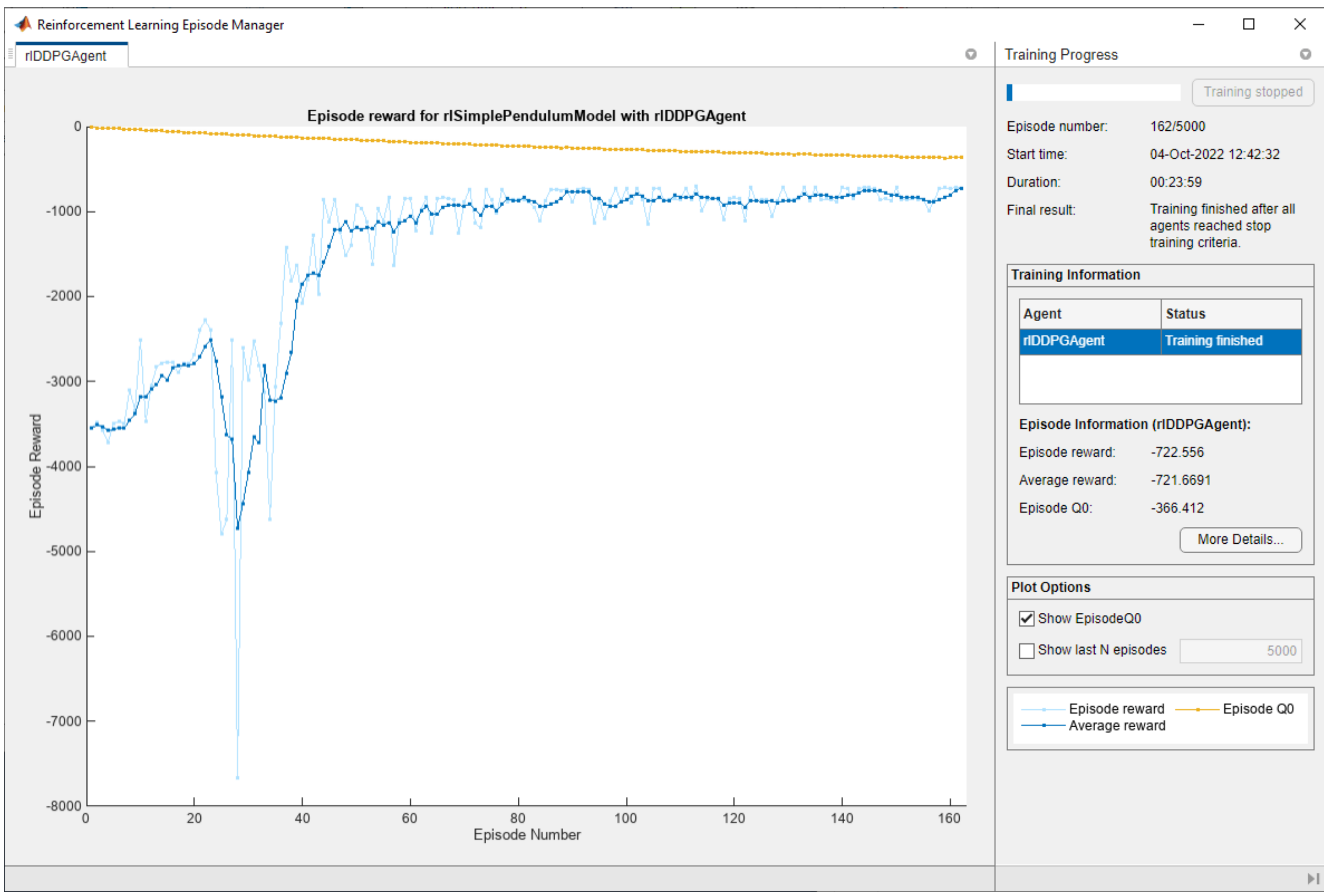
- Run training for at most 5000 episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –740 over five consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.
- Save a copy of the agent for each episode where the cumulative reward is greater than –740.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    ScoreAveragingWindowLength=5,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-740,...
    SaveAgentCriteria="EpisodeReward",...
    SaveAgentValue=-740);
```

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

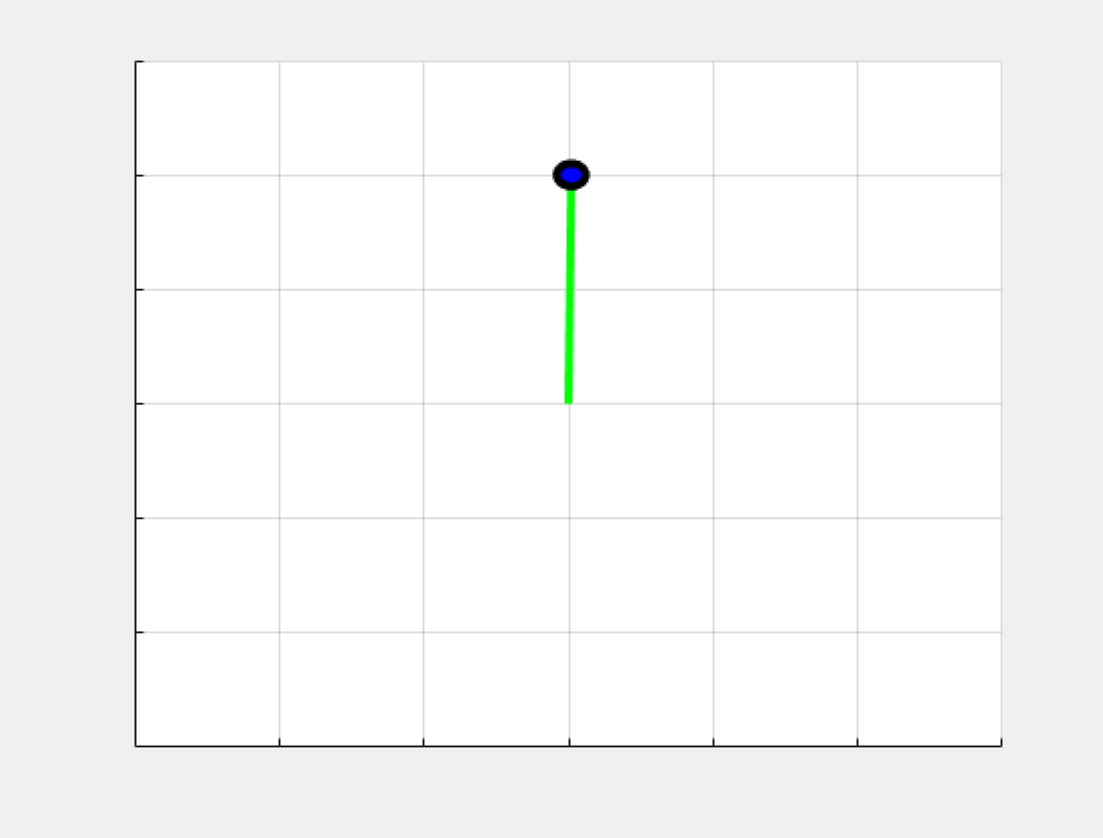
```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load("SimulinkPendulumDDPG.mat","agent")
end
```




## Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```



## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlQValueFunction` | `rlContinuousDeterministicActor` | `rlTrainingOptions` | `rlSimulationOptions` | `rlOptimizerOptions`

**Blocks**
RL Agent

**Blocks**
RL Agent

## Related Examples

- “Train DQN Agent to Swing Up and Balance Pendulum” on page 5-89
- “Train DDPG Agent to Swing Up and Balance Cart-Pole System” on page 5-106
- “Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal” on page 5-115

- “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation” on page 5-141

## More About

- “Load Predefined Simulink Environments” on page 2-30
- “Create Simulink Reinforcement Learning Environments” on page 2-8
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Train Reinforcement Learning Agents” on page 5-3

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

This example shows how to train a deep deterministic policy gradient (DDPG) agent to swing up and balance a cart-pole system modeled in Simscape™ Multibody™.

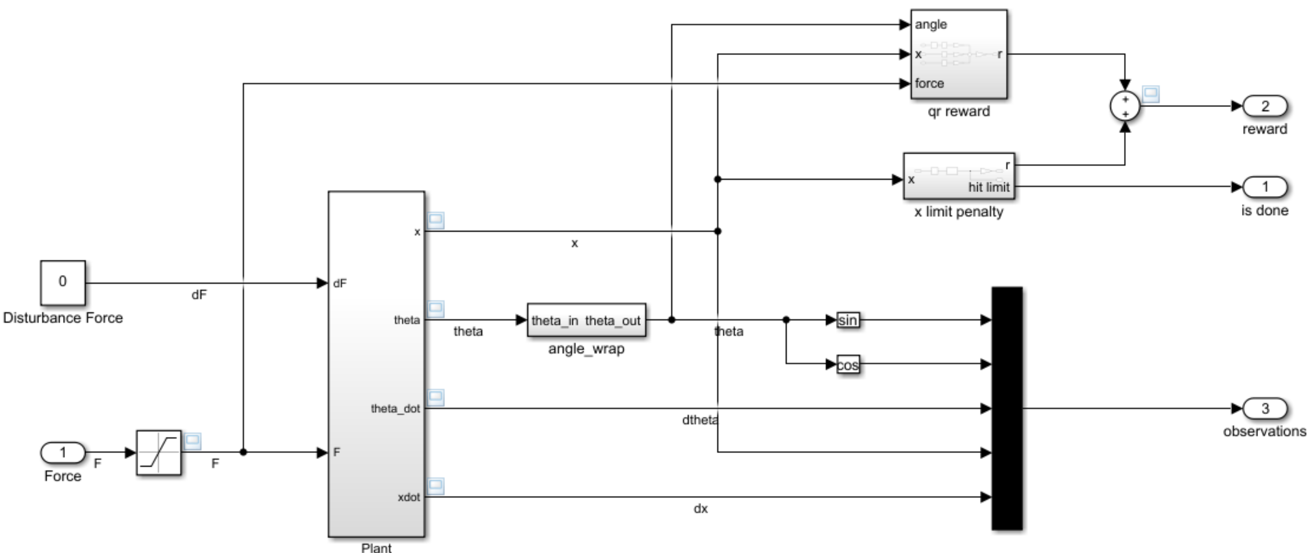
For more information on DDPG agents, see “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40. For an example showing how to train a DDPG agent in MATLAB®, see “Train DDPG Agent to Control Double Integrator System” on page 5-77.

### Cart-Pole Simscape Model

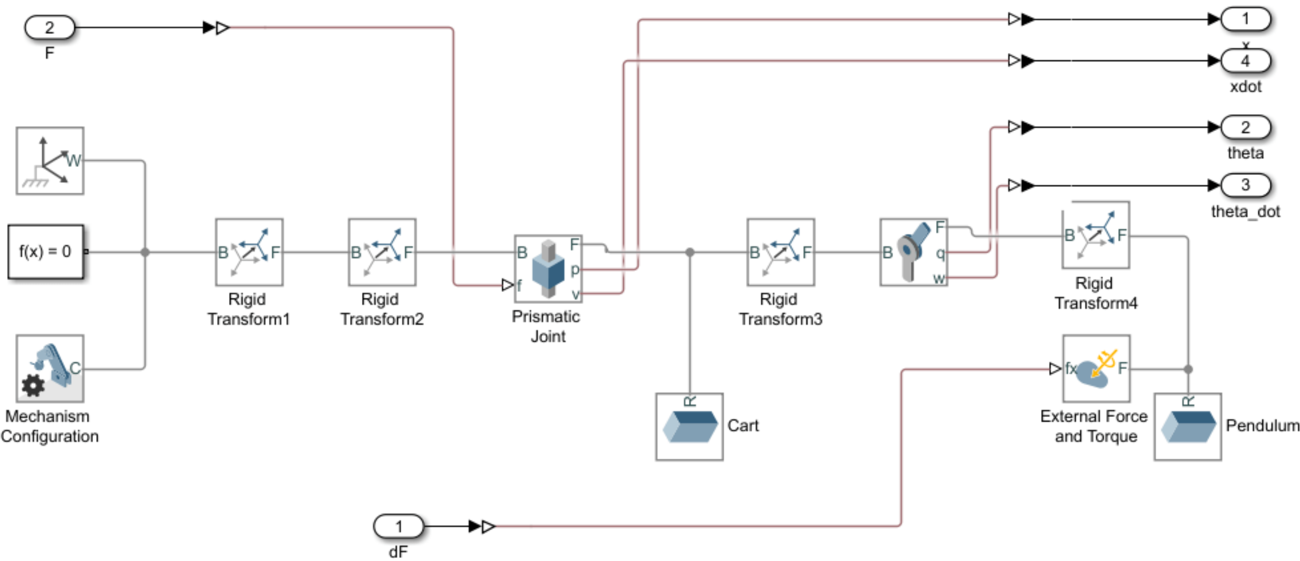
The reinforcement learning environment for this example is a pole attached to an unactuated joint on a cart, which moves along a frictionless track. The training goal is to make the pole stand upright without falling over using minimal control effort.

Open the model.

```
mdl = "rlCartPoleSimscapeModel";
open_system(mdl)
```




The cart-pole system is modeled using Simscape Multibody.




For this model:

- The upward balanced pole position is 0 radians, and the downward hanging position is `pi` radians.
- The force action signal from the agent to the environment is from –15 to 15 N.
- The observations from the environment are the position and velocity of the cart, and the sine, cosine, and derivative of the pole angle.
- The episode terminates if the cart moves more than 3.5 m from the original position.
- The reward $r_t$, provided at every timestep, is

$$\boldsymbol{r_t} = -0.1\left(5\theta_{\boldsymbol{t}}^{\;2} + x_{\boldsymbol{t}}^{\;2} + 0.05u_{t-1}^2\right) - 100B$$

Here:

- $\theta_t$ is the angle of displacement from the upright position of the pole.
- $x_t$ is the position displacement from the center position of the cart.
- $u_{t-1}$ is the control effort from the previous time step.
- $B$ is a flag (1 or 0) that indicates whether the cart is out of bounds.

For more information on this model, see “Load Predefined Simulink Environments” on page 2-30.

### Create Environment Interface

Create a predefined environment interface for the pole.

```
env = rlPredefinedEnv("CartPoleSimscapeModel-Continuous")
```

```
env = 
SimulinkEnvWithAgent with properties:

           Model : rlCartPoleSimscapeModel
```

```
      AgentBlock : rlCartPoleSimscapeModel/RL Agent
        ResetFcn : []
  UseFastRestart : on
```

The interface has a continuous action space where the agent can apply possible torque values from –15 to 15 N to the pole.

Obtain the observation and action information from the environment interface.

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

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

```
Ts = 0.02;
Tf = 25;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create DDPG Agent

DDPG agents use a parametrized Q-value function approximator to estimate the value of the policy. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for which receives the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` return the number of dimensions of the observation and action spaces, respectively, regardless of whether they are arranged as row vectors, column vectors, or matrices.

Define the network as an array of layer objects. Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

For more information on creating a deep neural network value function representation, see "Create Policies and Value Functions" on page 4-2.

```
% Define path for the state input
statePath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="NetObsInLayer")
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(200,Name="sPathOut")];

% Define path for the action input
actionPath = [
    featureInputLayer(prod(actInfo.Dimension),Name="NetActInLayer")
    fullyConnectedLayer(200,Name="aPathOut",BiasLearnRateFactor=0)];

% Define path for the critic output (value)
commonPath = [
```

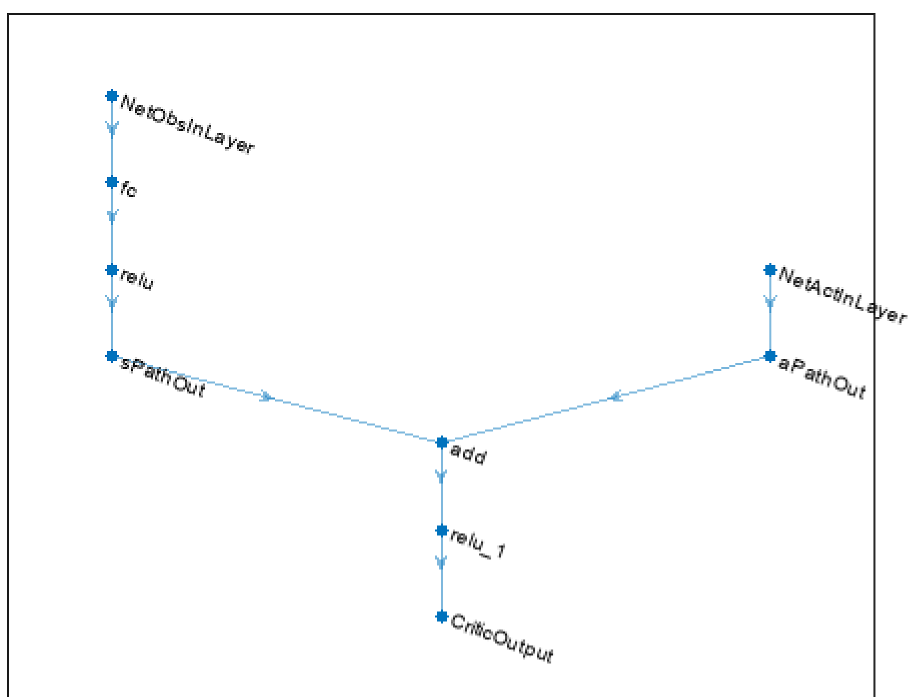
```
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(1,Name="CriticOutput")];

% Create layerGraph object and add layers
criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);

% Connect paths and convert to dlnetwork object
criticNetwork = connectLayers(criticNetwork,"sPathOut","add/in1");
criticNetwork = connectLayers(criticNetwork,"aPathOut","add/in2");
criticNetwork = dlnetwork(criticNetwork);
```

Display the number of weights and plot the network configuration.

```
summary(criticNetwork)
```

```
   Initialized: true

   Number of learnables: 27.1k

   Inputs:
      1   'NetObsInLayer'   5 features
      2   'NetActInLayer'   1 features
```

```
plot(criticNetwork)
```



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

```
critic = rlQValueFunction(criticNetwork, ...
    obsInfo,actInfo,...
    ObservationInputNames="NetObsInLayer", ...
    ActionInputNames="NetActInLayer");
```

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Since the output of `tanhLayer` is limited between -1 and 1, scale the network output to the range of the action using `scalingLayer`.

```
actorNetwork = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(200)
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension))
    tanhLayer
    scalingLayer(Scale=max(actInfo.UpperLimit))];
```

Convert to dlnetwork and display the number of weights.

```
actorNetwork = dlnetwork(actorNetwork);
summary(actorNetwork)
```

```
   Initialized: true

   Number of learnables: 26.7k

   Inputs:
      1   'input'   5 features
```

Create the actor in a similar manner to the critic. For more information, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNetwork,obsInfo,actInfo);
```

Specify training options for the critic and the actor using `rlOptimizerOptions`.

```
criticOptions = rlOptimizerOptions(LearnRate=1e-03,GradientThreshold=1);
actorOptions = rlOptimizerOptions(LearnRate=5e-04,GradientThreshold=1);
```

Specify the DDPG agent options using `rlDDPGAgentOptions`, and include the training options for the actor and critic.

```
agentOptions = rlDDPGAgentOptions(...
    SampleTime=Ts,...
    ActorOptimizerOptions=actorOptions,...
```

```
    CriticOptimizerOptions=criticOptions,...
    ExperienceBufferLength=1e6,...
    MiniBatchSize=128);
```

You can also modify the agent options using dot notation.

```
agentOptions.NoiseOptions.Variance = 0.4;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Alternatively, you can also create the agent first, and then access its option object and modify the options using dot notation.

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

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

### Train Agent

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

- Run each training episode for at most 2000 episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –400 over five consecutive episodes. At this point, the agent can quickly balance the pole in the upright position using minimal control effort.
- Save a copy of the agent for each episode where the cumulative reward is greater than –400.

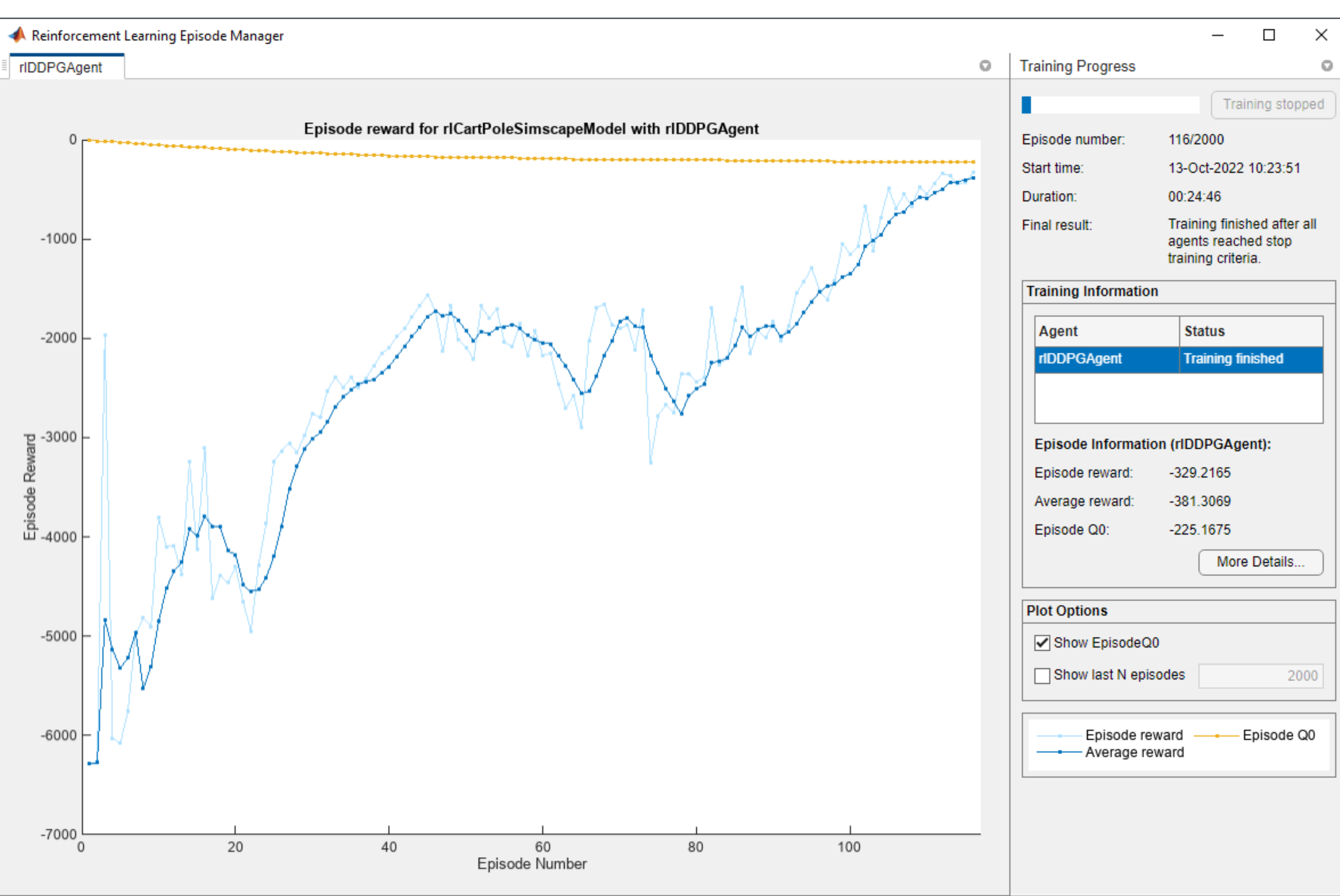
For more information, see `rlTrainingOptions`.

```
maxepisodes = 2000;
maxsteps = ceil(Tf/Ts);
trainingOptions = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    ScoreAveragingWindowLength=5,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-400,...
    SaveAgentCriteria="EpisodeReward",...
    SaveAgentValue=-400);
```

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

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
    % Load the pretrained agent for the example.
```

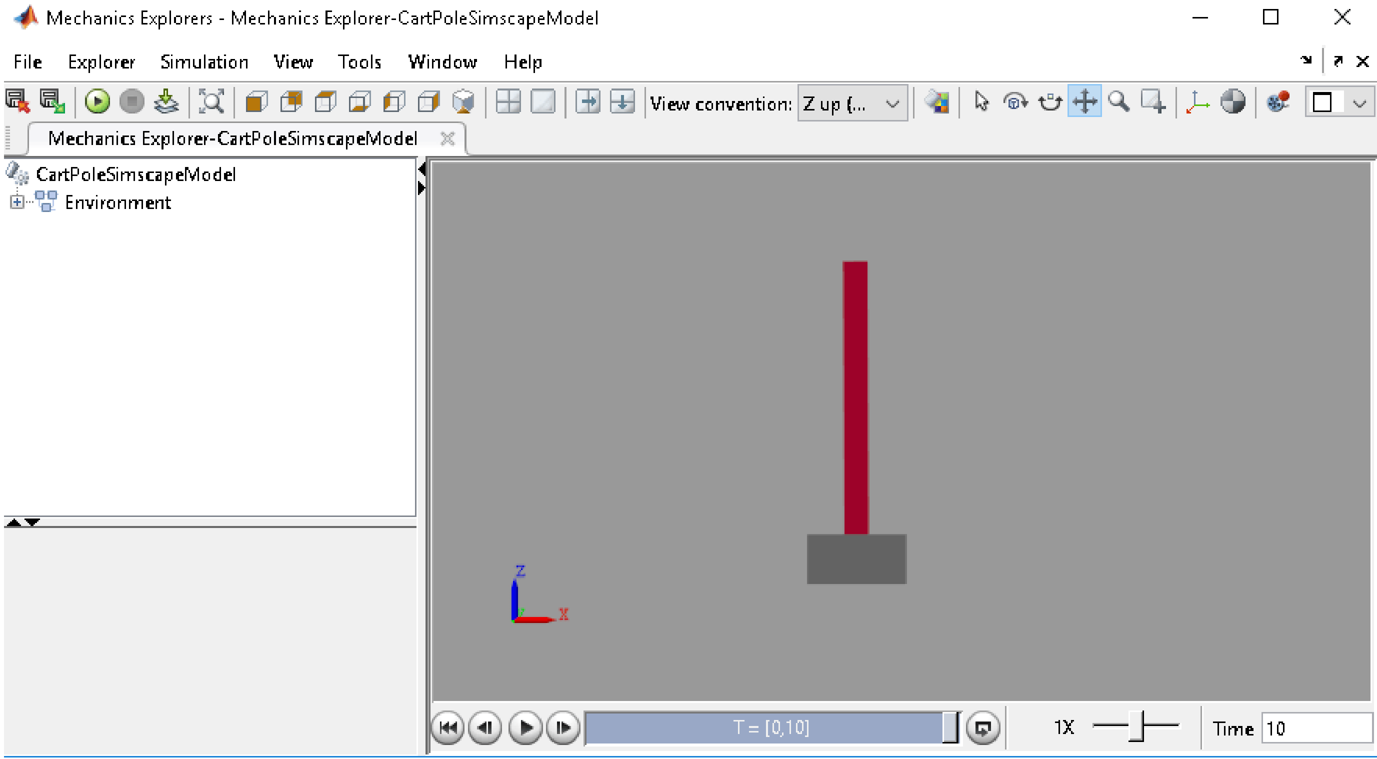
```
    load("SimscapeCartPoleDDPG.mat","agent")
end
```




## Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




```
bdclose(mdl)
```

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlQValueFunction` | `rlContinuousDeterministicActor` | `rlTrainingOptions` | `rlSimulationOptions` | `rlOptimizerOptions`

**Blocks**
RL Agent

## Related Examples

- “Train PG Agent to Balance Cart-Pole System” on page 5-57
- “Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal” on page 5-115
- “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation” on page 5-141

## More About


- “Load Predefined Simulink Environments” on page 2-30
- “Create Policies and Value Functions” on page 4-2
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Train Reinforcement Learning Agents” on page 5-3

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

This example shows how to convert a simple frictionless pendulum Simulink® model to a reinforcement learning environment interface, and how to train a deep deterministic policy gradient (DDPG) agent in this environment.

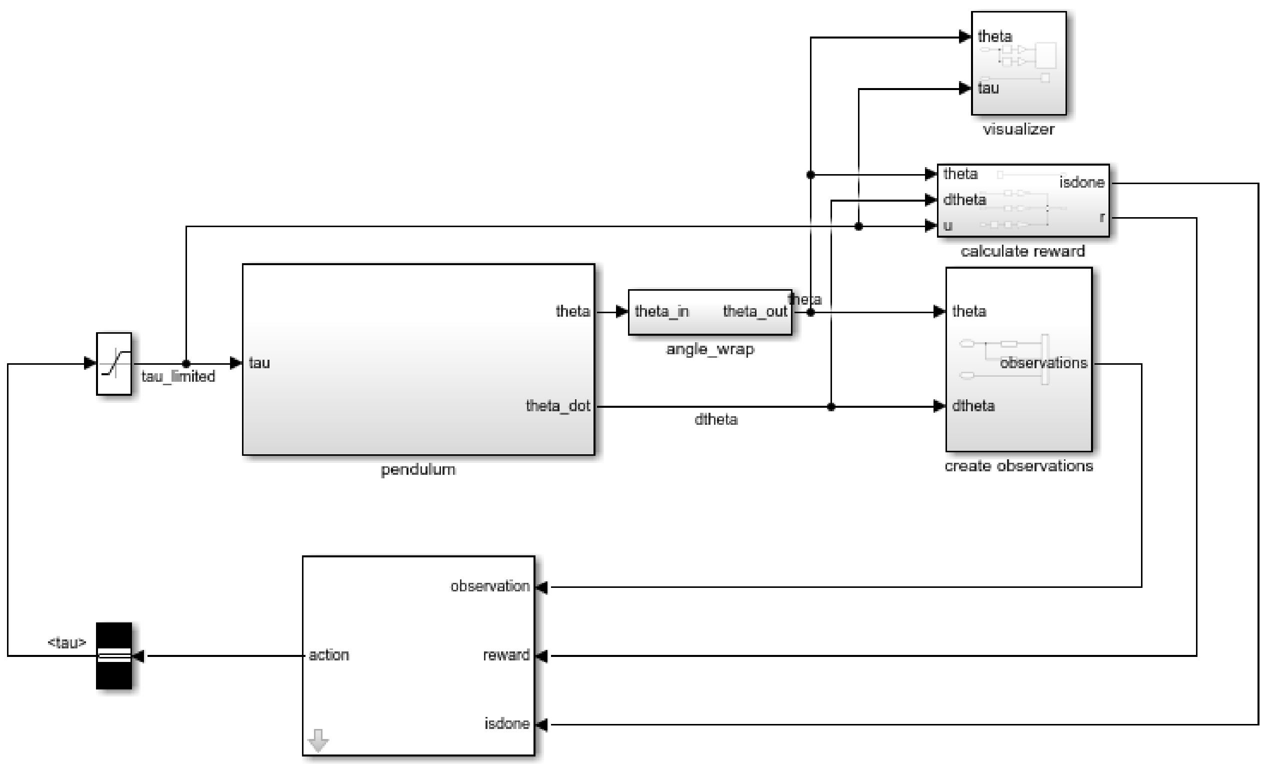
For more information on DDPG agents, see “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40. For an example showing how to train a DDPG agent in MATLAB®, see “Train DDPG Agent to Control Double Integrator System” on page 5-77.

### Pendulum Swing-Up Model with Bus

The starting model for this example is a simple frictionless pendulum. The training goal is to make the pendulum stand upright without falling over using minimal control effort.

Open the model.

```
mdl = "rlSimplePendulumModelBus";
open_system(mdl)
```




For this model:

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

- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are the sine of the pendulum angle, the cosine of the pendulum angle, and the pendulum angle derivative.
- Both the observation and action signals are Simulink buses.
- The reward $r_t$, provided at every time step, is

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

Here:

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

The model used in this example is similar to the simple pendulum model described in "Load Predefined Simulink Environments" on page 2-30. The difference is that the model in this example uses Simulink buses for the action and observation signals.

**Create Environment Interface with Bus**

The environment interface from a Simulink model is created using `rlSimulinkEnv`, which requires the name of the Simulink model, the path to the agent block, and observation and action reinforcement learning data specifications. For models that use bus signals for actions or observations, you can create the corresponding specifications using the `bus2RLSpec` function.

Specify the path to the agent block.

```
agentBlk = "rlSimplePendulumModelBus/RL Agent";
```

Create the observation `Bus` object. The channel names must correspond to the signal names specified in the Simulink buses used to create the observation and action signals.

```
obsBus = Simulink.Bus();
obs(1) = Simulink.BusElement;
obs(1).Name = "sin_theta";
obs(2) = Simulink.BusElement;
obs(2).Name = "cos_theta";
obs(3) = Simulink.BusElement;
obs(3).Name = "dtheta";
obsBus.Elements = obs;
```

Create the action `Bus` object.

```
actBus = Simulink.Bus();
act(1) = Simulink.BusElement;
act(1).Name = "tau";
act(1).Min = -2;
act(1).Max = 2;
actBus.Elements = act;
```

Create the action and observation specification objects using the Simulink buses.

```
obsInfo = bus2RLSpec("obsBus","Model",mdl)
```

```
obsInfo=1×3 object
  1x3 rlNumericSpec array with properties:

    LowerLimit
    UpperLimit
    Name
    Description
    Dimension
    DataType
```

```
actInfo = bus2RLSpec("actBus","Model",mdl)
```

```
actInfo = 
  rlNumericSpec with properties:

     LowerLimit: -2
     UpperLimit: 2
           Name: "tau"
    Description: ""
      Dimension: [1 1]
       DataType: "double"
```

Create the reinforcement learning environment for the pendulum model.

```
env = rlSimulinkEnv(mdl,agentBlk,obsInfo,actInfo);
```

To define the initial condition of the pendulum as hanging downward, specify an environment reset function using an anonymous function handle. This reset function sets the model workspace variable `theta0` to `pi`.

```
env.ResetFcn = @(in)setVariable(in,"theta0",pi,"Workspace",mdl);
```

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

```
Ts = 0.05;
Tf = 20;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

#### Create DDPG Agent

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with three input layers (each receiving one scalar observation, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects, and combine the outputs from the input layes using a `concatenationLayer`. Since the output of `tanhLayer` is limited between -1 and 1, scale the network output to the range of the action using `scalingLayer` .

For more information on creating actors based on deep neural networks, see “Create Policies and Value Functions” on page 4-2.

```
% Define input layers
sinThetaInputLayer = featureInputLayer(1,Name="sin_th_lyr");
cosThetaInputLayer = featureInputLayer(1,Name="cos_th_lyr");
dThetaInputLayer   = featureInputLayer(1,Name="dth_lyr");

% Define common path
commonPath = [
    concatenationLayer(1,3,Name="concat")
    fullyConnectedLayer(400)
    reluLayer
    fullyConnectedLayer(300)
    reluLayer
    fullyConnectedLayer(1)
    tanhLayer
    scalingLayer(Scale=max(actInfo.UpperLimit))];

% Create layergraph object and add layers
actorNetwork = layerGraph(sinThetaInputLayer);
actorNetwork = addLayers(actorNetwork,cosThetaInputLayer);
actorNetwork = addLayers(actorNetwork,dThetaInputLayer);
actorNetwork = addLayers(actorNetwork,commonPath);

% Connect layers
actorNetwork = connectLayers(actorNetwork,"sin_th_lyr","concat/in1");
actorNetwork = connectLayers(actorNetwork,"cos_th_lyr","concat/in2");
actorNetwork = connectLayers(actorNetwork,"dth_lyr","concat/in3");
```

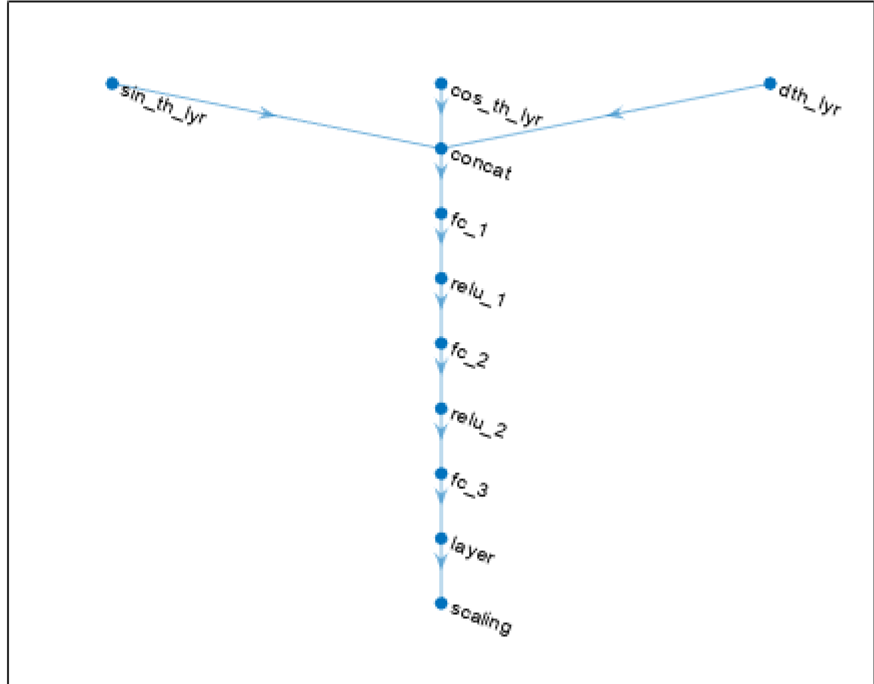
Convert to a `dlnetwork` object and display the number of weights.

```
actorNetwork = dlnetwork(actorNetwork);
summary(actorNetwork)
```

```
   Initialized: true

   Number of learnables: 122.2k

   Inputs:
      1   'sin_th_lyr'   1 features
      2   'cos_th_lyr'   1 features
      3   'dth_lyr'      1 features
```

View the actor network configuration.

```
figure
plot(layerGraph(actorNetwork))
```




Create the actor using the specified deep neural network. Specify the action and observation info for the actor, as well as the names of the network input layers to be connected with the observation channels. For more information, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNetwork,obsInfo,actInfo,...
    "ObservationInputNames",["sin_th_lyr","cos_th_lyr","dth_lyr"]);
```

DDPG agents use a parametrized Q-value function approximator to estimate the value of the policy. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for which receives the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with four input layers (three for the observation channels, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value).

Define each network path as an array of layer objects, assigning names to the input and output layers of each path. Use the input layers previously defined for the actor.

```
% Observation path
obsPath = [
    concatenationLayer(1,3,Name="concat")
    fullyConnectedLayer(400)
    reluLayer
    fullyConnectedLayer(300,Name="obsPathOutLyr")
    ];
```

```
% Action path
actPath = [
    featureInputLayer(1,Name= "action")
    fullyConnectedLayer(300, ...
        Name="actPathOutLyr", ...
        BiasLearnRateFactor=0)
        ];

% Common path
commonPath = [
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(1,Name="CriticOutput")
    ];

% Create layergraph object and add layers
criticNetwork = layerGraph(sinThetaInputLayer);
criticNetwork = addLayers(criticNetwork,cosThetaInputLayer);
criticNetwork = addLayers(criticNetwork,dThetaInputLayer);
criticNetwork = addLayers(criticNetwork,actPath);
criticNetwork = addLayers(criticNetwork,obsPath);
criticNetwork = addLayers(criticNetwork,commonPath);

% Connect Layers
criticNetwork = connectLayers(criticNetwork,"sin_th_lyr","concat/in1");
criticNetwork = connectLayers(criticNetwork,"cos_th_lyr","concat/in2");
criticNetwork = connectLayers(criticNetwork,"dth_lyr","concat/in3");
criticNetwork = connectLayers(criticNetwork,"obsPathOutLyr","add/in1");
criticNetwork = connectLayers(criticNetwork,"actPathOutLyr","add/in2");
```

Convert to `dlnetwork` and display the number of weights.

```
net = dlnetwork(criticNetwork);
%summary(criticNetwork)
```

Create the critic approximator object using `criticNet`, the environment observation and action specifications, and the names of the network input layers to be connected with the environment observation and action channels. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(criticNetwork, ...
    obsInfo,actInfo,...
    ObservationInputNames=["sin_th_lyr","cos_th_lyr","dth_lyr"], ...
    ActionInputNames="action");
```

Specify options for the critic representation using `rlOptimizerOptions`.

```
actorOptions = rlOptimizerOptions(LearnRate=1e-4,GradientThreshold=1);
criticOptions = rlOptimizerOptions(LearnRate=1e-03,GradientThreshold=1);
```

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

```
agentOpts = rlDDPGAgentOptions(...
    SampleTime=Ts,...
    ActorOptimizerOptions=actorOptions,...
    CriticOptimizerOptions=criticOptions,...
    ExperienceBufferLength=1e6,...
    MiniBatchSize=128);
```

```
agentOpts.NoiseOptions.Variance = 0.6;
agentOpts.NoiseOptions.VarianceDecayRate = 1e-5;
```

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

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

### Train Agent

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

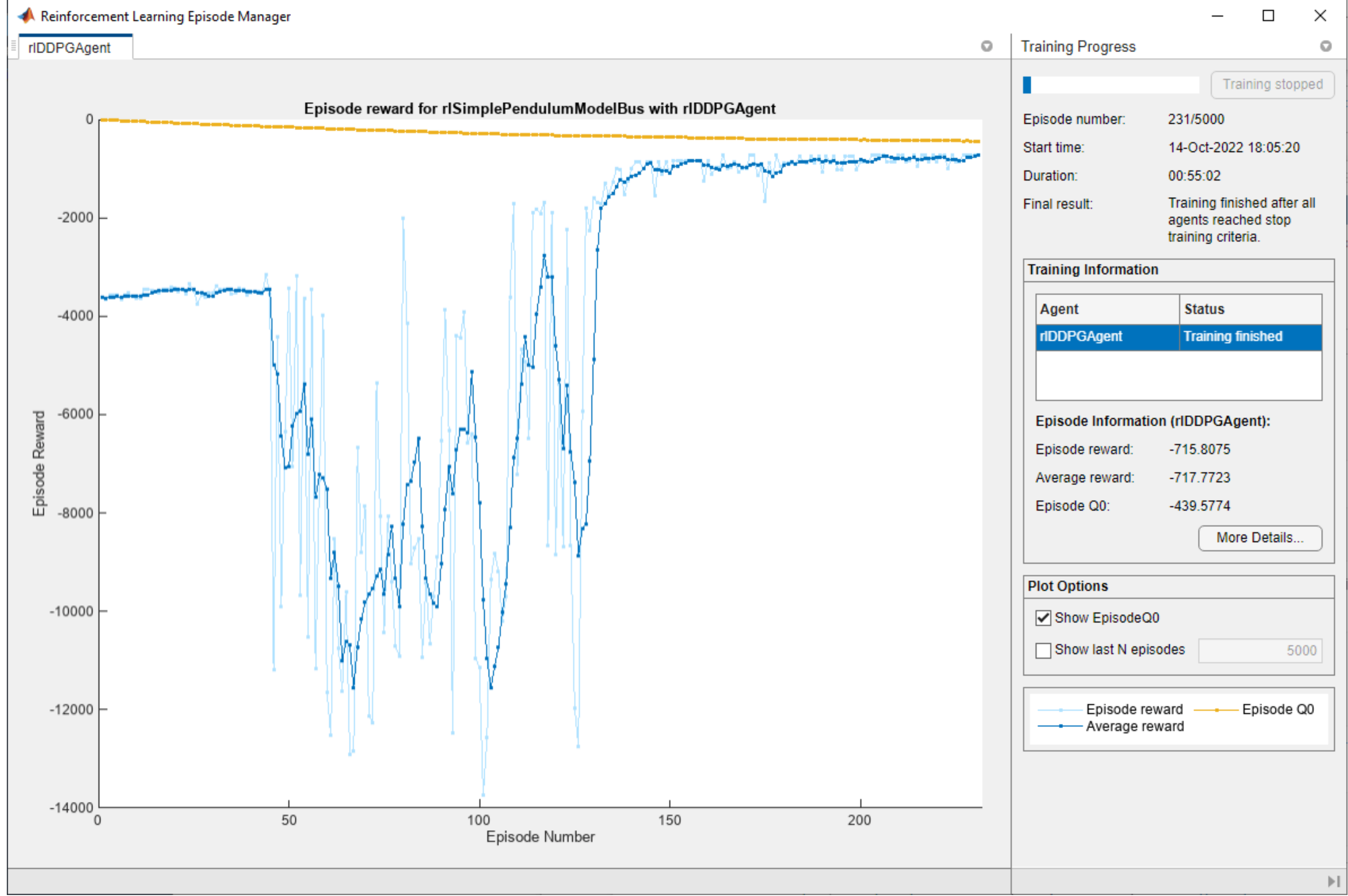
- Run each training for at most 50000 episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –740 over five consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.
- Save a copy of the agent for each episode where the cumulative reward is greater than –740.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    ScoreAveragingWindowLength=5,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-740);
```

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

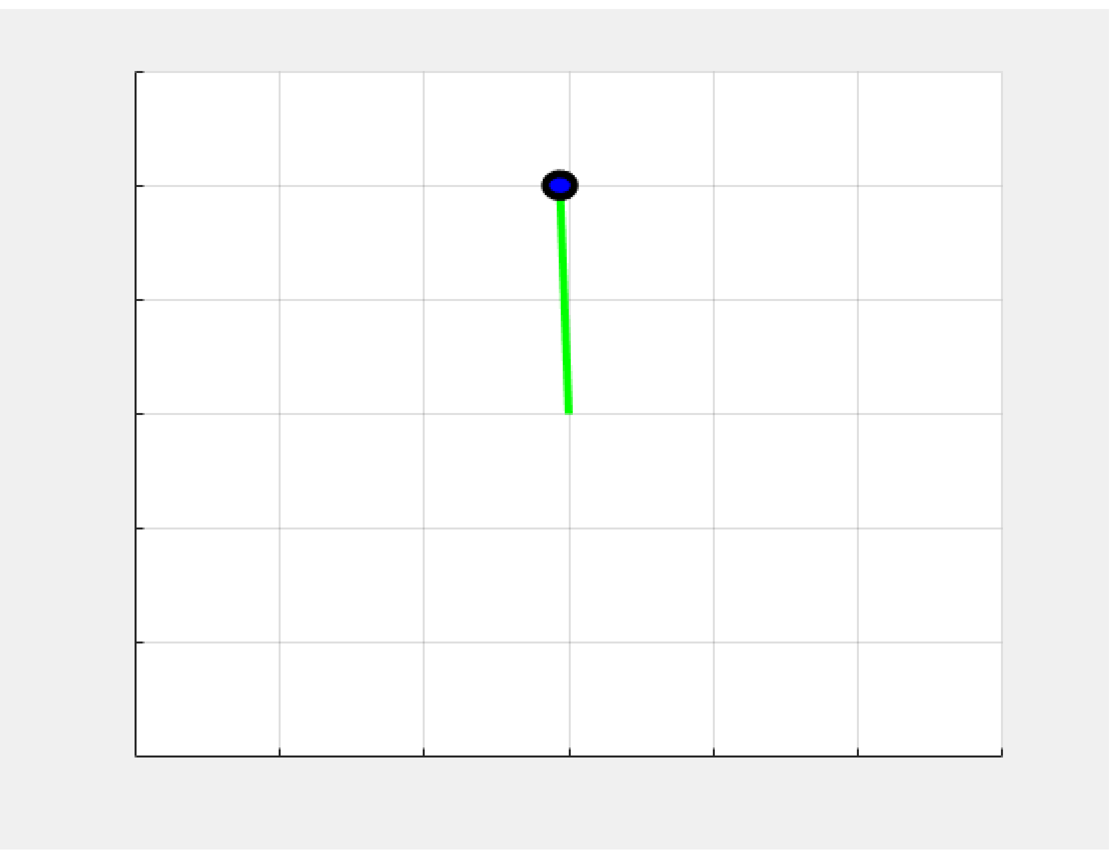
```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load("SimulinkPendBusDDPG.mat","agent")
end
```




### Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```



## See Also

**Functions**
rlSimulinkEnv | bus2RLSpec | train | sim

**Objects**
rlDDPGAgent | rlDDPGAgentOptions | rlQValueFunction | rlContinuousDeterministicActor | rlTrainingOptions | rlSimulationOptions | rlOptimizerOptions

**Blocks**
RL Agent

## Related Examples

- "Train DQN Agent to Swing Up and Balance Pendulum" on page 5-89
- "Train DDPG Agent to Swing Up and Balance Pendulum" on page 5-97
- "Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation" on page 5-141

## More About

- "Create Simulink Reinforcement Learning Environments" on page 2-8
- "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40

- "Train Reinforcement Learning Agents" on page 5-3

# Train Reinforcement Learning Agents to Control Quanser QUBE Pendulum

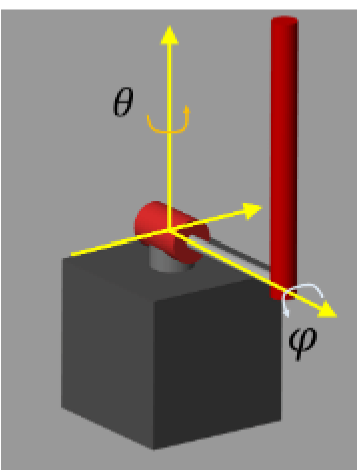
This example trains reinforcement learning (RL) agents to swing up and control a Quanser QUBE™-Servo 2 inverted pendulum system.

### Inverted Pendulum Model

The Quanser QUBE-Servo 2 pendulum system is an implementation of a rotational inverted pendulum. It consists of a motor arm, which is actuated by a DC servo motor, with a freely swinging pendulum arm attached to its end. The system is challenging to control because it is underactuated, nonminimum phase, and highly nonlinear [1].

In this example, the pendulum system is modeled in Simulink® using Simscape™ Electrical™ and Simscape Multibody™ components. For this system:

- $\theta$ is the motor arm angle and $\varphi$ is the pendulum angle.
- The motor arm angle is 0 radians when the arm is oriented horizontal and forward as shown in the diagram (counterclockwise is positive).
- The pendulum angle is 0 radians when the pendulum is oriented vertically downwards (counterclockwise is positive).
- Input of the plant model is a DC voltage signal for the motor. The voltage value ranges from –12 to 12 V.
- The pendulum and motor angles and angular velocities are measured by sensors.

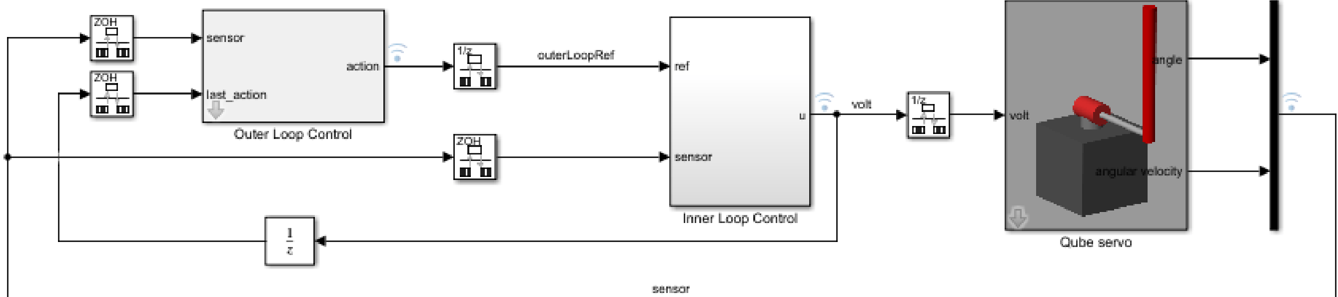



Load the parameters for this example using the `loadQubeParameters` helper script.

```
loadQubeParameters
```

Open the Simulink model.

```
mdl = "rlQubeServo";
open_system(mdl)
```



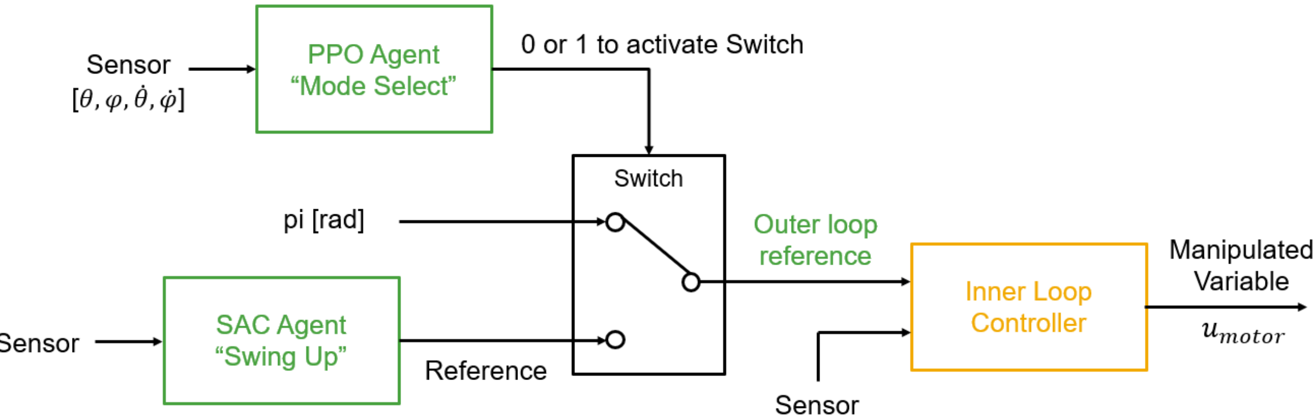

### Control Structure

In this example, the RL agents generate reference trajectories, which are then passed to an inner control loop.

### Outer-Loop Components

The outer loop of the control architecture injects the pendulum angle reference signal to the inner loop. It consists of the following reinforcement learning agents.

- Swing-up agent — A soft actor-critic (SAC) agent that computes reference angles for swinging up the pendulum arm.
- Mode-select agent — A proximal policy optimization (PPO) agent that performs a mode switching operation when the pendulum angle is close to the upright position ($\pi \pm \pi/6$ radians).

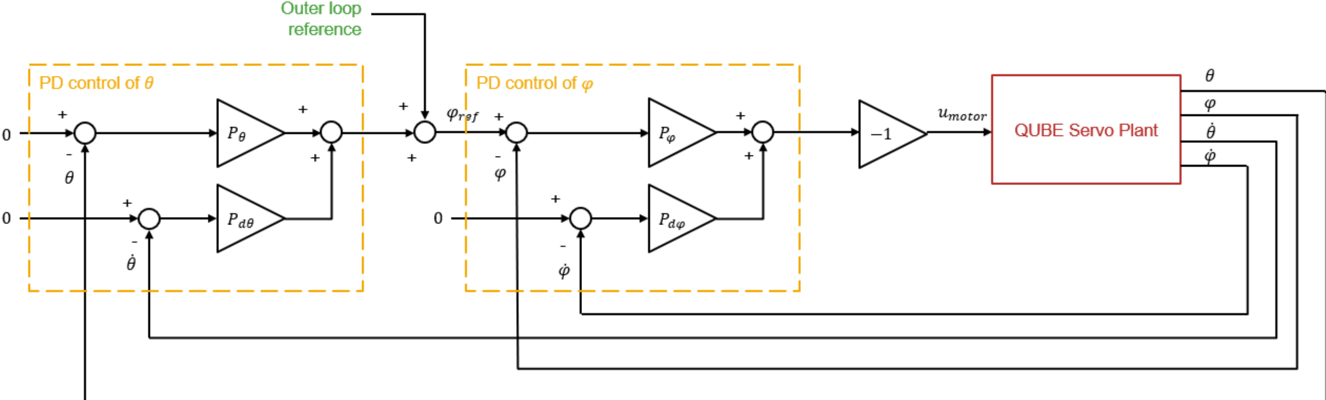



The mode-switching action (0 or 1) switches the outer-loop reference signal between the trajectory generated by the swing-up agent action and $\pi$ radians.

### Inner-Loop Components

The inner-loop components compute the low-level control input $u_{\text{motor}}$ (voltage) to stabilize the pendulum at the upright equilibrium point where the system is linearizable. Two proportional-

derivative (PD) controllers form the inner-loop control system as shown in the following figure. For this example, the gains $P_\theta$, $P_{d\theta}$, $P_\varphi$, and $P_{d\varphi}$ were tuned to 0.1620, 0.0356, 40, and 2, respectively.




### Train Swing-Up Agent

The swing-up agent in this example is modeled using the soft actor-critic (SAC) algorithm. For this agent:

- The environment is the pendulum system with the low-level controller. The mode-selection signal is always set to `1`.
- The observation is the vector $\left[\sin\theta, \cos\theta, \sin\varphi, \cos\varphi, \dot{\theta}, \dot{\varphi}\right]$.
- The action is the normalized outer-loop pendulum reference angle.
- The reward signal is as follows:

$$r = -\theta^2 - 0.1(\varphi - \theta)^2 - 0.1\dot{\varphi}^2 + F$$
$$F = \begin{cases} 100 & \theta \in \pi \pm \pi/6 \text{ radians and } \varphi \in \pm \pi \text{ radians} \\ 0 & \text{otherwise} \end{cases}$$

Open the Outer Loop Control block and set the **Design mode** parameter to `Swing up`. Doing so sets the mode-selection action to `1`, which configures the Switch block to pass the swing-up reference to the inner-loop controllers.

Alternatively, you can set this parameter using the following command.

```
set_param(mdl + "/Outer Loop Control","DesignModeChoice","Swing up");
```

Set the random seed for reproducibility.

```
rng(0)
```

Create the input and output specifications for the agent.

```
swingObsInfo = rlNumericSpec([6,1]);
swingActInfo = rlNumericSpec([1,1],LowerLimit=-1,UpperLimit=1);
```

Create the environment interface.

```
swingAgentBlk = ...
    mdl + "/Outer Loop Control/RL_Swing_Up/RL_agent_swing_up";
swingEnv = rlSimulinkEnv(mdl,swingAgentBlk,swingObsInfo,swingActInfo);
```

The agent trains from an experience buffer of maximum capacity `1e6` by randomly selecting mini-batches of size `128`. The discount factor of `0.99` is close to `1` and therefore favors long term reward with respect to a smaller value. For a full list of SAC hyperparameters and their descriptions, see `rlSACAgent`. Specify the agent hyperparameters for training.

```
swingAgentOpts = rlSACAgentOptions(...
    SampleTime=Ts,...
    TargetSmoothFactor=1e-3,...
    ExperienceBufferLength=1e6,...
    DiscountFactor=0.99,...
    MiniBatchSize=128);
```

The actor and critic neural networks of the swing-up agent are updated by the Adam (adaptive moment estimation) optimizer with the following configuration. Specify the optimizer options.

```
swingAgentOpts.ActorOptimizerOptions.Algorithm = "adam";
swingAgentOpts.ActorOptimizerOptions.LearnRate = 1e-4;
swingAgentOpts.ActorOptimizerOptions.GradientThreshold = 1;
for ct = 1:2
    swingAgentOpts.CriticOptimizerOptions(ct).Algorithm = "adam";
    swingAgentOpts.CriticOptimizerOptions(ct).LearnRate = 1e-3;
    swingAgentOpts.CriticOptimizerOptions(ct).GradientThreshold = 1;
    swingAgentOpts.CriticOptimizerOptions(ct).L2RegularizationFactor = 2e-4;
end
```

Create the agent.

```
initOptions = rlAgentInitializationOptions(NumHiddenUnit=300);
swingAgent = rlSACAgent(swingObsInfo,swingActInfo,initOptions,swingAgentOpts);
```

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

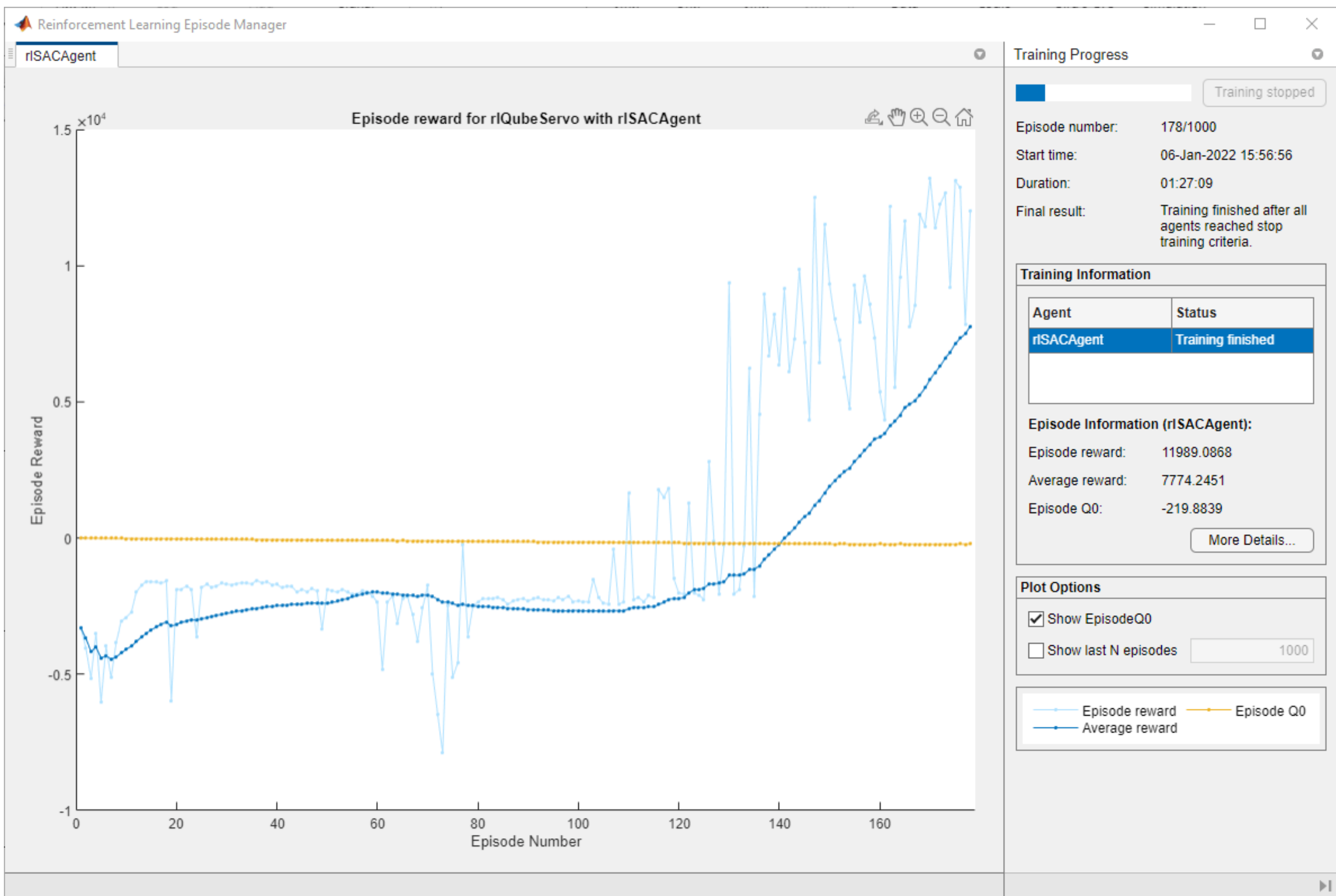
- Run each training for at most `1000` episodes, with each episode lasting at most `floor(Tf/Ts)` time steps.
- Stop training when the agent receives an average cumulative reward greater than `7500` over `50` consecutive episodes.

```
swingTrainOpts = rlTrainingOptions(...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=floor(Tf/Ts),...
    ScoreAveragingWindowLength=50,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=7500);
```

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

```
doSwingTraining = false;
if doSwingTraining
```

```
    swingTrainResult = train(swingAgent,swingEnv,swingTrainOpts);
else
    load("rlQubeServoAgents.mat","swingAgent");
end
```




## Train Mode-Select Agent

The mode-select agent in this example is modeled using the proximal policy optimization (PPO) algorithm. For this agent:

- The environment is the pendulum system with the low-level controller and the swing-up agent.
- The observation is the vector $\left[\sin\theta,\ \cos\theta,\ \sin\varphi,\ \cos\varphi,\ \dot{\theta},\ \dot{\varphi}\right]$.
- The action is `0` or `1` which determines which reference signal is sent to the PD controller.
- The reward signal is as follows:

$$r = -\theta^2 + G$$

$$G = \begin{cases} 1 & \theta \in \pi \pm \pi/6 \text{ radians} \\ 0 & \text{otherwise} \end{cases}$$

Open the Outer Loop Control block and set the **Design mode** parameter to `Mode select`.

Alternatively, you can set this parameter using the following command.

```
set_param(mdl + "/Outer Loop Control","DesignModeChoice","Mode select");
```

Create the input and output specifications for the agent.

```
modeObsInfo = rlNumericSpec([6,1]);
modeActInfo = rlFiniteSetSpec({0,1});
```

Create the environment.

```
modeAgentBlk = ...
    mdl + "/Outer Loop Control/RL_Mode_Select/RL_agent_select_mode";
modeEnv = rlSimulinkEnv(mdl,modeAgentBlk,modeObsInfo,modeActInfo);
```

The mode-select agent trains by first collecting trajectories up to the experience horizon of `floor(Tf/Ts)` steps. It then learns from the trajectory data using a mini-batch size of `128`. The discount factor of `0.99` favors long-term reward and an entropy loss weight of `1e-4` facilities exploration during training.

Specify the hyperparameters for the agent. For more information on PPO agent options, see `rlPPOAgentOptions`.

```
modeAgentOpts = rlPPOAgentOptions(...
    SampleTime=Ts,...
    DiscountFactor=0.99,...
    ExperienceHorizon=floor(Tf/Ts), ...
    MiniBatchSize=500, ...
    EntropyLossWeight=1e-4);
```

The actor and critic neural networks of the mode-select agent are updated by the Adam optimizer with the following configuration. Specify the optimizer options.

```
modeAgentOpts.ActorOptimizerOptions.LearnRate = 1e-4;
modeAgentOpts.ActorOptimizerOptions.GradientThreshold = 1;
modeAgentOpts.CriticOptimizerOptions.LearnRate = 1e-4;
```

Create the agent.

```
initOptions = rlAgentInitializationOptions(NumHiddenUnit=300);
modeAgent = rlPPOAgent(modeObsInfo,modeActInfo,initOptions,modeAgentOpts);
```

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

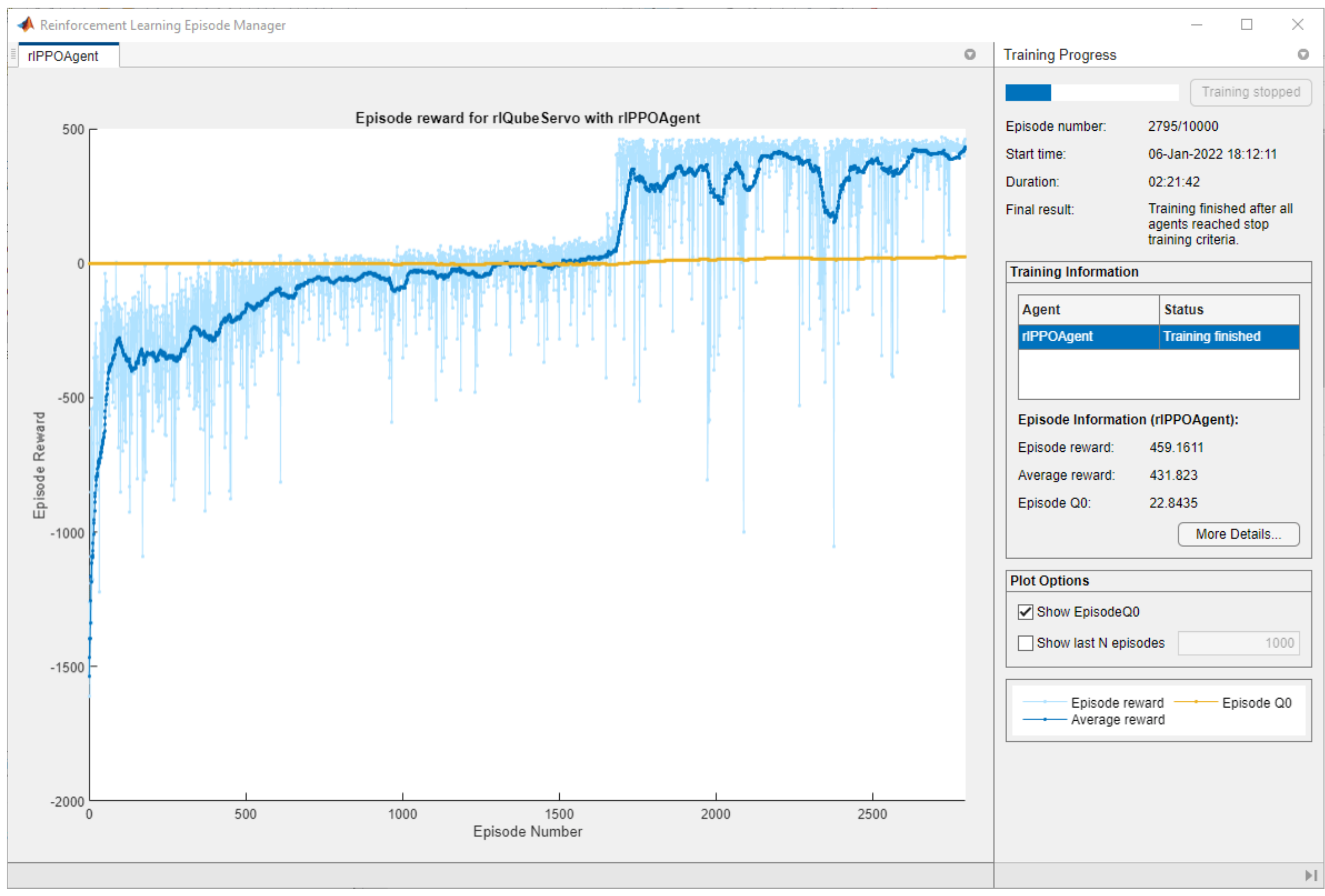
- Run each training for at most `1000` episodes with each episode lasting at most `floor(Tf/Ts)` time steps.
- Stop training when the agent receives an average cumulative reward greater than `430` over `50` consecutive episodes.

```
modeTrainOpts = rlTrainingOptions(...
    MaxEpisodes=10000,...
    MaxStepsPerEpisode=floor(Tf/Ts),...
    ScoreAveragingWindowLength=50,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=430);
```

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

agent by setting `doModeTraining` to `false`. To train the agent yourself, set `doModeTraining` to `true`.

```
doModeTraining = false;
if doModeTraining
    modeTrainResult = train(modeAgent,modeEnv,modeTrainOpts);
else
    load("rlQubeServoAgents.mat","modeAgent");
end
```




### Simulation

Reset the random seed.

```
rng(0)
```

Configure the trained agents to use greedy policy during simulation.

```
modeAgent.UseExplorationPolicy = false;
swingAgent.UseExplorationPolicy = false;
```

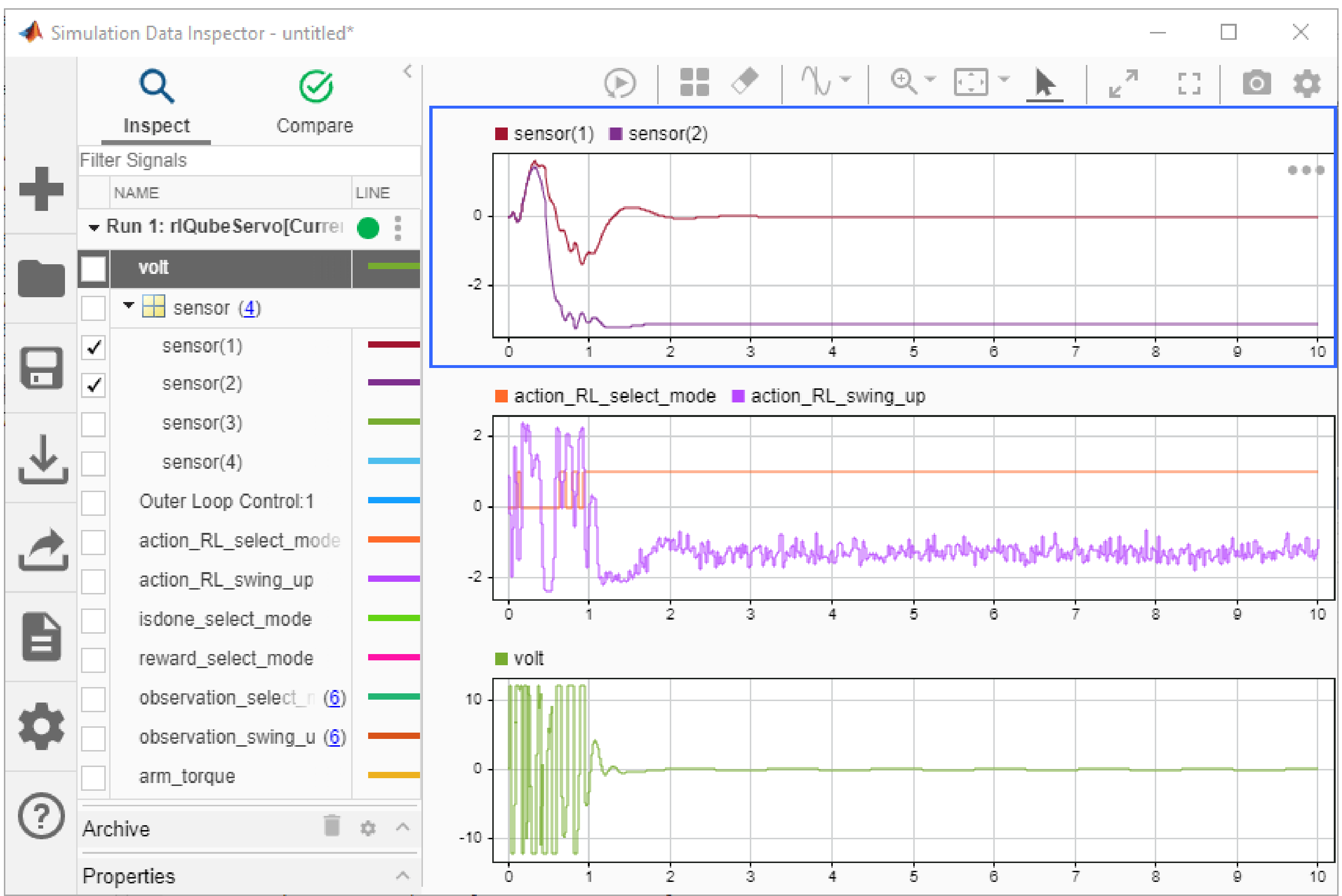
Ensure that the Outer Loop Control block is configured for mode-selection. Then, simulate the model.

```
set_param(mdl + "/Outer Loop Control","DesignModeChoice","Mode select");
sim(mdl);
```

View the performance of the agents in the Simulation Data Inspector. To open the Simulation Data Inspector, in the Simulink model window, on the **Simulation** tab, in the **Review Results** gallery, click **Data Inspector**.

In the plots:

- The measured values for $\theta$ (`sensor(1)`) and $\varphi$ (`sensor(2)`) are stabilized at `0` and $\pi$ radians respectively. The pendulum is stabilized at the upright equilibrium position.
- The `action_RL_select_mode` signal shows the mode switching operation and the `action_RL_swing_up` signal shows the swing up reference angles.
- The low-level control input is shown by the `volt` signal.




## References

[1] Cazzolato, Benjamin Seth, and Zebb Prime. 'On the Dynamics of the Furuta Pendulum'. *Journal of Control Science and Engineering* 2011 (2011): 1–8.

# See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlSACAgent` | `rlSACAgentOptions` | `rlPPOAgent` | `rlPPOAgentOptions` |
`rlAgentInitializationOptions`

**Blocks**
RL Agent

## Related Examples


- “Train DQN Agent to Swing Up and Balance Pendulum” on page 5-89
- “Train DDPG Agent to Swing Up and Balance Pendulum” on page 5-97
- “Run SIL and PIL Verification for Reinforcement Learning” on page 5-134
- “Train Multiple Agents to Perform Collaborative Task” on page 5-190


## More About


- “Create Simulink Reinforcement Learning Environments” on page 2-8
- “Proximal Policy Optimization (PPO) Agents” on page 3-49
- “Soft Actor-Critic (SAC) Agents” on page 3-35
- “Train Reinforcement Learning Agents” on page 5-3

# Run SIL and PIL Verification for Reinforcement Learning

This example shows how to perform software-in-the-loop (SIL) and processor-in-the-loop (PIL) verification workflows for reinforcement learning agents in Simulink®.

This example requires the following hardware.

- Raspberry Pi hardware.
- WIFI dongle or an Ethernet cable.
- Power source connected to a micro USB cable.

You must also download and install the following support packages using the **Add-Ons Explorer**.

- Simulink Support Package for Raspberry Pi Hardware. Follow the "Getting Started with Simulink Support Package for Raspberry Pi Hardware" (Simulink Support Package for Raspberry Pi Hardware) example to set up Raspberry Pi hardware.
- MATLAB® Coder™ Interface for Deep Learning. This will install the Intel® MKL-DNN and Arm® Compute libraries.

### Simulink Environment

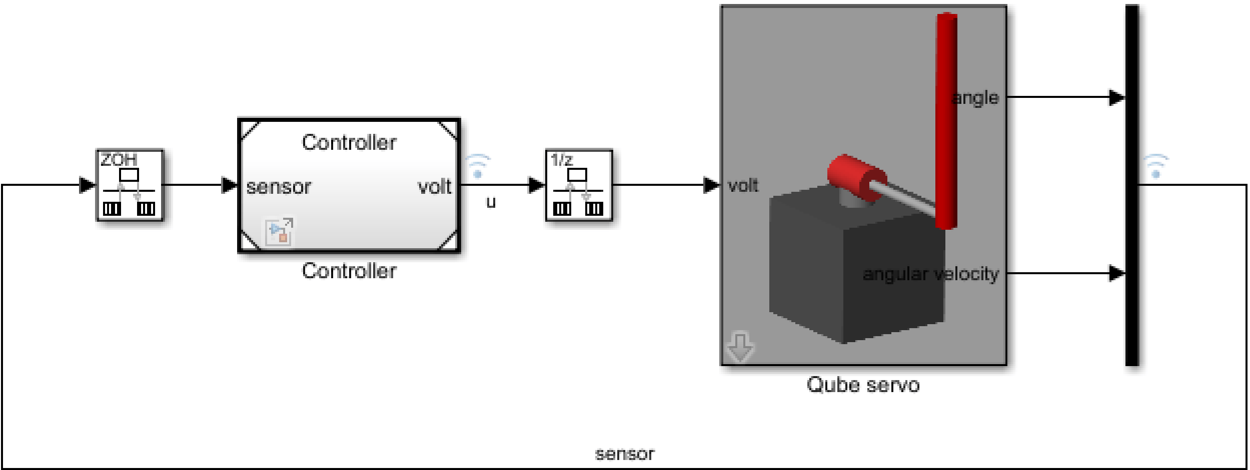
The environment for this example is a Quanser QUBE™-Servo 2 pendulum swing-up model. The swing-up and balancing actions are performed by a combination of proportional-derivative (PD) controllers and reinforcement learning (RL) agents. In this example, you will simulate the controllers in software-in-the-loop (SIL) and processor-in-the-loop (PIL) verification modes and compare the results with normal simulation. For more information, see "SIL and PIL Simulations" (Embedded Coder).

Load the parameters for the environment.

```
loadQubeParameters
```

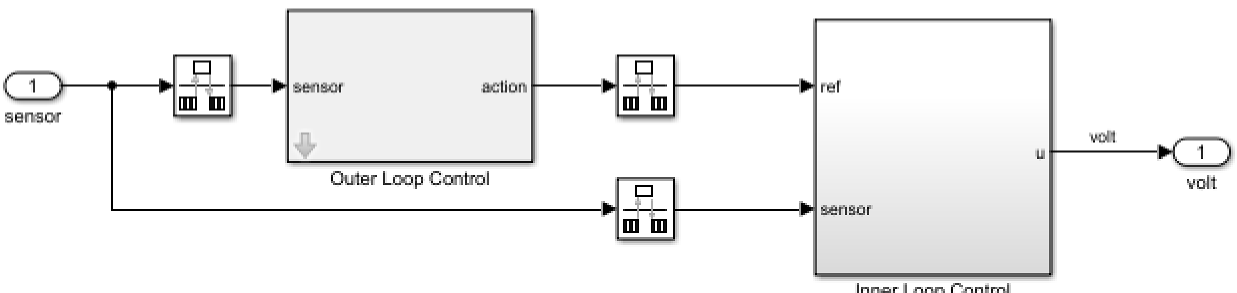
The top level model consists of the controller model reference and the pendulum environment. Open the model.

```
mdl = "rlQubeServo_SIL_PIL";
open_system(mdl)
```




Open the controller model reference.

```
open_system("Controller")
```




The overall control system consists of two RL Agents in the outer loop computing high level reference angles. The reference angles are sent to a low-level controller that stabilizes the pendulum system by computing the motor voltage. For more information on the controller design and training, see "Train Reinforcement Learning Agents to Control Quanser QUBE Pendulum" on page 5-125.

### Policy Evaluation

To generate code and deploy reinforcement learning policies to hardware, you can use one of the following methods.

1. Evaluate the policy using the Deep Learning Toolbox Predict block.
2. Evaluate the policy by generating MATLAB® code using `generatePolicyFunction`. This option is used in this example.

Load the RL agents from the `rlQubeServoSILPILAgents.mat` file.

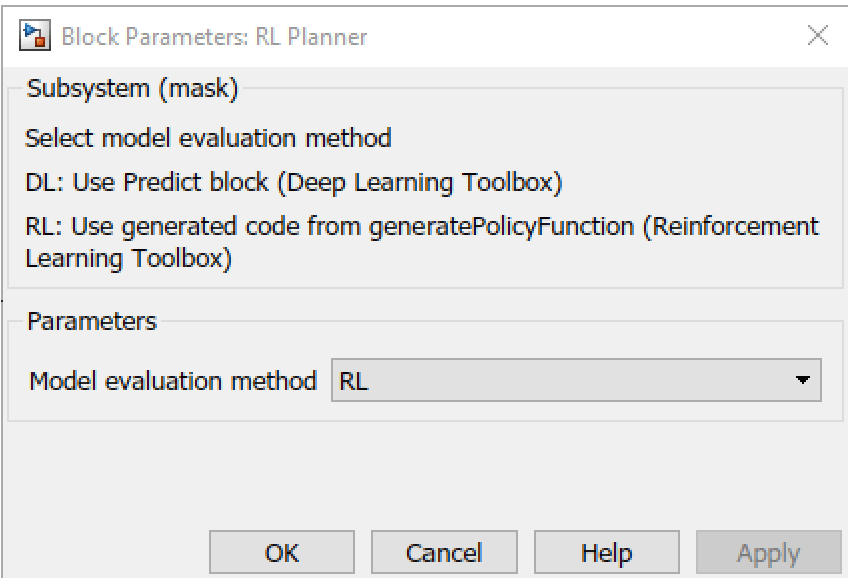
```
load("rlQubeServoSILPILAgents.mat","swingAgent","modeAgent");
```

Generate the following policy evaluation functions.

1. `evaluateAgentSelectPolicy` — For selecting the outer-loop pendulum reference angle
2. `evaluateAgentSwingPolicy` — For oscillating and swinging up the pendulum.

```
generatePolicyFunction(swingAgent,...
    "MATFileName","policy_swing.mat",...
    "FunctionName","evaluateAgentSwingPolicy");
generatePolicyFunction(modeAgent,...
    "MATFileName","policy_select.mat",...
    "FunctionName","evaluateAgentSelectPolicy");
```

Open the Outer Loop Control subsystem to choose the model evaluation method and select RL from the dropdown menu. Doing so activates policy execution using generated MATLAB code.




Alternatively, you can set this parameter using the following command.

```
set_param("Controller/Outer Loop Control","VChoice","RL");
```

The generated functions `evaluateAgentSelectPolicy` and `evaluateAgentSwingPolicy` are executed inside the Outer Loop Control subsystem using MATLAB Function blocks.

### Software-in-the-Loop Simulation

You can analyze code generation performance using software-in-the-loop (SIL) simulation. A SIL simulation generates and builds code on your development computer and then simulates the system using the generated code. You can then compare the results with the ones obtained from a Normal mode simulation.

### Configure Controller for SIL Simulation

The following steps show how to configure code generation settings in Simulink for SIL simulation. You can skip these steps and use preconfigured settings with the following command, which sets the appropriate configuration reference for SIL simulation.

```
setActiveConfigSet("Controller","configSILReferenceRL");
```

Otherwise,

- Open the Controller model.
- In the Simulink model window, on the **Modeling** tab, click **Model Settings**.
- In the Configuration Parameters window, in the **Solver** section, set the **Type** parameter to `Fixed-step` and **Solver** parameter to `discrete`. Set the **Fixed-step size** parameter to `ts_PID`, which is 0.005 s.
- In the **Simulation Target** section, set the **Language** parameter to `C++` and the **Target Library** parameter to `MKL-DNN`. If using the Deep Learning Toolbox Predict block for evaluation, set **Language** to `C`.
- In the **Code Generation** section, set the **System target file** parameter to `ert.tlc` and the **Language** parameter to `C`. Also, select an appropriate **Toolchain** parameter.
- Optionally, you can set the **Language** to `C++` and set the **Target library** in the **Code Generation > Interface** section to **MKL-DNN**. Doing so generates C++ code for the controller.
- Save the model.

#### View Generated Code

Optionally, you can view the generated code for the controller from the C-code perspective.

- In the Simulink model window, on the **Apps** tab, in the gallery, click **Embedded Coder**.
- To generate code and display it in the Code panel, on the **C Code** tab, click **Build**,.
- Ensure that there are no errors in this process. You can reconfigure the code generation settings to optimize the generated code.

#### Configure Top-Level Model for SIL Simulation

Open the top-level model `rlQubeServo_SIL_PIL.slx` and specify the simulation mode for the Controller model reference.

To configure the simulation mode of the Controller model reference, right-click the Controller subsystem and select **Block Parameters**. Then, in the Block Parameters dialog box, set the **Simulation mode** parameter to `Software-in-the-loop (SIL)`.

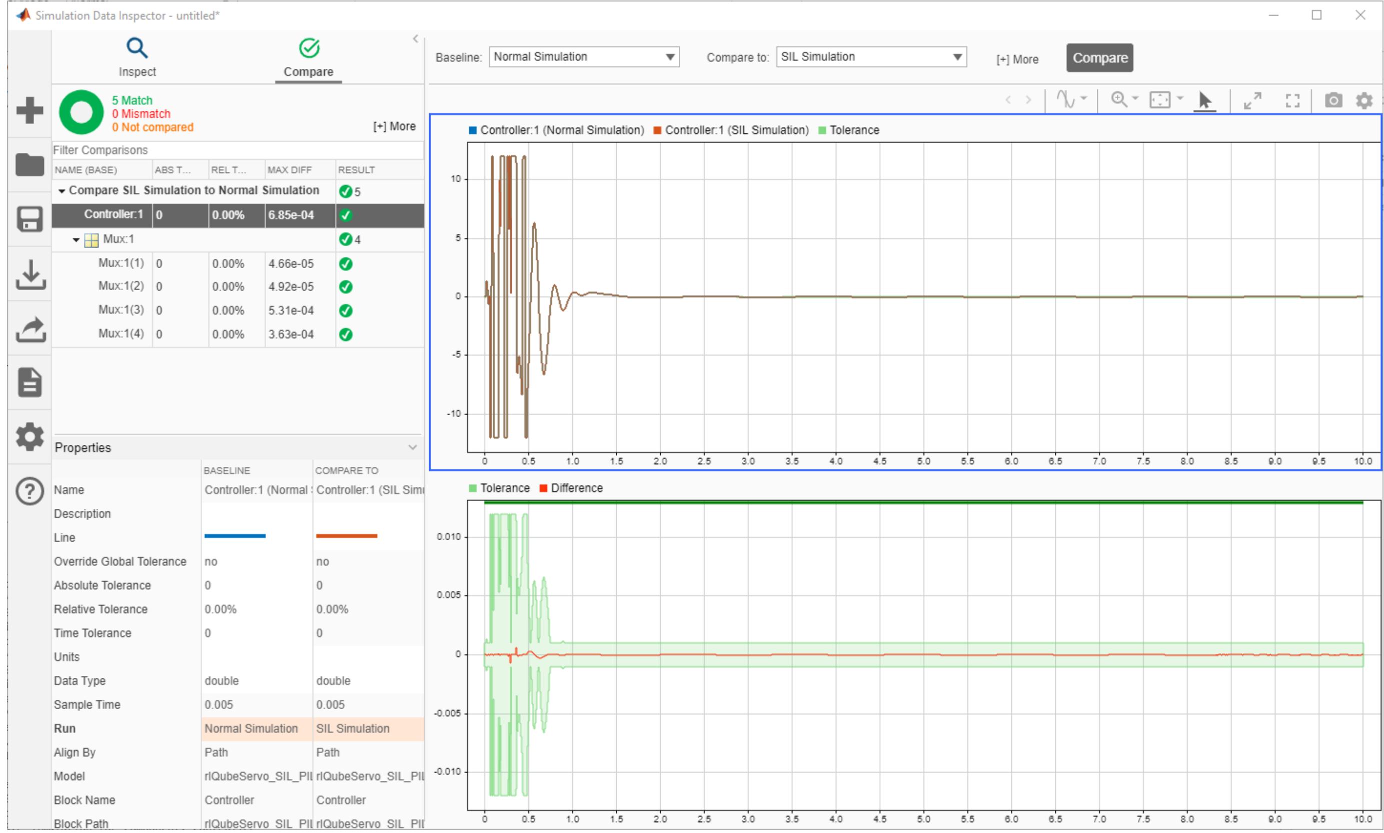
#### Run SIL Simulation

To run the SIL simulation, on the **Apps** tab, click **SIL/PIL Manager**.

On the **SIL/PIL** tab, in the **System Under Test** drop-down menu, select `Model blocks in SIL/PIL mode`. Then, in the **Top Model Mode**, select `Normal`.

To simulate the model, generate and run the code in a SIL simulation, and compare the results, click **Run Verification**. The results are shown in the Simulink Data Inspector.

A comparison of controller output values is shown between Normal and SIL simulations. The error tolerances are acceptable for this example.




## Processor-in-the-Loop Simulation

In a processor-in-the-loop (PIL) simulation, you can generate code for the target hardware (in this case the Raspberry Pi), and deploy and run the code from the hardware. The results of the PIL simulation are transferred to Simulink to verify the numerical equivalence of the simulation and the code generation results. The PIL verification process is an important part of the design cycle to ensure that the behavior of the deployment code matches the design.

## Configure Controller for PIL Simulation

To set up PIL simulation:

- Connect the Raspberry Pi to the host computer and power it on.
- Execute the `raspi` (MATLAB Support Package for Raspberry Pi Hardware) command in MATLAB to ensure that the hardware is connected. This command also displays the device address.

Follow the configuration steps from the SIL simulation workflow. You can alternatively use preconfigured settings with the following command.

```
setActiveConfigSet("Controller","configPILReference");
```

In addition to these settings, in the Configuration Parameters dialog box, in the **Hardware Implementation** section, set the **Hardware board** parameter to `Raspberry Pi` and enter the board parameters. Specify the **Device Address**, **Username**, and **Password** as parameters to appropriate values.

To configure the top-level model for PIL simulation, first open the top-level model `rlQubeServo_SIL_PIL.slx`.

Then, right-click the Controller subsystem and select **Block Parameters**. In the Block Parameters dialog box, set the **Simulation mode** parameter to `Processor-in-the-loop (PIL)`.

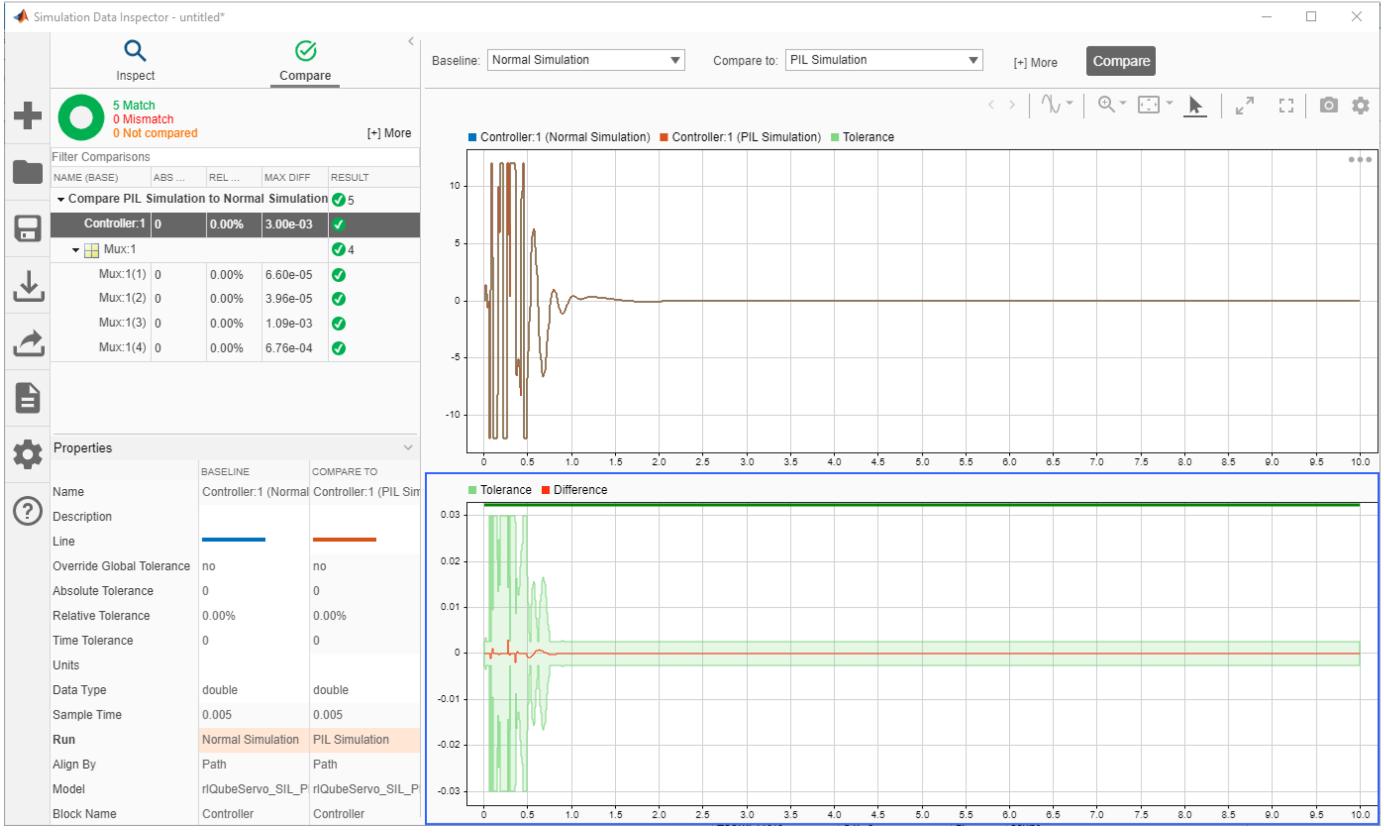
### Run PIL Simulation

To run the PIL simulation, on the **Apps** tab, click **SIL/PIL Manager**.

On the **SIL/PIL** tab, in the **System Under Test** drop-down menu, select `Model blocks in SIL/PIL mode`. Then, in the **Top Model Mode**, select `Normal`.

To simulate the model, generate and run the code on the hardware, and compare the results, click **Run Verification**. The results are shown in the Simulink Data Inspector.

A comparison of controller output values is shown between Normal and PIL simulations. The error tolerances are acceptable for this example.




## See Also

**Functions**
`generatePolicyFunction` | `raspi` | `generatePolicyBlock`

**Objects**
`rlSACAgent` | `rlPPOAgent` | `rlTrainingOptions`

**Blocks**
Predict | RL Agent

## Related Examples

- "Train DDPG Agent to Swing Up and Balance Pendulum" on page 5-97
- "Train Reinforcement Learning Agents to Control Quanser QUBE Pendulum" on page 5-125

## More About

- "Deploy Trained Reinforcement Learning Policies" on page 6-2
- "SIL and PIL Simulations" (Embedded Coder)
- "Getting Started with Simulink Support Package for Raspberry Pi Hardware" (Simulink Support Package for Raspberry Pi Hardware)

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

This example shows how to train a deep deterministic policy gradient (DDPG) agent to swing up and balance a pendulum with an image observation modeled in MATLAB®.

For more information on DDPG agents, see "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40.

### Simple Pendulum with Image MATLAB Environment

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

For this environment:

- The upward balanced pendulum position is 0 radians, and the downward hanging position is `pi` radians.
- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are an image indicating the location of the pendulum mass and the pendulum angular velocity.
- The reward $r_t$, provided at every time step, is

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

Here:

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

For more information on this model, see "Load Predefined Control System Environments" on page 2-23.

### Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv("SimplePendulumWithImage-Continuous")
```

```
env = 
  SimplePendlumWithImageContinuousAction with properties:

             Mass: 1
        RodLength: 1
       RodInertia: 0
          Gravity: 9.8100
     DampingRatio: 0
    MaximumTorque: 2
               Ts: 0.0500
```

```
           State: [2x1 double]
               Q: [2x2 double]
               R: 1.0000e-03
```

The interface has a continuous action space where the agent can apply a torque between –2 to 2 N·m.

Obtain the observation and action specification from the environment interface.

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

#### Create DDPG Agent

DDPG agents use a parametrized Q-value function approximator to estimate the value of the policy. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for which receives the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a convolutional neural network (CNN) with three input layers (one for each observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value).

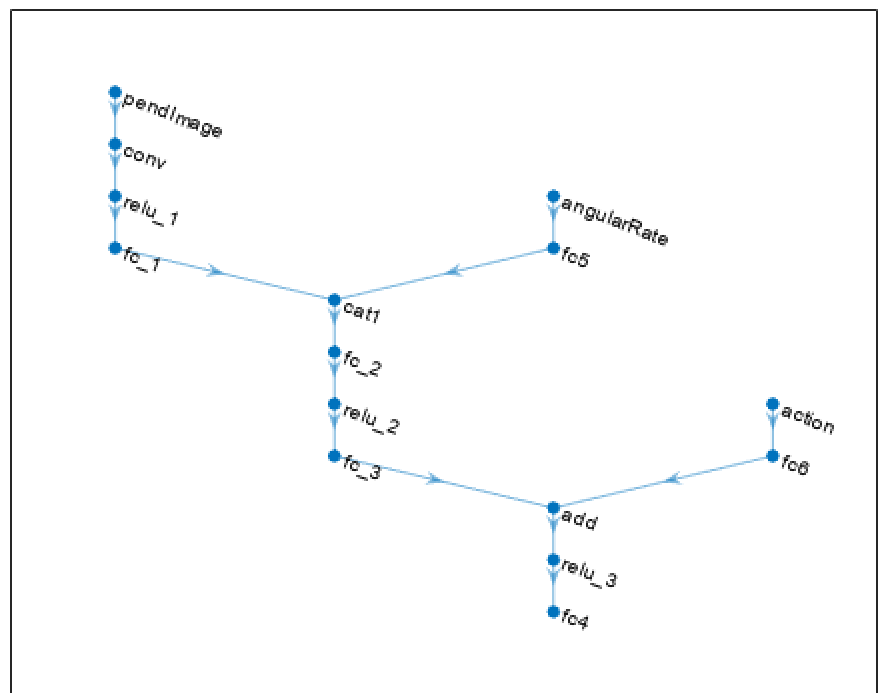
Define each network path as an array of layer objects, and assign names to the input and output layers of each path, as well as to the addition and concatenation layers. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel. For more information on creating representations, see “Create Policies and Value Functions” on page 4-2.

```
hiddenLayerSize1 = 400;
hiddenLayerSize2 = 300;

% Image input path
imgPath = [
    imageInputLayer(obsInfo(1).Dimension, ...
        Normalization="none", ...
        Name=obsInfo(1).Name)
    convolution2dLayer(10,2,Stride=5,Padding=0)
    reluLayer
    fullyConnectedLayer(2)
    concatenationLayer(3,2,Name="cat1")
    fullyConnectedLayer(hiddenLayerSize1)
    reluLayer
    fullyConnectedLayer(hiddenLayerSize2)
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(1,Name="fc4")
    ];

% d(theta)/dt input path
dthPath = [
    imageInputLayer(obsInfo(2).Dimension, ...
```

```
        Normalization="none", ...
        Name=obsInfo(2).Name)
    fullyConnectedLayer(1,Name="fc5", ...
        BiasLearnRateFactor=0, ...
        Bias=0)
    ];

% Action path
actPath =[
    imageInputLayer(actInfo(1).Dimension, ...
        Normalization="none", ...
        Name="action")
    fullyConnectedLayer(hiddenLayerSize2, ...
        Name="fc6", ...
        BiasLearnRateFactor=0, ...
        Bias=zeros(hiddenLayerSize2,1))
    ];

% Assemble paths
criticNetwork = layerGraph(imgPath);
criticNetwork = addLayers(criticNetwork,dthPath);
criticNetwork = addLayers(criticNetwork,actPath);
criticNetwork = connectLayers(criticNetwork,"fc5","cat1/in2");
criticNetwork = connectLayers(criticNetwork,"fc6","add/in2");
```

View the critic network configuration and display the number of parameters.

```
figure
plot(criticNetwork)
```




```
%summary(criticNetwork)
```

Create the critic representation using the specified neural network and the environment action and observation specifications. Pass as additional arguments also the names of the network layers to be connected with the observation and action channels. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(criticNetwork, ...
    obsInfo,actInfo,...
    ObservationInputNames={"pendImage","angularRate"}, ...
    ActionInputNames={"action"});
```

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is implemented by a continuous deterministic actor. A continuous deterministic actor implements a parametrized deterministic policy for a continuous action space. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with two input layers (receiving the content of the two environment observation channels, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects.

```
% Image input path
imgPath = [
    imageInputLayer(obsInfo(1).Dimension, ...
```

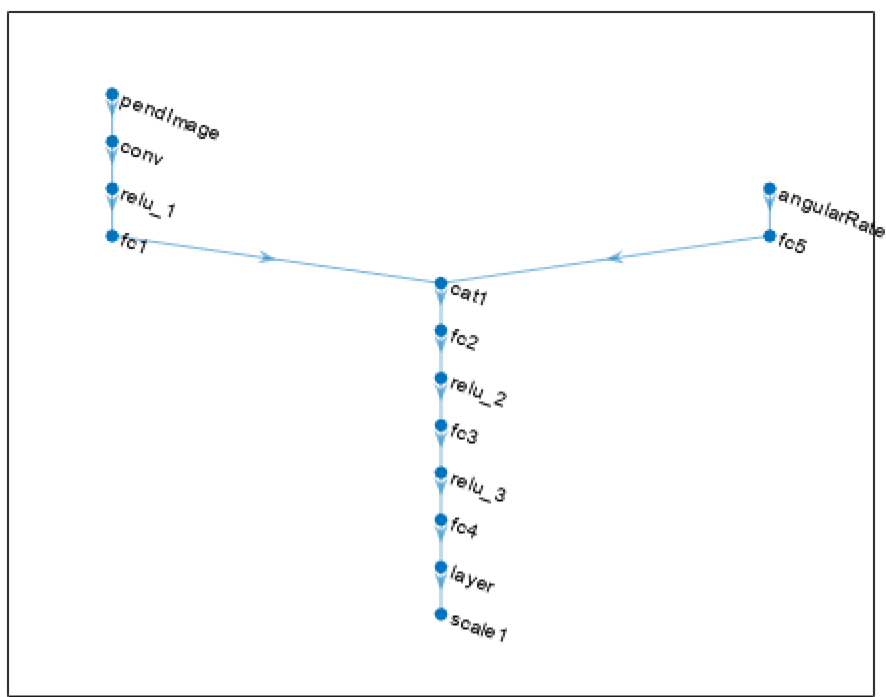
```
        Normalization="none", ...
        Name=obsInfo(1).Name)
    convolution2dLayer(10,2,Stride=5,Padding=0)
    reluLayer
    fullyConnectedLayer(2,Name="fc1")
    concatenationLayer(3,2,Name="cat1")
    fullyConnectedLayer(hiddenLayerSize1,Name="fc2")
    reluLayer
    fullyConnectedLayer(hiddenLayerSize2,Name="fc3")
    reluLayer
    fullyConnectedLayer(1,Name="fc4")
    tanhLayer
    scalingLayer(Name="scale1", ...
        Scale=max(actInfo.UpperLimit))
    ];

% d(theta)/dt input layer
dthPath = [
    imageInputLayer(obsInfo(2).Dimension, ...
        Normalization="none", ...
        Name=obsInfo(2).Name)
    fullyConnectedLayer(1, ...
        Name="fc5", ...
        BiasLearnRateFactor=0, ...
        Bias=0)
    ];

% Assemble actor network
actorNetwork = layerGraph(imgPath);
actorNetwork = addLayers(actorNetwork,dthPath);
actorNetwork = connectLayers(actorNetwork,"fc5","cat1/in2");
```

Create the actor using the specified neural network. For more information, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNetwork, ...
    obsInfo,actInfo, ...
    ObservationInputNames={"pendImage","angularRate"});
```

View the actor network configuration and display the number of weights.

```
figure
plot(actorNetwork)
```




```
%summary(actorNetwork)
```

Specify options for the actor and critic using `rlOptimizerOptions`.

```
criticOptions = rlOptimizerOptions(LearnRate=1e-03,GradientThreshold=1);
actorOptions = rlOptimizerOptions(LearnRate=1e-04,GradientThreshold=1);
```

Training performance using the GPU is impacted by the batch size, network structure, and the hardware itself. Therefore, using a GPU does not always guarantee a better training performance. For more information on supported GPUs, see “GPU Computing Requirements” (Parallel Computing Toolbox).

Uncomment the following line to train the critic using a GPU.

```
% criticOptions.UseDevice = "gpu";
```

Uncomment the following line to train the actor using a GPU.

```
% actorOptions.UseDevice = "gpu";
```

Specify the DDPG agent options using `rlDDPGAgentOptions`.

```
agentOptions = rlDDPGAgentOptions(...
    SampleTime=env.Ts,...
    TargetSmoothFactor=1e-3,...
    ExperienceBufferLength=1e6,...
    DiscountFactor=0.99,...
    MiniBatchSize=128);
```

You can also specify options using dot notation.

```
agentOptions.NoiseOptions.Variance = 0.6;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;
```

Specify the training options for the function approximator objects.

```
agentOptions.CriticOptimizerOptions = criticOptions;
agentOptions.ActorOptimizerOptions = actorOptions;
```

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

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

### Train Agent

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

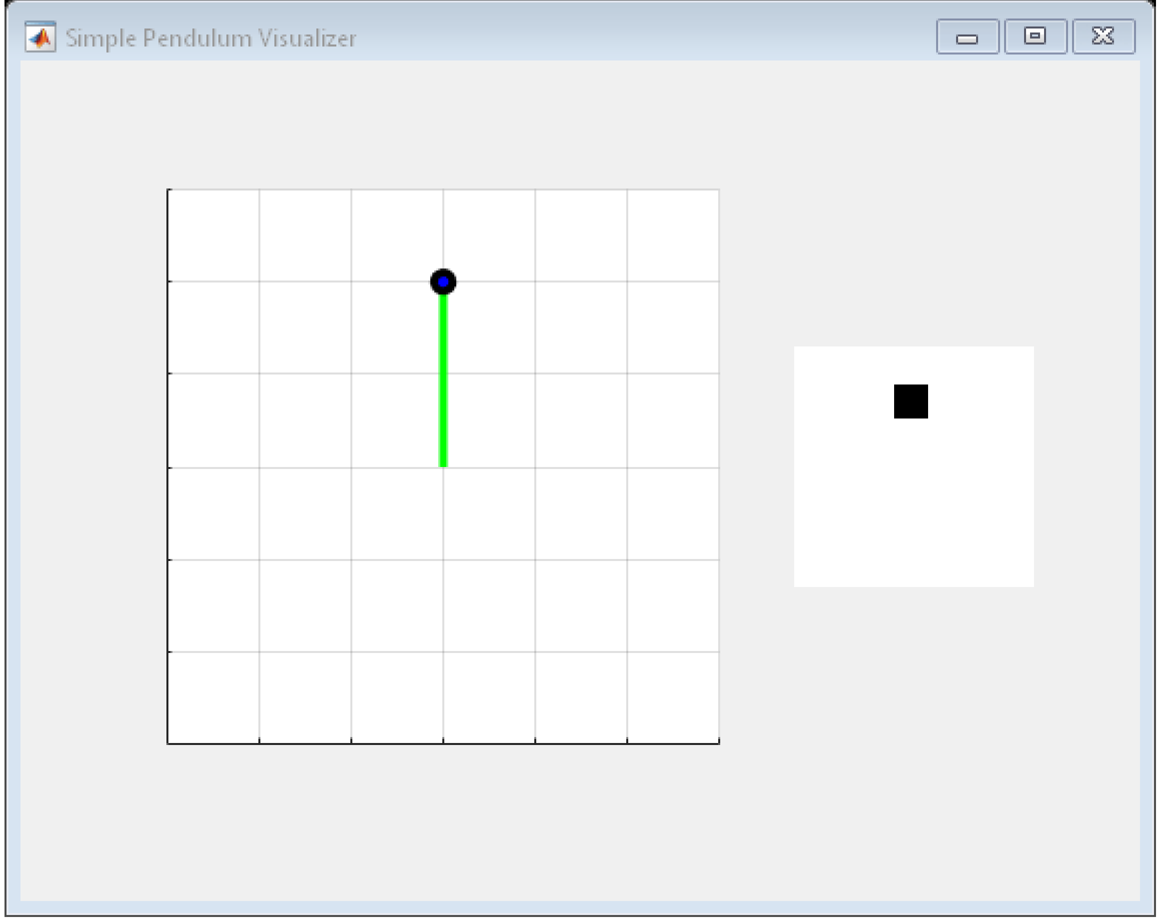
- Run each training for at most 5000 episodes, with each episode lasting at most 400 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option).
- Stop training when the agent receives a moving average cumulative reward greater than -740 over ten consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = 400;
trainingOptions = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-740);
```

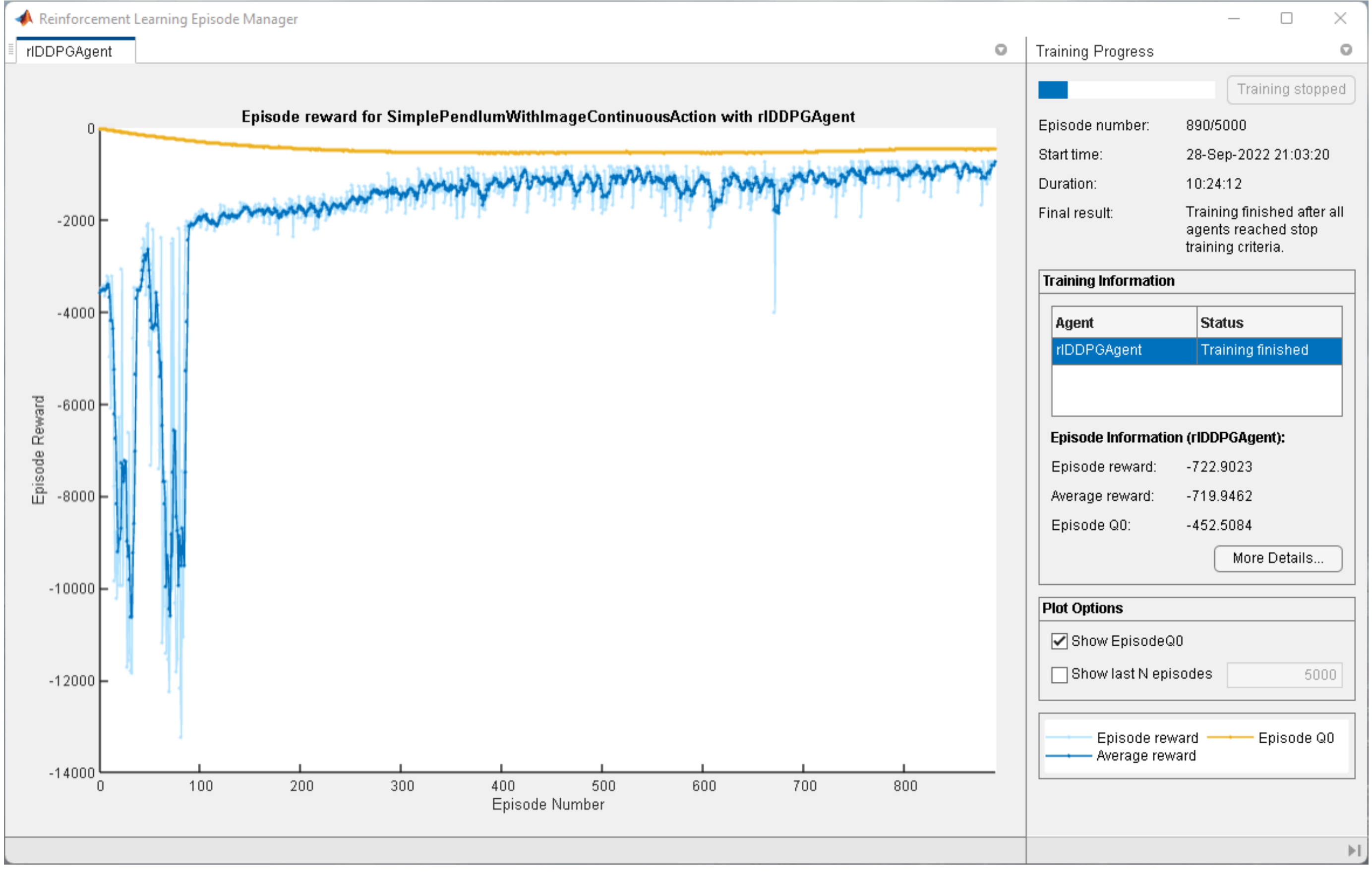
You can visualize the pendulum by using the `plot` function during training or simulation.

```
plot(env)
```




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

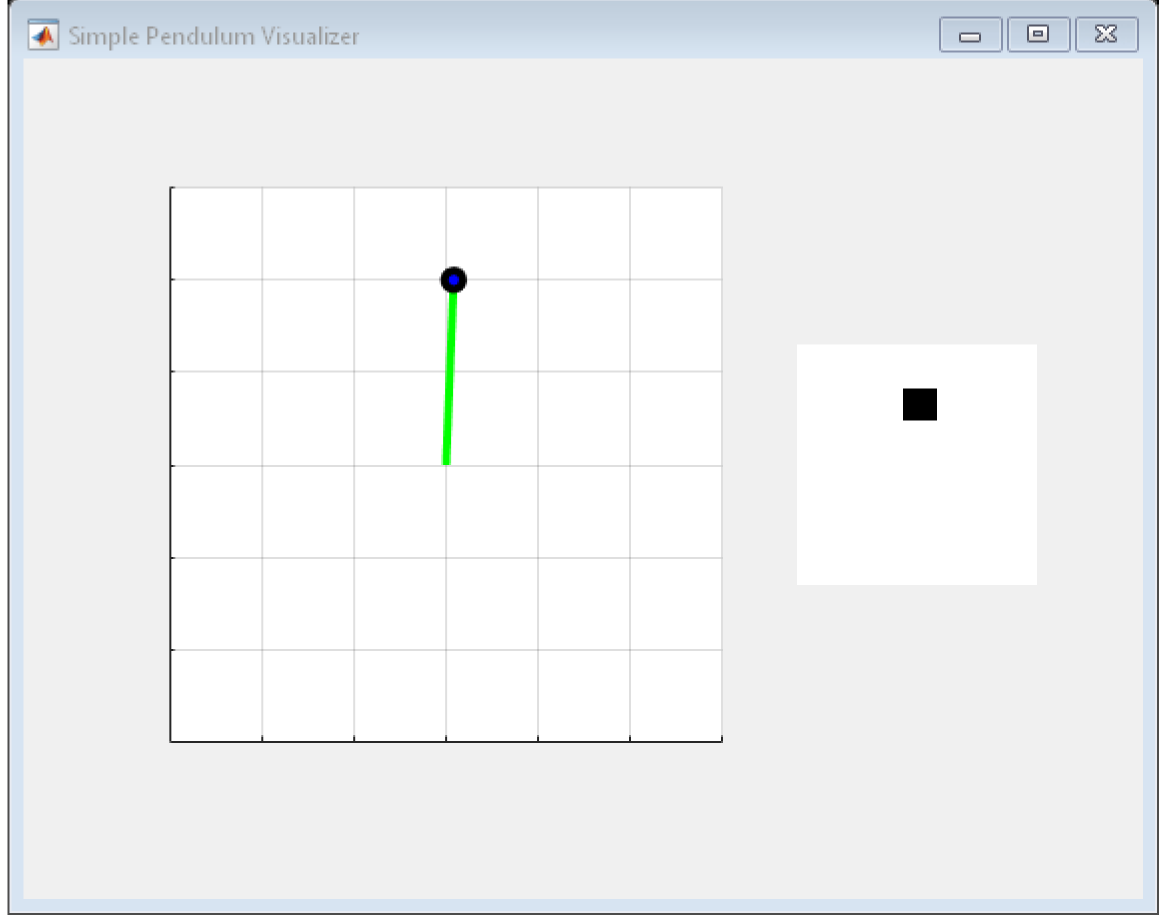
```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
    % Load pretrained agent for the example.
    load("SimplePendulumWithImageDDPG.mat","agent")
end
```




## Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




## See Also

**Functions**
train | sim

**Objects**
rlDDPGAgent | rlDDPGAgentOptions | rlQValueFunction | rlContinuousDeterministicActor | rlTrainingOptions | rlSimulationOptions | rlOptimizerOptions

**Blocks**
RL Agent

## Related Examples

- "Train DQN Agent to Swing Up and Balance Pendulum" on page 5-89
- "Train DDPG Agent to Swing Up and Balance Pendulum" on page 5-97
- "Train DDPG Agent to Swing Up and Balance Pendulum with Bus Signal" on page 5-115

## More About

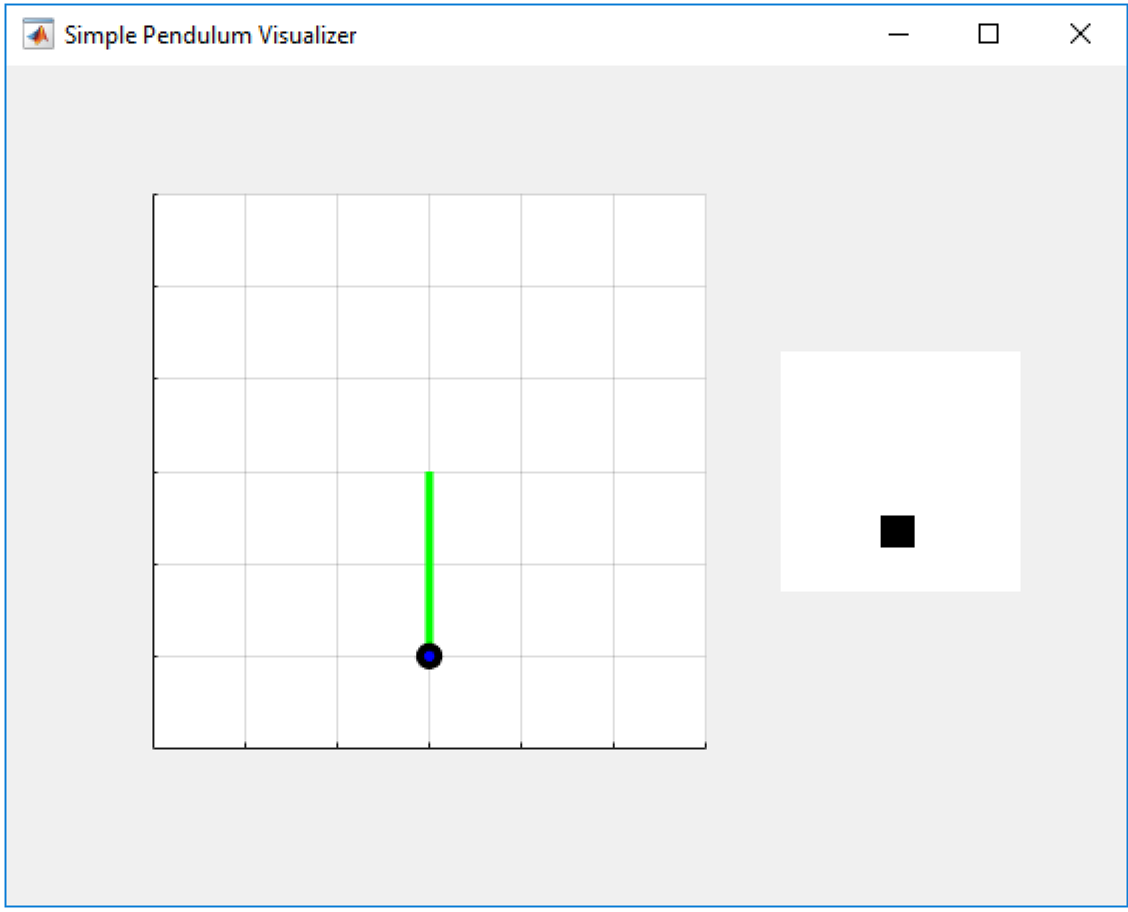
- "Load Predefined Control System Environments" on page 2-23
- "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40
- "Create Policies and Value Functions" on page 4-2
- "Train Reinforcement Learning Agents" on page 5-3

# Create DQN Agent Using Deep Network Designer and Train Using Image Observations

This example shows how to create a deep Q-learning network (DQN) agent that can swing up and balance a pendulum modeled in MATLAB®. In this example, you create the DQN agent using Deep Network Designer. For more information on DQN agents, see “Deep Q-Network (DQN) Agents” on page 3-23.

#### Pendulum Swing-Up with Image MATLAB Environment

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




For this environment:

- The upward balanced pendulum position is 0 radians, and the downward hanging position is `pi` radians.
- The torque action signal from the agent to the environment can take any of the five possible integer values from –2 to 2 N·m.

- The observations from the environment are the simplified grayscale image of the pendulum and the pendulum angle derivative.
- The reward $r_t$, provided at every time step, is

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

Here:

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

For more information on the continuous action space version of this model, see “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation” on page 5-141.

#### Create Environment Interface

Create a predefined environment interface for the pendulum.

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

The interface has two observations. The first observation, named `"pendImage"`, is a 50-by-50 grayscale image.

```
obsInfo = getObservationInfo(env);
obsInfo(1)
```

```
ans = 
  rlNumericSpec with properties:

     LowerLimit: 0
     UpperLimit: 1
           Name: "pendImage"
    Description: [0x0 string]
      Dimension: [50 50]
       DataType: "double"
```

The second observation, named `"angularRate"`, is the angular velocity of the pendulum.

```
obsInfo(2)
```

```
ans = 
  rlNumericSpec with properties:

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

The interface has a discrete action space where the agent can apply one of five possible torque values to the pendulum: –2, –1, 0, 1, or 2 N·m.

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

```
actInfo = 
  rlFiniteSetSpec with properties:

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Construct Critic Network Using Deep Network Designer

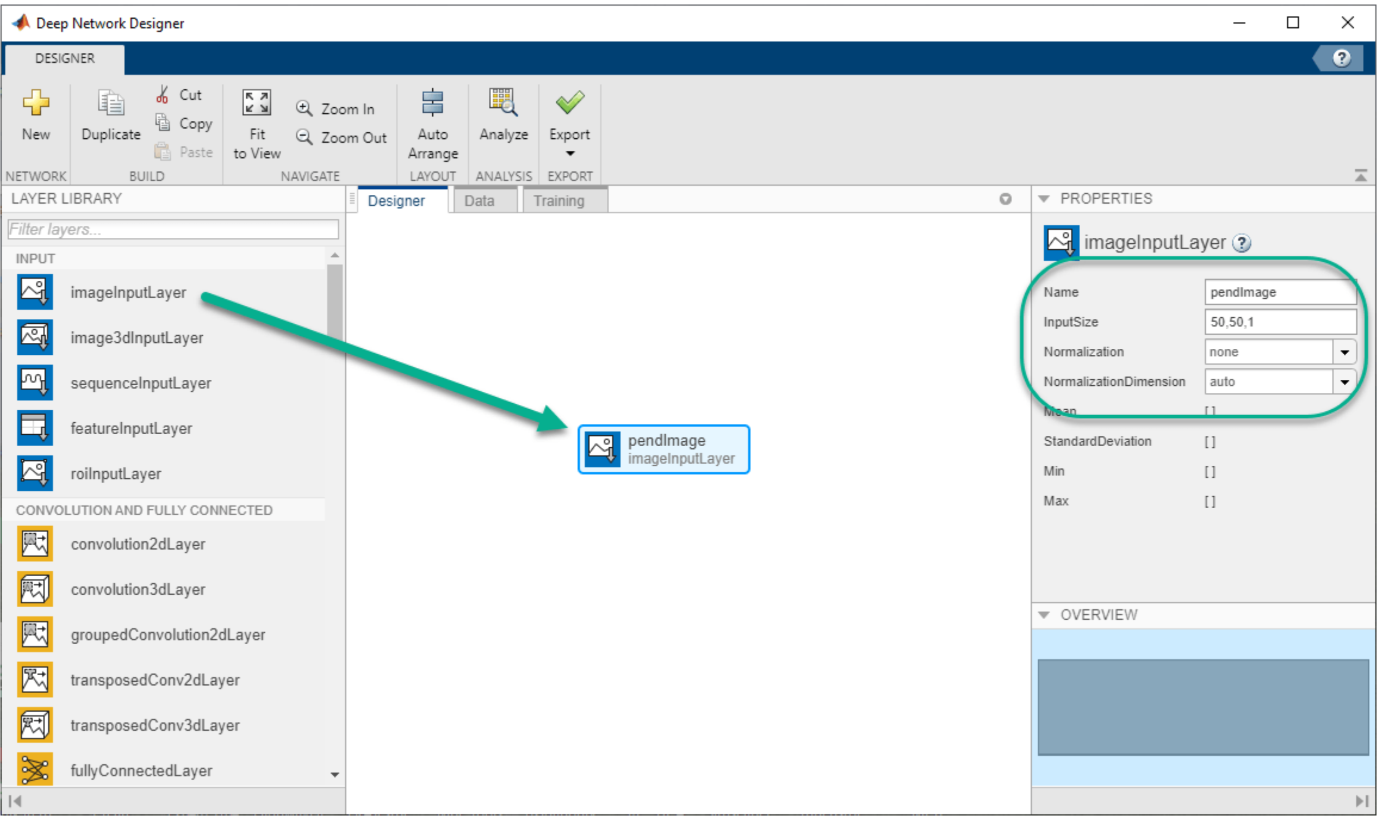
DQN agents use a parametrized Q-value function approximator to estimate the value of the policy. Since a DQN agents has a discrete action space, you can use a vector (that is multi-output) Q-value function critic, which is generally more efficient than a comparable single-output critic. However, for this example, use a standard single-output Q-value function critic.

To model the parametrized Q-value function within the critic, use a neural network with three input layers (two for the observation channels, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value). For more information on creating Q-value function representations based on a deep neural network, see “Create Policies and Value Functions” on page 4-2.

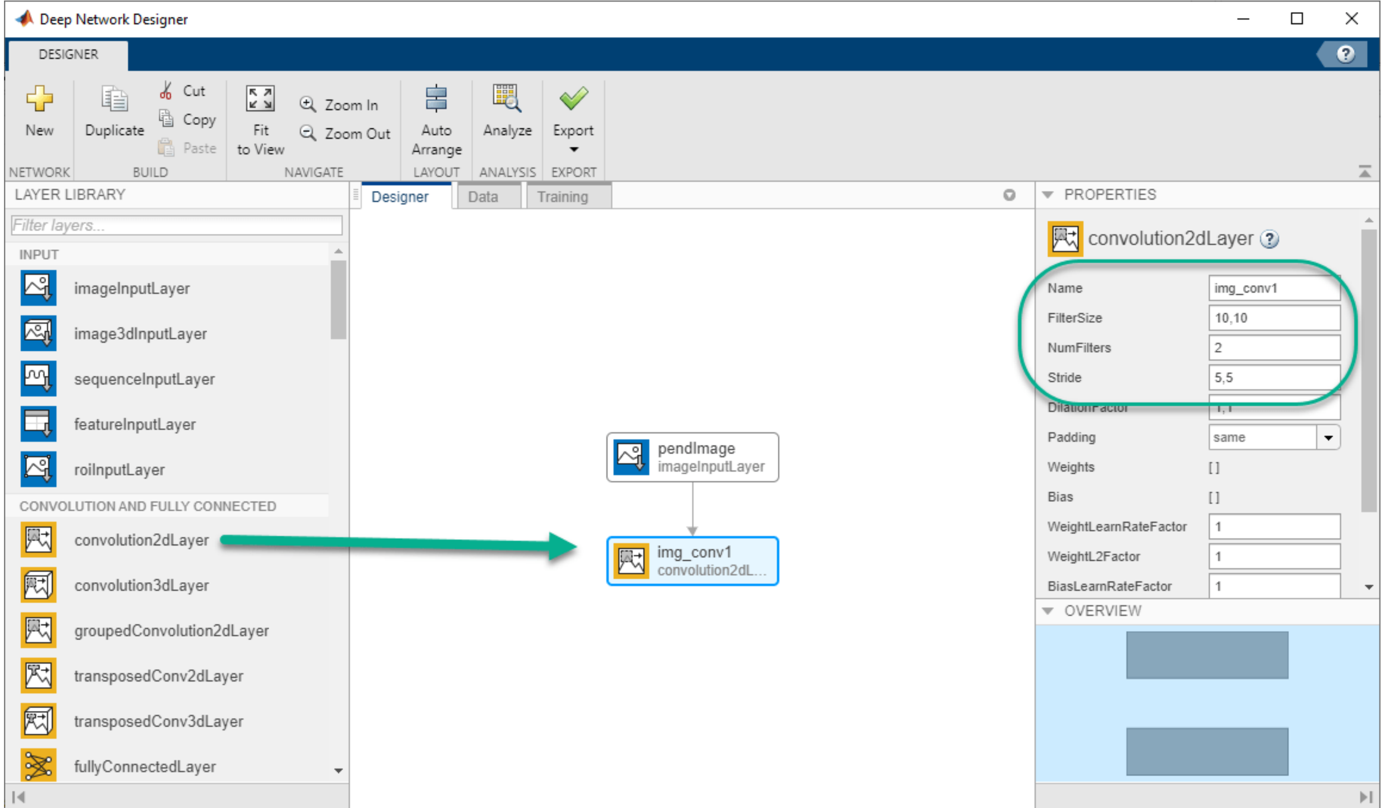
Construct the critic network interactively by using the Deep Network Designer app. To do so, you first create separate input paths for each observation and action. These paths learn lower-level features from their respective inputs. You then create a common output path that combines the outputs from the input paths.

#### Create Image Observation Path

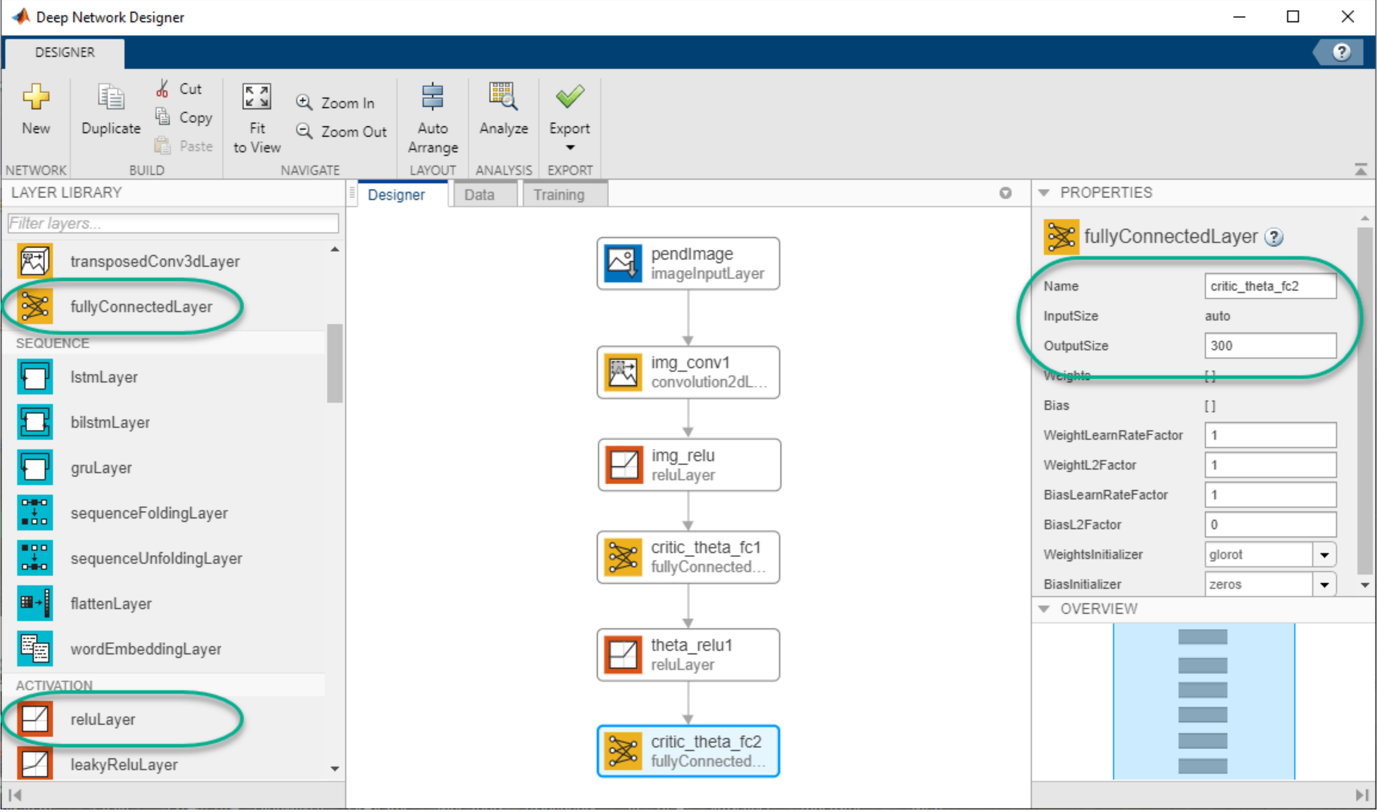
To create the image observation path, first drag an `imageInputLayer` from the **Layer Library** pane to the canvas. Set the layer **InputSize** to `50,50,1` for the image observation, and set **Normalization** to `none`.




Second, drag a `convolution2DLayer` to the canvas and connect the input of this layer to the output of the `imageInputLayer`. Create a convolution layer with 2 filters (**NumFilters** property) that have a height and width of `10` (**FilterSize** property), and use a stride of `5` in the horizontal and vertical directions (**Stride** property).




Finally, complete the image path network with two sets of `reLULayer` and `fullyConnectedLayer` layers. The output sizes of the first and second `fullyConnectedLayer` layers are 400 and 300, respectively.




### Create All Input Paths and Output Path

Construct the other input paths and the output path in a similar manner. For this example, use the following options.

Angular velocity path (scalar input):

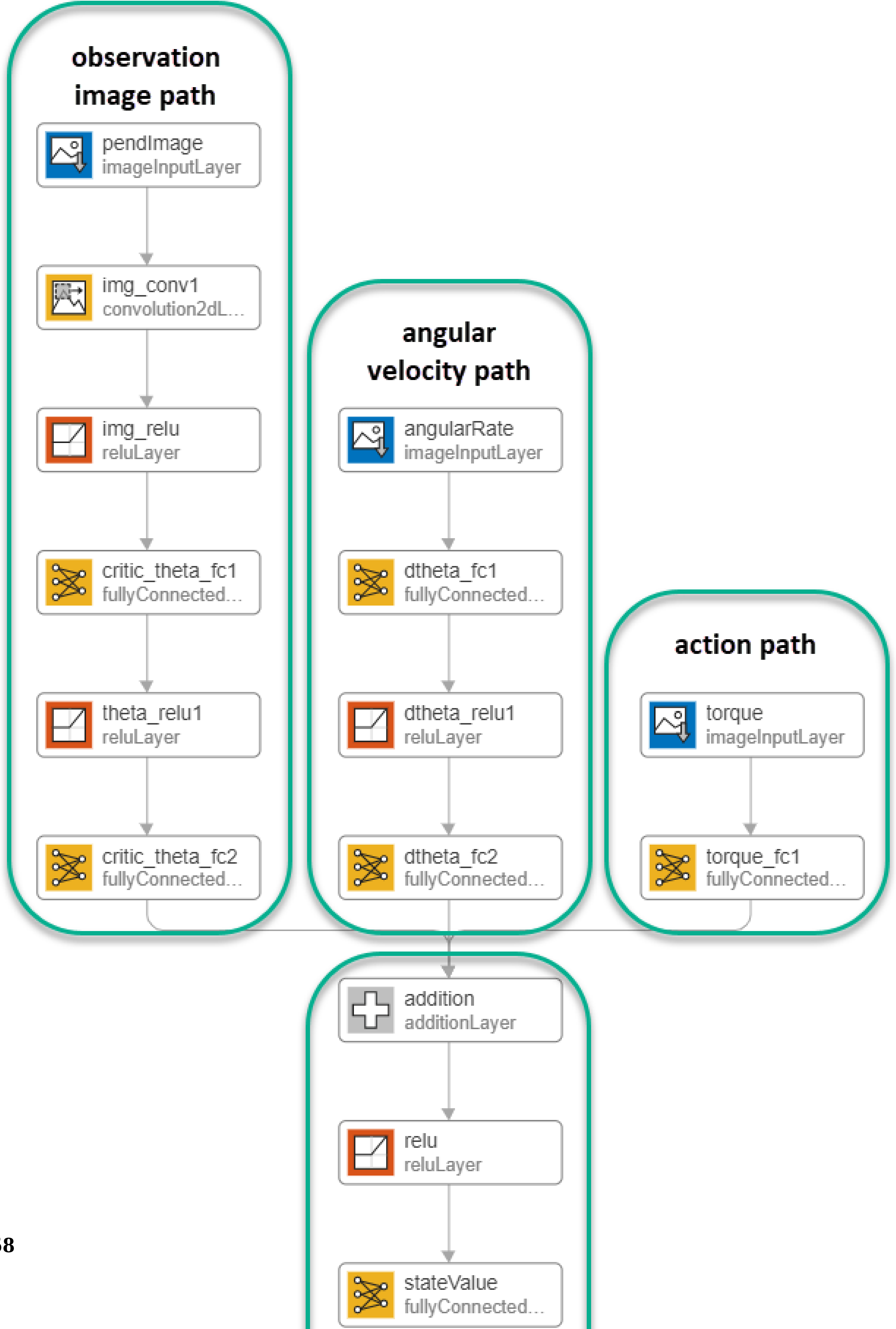
- `imageInputLayer` — Set **InputSize** to `1,1` and **Normalization** to `none`.
- `fullyConnectedLayer` — Set **OutputSize** to `400`.
- `reLULayer`
- `fullyConnectedLayer` — Set **OutputSize** to `300`.

Action path (scalar input):

- `imageInputLayer` — Set **InputSize** to `1,1` and **Normalization** to `none`.
- `fullyConnectedLayer` — Set **OutputSize** to `300`.

Output path:

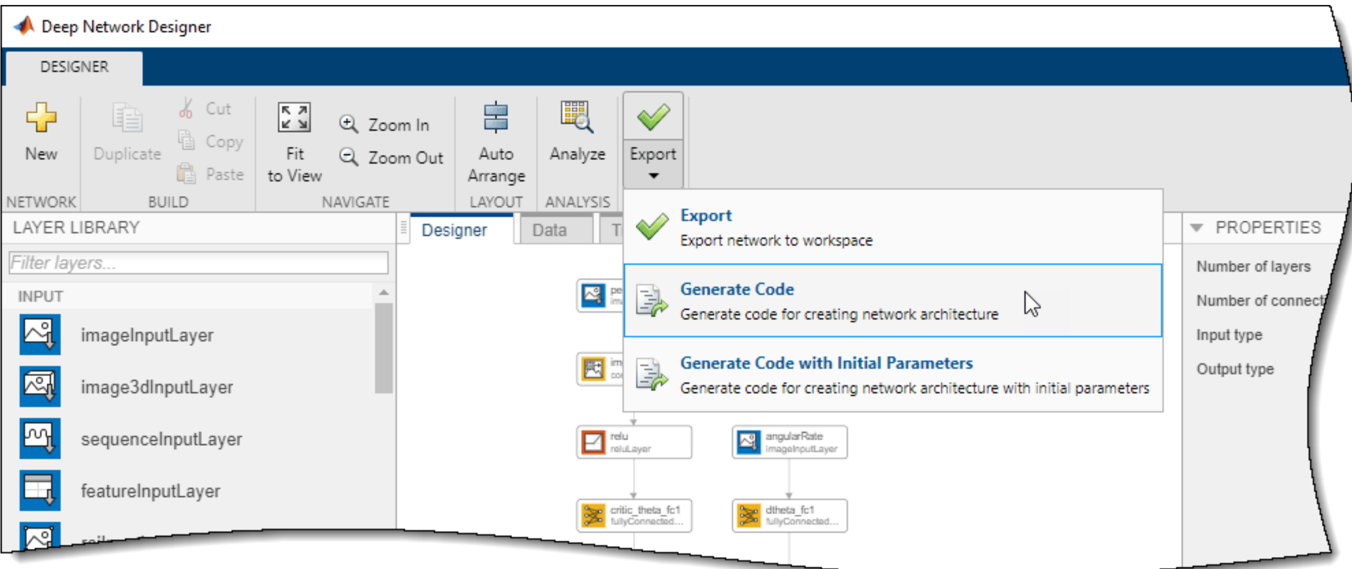
- `additionLayer` — Connect the output of all input paths to the input of this layer.
- `reLULayer`
- `fullyConnectedLayer` — Set **OutputSize** to `1` for the scalar value function.


observation image path
pendImage
imageInputLayer
img_conv1
convolution2dL...
img_relu
reluLayer
critic_theta_fc1
fullyConnected...
theta_relu1
reluLayer
critic_theta_fc2
fullyConnected...
angular velocity path
angularRate
imageInputLayer
dtheta_fc1
fullyConnected...
dtheta_relu1
reluLayer
dtheta_fc2
fullyConnected...
action path
torque
imageInputLayer
torque_fc1
fullyConnected...
addition
additionLayer
relu
reluLayer
stateValue
fullyConnected...

### Export Network from Deep Network Designer

To export the network to the MATLAB workspace, in **Deep Network Designer**, click **Export**. **Deep Network Designer** exports the network as a new variable containing the network layers. You can create the critic representation using this layer network variable.

Alternatively, to generate equivalent MATLAB code for the network, click **Export > Generate Code**.



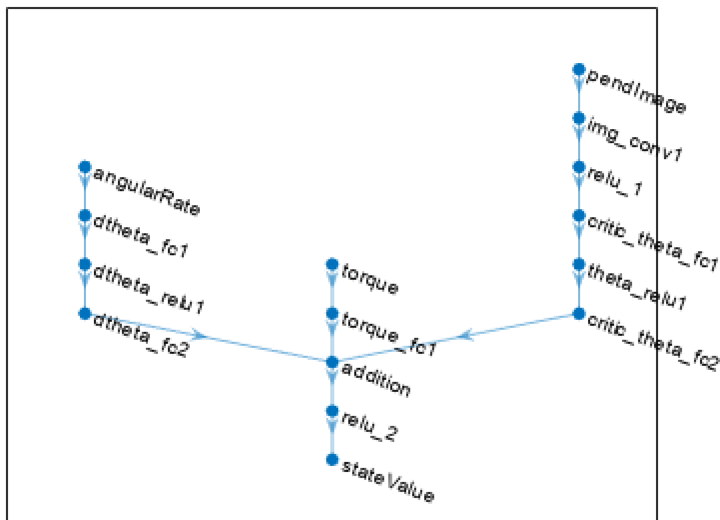

The generated code is as follows.

```
lgraph = layerGraph();

tempLayers = [
    imageInputLayer([1 1 1],"Name","angularRate","Normalization","none")
    fullyConnectedLayer(400,"Name","dtheta_fc1")
    reluLayer("Name","dtheta_relu1")
    fullyConnectedLayer(300,"Name","dtheta_fc2")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    imageInputLayer([1 1 1],"Name","torque","Normalization","none")
    fullyConnectedLayer(300,"Name","torque_fc1")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    imageInputLayer([50 50 1],"Name","pendImage","Normalization","none")
    convolution2dLayer([10 10],2,"Name","img_conv1","Padding","same","Stride",[5 5])
    reluLayer("Name","relu_1")
    fullyConnectedLayer(400,"Name","critic_theta_fc1")
    reluLayer("Name","theta_relu1")
    fullyConnectedLayer(300,"Name","critic_theta_fc2")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(3,"Name","addition")
    reluLayer("Name","relu_2")
```

```
    fullyConnectedLayer(1,"Name","stateValue")];
lgraph = addLayers(lgraph,tempLayers);

lgraph = connectLayers(lgraph,"torque_fc1","addition/in3");
lgraph = connectLayers(lgraph,"critic_theta_fc2","addition/in1");
lgraph = connectLayers(lgraph,"dtheta_fc2","addition/in2");
```

View the critic network configuration.

```
figure
plot(lgraph)
```




Convert to a `dlnetwork` object and display the number of parameters.

```
net = dlnetwork(lgraph);
summary(net)
```

```
   Initialized: true

   Number of learnables: 322.9k

   Inputs:
      1   'angularRate'   1x1x1 images
      2   'torque'        1x1x1 images
      3   'pendImage'     50x50x1 images
```

Create the critic using the neural network, the action and observation specifications, and the names of the input layers to be connected to the observations and action channels. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(net,obsInfo,actInfo,...
    "ObservationInputNames",["pendImage","angularRate"], ...
    "ActionInputNames","torque");
```

Specify options for the critic using `rlOptimizerOptions`.

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

Specify the DQN agent options using `rlDQNAgentOptions`. Include the training options for the actor and critic.

```
agentOpts = rlDQNAgentOptions(...
    UseDoubleDQN=false,...
    CriticOptimizerOptions=criticOpts,...
    ExperienceBufferLength=1e6,...
    SampleTime=env.Ts);
```

You can also set or modify agent options using dot notation.

```
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 1e-5;
```

Alternatively, you can create the agent first, and then modify its options using dot notation.

Create the DQN agent using the critic and the agent options object. For more information, see `rlDQNAgent`.

```
agent = rlDQNAgent(critic,agentOpts);
```

#### Train Agent

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

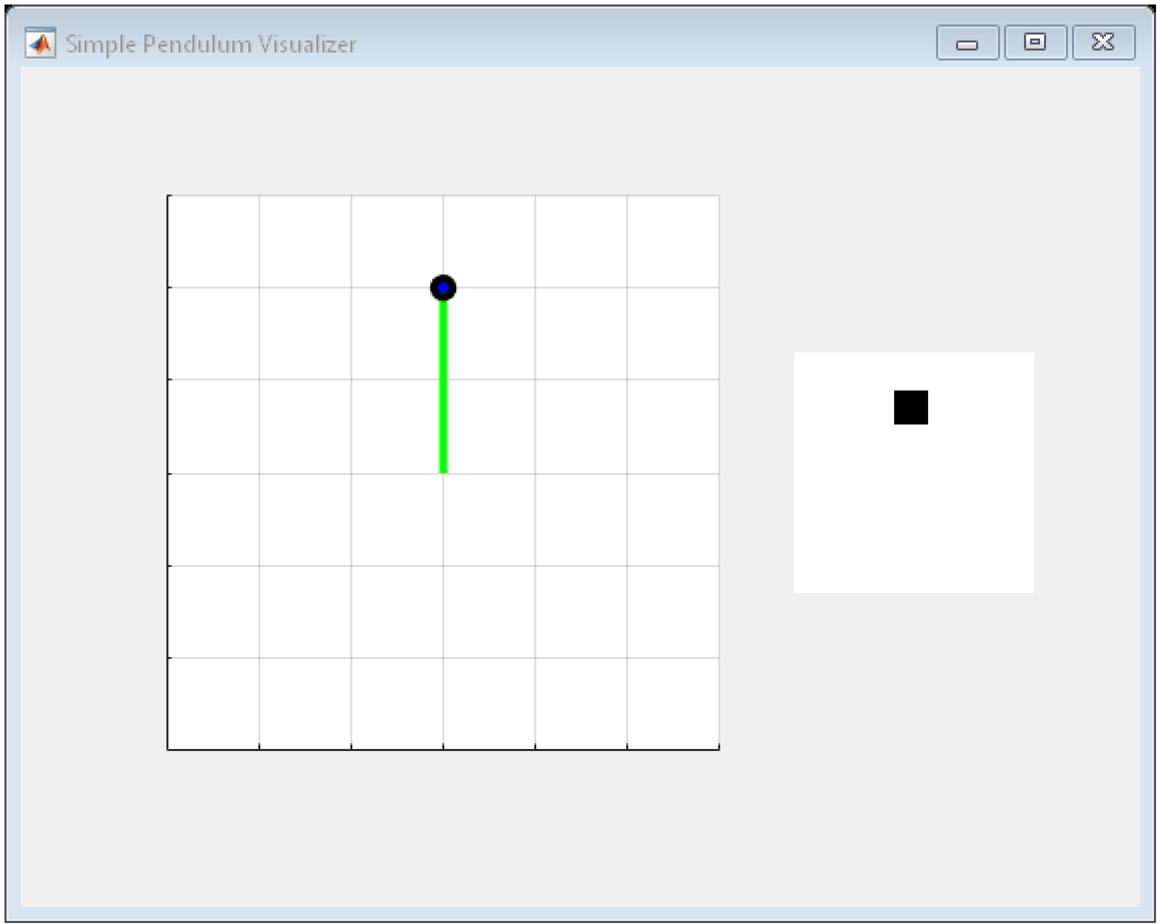
- Run each training for at most 5000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –1000 over the default window length of five consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=5000,...
    MaxStepsPerEpisode=500,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-1000);
```

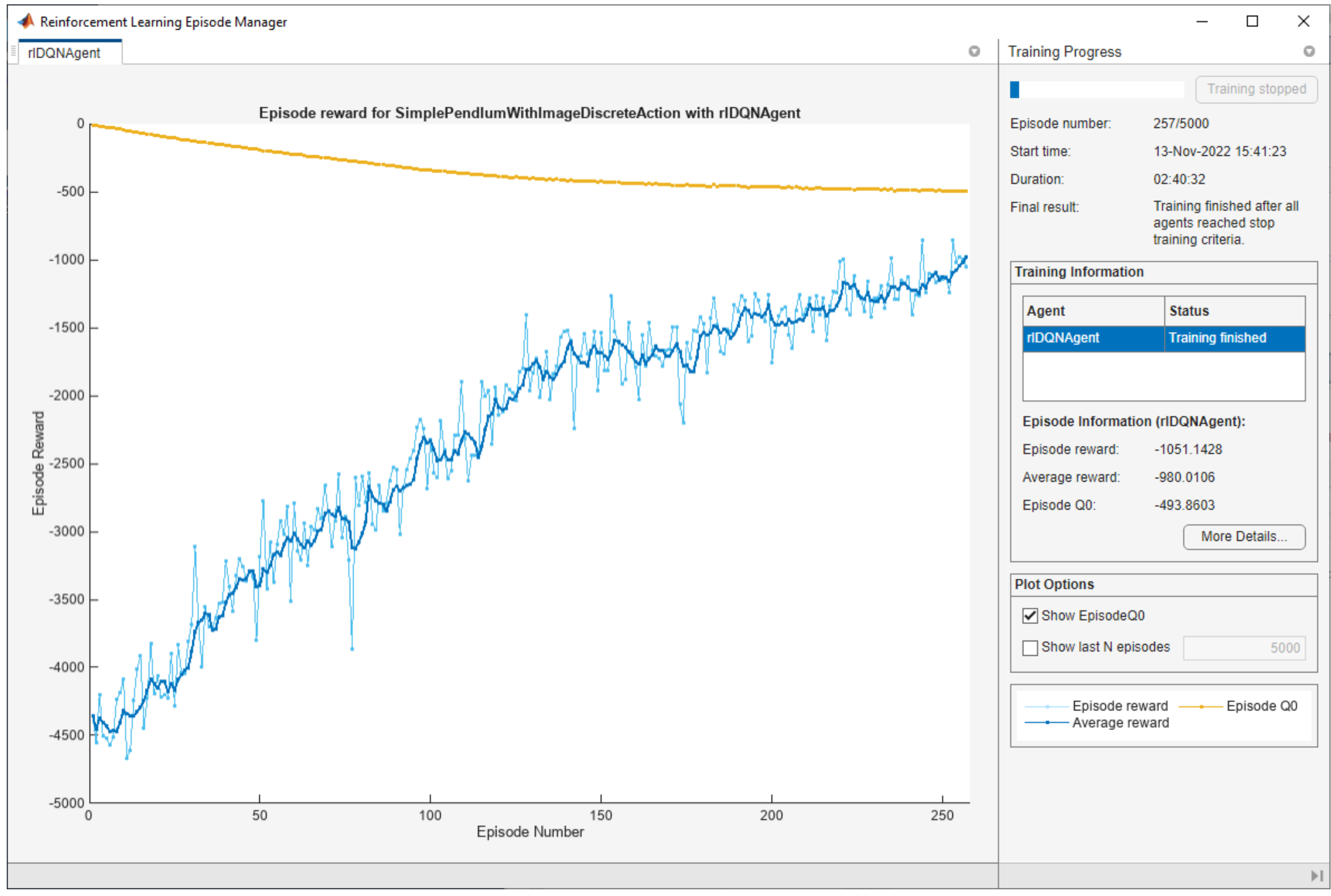
Visualize the pendulum system during training or simulation using the `plot` function.

```
plot(env)
```




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

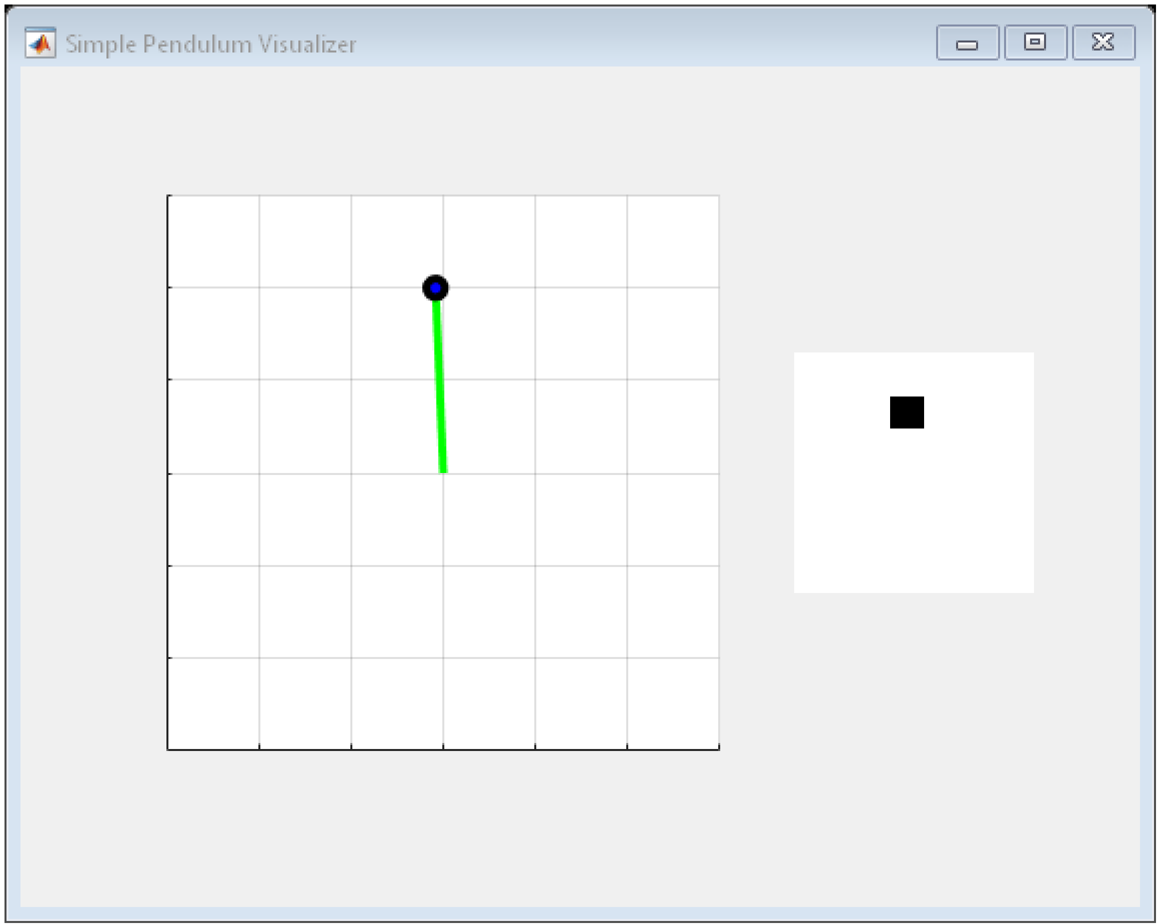
```
doTraining = false;

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




## Simulate DQN Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




```
totalReward = sum(experience.Reward)
```

```
totalReward = -713.0336
```

## See Also

**Apps**
**Deep Network Designer** | **Reinforcement Learning Designer**

**Functions**
`train` | `sim`

**Objects**
`rlDQNAgent` | `rlDQNAgentOptions` | `rlTrainingOptions`

## Related Examples

- "Train DQN Agent to Swing Up and Balance Pendulum" on page 5-89
- "Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation" on page 5-141

## More About


- “Load Predefined Control System Environments” on page 2-23
- “Deep Q-Network (DQN) Agents” on page 3-23
- “Create Policies and Value Functions” on page 4-2
- “Train Reinforcement Learning Agents” on page 5-3

# Train AC Agent to Balance Cart-Pole System Using Parallel Computing

This example shows how to train an actor-critic (AC) agent to balance a cart-pole system modeled in MATLAB® by using asynchronous parallel training. For an example that shows how to train the agent without using parallel training, see “Train AC Agent to Balance Cart-Pole System” on page 5-63.

### Actor Parallel Training

When you use parallel computing with AC agents, each worker generates experiences from its copy of the agent and the environment. After every `N` steps, the worker computes gradients from the experiences and sends the computed gradients back to the client agent (the agent associated with the MATLAB® process which starts the training). The client averages the gradients, updates the network parameters and sends the updated parameters back to the workers to they can continue simulating the agent with the new parameters.

This type of parallel training is also known as gradient-based parallelization, and allows you to achieve, in principle, a speed improvement which is nearly linear in the number of workers. However, this option requires synchronous training (that is the `Mode` property of the `rlTrainingOptions` object that you pass to the train function must be set to `sync`). This means that workers must pause execution until all workers are finished, and as a result the training only advances as fast as the slowest worker allows.

For more information about synchronous versus asynchronous parallelization, see “Train Agents Using Parallel Computing and GPUs” on page 5-8.

### Create Cart-Pole MATLAB Environment Interface

Create a predefined environment interface for the cart-pole system. For more information on this environment, see “Load Predefined Control System Environments” on page 2-23.

```
env = rlPredefinedEnv("CartPole-Discrete");
env.PenaltyForFalling = -10;
```

Obtain the observation and action information from the environment interface.

```
obsInfo = getObservationInfo(env);
numObservations = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create AC Agent

Actor-critic agents use a parametrized value function approximator to estimate the value of the policy. A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (which receives the content of the observation channel, as specified by obs`Info`) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` returns the total number of

dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

Define the network as an array of layer objects.

```
criticNetwork = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Create the value function approximator object using `criticNetwork` and the environment action and observation specifications.

```
critic = rlValueFunction(criticNetwork,obsInfo);
```

Actor-critic agents use a parametrized stochastic policy, which for discrete action spaces is implemented by a discrete categorical actor. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by `actInfo`. Note that `numel(actInfo.Dimension)` returns the number of elements of the discrete action space.

Define the network as an array of layer objects.

```
actorNetwork = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(numel(actInfo.Dimension))];
```

Create the actor approximator object using `actorNetwork` and the environment action and observation specifications.

```
actor = rlDiscreteCategoricalActor(actorNetwork,obsInfo,actInfo);
```

For more information on creating approximator objects such as actors and critics, see "Create Policies and Value Functions" on page 4-2.

Specify options for the critic and actor using `rlOptimizerOptions`.

```
criticOpts = rlOptimizerOptions(LearnRate=1e-2,GradientThreshold=1);
actorOpts = rlOptimizerOptions(LearnRate=1e-2,GradientThreshold=1);
```

Specify the AC agent options using `rlACAgentOptions`, include the training options for the actor and critic.

```
agentOpts = rlACAgentOptions(...
    ActorOptimizerOptions=actorOpts,...
    CriticOptimizerOptions=criticOpts,...
    EntropyLossWeight=0.01);
```

Create the agent using the specified actor representation and agent options. For more information, see `rlACAgent`.

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

### Parallel Training Options

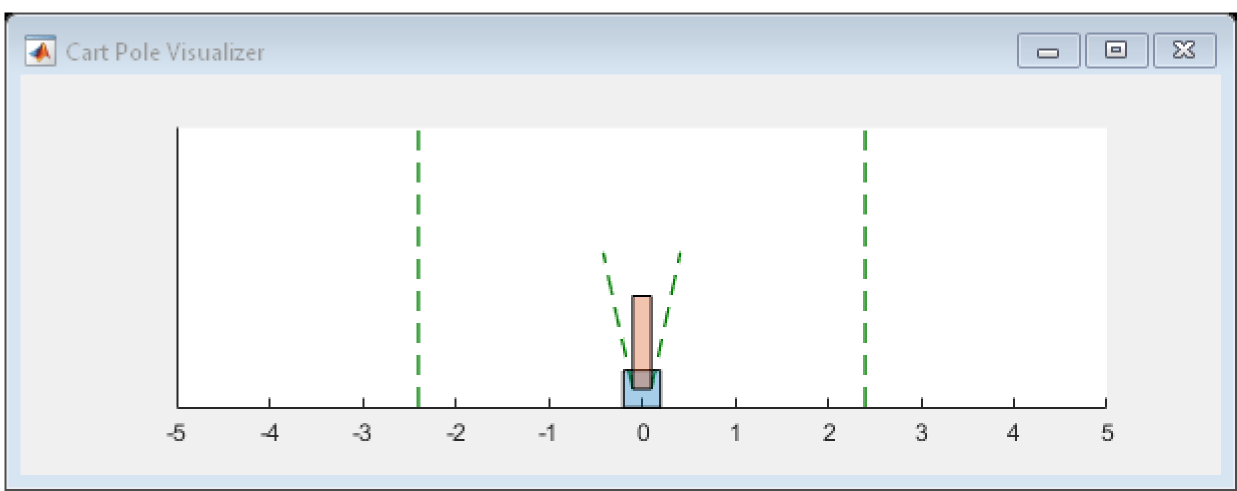
To train the agent, first specify the training options. For this example, use the following options.

- Run each training for at most `1000` episodes, with each episode lasting at most `500` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option).
- Stop training when the agent receives an average cumulative reward greater than `500` over `10` consecutive episodes. At this point, the agent can balance the pendulum in the upright position.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=500,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=500,...
    ScoreAveragingWindowLength=10);
```

You can visualize the cart-pole system can during training or simulation using the `plot` function.

```
plot(env)
```




To train the agent using parallel computing, specify the following training options.

- Set the `UseParallel` option to `True`.
- Train the agent in parallel asynchronously by setting the `ParallelizationOptions.Mode` option to `"async"`.

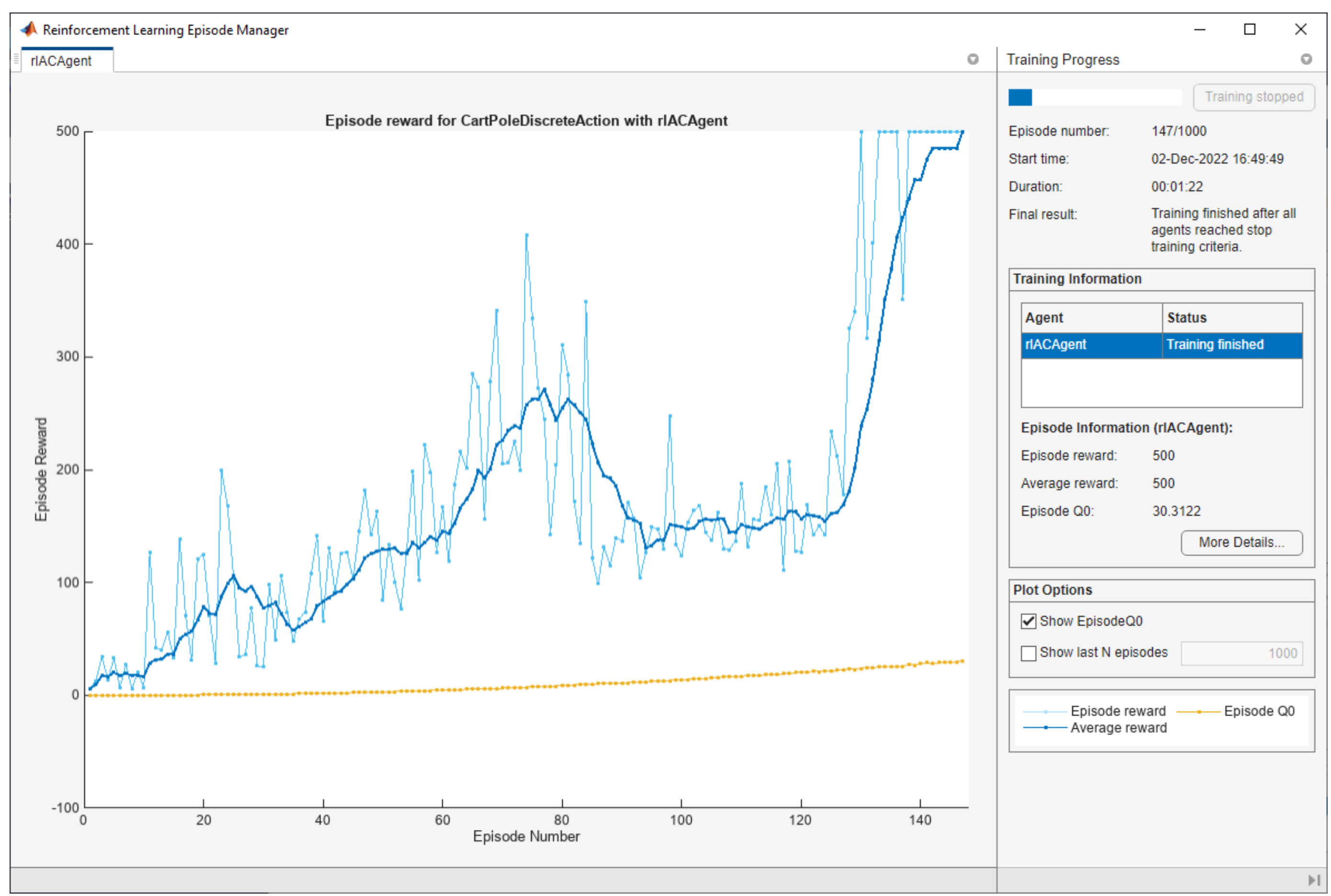
```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = "async";
```

For more information, see `rlTrainingOptions`.

### Train Agent

Train the agent using the `train` function. Training the agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`. Due to randomness in the asynchronous parallel training, you can expect different training results from the following training plot. The plot shows the result of training with six workers.

```
doTraining = false;

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



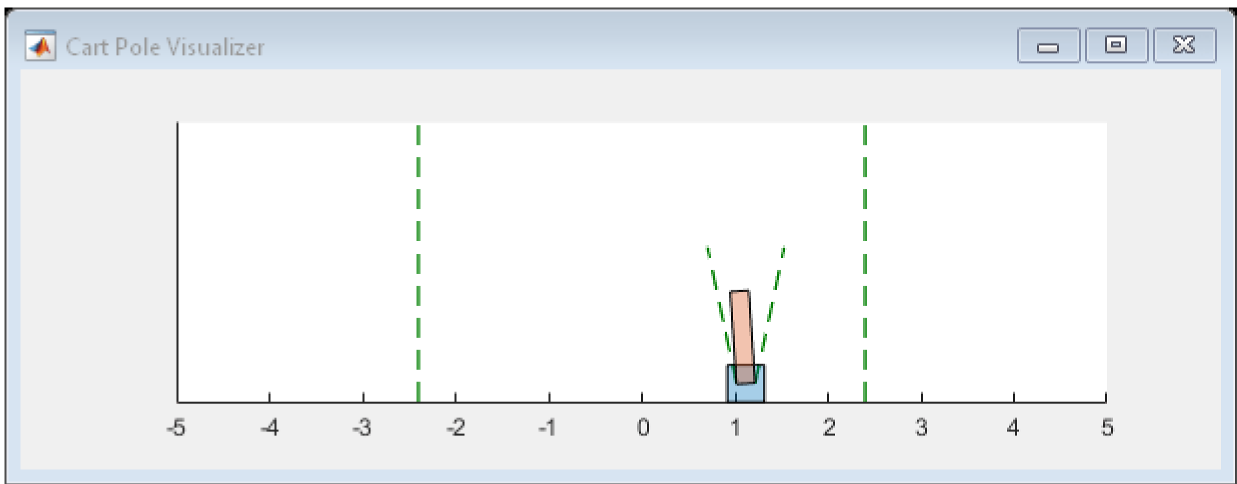

### Simulate AC Agent

You can visualize the cart-pole system with the plot function during simulation.

```
plot(env)
```

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




```
totalReward = sum(experience.Reward)
```

```
totalReward = 500
```

**References**

[1] Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 'Asynchronous Methods for Deep Reinforcement Learning'. *ArXiv:1602.01783 [Cs]*, 16 June 2016. https://arxiv.org/abs/1602.01783.

## See Also

**Apps**
**Reinforcement Learning Designer**

**Functions**
`train` | `sim`

**Objects**
`rlACAgent` | `rlACAgentOptions` | `rlTrainingOptions`

## Related Examples

- "Configure Options for A3C Training"
- "Train AC Agent to Balance Cart-Pole System" on page 5-63
- "Train DQN Agent for Lane Keeping Assist Using Parallel Computing" on page 5-257
- "Train Biped Robot to Walk Using Reinforcement Learning Agents" on page 5-267

## More About


- “Load Predefined Control System Environments” on page 2-23
- “Actor-Critic (AC) Agents” on page 3-31
- “Create Policies and Value Functions” on page 4-2
- “Train Agents Using Parallel Computing and GPUs” on page 5-8

# Train DDPG Agent to Control Flying Robot

This example shows how to train a deep deterministic policy gradient (DDPG) agent to generate trajectories for a flying robot modeled in Simulink®. For more information on DDPG agents, see "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40.

### Flying Robot Model

The reinforcement learning environment for this example is a flying robot with its initial condition randomized around a ring having a radius of 15 m. The orientation of the robot is also randomized. The robot has two thrusters mounted on the side of the body that are used to propel and steer the robot. The training goal is to drive the robot from its initial condition to the origin facing east.

Open the model.

```
mdl = "rlFlyingRobotEnv";
open_system(mdl)
```

Set the initial model state variables.

```
theta0 = 0;
x0 = -15;
y0 = 0;
```

Define the sample time `Ts` and the simulation duration `Tf`.

```
Ts = 0.4;
Tf = 30;
```

For this model:

- The goal orientation is `0` rad (robot facing east).
- The thrust from each actuator is bounded from -1 to 1 N
- The observations from the environment are the position, orientation (sine and cosine of orientation), velocity, and angular velocity of the robot.
- The reward $r_t$ provided at every time step is

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

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

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

$$r_t = r_1 + r_2 + r_3$$

where:

- $x_t$ is the position of the robot along the x-axis.
- $y_t$ is the position of the robot along the y-axis.
- $\theta_t$ is the orientation of the robot.

- $L_{t-1}$ is the control effort from the left thruster.
- $R_{t-1}$ is the control effort from the right thruster.
- $r_1$ is the reward when the robot is close to the goal.
- $r_2$ is the penalty when the robot drives beyond 20 m in either the x or y direction. The simulation is terminated when $r_2 < 0$.
- $r_3$ is a QR penalty that penalizes distance from the goal and control effort.

### Create Integrated Model

To train an agent for the `FlyingRobotEnv` model, use the `createIntegratedEnv` function to automatically generate a Simulink model containing an RL Agent block that is ready for training.

```
integratedMdl = "IntegratedFlyingRobot";
[~,agentBlk,obsInfo,actInfo] = ...
    createIntegratedEnv(mdl,integratedMdl);
```

### Actions and Observations

Before creating the environment object, specify names for the observation and action specifications, and bound the thrust actions between -1 and 1.

The observation vector for this environment is $\left[x\ \ y\ \ \dot{x}\ \ \dot{y}\ \ \sin(\theta)\ \ \cos(\theta)\ \ \dot{\theta}\right]^T$. Assign a name to the environment observation channel.

```
obsInfo.Name = "observations";
```

The action vector for this environment is $[T_R\ \ T_L]^T$. Assign a name, as well as upper and lower limits, to the environment action channel.

```
actInfo.Name = "thrusts";
actInfo.LowerLimit = -ones(prod(actInfo.Dimension),1);
actInfo.UpperLimit =  ones(prod(actInfo.Dimension),1);
```

Note that `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` return the number of dimensions of the observation and action spaces, respectively, regardless of whether they are arranged as row vectors, column vectors, or matrices.

### Create Environment Object

Create an environment object using the integrated Simulink model.

```
env = rlSimulinkEnv( ...
    integratedMdl, ...
    agentBlk, ...
    obsInfo, ...
    actInfo);
```

### Reset Function

Create a custom reset function that randomizes the initial position of the robot along a ring of radius 15 m and the initial orientation. For details on the reset function, see `flyingRobotResetFcn`.

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create DDPG agent

DDPG agents use a parametrized Q-value function approximator to estimate the value of the policy. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward given the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value).

Define each network path as an array of layer objects. Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% Specify the number of outputs for the hidden layers.
hiddenLayerSize = 100;

% Define observation path layers
observationPath = [
    featureInputLayer( ...
        prod(obsInfo.Dimension),Name="obsInLyr")
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(hiddenLayerSize)
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(1,Name="fc4")
    ];

% Define action path layers
actionPath = [
    featureInputLayer( ...
        prod(actInfo.Dimension), ...
        Name="actInLyr")
    fullyConnectedLayer(hiddenLayerSize,Name="fc5")
    ];

% Create the layer graph.
criticNetwork = layerGraph(observationPath);
criticNetwork = addLayers(criticNetwork,actionPath);

% Connect actionPath to observationPath.
criticNetwork = connectLayers(criticNetwork,"fc5","add/in2");

% Create dlnetwork from layer graph
criticNetwork = dlnetwork(criticNetwork);

% Display the number of parameters
summary(criticNetwork)
```

```
   Initialized: true

   Number of learnables: 21.4k

   Inputs:
      1   'obsInLyr'   7 features
      2   'actInLyr'   2 features
```

Create the critic using `criticNetwork`, the environment specifications, and the names of the network input layers to be connected to the observation and action channels. For more information see `rlQValueFunction`.

```
critic = rlQValueFunction(criticNetwork,obsInfo,actInfo,...
    ObservationInputNames="obsInLyr",ActionInputNames="actInLyr");
```

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects.

```
actorNetwork = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension))
    tanhLayer
    ];
```

Convert the array of layer object to a `dlnetwork` object and display the number of parameters.

```
actorNetwork = dlnetwork(actorNetwork);
summary(actorNetwork)
```

```
   Initialized: true

   Number of learnables: 21.2k

   Inputs:
      1   'input'   7 features
```

Define the actor using `actorNetwork`, and the specifications for the action and observation channels. For more information, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNetwork,obsInfo,actInfo);
```

Specify options for the critic and the actor using `rlOptimizerOptions`.

```
criticOptions = rlOptimizerOptions(LearnRate=1e-03,GradientThreshold=1);
actorOptions = rlOptimizerOptions(LearnRate=1e-04,GradientThreshold=1);
```

Specify the DDPG agent options using `rlDDPGAgentOptions`, include the training options for the actor and critic.

```
agentOptions = rlDDPGAgentOptions(...
    SampleTime=Ts,...
    ActorOptimizerOptions=actorOptions,...
    CriticOptimizerOptions=criticOptions,...
    ExperienceBufferLength=1e6 ,...
    MiniBatchSize=256);
agentOptions.NoiseOptions.Variance = 1e-1;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;
```

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

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

Alternatively, you can create the agent first, and then access its option object and modify the options using dot notation.

### Train Agent

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

- Run each training for at most `20000` episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than `415` over 10 consecutive episodes. At this point, the agent can drive the flying robot to the goal position.
- Save a copy of the agent for each episode where the cumulative reward is greater than `415`.

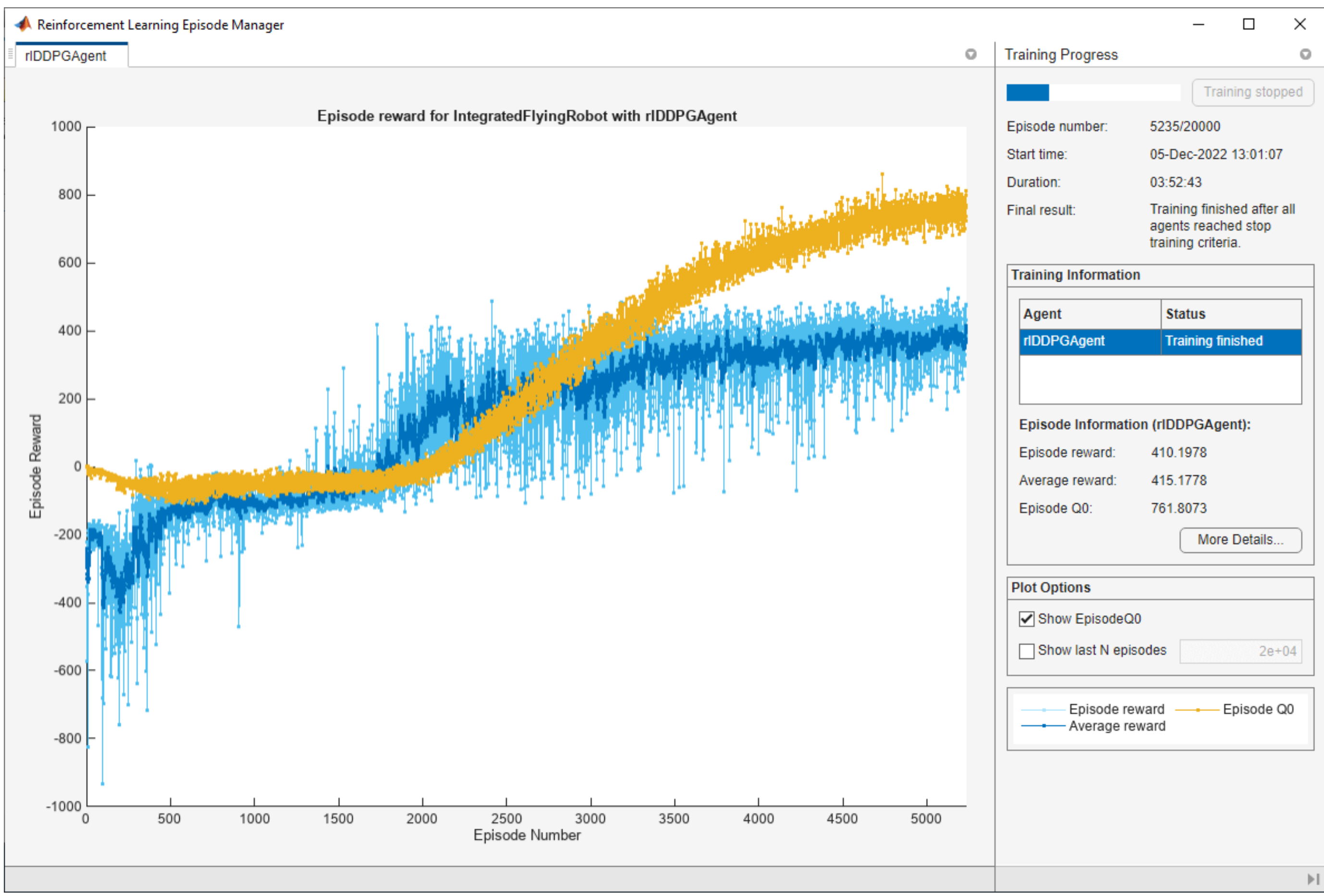
For more information, see `rlTrainingOptions`.

```
maxepisodes = 20000;
maxsteps = ceil(Tf/Ts);
trainingOptions = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    StopOnError="on",...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=415,...
    ScoreAveragingWindowLength=10,...
    SaveAgentCriteria="EpisodeReward",...
    SaveAgentValue=415);
```

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

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
```

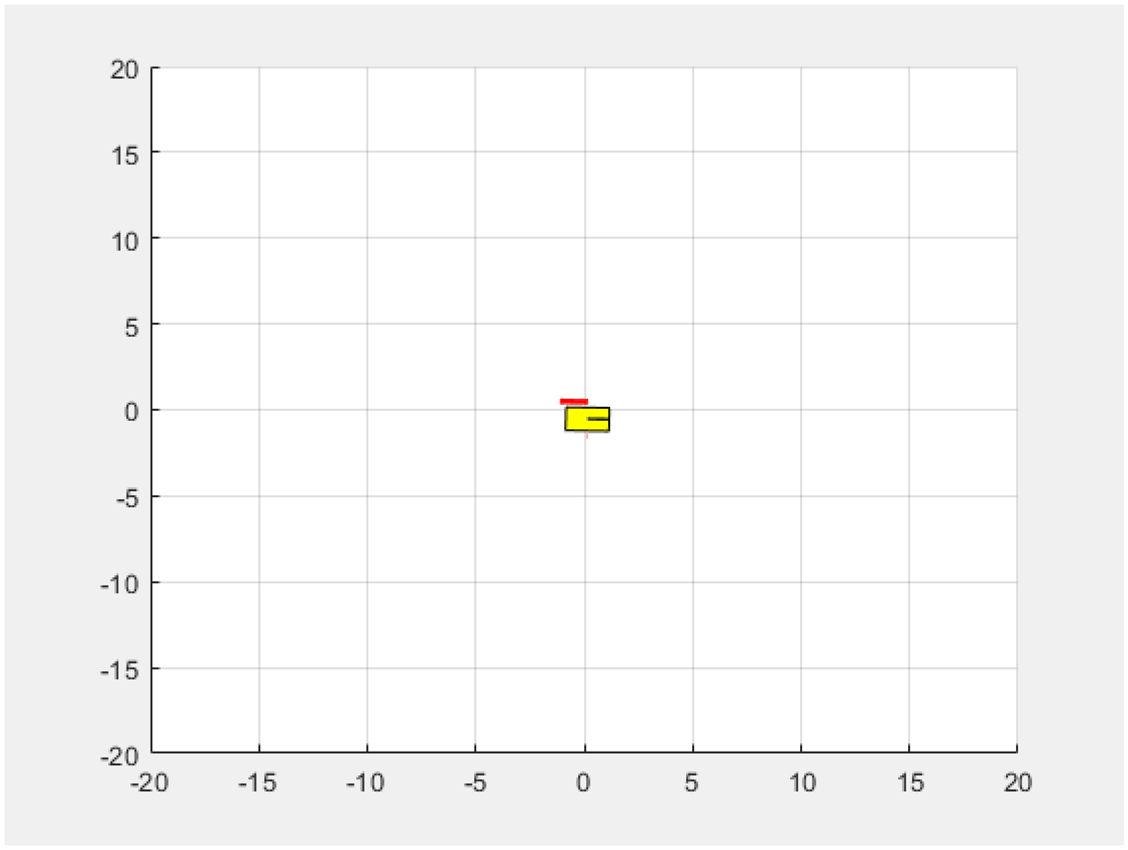
```
else
    % Load the pretrained agent for the example.
    load("FlyingRobotDDPG.mat","agent")
end
```




### Simulate DDPG Agent

To validate the performance of the trained agent, simulate the agent within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=maxsteps);
experience = sim(env,agent,simOptions);
```




## See Also

**Functions**
`train` | `sim` | `createIntegratedEnv` | `rlSimulinkEnv`

**Objects**
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlQValueFunction` | `rlContinuousDeterministicActor` | `rlTrainingOptions` | `rlSimulationOptions` | `rlOptimizerOptions`

**Blocks**
RL Agent

## Related Examples

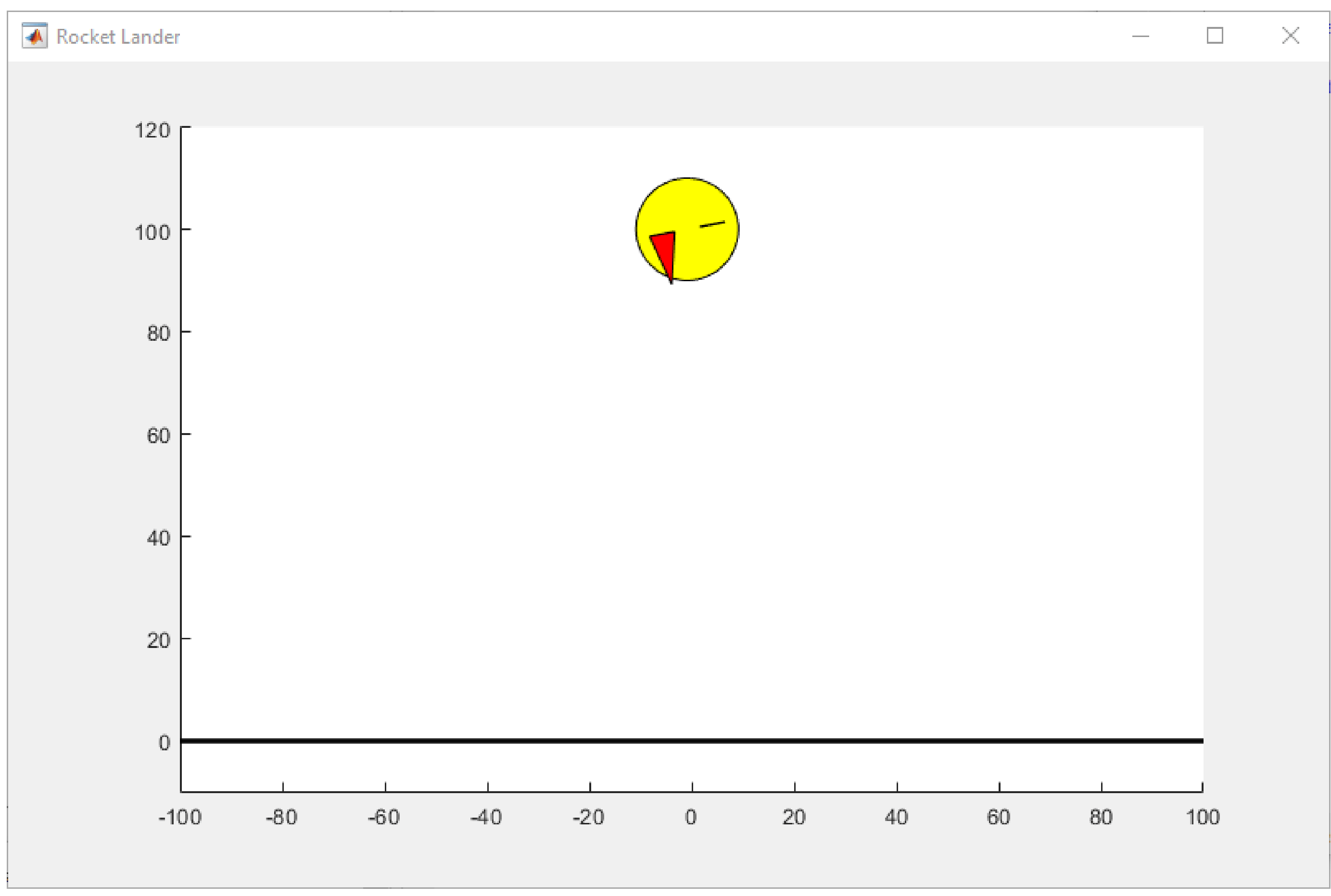
- “Train DDPG Agent to Swing Up and Balance Cart-Pole System” on page 5-106
- “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation” on page 5-141
- “Train PPO Agent for a Lander Vehicle” on page 5-180
- “Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC” (Model Predictive Control Toolbox)

## More About


- “Create Simulink Reinforcement Learning Environments” on page 2-8
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Train Reinforcement Learning Agents” on page 5-3

# Train PPO Agent for a Lander Vehicle

This example shows how to train a proximal policy optimization (PPO) agent with a discrete action space to land an airborne vehicle on the ground. For more information on PPO agents, see “Proximal Policy Optimization (PPO) Agents” on page 3-49.




### Environment

The environment in this example is a lander vehicle represented by a 3-DOF circular disc with mass. The vehicle has two thrusters for forward and rotational motion. Gravity acts vertically downwards, and there are no aerodynamic drag forces. The training goal is to make the vehicle land on the ground at a specified location.

For this environment:

- Motion of the lander vehicle is bounded in X (horizontal axis) from -100 to 100 meters and Y (vertical axis) from 0 to 120 meters.
- The goal position is at (0,0) meters and the goal orientation is 0 radians.
- The maximum thrust applied by each thruster is 8.5 N.
- The sample time is 0.1 seconds.

- The observations from the environment are the vehicle's position $(x, y)$, orientation $(\theta)$, velocity $(\dot{x}, \dot{y})$, angular velocity $\left(\dot{\theta}\right)$, and a sensor reading that detects rough landing (-1), soft landing (1) or airborne (0) condition. The observations are normalized between -1 and 1.
- The environment has a discrete action space. At every time step, the agent selects one of the following nine discrete action pairs:

$L,\ L$ − do nothing
$L,\ M$ − fire right (med)
$L,\ H$ − fire right (high)
$M,\ L$ − fire left (med)
$M,\ M$ − fire left (med) + right (med)
$M,\ H$ − fire left (med) + right (high)
$H,\ L$ − fire left (high)
$H,\ M$ − fire left (high) + right (med)
$H,\ H$ − fire left (high) + right (high)

Here, $L = 0.0, M = 0.5$ and $H = 1.0$ are normalized thrust values for each thruster. The environment `step` function scales these values to determine the actual thrust values.

- At the beginning of every episode, the vehicle starts from a random initial $x$ position and orientation. The altitude is always reset to 100 meters.
- The reward $r_t$ provided at the time step $t$ is as follows.

$$r_t = (s_t - s_{t-1}) - 0.1\theta_t^2 - 0.01\left(L_t^2 + R_t^2\right) + 500c$$

$$s_t = 1 - \left(\sqrt{\hat{d}_t} + \frac{\sqrt{\hat{v}_t}}{2}\right)$$

$$c = (y_t \le 0)\ \&\&\ (\dot{y}_t \ge\ -0.5\ \&\&\ |\dot{x}_t| \le 0.5)$$

Here:

- $x_t$,$y_t$,$\dot{x}_t$, and $\dot{y}_t$ are the positions and velocities of the lander vehicle along the x and y axes.
- $\hat{d}_t = \sqrt{{x_t}^2 + {y_t}^2}/d_{\max}$ is the normalized distance of the lander vehicle from the goal position.
- $\hat{v}_t = \sqrt{{\dot{x}_t}^2 + {\dot{y}_t}^2}/v_{\max}$ is the normalized speed of the lander vehicle.
- $d_{\max}$ and $v_{\max}$ are the maximum distances and speeds.
- $\theta_t$ is the orientation with respect to the vertical axis.
- $L_t$ and $R_t$ are the action values for the left and right thrusters.
- $c$ is a sparse reward for soft-landing with horizontal and vertical velocities less than 0.5 m/s.

### Create MATLAB Environment

Create a MATLAB environment for the lander vehicle using the lander `LanderVehicle` class.

```
env = LanderVehicle()
```

```
env = 
  LanderVehicle with properties:
```

```
                Mass: 1
                  L1: 10
                  L2: 5
             Gravity: 9.8060
        ThrustLimits: [0 8.5000]
                  Ts: 0.1000
               State: [6x1 double]
          LastAction: [2x1 double]
         LastShaping: 0
    DistanceIntegral: 0
    VelocityIntegral: 0
           TimeCount: 0
```

Obtain the observation and action specifications from the environment.

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

The training can be sensitive to the initial network weights and biases, and results can vary with different sets of values. The network weights are randomly initialized to small values in this example. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

#### Create PPO Agent

PPO agents use a parametrized value function approximator to estimate the value of the policy. A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (which receives the content of the observation channel, as specified by `obsInfo`) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` returns the total number of dimensions of the observation space regardless of whether the observation space is a column vector, row vector, or matrix.

```
numObs = prod(obsInfo.Dimension);
criticLayerSizes = [400 300];
actorLayerSizes = [400 300];
```

Define the network as an array of layer objects.

```
criticNetwork = [
    featureInputLayer(numObs)
    fullyConnectedLayer(criticLayerSizes(1), ...
        Weights=sqrt(2/numObs)*...
            (rand(criticLayerSizes(1),numObs)-0.5), ...
        Bias=1e-3*ones(criticLayerSizes(1),1))
    reluLayer
    fullyConnectedLayer(criticLayerSizes(2), ...
        Weights=sqrt(2/criticLayerSizes(1))*...
            (rand(criticLayerSizes(2),criticLayerSizes(1))-0.5), ...
        Bias=1e-3*ones(criticLayerSizes(2),1))
    reluLayer
    fullyConnectedLayer(1, ...
```

```
        Weights=sqrt(2/criticLayerSizes(2))* ...
            (rand(1,criticLayerSizes(2))-0.5), ...
        Bias=1e-3)
    ];
```

Convert to `dlnetwork` and display the number of weights.

```
criticNetwork = dlnetwork(criticNetwork);
summary(criticNetwork)
```

```
   Initialized: true

   Number of learnables: 123.8k

   Inputs:
      1   'input'   7 features
```

Create the critic approximator object using `criticNet` and the observation specification. For more information on value function approximators, see `rlValueFunction`.

```
critic = rlValueFunction(criticNetwork,obsInfo);
```

Policy gradient agents use a parametrized stochastic policy, which for discrete action spaces is implemented by a discrete categorical actor. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by `actInfo`. Note that `numel(actInfo.Dimension)` returns the number of elements of the discrete action space.

Define the network as an array of layer objects.

```
actorNetwork = [
    featureInputLayer(numObs)
    fullyConnectedLayer(actorLayerSizes(1), ...
        Weights=sqrt(2/numObs)*...
            (rand(actorLayerSizes(1),numObs)-0.5), ...
        Bias=1e-3*ones(actorLayerSizes(1),1))
    reluLayer
    fullyConnectedLayer(actorLayerSizes(2), ...
        Weights=sqrt(2/actorLayerSizes(1))*...
            (rand(actorLayerSizes(2),actorLayerSizes(1))-0.5), ...
        Bias=1e-3*ones(actorLayerSizes(2),1))
    reluLayer
    fullyConnectedLayer(numel(actInfo.Elements), ...
        Weights=sqrt(2/actorLayerSizes(2))*...
            (rand(numel(actInfo.Elements),actorLayerSizes(2))-0.5), ...
        Bias=1e-3*ones(numel(actInfo.Elements),1))
    softmaxLayer
    ];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNetwork = dlnetwork(actorNetwork);
summary(actorNetwork)
```

```
   Initialized: true

   Number of learnables: 126.2k

   Inputs:
      1   'input'   7 features
```

Create the actor using `actorNet` and the observation and action specifications. For more information on discrete categorical actors, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNetwork,obsInfo,actInfo);
```

Specify training options for the critic and the actor using `rlOptimizerOptions`.

```
actorOpts = rlOptimizerOptions(LearnRate=1e-4);
criticOpts = rlOptimizerOptions(LearnRate=1e-4);
```

Specify the agent hyperparameters using an `rlPPOAgentOptions` object, include the training options for the actor and critic.

```
agentOpts = rlPPOAgentOptions(...
    ExperienceHorizon=600,...
    ClipFactor=0.02,...
    EntropyLossWeight=0.01,...
    ActorOptimizerOptions=actorOpts,...
    CriticOptimizerOptions=criticOpts,...
    NumEpoch=3,...
    AdvantageEstimateMethod="gae",...
    GAEFactor=0.95,...
    SampleTime=0.1,...
    DiscountFactor=0.997);
```

For these hyperparameters:

- The agent collects experiences until it reaches the experience horizon of 600 steps or episode termination and then trains from mini-batches of 128 experiences for 3 epochs.
- For improving training stability, use an objective function clip factor of 0.02.
- A discount factor value of 0.997 promotes long term rewards.
- Variance in critic output is reduced by using the Generalized Advantage Estimate method with a GAE factor of 0.95.
- The `EntropyLossWeight` term of 0.01 enhances exploration during training.

Create the PPO agent.

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

Alternatively, you can create the agent first, and then access its option object and modify the options using dot notation.

#### Train Agent

To train the PPO agent, specify the following training options.

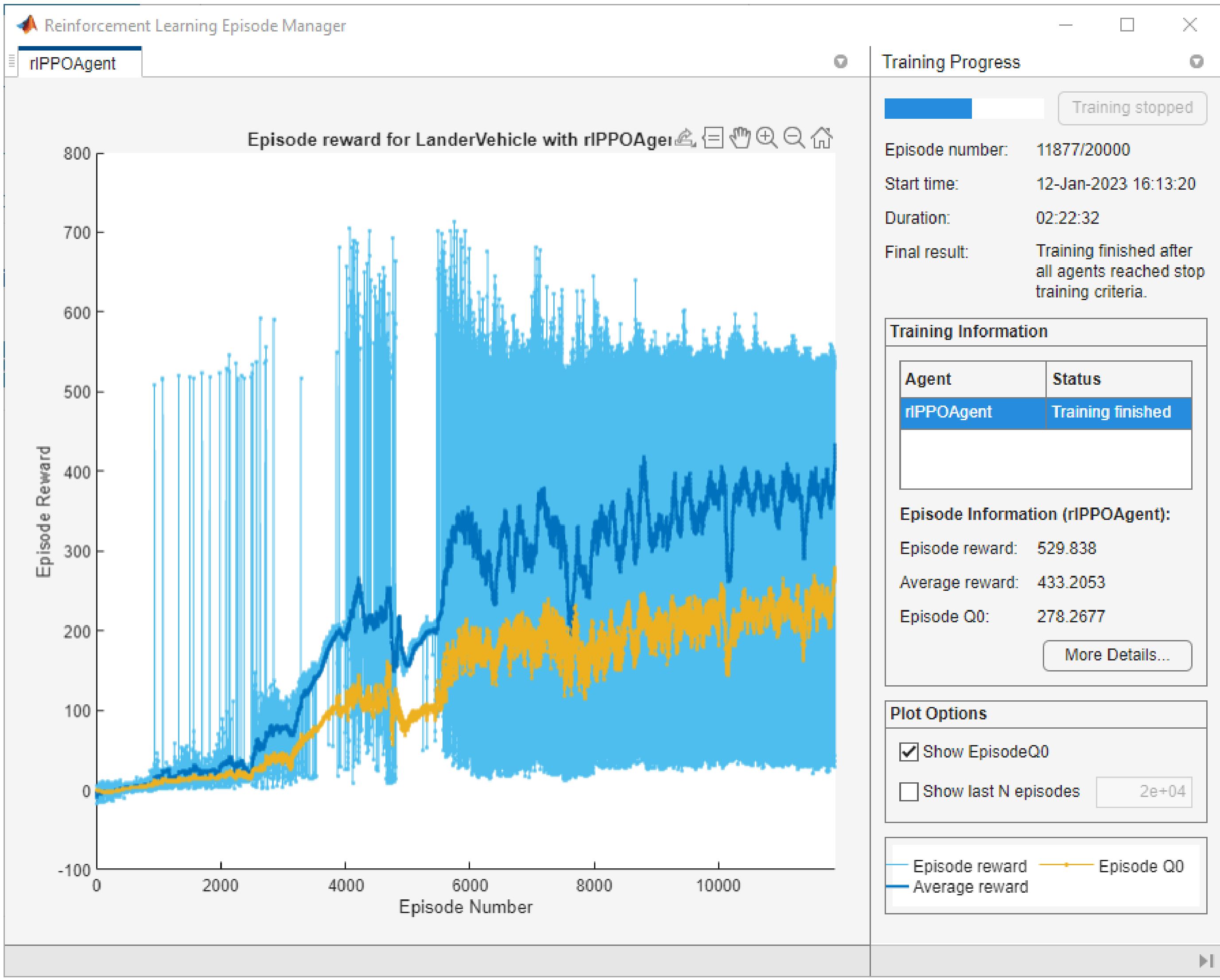
- Run the training for at most 20000 episodes, with each episode lasting at most 600 time steps.
- Stop the training when the average reward over 100 consecutive episodes is 450 or more.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=20000,...
    MaxStepsPerEpisode=600,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=430,...
    ScoreAveragingWindowLength=100);
```

Train the agent using the `train` function. Due to the complexity of the environment, training process is computationally intensive and takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`.

```
doTraining = false;
if doTraining
    trainingStats = train(agent, env, trainOpts);
else
    load("landerVehicleAgent.mat");
end
```

An example training session is shown below. The actual results may vary because of randomness in the training process.




### Simulate

Plot the environment first to create a visualization for the lander vehicle.

```
plot(env)
```

Set the random seed for simulation reproducibility.

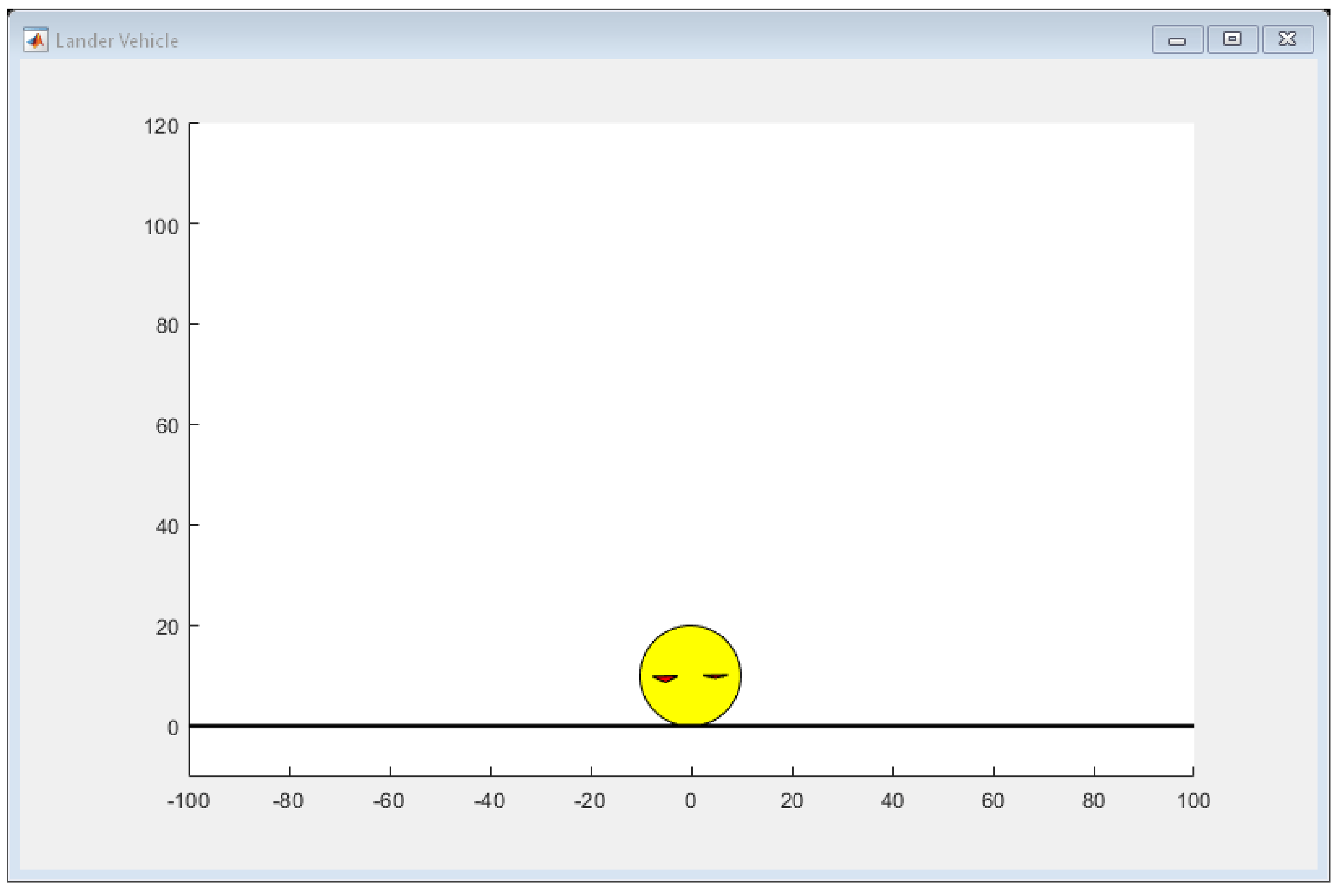
```
rng(10)
```

Set up simulation options to perform 5 simulations. For more information see `rlSimulationOptions`.

```
simOptions = rlSimulationOptions(MaxSteps=600);
simOptions.NumSimulations = 5;
```

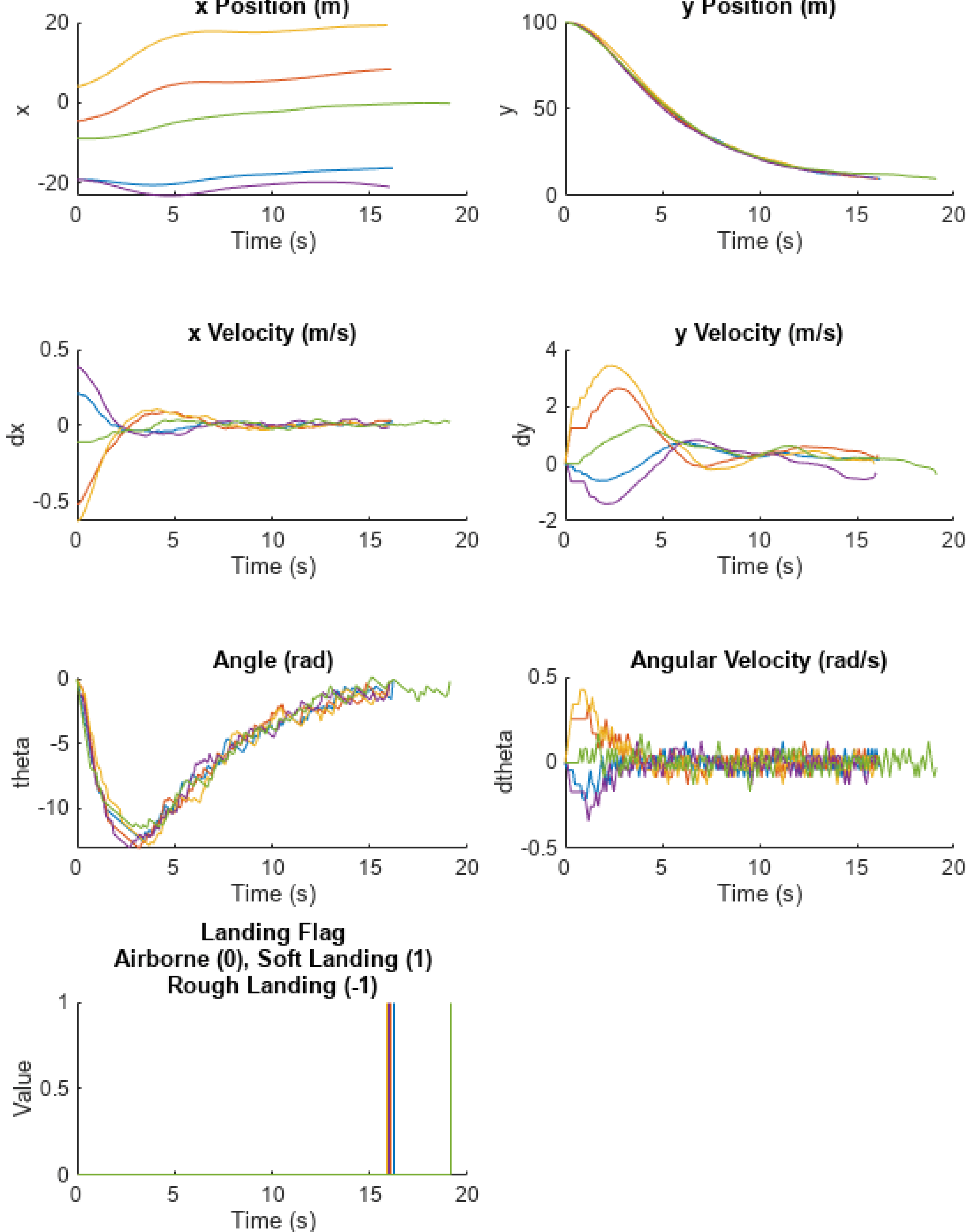
Simulate the trained agent within the environment. For more information see `sim`.

```
experience = sim(env, agent, simOptions);
```




Plot the time history of the states for all simulations using the helper function `plotLanderVehicleTrajectory` provided in the example folder.

```
% Observations to plot
obsToPlot = ["x", "y", "dx", "dy", "theta", "dtheta", "landing"];

% Create a figure
f = figure();
f.Position(3:4) = [800,1000];

% Create a tiled layout for the plots
t = tiledlayout(f, 4, 2, TileSpacing="compact");

% Plot the data
for ct = 1:numel(obsToPlot)
    ax = nexttile(t);
    plotLanderVehicleTrajectory(ax, experience, env, obsToPlot(ct));
end
```




## See Also

**Functions**

`train` | `sim`

**Objects**
`rlPPOAgent` | `rlPPOAgentOptions` | `rlTrainingOptions`

## Related Examples

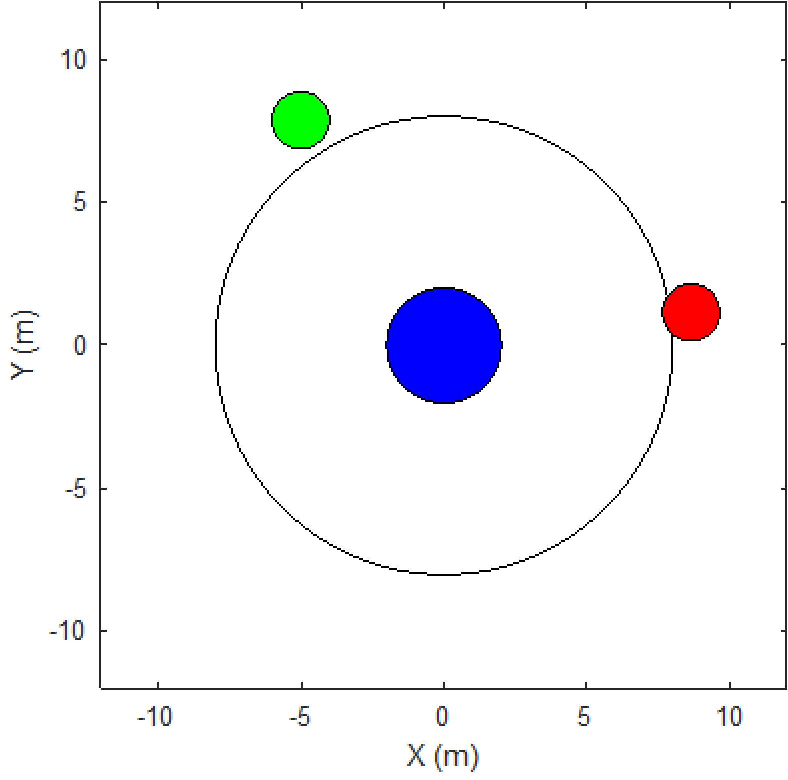
- “Train DDPG Agent to Control Flying Robot” on page 5-172
- “Train PPO Agent for Automatic Parking Valet” on page 5-235
- “Train Multiple Agents to Perform Collaborative Task” on page 5-190

## More About

- “Create MATLAB Reinforcement Learning Environments” on page 2-2
- “Proximal Policy Optimization (PPO) Agents” on page 3-49
- “Train Reinforcement Learning Agents” on page 5-3

# Train Multiple Agents to Perform Collaborative Task

This example shows how to set up a multi-agent training session on a Simulink® environment. In the example, you train two agents to collaboratively perform the task of moving an object.



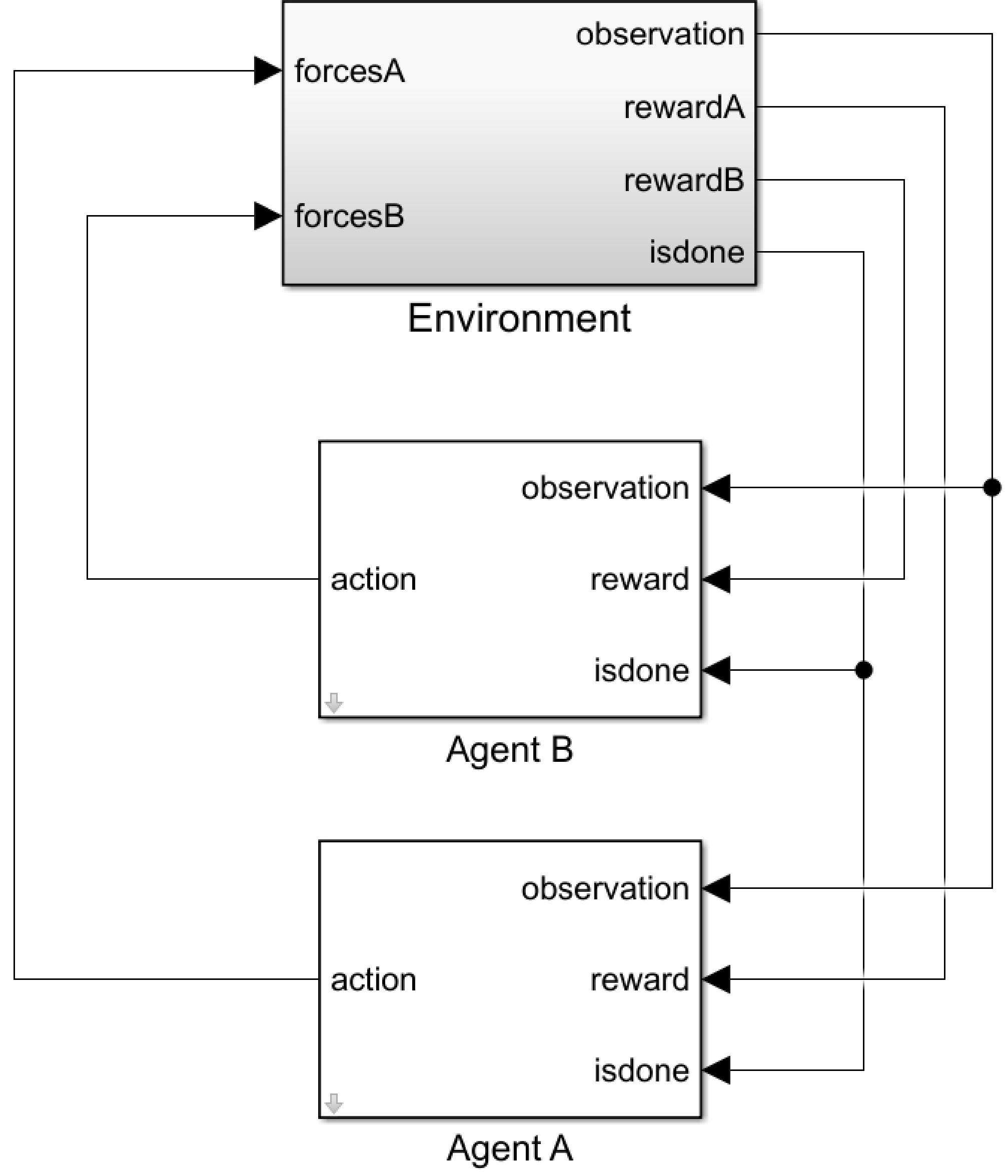

The environment in this example is a frictionless two dimensional surface containing elements represented by circles. A target object C is represented by the blue circle with a radius of 2 m, and robots A (red) and B (green) are represented by smaller circles with radii of 1 m each. The robots attempt to move object C outside a circular ring of a radius 8 m by applying forces through collision. All elements within the environment have mass and obey Newton's laws of motion. In addition, contact forces between the elements and the environment boundaries are modeled as spring and mass damper systems. The elements can move on the surface through the application of externally applied forces in the X and Y directions. There is no motion in the third dimension and the total energy of the system is conserved.

Set the random seed and create the set of parameters required for this example.

```
rng(10)
rlCollaborativeTaskParams
```

Open the Simulink model.

```
mdl = "rlCollaborativeTask";
open_system(mdl)
```




For this environment:

- The 2-dimensional space is bounded from –12 m to 12 m in both the X and Y directions.
- The contact spring stiffness and damping values are 100 N/m and 0.1 N/m/s, respectively.
- The agents share the same observations for positions, velocities of A, B, and C and the action values from the last time step.
- The simulation terminates when object C moves outside the circular ring.
- At each time step, the agents receive the following reward:

$$r_A = r_{\text{global}} + r_{\text{local},A}$$
$$r_B = r_{\text{global}} + r_{\text{local},B}$$
$$r_{\text{global}} = 0.001 d_c$$
$$r_{\text{local},A} = -0.005 d_{\text{AC}} - 0.008 u_A^2$$
$$r_{\text{local},B} = -0.005 d_{\text{BC}} - 0.008 u_B^2$$

Here:

- $r_A$and $r_B$ are the rewards received by agents A and B, respectively.
- $r_{\text{global}}$ is a team reward that is received by both agents as object C moves closer towards the boundary of the ring.
- $r_{\text{local},A}$ and $r_{\text{local},B}$ are local penalties received by agents A and B based on their distances from object C and the magnitude of the action from the last time step.
- $d_C$ is the distance of object C from the center of the ring.
- $d_{\text{AC}}$ and $d_{\text{BC}}$ are the distances between agent A and object C and agent B and object C, respectively.
- The agents apply external forces on the robots that result in motion. $u_A$ and $u_B$ are the action values of the two agents A and B from the last time step. The range of action values is between -1 and 1.

#### Environment

To create a multi-agent environment, specify the block paths of the agents using a string array. Also, specify the observation and action specification objects using cell arrays. The order of the specification objects in the cell array must match the order specified in the block path array. When agents are available in the MATLAB workspace at the time of environment creation, the observation and action specification arrays are optional. For more information on creating multi-agent environments, see `rlSimulinkEnv`.

Create the I/O specifications for the environment. In this example, the agents are homogeneous and have the same I/O specifications.

```
% Number of observations
numObs = 16;

% Number of actions
numAct = 2;

% Maximum value of externally applied force (N)
maxF = 1.0;

% I/O specifications for each agent
oinfo = rlNumericSpec([numObs,1]);
ainfo = rlNumericSpec([numAct,1], ...
    UpperLimit= maxF, ...
    LowerLimit= -maxF);
oinfo.Name = "observations";
ainfo.Name = "forces";
```

Create the Simulink environment interface.

```
blks = ["rlCollaborativeTask/Agent A", "rlCollaborativeTask/Agent B"];
obsInfos = {oinfo,oinfo};
actInfos = {ainfo,ainfo};
env = rlSimulinkEnv(mdl,blks,obsInfos,actInfos);
```

Specify a reset function for the environment. The reset function `resetRobots` ensures that the robots start from random initial positions at the beginning of each episode.

```
env.ResetFcn = @(in) resetRobots(in,RA,RB,RC,boundaryR);
```

### Agents

This example uses two Proximal Policy Optimization (PPO) agents with continuous action spaces. The agents apply external forces on the robots that result in motion. To learn more about PPO agents, see “Proximal Policy Optimization (PPO) Agents” on page 3-49.

The agents collect experiences until the experience horizon (600 steps) is reached. After trajectory completion, the agents learn from mini-batches of 300 experiences. An objective function clip factor of 0.2 is used to improve training stability and a discount factor of 0.99 is used to encourage long-term rewards.

Specify the agent options for this example.

```
agentOptions = rlPPOAgentOptions(...
    ExperienceHorizon=600,...
    ClipFactor=0.2,...
    EntropyLossWeight=0.01,...
    MiniBatchSize=300,...
    NumEpoch=4,...
    AdvantageEstimateMethod="gae",...
    GAEFactor=0.95,...
    SampleTime=Ts,...
    DiscountFactor=0.99);
agentOptions.ActorOptimizerOptions.LearnRate  = 1e-4;
agentOptions.CriticOptimizerOptions.LearnRate = 1e-4;
```

Create the agents using the default agent creation syntax. For more information see `rlPPOAgent`.

```
agentA = rlPPOAgent(oinfo, ainfo, ...
    rlAgentInitializationOptions(NumHiddenUnit= 200), agentOptions);
agentB = rlPPOAgent(oinfo, ainfo, ...
    rlAgentInitializationOptions(NumHiddenUnit= 200), agentOptions);
```

### Training

To train multiple agents, you can pass an array of agents to the `train` function. The order of agents in the array must match the order of agent block paths specified during environment creation. Doing so ensures that the agent objects are linked to their appropriate I/O interfaces in the environment.

You can train multiple agents in a decentralized or centralized manner. In decentralized training, agents collect their own set of experiences during the episodes and learn independently from those experiences. In centralized training, the agents share the collected experiences and learn from them together. The actor and critic functions are synchronized between the agents after trajectory completion.

To configure a multi-agent training, you can create agent groups and specify a learning strategy for each group through the `rlMultiAgentTrainingOptions` object. Each agent group may contain

unique agent indices, and the learning strategy can be "centralized" or "decentralized". For example, you can use the following command to configure training for three agent groups with different learning strategies. The agents with indices [1,2] and [3,4] learn in a centralized manner while agent 4 learns in a decentralized manner.

```
opts = rlMultiAgentTrainingOptions(AgentGroups= {[1,2], 4, [3,5]},
LearningStrategy= ["centralized","decentralized","centralized"])
```

For more information on multi-agent training, type `help rlMultiAgentTrainingOptions` in MATLAB.

You can perform decentralized or centralized training by running one of the following sections using the Run Section button.

### 1. Decentralized Training

To configure decentralized multi-agent training for this example:

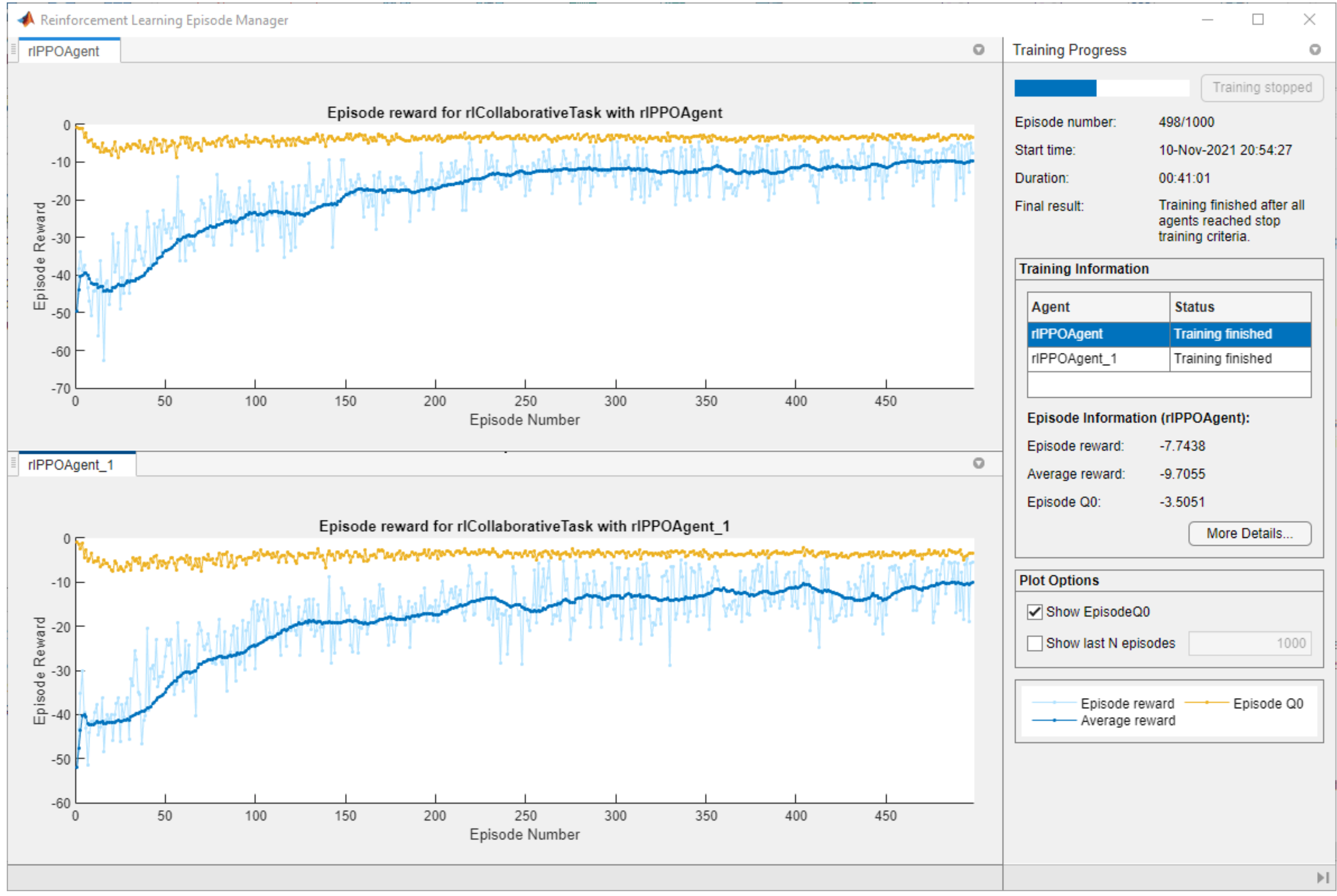
- Automatically assign agent groups using the `AgentGroups=auto` option. This allocates each agent in a separate group.
- Specify the `"decentralized"` learning strategy.
- Run the training for at most 1000 episodes, with each episode lasting at most 600 time steps.
- Stop the training of an agent when its average reward over 30 consecutive episodes is –10 or more.

```
trainOpts = rlMultiAgentTrainingOptions(...
    AgentGroups="auto",...
    LearningStrategy="decentralized",...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=600,...
    ScoreAveragingWindowLength=30,...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-10);
```

Train the agents using the `train` function. Training can take several hours to complete depending on the available computational power. To save time, load the MAT file `decentralizedAgents.mat` which contains a set of pretrained agents. To train the agents yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    decentralizedTrainResults = train([agentA,agentB],env,trainOpts);
else
    load("decentralizedAgents.mat");
end
```

The following figure shows a snapshot of decentralized training progress. You can expect different results due to randomness in the training process.




## 2. Centralized Training

To configure centralized multi-agent training for this example:

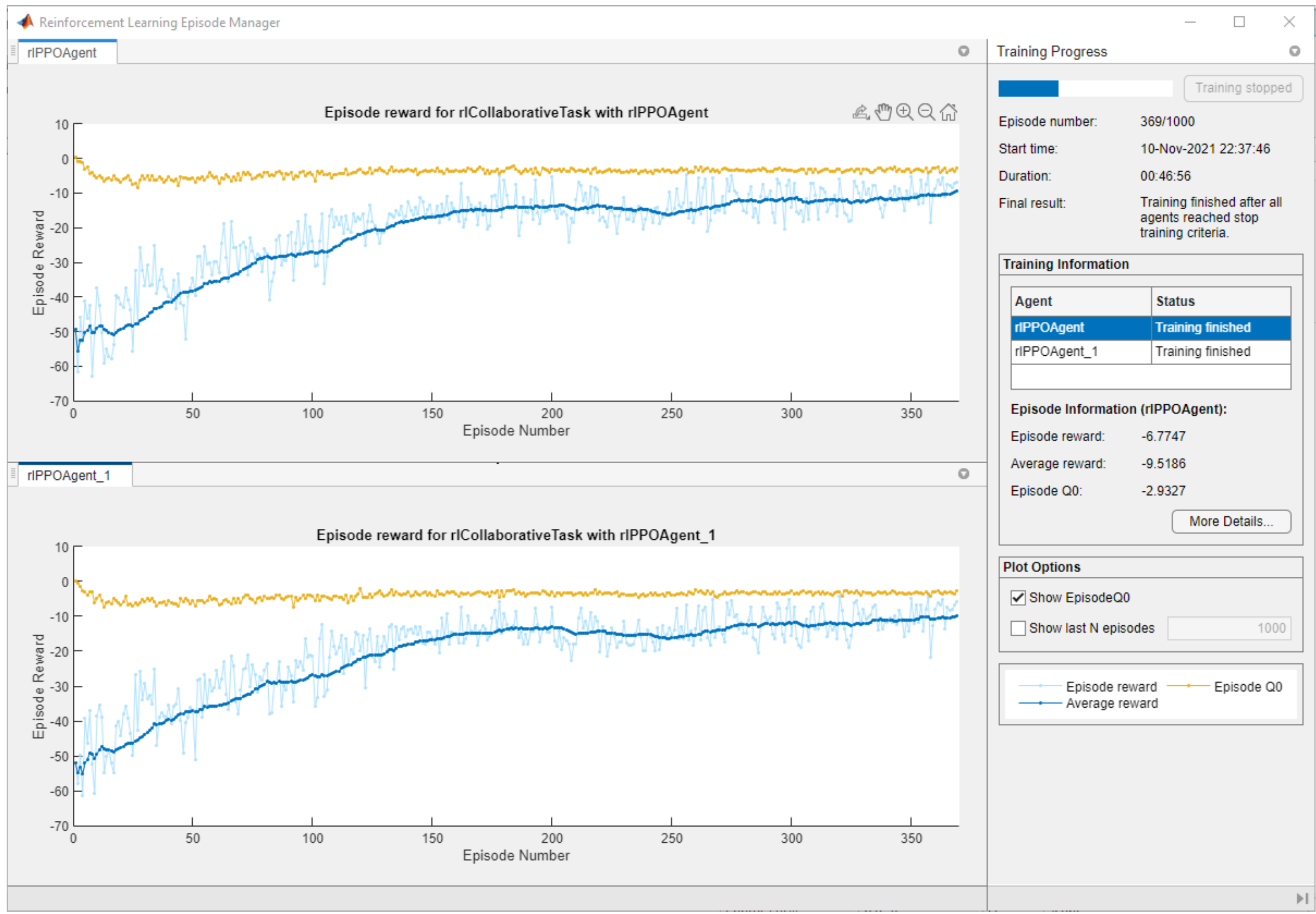
- Allocate both agents (with indices `1` and `2`) in a single group. You can do this by specifying the agent indices in the `"AgentGroups"` option.
- Specify the `"centralized"` learning strategy.
- Run the training for at most 1000 episodes, with each episode lasting at most 600 time steps.
- Stop the training of an agent when its average reward over 30 consecutive episodes is –10 or more.

```
trainOpts = rlMultiAgentTrainingOptions(...
    AgentGroups={[1,2]},...
    LearningStrategy="centralized",...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=600,...
    ScoreAveragingWindowLength=30,...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-10);
```

Train the agents using the `train` function. Training can take several hours to complete depending on the available computational power. To save time, load the MAT file `centralizedAgents.mat` which contains a set of pretrained agents. To train the agents yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    centralizedTrainResults = train([agentA,agentB],env,trainOpts);
else
    load("centralizedAgents.mat");
end
```

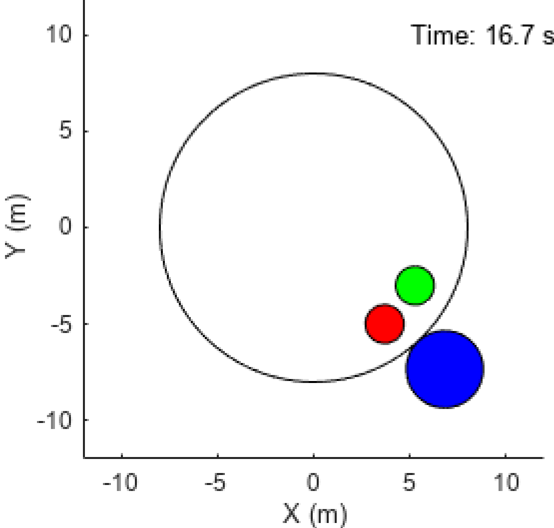
The following figure shows a snapshot of centralized training progress. You can expect different results due to randomness in the training process.




### Simulation

Once the training is finished, simulate the trained agents with the environment.

```
simOptions = rlSimulationOptions(MaxSteps=300);
exp = sim(env,[agentA agentB],simOptions);
```




For more information on agent simulation, see `rlSimulationOptions` and `sim`.

## See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlPPOAgent` | `rlPPOAgentOptions` | `rlTrainingOptions` | `rlSimulationOptions`

**Blocks**
RL Agent

## Related Examples

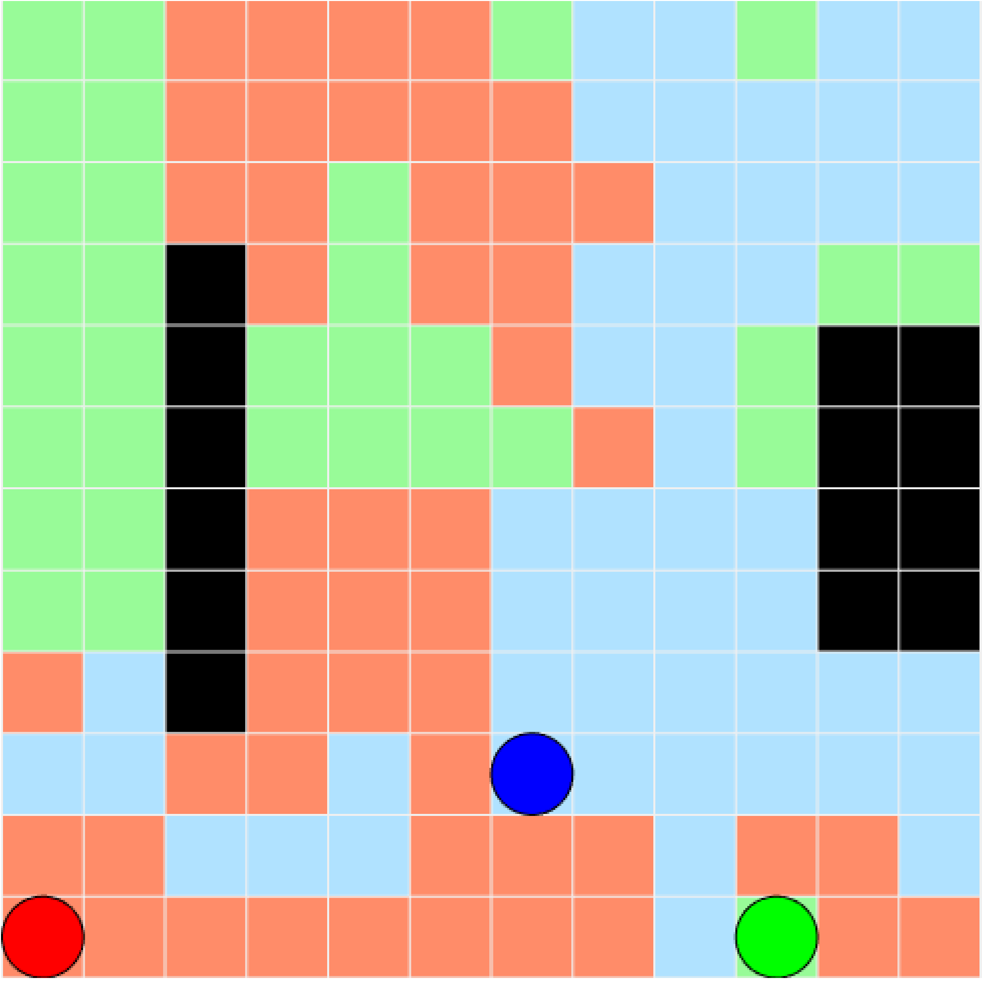
- “Train PPO Agent for a Lander Vehicle” on page 5-180
- “Train Multiple Agents for Area Coverage” on page 5-198
- “Train Multiple Agents for Path Following Control” on page 5-206

## More About

- “Proximal Policy Optimization (PPO) Agents” on page 3-49
- “Train Reinforcement Learning Agents” on page 5-3

# Train Multiple Agents for Area Coverage

This example demonstrates a multi-agent collaborative-competitive task in which you train three proximal policy optimization (PPO) agents to explore all areas within a grid-world environment.



Multi-agent training is supported for Simulink® environments only. As shown in this example, if you define your environment behavior using a MATLAB® System object, you can incorporate it into a Simulink environment using a MATLAB System (Simulink) block.

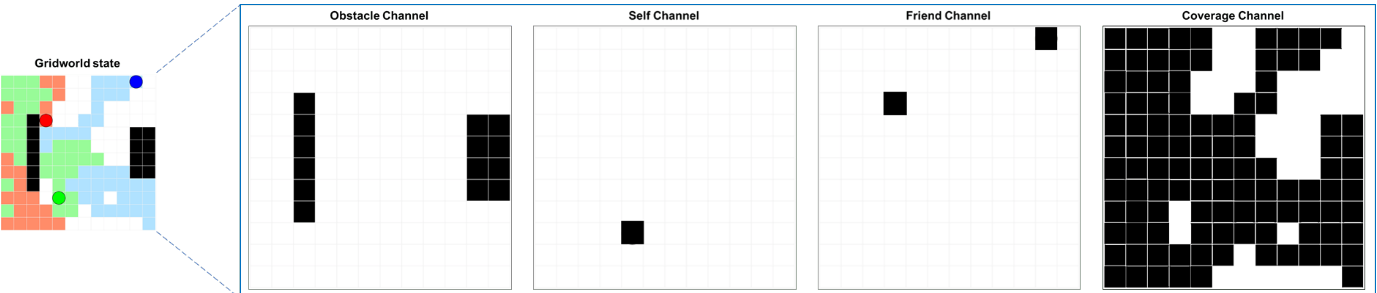
### Create Environment

The environment in this example is a 12x12 grid world containing obstacles, with unexplored cells marked in white and obstacles marked in black. There are three robots in the environment represented by the red, green, and blue circles. Three proximal policy optimization agents with discrete action spaces control the robots. To learn more about PPO agents, see “Proximal Policy Optimization (PPO) Agents” on page 3-49.

The agents provide one of five possible movement actions (WAIT, UP, DOWN, LEFT, or RIGHT) to their respective robots. The robots decide whether an action is legal or illegal. For example, an action of moving LEFT when the robot is located next to the left boundary of the environment is deemed

illegal. Similarly, actions for colliding against obstacles and other agents in the environment are illegal actions and draw penalties. The environment dynamics are deterministic, which means the robots execute legal and illegal actions with 100% and 0% probabilities, respectively. The overall goal is to explore all cells as quickly as possible.

At each time step, an agent observes the state of the environment through a set of four images that identify the cells with obstacles, current position of the robot that is being controlled, position of other robots, and cells that have been explored during the episode. These images are combined to create a 4-channel 12x12 image observation set. The following figure shows an example of what the agent controlling the green robot observes for a given time step.




For the grid world environment:

- The search area is a 12x12 grid with obstacles.
- The observation for each agent is a 12x12x4 image.
- The discrete action set is a set of five actions (WAIT=0, UP=1, DOWN=2, LEFT=3, RIGHT=4).
- The simulation terminates when the grid is fully explored or the maximum number of steps is reached.

At each time step, agents receive the following rewards and penalties.

- +1 for moving to a previously unexplored cell (white).
- -0.5 for an illegal action (attempt to move outside the boundary or collide against other robots and obstacles)
- -0.05 for an action that results in movement (movement cost).
- -0.1 for an action that results in no motion (lazy penalty).
- If the grid is fully explored, +200 times the coverage contribution for that robot during the episode (ratio of cells explored to total cells)

Define the locations of obstacles within the grid using a matrix of indices. The first column contains the row indices, and the second column contains the column indices.

```
obsMat = [4 3; 5 3; 6 3; 7 3; 8 3; 9 3; 5 11;
    6 11; 7 11; 8 11; 5 12; 6 12; 7 12; 8 12];
```

Initialize the robot positions.

```
sA0 = [2 2];
sB0 = [11 4];
sC0 = [3 12];
s0 = [sA0; sB0; sC0];
```

Specify the sample time, simulation time, and maximum number of steps per episode.

```
Ts = 0.1;
Tf = 100;
maxsteps = ceil(Tf/Ts);
```

Open the Simulink model.

```
mdl = "rlAreaCoverage";
open_system(mdl)
```

Copyright 2020 The MathWorks, Inc.

The `GridWorld` block is a MATLAB System block representing the training environment. The System object for this environment is defined in `GridWorld.m`.

In this example, the agents are homogeneous and have the same observation and action specifications. Create the observation and action specifications for the environment. For more information, see `rlNumericSpec` and `rlFiniteSetSpec`.

```
% Define observation specifications.
obsSize = [12 12 4];
oinfo = rlNumericSpec(obsSize);
oinfo.Name = "observations";

% Define action specifications.
numAct = 5;
actionSpace = {0,1,2,3,4};
ainfo = rlFiniteSetSpec(actionSpace);
ainfo.Name = "actions";
```

Specify the block paths for the agents.

```
blks = mdl + ["/Agent A (Red)","/Agent B (Green)","/Agent C (Blue)"];
```

Create the environment interface, specifying the same observation and action specifications for all three agents.

```
env = rlSimulinkEnv(mdl,blks,{oinfo,oinfo,oinfo},{ainfo,ainfo,ainfo});
```

Specify a reset function for the environment. The reset function `resetMap` ensures that the robots start from random initial positions at the beginning of each episode. The random initialization makes the agents robust to different starting positions and improves training convergence.

```
env.ResetFcn = @(in) resetMap(in, obsMat);
```

### Create Agents

The PPO agents in this example operate on a discrete action space and rely on actor and critic functions to learn the optimal policies. The agents maintain deep neural network-based function approximators for the actors and critics with similar network structures (a combination of convolution and fully connected layers). The critic outputs a scalar value representing the state value $V(s)$. The actor outputs the probabilities $\pi(a|s)$ of taking each of the five actions WAIT, UP, DOWN, LEFT, or RIGHT.

Set the random seed for reproducibility.

```
rng(0)
```

Create the actor and critic functions using the following steps.

1. Create the actor and critic deep neural networks.
2. Create the actor function objects using the `rlDiscreteCategoricalActor` command.
3. Create the critic function objects using the `rlValueFunction` command.

Use the same network structure and representation options for all three agents.

```
for idx = 1:3

    % Create actor deep neural network.
    actorNetWork = [
        imageInputLayer(obsSize,Normalization="none")
        convolution2dLayer(8,16, ...
            Stride=1,Padding=1,WeightsInitializer="he")
        reluLayer
        convolution2dLayer(4,8, ...
            Stride=1,Padding="same",WeightsInitializer="he")
        reluLayer
        fullyConnectedLayer(256,WeightsInitializer="he")
        reluLayer
        fullyConnectedLayer(128,WeightsInitializer="he")
        reluLayer
        fullyConnectedLayer(64,WeightsInitializer="he")
        reluLayer
        fullyConnectedLayer(numAct)
        softmaxLayer
        ];
    actorNetWork = dlnetwork(actorNetWork);

    % Create critic deep neural network.
    criticNetwork = [
        imageInputLayer(obsSize,Normalization="none")
        convolution2dLayer(8,16, ...
            Stride=1,Padding=1,WeightsInitializer="he")
        reluLayer
```

```
        convolution2dLayer(4,8, ...
            Stride=1,Padding="same",WeightsInitializer="he")
        reluLayer
        fullyConnectedLayer(256,WeightsInitializer="he")
        reluLayer
        fullyConnectedLayer(128,WeightsInitializer="he")
        reluLayer
        fullyConnectedLayer(64,WeightsInitializer="he")
        reluLayer
        fullyConnectedLayer(1)
        ];
    criticNetwork = dlnetwork(criticNetwork);

    % create actor and critic
    actor(idx) = rlDiscreteCategoricalActor(actorNetWork,oinfo,ainfo);
    critic(idx) = rlValueFunction(criticNetwork,oinfo);
end
```

Specify training options for the critic and the actor using `rlOptimizerOptions`.

```
actorOpts = rlOptimizerOptions(LearnRate=1e-4,GradientThreshold=1);
criticOpts = rlOptimizerOptions(LearnRate=1e-4,GradientThreshold=1);
```

Specify the agent options using `rlPPOAgentOptions`, include the training options for the actor and critic. Use the same options for all three agents. During training, agents collect experiences until they reach the experience horizon of 128 steps and then train from mini-batches of 64 experiences. An objective function clip factor of 0.2 improves training stability, and a discount factor value of 0.995 encourages long-term rewards.

```
opt = rlPPOAgentOptions(...
    ActorOptimizerOptions=actorOpts,...
    CriticOptimizerOptions=criticOpts,...
    ExperienceHorizon=128,...
    ClipFactor=0.2,...
    EntropyLossWeight=0.01,...
    MiniBatchSize=64,...
    NumEpoch=3,...
    AdvantageEstimateMethod="gae",...
    GAEFactor=0.95,...
    SampleTime=Ts,...
    DiscountFactor=0.995);
```

Create the agents using the defined actors, critics, and options.

```
agentA = rlPPOAgent(actor(1),critic(1),opt);
agentB = rlPPOAgent(actor(2),critic(2),opt);
agentC = rlPPOAgent(actor(3),critic(3),opt);
```

Alternatively, you can create the agents first, and then access their option object to modify the options using dot notation.

#### Train Agents

In this example, the agents are trained independently in decentralized manner. Specify the following options for training the agents.

- Automatically assign agent groups using the `AgentGroups=auto` option. This allocates each agent in a separate group.

- Specify the `"decentralized"` learning strategy.
- Run the training for at most 1000 episodes, with each episode lasting at most 5000 time steps.
- Stop the training of an agent when its average reward over 100 consecutive episodes is 80 or more.

```
trainOpts = rlMultiAgentTrainingOptions(...
    "AgentGroups","auto",...
    "LearningStrategy","decentralized",...
    MaxEpisodes=1000,...
    MaxStepsPerEpisode=maxsteps,...
    Plots="training-progress",...
    ScoreAveragingWindowLength=100,...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=80);
```

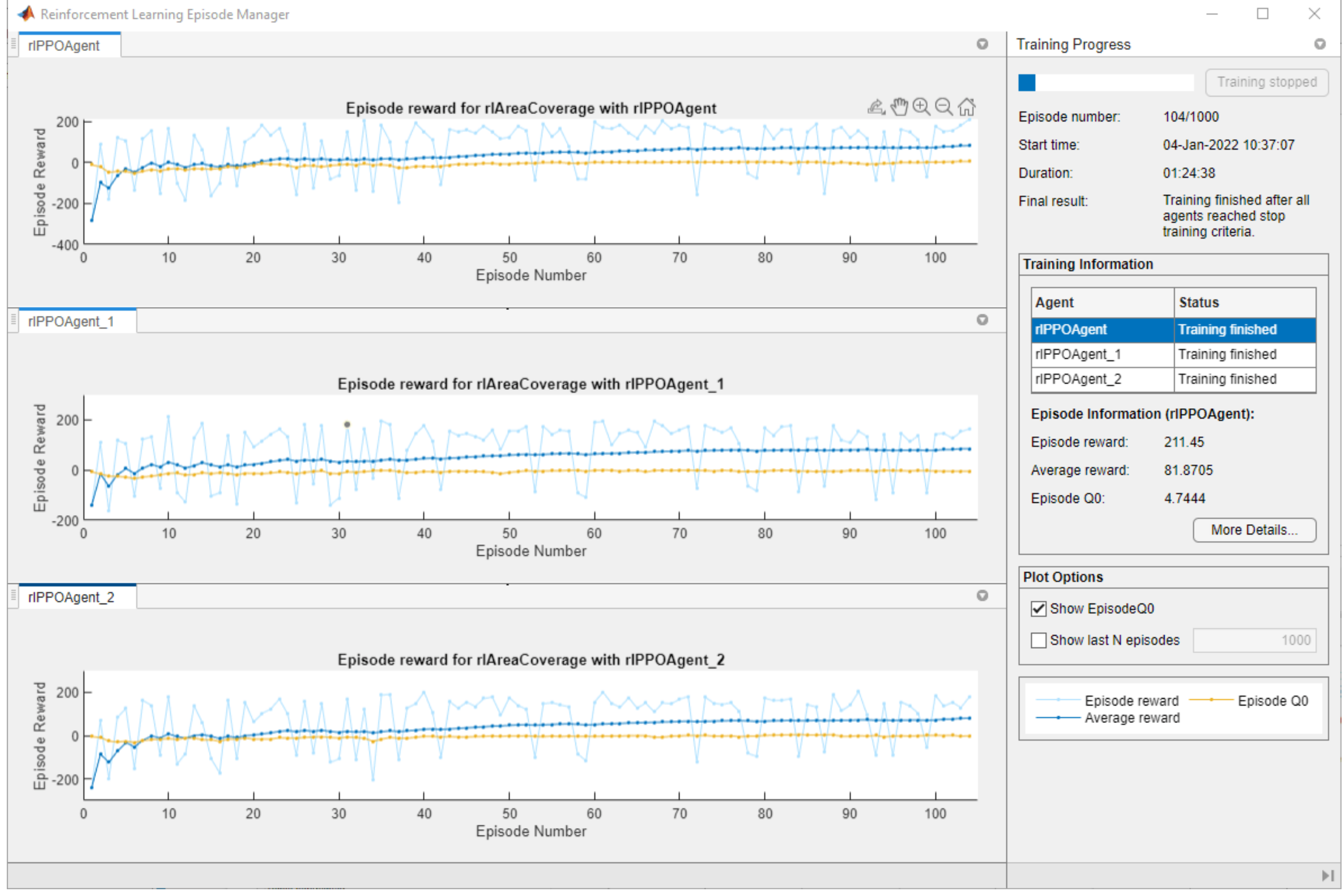
For more information on multi-agent training, type `help rlMultiAgentTrainingOptions` in MATLAB.

To train the agents, specify an array of agents to the `train` function. The order of the agents in the array must match the order of agent block paths specified during environment creation. Doing so ensures that the agent objects are linked to the appropriate action and observation specifications in the environment.

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

```
doTraining = false;
if doTraining
    result = train([agentA,agentB,agentC],env,trainOpts);
else
    load("rlAreaCoverageAgents.mat");
end
```

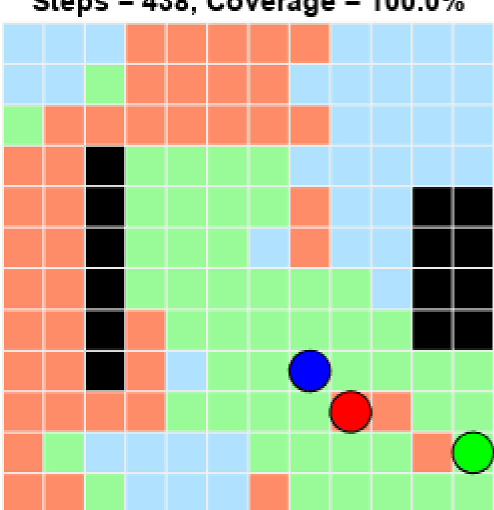
The following figure shows a snapshot of the training progress. You can expect different results due to randomness in the training process.




### Simulate Agents

Simulate the trained agents within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
rng(0) % reset the random seed
simOpts = rlSimulationOptions(MaxSteps=maxsteps);
experience = sim(env,[agentA,agentB,agentC],simOpts);
```




The agents successfully cover the entire grid world.

## See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv` | `rlNumericSpec` | `rlFiniteSetSpec`

**Objects**
`rlPPOAgent` | `rlPPOAgentOptions` | `rlTrainingOptions` | `rlSimulationOptions` | `rlOptimizerOptions`

**Blocks**
RL Agent

## Related Examples

- “Train PPO Agent for a Lander Vehicle” on page 5-180
- “Train Multiple Agents to Perform Collaborative Task” on page 5-190
- “Train Multiple Agents for Path Following Control” on page 5-206

## More About

- “Proximal Policy Optimization (PPO) Agents” on page 3-49
- “Train Reinforcement Learning Agents” on page 5-3

# Train Multiple Agents for Path Following Control

This example shows how to train multiple agents to collaboratively perform path-following control (PFC) for a vehicle. The goal of PFC is to make the ego vehicle travel at a set velocity while maintaining a safe distance from a lead car by controlling longitudinal acceleration and braking, and also while keeping the vehicle travelling along the centerline of its lane by controlling the front steering angle. For more information on PFC, see Path Following Control System (Model Predictive Control Toolbox).

### Overview

An example that trains a reinforcement learning agent to perform PFC is shown in "Train DDPG Agent for Path-Following Control" on page 5-247. In that example, a single deep deterministic policy gradient (DDPG) agent is trained to control both the longitudinal speed and lateral steering of the ego vehicle. In this example, you train two reinforcement learning agents — A DDPG agent provides continuous acceleration values for the longitudinal control loop and a deep Q-network (DQN) agent provides discrete steering angle values for the lateral control loop.

The trained agents perform PFC through cooperative behavior and achieve satisfactory results.

### Create Environment

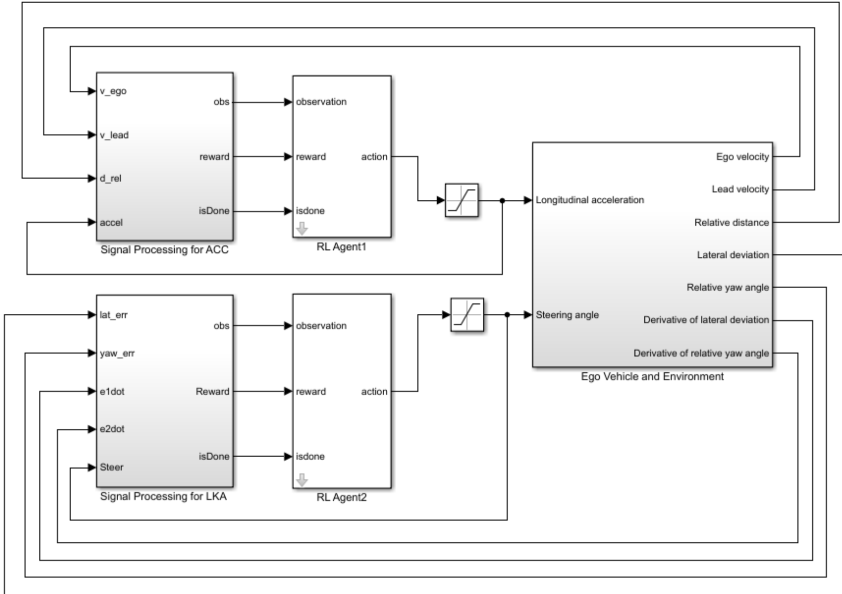
The environment for this example includes a simple bicycle model for the ego car and a simple longitudinal model for the lead car. The training goal is to make the ego car travel at a set velocity while maintaining a safe distance from lead car by controlling longitudinal acceleration and braking, while also keeping the ego car travelling along the centerline of its lane by controlling the front steering angle.

Load the environment parameters.

```
multiAgentPFCParams
```

Open the Simulink model.

```
mdl = "rlMultiAgentPFC";
open_system(mdl)
```




In this model, the two reinforcement learning agents (RL Agent1 and RL Agent2) provide longitudinal acceleration and steering angle signals, respectively.

The simulation terminates when any of the following conditions occur.

- $|e_1| > 1$ (magnitude of the lateral deviation exceeds 1)
- $V_{ego} < 0.5$ (longitudinal velocity of the ego car drops below 0.5.
- $D_{rel} < 0$ (distance between the ego and lead car is below zero)

For the longitudinal controller (RL Agent1):

- The reference velocity for the ego car $V_{ref}$ is defined as follows. If the relative distance is less than the safe distance, the ego car tracks the minimum of the lead car velocity and driver-set velocity. In this manner, the ego car maintains some distance from the lead car. If the relative distance is greater than the safe distance, the ego car tracks the driver-set velocity. In this example, the safe distance is defined as a linear function of the ego car longitudinal velocity $V$, that is, $t_{gap}*V + D_{default}$. The safe distance determines the tracking velocity for the ego car.
- The observations from the environment contain the longitudinal measurements: the velocity error $e_V = V_{ref} - V$, its integral $\int e$, and the ego car longitudinal velocity $V$.
- The action signal consists of continuous acceleration values between -3 and 2 m/s^2.
- The reward $r_t$, provided at every time step $t$, is

$$r_t = -(10e_V^2 + 100a_{t-1}^2) \times 0.001 - 10F_t + M_t$$

Here, $a_{t-1}$ is the acceleration input from the previous time step, and:

- $F_t = 1$ if the simulation is terminated, otherwise $F_t = 0$.
- $M_t = 1$ if $e_V^2 < 1$, otherwise $M_t = 0$.

For the lateral controller (RL Agent2):

- The observations from the environment contain the lateral measurements: the lateral deviation $e_1$, relative yaw angle $e_2$, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.
- The action signal consists of discrete steering angle actions which take values from -15 degrees (-0.2618 rad) to 15 degrees (0.2618 rad) in steps of 1 degree (0.0175 rad).
- The reward $r_t$, provided at every time step $t$, is

$$r_t = -(100e_1^2 + 500u_{t-1}^2) \times 0.001 - 10F_t + 2H_t$$

Here, $u_{t-1}$ is the steering input from the previous time step, $a_{t-1}$ is the acceleration input from the previous time step, and:

- $F_t = 1$if the simulation is terminated, otherwise $F_t = 0$.
- $H_t = 1$ $e_1^2 < 0.01$, otherwise $H_t = 0$.

The logical terms in the reward functions ($F_t$, $M_t$, and $H_t$) penalize the agents if the simulation terminates early, while encouraging the agents to make both the lateral error and velocity error small.

Create the observation and action specifications for longitudinal control loop.

```
obsInfo1 = rlNumericSpec([3 1]);
actInfo1 = rlNumericSpec([1 1],LowerLimit=-3,UpperLimit=2);
```

Create the observation and action specifications for lateral control loop.

```
obsInfo2 = rlNumericSpec([6 1]);
actInfo2 = rlFiniteSetSpec((-15:15)*pi/180);
```

Combine the observation and action specifications as a cell array.

```
obsInfo = {obsInfo1,obsInfo2};
actInfo = {actInfo1,actInfo2};
```

Create a Simulink environment interface, specifying the block paths for both agent blocks. The order of the block paths must match the order of the observation and action specification cell arrays.

```
blks = mdl + ["/RL Agent1", "/RL Agent2"];
env = rlSimulinkEnv(mdl,blks,obsInfo,actInfo);
```

Specify a reset function for the environment using the `ResetFcn` property. The function `pfcResetFcn` randomly sets the initial poses of the lead and ego vehicles at the beginning of every episode during training.

```
env.ResetFcn = @pfcResetFcn;
```

### Create Agents

For this example you create two reinforcement learning agents. First, fix the random seed for reproducibility.

```
rng(0)
```

Both agents operate at the same sample time in this example. Set the sample time value (in seconds).

```
Ts = 0.1;
```

### Longitudinal Control

The agent for the longitudinal control loop is a DDPG agent. A DDPG agent approximates the long-term reward given observations and actions using a critic value function representation and selects actions using an actor policy representation. For more information on creating deep neural network value function and policy representations, see "Create Policies and Value Functions" on page 4-2.

Use the `createCCAgent` function to create a DDPG agent for longitudinal control. The structure of this agent is similar to the "Train DDPG Agent for Adaptive Cruise Control" on page 5-215 example.

```
agent1 = createACCAgent(obsInfo1,actInfo1,Ts);
```

### Lateral Control

The agent for the lateral control loop is a DQN agent. A DQN agent approximates the long-term reward given observations and actions using a critic value function representation.

Use the `createLKAAgent` function to create a DQN agent for lateral control. The structure of this agent is similar to the "Train DQN Agent for Lane Keeping Assist" on page 5-226 example.

```
agent2 = createLKAAgent(obsInfo2,actInfo2,Ts);
```

### Train Agents

Specify the training options. For this example, use the following options.

- Run each training episode for at most 5000 episodes, with each episode lasting at most `maxsteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Verbose` and `Plots` options).
- Stop training the DDPG and DQN agents when they receive an average reward greater than 480 and 1195, respectively. When one agent reaches its stop criteria, it simulates its own policy without learning while the other agent continues training.

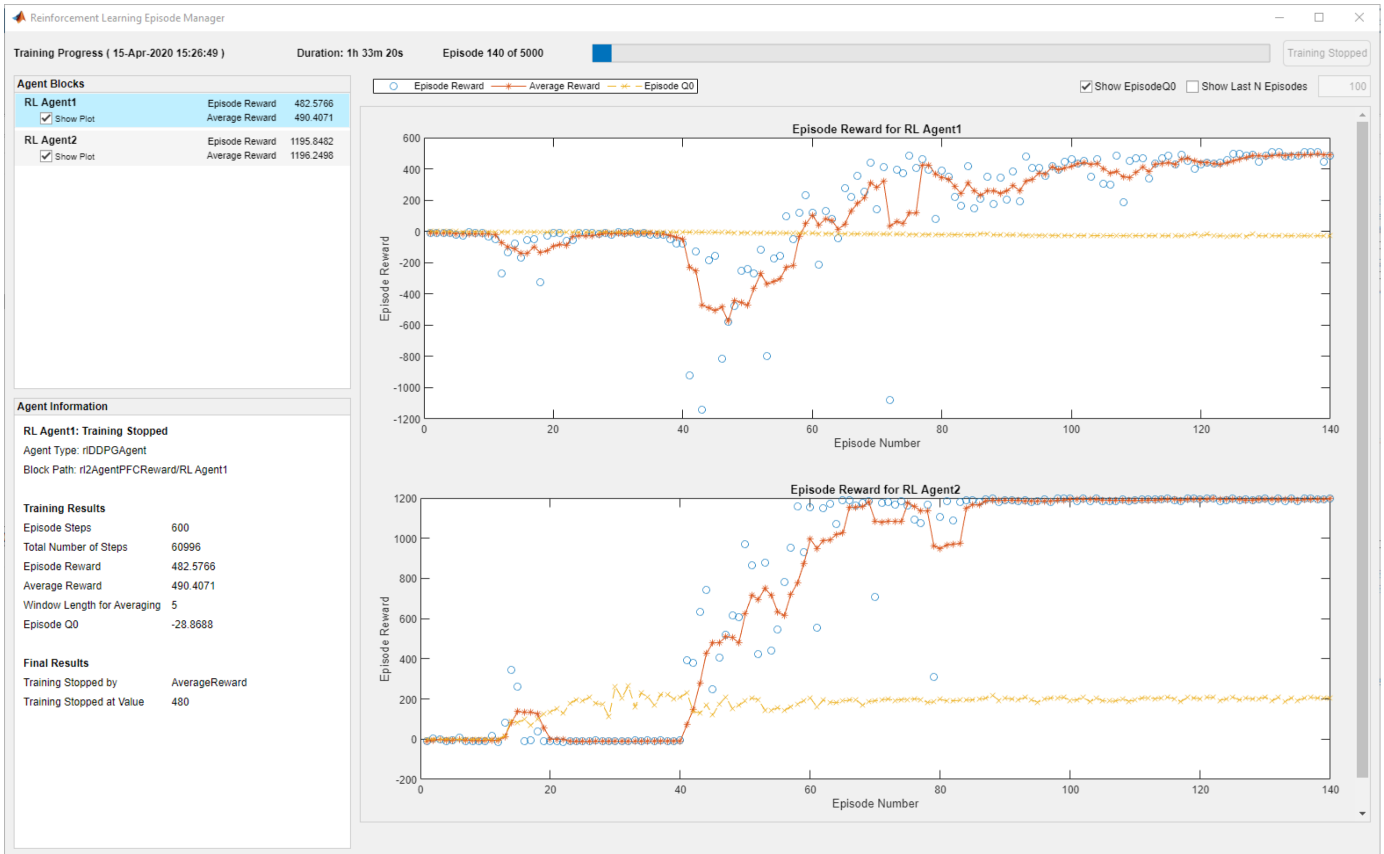
```
Tf = 60; % simulation time
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=[480,1195]);
```

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

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train([agent1,agent2],env,trainingOpts);
```

```
else
    % Load pretrained agents for the example.
    load("rlPFCAgents.mat")
end
```

The following figure shows a snapshot of the training progress for the two agents.




## Simulate Agents

To validate the performance of the trained agents, simulate the agents within the Simulink environment by uncommenting the following commands. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

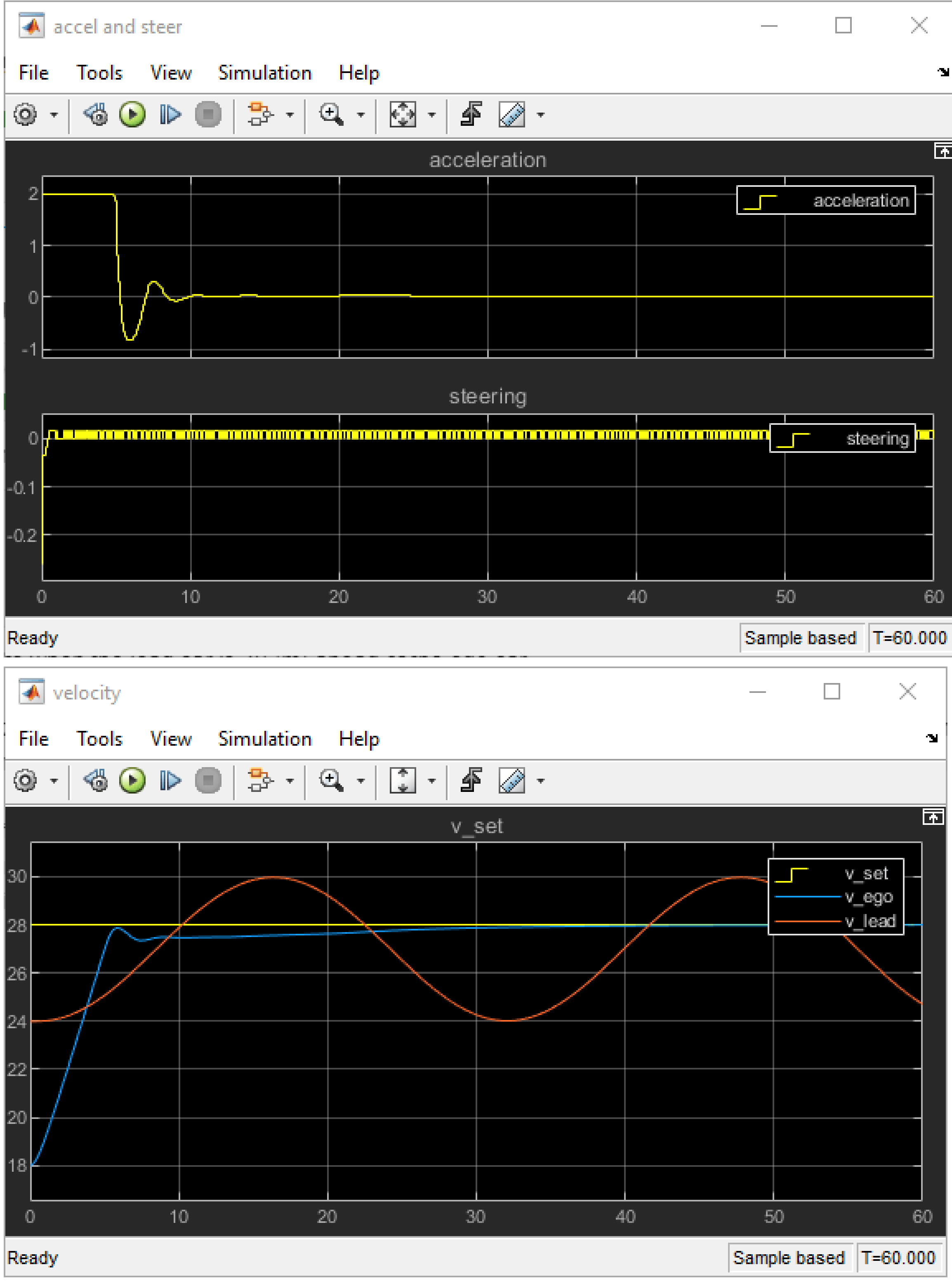
```
% simOptions = rlSimulationOptions(MaxSteps=maxsteps);
% experience = sim(env,[agent1, agent2],simOptions);
```

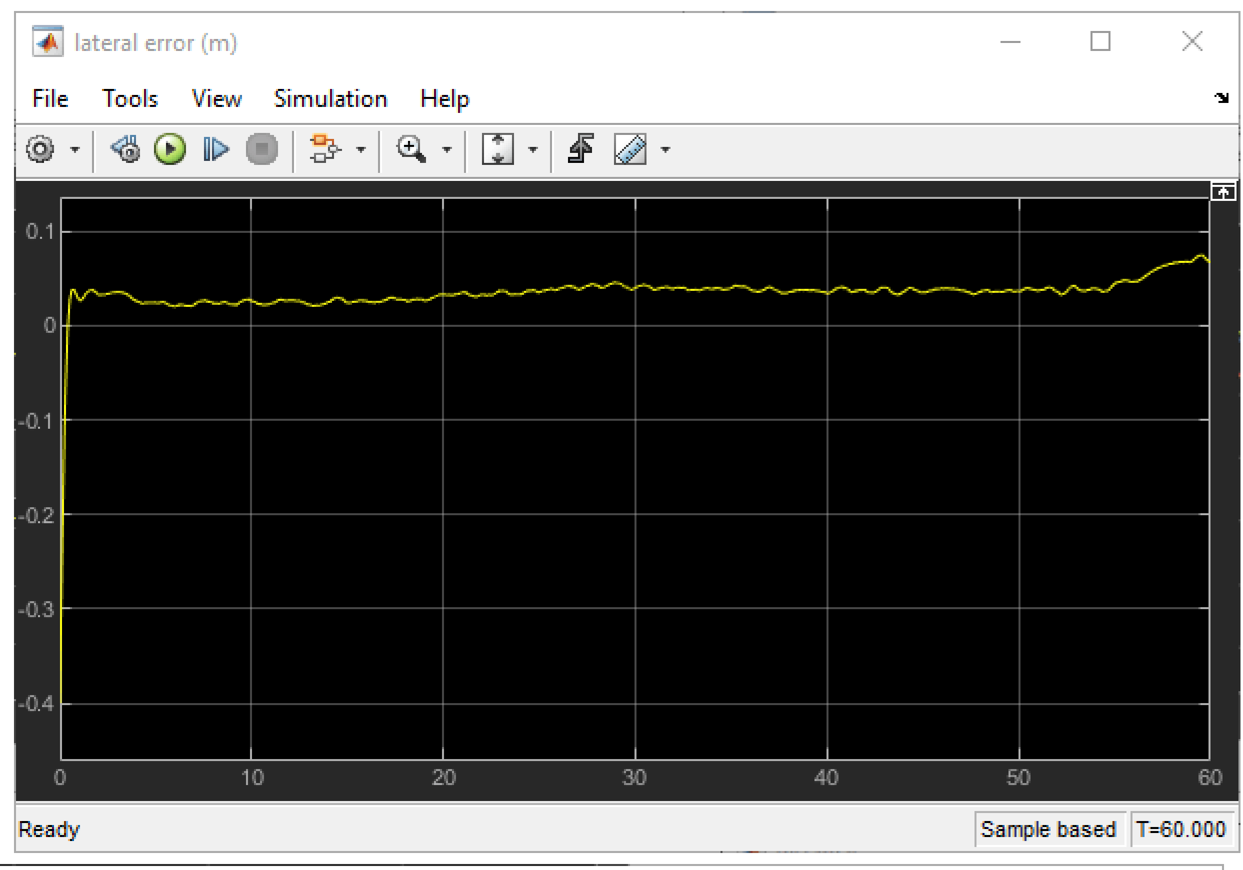
To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

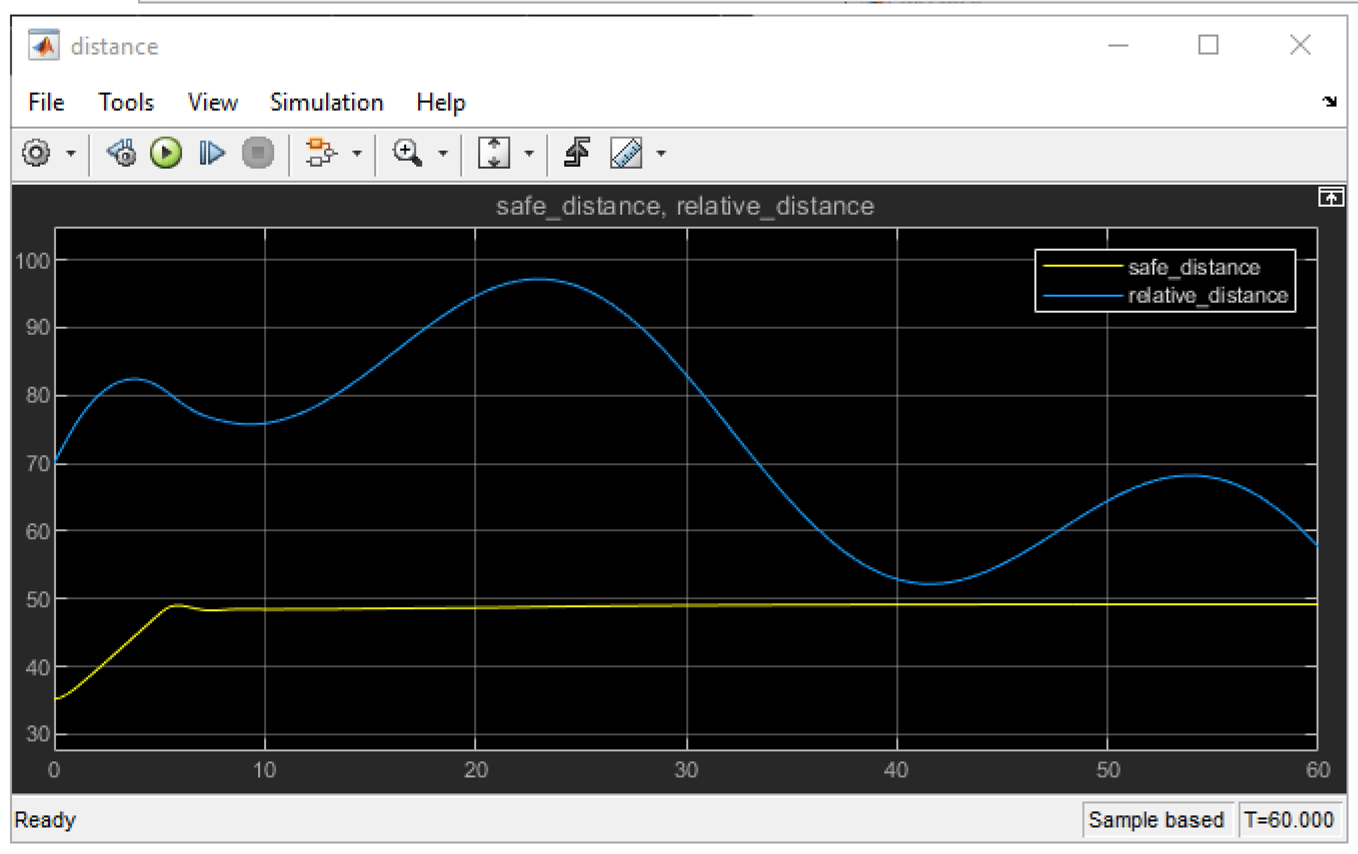
```
e1_initial = -0.4;
e2_initial = 0.1;
x0_lead = 80;
sim(mdl)
```

The following plots show the results when the lead car is 70 m ahead of the ego car at the beginning of simulation.

- The lead car changes speed from 24 m/s to 30 m/s periodically (top-right plot). The ego car maintains a safe distance throughout the simulation (bottom-right plot).
- From 0 to 30 seconds, the ego car tracks the set velocity (top-right plot) and experiences some acceleration (top-left plot). After that, the acceleration is reduced to 0.
- The bottom-left plot shows the lateral deviation. As shown in the plot, the lateral deviation is greatly decreased within 1 second. The lateral deviation remains less than 0.1 m.


accel and steer
File Tools View Simulation Help
acceleration
acceleration
steering
steering
Ready
Sample based
T=60.000
velocity
File Tools View Simulation Help
v_set
v_set
v_ego
v_lead
Ready
Sample based
T=60.000




## See Also



**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlDQNAgent` | `rlDQNAgentOptions` | `rlDDPGAgent` | `rlDDPGAgentOptions` | `rlTrainingOptions` | `rlSimulationOptions`

**Blocks**
RL Agent

## Related Examples

- “Train Multiple Agents for Area Coverage” on page 5-198
- “Train Multiple Agents to Perform Collaborative Task” on page 5-190
- “Train DDPG Agent for Adaptive Cruise Control” on page 5-215
- “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox)

## More About

- “Create Policies and Value Functions” on page 4-2
- “Deep Q-Network (DQN) Agents” on page 3-23
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Train Reinforcement Learning Agents” on page 5-3

# Train DDPG Agent for Adaptive Cruise Control

This example shows how to train a deep deterministic policy gradient (DDPG) agent for adaptive cruise control (ACC) in Simulink®. For more information on DDPG agents, see "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40.

### Simulink Model

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

Specify the initial position and velocity for the two vehicles.

```
x0_lead = 50;   % initial position for lead car (m)
v0_lead = 25;   % initial velocity for lead car (m/s)
x0_ego = 10;    % initial position for ego car (m)
v0_ego = 20;    % initial velocity for ego car (m/s)
```

Specify standstill default spacing (m), time gap (s) and driver-set velocity (m/s).

```
D_default = 10;
t_gap = 1.4;
v_set = 30;
```

To simulate the physical limitations of the vehicle dynamics, constraint the acceleration to the range `[–3,2]` m/s^2.

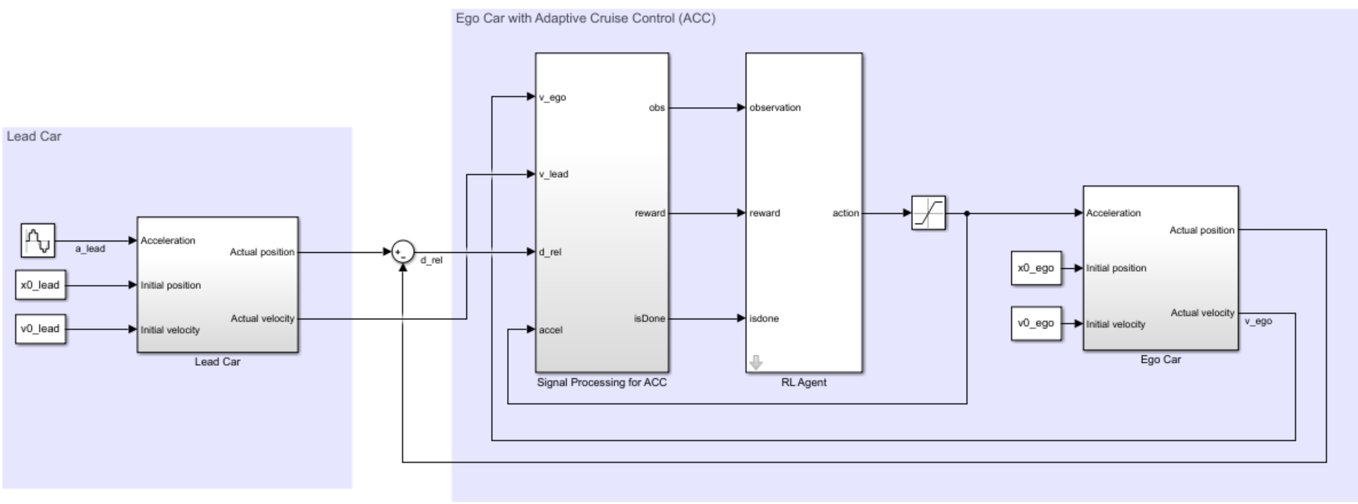
```
amin_ego = -3;
amax_ego = 2;
```

Define the sample time `Ts` and simulation duration `Tf` in seconds.

```
Ts = 0.1;
Tf = 60;
```

Open the model.

```
mdl = "rlACCMdl";
open_system(mdl)
agentblk = mdl + "/RL Agent";
```




Copyright 2018 The MathWorks, Inc.

For this model:

- The acceleration action signal from the agent to the environment is from –3 to 2 m/s^2.
- The reference velocity for the ego car $V_{ref}$ is defined as follows. If the relative distance is less than the safe distance, the ego car tracks the minimum of the lead car velocity and driver-set velocity. In this manner, the ego car maintains some distance from the lead car. If the relative distance is greater than the safe distance, the ego car tracks the driver-set velocity. In this example, the safe distance is defined as a linear function of the ego car longitudinal velocity $V$; that is, $t_{gap} * V + D_{default}$. The safe distance determines the reference tracking velocity for the ego car.
- The observations from the environment are the velocity error $e = V_{ref} - V_{ego}$, its integral $\int e$, and the ego car longitudinal velocity $V$.
- The simulation is terminated when longitudinal velocity of the ego car is less than 0, or the relative distance between the lead car and ego car becomes less than 0.
- The reward $r_t$, provided at every time step $t$, is

$$r_t = -(0.1e_t^2 + u_{t-1}^2) + M_t$$

where $u_{t-1}$ is the control input from the previous time step. The logical value $M_t = 1$ if velocity error $e_t^2 <= 0.25$; otherwise, $M_t = 0$.

#### Create Environment Interface

Create a reinforcement learning environment interface for the model.

Create the observation specification.

```
obsInfo = rlNumericSpec([3 1], ...
    LowerLimit=-inf*ones(3,1), ...
    UpperLimit=inf*ones(3,1));
obsInfo.Name = "observations";
obsInfo.Description = ...
    "velocity error and ego velocity";
```

Create the action specification.

```
actInfo = rlNumericSpec([1 1], ...
    LowerLimit=-3,UpperLimit=2);
actInfo.Name = "acceleration";
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
```

To define the initial condition for the position of the lead car, specify an environment reset function using an anonymous function handle. The reset function `localResetFcn`, which is defined at the end of the example, randomizes the initial position of the lead car.

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

Fix the random generator seed for reproducibility.

```
rng("default")
```

#### Create DDPG agent

DDPG agents use a parametrized Q-value function critic to estimate the value of the policy. A Q-value function takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward given the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value).

Define each network path as an array of layer objects. Use `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` to return the number of dimensions of the observation and action spaces.

Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

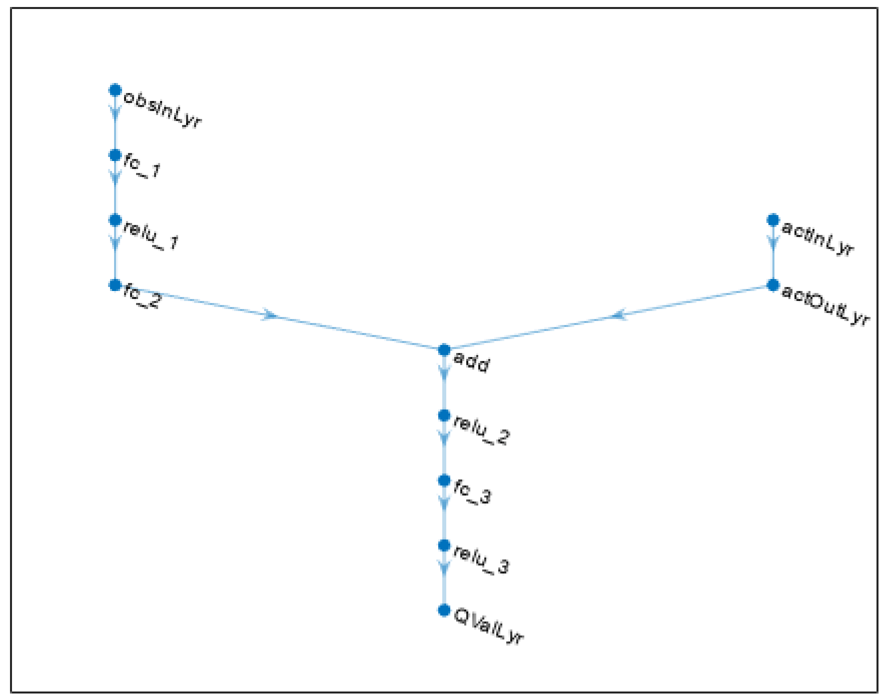
```
L = 48; % number of neurons

% Main path
mainPath = [
    featureInputLayer( ...
        prod(obsInfo.Dimension), ...
        Name="obsInLyr")
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(L)
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(1,Name="QValLyr")
    ];

% Action path
```

```
actionPath = [
    featureInputLayer( ...
        prod(actInfo.Dimension), ...
        Name="actInLyr")
    fullyConnectedLayer(L,Name="actOutLyr")
    ];

% Assemble layers into a layergraph object
criticNet = layerGraph(mainPath);
criticNet = addLayers(criticNet, actionPath);

% Connect layers
criticNet = connectLayers(criticNet,"actOutLyr","add/in2");

% Convert to dlnetwork and display number of weights
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 5k

   Inputs:
      1   'obsInLyr'   3 features
      2   'actInLyr'   1 features
```

View the critic network configuration.

```
plot(criticNet)
```




Create the critic approximator object using `criticNet`, the environment observation and action specifications, and the names of the network input layers to be connected with the environment observation and action channels. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(criticNet,obsInfo,actInfo,...
    ObservationInputNames="obsInLyr",ActionInputNames="actInLyr");
```

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects. Use a `tanhLayer` followed by a `scalingLayer` to scale the network output to the action range.

```
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(prod(actInfo.Dimension))
```

```
    tanhLayer
    scalingLayer(Scale=2.5,Bias=-0.5)
    ];
```

Convert to `dlnetwork` and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 4.9k

   Inputs:
      1   'input'   3 features
```

Create the actor using `actorNet` and the observation and action specifications. For more information on continuous deterministic actors, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNet, ...
    obsInfo,actInfo);
```

Specify training options for the critic and the actor using `rlOptimizerOptions`.

```
criticOptions = rlOptimizerOptions( ...
    LearnRate=1e-3, ...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-4);
actorOptions = rlOptimizerOptions( ...
    LearnRate=1e-4, ...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-4);
```

Specify the DDPG agent options using `rlDDPGAgentOptions`, include the training options for the actor and critic.

```
agentOptions = rlDDPGAgentOptions(...
    SampleTime=Ts,...
    ActorOptimizerOptions=actorOptions,...
    CriticOptimizerOptions=criticOptions,...
    ExperienceBufferLength=1e6);
```

You can also set or modify the agent options using dot notation.

```
agentOptions.NoiseOptions.Variance = 0.6;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Alternatively, you can create the agent first, and then access its option object and modify the options using dot notation.

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

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

### Train Agent

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

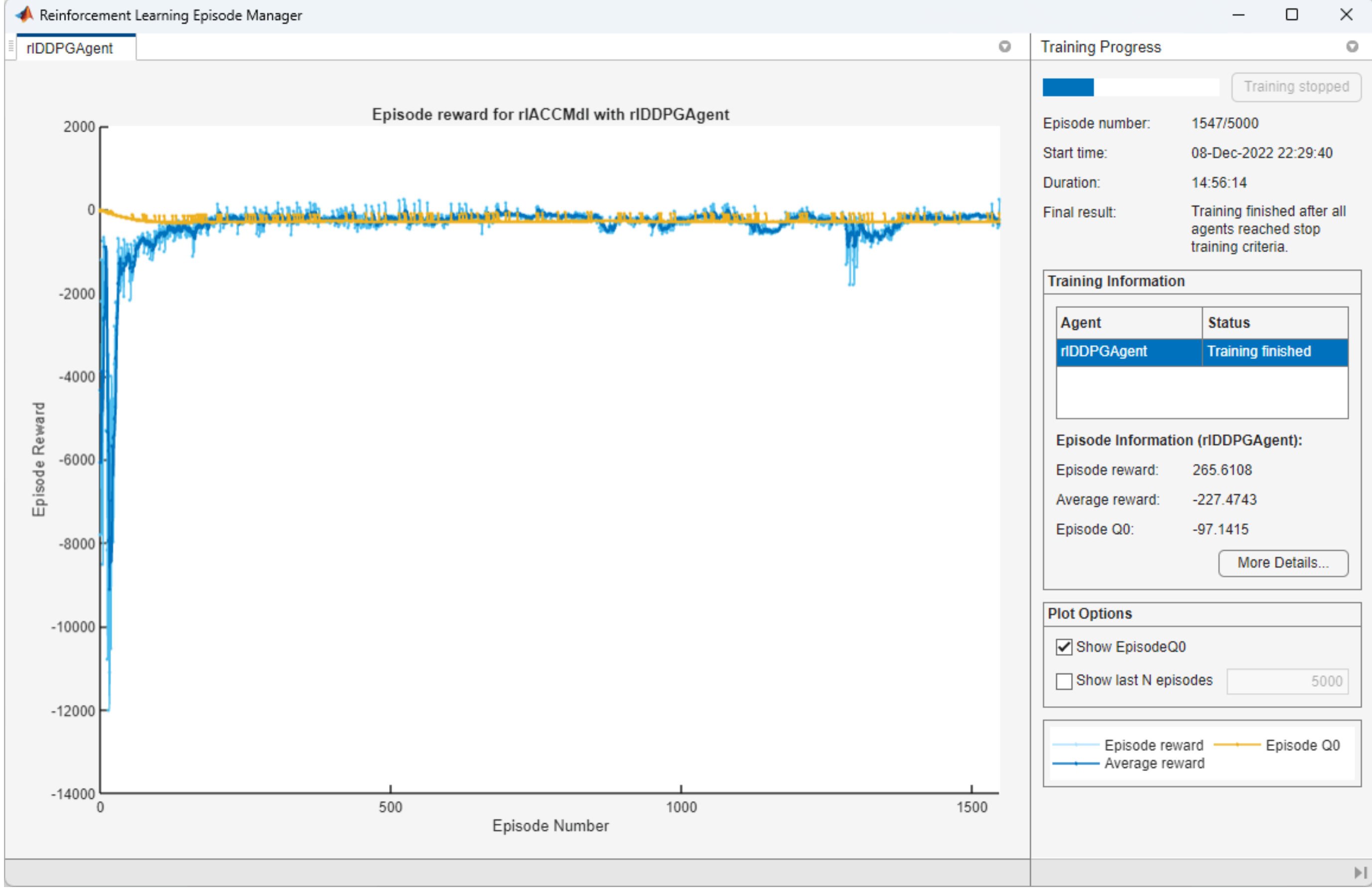
- Run each training episode for at most `5000` episodes, with each episode lasting at most 600 time steps.
- Display the training progress in the Episode Manager dialog box.
- Stop training when the agent receives an episode reward greater than 260.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="EpisodeReward",...
    StopTrainingValue=260);
```

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

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load a pretrained agent for the example.
    load("SimulinkACCDDPG.mat","agent")
end
```




### Simulate DDPG Agent

To validate the performance of the trained agent, you can simulate the agent within the Simulink environment using the following commands. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=maxsteps);
experience = sim(env,agent,simOptions);
```

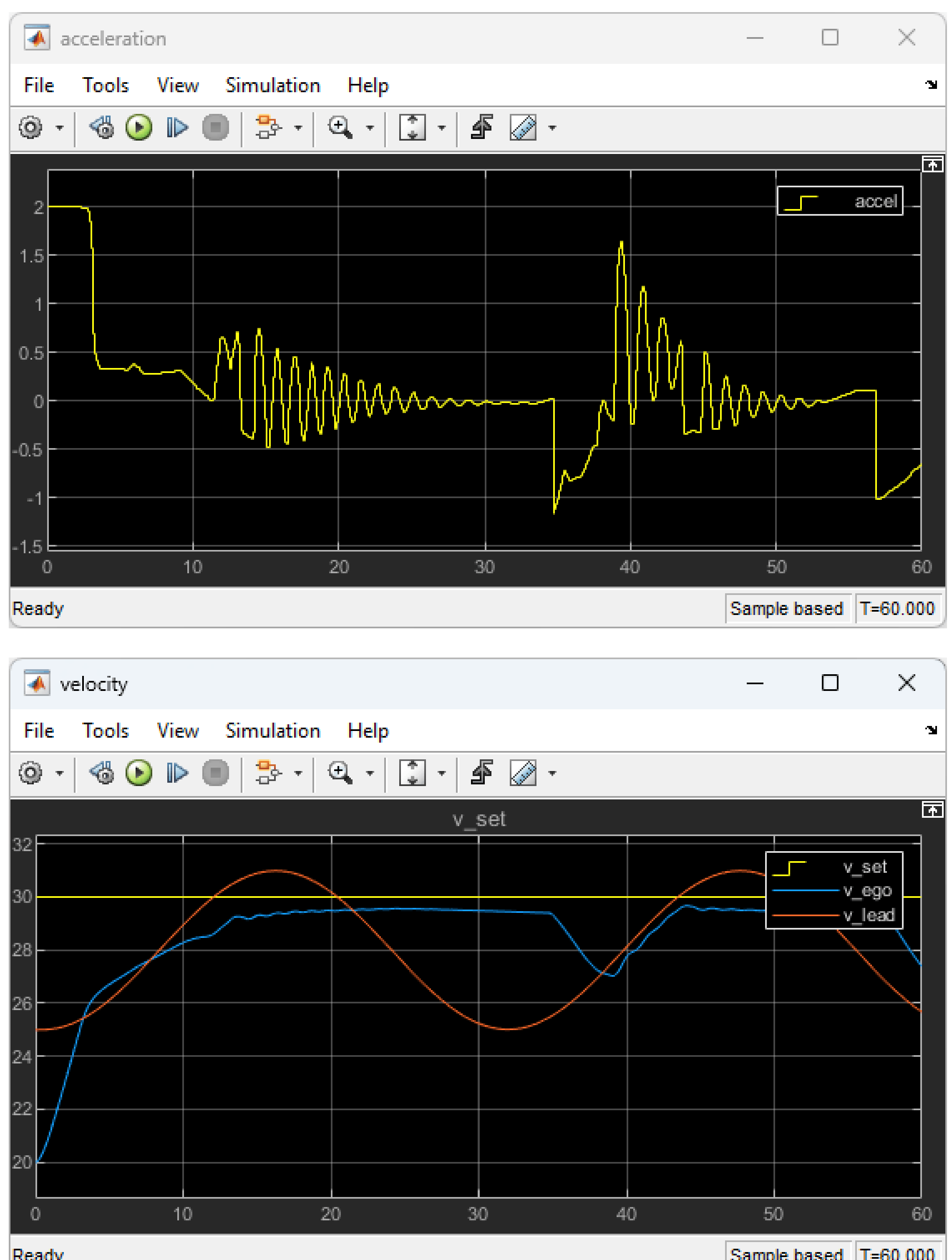
To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

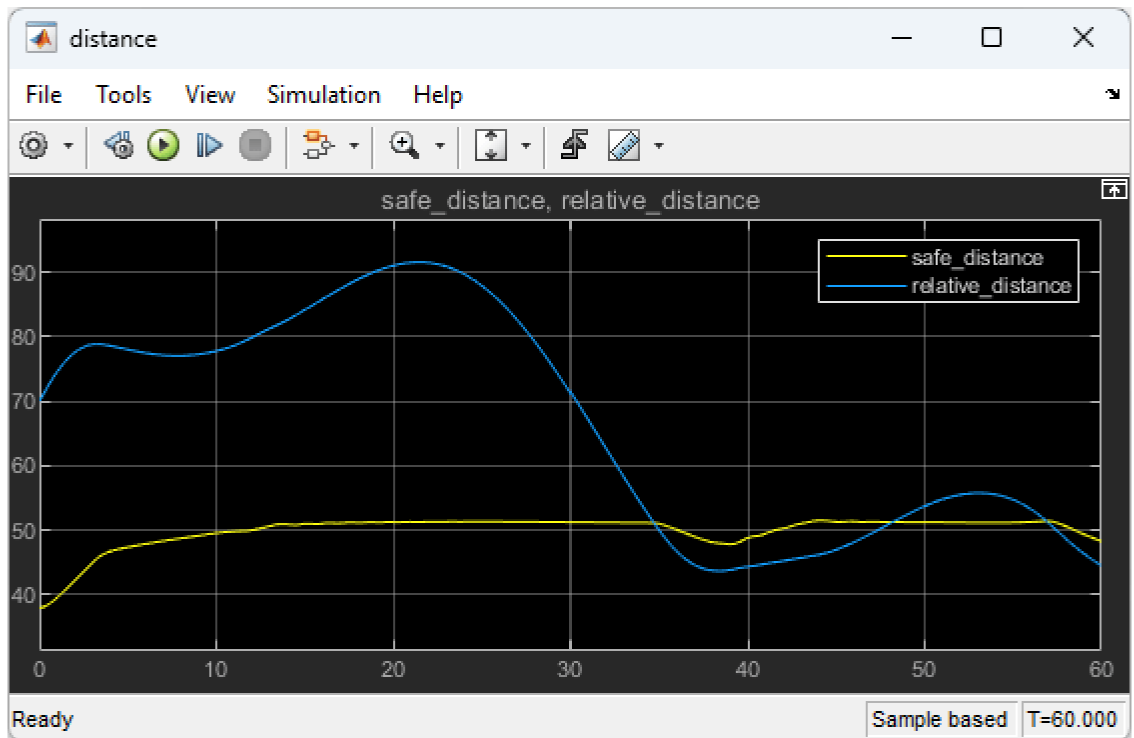
```
x0_lead = 80;
sim(mdl)
```

The following plots show the simulation results when lead car is 70 (m) ahead of the ego car.

- In the first 35 seconds, the relative distance is greater than the safe distance (bottom plot), so the ego car tracks set velocity (middle plot). To speed up and reach the set velocity, acceleration is initially positive (top plot).
- From 35 to 48 seconds, the relative distance is less than the safe distance (bottom plot), so the ego car tracks the minimum of the lead velocity and set velocity. To slow down and track the lead car velocity, the acceleration turns negative from 35 to approximately 37 seconds (top plot). Afterwards, the ego car adjusts its acceleration to keep on tracking either the minimum between

the lead velocity and the set velocity, or the set velocity, depending on whether the relative distance is deemed to be safe or not.






Close the Simulink model.

```
bdclose(mdl)
```

**Reset Function**

```
function in = localResetFcn(in)
% Reset the initial position of the lead car.
in = setVariable(in,"x0_lead",40+randi(60,1,1));
end
```

## See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlQValueFunction` | `rlContinuousDeterministicActor` | `rlTrainingOptions` | `rlSimulationOptions` | `rlOptimizerOptions`

**Blocks**
RL Agent

## Related Examples

- “Train Multiple Agents for Path Following Control” on page 5-206
- “Train DDPG Agent for Path-Following Control” on page 5-247

- “Train DQN Agent for Lane Keeping Assist” on page 5-226
- “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox)

## More About

- “Create Policies and Value Functions” on page 4-2
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Train Reinforcement Learning Agents” on page 5-3

# Train DQN Agent for Lane Keeping Assist

This example shows how to train a deep Q-learning network (DQN) agent for lane keeping assist (LKA) in Simulink®. For more information on DQN agents, see "Deep Q-Network (DQN) Agents" on page 3-23.

### Simulink Model for Ego Car

The reinforcement learning environment for this example is a simple bicycle model for the ego vehicle dynamics. The training goal is to keep the ego vehicle traveling along the centerline of the lanes by adjusting the front steering angle. This example uses the same vehicle model as in "Lane Keeping Assist System Using Model Predictive Control" (Model Predictive Control Toolbox). The ego car dynamics is specified by the following parameters.

```
m = 1575;   % total vehicle mass (kg)
Iz = 2875;  % yaw moment of inertia (mNs^2)
lf = 1.2;   % longitudinal distance from center of gravity to front tires (m)
lr = 1.6;   % longitudinal distance from center of gravity to rear tires (m)
Cf = 19000; % cornering stiffness of front tires (N/rad)
Cr = 33000; % cornering stiffness of rear tires (N/rad)
Vx = 15;    % longitudinal velocity (m/s)
```

Define the sample time `Ts` and simulation duration `Tf` in seconds.

```
Ts = 0.1;
T = 15;
```

The output of the LKA system is the front steering angle of the ego car. To simulate the physical limitations of the ego car, constrain the steering angle to the range `[-0.5,0.5]` rad.

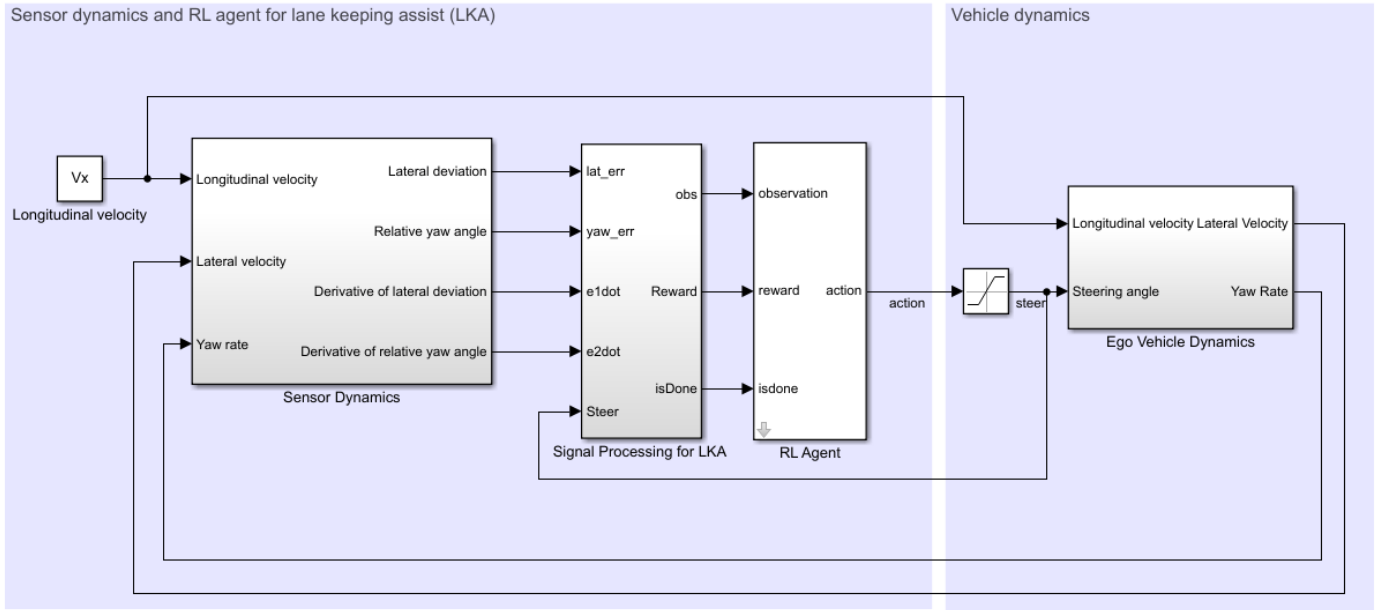
```
u_min = -0.5;
u_max = 0.5;
```

The curvature of the road is defined by a constant 0.001 ($m^{-1}$). The initial value for the lateral deviation is 0.2 m and the initial value for the relative yaw angle is –0.1 rad.

```
rho = 0.001;
e1_initial = 0.2;
e2_initial = -0.1;
```

Open the model.

```
mdl = "rlLKAMdl";
open_system(mdl);
agentblk = mdl + "/RL Agent";
```




Copyright 2018 The MathWorks, Inc.

For this model:

- The steering-angle action signal from the agent to the environment is from –15 degrees to 15 degrees.
- The observations from the environment are the lateral deviation $e_1$, the relative yaw angle $e_2$, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.
- The simulation is terminated when the lateral deviation $|e_1| > 1$.
- The reward $r_t$, provided at every time step $t$, is

$$r_t = -(10e_1^2 + 5e_2^2 + 2u^2 + 5\dot{e}_1^2 + 5\dot{e}_2^2)$$

where $u$ is the control input from the previous time step $t-1$.

**Create Environment Interface**

Create a reinforcement learning environment interface for the ego vehicle. To do so, first create the observation and action specifications.

```
observationInfo = rlNumericSpec([6 1], ...
    LowerLimit=-inf*ones(6,1), ...
    UpperLimit=inf*ones(6,1));
observationInfo.Name = "observations";
observationInfo.Description = "Lateral deviation and relative yaw angle";
actionInfo = rlFiniteSetSpec((-15:15)*pi/180);
actionInfo.Name = "steering";
```

Then, create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

The interface has a discrete action space where the agent can apply one of 31 possible steering angles from –15 degrees to 15 degrees. The observation is the six-dimensional vector containing lateral deviation, relative yaw angle, as well as their derivatives and integrals with respect to time.

To define the initial condition for lateral deviation and relative yaw angle, specify an environment reset function using an anonymous function handle. This reset function randomizes the initial values for the lateral deviation and relative yaw angle.

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

**Create DQN agent**

DQN agents use a parametrized Q-value function approximator to estimate the value of the policy.

Since DQN agents have a discrete action space, you have the option to create a vector (that is multi-output) Q-value function critic, which is generally more efficient than a comparable single-output critic

A vector Q-value function takes only the observation as input and returns as output a single vector with as many elements as the number of possible actions. The value of each output element represents the expected discounted cumulative long-term reward when an agent starts from the state corresponding to the given observation and executes the action corresponding to the element number (and follows a given policy afterwards)

To model the parametrized Q-value function within the critic, use a deep neural network with one input (the six-dimensional observed state) and one output vector with 31 elements (evenly spaced steering angles from -15 to 15 degrees).

```
% Define number of inputs, neurons, and outputs
nI = observationInfo.Dimension(1);  % number of inputs (6)
nL = 24;                            % number of neurons
nO = numel(actionInfo.Elements);    % number of outputs (31)

% Define network as array of layer objects
dnn = [
    featureInputLayer(nI)
    fullyConnectedLayer(nL)
    reluLayer
    fullyConnectedLayer(nL)
    reluLayer
    fullyConnectedLayer(nO)
    ];

% Convert to dlnetwork object
dnn = dlnetwork(dnn);
```

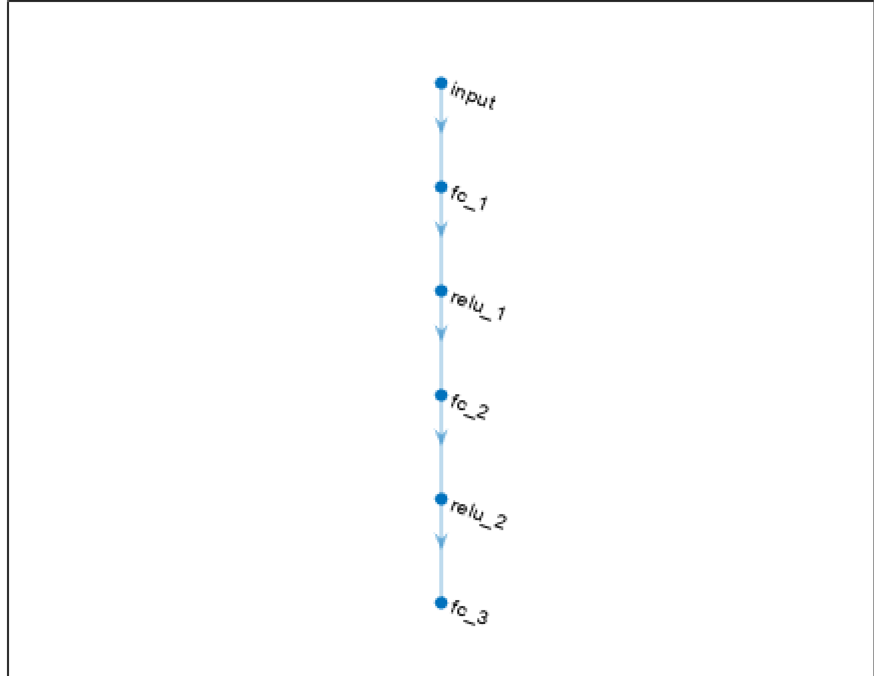
Display the number of parameters and view the network configuration.

```
summary(dnn)
```

```
   Initialized: true

   Number of learnables: 1.5k
```

```
   Inputs:
      1   'input'   6 features
```

```
plot(dnn)
```




Create the critic using `dnn` and the observation and action specifications. For more information, see `rlQValueFunction`.

```
critic = rlVectorQValueFunction(dnn,observationInfo,actionInfo);
```

Specify training options for the critic using `rlOptimizerOptions`.

```
criticOptions = rlOptimizerOptions( ...
    LearnRate=1e-4, ...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-4);
```

Specify the DQN agent options using `rlDQNAgentOptions`, include the training options for the critic.

```
agentOptions = rlDQNAgentOptions(...
    SampleTime=Ts,...
    UseDoubleDQN=true,...
    CriticOptimizerOptions=criticOptions,...
    ExperienceBufferLength=1e6,...
    MiniBatchSize=64);
```

Then, create the DQN agent using the specified critic representation and agent options. For more information, see `rlDQNAgent`.

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

### Train Agent

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

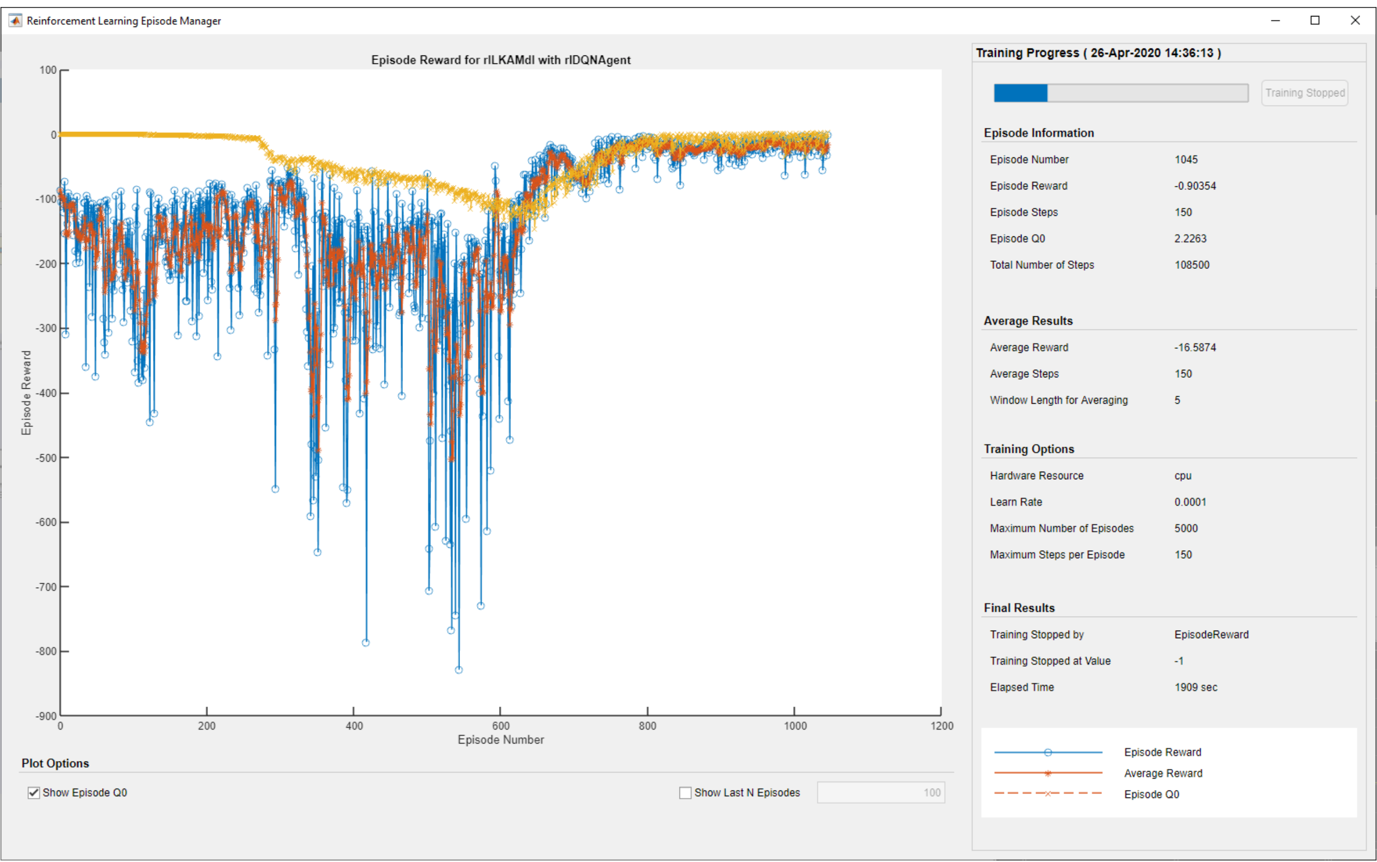
- Run each training episode for at most 5000 episodes, with each episode lasting at most `ceil(T/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option to `training-progress`) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the episode reward reaches –1.
- Save a copy of the agent for each episode where the cumulative reward is greater than –2.5.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(T/Ts);
trainingOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="EpisodeReward",...
    StopTrainingValue=-1,...
    SaveAgentCriteria="EpisodeReward",...
    SaveAgentValue=-2.5);
```

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

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load the pretrained agent for the example.
    load("SimulinkLKADQNMulti.mat","agent")
end
```




### Simulate DQN Agent

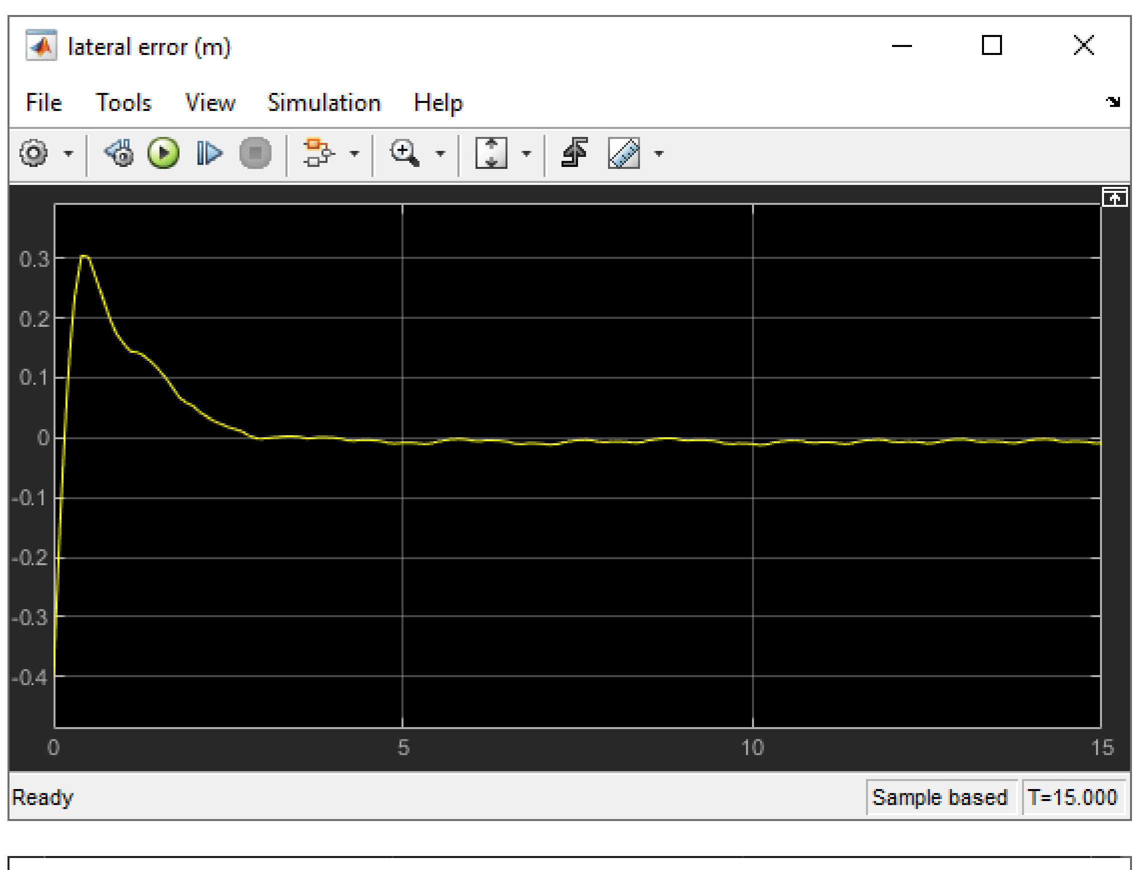
To validate the performance of the trained agent, uncomment the following two lines and simulate the agent within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

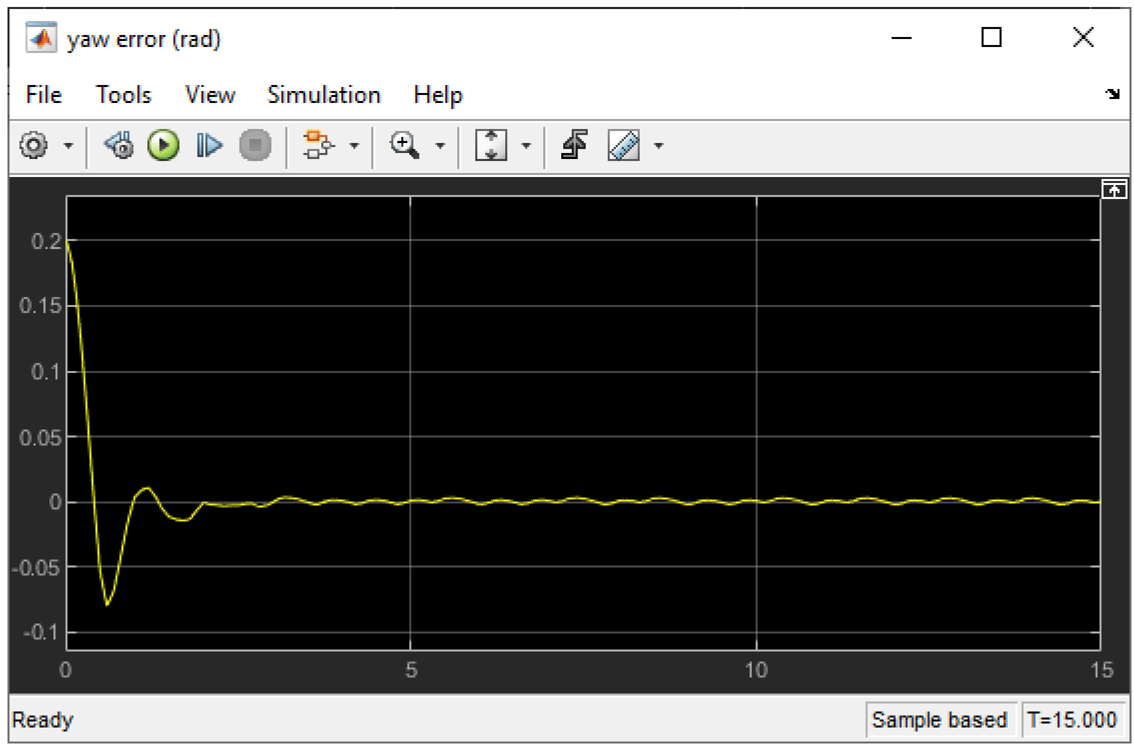
```
% simOptions = rlSimulationOptions(MaxSteps=maxsteps);
% experience = sim(env,agent,simOptions);
```

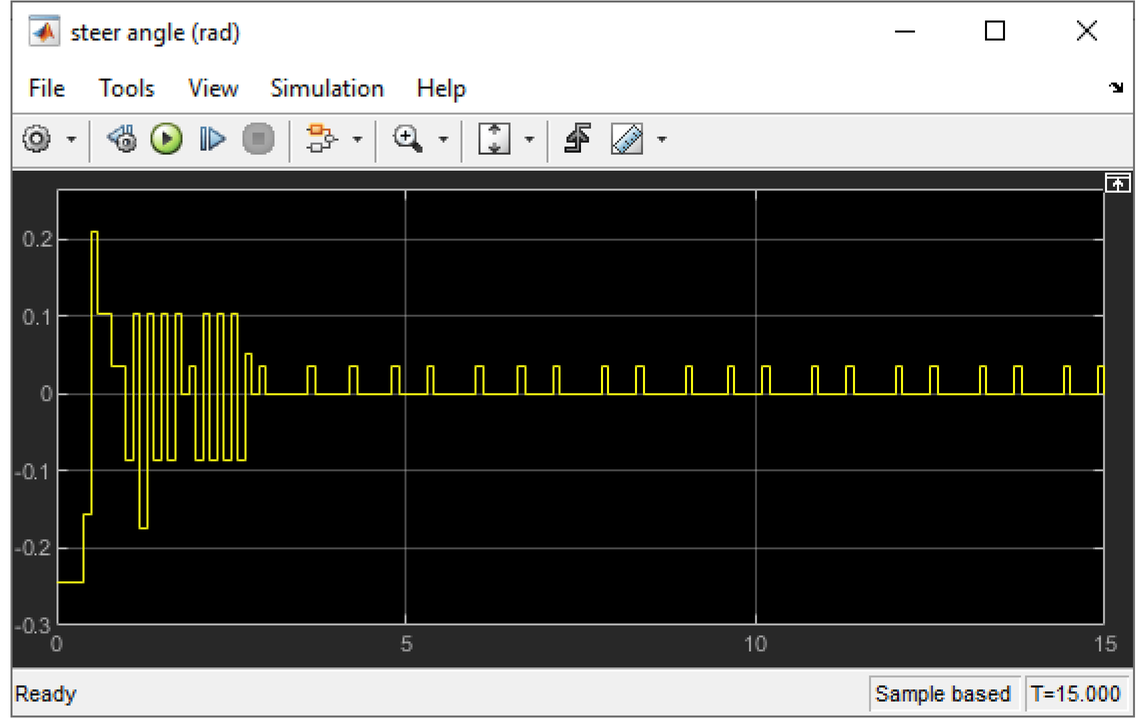
To demonstrate the trained agent on deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.2;
sim(mdl)
```

As the plots show, the lateral error (top plot) and relative yaw angle (middle plot) are both driven close to zero. The vehicle starts from off the centerline (–0.4 m) and with a nonzero yaw angle error (0.2 rad). The lane keeping assist makes the ego car travel along the centerline after about 2.5 seconds. The steering angle (bottom plot) shows that the controller reaches steady state after about 2 seconds.


lateral error (m)
File Tools View Simulation Help
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
5
10
15
Ready
Sample based
T=15.000


yaw error (rad)
File Tools View Simulation Help
0.2
0.15
0.1
0.05
0
-0.05
-0.1
0
5
10
15
Ready
Sample based
T=15.000




Close the Simulink model.

```
if ~doTraining
    %bdclose(mdl)
end
```

### Reset Function

```
function in = localResetFcn(in)
    % reset lateral deviation and relative yaw angle to random values
    in = setVariable(in,"e1_initial", 0.5*(-1+2*rand));
    in = setVariable(in,"e2_initial", 0.1*(-1+2*rand));
end
```

## See Also

**Functions**
train | sim | rlSimulinkEnv

**Objects**
rlDQNAgent | rlDQNAgentOptions | rlTrainingOptions | rlOptimizerOptions | rlSimulationOptions

**Blocks**
RL Agent

## Related Examples

- "Train DQN Agent for Lane Keeping Assist Using Parallel Computing" on page 5-257

- “Train DDPG Agent for Path-Following Control” on page 5-247
- “Lane Keeping Assist System Using Model Predictive Control” (Model Predictive Control Toolbox)

## More About

- “Create Policies and Value Functions” on page 4-2
- “Deep Q-Network (DQN) Agents” on page 3-23
- “Train Reinforcement Learning Agents” on page 5-3

# Train PPO Agent for Automatic Parking Valet

This example demonstrates the design of a hybrid controller for an automatic search and parking task. The hybrid controller uses model predictive control (MPC) to follow a reference path in a parking lot and a trained reinforcement learning (RL) agent to perform the parking maneuver.

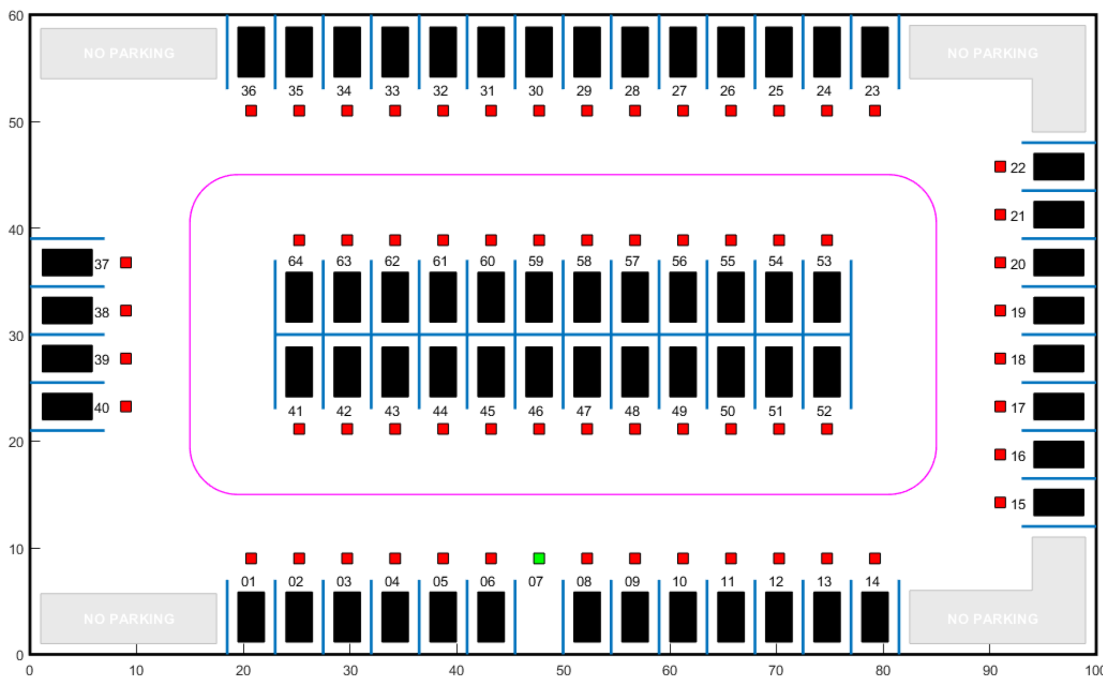
The automatic parking algorithm in this example executes a series of maneuvers while simultaneously sensing and avoiding obstacles in tight spaces. It switches between an adaptive MPC controller and an RL agent to complete the parking maneuver. The MPC controller moves the vehicle at a constant speed along a reference path while an algorithm searches for an empty parking spot. When a spot is found, the RL Agent takes over and executes a pretrained parking maneuver. Prior knowledge of the environment (the parking lot) including the locations of the empty spots and parked vehicles is available to the controllers.

### Parking Lot

The parking lot is represented by the `ParkingLot` class, which stores information about the ego vehicle, empty parking spots, and static obstacles (parked cars). Each parking spot has a unique index number and an indicator light that is either green (free) or red (occupied). Parked vehicles are represented in black.

Create a `ParkingLot` object with a free spot at location 7.

```
freeSpotIdx = 7;
map = ParkingLot(freeSpotIdx);
```



Specify an initial pose $(X_0, Y_0, \theta_0)$ for the ego vehicle. The target pose is determined based on the first available free spot as the vehicle navigates the parking lot.

```
egoInitialPose = [20, 15, 0];
```

Compute the target pose for the vehicle using the `createTargetPose` function. The target pose corresponds to the location in `freeSpotIdx`.

```
egoTargetPose = createTargetPose(map,freeSpotIdx)
```

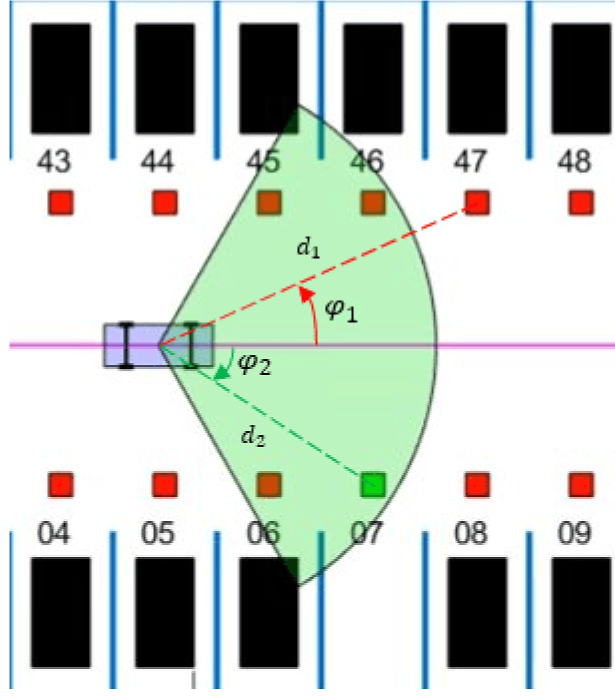
```
egoTargetPose = 1×3

   47.7500    4.9000   -1.5708
```

**Sensor Modules**

The parking algorithm uses camera and lidar sensors to gather information from the environment.

**Camera**

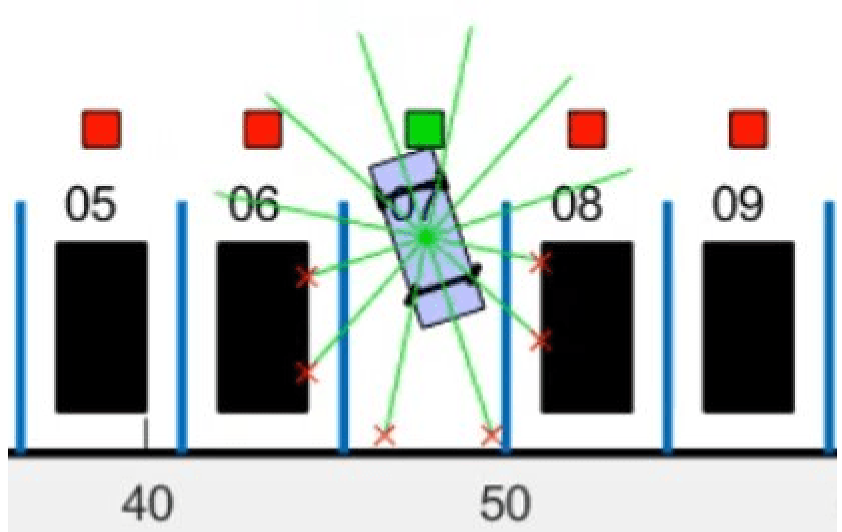
The field of view of a camera mounted on the ego vehicle is represented by the area shaded in green in the following figure. The camera has a field of view $\varphi$ bounded by $\pm 60$ degrees and a maximum measurement depth $d_{max}$ of 10 m.




As the ego vehicle moves forward, the camera module senses the parking spots that fall within the field of view and determines whether a spot is free or occupied. For simplicity, this action is implemented using geometrical relationships between the spot locations and the current vehicle pose. A parking spot is within the camera range if $d_i \le d_{max}$ and $\varphi_{min} \le \varphi_i \le \varphi_{max}$, where $d_i$ is the distance to the parking spot and $\varphi_i$ is the angle to the parking spot.

### Lidar

The reinforcement learning agent uses lidar sensor readings to determine the proximity of the ego vehicle to other vehicles in the environment. The lidar sensor in this example is also modeled using geometrical relationships. Lidar distances are measured along 12 line segments that radially emerge from the center of the ego vehicle. When a lidar line intersects an obstacle, it returns the distance of the obstacle from the vehicle. The maximum measurable lidar distance along any line segment is 6 m.




### Auto Parking Valet Model

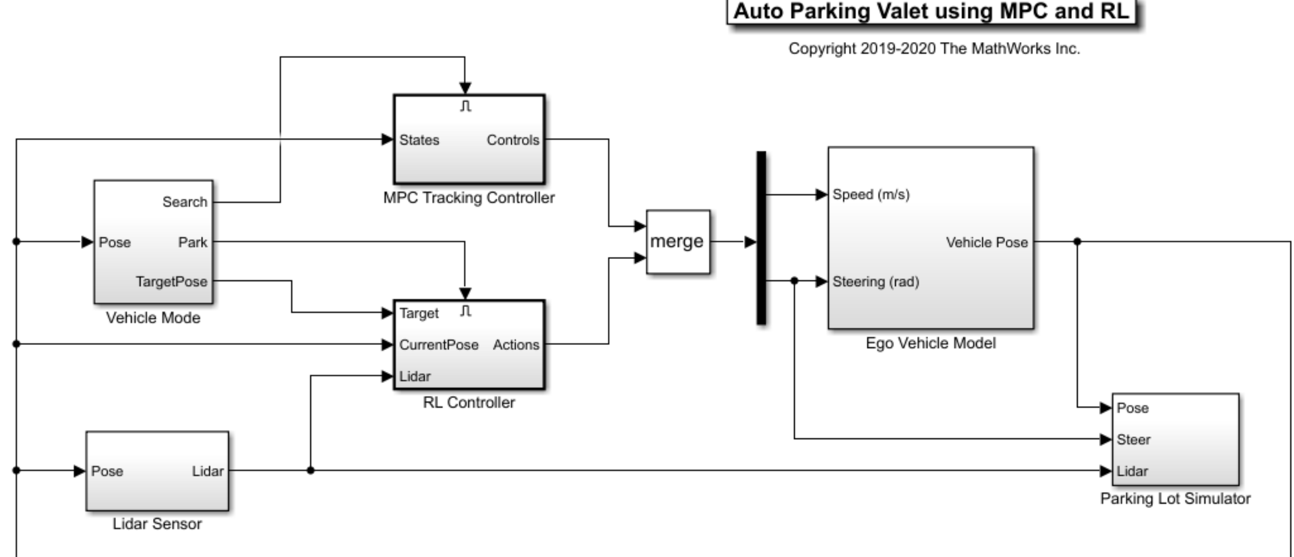
The parking valet model, including the controllers, ego vehicle, sensors, and parking lot, is implemented in Simulink®.

Load the auto parking valet parameters.

```
autoParkingValetParams
```

Open the Simulink model.

```
mdl = "rlAutoParkingValet";
open_system(mdl)
```




The ego vehicle dynamics in this model are represented by a single-track bicycle model with two inputs: vehicle speed $v$ (m/s) and steering angle $\delta$ (radians). The MPC and RL controllers are placed within Enabled Subsystem blocks that are activated by signals representing whether the vehicle has to search for an empty spot or execute a parking maneuver. The enable signals are determined by the Camera algorithm within the Vehicle Mode subsystem. Initially, the vehicle is in *search* mode and the MPC controller tracks the reference path. When a free spot is found, *park* mode is activated and the RL agent executes the parking maneuver.

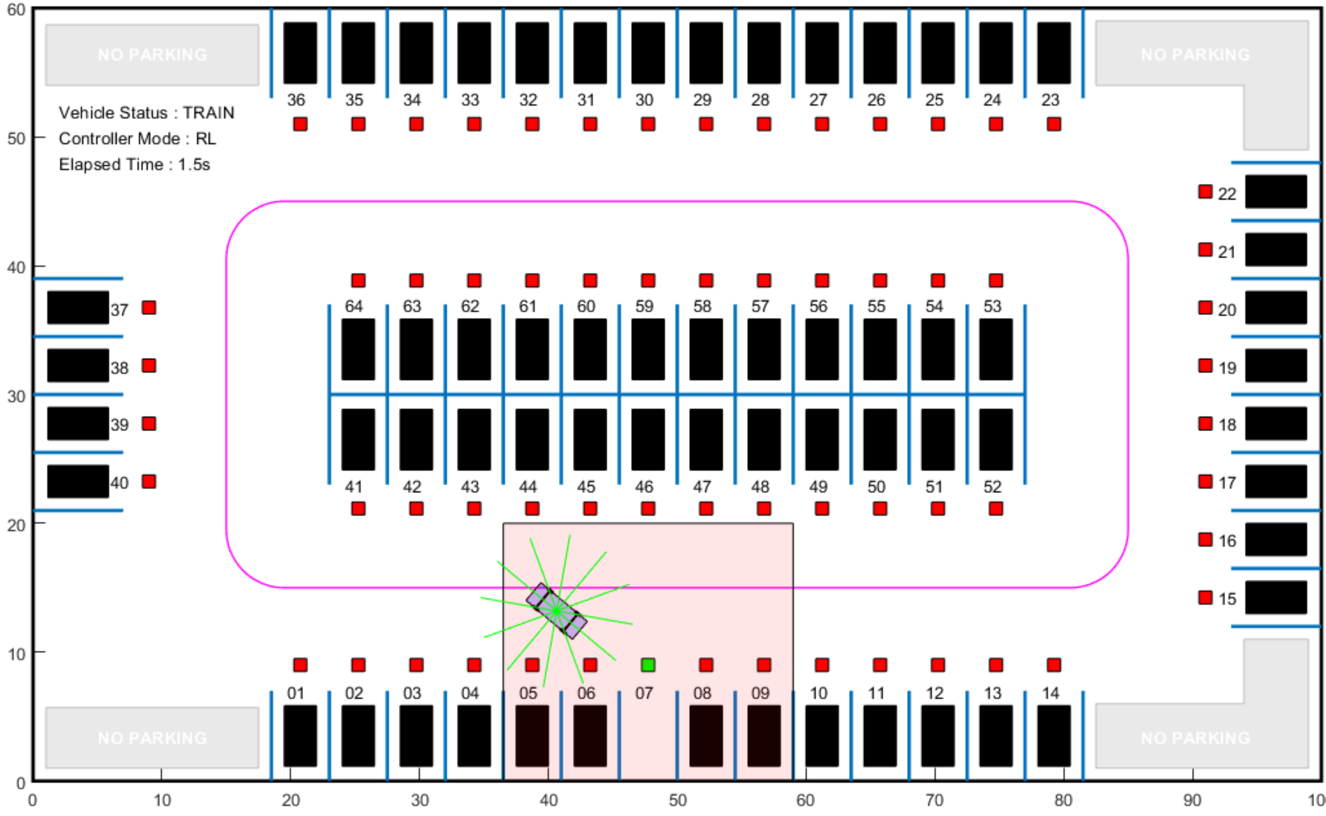
### Adaptive Model Predictive Controller

Create the adaptive MPC controller object for reference trajectory tracking using the `createMPCForParking` script. For more information on adaptive MPC, see “Adaptive MPC” (Model Predictive Control Toolbox).

```
createMPCForParking
```

### Reinforcement Learning Environment

The environment for training the RL agent is the region shaded in red in the following figure. Due to symmetry in the parking lot, training within this region is sufficient for the policy to adjust to other regions after applying appropriate coordinate transformations to the observations. Using this smaller training region significantly reduces training time compared to training over the entire parking lot.




For this environment:

- The training region is a 22.5 m x 20 m space with the target spot at its horizontal center.
- The observations are the position errors $X_e$ and $Y_e$ of the ego vehicle with respect to the target pose, the sine and cosine of the true heading angle $\theta$, and the lidar sensor readings.
- The vehicle speed during parking is a constant 2 m/s.
- The action signals are discrete steering angles that range between +/- 45 degrees in steps of 15 degrees.
- The vehicle is considered parked if the errors with respect to target pose are within specified tolerances of +/- 0.75 m (position) and +/-10 degrees (orientation).
- The episode terminates if the ego vehicle goes out of the bounds of the training region, collides with an obstacle, or parks successfully.
- The reward $r_t$ provided at time $t$, is:

$$r_t = 2e^{-\left(0.05X_e^2 + 0.04Y_e^2\right)} + 0.5e^{-40\theta_e^2} - 0.05\delta^2 + 100f_t - 50g_t$$

Here, $X_e$, $Y_e$, and $\theta_e$ are the position and heading angle errors of the ego vehicle from the target pose, and $\delta$ is the steering angle. $f_t$ (0 or 1) indicates whether the vehicle has parked and $g_t$ (0 or 1) indicates if the vehicle has collided with an obstacle at time $t$.

The coordinate transformations on vehicle pose $(X, Y, \theta)$ observations for different parking spot locations are as follows:

- 1-14: no transformation
- 15-22: $\bar{X} = Y, \bar{Y} = -X, \bar{\theta} = \theta - \pi/2$
- 23-36: $\bar{X} = 100 - X, \bar{Y} = 60 - Y, \bar{\theta} = \theta - \pi$
- 37-40: $\bar{X} = 60 - Y, \bar{Y} = X, \bar{\theta} = \theta - 3\pi/2$
- 41-52: $\bar{X} = 100 - X, \bar{Y} = 30 - Y, \bar{\theta} = \theta + \pi$
- 53-64: $\bar{X} = X, \bar{Y} = Y - 28, \bar{\theta} = \theta$

Create the observation and action specifications for the environment.

```
numObservations = 16;
obsInfo = rlNumericSpec([numObservations 1]);
obsInfo.Name = "observations";

steerMax = pi/4;
discreteSteerAngles = -steerMax : deg2rad(15) : steerMax;
actInfo = rlFiniteSetSpec(num2cell(discreteSteerAngles));
actInfo.Name = "actions";
numActions = numel(actInfo.Elements);
```

Create the Simulink environment interface, specifying the path to the RL Agent block.

```
blk = mdl + "/RL Controller/RL Agent";
env = rlSimulinkEnv(mdl,blk,obsInfo,actInfo);
```

Specify a reset function for training. The `autoParkingValetResetFcn` function resets the initial pose of the ego vehicle to random values at the start of each episode.

```
env.ResetFcn = @autoParkingValetResetFcn;
```

For more information on creating Simulink environments, see `rlSimulinkEnv`.

### Create Agent

The RL agent in this example is a proximal policy optimization (PPO) agent with a discrete action space. To learn more about PPO agents, see “Proximal Policy Optimization (PPO) Agents” on page 3-49.

Set the random seed generator for reproducibility.

```
rng(0)
```

PPO agents use a parametrized value function approximator to estimate the value of the policy. A value-function critic takes the current observation as input and returns a single scalar as output (the estimated discounted cumulative long-term reward for following the policy from the state corresponding to the current observation).

To model the parametrized value function within the critic, use a neural network with one input layer (which receives the content of the observation channel, as specified by `obsInfo`) and one output layer (which returns the scalar value).

Define the network as an array of layer objects.

```
criticNet = [
    featureInputLayer(numObservations)
```

```
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(1)
    ];
```

Convert the network to a `dlnetwork` object, and display the number of parameters.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 35.3k

   Inputs:
      1   'input'   16 features
```

Create the critic for the PPO agent. For more information, see `rlValueFunction` and `rlOptimizerOptions`.

```
critic = rlValueFunction(criticNet,obsInfo);
```

Policy gradient agents use a parametrized stochastic policy, which for discrete action spaces is implemented by a discrete categorical actor. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by `actInfo`.

Define the network as an array of layer objects

```
actorNet = [
    featureInputLayer(numObservations)
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(numActions)
    softmaxLayer
    ];
```

Convert the network to a `dlnetwork` object, and display the number of parameters.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 19.5k

   Inputs:
      1   'input'   16 features
```

Create a discrete categorical actor for the PPO agent. For more information, see `rlDiscreteCategoricalActor`.

```
actor = rlDiscreteCategoricalActor(actorNet,obsInfo,actInfo);
```

Specify training options for the critic and the actor using `rlOptimizerOptions`.

```
criticOptions = rlOptimizerOptions( ...
    LearnRate=1e-3, ...
    GradientThreshold=1);
actorOptions = rlOptimizerOptions( ...
    LearnRate=2e-4, ...
    GradientThreshold=1);
```

Specify the agent options using `rlPPOAgentOptions`, include the options for the actor and the critic.

```
agentOpts = rlPPOAgentOptions(...
    SampleTime=Ts,...
    ActorOptimizerOptions=actorOptions,...
    CriticOptimizerOptions=criticOptions,...
    ExperienceHorizon=200,...
    ClipFactor=0.2,...
    EntropyLossWeight=0.01,...
    MiniBatchSize=64,...
    NumEpoch=3,...
    AdvantageEstimateMethod="gae",...
    GAEFactor=0.95,...
    DiscountFactor=0.998);
```

According to these training options, the agent collects experiences until it reaches experience horizon of 200 steps or the episode terminates and then trains from mini-batches of 64 experiences for three epochs. An objective function clip factor of 0.2 improves training stability and a discount factor value of 0.998 encourages long term rewards. Variance in critic the output is reduced by the generalized advantage estimate method with a GAE factor of 0.95.

Create the agent using the actor, the critic, and the agent options object.

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

### Train Agent

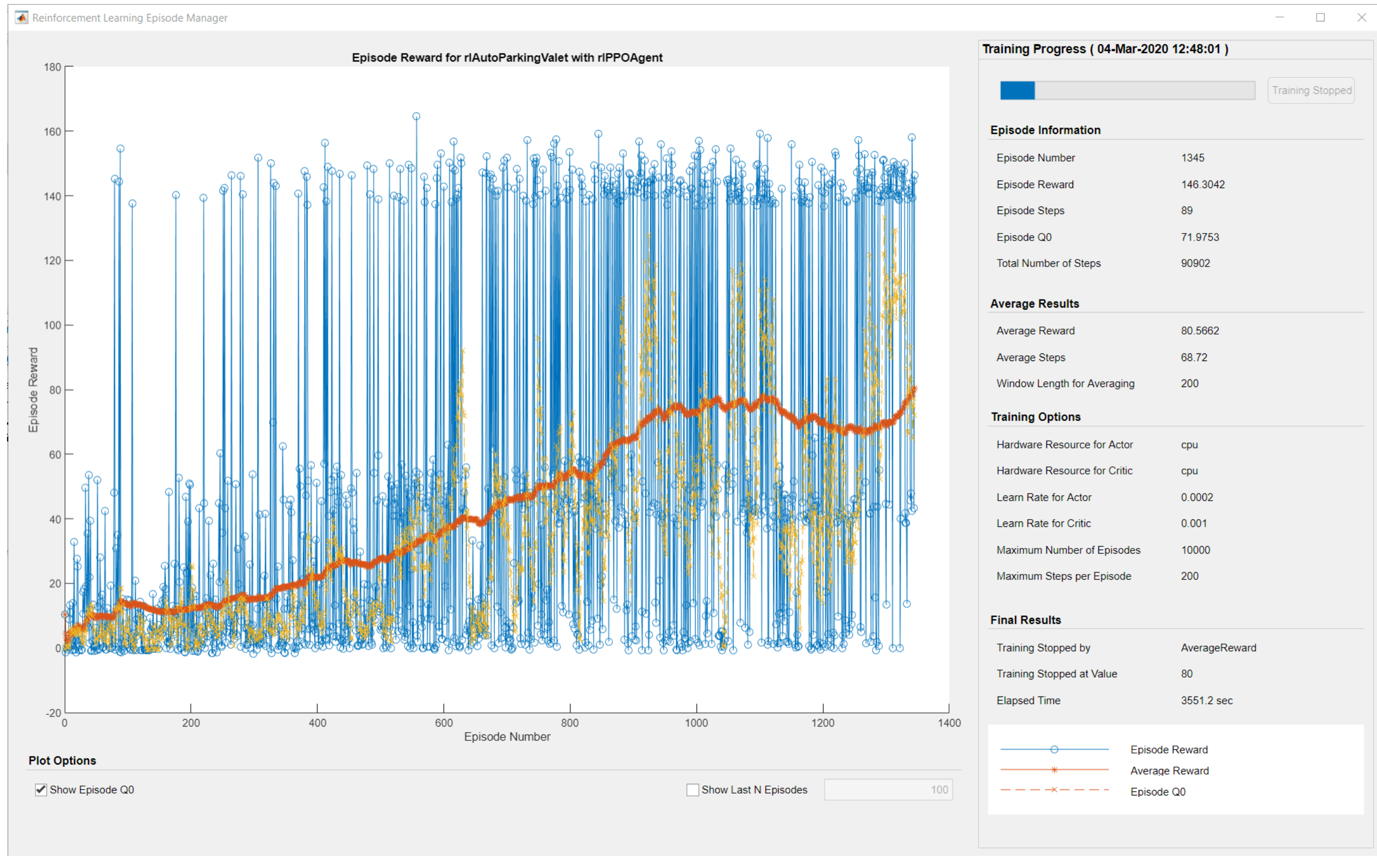
For this example, you train the agent for a maximum of 10000 episodes, with each episode lasting a maximum of 200 time steps. The training terminates when the maximum number of episodes is reached or the average reward over 100 episodes exceeds 100.

Specify the options for training using an `rlTrainingOptions` object.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=10000,...
    MaxStepsPerEpisode=200,...
    ScoreAveragingWindowLength=200,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=80);
```

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

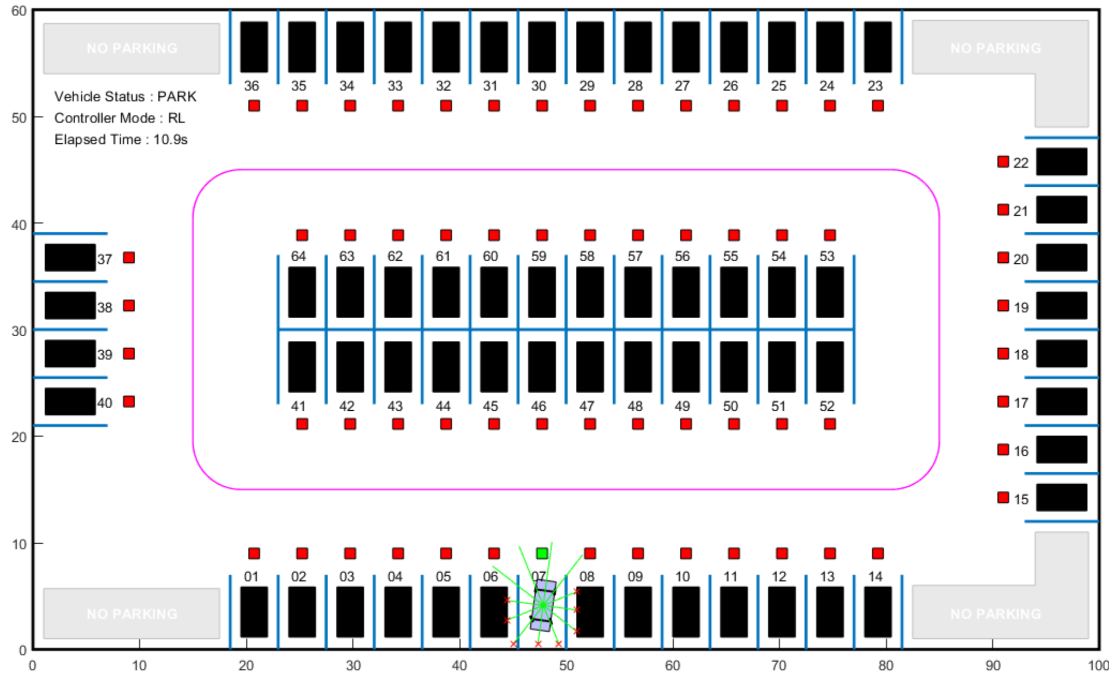
```
doTraining = false;
if doTraining
    trainingStats = train(agent,env,trainOpts);
else
    load('rlAutoParkingValetAgent.mat','agent');
end
```




### Simulate Agent

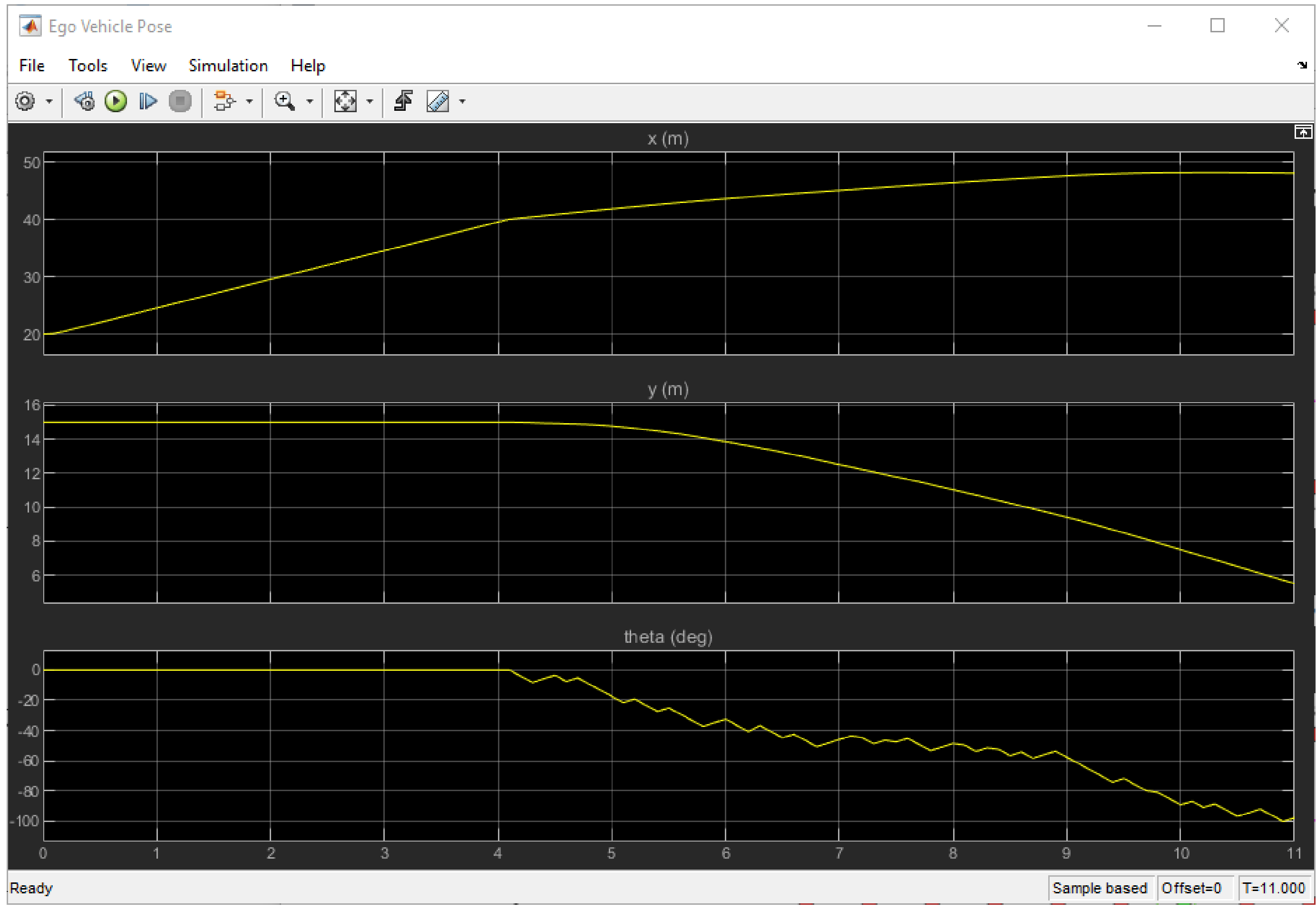
Simulate the model to park the vehicle in the free parking spot. To simulate the vehicle parking in different locations, change the free spot location in the following code.

```
freeSpotIdx = 7;  % free spot location
sim(mdl);
```




The vehicle reaches the target pose within the specified error tolerances of +/- 0.75 m (position) and +/-10 degrees (orientation).

To view the ego vehicle position and orientation, open the Ego Vehicle Pose scope.




## See Also

**Functions**
train | sim | rlSimulinkEnv

**Objects**
rlPPOAgent | rlPPOAgentOptions | rlOptimizerOptions | rlTrainingOptions | rlSimulationOptions

**Blocks**
RL Agent

## Related Examples

- “Automatic Parking Valet with Unreal Engine Simulation” on page 5-300
- “Parking Valet Using Multistage Nonlinear Model Predictive Control” (Model Predictive Control Toolbox)
- “Parallel Parking Using Nonlinear Model Predictive Control” (Model Predictive Control Toolbox)
- “Parallel Parking Using RRT Planner and MPC Tracking Controller” (Model Predictive Control Toolbox)

## More About

- “Create Policies and Value Functions” on page 4-2
- “Train Reinforcement Learning Agents” on page 5-3

# Train DDPG Agent for Path-Following Control

This example shows how to train a deep deterministic policy gradient (DDPG) agent for path-following control (PFC) in Simulink®. For more information on DDPG agents, see "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40.

### Simulink Model

The reinforcement learning environment for this example consists in a simple bicycle model for the ego car together with a simple longitudinal model for the lead car. The training goal is to make the ego car travel at a set velocity while maintaining a safe distance from lead car by controlling longitudinal acceleration and braking, and also while keeping the ego car travelling along the centerline of its lane by controlling the front steering angle. For more information on PFC, see Path Following Control System (Model Predictive Control Toolbox). The ego car dynamics are specified by the following parameters.

```
m = 1600;   % total vehicle mass (kg)
Iz = 2875;  % yaw moment of inertia (mNs^2)
lf = 1.4;   % long. distance from center of gravity to front tires (m)
lr = 1.6;   % long. distance from center of gravity to rear tires (m)
Cf = 19000; % cornering stiffness of front tires (N/rad)
Cr = 33000; % cornering stiffness of rear tires (N/rad)
tau = 0.5;  % longitudinal time constant
```

Specify the initial position and velocity for the two vehicles.

```
x0_lead = 50;   % initial position for lead car (m)
v0_lead = 24;   % initial velocity for lead car (m/s)
x0_ego = 10;    % initial position for ego car (m)
v0_ego = 18;    % initial velocity for ego car (m/s)
```

Specify the standstill default spacing (m), time gap (s), and driver-set velocity (m/s).

```
D_default = 10;
t_gap = 1.4;
v_set = 28;
```

To simulate the physical limitations of the vehicle dynamics, constrain the acceleration to the range `[–3,2]` (m/s^2), and the steering angle to the range `[–0.2618,0.2618]` (rad), that is -15 and 15 degrees.

```
amin_ego = -3;
amax_ego = 2;
umin_ego = -0.2618; % +15 deg
umax_ego = 0.2618; % -15 deg
```

The curvature of the road is defined by a constant 0.001 ($m^{-1}$). The initial value for lateral deviation is 0.2 m and the initial value for the relative yaw angle is –0.1 rad.

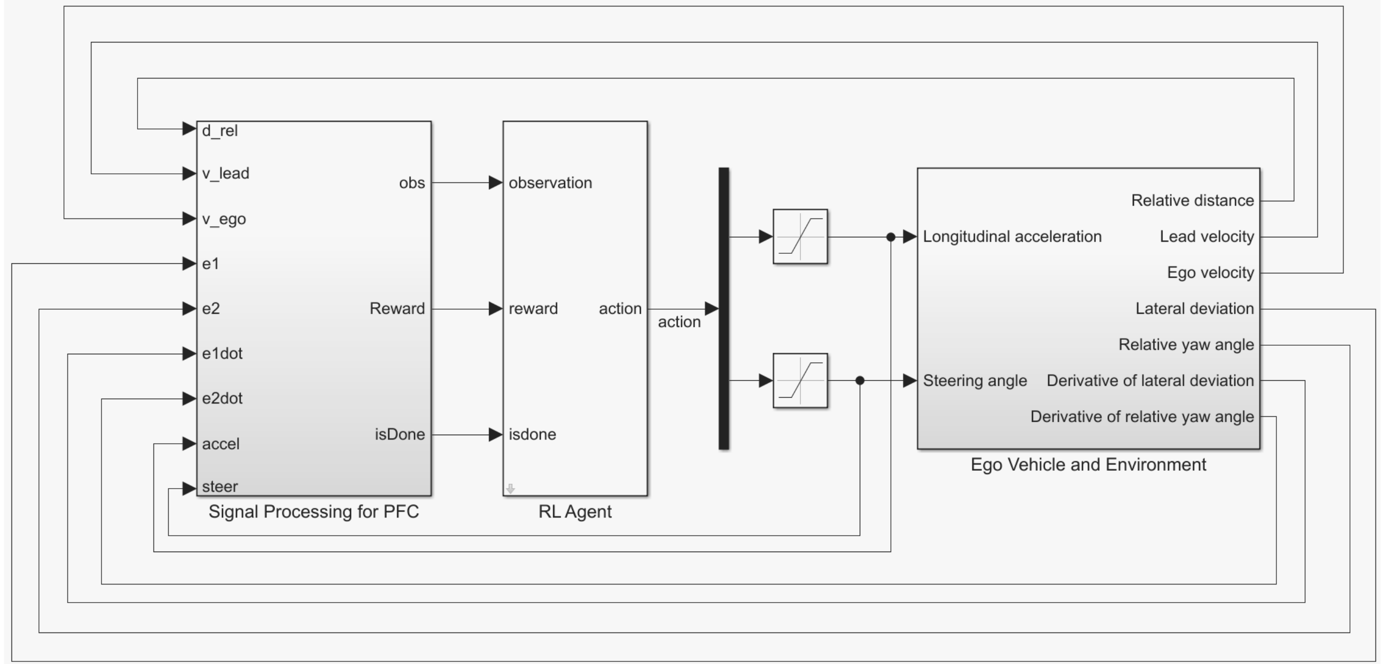
```
rho = 0.001;
e1_initial = 0.2;
e2_initial = -0.1;
```

Define the sample time `Ts` and simulation duration `Tf` in seconds.

```
Ts = 0.1;
Tf = 60;
```

Open the model.

```
mdl = "rlPFCMdl";
open_system(mdl)
agentblk = mdl + "/RL Agent";
```




For this model:

- The action signal consists of acceleration and steering angle actions. The acceleration action signal takes value between –3 and 2 (m/s^2). The steering action signal takes a value between –15 degrees (–0.2618 rad) and 15 degrees (0.2618 rad).
- The reference velocity for the ego car $V_{ref}$ is defined as follows. If the relative distance is less than the safe distance, the ego car tracks the minimum of the lead car velocity and driver-set velocity. In this manner, the ego car maintains some distance from the lead car. If the relative distance is greater than the safe distance, the ego car tracks the driver-set velocity. In this example, the safe distance is defined as a linear function of the ego car longitudinal velocity $V$, that is, $t_{gap} * V + D_{default}$. The safe distance determines the tracking velocity for the ego car.
- The observations from the environment contain the longitudinal measurements: the velocity error $e_V = V_{ref} - V_{ego}$, its integral $\int e$, and the ego car longitudinal velocity $V$. In addition, the observations contain the lateral measurements: the lateral deviation $e_1$, relative yaw angle $e_2$, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.
- The simulation terminates when the lateral deviation $|e_1| > 1$, when the longitudinal velocity $V_{ego} < 0.5$, or when the relative distance between the lead car and ego car $D_{rel} < 0$.
- The reward $r_t$, provided at every time step $t$, is

$$r_t = -(100e_1^2 + 500u_{t-1}^2 + 10e_V^2 + 100a_{t-1}^2) \times 1e^{-3} - 10F_t + 2H_t + M_t$$

where $u_{t-1}$ is the steering input from the previous time step $t-1$, $a_{t-1}$ is the acceleration input from the previous time step. The three logical values are as follows.

- $F_t = 1$if simulation is terminated, otherwise $F_t = 0$
- $H_t = 1$ if lateral error $e_1^2 < 0.01$, otherwise $H_t = 0$
- $M_t = 1$ if velocity error $e_V^2 < 1$, otherwise $M_t = 0$

The three logical terms in the reward encourage the agent to make both lateral error and velocity error small, and in the meantime, penalize the agent if the simulation is terminated early.

#### Create Environment Interface

Create an environment interface for the Simulink model.

Create the observation specification.

```
obsInfo = rlNumericSpec([9 1], ...
    LowerLimit=-inf*ones(9,1), ...
    UpperLimit=inf*ones(9,1));
obsInfo.Name = "observations";
```

Create the action specification.

```
actInfo = rlNumericSpec([2 1], ...
    LowerLimit=[-3;-0.2618], ...
    UpperLimit=[2;0.2618]);
actInfo.Name = "accel;steer";
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
```

To define the initial conditions, specify an environment reset function using an anonymous function handle. The reset function `localResetFcn`, which is defined at the end of the example, randomizes the initial position of the lead car, the lateral deviation, and the relative yaw angle.

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

#### Create DDPG Agent

DDPG agents use a parametrized Q-value function approximator to estimate the value of the policy. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward given the action from the state corresponding to the current observation, and following the policy thereafter).

To model the parametrized Q-value function within the critic, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value). Note that `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` return the number of dimensions of

the observation and action spaces, respectively, regardless of whether they are arranged as row vectors, column vectors, or matrices.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% Number of neurons
L = 100;

% Main path
mainPath = [
    featureInputLayer(prod(obsInfo.Dimension),Name="obsInLyr")
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(L)
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(1,Name="QValLyr")
    ];

% Action path
actionPath = [
    featureInputLayer(prod(actInfo.Dimension),Name="actInLyr")
    fullyConnectedLayer(L,Name="actOutLyr")
    ];

% Assemble layergraph object
criticNet = layerGraph(mainPath);
criticNet = addLayers(criticNet,actionPath);
criticNet = connectLayers(criticNet,"actOutLyr","add/in2");
```

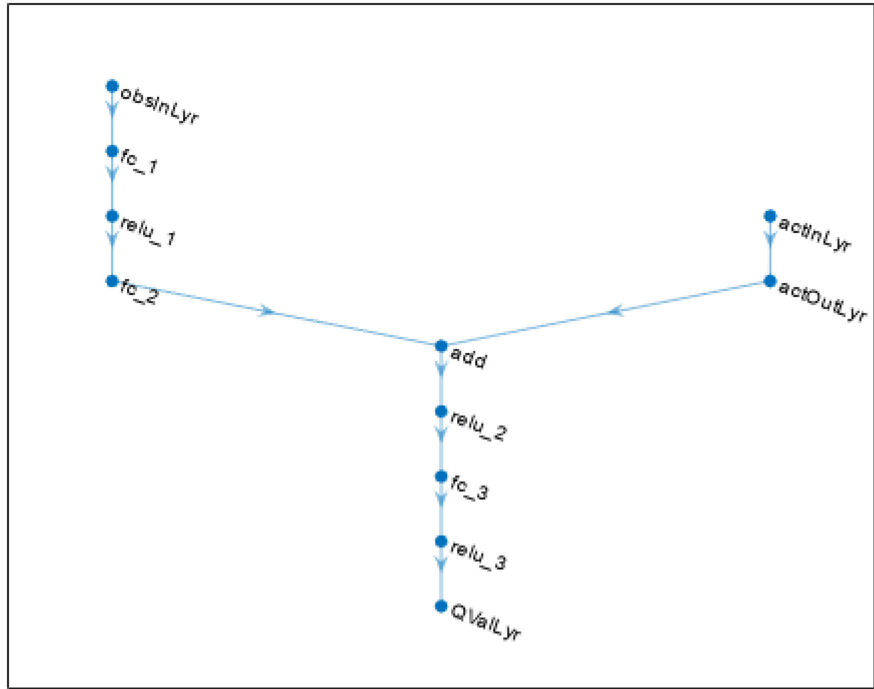
Convert to dlnetwork object and display the number of weights.

```
criticNet = dlnetwork(criticNet);
summary(criticNet)
```

```
   Initialized: true

   Number of learnables: 21.6k

   Inputs:
      1   'obsInLyr'   9 features
      2   'actInLyr'   2 features
```

View the critic network configuration.

```
figure
plot(criticNet)
```




Create the critic using the specified neural network and the environment action and observation specifications. Pass as additional arguments also the names of the network layers to be connected with the observation and action channels. For more information, see `rlQValueFunction`.

```
critic = rlQValueFunction(criticNet,obsInfo,actInfo,...
    ObservationInputNames="obsInLyr",ActionInputNames="actInLyr");
```

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer (which returns the action to the environment action channel, as specified by `actInfo`).

Define the network as an array of layer objects.

```
actorNet = [
    featureInputLayer(prod(obsInfo.Dimension))
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(L)
    reluLayer
    fullyConnectedLayer(2)
    tanhLayer
```

```
    scalingLayer(Scale=[2.5;0.2618],Bias=[-0.5;0])
    ];
```

Convert to dlnetwork object and display the number of weights.

```
actorNet = dlnetwork(actorNet);
summary(actorNet)
```

```
   Initialized: true

   Number of learnables: 21.4k

   Inputs:
      1   'input'   9 features
```

Construct the actor similarly to the critic. For more information, see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actorNet,obsInfo,actInfo);
```

Specify training options for the critic and the actor using `rlOptimizerOptions`.

```
criticOptions = rlOptimizerOptions( ...
    LearnRate=1e-3, ...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-4);
actorOptions = rlOptimizerOptions( ...
    LearnRate=1e-4, ...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-4);
```

Specify the DDPG agent options using `rlDDPGAgentOptions`, include the options for the actor and the critic.

```
agentOptions = rlDDPGAgentOptions(...
    SampleTime=Ts,...
    ActorOptimizerOptions=actorOptions,...
    CriticOptimizerOptions=criticOptions,...
    ExperienceBufferLength=1e6);
agentOptions.NoiseOptions.Variance = [0.6;0.1];
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Create the DDPG agent using the actor, the critic, and the agent options. For more information, see `rlDDPGAgent`.

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

### Train Agent

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

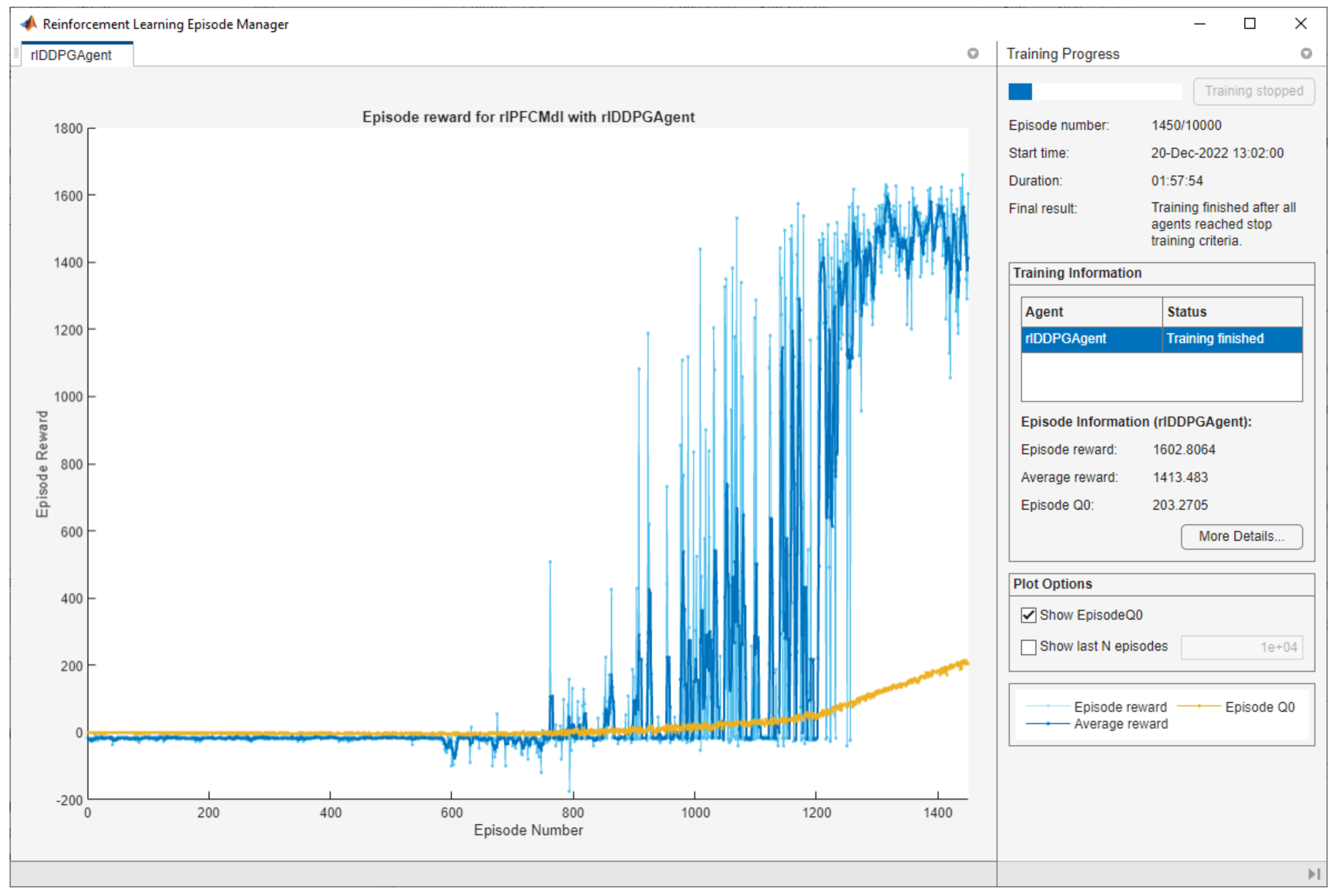
- Run each training episode for at most `10000` episodes, with each episode lasting at most `maxsteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Verbose` and `Plots` options).
- Stop training when the agent receives an cumulative episode reward greater than `1700`.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 1e4;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="EpisodeCount",...
    StopTrainingValue=1450);
```

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

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load a pretrained agent for the example.
    load("SimulinkPFCDDPG.mat","agent")
end
```



### Simulate DDPG Agent

To validate the performance of the trained agent, simulate the agent within the Simulink environment by uncommenting the following commands. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

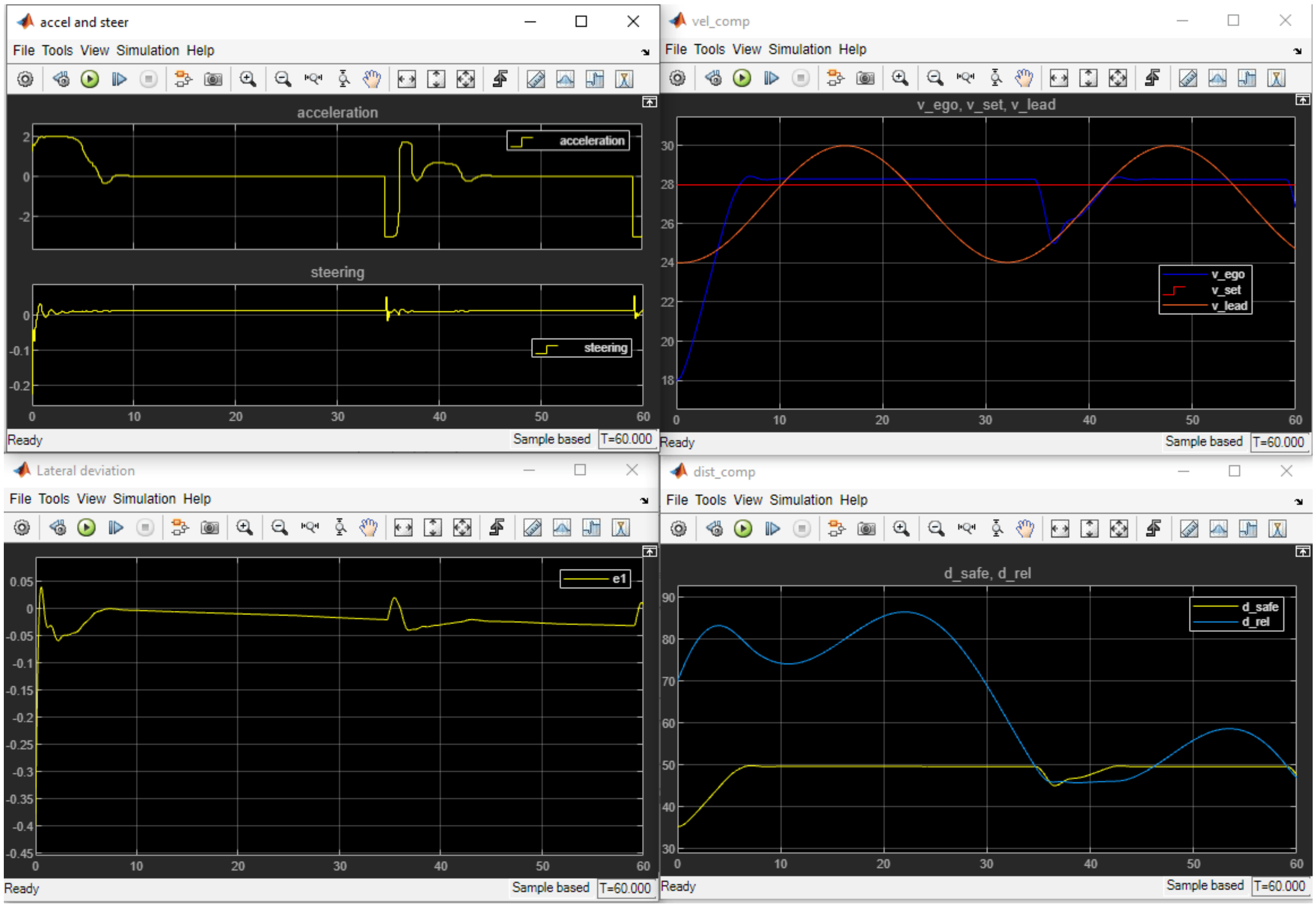
```
% simOptions = rlSimulationOptions(MaxSteps=maxsteps);
% experience = sim(env,agent,simOptions);
```

To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.1;
x0_lead = 80;
sim(mdl)
```

The following plots show the simulation results when the lead car is 70 (m) ahead of the ego car.

- In the first 35 seconds, the relative distance is greater than the safe distance (bottom-right plot), so the ego car tracks the set velocity (top-right plot). To speed up and reach the set velocity, the acceleration is mostly nonnegative (top-left plot).
- From 35 to 42 seconds, the relative distance is mostly less than the safe distance (bottom-right plot), so the ego car tracks the minimum of the lead velocity and set velocity. Because the lead velocity is less than the set velocity (top-right plot), to track the lead velocity, the acceleration becomes nonzero (top-left plot).
- From 42 to 58 seconds, the ego car tracks the set velocity (top-right plot) and the acceleration remains zero (top-left plot).
- From 58 to 60 seconds, the relative distance becomes less than the safe distance (bottom-right plot), so the ego car slows down and tracks the lead velocity.
- The bottom-left plot shows the lateral deviation. As shown in the plot, the lateral deviation is greatly decreased within 1 second. The lateral deviation remains less than 0.05 m.




Close the Simulink model.

```
bdclose(mdl)
```

**Reset Function**

```
function in = localResetFcn(in)

    % random value for initial position of lead car
    in = setVariable(in,"x0_lead",40+randi(60,1,1));

    % random value for lateral deviation
    in = setVariable(in,"e1_initial", 0.5*(-1+2*rand));

    % random value for relative yaw angle
    in = setVariable(in,"e2_initial", 0.1*(-1+2*rand));

end
```

## See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlTrainingOptions` | `rlSimulationOptions` | `rlOptimizerOptions`

**Blocks**
RL Agent

## Related Examples

- "Train DQN Agent for Lane Keeping Assist" on page 5-226
- "Train Multiple Agents for Path Following Control" on page 5-206
- "Lane Following Using Nonlinear Model Predictive Control" (Model Predictive Control Toolbox)
- "Lane Following Control with Sensor Fusion and Lane Detection" (Model Predictive Control Toolbox)

## More About

- "Create Policies and Value Functions" on page 4-2
- "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40
- "Train Reinforcement Learning Agents" on page 5-3

# Train DQN Agent for Lane Keeping Assist Using Parallel Computing

This example shows how to train a deep Q-learning network (DQN) agent for lane keeping assist (LKA) in Simulink® using parallel training. For an example that shows how to train the agent without using parallel training, see “Train DQN Agent for Lane Keeping Assist” on page 5-226.

For more information on DQN agents, see “Deep Q-Network (DQN) Agents” on page 3-23. For an example that trains a DQN agent in MATLAB®, see “Train DQN Agent to Balance Cart-Pole System” on page 5-50.

### DQN Parallel Training Overview

In a DQN agent, each worker generates new experiences from its copy of the agent and the environment. After every **N** steps, the worker sends experiences to the client agent (the agent associated with the MATLAB® process which starts the training). The client agent updates its parameters as follows.

- For asynchronous training, the client agent calculates gradients and updates agent parameters from the received experiences, without waiting to receive experiences from all the workers. The client then sends the updated parameters back to the worker that provided the experiences. Then, the worker updates its copy of the agent and continues to generate experiences using its copy of the environment.
- For synchronous training, the client agent waits to receive experiences from all of the workers and then calculates the gradients from all these experiences. The client updated the agent parameters, and sends the updated parameters to all the workers at the same time. Then, all workers use a single updated agent copy, together with their copy of the environment, to generate experiences.

### Simulink Model for Ego Car

The reinforcement learning environment for this example is a simple bicycle model for the ego vehicle dynamics. The training goal is to keep the ego vehicle traveling along the centerline of the lanes by adjusting the front steering angle. This example uses the same vehicle model as “Train DQN Agent for Lane Keeping Assist” on page 5-226.

```
m = 1575;   % total vehicle mass (kg)
Iz = 2875;  % yaw moment of inertia (mNs^2)
lf = 1.2;   % longitudinal distance from center of gravity to front tires (m)
lr = 1.6;   % longitudinal distance from center of gravity to rear tires (m)
Cf = 19000; % cornering stiffness of front tires (N/rad)
Cr = 33000; % cornering stiffness of rear tires (N/rad)
Vx = 15;    % longitudinal velocity (m/s)
```

Define the sample time `Ts` and simulation duration `T` in seconds.

```
Ts = 0.1;
T = 15;
```

The output of the LKA system is the front steering angle of the ego car. To simulate the physical steering limits of the ego car, constrain the steering angle to the range `[–0.5,0.5]` rad.

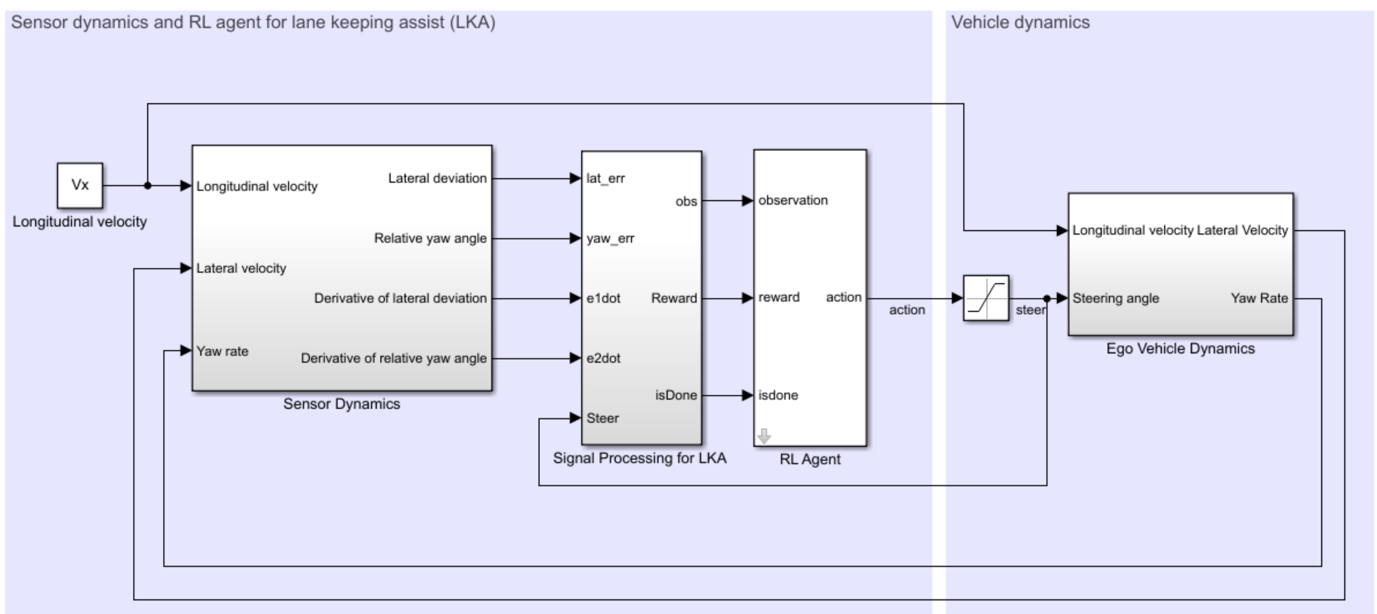
```
u_min = -0.5;
u_max = 0.5;
```

The curvature of the road is defined by a constant 0.001 ($m^{-1}$). The initial value for the lateral deviation is `0.2` m and the initial value for the relative yaw angle is `–0.1` rad.

```
rho = 0.001;
e1_initial = 0.2;
e2_initial = -0.1;
```

Open the model.

```
mdl = "rlLKAMdl";
open_system(mdl)
agentblk = mdl + "/RL Agent";
```




For this model:

- The steering-angle action signal from the agent to the environment is from –15 degrees to 15 degrees.
- The observations from the environment are the lateral deviation $e_1$, relative yaw angle $e_2$, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.
- The simulation is terminated when the lateral deviation $|e_1| > 1$.
- The reward $r_t$, provided at every time step $t$, is

$$r_t = -(10e_1^2 + 5e_2^2 + 2u^2 + 5\dot{e}_1^2 + 5\dot{e}_2^2)$$

where $u$ is the control input from the previous time step $t-1$.

### Create Environment Interface

Create a reinforcement learning environment interface for the ego vehicle.

Define the observation information.

```
obsInfo = rlNumericSpec([6 1], ...
    LowerLimit=-inf*ones(6,1), ...
    UpperLimit=inf*ones(6,1));

obsInfo.Name = "observations";
obsInfo.Description = "lateral deviation and relative yaw angle";
```

Define the action information.

```
actInfo = rlFiniteSetSpec((-15:15)*pi/180);
actInfo.Name = "steering";
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
```

The interface has a discrete action space where the agent can apply one of 31 possible steering angles from –15 degrees to 15 degrees. The observation is the six-dimensional vector containing lateral deviation, relative yaw angle, as well as their derivatives and integrals with respect to time.

To define the initial condition for the lateral deviation and relative yaw angle, specify an environment reset function using an anonymous function handle. `localResetFcn`, which is defined at the end of this example, randomizes the initial lateral deviation and relative yaw angle.

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

#### Create DQN Agent

DQN agents use a parametrized Q-value function approximator to estimate the value of the policy. Since DQN agents have a discrete action space, you have the option to create a vector (that is multi-output) Q-value function critic, which is generally more efficient than a comparable single-output critic.

A vector Q-value function takes only the observation as input and returns as output a single vector with as many elements as the number of possible actions. The value of each output element represents the expected discounted cumulative long-term reward when an agent starts from the state corresponding to the given observation and executes the action corresponding to the element number (and follows a given policy afterwards).

To model the parametrized Q-value function within the critic, use a neural network with one input (the six-dimensional observed state) and one output vector with 31 elements (evenly spaced steering angles from -15 to 15 degrees). Get the number of dimensions of the observation space and the number of elements of the discrete action space from the environment specifications.

```
nI = obsInfo.Dimension(1);      % number of inputs (6)
nL = 120;                       % number of neurons
nO = numel(actInfo.Elements);   % number of outputs (31)
```

Define the network as an array of layer objects.

```
dnn = [
    featureInputLayer(nI)
    fullyConnectedLayer(nL)
    reluLayer
    fullyConnectedLayer(nL)
```

```
    reluLayer
    fullyConnectedLayer(nO)
    ];
```

The critic network is initialized randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

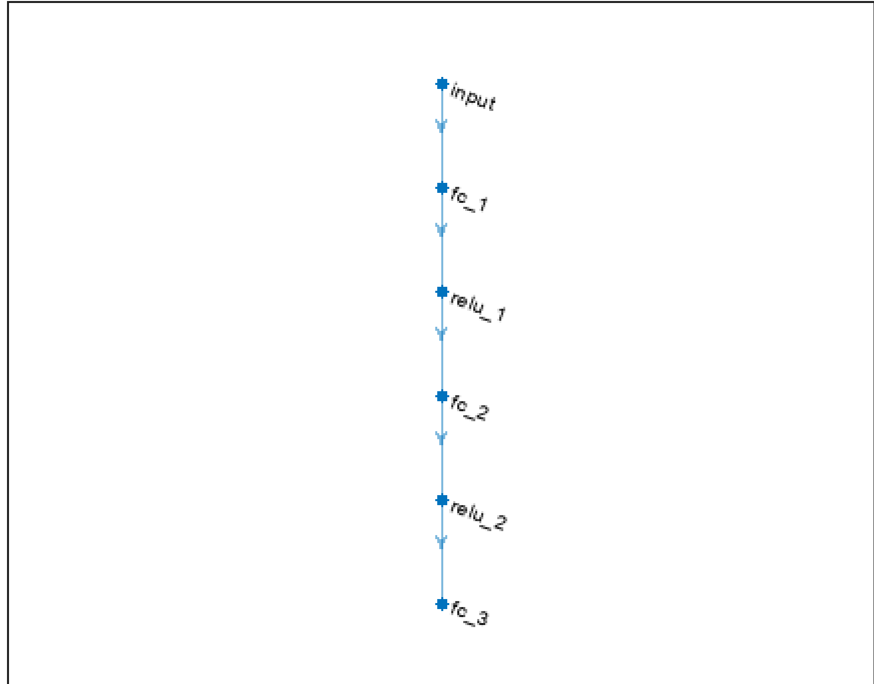
Convert to a `dlnetwork` object and display the number of parameters.

```
dnn = dlnetwork(dnn);
summary(dnn)
```

```
   Initialized: true

   Number of learnables: 19.1k

   Inputs:
      1   'input'   6 features
```

View the network configuration.

```
plot(dnn)
```




Create the critic using `dnn` and the environment specifications. For more information on vector Q-value function approximators, see `rlVectorQValueFunction`.

```
critic = rlVectorQValueFunction(dnn,obsInfo,actInfo);
```

Specify training options for the critic using `rlOptimizerOptions`.

```
criticOptions = rlOptimizerOptions( ...
    LearnRate=1e-4, ...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-4);
```

Specify the DQN agent options using `rlDQNAgentOptions`, include the critic options object.

```
agentOpts = rlDQNAgentOptions(...
    SampleTime=Ts,...
    UseDoubleDQN=true,...
    CriticOptimizerOptions=criticOptions,...
    ExperienceBufferLength=1e6,...
    MiniBatchSize=256);
```

You can also set or modify the agent options using dot notation.

```
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 1e-4;
```

Alternatively, you can create the agent first, and then access its option object and modify the options using dot notation.

Create the DQN agent using the specified critic and the agent options. For more information, see `rlDQNAgent`.

```
agent = rlDQNAgent(critic,agentOpts);
```

### Training Options

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

- Run each training for at most 10000 episodes, with each episode lasting at most `ceil(T/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box only (set the `Plots` and `Verbose` options accordingly).
- Stop training when the episode reward reaches -1.
- Save a copy of the agent for each episode where the cumulative reward is greater than 100.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 10000;
maxsteps = ceil(T/Ts);
trainOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes, ...
    MaxStepsPerEpisode=maxsteps, ...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="EpisodeReward",...
    StopTrainingValue= -1,...
    SaveAgentCriteria="EpisodeReward",...
    SaveAgentValue=100);
```

### Parallel Training Options

To train the agent in parallel, specify the following training options.

- Set the UseParallel option to true.
- Train agent in parallel asynchronously by setting the ParallelizationOptions.Mode option to "async".

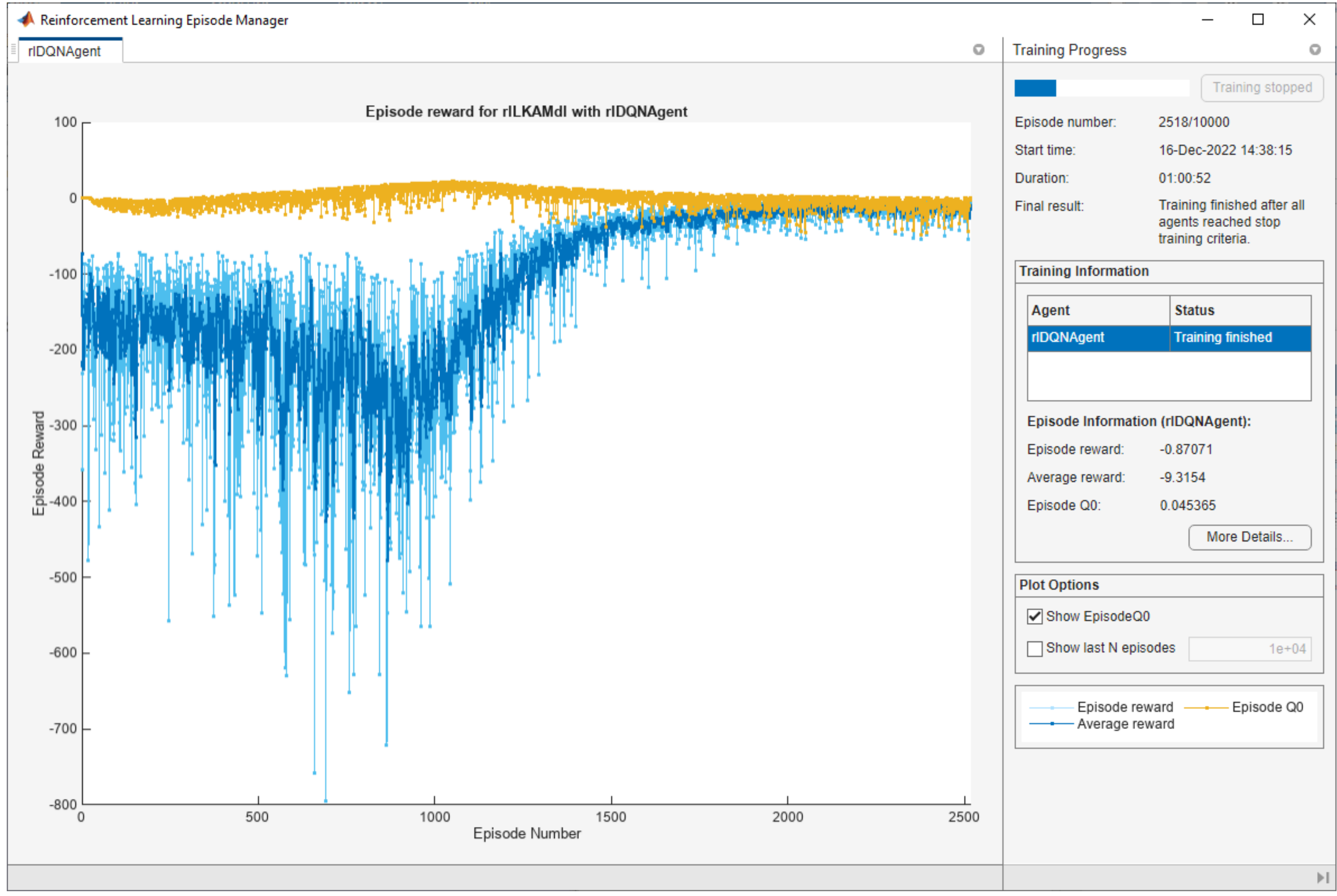
```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = "async";
```

For more information, see rlTrainingOptions.

#### Train Agent

Train the agent using the train function. Training the agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting doTraining to false. To train the agent yourself, set doTraining to true. Due to randomness of the parallel training, you can expect different training results from the plot below. The plot shows the result of training with four workers.

```
doTraining = false;

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




## Simulate the Agent

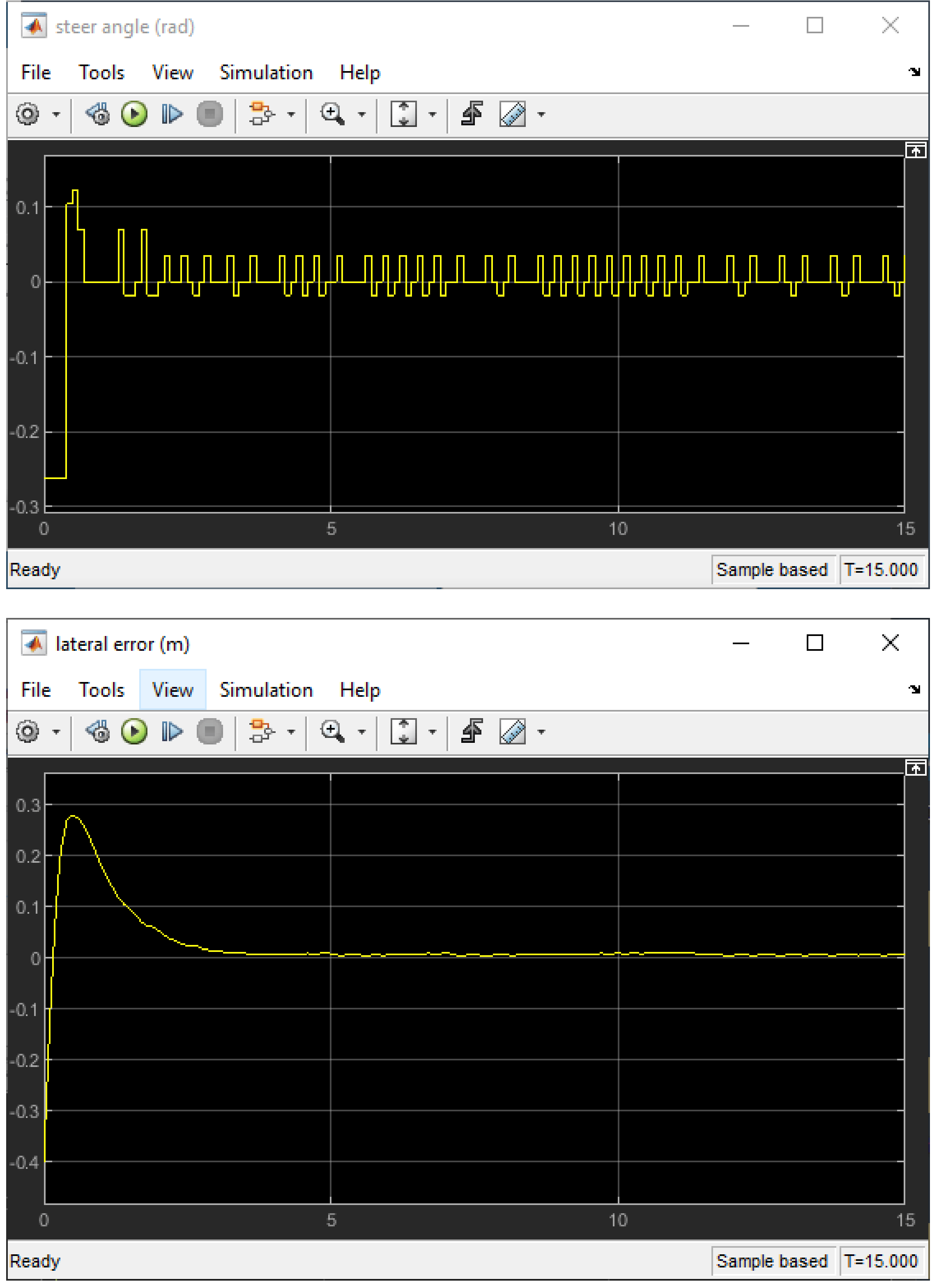
To validate the performance of the trained agent, uncomment the following two lines and simulate the agent within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

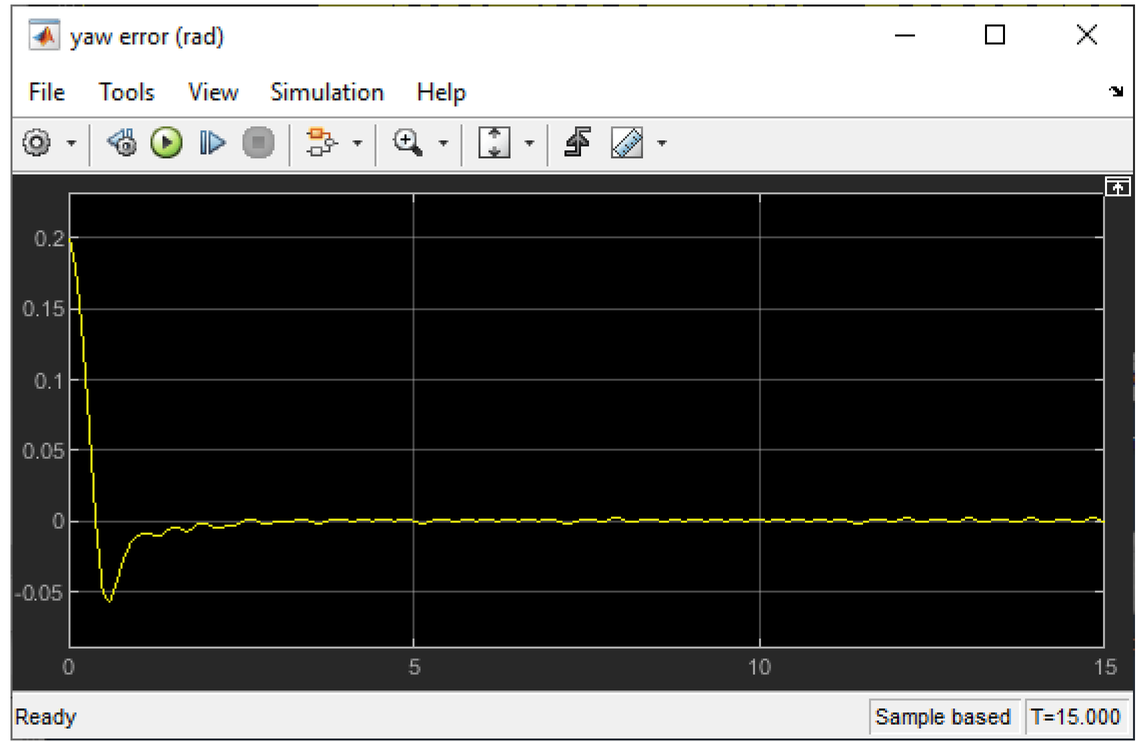
```
% simOptions = rlSimulationOptions(MaxSteps=maxsteps);
% experience = sim(env,agent,simOptions);
```

To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.2;
sim(mdl)
```

As shown below, the lateral error (middle plot) and relative yaw angle (bottom plot) are both driven to zero. The vehicle starts from off centerline (–0.4 m) and nonzero yaw angle error (0.2 rad). The LKA enables the ego car to travel along the centerline after 2.5 seconds. The steering angle (top plot) shows that the controller reaches steady state after 2 seconds.


steer angle (rad)
File Tools View Simulation Help
0.1
0
-0.1
-0.2
-0.3
0
5
10
15
Ready
Sample based
T=15.000
lateral error (m)
File Tools View Simulation Help
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
5
10
15
Ready
Sample based
T=15.000




**Local Function**

```
function in = localResetFcn(in)
% set initial lateral deviation and relative yaw angle to random values
in = setVariable(in,"e1_initial", 0.5*(-1+2*rand));
in = setVariable(in,"e2_initial", 0.1*(-1+2*rand));
end
```

## See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlDQNAgent` | `rlDQNAgentOptions` | `rlOptimizerOptions` | `rlTrainingOptions` | `rlSimulationOptions`

**Blocks**
RL Agent

## Related Examples

- “Train DQN Agent for Lane Keeping Assist” on page 5-226
- “Lane Keeping Assist with Lane Detection” (Model Predictive Control Toolbox)
- “Lane Keeping Assist System Using Model Predictive Control” (Model Predictive Control Toolbox)
- “Train AC Agent to Balance Cart-Pole System Using Parallel Computing” on page 5-166

- “Train Biped Robot to Walk Using Reinforcement Learning Agents” on page 5-267


## More About


- “Create Policies and Value Functions” on page 4-2
- “Deep Q-Network (DQN) Agents” on page 3-23
- “Train Reinforcement Learning Agents” on page 5-3
- “Train Agents Using Parallel Computing and GPUs” on page 5-8

# Train Biped Robot to Walk Using Reinforcement Learning Agents

This example shows how to train a biped robot to walk using either a deep deterministic policy gradient (DDPG) agent or a twin-delayed deep deterministic policy gradient (TD3) agent. In the example, you also compare the performance of these trained agents. The robot in this example is modeled in Simscape™ Multibody™.

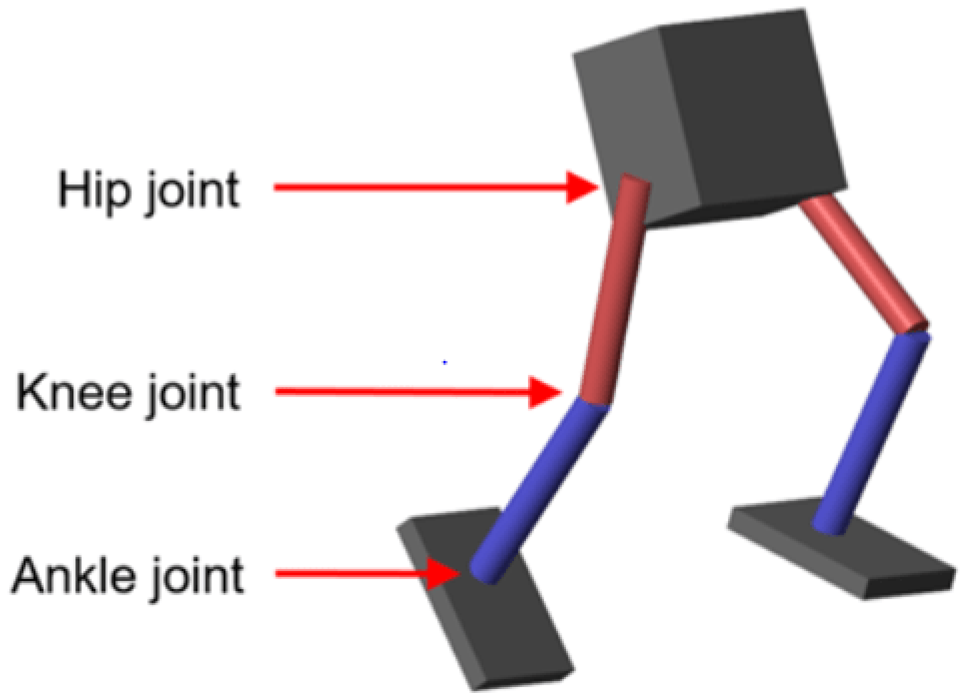
For more information on these agents, see "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40 and "Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents" on page 3-44.

For the purpose of comparison in this example, this example trains both agents on the biped robot environment with the same model parameters. The example also configures the agents to have the following settings in common.

- Initial condition strategy of the biped robot
- Network structure of actor and critic, inspired by [1]
- Options for actor and critic representations
- Training options (sample time, discount factor, mini-batch size, experience buffer length, exploration noise)

### Biped Robot Model

The reinforcement learning environment for this example is a biped robot. The training goal is to make the robot walk in a straight line using minimal control effort.

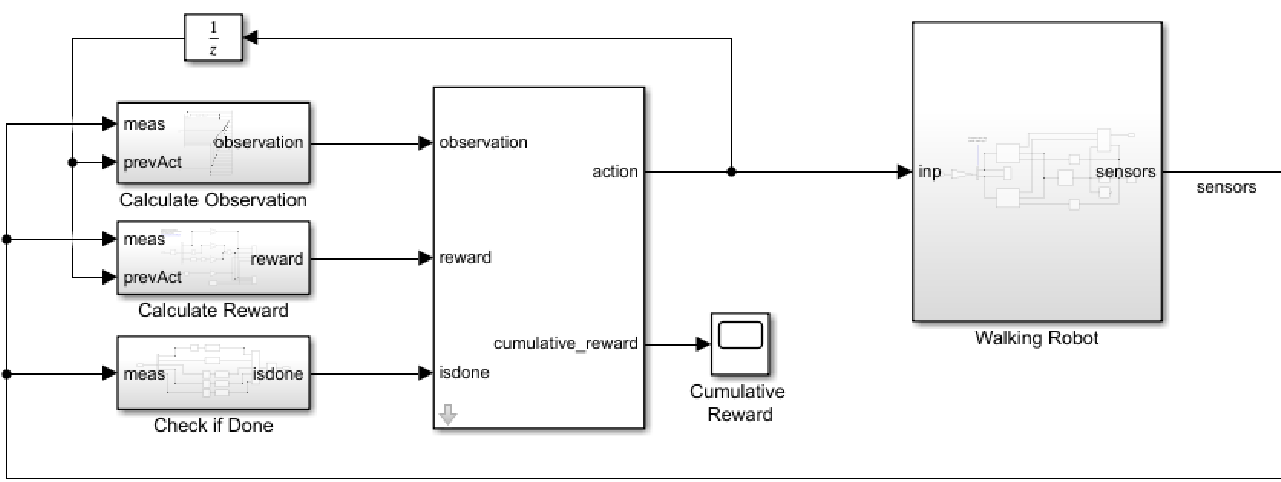



Load the parameters of the model into the MATLAB® workspace.

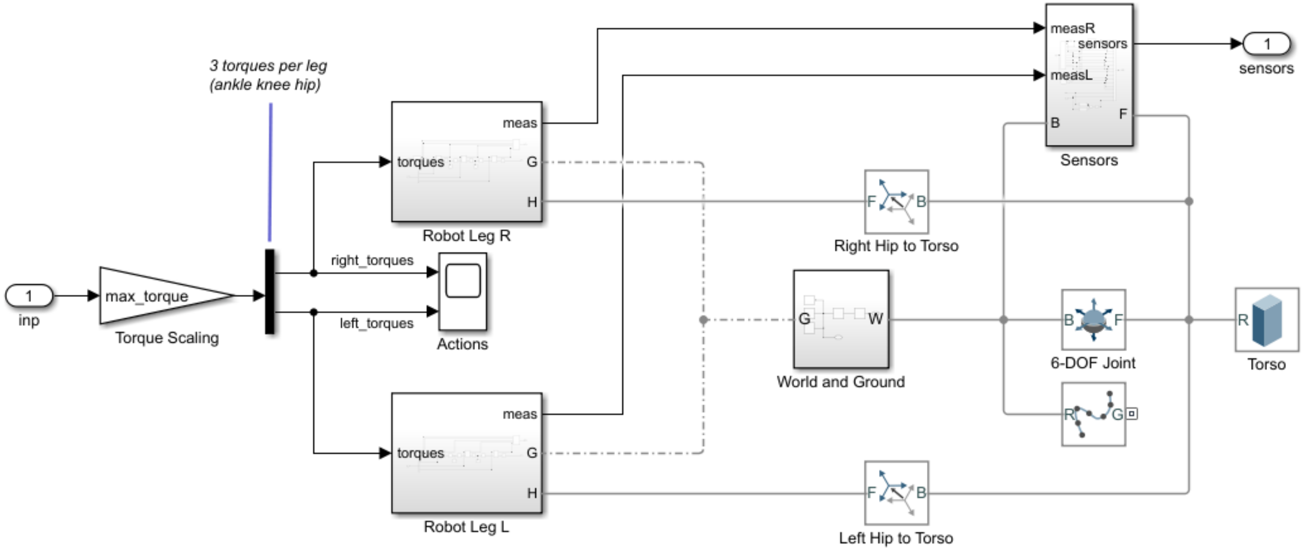
```
robotParametersRL
```

Open the Simulink model.

```
mdl = "rlWalkingBipedRobot";
open_system(mdl)
```




The robot is modeled using Simscape Multibody.




For this model:

- In the neutral 0 rad position, both of the legs are straight and the ankles are flat.
- The foot contact is modeled using the Spatial Contact Force (Simscape Multibody) block.
- The agent can control 3 individual joints (ankle, knee, and hip) on both legs of the robot by applying torque signals from `-3` to `3` N·m. The actual computed action signals are normalized between `-1` and `1`.

The environment provides the following 29 observations to the agent.

- Y (lateral) and Z (vertical) translations of the torso center of mass. The translation in the Z direction is normalized to a similar range as the other observations.
- X (forward), Y (lateral), and Z (vertical) translation velocities.
- Yaw, pitch, and roll angles of the torso.
- Yaw, pitch, and roll angular velocities of the torso.
- Angular positions and velocities of the three joints (ankle, knee, hip) on both legs.
- Action values from the previous time step.

The episode terminates if either of the following conditions occur.

- The robot torso center of mass is less than 0.1 m in the Z direction (the robot falls) or more than 1 m in the either Y direction (the robot moves too far to the side).
- The absolute value of either the roll, pitch, or yaw is greater than 0.7854 rad.

The following reward function $r_t$, which is provided at every time step is inspired by [2].

$$\boldsymbol{r_t} = v_x - 3y^2 - 50\hat{z}^2 + 25\frac{\mathrm{Ts}}{\mathrm{Tf}} - 0.02\sum_i {u_{t-1}^i}^2$$

Here:

- $v_x$ is the translation velocity in the X direction (forward toward goal) of the robot.
- $y$ is the lateral translation displacement of the robot from the target straight line trajectory.
- $\hat{z}$ is the normalized vertical translation displacement of the robot center of mass.
- $u_{t-1}^i$ is the torque from joint $i$ from the previous time step.
- Ts is the sample time of the environment.
- Tf is the final simulation time of the environment.

This reward function encourages the agent to move forward by providing a positive reward for positive forward velocity. It also encourages the agent to avoid episode termination by providing a constant reward ($25\frac{\mathrm{Ts}}{\mathrm{Tf}}$) at every time step. The other terms in the reward function are penalties for substantial changes in lateral and vertical translations, and for the use of excess control effort.

#### Create Environment Interface

Create the observation specification.

```
numObs = 29;
obsInfo = rlNumericSpec([numObs 1]);
obsInfo.Name = "observations";
```

Create the action specification.

```
numAct = 6;
actInfo = rlNumericSpec([numAct 1],LowerLimit=-1,UpperLimit=1);
actInfo.Name = "foot_torque";
```

Create the environment interface for the walking robot model.

```
blk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,blk,obsInfo,actInfo);
env.ResetFcn = @(in) walkerResetFcn(in, ...
    upper_leg_length/100, ...
    lower_leg_length/100, ...
    h/100);
```

### Select and Create Agent for Training

This example provides the option to train the robot either using either a DDPG or TD3 agent. To simulate the robot with the agent of your choice, set the `AgentSelection` flag accordingly.

```
AgentSelection = "DDPG";
switch AgentSelection
    case "DDPG"
        agent = createDDPGAgent(numObs,obsInfo,numAct,actInfo,Ts);
    case "TD3"
        agent = createTD3Agent(numObs,obsInfo,numAct,actInfo,Ts);
    otherwise
        disp("Assign AgentSelection to DDPG or TD3")
end
```

The `createDDPGAgent` and `createTD3Agent` helper functions perform the following actions.

- Create actor and critic networks.
- Specify options for actor and critic representations.
- Create actor and critic representations using created networks and specified options.
- Configure agent specific options.
- Create agent.

#### DDPG Agent

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor, and a parametrized Q-value function approximator to estimate the value of the policy. Use neural networks to model both the policy and the Q-value function. The actor and critic networks for this example are inspired by [1].

For details on how the DDPG agent is created, see the `createDDPGAgent` helper function. For information on configuring DDPG agent options, see `rlDDPGAgentOptions`.

For more information on creating a deep neural network value function representation, see “Create Policies and Value Functions” on page 4-2. For an example that creates neural networks for DDPG agents, see “Train DDPG Agent to Control Double Integrator System” on page 5-77.

#### TD3 Agent

The critic of a DDPG agent can overestimate the Q value. Since the agent uses the Q value to update its policy (actor), the resultant policy can be suboptimal and can accumulate training errors that can lead to divergent behavior. The TD3 algorithm is an extension of DDPG with improvements that make it more robust by preventing overestimation of Q values [3].

- Two critic networks — TD3 agents learn two critic networks independently and use the minimum value function estimate to update the actor (policy). Doing so prevents accumulation of error in subsequent steps and overestimation of Q values.
- Addition of target policy noise — Adding clipped noise to value functions smooths out Q function values over similar actions. Doing so prevents learning an incorrect sharp peak of noisy value estimate.
- Delayed policy and target updates — For a TD3 agent, delaying the actor network update allows more time for the Q function to reduce error (get closer to the required target) before updating the policy. Doing so prevents variance in value estimates and results in a higher quality policy update.

The structure of the actor and critic networks used for this agent are the same as the ones used for DDPG agent. For details on the creating the TD3 agent, see the `createTD3Agent` helper function. For information on configuring TD3 agent options, see `rlTD3AgentOptions`.

#### Specify Training Options and Train Agent

For this example, the training options for the DDPG and TD3 agents are the same.

- Run each training session for 2000 episodes with each episode lasting at most `maxSteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option).
- Terminate the training only when it reaches the maximum number of episodes (`maxEpisodes`). Doing so allows the comparison of the learning curves for multiple agents over the entire training session.

For more information and additional options, see `rlTrainingOptions`.

```
maxEpisodes = 2000;
maxSteps = floor(Tf/Ts);
trainOpts = rlTrainingOptions(...
    MaxEpisodes=maxEpisodes,...
    MaxStepsPerEpisode=maxSteps,...
    ScoreAveragingWindowLength=250,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="EpisodeCount",...
    StopTrainingValue=maxEpisodes,...
    SaveAgentCriteria="EpisodeCount",...
    SaveAgentValue=maxEpisodes);
```

To train the agent in parallel, specify the following training options. Training in parallel requires Parallel Computing Toolbox™. If you do not have Parallel Computing Toolbox software installed, set `UseParallel` to `false`.

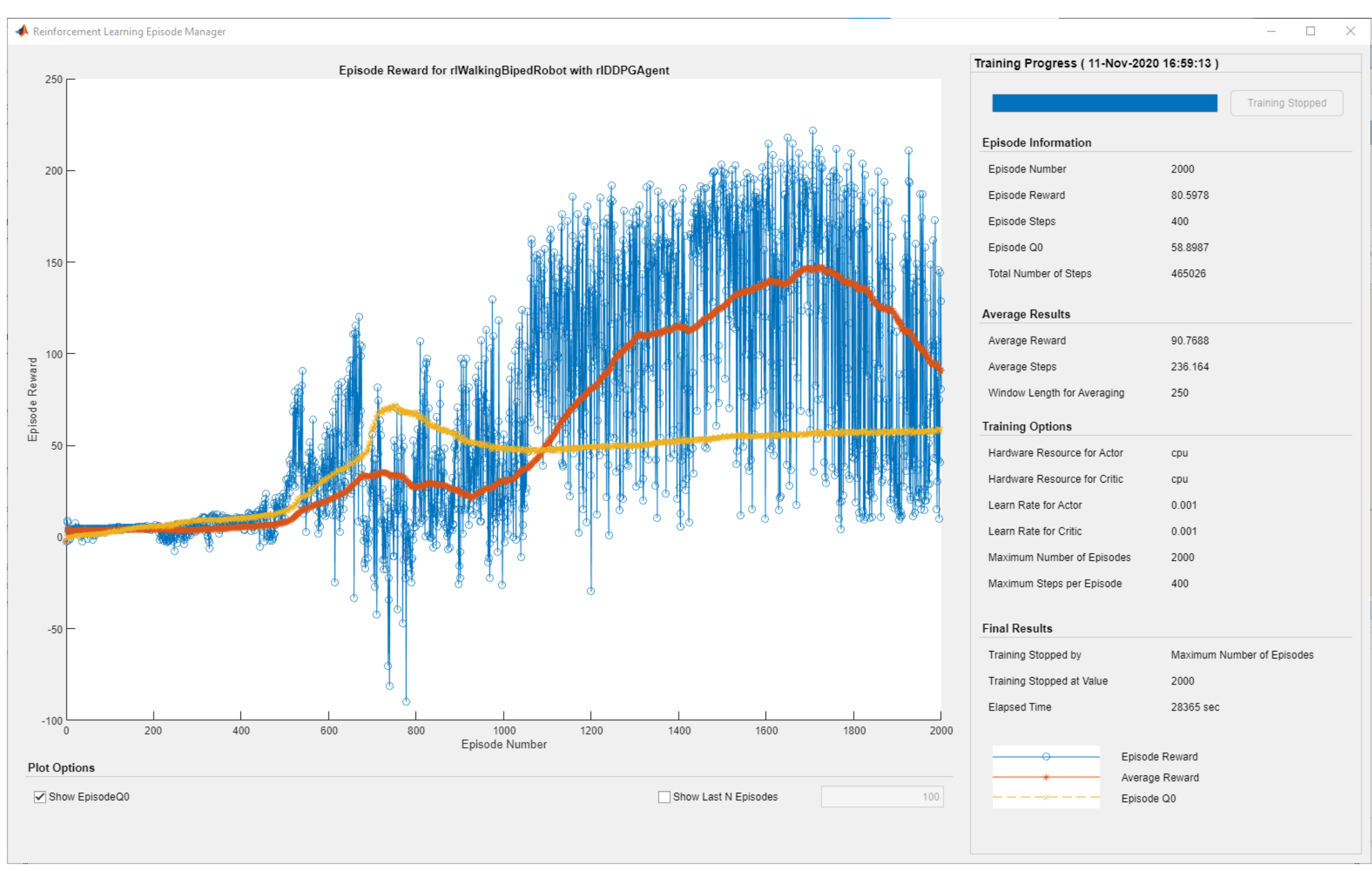
- Set the `UseParallel` option to `true`.
- Train the agent in parallel asynchronously.
- After every 32 steps, have each worker send experiences to the parallel pool client (the MATLAB® process which starts the training). DDPG and TD3 agents require workers to send experiences to the client.

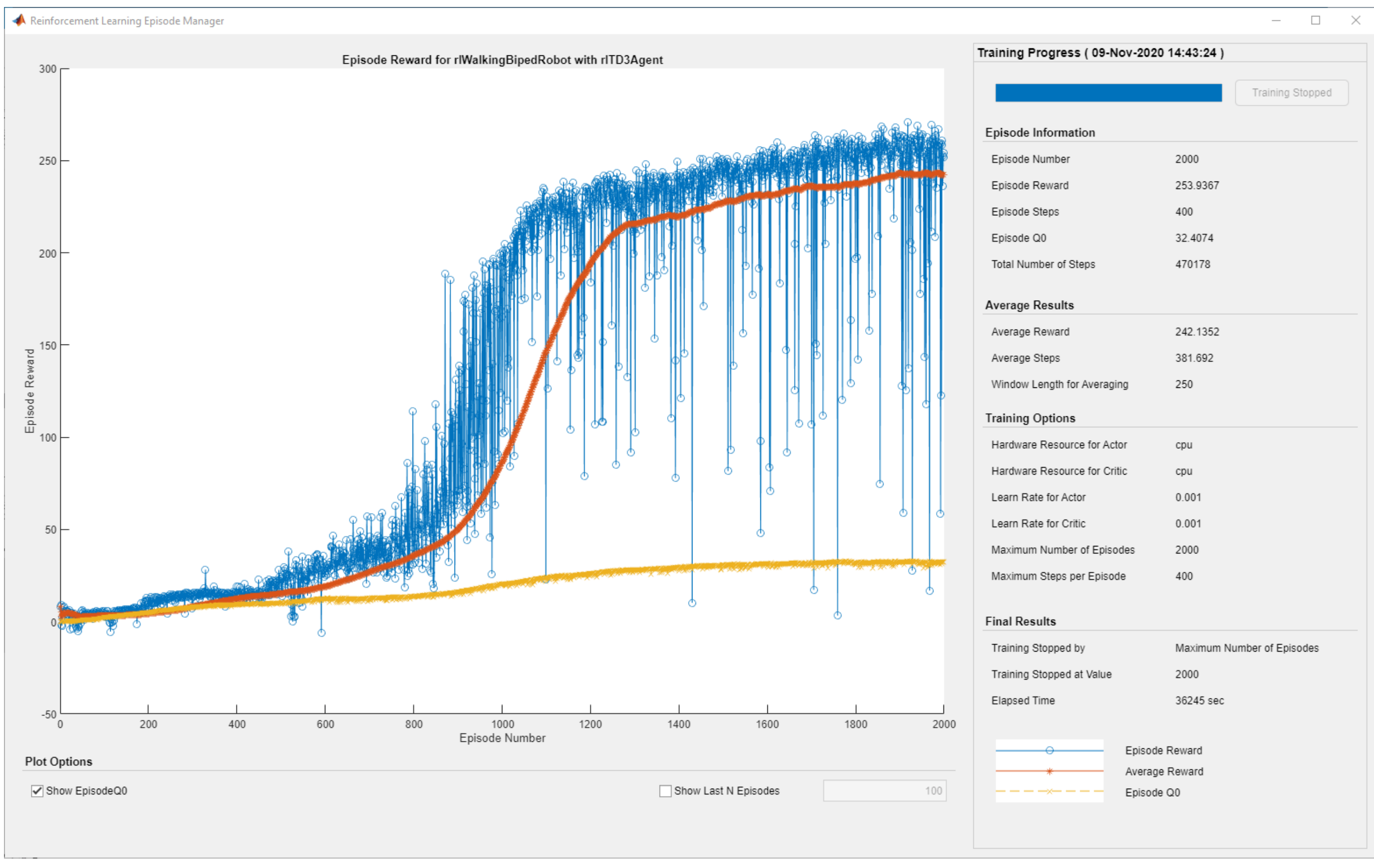
```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = "async";
```

```
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 32;
trainOpts.ParallelizationOptions.DataToSendFromWorkers = "Experiences";
```

Train the agent using the `train` function. This process is computationally intensive and takes several hours to complete for each agent. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`. Due to randomness in the parallel training, you can expect different training results from the plots that follow. The pretrained agents were trained in parallel using four workers.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load a pretrained agent for the selected agent type.
    if strcmp(AgentSelection,"DDPG")
       load("rlWalkingBipedRobotDDPG.mat","agent")
    else
       load("rlWalkingBipedRobotTD3.mat","agent")
    end
end
```






For the preceding example training curves, the average time per training step for the DDPG and TD3 agents are 0.11 and 0.12 seconds, respectively. The TD3 agent takes more training time per step because it updates two critic networks compared to the single critic used for DDPG.

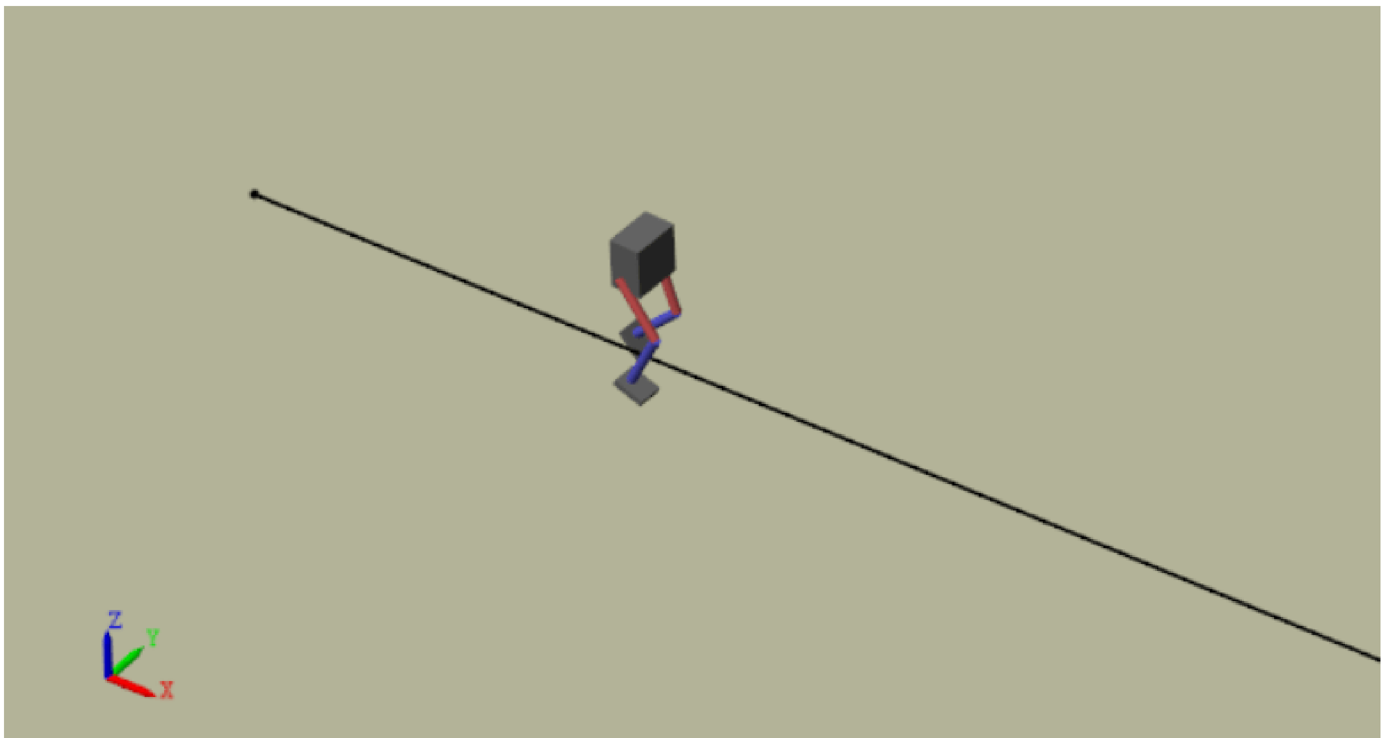
### Simulate Trained Agents

Fix the random generator seed for reproducibility.

```
rng(0)
```

To validate the performance of the trained agent, simulate it within the biped robot environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=maxSteps);
experience = sim(env,agent,simOptions);
```

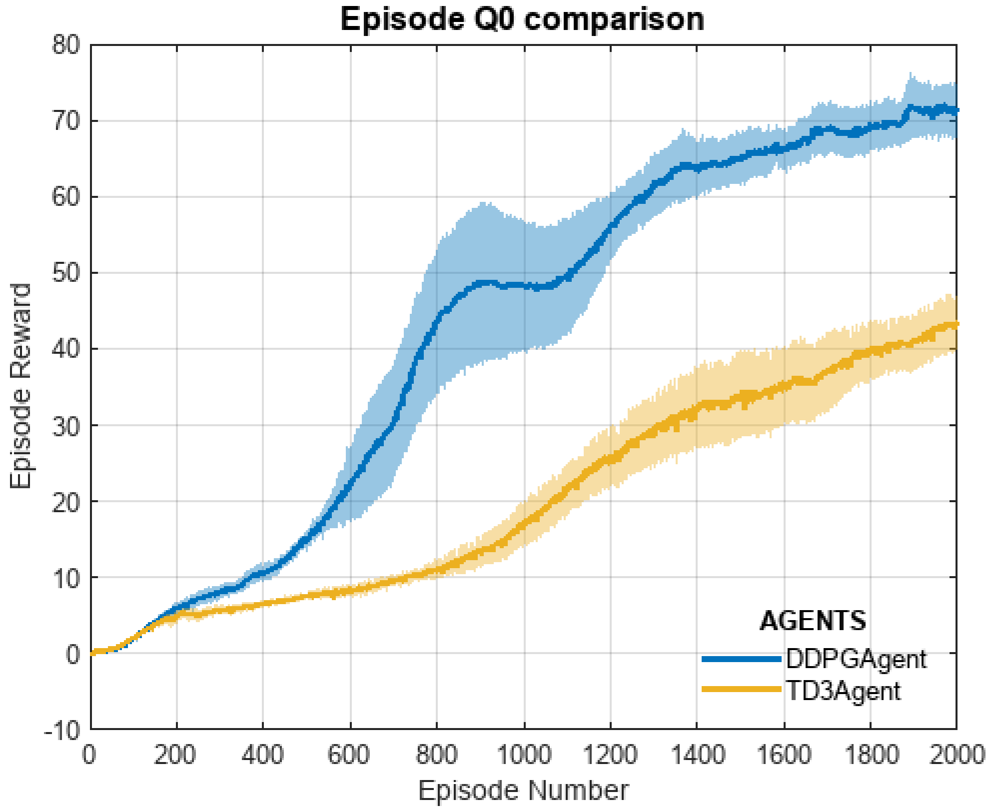


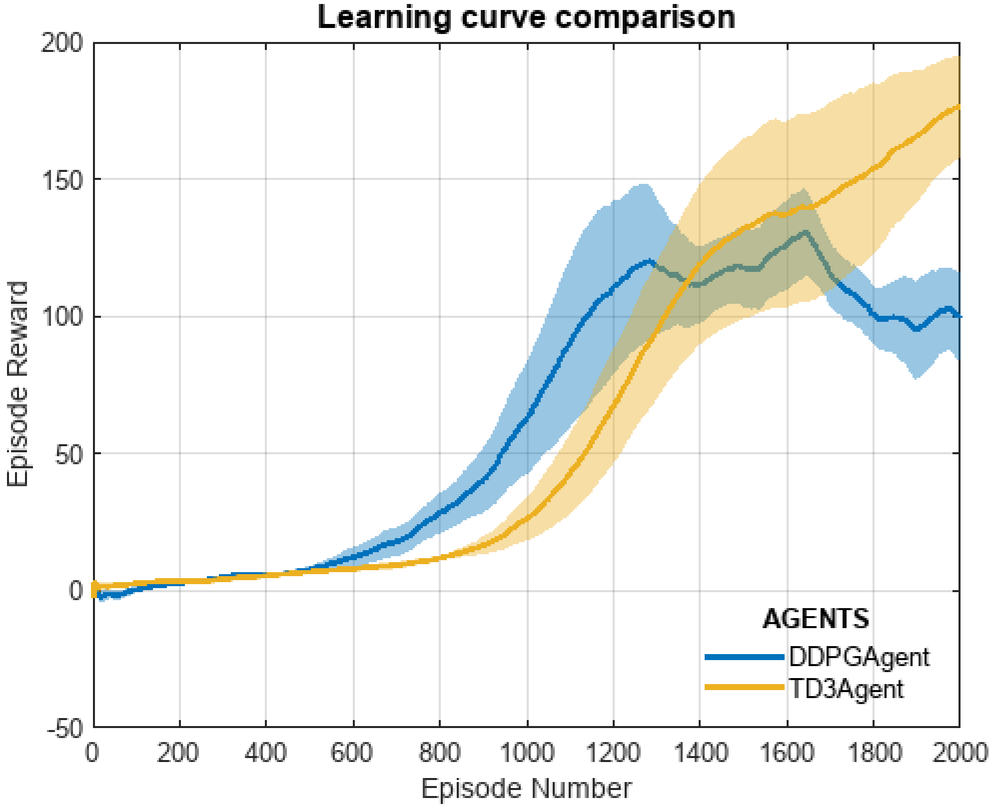

### Compare Agent Performance

For the following agent comparison, each agent was trained five times using a different random seed each time. Due to the random exploration noise and the randomness in the parallel training, the learning curve for each run is different. Since the training of agents for multiple runs takes several days to complete, this comparison uses pretrained agents.

For the DDPG and TD3 agents, plot the average and standard deviation of the episode reward (top plot) and the episode Q0 value (bottom plot). The episode Q0 value is the critic estimate of the discounted long-term reward at the start of each episode given the initial observation of the environment. For a well-designed critic, the episode Q0 value approaches the true discounted long-term reward.

```
comparePerformance("DDPGAgent","TD3Agent")
```


Episode Q0 comparison
Episode Reward
Episode Number
AGENTS
DDPGAgent
TD3Agent
80
70
60
50
40
30
20
10
0
-10
0
200
400
600
800
1000
1200
1400
1600
1800
2000




Based on the Learning curve comparison plot:

- The DDPG agent appears to pick up learning faster (around episode number 600 on average) but hits a local minimum. TD3 starts slower but eventually achieves higher rewards than DDPG as it avoids overestimation of Q values.
- The TD3 agent shows a steady improvement in its learning curve, which suggests improved stability when compared to the DDPG agent.

Based on the Episode Q0 comparison plot:

- For the TD3 agent, the critic estimate of the discounted long-term reward (for 2000 episodes) is lower compared to the DDPG agent. This difference is because the TD3 algorithm takes a conservative approach in updating its targets by using a minimum of two Q functions. This behavior is further enhanced because of delayed updates to the targets.
- Although the TD3 estimate for these 2000 episodes is low, the TD3 agent shows a steady increase in the episode Q0 values, unlike the DDPG agent.

In this example, the training was stopped at 2000 episodes. For a larger training period, the TD3 agent with its steady increase in estimates shows the potential to converge to the true discounted long-term reward.

For another example on how to train a humanoid robot to walk using a DDPG agent, see “Train Humanoid Walker” (Simscape Multibody). For an example on how to train a quadruped robot to walk using a DDPG agent, see “Quadruped Robot Locomotion Using DDPG Agent” on page 5-278.

### References

[1] Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous Control with Deep Reinforcement Learning." Preprint, submitted July 5, 2019. https://arxiv.org/abs/1509.02971.

[2] Heess, Nicolas, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, et al. "Emergence of Locomotion Behaviours in Rich Environments." Preprint, submitted July 10, 2017. https://arxiv.org/abs/1707.02286.

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

## See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlTD3Agent` | `rlTD3AgentOptions` | `rlQValueFunction` | `rlContinuousDeterministicActor` | `rlOptimizerOptions` | `rlTrainingOptions` | `rlSimulationOptions`

**Blocks**
RL Agent

## Related Examples

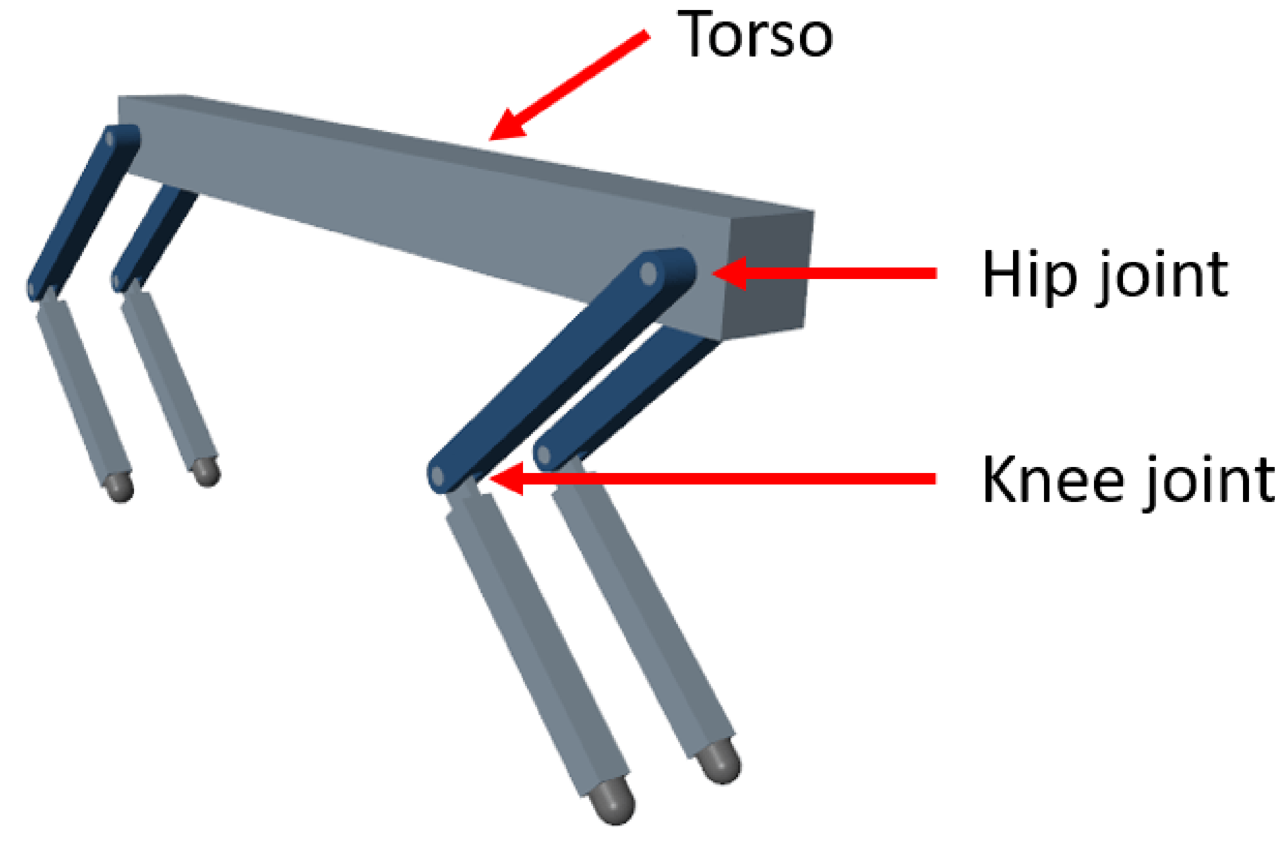
- “Train AC Agent to Balance Cart-Pole System Using Parallel Computing” on page 5-166
- “Train DQN Agent for Lane Keeping Assist Using Parallel Computing” on page 5-257
- “Quadruped Robot Locomotion Using DDPG Agent” on page 5-278

## More About

- “Define Reward Signals” on page 2-14
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents” on page 3-44
- “Train Agents Using Parallel Computing and GPUs” on page 5-8

# Quadruped Robot Locomotion Using DDPG Agent

This example shows how to train a quadruped robot to walk using a deep deterministic policy gradient (DDPG) agent. The robot in this example is modeled using Simscape™ Multibody™. For more information on DDPG agents, see “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40.




Load the necessary parameters into the base workspace in MATLAB®.

```
initializeRobotParameters
```

### Quadruped Robot Model

The environment for this example is a quadruped robot, and the training goal is to make the robot walk in a straight line using minimal control effort.

Open the model.

```
mdl = "rlQuadrupedRobot";
open_system(mdl)
```

The robot is modeled using Simscape Multibody with its main structural components consisting of four legs and a torso. The legs are connected to the torso through revolute joints that enable rotation of the legs with respect to the torso. The joints are actuated by torque signals provided by the RL Agent.

**Observations**

The robot environment provides 44 observations to the agent, each normalized between –1 and 1. These observations are:

- Y (vertical) and Y (lateral) position of the torso center of mass
- Quaternion representing the orientation of the torso
- X (forward), Y (vertical), and Z (lateral) velocities of the torso at the center of mass
- Roll, pitch, and yaw rates of the torso
- Angular positions and velocities of the hip and knee joints for each leg
- Normal and friction force due to ground contact for each leg
- Action values (torque for each joint) from the previous time step

For all four legs, the initial values for the hip and knee joint angles are set to –0.8234 and 1.6468 radians, respectively. The neutral positions of the joints are set at 0 radian. The legs are in neutral position when they are stretched to their maximum and are aligned perpendicularly to the ground.

**Actions**

The agent generates eight actions normalized between –1 and 1. After multiplying with a scaling factor, these correspond to the eight joint torque signals for the revolute joints. The overall joint torque bounds are +/– 10 N·m for each joint.

#### Reward

The following reward is provided to the agent at each time step during training. This reward function encourages the agent to move forward by providing a positive reward for positive forward velocity. It also encourages the agent to avoid early termination by providing a constant reward ($25T_s/T_f$) at each time step. The remaining terms in the reward function are penalties that discourage unwanted states, such as large deviations from the desired height and orientation or the use of excessive joint torques.

$$r_t = v_x + 25\frac{T_s}{T_f} - 50\hat{y}^2 - 20\theta^2 - 0.02\sum_i {u_{t-1}^i}^2$$

where

- $v_x$ is the velocity of the torso's center of mass in the x-direction.
- $T_s$ and $T_f$ are the sample time and final simulation time of the environment, respectively.
- $\hat{y}$ is the scaled height error of the torso's center of mass from the desired height of 0.75 m.
- $\theta$ is the pitch angle of the torso.
- $u_{t-1}^i$ is the action value for joint $i$ from the previous time step.

#### Episode Termination

During training or simulation, the episode terminates if any of the following situations occur.

- The height of the torso center of mass from the ground is below 0.5 m (fallen).
- The head or tail of the torso is below the ground.
- Any knee joint is below the ground.
- Roll, pitch, or yaw angles are outside bounds (+/– 0.1745, +/– 0.1745, and +/– 0.3491 radians, respectively).

#### Create Environment Interface

Specify the parameters for the observation set.

```
numObs = 44;
obsInfo = rlNumericSpec([numObs 1]);
obsInfo.Name = "observations";
```

Specify the parameters for the action set.

```
numAct = 8;
actInfo = rlNumericSpec([numAct 1],LowerLimit=-1,UpperLimit= 1);
actInfo.Name = "torque";
```

Create the environment using the reinforcement learning model.

```
blk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,blk,obsInfo,actInfo);
```

During training, the reset function introduces random deviations into the initial joint angles and angular velocities.

```
env.ResetFcn = @quadrupedResetFcn;
```

### Create DDPG agent

The DDPG agent approximates the long-term reward given observations and actions by using a critic value function representation. The agent also decides which action to take given the observations, using an actor representation. The actor and critic networks for this example are inspired by [2].

For more information on creating a deep neural network value function representation, see “Create Policies and Value Functions” on page 4-2. For an example that creates neural networks for DDPG agents, see “Train DDPG Agent to Control Double Integrator System” on page 5-77.

Set the random seed.

```
rng(0)
```

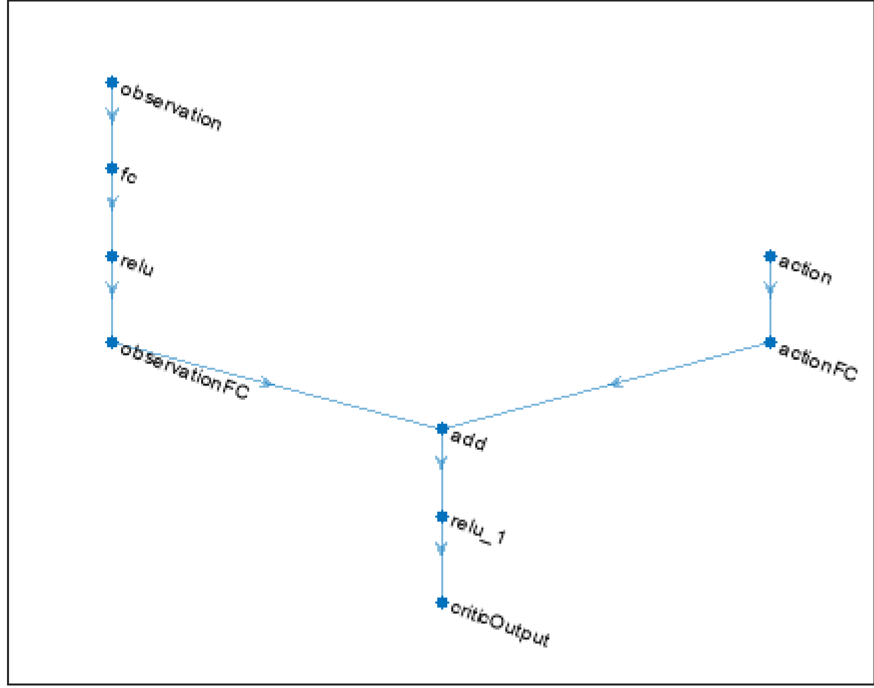
Create the networks in the MATLAB workspace using the `createNetworks` helper function.

```
createNetworks
```

You can also create your actor and critic networks interactively using the Deep Network Designer app.

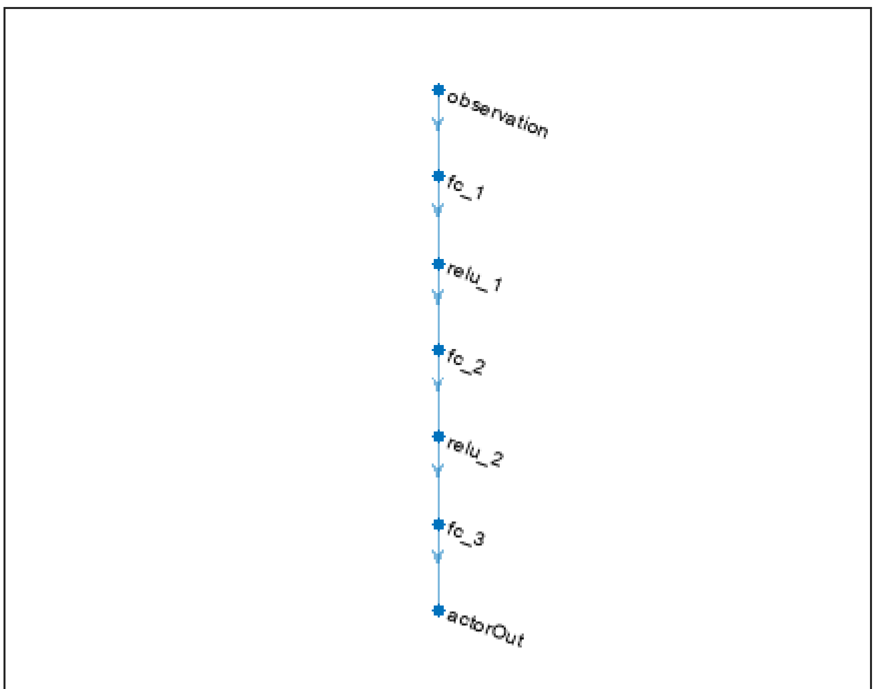
View the critic network configuration.

```
plot(criticNetwork)
```




View the actor network configuration.

```
plot(actorNetwork)
```




Specify the agent options using `rlDDPGAgentOptions`.

```
agentOptions = rlDDPGAgentOptions();
agentOptions.SampleTime = Ts;
agentOptions.DiscountFactor = 0.99;
agentOptions.MiniBatchSize = 128;
agentOptions.ExperienceBufferLength = 1e6;
agentOptions.TargetSmoothFactor = 1e-3;
agentOptions.NoiseOptions.MeanAttractionConstant = 0.15;
agentOptions.NoiseOptions.Variance = 0.1;
```

Specify optimizer options for the actor and critic.

```
agentOptions.ActorOptimizerOptions.Algorithm = "adam";
agentOptions.ActorOptimizerOptions.LearnRate = 1e-4;
agentOptions.ActorOptimizerOptions.GradientThreshold = 1;
agentOptions.ActorOptimizerOptions.L2RegularizationFactor = 1e-5;

agentOptions.CriticOptimizerOptions.Algorithm = "adam";
agentOptions.CriticOptimizerOptions.LearnRate = 1e-3;
agentOptions.CriticOptimizerOptions.GradientThreshold = 1;
agentOptions.CriticOptimizerOptions.L2RegularizationFactor = 2e-4;
```

Create the `rlDDPGAgent` object for the agent.

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

### Specify Training Options

To train the agent, first specify the following training options:

- Run each training episode for at most 10,000 episodes, with each episode lasting at most `maxSteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option).
- Stop training when the agent receives an average cumulative reward greater than 190 over 250 consecutive episodes.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=10000,...
    MaxStepsPerEpisode=floor(Tf/Ts),...
    ScoreAveragingWindowLength=250,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=210);
```

To train the agent in parallel, specify the following training options. Training in parallel requires Parallel Computing Toolbox™ software. If you do not have Parallel Computing Toolbox™ software installed, set `UseParallel` to `false`.

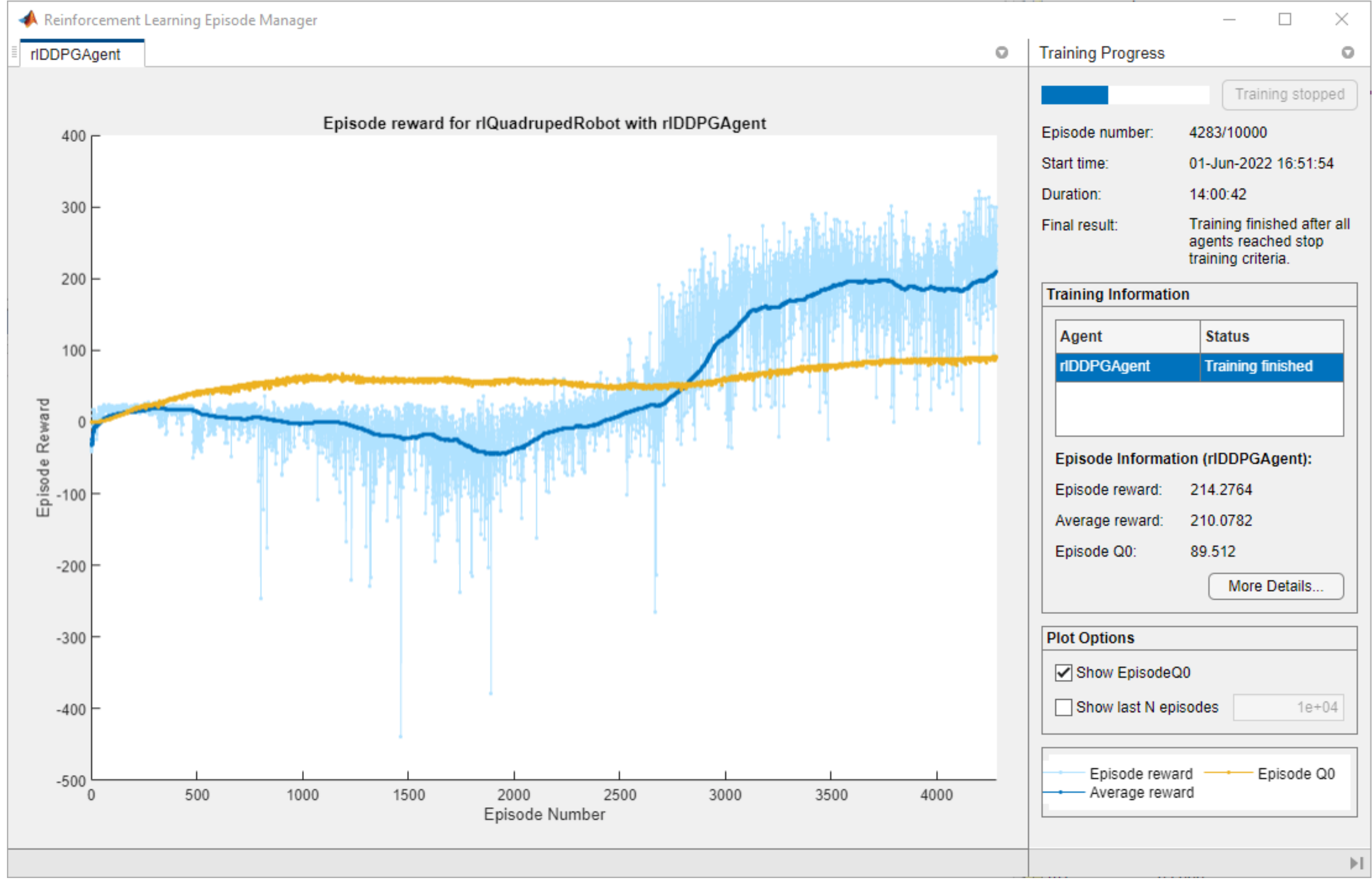
- Set the `UseParallel` option to `true`.
- Train the agent in parallel asynchronously.
- DDPG agents require workers to send `"Experiences"` to the host.

```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = "async";
trainOpts.ParallelizationOptions.DataToSendFromWorkers = "Experiences";
```

### Train Agent

Train the agent using the `train` function. Due to the complexity of the robot model, this process is computationally intensive and takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`. Due to the randomness of parallel training, you can expect different training results from the plot below.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load pretrained agent parameters for the example.
    load("rlQuadrupedAgentParams.mat","params")
    setLearnableParameters(agent, params);
end
```



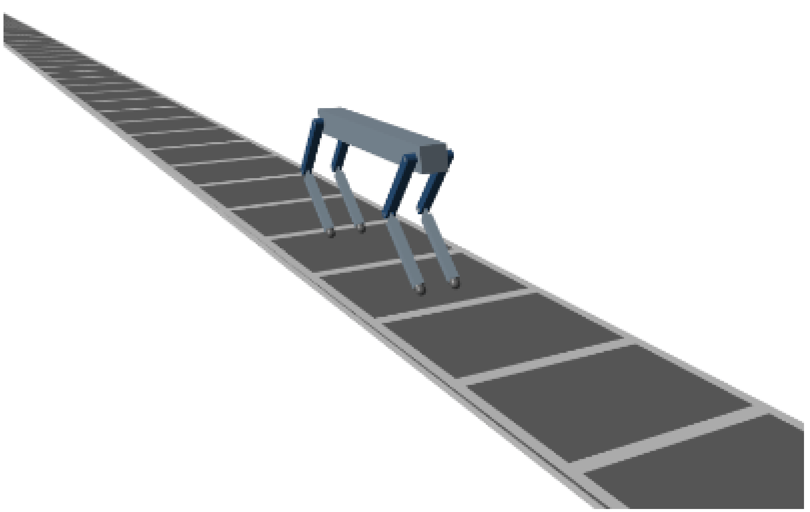

### Simulate Trained Agent

Fix the random generator seed for reproducibility.

```
rng(0)
```

To validate the performance of the trained agent, simulate it within the robot environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=floor(Tf/Ts));
experience = sim(env,agent,simOptions);
```



#### References

[1] Heess, Nicolas, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, et al. 'Emergence of Locomotion Behaviours in Rich Environments'. *ArXiv:1707.02286 [Cs]*, 10 July 2017. https://arxiv.org/abs/1707.02286.

[2] Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 'Continuous Control with Deep Reinforcement Learning'. *ArXiv:1509.02971 [Cs, Stat]*, 5 July 2019. https://arxiv.org/abs/1509.02971.

## See Also

#### Functions
`train` | `sim` | `rlSimulinkEnv`

#### Objects
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlQValueFunction` | `rlContinuousDeterministicActor` | `rlTrainingOptions` | `rlSimulationOptions` | `rlOptimizerOptions`

#### Blocks
RL Agent

## Related Examples

- "Train Biped Robot to Walk Using Reinforcement Learning Agents" on page 5-267

## More About

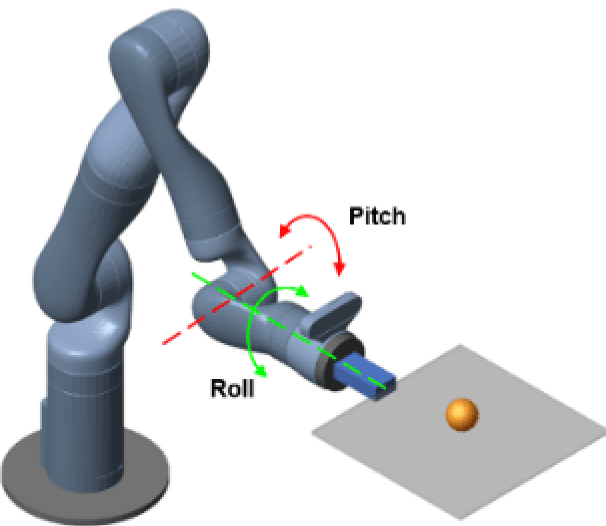
- "Define Reward Signals" on page 2-14
- "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40
- "Train Reinforcement Learning Agents" on page 5-3

# Train SAC Agent for Ball Balance Control

This example shows how to train a soft actor-critic (SAC) reinforcement learning agent to control a robot arm for a ball-balancing task.

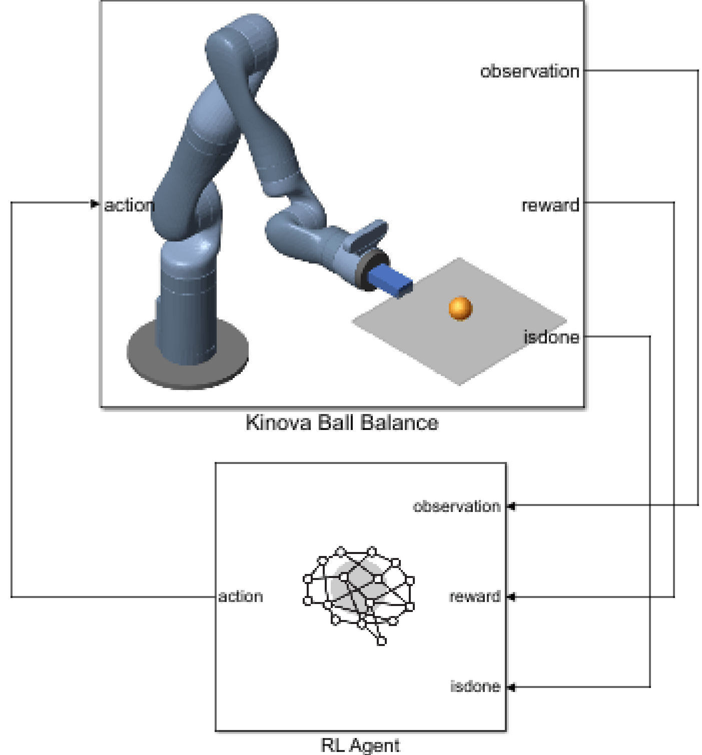
### Introduction

The robot arm in this example is a Kinova Gen3 robot, which is a seven degree-of-freedom (DOF) manipulator. The arm is tasked to balance a ping pong ball at the center of a flat surface (plate) attached to the robot gripper. Only the final two joints are actuated and contribute to motion in the pitch and roll axes as shown in the following figure. The remaining joints are fixed and do not contribute to motion.



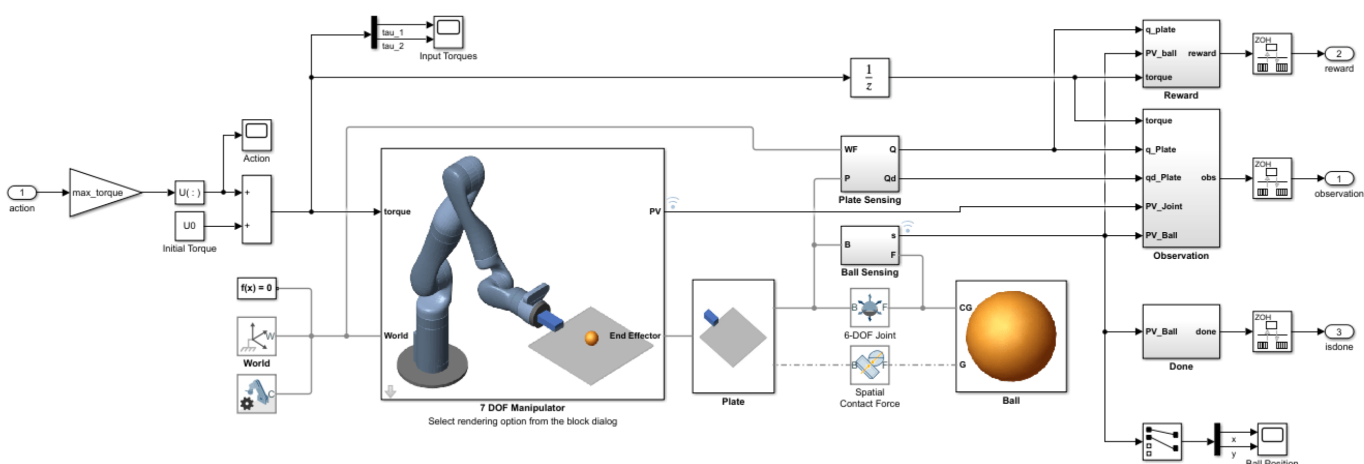

Open the Simulink® model to view the system. The model contains a Kinova Ball Balance subsystem connected to an RL Agent block. The agent applies an action to the robot subsystem and receives the resulting observation, reward, and is-done signals.

```
open_system("rlKinovaBallBalance")
```




View to the Kinova Ball Balance subsystem.

```
open_system("rlKinovaBallBalance/Kinova Ball Balance")
```




In this model:

- The physical components of the system (manipulator, ball, and plate) are modeled using Simscape™ Multibody™ components.

- The plate is constrained to the end effector of the manipulator.
- The ball has six degrees of freedom and can move freely in space.
- Contact forces between the ball and plate are modeled using the Spatial Contact Force block.
- Control inputs to the manipulator are the torque signals for the actuated joints.

If you have the Robotics System Toolbox Robot Library Data support package, you can view a 3-D animation of the manipulator in the Mechanics Explorer. To do so, open the 7 DOF Manipulator subsystem and set its **Visualization** parameter to `3D Mesh`. If you do have the support package installed, set the **Visualization** parameter to `None`. To download and install the support package, use the Add-On Explorer. For more information see "Get and Manage Add-Ons".

Create the parameters for the example by running the `kinova_params` script included with this example. When you have the Robotics System Toolbox Robot Library Data support package installed, this script also adds the necessary mesh files to the MATLAB® path.

```
kinova_params
```

### Define Environment

To train a reinforcement learning agent, you must define the environment with which it will interact. For the ball balancing environment:

- The observations are represented by a 22 element vector that contains information about the positions (sine and cosine of joint angles) and velocities (joint angle derivatives) of the two actuated joints, positions (x and y distances from plate center) and velocities (x and y derivatives) of the ball, orientation (quaternions) and velocities (quaternion derivatives) of the plate, joint torques from the last time step, ball radius, and mass.
- The actions are normalized joint torque values.
- The sample time is $T_s = 0.01s$, and the simulation time is $T_f = 10s$.
- The simulation terminates when the ball falls off the plate.
- The reward $r_t$ at time step $t$ is given by:

$$r_t = r_{\text{ball}} + r_{\text{plate}} + r_{\text{action}}$$

$$r_{\text{ball}} = e^{-0.001\left(x^2 + y^2\right)}$$

$$r_{\text{plate}} = -0.1\left(\phi^2 + \theta^2 + \psi^2\right)$$

$$r_{\text{action}} = -0.05\left(\tau_1^2 + \tau_2^2\right)$$

Here, $r_{\text{ball}}$ is a reward for the ball moving closer to the center of the plate, $r_{\text{plate}}$ is a penalty for plate orientation, and $r_{\text{action}}$ is a penalty for control effort. $\phi$, $\theta$, and $\psi$ are the respective roll, pitch, and yaw angles of the plate in radians. $\tau_1$ and $\tau_2$ are the joint torques.

Create the observation and action specifications for the environment using continuous observation and action spaces.

```
numObs = 22;  % Number of dimension of the observation space
numAct = 2;   % Number of dimension of the action space

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

```
actInfo = rlNumericSpec([numAct 1]);
actInfo.LowerLimit = -1;
actInfo.UpperLimit = 1;
```

Create the Simulink environment interface using the observation and action specifications. For more information on creating Simulink environments, see `rlSimulinkEnv`.

```
mdl = "rlKinovaBallBalance";
blk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,blk,obsInfo,actInfo);
```

Specify a reset function for the environment using the `ResetFcn` parameter.

```
env.ResetFcn = @kinovaResetFcn;
```

This reset function (provided at the end of this example) randomly initializes the initial x and y positions of the ball with respect to the center of the plate. For more robust training, you can also randomize other parameters inside the reset function, such as the mass and radius of the ball.

Specify the sample time `Ts` and simulation time `Tf`.

```
Ts = 0.01;
Tf = 10;
```

**Create Agent**

The agent in this example is a soft actor-critic (SAC) agent. SAC agents use one or two parametrized Q-value function approximators to estimate the value of the policy. A Q-value function critic takes the current observation and an action as inputs and returns a single scalar as output (the estimated discounted cumulative long-term reward for which receives the action from the state corresponding to the current observation, and following the policy thereafter). For more information on SAC agents, see “Soft Actor-Critic (SAC) Agents” on page 3-35.

The SAC agent in this example uses two critics. To model the parametrized Q-value functions within the critics, use a neural network with two input layers (one for the observation channel, as specified by `obsInfo`, and the other for the action channel, as specified by `actInfo`) and one output layer (which returns the scalar value).

Define each network path as an array of layer objects. Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

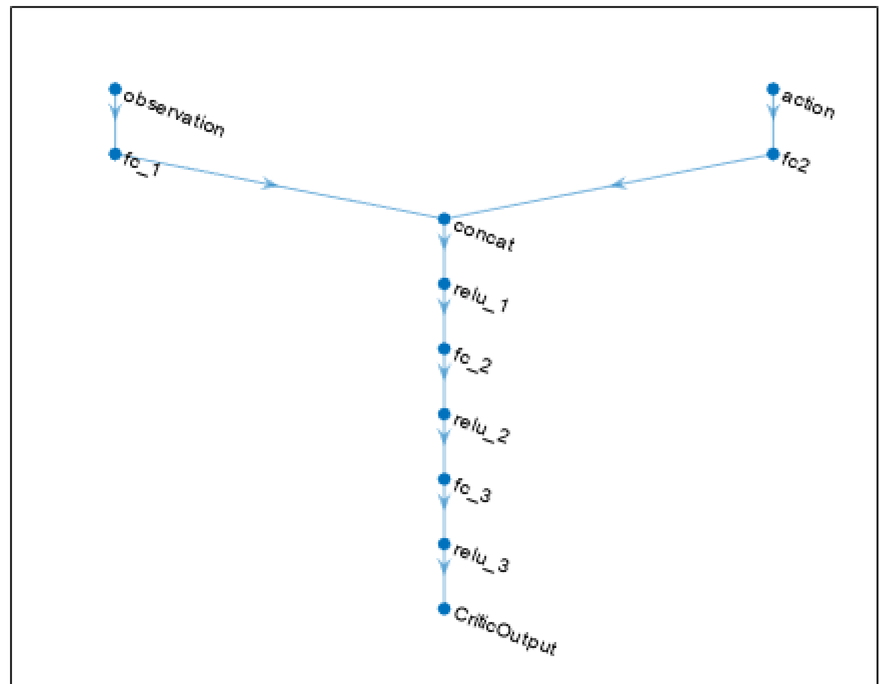
For more information on creating deep neural networks for reinforcement learning agents, see “Create Policies and Value Functions” on page 4-2.

```
% Set the random seed for reproducibility.
rng(0)

% Define the network layers.
cnet = [
    featureInputLayer(numObs,Name="observation")
    fullyConnectedLayer(128)
    concatenationLayer(1,2,Name="concat")
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(32)
```

```
    reluLayer
    fullyConnectedLayer(1,Name="CriticOutput")];
actionPath = [
    featureInputLayer(numAct,Name="action")
    fullyConnectedLayer(128,Name="fc2")];

% Connect the layers.
criticNetwork = layerGraph(cnet);
criticNetwork = addLayers(criticNetwork, actionPath);
criticNetwork = connectLayers(criticNetwork,"fc2","concat/in2");
```

View the critic neural network.

```
plot(criticNetwork)
```




When using two critics, a SAC agent requires them to have different initial parameters. Create and initialize two `dlnetwork` objects.

```
criticdlnet = dlnetwork(criticNetwork,'Initialize',false);
criticdlnet1 = initialize(criticdlnet);
criticdlnet2 = initialize(criticdlnet);
```

Create the critic functions using `rlQValueFunction`.

```
critic1 = rlQValueFunction(criticdlnet1,obsInfo,actInfo, ...
    ObservationInputNames="observation");
critic2 = rlQValueFunction(criticdlnet2,obsInfo,actInfo, ...
    ObservationInputNames="observation");
```

Soft Actor-critic agents use a parametrized stochastic policy over a continuous action space, which is implemented by a continuous Gaussian actor.

This actor takes an observation as input and returns as output a random action sampled from a Gaussian probability distribution.

To approximate the mean values and standard deviations of the Gaussian distribution, you must use a neural network with two output layers, each having as many elements as the dimension of the action space. One output layer must return a vector containing the mean values for each action dimension. The other must return a vector containing the standard deviation for each action dimension.

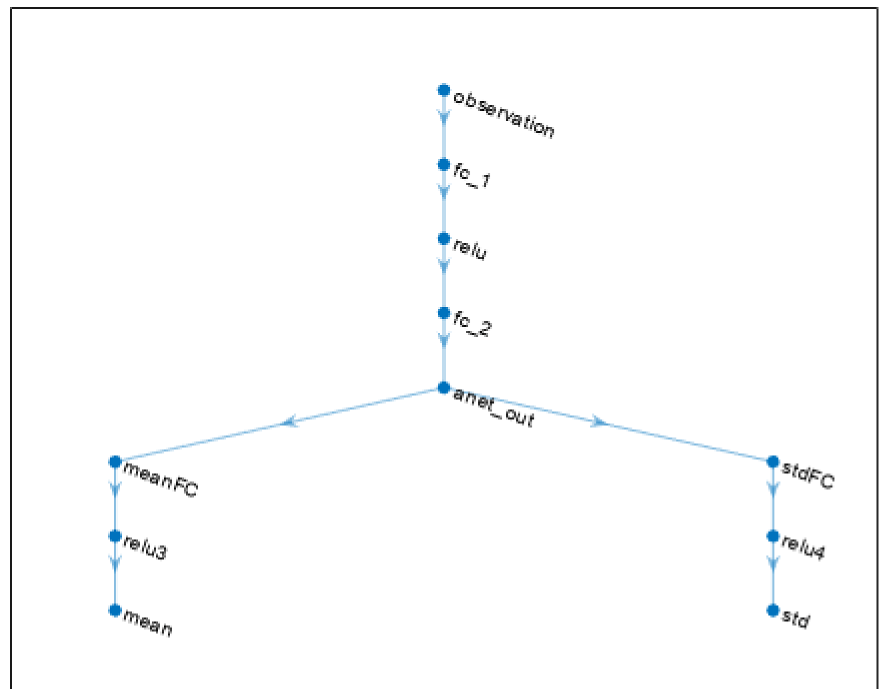
Since standard deviations must be nonnegative, use a softplus or ReLU layer to enforce nonnegativity. The SAC agent automatically reads the action range from the `UpperLimit` and `LowerLimit` properties of `actInfo` (which is used to create the actor), and then internally scales the distribution and bounds the action. Therefore, do not add a `tanhLayer` as the last nonlinear layer in the mean output path.

Define each network path as an array of layer objects, and assign names to the input and output layers of each path.

```
% Create the actor network layers.
commonPath = [
    featureInputLayer(numObs,Name="observation")
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer(Name="anet_out")];
meanPath = [
    fullyConnectedLayer(32,Name="meanFC")
    reluLayer(Name="relu3")
    fullyConnectedLayer(numAct,Name="mean")];
stdPath = [
    fullyConnectedLayer(numAct,Name="stdFC")
    reluLayer(Name="relu4")
    softplusLayer(Name="std")];

% Connect the layers.
actorNetwork = layerGraph(commonPath);
actorNetwork = addLayers(actorNetwork,meanPath);
actorNetwork = addLayers(actorNetwork,stdPath);
actorNetwork = connectLayers(actorNetwork,"anet_out","meanFC/in");
actorNetwork = connectLayers(actorNetwork,"anet_out","stdFC/in");
```

View the actor neural network.

```
plot(actorNetwork)
```




Create the actor function using `rlContinuousGaussianActor`.

```
actordlnet = dlnetwork(actorNetwork);
actor = rlContinuousGaussianActor(actordlnet, obsInfo, actInfo, ...
    ObservationInputNames="observation", ...
    ActionMeanOutputNames="mean", ...
    ActionStandardDeviationOutputNames="std");
```

The SAC agent in this example trains from an experience buffer of maximum capacity 1e6 by randomly selecting mini-batches of size 128. The discount factor of 0.99 is close to 1 and therefore favors long term reward with respect to a smaller value. For a full list of SAC hyperparameters and their descriptions, see `rlSACAgentOptions`.

Specify the agent hyperparameters for training.

```
agentOpts = rlSACAgentOptions( ...
    SampleTime=Ts, ...
    TargetSmoothFactor=1e-3, ...
    ExperienceBufferLength=1e6, ...
    MiniBatchSize=128, ...
    NumWarmStartSteps=1000, ...
    DiscountFactor=0.99);
```

For this example the actor and critic neural networks are updated using the Adam algorithm with a learn rate of 1e-4 and gradient threshold of 1. Specify the optimizer parameters.

```
agentOpts.ActorOptimizerOptions.Algorithm = "adam";
agentOpts.ActorOptimizerOptions.LearnRate = 1e-4;
```

```
agentOpts.ActorOptimizerOptions.GradientThreshold = 1;

for ct = 1:2
    agentOpts.CriticOptimizerOptions(ct).Algorithm = "adam";
    agentOpts.CriticOptimizerOptions(ct).LearnRate = 1e-4;
    agentOpts.CriticOptimizerOptions(ct).GradientThreshold = 1;
end
```

Create the SAC agent.

```
agent = rlSACAgent(actor,[critic1,critic2],agentOpts);
```

#### Train Agent

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

- Run each training for at most 5000 episodes, with each episode lasting at most `floor(Tf/Ts)` time steps.
- Stop training when the agent receives an average cumulative reward greater than 675 over 100 consecutive episodes.
- To speed up training set the `UseParallel` option to `true`, which requires Parallel Computing Toolbox™ software. If you do not have the software installed set the option to `false`.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=6000, ...
    MaxStepsPerEpisode=floor(Tf/Ts), ...
    ScoreAveragingWindowLength=100, ...
    Plots="training-progress", ...
    StopTrainingCriteria="AverageReward", ...
    StopTrainingValue=675, ...
    UseParallel=false);
```

Specify a visualization flag `doViz` that shows animation of the ball in a MATLAB figure. For parallel training, disable all visualization.

```
if trainOpts.UseParallel
    % Disable visualization in Simscape Mechanics Explorer
    set_param(mdl, SimMechanicsOpenEditorOnUpdate="off");
    set_param(mdl+"/Kinova Ball Balance/7 DOF Manipulator", ...
        "VChoice", "None");
    % Disable animation in MATLAB figure
    doViz = false;
    save_system(mdl);
else
    % Enable visualization in Simscape Mechanics Explorer
    set_param(mdl, SimMechanicsOpenEditorOnUpdate="on");
    % Enable animation in MATLAB figure
    doViz = true;
end
```

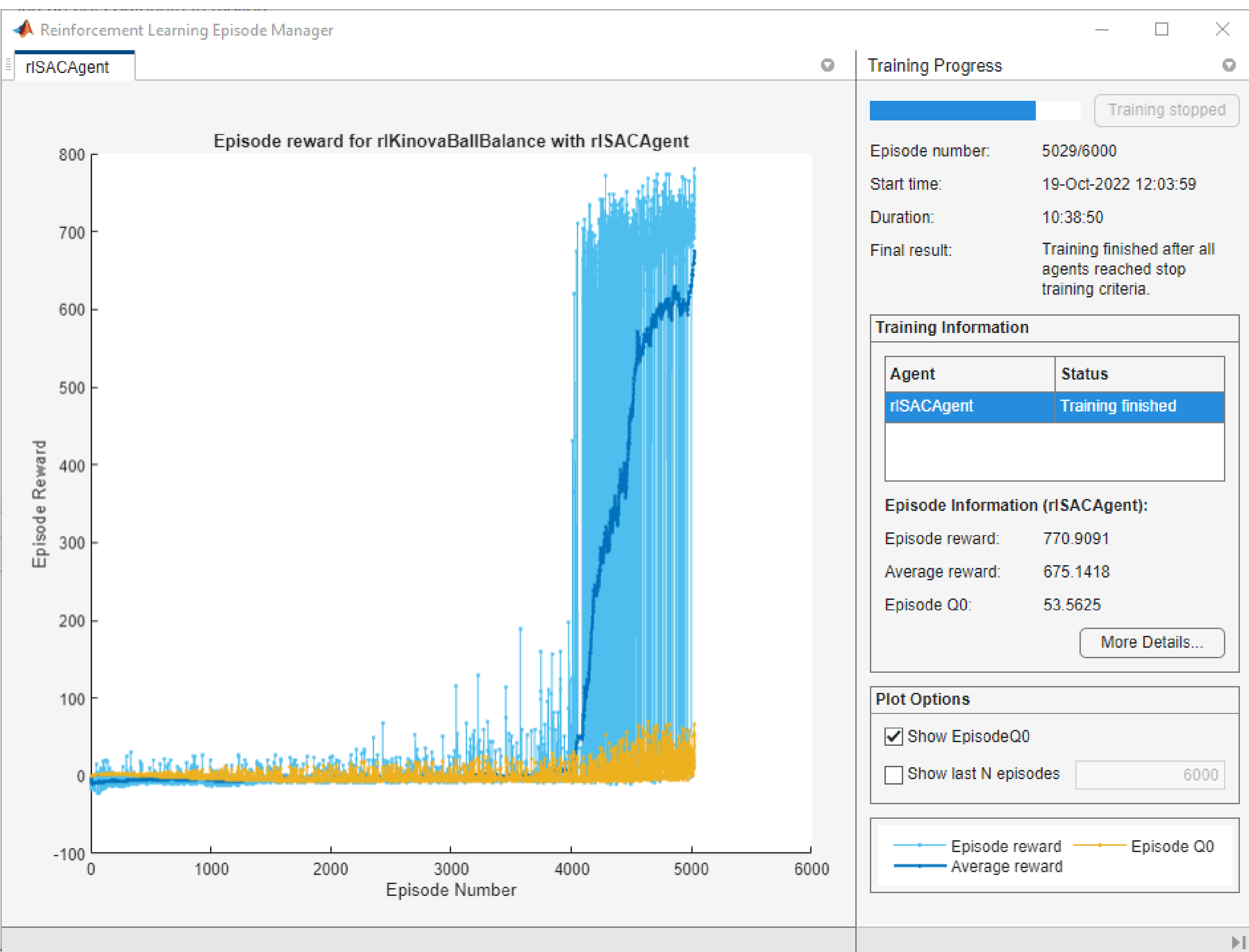
You can log training data to disk using the `rlDataLogger` function. For this example, log the episode experience and actor and critic function losses. The functions `logAgentLearnData` and `logEpisodeData` are defined at the end of this Live Script. For more information see “Log Training Data to Disk” on page 5-24.

```
logger = rlDataLogger();
logger.AgentLearnFinishedFcn = @logAgentLearnData;
logger.EpisodeFinishedFcn    = @(data) logEpisodeData(data, doViz);
```

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

```
doTraining = false;
if doTraining
    trainResult = train(agent,env,trainOpts,Logger=logger);
else
    load("kinovaBallBalanceAgent.mat")
end
```

A snapshot of training progress is shown in the following figure. You can expect different results due to randomness in the training process.



#### Simulate Trained Agent

Specify an initial position for the ball with respect to the plate center. To randomize the initial ball position during simulation, set the `userSpecifiedConditions` flag to `false`.

```
userSpecifiedConditions = true;
if userSpecifiedConditions
    ball.x0 = 0.10;
    ball.y0 = -0.10;
    env.ResetFcn = [];
else
    env.ResetFcn = @kinovaResetFcn;
end
```

Create a simulation options object for configuring the simulation. The agent will be simulated for a maximum of `floor(Tf/Ts)` steps per simulation episode.

```
simOpts = rlSimulationOptions(MaxSteps=floor(Tf/Ts));
```

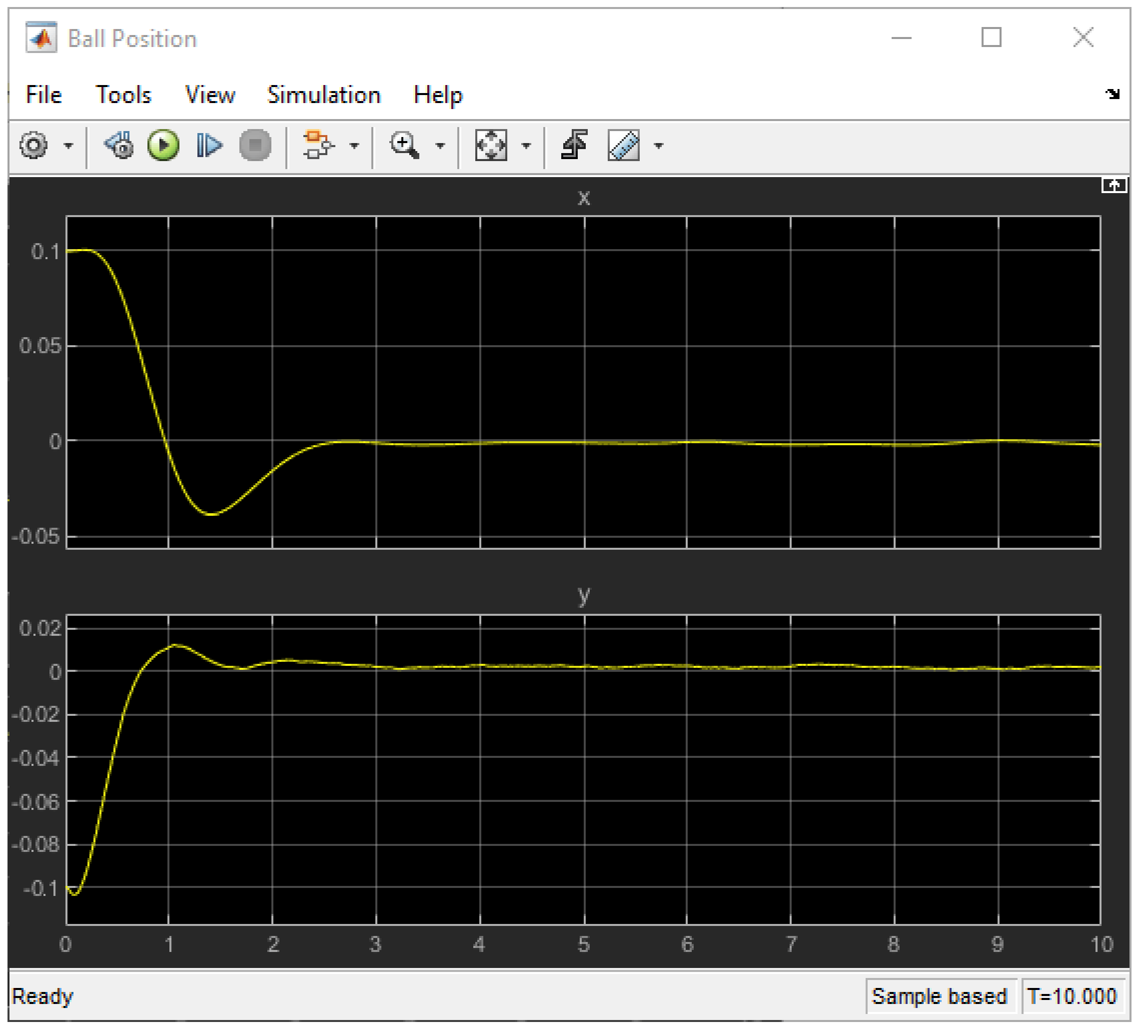
Enable visualization during simulation.

```
set_param(mdl, SimMechanicsOpenEditorOnUpdate="on");
doViz = true;
```

Simulate the agent.

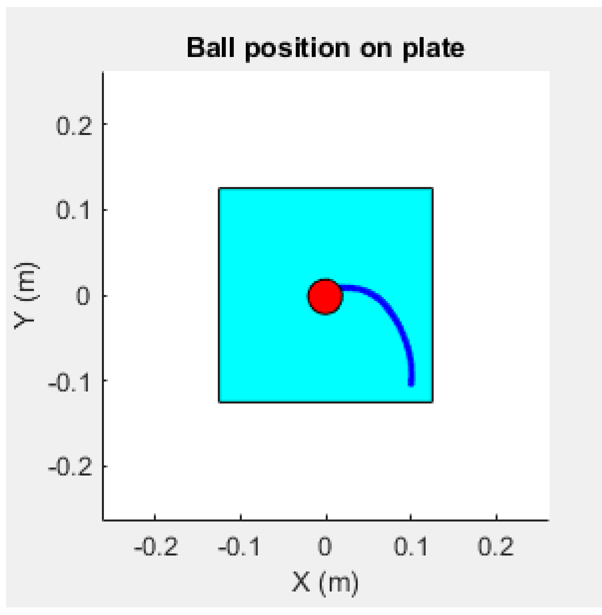
```
experiences = sim(agent,env,simOpts);
```

View the trajectory of the ball using the Ball Position scope block.




View the animation of the ball on the plate in a MATLAB figure.

```
fig = animatedPath(experiences);
```




**Environment Reset Function**

```
function in = kinovaResetFcn(in)
    % KinovaResetFcn is used to randomize the initial joint angles R6_q0,
    % R7_q0 and the initial wrist and hand torque values.

    % Ball parameters
    ball.radius = 0.02;     % m
    ball.mass   = 0.0027;   % kg
    ball.shell  = 0.0002;   % m

    % Calculate ball moment of inertia.
    ball.moi = calcMOI(ball.radius,ball.shell,ball.mass);

    % Initial conditions. +z is vertically upward.
    % Randomize the x and y distances within the plate.
    ball.x0  = -0.125 + 0.25*rand;  % m, initial x distance from plate center
    ball.y0  = -0.125 + 0.25*rand;  % m, initial y distance from plate center
    ball.z0  = ball.radius;         % m, initial z height from plate surface

    ball.dx0 = 0;   % m/s, ball initial x velocity
    ball.dy0 = 0;   % m/s, ball initial y velocity
    ball.dz0 = 0;   % m/s, ball initial z velocity

    % Contact friction parameters
    ball.staticfriction     = 0.5;
    ball.dynamicfriction    = 0.3;
    ball.criticalvelocity   = 1e-3;

    % Convert coefficient of restitution to spring-damper parameters.
    coeff_restitution = 0.89;
    [k, c, w] = cor2SpringDamperParams(coeff_restitution,ball.mass);
    ball.stiffness = k;
    ball.damping = c;
    ball.transitionwidth = w;
```

```
    in = setVariable(in,"ball",ball);

    % Randomize joint angles within a range of +/- 5 deg from the
    % starting positions of the joints.
    R6_q0 = deg2rad(-65) + deg2rad(-5+10*rand);
    R7_q0 = deg2rad(-90) + deg2rad(-5+10*rand);
    in = setVariable(in,"R6_q0",R6_q0);
    in = setVariable(in,"R7_q0",R7_q0);

    % Compute approximate initial joint torques that hold the ball,
    % plate and arm at their initial congifuration
    g = 9.80665;
    wrist_torque_0 = ...
        (-1.882 + ball.x0 * ball.mass * g) * cos(deg2rad(-65) - R6_q0);
    hand_torque_0 = ...
        (0.0002349 - ball.y0 * ball.mass * g) * cos(deg2rad(-90) - R7_q0);
    U0 = [wrist_torque_0 hand_torque_0];
    in = setVariable(in,"U0",U0);
end
```

### Data Logging Functions

```
function dataToLog = logAgentLearnData(data)
% This function is executed after completion
% of the agent's learning subroutine

dataToLog.ActorLoss = data.ActorLoss;
dataToLog.CriticLoss = data.CriticLoss;

end

function dataToLog = logEpisodeData(data, doViz)
% This function is executed after the completion of an episode

dataToLog.Experience = data.Experience;

% Show an animation after episode completion
if doViz
    animatedPath(data.Experience);
end

end
```

## See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlSACAgent` | `rlSACAgentOptions` | `rlTrainingOptions` | `rlSimulationOptions`

**Blocks**
RL Agent

## Related Examples

- "Train Biped Robot to Walk Using Reinforcement Learning Agents" on page 5-267
- "Control Robot Manipulator Using Passivity-Based Nonlinear MPC" (Model Predictive Control Toolbox)

## More About

- "Define Reward Signals" on page 2-14
- "Soft Actor-Critic (SAC) Agents" on page 3-35
- "Train Reinforcement Learning Agents" on page 5-3

# Automatic Parking Valet with Unreal Engine Simulation

This example shows the design of a hybrid controller for an automatic searching and parking task. You will learn how to combine adaptive model predictive control (MPC) with reinforcement learning (RL) to perform a parking maneuver. The path of the vehicle is visualized using the Unreal Engine® simulation environment.

The control objective is to park the vehicle in an empty spot after starting from an initial pose. The control algorithm executes a series of maneuvers while sensing and avoiding obstacles in tight spaces. It switches between an adaptive MPC controller and a reinforcement learning agent to complete the parking maneuver. The adaptive MPC controller moves the vehicle at a constant speed along a reference path while searching for an empty parking spot. When a spot is found, the reinforcement learning agent takes over and executes a pretrained parking maneuver. Prior knowledge of the environment (the parking lot) including the locations of the empty spots and parked vehicles is available to both the controllers.

#### Parking Lot

The parking lot environment used in this example is a subsection of the Large Parking Lot (Automated Driving Toolbox) scene. The parking lot is represented by the `ParkingLotEnvironment` object, which stores information on the ego vehicle, empty parking spots, and static obstacles (parked cars and boundaries). Each parking spot has a unique index number and an indicator light that is either green (free) or red (occupied). Parked vehicles are represented in black while other boundaries are outlined in green.

Specify a sample time $T_s$ (seconds) for the controllers and a simulation time $T_f$ (seconds).

```
Ts = 0.1;
Tf = 50;
```

Create a reference path for the ego vehicle using the `createReferenceTrajectory` helper function included with this example. The reference path starts from the south-east corner of the parking lot and ends in the west as displayed with the dashed pink line.

```
xRef = createReferenceTrajectory(Ts,Tf);
```

Create a `ParkingLotEnvironment` object with a free spot at index 32 and the specified reference path `xRef`.

```
freeSpotIndex = 32;
map = ParkingLotEnvironment(freeSpotIndex,"Route",xRef);
```

Specify an initial pose $(X_0, Y_0, \theta_0)$ for the ego vehicle. The units of $X_0$ and $Y_0$ are meters and $\theta_0$ is in radians.

```
egoInitialPose = [40 -55 pi/2];
```

Compute the target pose for the vehicle using the `createTargetPose` function. The target pose corresponds to the location in `freeSpotIdx`.

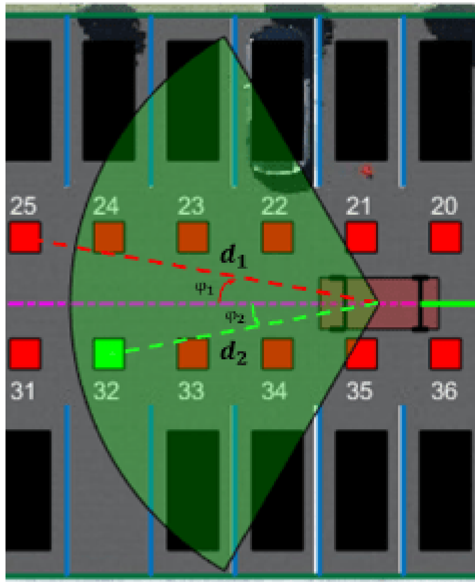
```
egoTargetPose = createTargetPose(map,freeSpotIndex);
```

### Sensor Modules

The parking algorithm uses geometric approximations of a camera and a lidar sensor to gather information from the parking lot environment.

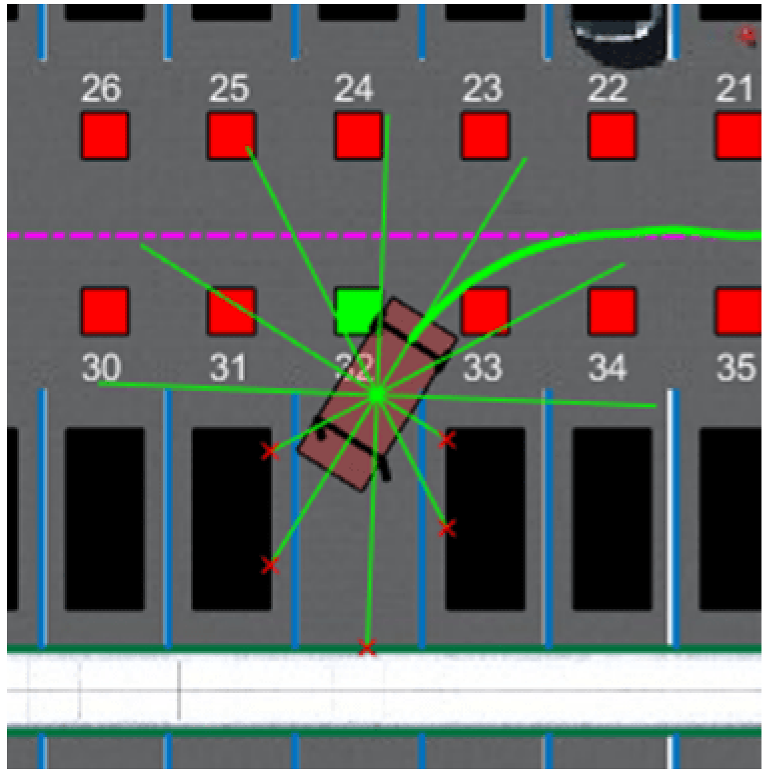
#### Camera

In this example, the field of view of a camera mounted on the ego vehicle is represented by the area shaded in green in the following figure. The camera has a field of view $\varphi$ bounded by $\pm\pi/3$ radians and a maximum measurement depth $d_{max}$ of 10 m. As the ego vehicle moves forward, the camera module senses the parking spots within the field of view and determines whether a spot is free or occupied. For simplicity, this action is implemented using geometrical relationships between the spot locations and the current vehicle pose. A parking spot is within the camera range if $d_i \leq d_{max}$ and $\varphi_{min} \leq \varphi_i \leq \varphi_{max}$, where $d_i$ is the distance to the parking spot and $\varphi_i$ is the angle to the parking spot.




#### Lidar

The lidar sensor in this example is modeled using radial line segments emerging from the geometric center of the vehicle. Distances to obstacles are measured along these line segments. The maximum measurable lidar distance along any line segment is 6 m. The reinforcement learning agent uses these readings to determine the proximity of the ego vehicle to other vehicles in the environment.



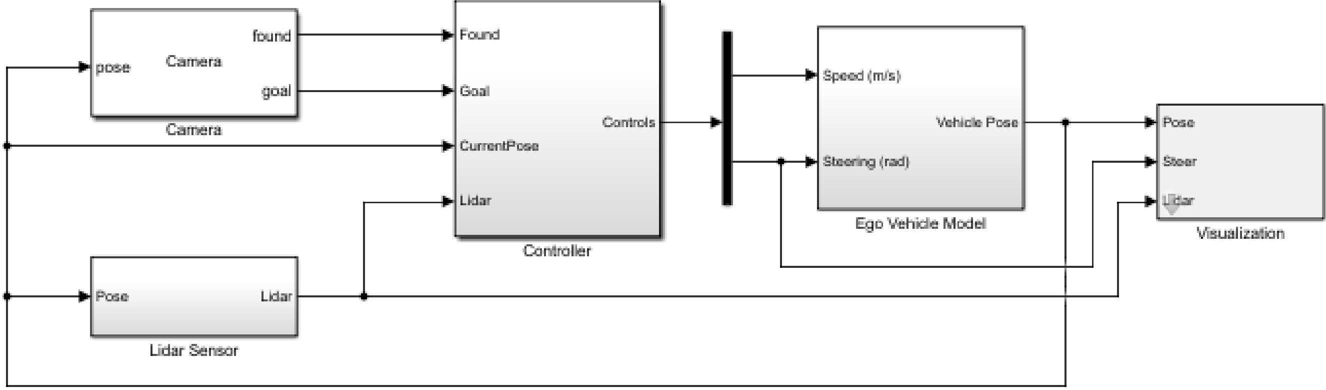

#### Auto Parking Valet Model

Load the auto parking valet parameters.

```
autoParkingValetParams3D
```

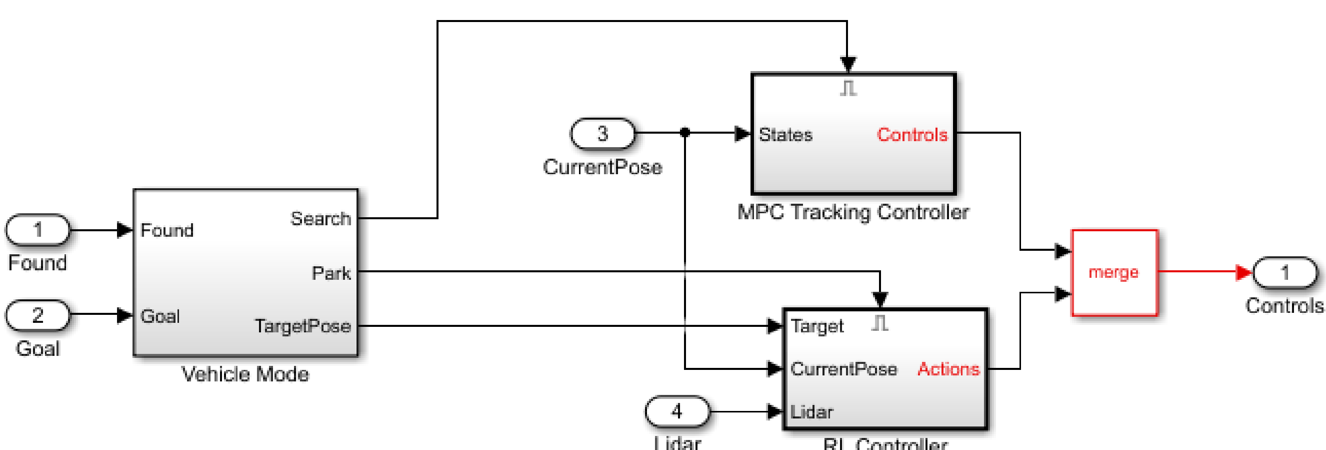
The parking valet model, including the controllers, ego vehicle, sensors, and parking lot, is implemented in a Simulink model. Open the model.

```
mdl = "rlAutoParkingValet3D";
open_system(mdl)
```

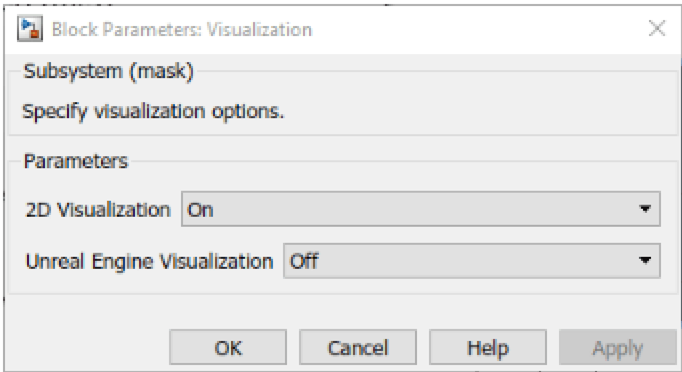



In this model:

- The vehicle is modeled in the Ego Vehicle Model subsystem. The kinematics is represented by a single-track bicycle kinematics model with two input signals: vehicle speed $v$ (m/s) and steering angle $\delta$ (radians).
- The adaptive MPC and RL agent blocks are found in the MPC Tracking Controller and RL Controller subsystems, respectively.
- Mode switching between the controllers is handled by the Vehicle Mode subsystem, which outputs Search and Park signals. Initially, the vehicle is in search mode and the adaptive MPC controller tracks the reference path. When a free spot is found, the Park signal activates the RL Agent to perform the parking maneuver.



- The Visualization Subsystem handles animation of the environment. Double-click this subsystem to specify the visualization options.




To plot the environment in a figure, set the **2D Visualization** parameter to `On`.

If you have Automated Driving Toolbox™ software installed, you can set the **Unreal Engine Visualization** parameter to `On` to display the vehicle animation in the Unreal Engine environment. Enabling the Unreal Engine simulation can degrade simulation performance. Therefore, set the parameter to `Off` when training the agent.

### Adaptive Model Predictive Controller Design

Create the adaptive MPC controller object for reference trajectory tracking using the `createMPCForParking` script. For more information on adaptive MPC, see "Adaptive MPC" (Model Predictive Control Toolbox).

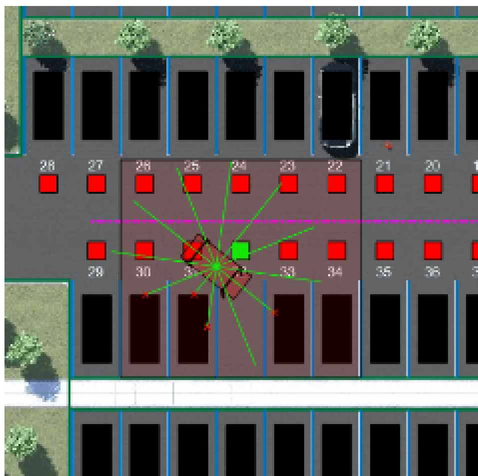
```
createMPCForParking3D;
```

### Reinforcement Learning Controller Design

To train the reinforcement learning agent, you must create an environment interface and an agent object.

### Create Environment

The environment for training is the region shaded in red in the following figure. Due to symmetry in the parking lot, training within this region is sufficient for the policy to adjust to other regions after coordinate transformations are applied to the observations. Constraining the training to this region also significantly reduces training duration when compared to training over the entire parking lot space.



For this environment:

- The training region is a 13.625 m x 12.34 m space with the target spot at its horizontal center.
- The observations are the position errors $X_e$ and $Y_e$ of the ego vehicle with respect to the target pose, the sine and cosine of the true heading angle $\theta$, and the lidar sensor readings.
- The vehicle speed during parking is a constant 2 m/s.
- The action signals are discrete steering angles that range between +/- $\pi/4$ radians in steps of 0.2618 radians or 15 degrees.
- The vehicle is considered parked if the errors with respect to target pose are within specified tolerances of +/- 0.75 m (position) and +/-10 degrees (orientation).
- The episode terminates if the ego vehicle goes out of the bounds of the training region, collides with an obstacle, or parks successfully.
- The reward $r_t$ provided at time $t$, is:

$$r_t = 2e^{-\left(0.05{X_e}^2 + 0.04{Y_e}^2\right)} + 0.5e^{-40{\theta_e}^2} - 0.05\delta^2 + 100f_t - 50g_t$$

Here, $X_e$, $Y_e$, and $\theta_e$ are the position and heading angle errors of the ego vehicle from the target pose, while $\delta$ is the steering angle. $f_t$ (0 or 1) indicates whether the vehicle has parked and $g_t$ (0 or 1) indicates if the vehicle has collided with an obstacle or left the training region at time $t$.

The coordinate transformations on vehicle pose $(X, Y, \theta)$ observations for different parking spot locations are as follows:

- Parking spots 1-14: $\overline{X} = X, \overline{Y} = Y + 20.41,\ \overline{\theta} = \theta$
- Parking spots 15-28: $\overline{X} = 41 - X, \overline{Y} = -64.485 - Y,\ \overline{\theta} = \theta - \pi$
- Parking spots 29-37: no transformation
- Parking spots38-46: $\overline{X} = 41 - X, \overline{Y} = -84.48 - Y, \overline{\theta} = \theta - \pi$

Create the observation and action specifications for the environment.

```
nObs = 16;
nAct = 1;
obsInfo = rlNumericSpec([nObs 1]);
```

```
obsInfo.Name = "observations";
actInfo = rlNumericSpec([nAct 1],LowerLimit=-1,UpperLimit=1);
actInfo.Name = "actions";
```

Create the Simulink environment interface, specifying the path to the RL Agent block.

```
blk = mdl + "/Controller/RL Controller/RL Agent";
env = rlSimulinkEnv(mdl,blk,obsInfo,actInfo);
```

Specify a reset function for training. The `autoParkingValetResetFcn` function resets the initial pose of the ego vehicle to random values at the start of each episode.

```
env.ResetFcn = @autoParkingValetResetFcn3D;
```

For more information on creating Simulink environments, see `rlSimulinkEnv`.

### Create Agent

The agent in this example is a twin-delayed deep deterministic policy gradient (TD3) agent. TD3 agents rely on actor and critic approximator objects to learn the optimal policy. To learn more about TD3 agents, see “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents” on page 3-44.

The actor and critic networks are initialized randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

TD3 agents use two parametrized Q-value function approximators to estimate the value of the policy. To model the parametrized Q-value function within both critics, use a neural network with 16 inputs and one output. The output of the critic network is the state-action value function for a taking a given action from a given observation.

Define each network path as an array of layer objects. Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% Main path
mainPath = [
    featureInputLayer(nObs,Name="StateInLyr")
    fullyConnectedLayer(128)
    concatenationLayer(1,2,Name="concat")
    reluLayer
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(1,Name="CriticOutLyr")
    ];

% Action path
actionPath = [
    featureInputLayer(nAct,Name="ActionInLyr")
    fullyConnectedLayer(128,Name="fc2")
    ];

% Convert to layergraph object and connect layers
criticNet = layerGraph(mainPath);
criticNet = addLayers(criticNet, actionPath);
criticNet = connectLayers(criticNet,"fc2","concat/in2");
```

Convert to a `dlnetwork` object and display the number of learnable parameters.

```
criticdlnet = dlnetwork(criticNet);
summary(criticdlnet)
```

```
   Initialized: true

   Number of learnables: 35.4k

   Inputs:
      1   'StateInLyr'    16 features
      2   'ActionInLyr'   1 features
```

Create the critics using `criticdlnet`, the environment observation and action specifications, and the names of the network input layers to be connected with the environment observation and action channels. For more information see `rlQValueFunction`.

```
critic1 = rlQValueFunction(criticNet,obsInfo,actInfo,...
    ObservationInputNames="StateInLyr",ActionInputNames="ActionInLyr");
critic2 = rlQValueFunction(criticNet,obsInfo,actInfo,...
    ObservationInputNames="StateInLyr",ActionInputNames="ActionInLyr");
```

Create the neural network for the actor. The output of the actor network is the steering angle.

```
anet = [
    featureInputLayer(nObs)
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(nAct)
    tanhLayer
    ];

actorNet = layerGraph(anet);
```

Convert to a `dlnetwork` object and display the number of learnable parameters.

```
actordlnet = dlnetwork(actorNet);
summary(actordlnet)
```

```
   Initialized: true

   Number of learnables: 18.8k

   Inputs:
      1   'input'   16 features
```

Create the actor object for the TD3 agent. For more information see `rlContinuousDeterministicActor`.

```
actor = rlContinuousDeterministicActor(actordlnet,obsInfo,actInfo);
```

Specify the agent options and create the TD3 agent. For more information on TD3 agent options, see `rlTD3AgentOptions`.

```
agentOpts = rlTD3AgentOptions(SampleTime=Ts, ...
    DiscountFactor=0.99, ...
```

```
    ExperienceBufferLength=1e6, ...
    MiniBatchSize=128);
```

Set the noise options for exploration.

```
agentOpts.ExplorationModel.StandardDeviation = 0.1;
agentOpts.ExplorationModel.StandardDeviationDecayRate = 1e-4;
agentOpts.ExplorationModel.StandardDeviationMin = 0.01;
```

For this example, set the actor and critic learn rates to `1e-3` and `2e-3`, respectively. Set a gradient threshold factor of `1` to limit the gradients during training. For more information, see `rlOptimizerOptions`.

Specify training options for the actor.

```
agentOpts.ActorOptimizerOptions.LearnRate = 1e-3;
agentOpts.ActorOptimizerOptions.GradientThreshold = 1;
agentOpts.ActorOptimizerOptions.L2RegularizationFactor = 1e-3;
```

Specify training options for the critic.

```
agentOpts.CriticOptimizerOptions(1).LearnRate = 2e-3;
agentOpts.CriticOptimizerOptions(2).LearnRate = 2e-3;
agentOpts.CriticOptimizerOptions(1).GradientThreshold = 1;
agentOpts.CriticOptimizerOptions(2).GradientThreshold = 1;
```

Create the agent using the actor, the critics, and the agent options objects. For more information, see `rlTD3Agent`.

```
agent = rlTD3Agent(actor,[critic1 critic2], agentOpts);
```

#### Train Agent

To train the agent first specify the training options.

The agent is trained for a maximum of `10000` episodes with each episode lasting a maximum of `200` time steps. The training terminates when the maximum number of episodes is reached or the average reward over `200` episodes reaches the value of `120` or more. Specify the options for training using the `rlTrainingOptions` function.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=10000,...
    MaxStepsPerEpisode=200,...
    ScoreAveragingWindowLength=200,...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=120);
```

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

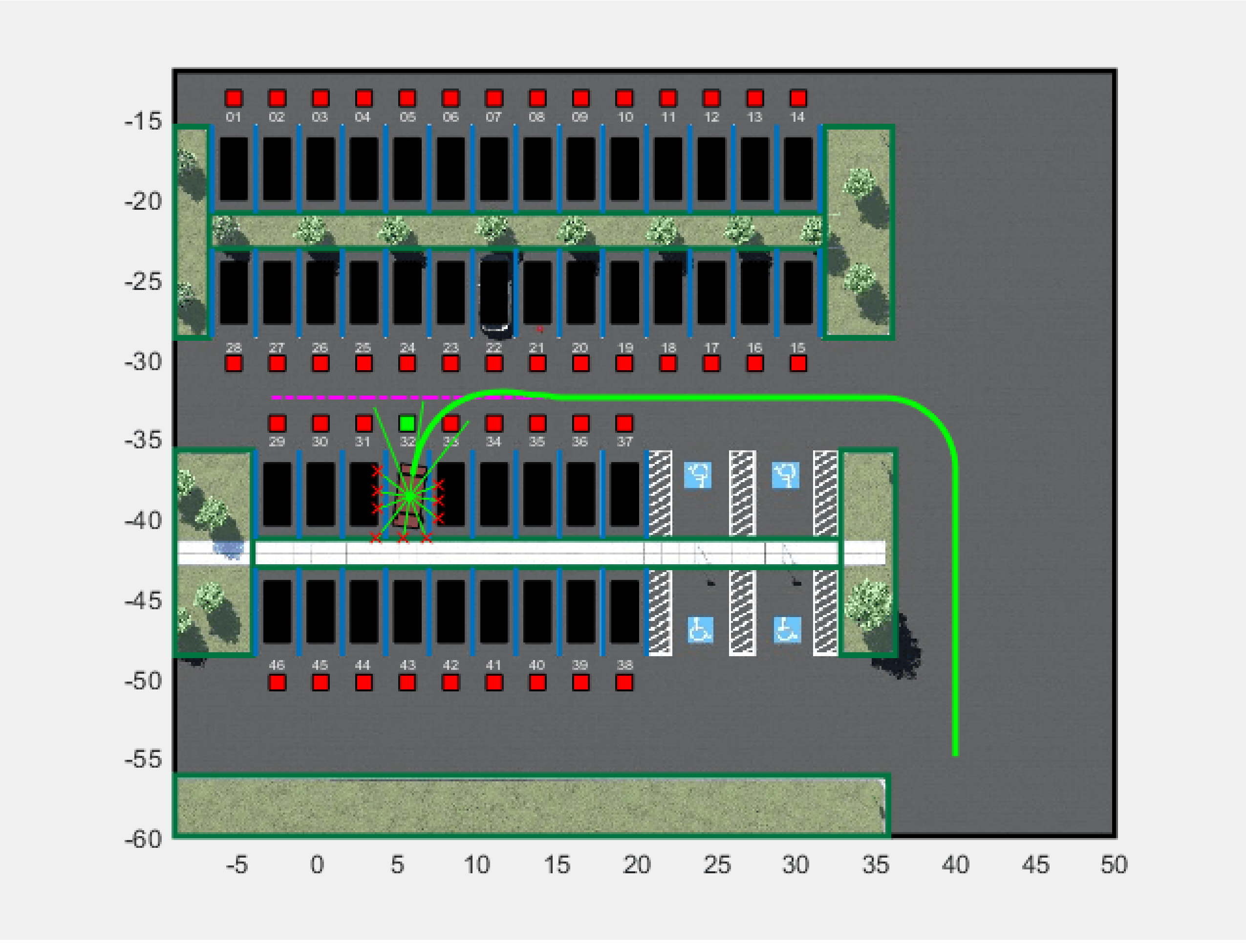
```
doTraining = false;
if doTraining
    trainingResult = train(agent,env,trainOpts);
else
```

```
    load("rlAutoParkingValetAgent.mat","agent");
end
```

### Simulate Parking Task

To Validate the trained agent, simulate the model and observe the parking maneuver.

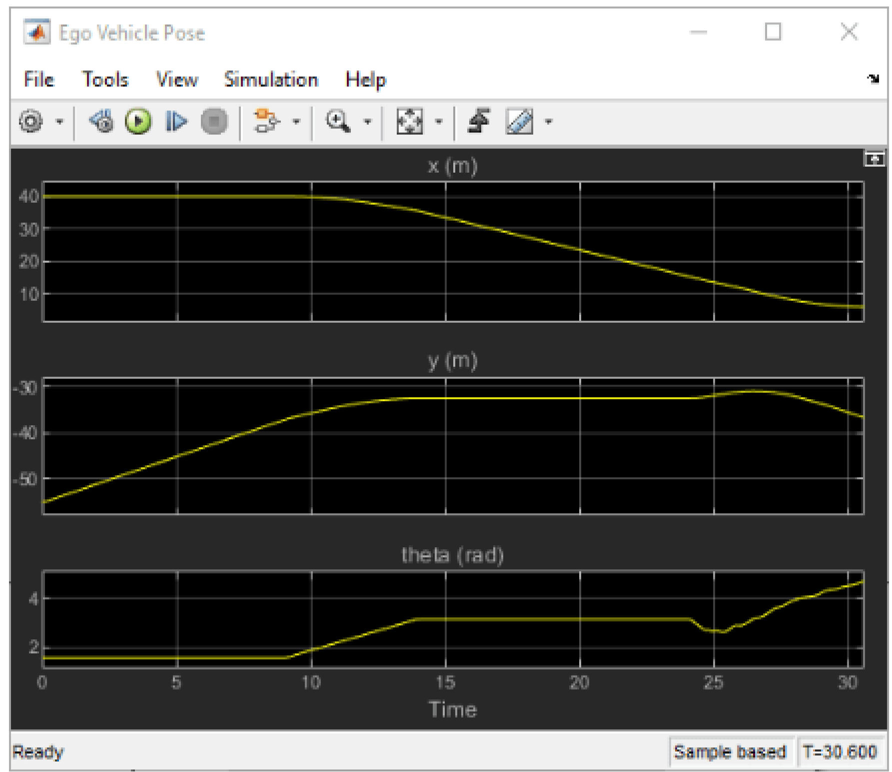
```
sim(mdl);
```




The vehicle tracks the reference path using the MPC controller before switching to the RL controller when the target spot is detected. The vehicle then completes the parking maneuver.

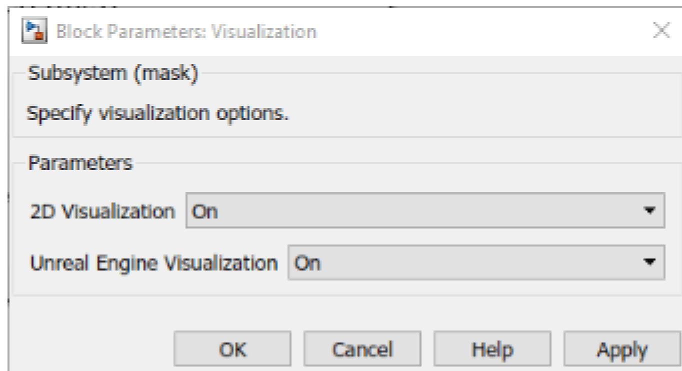
To view the trajectory, open the Ego Vehicle Pose scope.

```
open_system(mdl + "/Ego Vehicle Model/Ego Vehicle Pose")
```



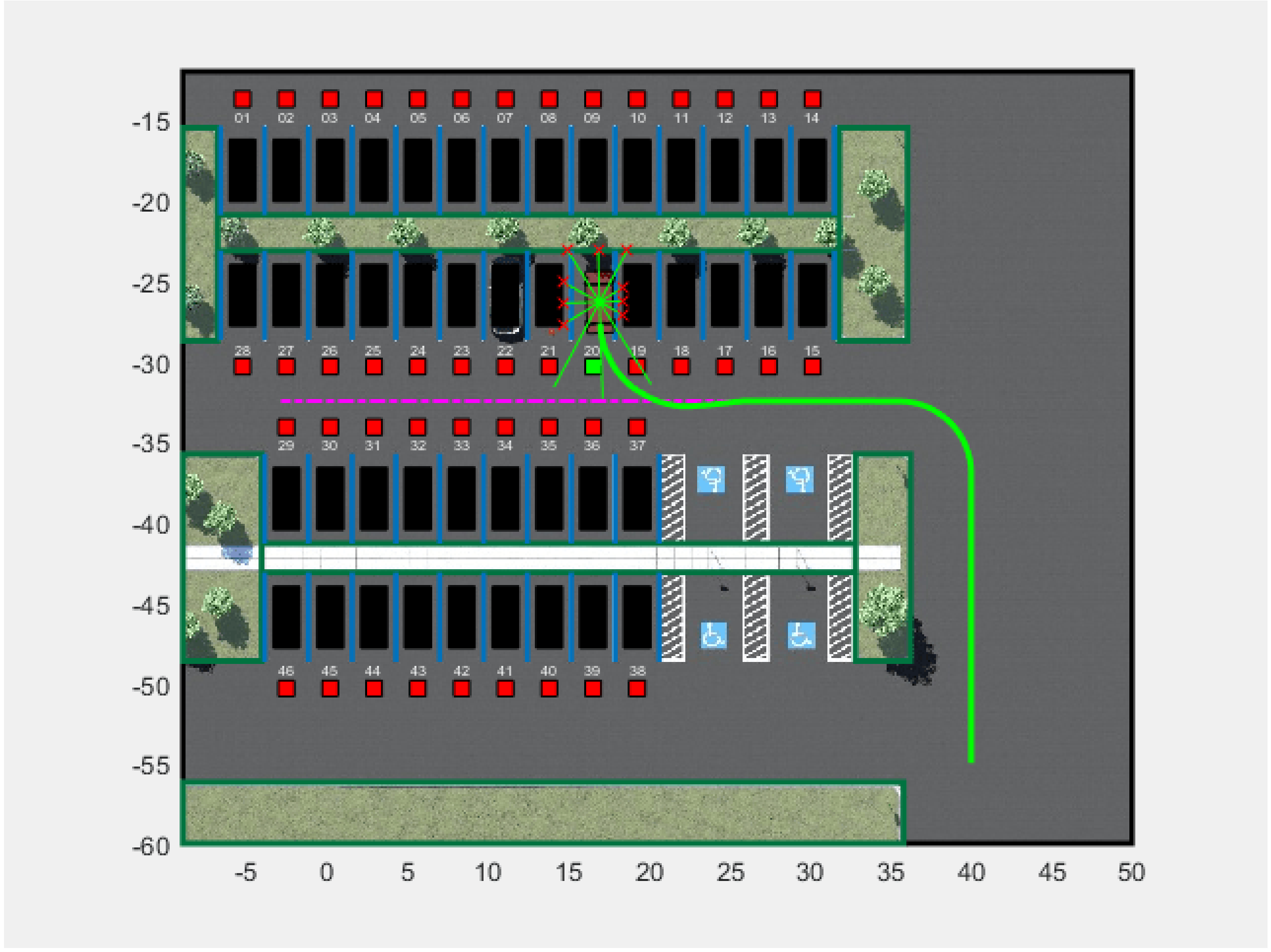

### Unreal Engine Simulation

Turn on the Unreal Engine visualization by opening the Visualization block and setting the **Unreal Engine Visualization** parameter to `On`. Initializing the Unreal Engine simulation environment can take a few seconds..




Park the vehicle in a different spot.

```
freeSpotIndex = 20;
sim(mdl)
```




## See Also

**Functions**
`train` | `rlSimulinkEnv`

**Objects**
`rlTD3Agent` | `rlTD3AgentOptions` | `rlTrainingOptions` | `rlOptimizerOptions`

**Blocks**
RL Agent

## Related Examples

- “Train PPO Agent for Automatic Parking Valet” on page 5-235
- “Parking Valet Using Multistage Nonlinear Model Predictive Control” (Model Predictive Control Toolbox)
- “Parallel Parking Using Nonlinear Model Predictive Control” (Model Predictive Control Toolbox)
- “Parallel Parking Using RRT Planner and MPC Tracking Controller” (Model Predictive Control Toolbox)

## More About

- “Adaptive MPC” (Model Predictive Control Toolbox)
- “Create Policies and Value Functions” on page 4-2
- “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents” on page 3-44
- “Train Reinforcement Learning Agents” on page 5-3

# Generate Reward Function from a Model Predictive Controller for a Servomotor

This example shows how to automatically generate a reward function from cost and constraint specifications defined in a model predictive controller object. You then use the generated reward function to train a reinforcement learning agent.

### Introduction

You can use the `generateRewardFunction` to generate a reward function for reinforcement learning, starting from cost and constraints specified in a model predictive controller. The resulting reward signal is a sum of costs (as defined by an objective function) and constraint violation penalties depending on the current state of the environment.

This example is based on the “DC Servomotor with Constraint on Unmeasured Output” (Model Predictive Control Toolbox) example, in which you design a model predictive controller for a DC servomechanism under voltage and shaft torque constraints. Here, you will convert the cost and constraints specifications defined in the `mpc` object into a reward function and use it to train an agent to control the servomotor.

Open the Simulink model for this example which is based on the above MPC example but has been modified for reinforcement learning.

```
mdl = "rl_motor";
open_system(mdl);
```

### Create Model Predictive Controller

Create the open-loop dynamic model of the motor, defined in `plant` and the maximum admissible torque `tau` using an helper function.

```
[plant,tau] = mpcmotormodel;
```

Specify input and output signal types for the MPC controller. The shaft angular position, is measured as first output. The second output, torque, is unmeasurable.

```
plant = setmpcsignals(plant, MV=1, MO=1, UO=2);
```

Specify constraints on the manipulated variable, and define a scale factor.

```
MV = struct(Min=-220, Max=220, ScaleFactor=440);
```

Impose torque constraints during the first three prediction steps, and specify scale factor for both shaft position and torque.

```
OV = struct(Min={-Inf, [-tau;-tau;-tau;-Inf]}, ...
            Max={Inf, [tau;tau;tau;Inf]}, ...
            ScaleFactor={2*pi, 2*tau});
```

Specify weights for the quadratic cost function to achieve angular position tracking. Set to zero the weight for the torque, thereby allowing it to float within its constraint.

```
Weights = struct(MV=0, MVRate=0.1, OV=[0.1 0]);
```

Create an MPC controller for the `plant` model with a sample time of `0.1` s, a prediction horizon `10` steps, and a control horizon of `2` steps, using the previously defined structures for the weights, manipulated variables, and output variables.

```
mpcobj = mpc(plant, 0.1, 10, 2, Weights, MV, OV);
```

Display the controller specifications.

```
mpcobj
```

```
MPC object (created on 04-Mar-2023 01:29:42):
---------------------------------------------
Sampling time:      0.1 (seconds)
Prediction Horizon: 10
Control Horizon:    2

Plant Model:
                                      --------------
      1  manipulated variable(s)   -->|  4 states  |
                                      |            |-->  1 measured output(s)
      0  measured disturbance(s)   -->|  1 inputs  |
                                      |            |-->  1 unmeasured output(s)
      0  unmeasured disturbance(s) -->|  2 outputs |
                                      --------------
Indices:
  (input vector)    Manipulated variables: [1 ]
  (output vector)        Measured outputs: [1 ]
                       Unmeasured outputs: [2 ]
```

```
Disturbance and Noise Models:
        Output disturbance model: default (type "getoutdist(mpcobj)" for details)
         Measurement noise model: default (unity gain after scaling)

Weights:
        ManipulatedVariables: 0
    ManipulatedVariablesRate: 0.1000
             OutputVariables: [0.1000 0]
                         ECR: 10000

State Estimation:  Default Kalman Filter (type "getEstimator(mpcobj)" for details)

Constraints:
 -220 <= MV1 (V) <= 220, MV1/rate (V) is unconstrained,       MO1 (rad) is unconstrained
                                                        -78.54 <= UO1 (Nm)(t+1) <= 78.54
                                                        -78.54 <= UO1 (Nm)(t+2) <= 78.54
                                                        -78.54 <= UO1 (Nm)(t+3) <= 78.54
                                                          UO1 (Nm)(t+4) is unconstrained

Use built-in "active-set" QP solver with MaxIterations of 120.
```

The controller operates on a plant with 4 states, 1 input (voltage) and 2 output signals (angle and torque) and has the following specifications:

- The cost function weights for the manipulated variable, manipulated variable rate and output variables are 0, 0.1 and [0.1 0] respectively.
- The manipulated variable is constrained between -220V and 220V.
- The manipulated variable rate is unconstrained.
- The first output variable (angle) is unconstrained but the second (torque) is constrained between -78.54 Nm and 78.54 Nm in the first three prediction time steps and unconstrained in the fourth step.

Note that for reinforcement learning only the constraints specification from the first prediction time step will be used since the reward is computed for a single time step.

### Generate the Reward Function

Generate the reward function code from specifications in the `mpc` object using `generateRewardFunction`. The code is displayed in the MATLAB Editor.

```
generateRewardFunction(mpcobj)
```

The generated reward function is a starting point for reward design. You can modify the function with different penalty function choices and tune the weights. For this example, make the following change to the generated code:

- Scale the original cost weights `Qy` and `Qmvrate` by a factor of `50` and `10` respectively.
- Scale the penalty weights `Wy`, `Wmv` and `Wmvrate` by a factor of `1e-2`, `1e-3` and `1e-3` respectively.
- The default exterior penalty function method is `step`. Change the method to `quadratic`.

After you make changes, the cost and penalty specifications should be as follows:

```
Qy      = 50 * [0.1 0];
Qmv     = 0;
Qmvrate = 10 * 0.1;
```

```
Wy      = 1e-2 * [1 1];
Wmv     = 1e-3;
Wmvrate = 1e-3;
Py      = Wy      * exteriorPenalty(y,ymin,ymax,'quadratic');
Pmv     = Wmv     * exteriorPenalty(mv,mvmin,mvmax,'quadratic');
Pmvrate = Wmvrate * exteriorPenalty(mv-lastmv,mvratemin,mvratemax,'quadratic');
```

For this example, the modified code has been saved in the MATLAB function file `rewardFunctionMpc.m`. Display the generated reward function.

```
type rewardFunctionMpc.m
```

```
function reward = rewardFunctionMpc(y,refy,mv,refmv,lastmv)
% REWARDFUNCTIONMPC generates rewards from MPC specifications.
%
% Description of input arguments:
%
% y : Output variable from plant at step k+1
% refy : Reference output variable at step k+1
% mv : Manipulated variable at step k
% refmv : Reference manipulated variable at step k
% lastmv : Manipulated variable at step k-1
%
% Limitations (MPC and NLMPC):
%     - Reward computed based on first step in prediction horizon.
%       Therefore, signal previewing and control horizon settings are ignored.
%     - Online cost and constraint update is not supported.
%     - Custom cost and constraint specifications are not considered.
%     - Time varying cost weights and constraints are not supported.
%     - Mixed constraint specifications are not considered (for the MPC case).

% Reinforcement Learning Toolbox
% 02-Jun-2021 16:05:41

%#codegen

%% Specifications from MPC object
% Standard linear bounds as specified in 'States', 'OutputVariables', and
% 'ManipulatedVariables' properties
ymin = [-Inf -78.5398163397448];
ymax = [Inf 78.5398163397448];
mvmin = -220;
mvmax = 220;
mvratemin = -Inf;
mvratemax = Inf;

% Scale factors as specified in 'States', 'OutputVariables', and
% 'ManipulatedVariables' properties
Sy  = [6.28318530717959 157.07963267949];
Smv = 440;

% Standard cost weights as specified in 'Weights' property
Qy      = 50 * [0.1 0];
Qmv     = 0;
Qmvrate = 10 * 0.1;

%% Compute cost
dy      = (refy(:)-y(:)) ./ Sy';
```

```
dmv     = (refmv(:)-mv(:)) ./ Smv';
dmvrate = (mv(:)-lastmv(:)) ./ Smv';
Jy      = dy'      * diag(Qy.^2)      * dy;
Jmv     = dmv'     * diag(Qmv.^2)     * dmv;
Jmvrate = dmvrate' * diag(Qmvrate.^2) * dmvrate;
Cost    = Jy + Jmv + Jmvrate;

%% Penalty function weight (specify nonnegative)
Wy = 1e-2 * [1 1];
Wmv = 1e-3;
Wmvrate = 1e-3;

%% Compute penalty
% Penalty is computed for violation of linear bound constraints.
%
% To compute exterior bound penalty, use the exteriorPenalty function and
% specify the penalty method as 'step' or 'quadratic'.
%
% Alternatively, use the hyperbolicPenalty or barrierPenalty function for
% computing hyperbolic and barrier penalties.
%
% For more information, see help for these functions.
%
% Set Pmv value to 0 if the RL agent action specification has
% appropriate 'LowerLimit' and 'UpperLimit' values.
Py      = Wy      * exteriorPenalty(y,ymin,ymax,'quadratic');
Pmv     = Wmv     * exteriorPenalty(mv,mvmin,mvmax,'quadratic');
Pmvrate = Wmvrate * exteriorPenalty(mv-lastmv,mvratemin,mvratemax,'quadratic');
Penalty = Py + Pmv + Pmvrate;

%% Compute reward
reward = -(Cost + Penalty);
end
```

To integrate this reward function, open the MATLAB Function block in the Simulink model.

```
open_system("rl_motor/Reward Function")
```

Append the function with the following line of code and save the model.

```
r = rewardFunctionMpc(y,refy,mv,refmv,lastmv);
```

The MATLAB Function block will now execute `rewardFunctionMpc.m` during simulation.

For this example, the MATLAB Function block has already been modified and saved.

#### Create a Reinforcement Learning Environment

The environment dynamics are modeled in the **Servomechanism** subsystem. For this environment,

- The observations are the reference signals (angle and torque), output variables (angle and torque), and their integrals from the last 3 time steps. The angle and torque signals are normalized by multiplying with the gain `[0.1 1/tau]`.
- The action is the voltage $V$ applied to the servomotor. The action values are limited between `-220` and `220`.
- The sample time is $T_s = 0.1s$.

- The total simulation time is $T_f = 20s$.

Specify the total simulation time and sample time.

```
Tf = 20;
Ts = 0.1;
```

Create observation and action specifications for the environment.

```
numObs = 24;
numAct = 1;
oinfo = rlNumericSpec([numObs 1]);
ainfo = rlNumericSpec([numAct 1], ...
    LowerLimit=-220, ...
    UpperLimit=220);
```

Create the reinforcement learning environment using the `rlSimulinkEnv` function.

```
blk = "rl_motor/RL Agent";
env = rlSimulinkEnv(mdl, blk, oinfo, ainfo);
```

### Create a Reinforcement Learning Agent

Fix the random seed for reproducibility.

```
rng(0)
```

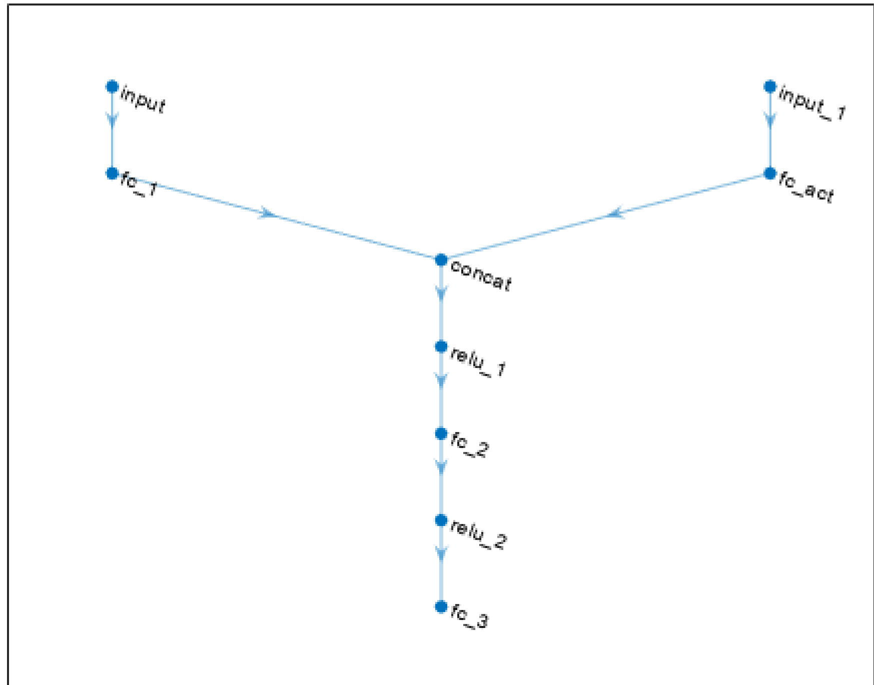
The agent used in this example is a twin-delayed deep deterministic policy gradient (TD3) agent. TD3 agents use two parametrized Q-value function approximators to estimate the value (that is the expected cumulative long-term reward) of the policy. To model the parametrized Q-value function within both critics, use a neural network with two inputs (the observation and action) and one output (the value of the policy when taking a given action from the state corresponding to a given observation). For more information on TD3 agents, see "Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents" on page 3-44.

Define each network path as an array of layer objects. Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
mainPath = [
    featureInputLayer(numObs)
    fullyConnectedLayer(128)
    concatenationLayer(1, 2, Name="concat")
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(1)];
actionPath = [
    featureInputLayer(numAct)
    fullyConnectedLayer(8, Name="fc_act")];

% Create layerGraph object and add layers
criticNet = layerGraph(mainPath);
criticNet = addLayers(criticNet, actionPath);

% Connect Layers
criticNet = connectLayers(criticNet, "fc_act", "concat/in2");
```

Plot the critic network structure.

```
plot(criticNet);
```




Create the critic function objects using `rlQValueFunction`. The critic function object encapsulates the critic by wrapping around the critic deep neural network. To make sure the critics have different initial weights, explicitly initialize each network before using them to create the critics.

```
% Convert the neural network to a dlnetwork object without initializing the networks
criticdlNet = dlnetwork(criticNet, Initialize=false);

% Create the two critic functions for the TD3 agent
critic1 = rlQValueFunction(initialize(criticdlNet), oinfo, ainfo);
critic2 = rlQValueFunction(initialize(criticdlNet), oinfo, ainfo);
```

TD3 agents learn a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by obsInfo) and one output layer (which returns the action to the environment action channel, as specified by `ainfo`).

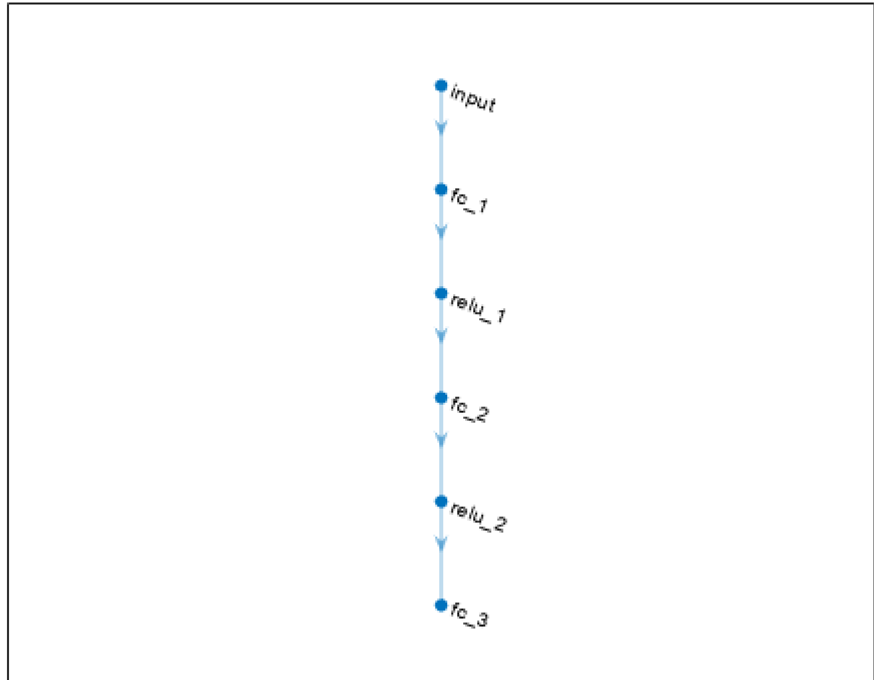
Define the network as an array of layer objects.

```
actorNet = [
    featureInputLayer(numObs)
```

```
    fullyConnectedLayer(128)
    reluLayer
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(numAct)];
```

Plot the actor network.

```
plot(layerGraph(actorNet));
```




Create a deterministic actor function that is responsible for modeling the policy of the agent. For more information, see `rlContinuousDeterministicActor`.

```
actordlNet = dlnetwork(actorNet);
actor = rlContinuousDeterministicActor(actordlNet, oinfo, ainfo);
```

Specify the agent options using `rlTD3AgentOptions`. The agent trains from an experience buffer of maximum capacity `1e6` by randomly selecting mini-batches of size `256`. The discount factor of `0.995` favors long-term rewards.

```
agentOpts = rlTD3AgentOptions(SampleTime=Ts, ...
    DiscountFactor=0.995, ...
    ExperienceBufferLength=1e6, ...
    MiniBatchSize=256);
```

Specify optimizer options for the actor and critic functions. For this example, you will choose a learn rate of `1e-3` and a gradient threshold of `1` for the actor and critics.

```
% Critic optimizer options
for idx = 1:2
    agentOpts.CriticOptimizerOptions(idx).LearnRate = 1e-3;
    agentOpts.CriticOptimizerOptions(idx).GradientThreshold = 1;
end

% Actor optimizer options
agentOpts.ActorOptimizerOptions.LearnRate = 1e-3;
agentOpts.ActorOptimizerOptions.GradientThreshold = 1;
```

During training, the agent explores the action space using a Gaussian action noise model. Set the standard deviation and decay rate of the noise using the `ExplorationModel` property. The noise has an initial standard deviation of `100` which exponentially decays at the rate of 1e-5 until it reaches a mininum value of 1e-3. This favors exploration towards the beginning of training and exploitation in later stages.

```
agentOpts.ExplorationModel.StandardDeviationMin = 1e-3;
agentOpts.ExplorationModel.StandardDeviation = 100;
agentOpts.ExplorationModel.StandardDeviationDecayRate = 1e-5;
```

Create the TD3 agent using the actor and critic representations. For more information on TD3 agents, see `rlTD3Agent`.

```
agent = rlTD3Agent(actor, [critic1,critic2], agentOpts);
```

### Train the Agent

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

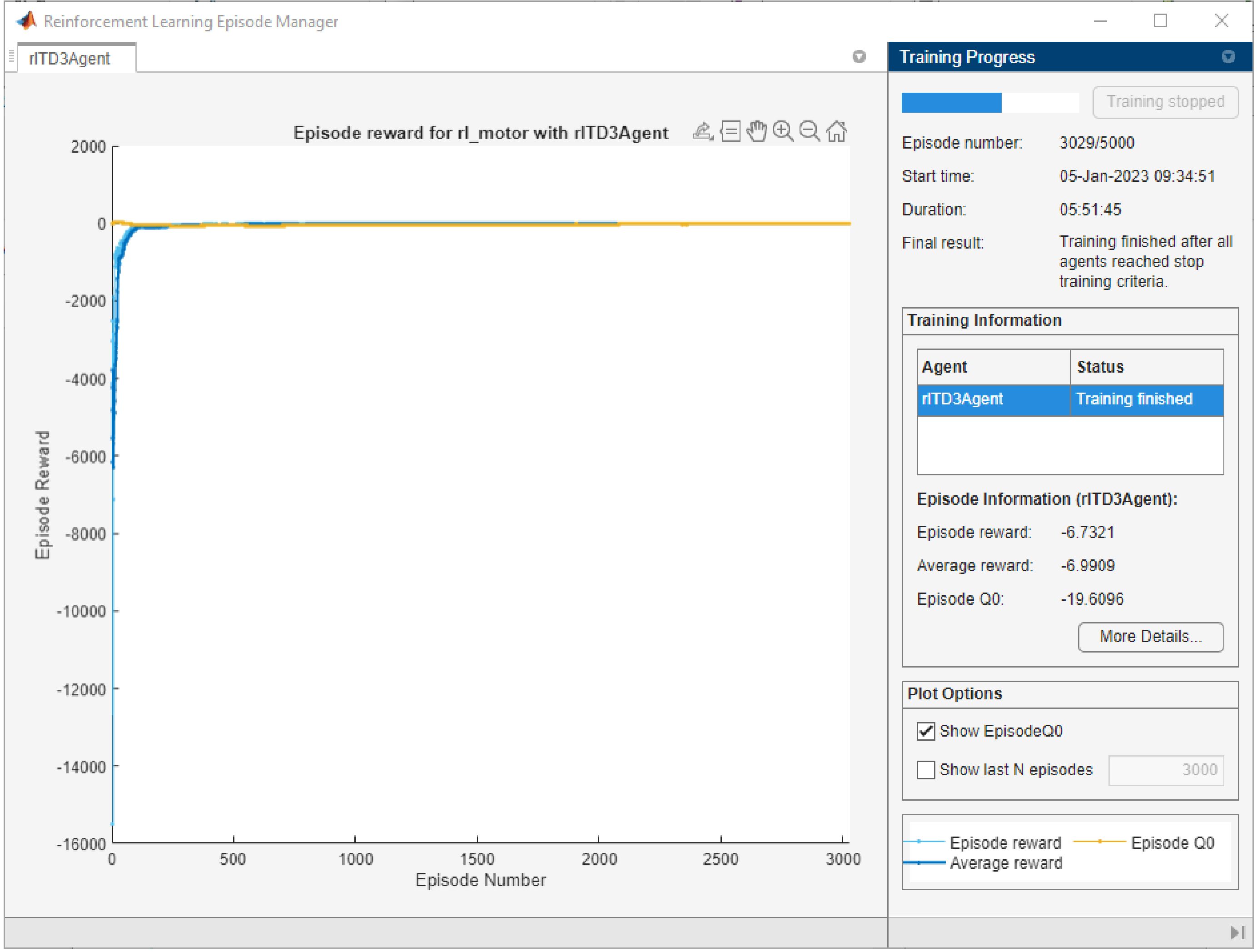
- Run each training for at most `5000` episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Stop the training when the agent receives an average cumulative reward greater than `-7` over `20` consecutive episodes. At this point, the agent can track the reference signal.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=5000, ...
    MaxStepsPerEpisode=ceil(Tf/Ts), ...
    StopTrainingCriteria="AverageReward", ...
    StopTrainingValue=-7, ...
    ScoreAveragingWindowLength=20);
```

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

```
doTraining = false;
if doTraining
    trainResult = train(agent, env, trainOpts);
else
    load('rlDCServomotorTD3Agent.mat')
end
```

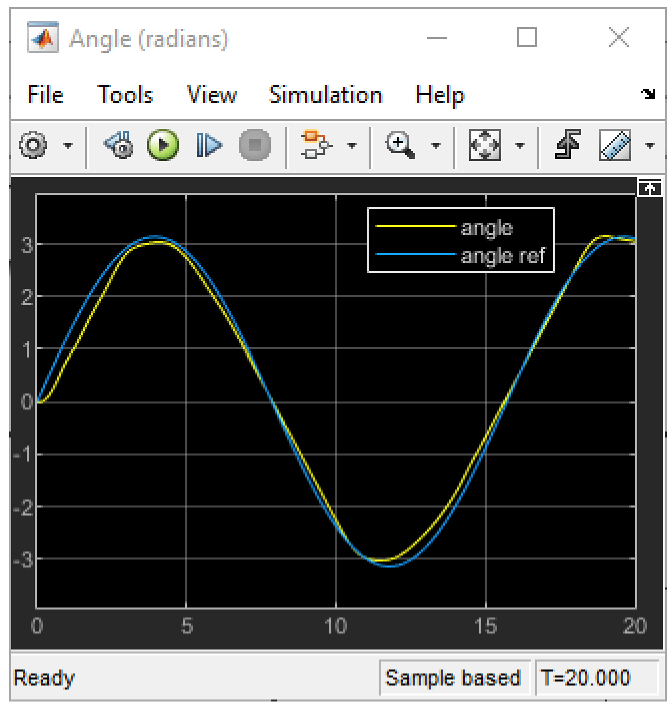
A snapshot of the training progress is shown in the following figure. You can expect different results due to inherent randomness in the training process.

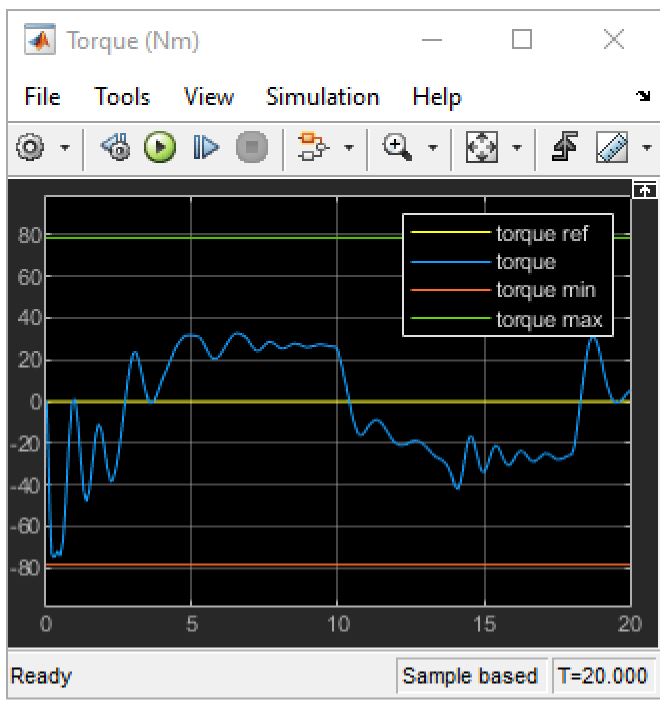


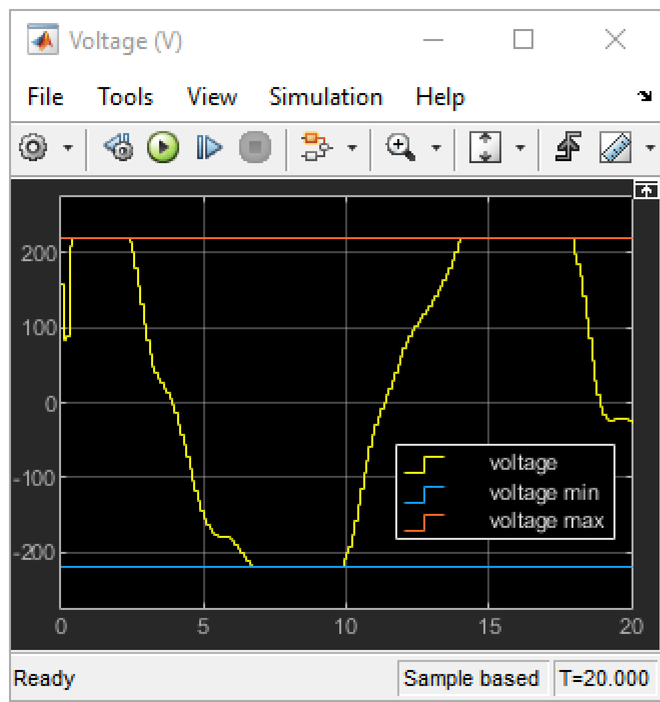

### Validate Controller Response

To validate the performance of the trained agent, simulate the model and view the response in the Scope blocks. The reinforcement learning agent is able to track the reference angle while satisfying the constraints on torque and voltage.

```
sim(mdl);
```


Angle (radians)
File Tools View Simulation Help
angle
angle ref
3
2
1
0
-1
-2
-3
0 5 10 15 20
Ready
Sample based
T=20.000


Torque (Nm)
File Tools View Simulation Help
torque ref
torque
torque min
torque max
80
60
40
20
0
-20
-40
-60
-80
0 5 10 15 20
Ready
Sample based
T=20.000




*Copyright 2021-2023 The MathWorks, Inc..*

## See Also

**Functions**
`generateRewardFunction` | `train` | `rlSimulinkEnv`

**Objects**
`rlTD3Agent` | `rlTD3AgentOptions` | `rlTrainingOptions`

**Blocks**
RL Agent

## Related Examples

- “DC Servomotor with Constraint on Unmeasured Output” (Model Predictive Control Toolbox)
- “Train Biped Robot to Walk Using Reinforcement Learning Agents” on page 5-267
- “Generate Reward Function from a Model Verification Block for a Water Tank System” on page 5-327

## More About

- “Define Reward Signals” on page 2-14
- “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents” on page 3-44
- “Train Reinforcement Learning Agents” on page 5-3
- “What is Model Predictive Control?” (Model Predictive Control Toolbox)

# Generate Reward Function from a Model Verification Block for a Water Tank System

This example shows how to automatically generate a reward function from performance requirement defined in a Simulink® Design Optimization™ model verification block. You then use the generated reward function to train a reinforcement learning agent.

### Introduction

You can use the `generateRewardFunction` to generate a reward function for reinforcement learning, starting from performance constraints specified in a Simulink Design Optimization model verification block. The resulting reward signal is a sum of weighted penalties on constraint violations by the current state of the environment.

In this example, you will convert the cost and constraint specifications defined in a **Check Step Response Characteristics** block for a water tank system into a reward function. You then use the reward function and use it to train an agent to control the water tank.

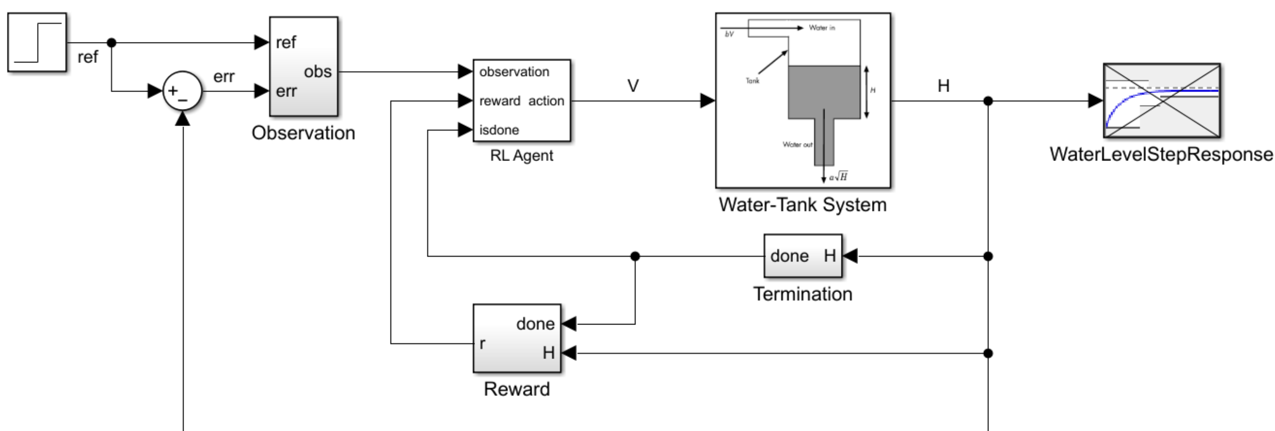
Specify parameters for this example.

```
% Watertank parameters
a = 2;
b = 5;
A = 20;

% Initial and final height
h0 = 1;
hf = 2;

% Simulation and sample times
Tf = 10;
Ts = 0.1;
```

The original model for this example is the “watertank Simulink Model” (Simulink Control Design).

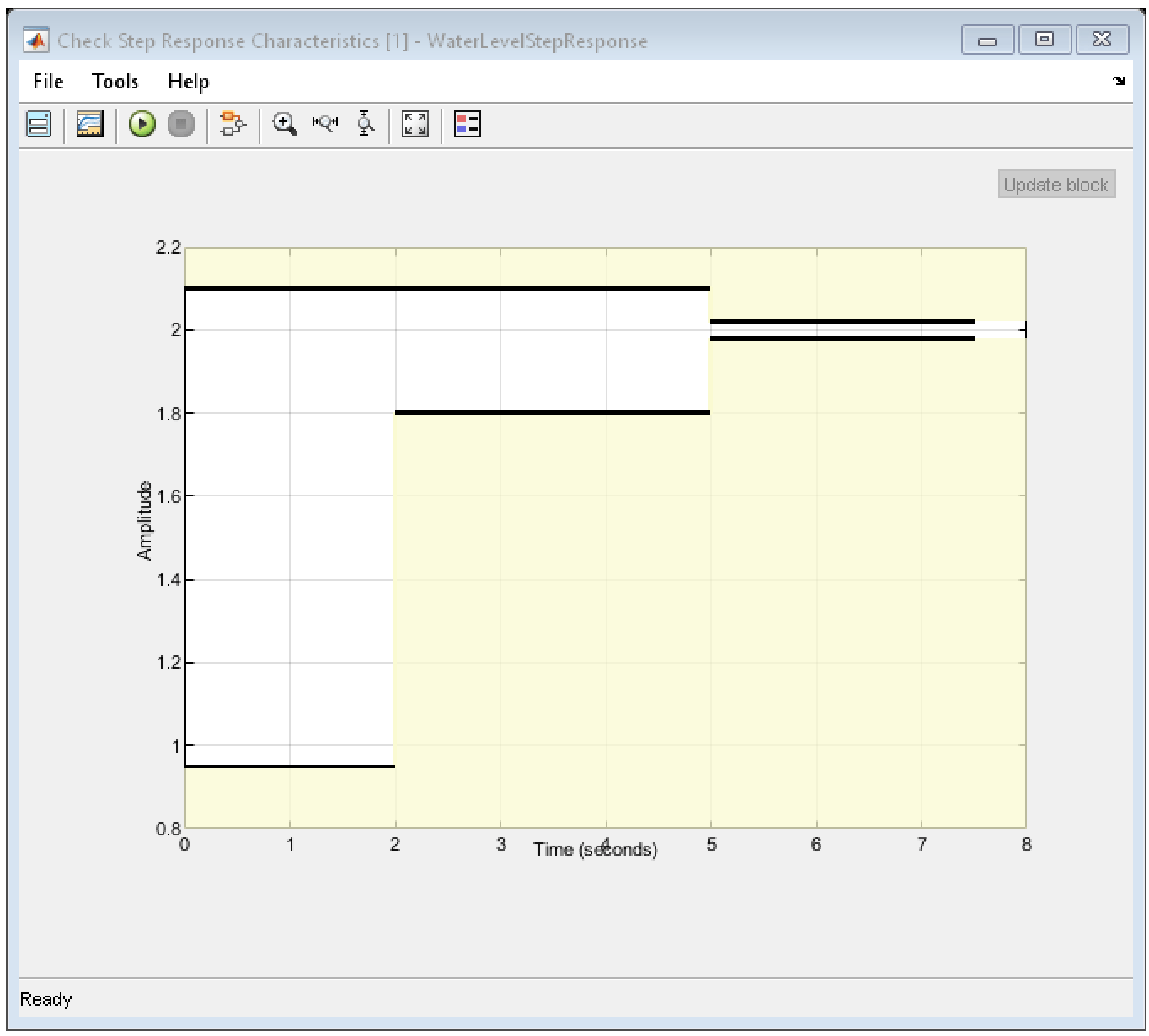
Open the model.

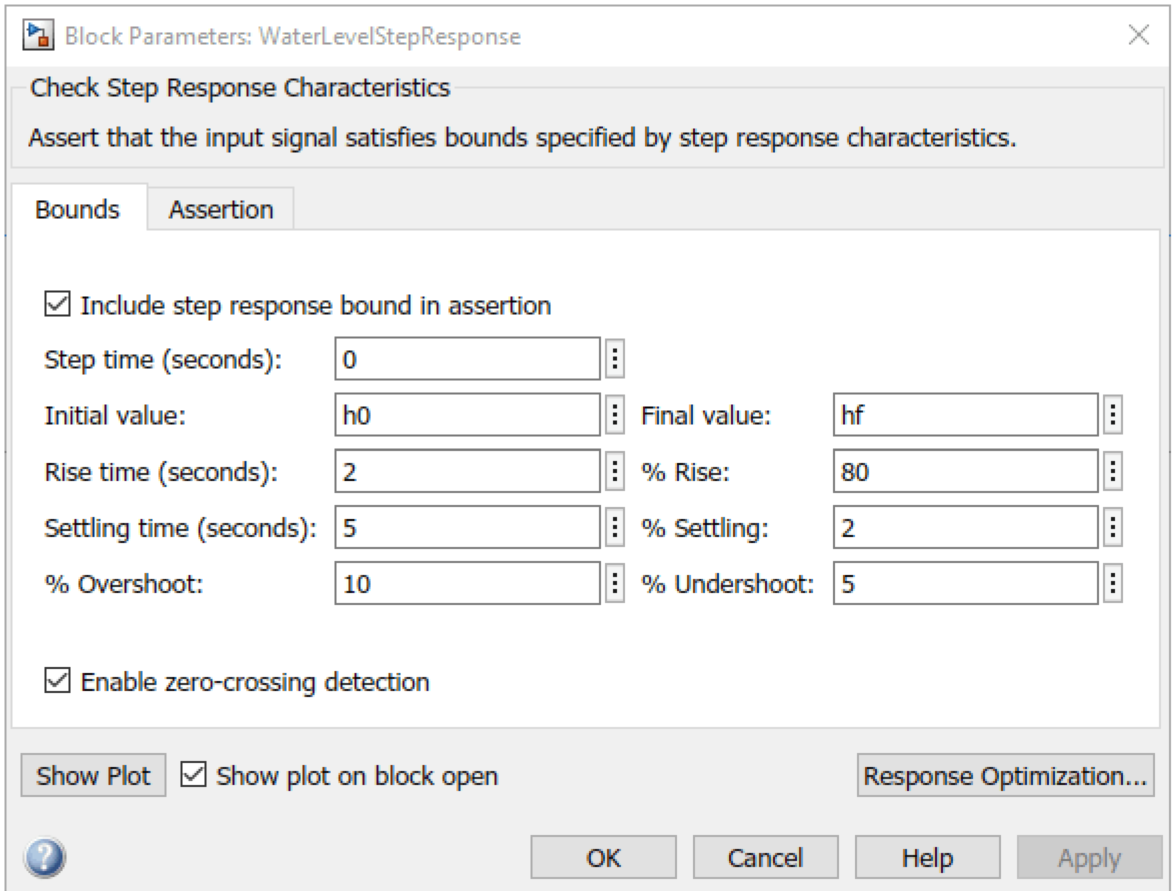
```
mdl = "rlWatertankStepInput";
open_system(mdl)
```




The model in this example has been modified for reinforcement learning. The goal is to control the level of the water in the tank using a reinforcement learning agent, while satisfying the response characteristics defined in the **Check Step Response Characteristics** block. Open the block to view the desired step response specifications.

```
verifBlk = mdl + "/WaterLevelStepResponse";
open_system(verifBlk)
```


Check Step Response Characteristics [1] - WaterLevelStepResponse
File
Tools
Help
Update block
2.2
2
1.8
1.6
1.4
1.2
1
0.8
Amplitude
0
1
2
3
4
5
6
7
8
Time (seconds)
Ready




### Generate a Reward Function

Generate the reward function code from specifications in the **WaterLevelStepResponse** block using `generateRewardFunction`. The code is displayed in the MATLAB Editor.

```
generateRewardFunction(blk)
```

The generated reward function is a starting point for reward design. You may modify the function by choosing different penalty functions and tuning the penalty weights. For this example, make the following change to the generated code:

- The default penalty weight is `1`. Set weight to `10`.
- The default exterior penalty function method is step. Change the method to `quadratic`.

After you make changes, the weight and penalty specifications should be as follows:

```
Weight = 10;
Penalty = sum(exteriorPenalty(x,Block1_xmin,Block1_xmax,'quadratic'));
```

For this example, the modified code has been saved in the MATLAB function file `rewardFunctionVfb.m`. Display the generated reward function.

```
type rewardFunctionVfb.m
```

```
function reward = rewardFunctionVfb(x,t)
% REWARDFUNCTION generates rewards from Simulink block specifications.
```

```
%
% x : Input of watertank_stepinput_rl/WaterLevelStepResponse
% t : Simulation time (s)

% Reinforcement Learning Toolbox
% 26-Apr-2021 13:05:16

%#codegen

%% Specifications from watertank_stepinput_rl/WaterLevelStepResponse
Block1_InitialValue = 1;
Block1_FinalValue = 2;
Block1_StepTime = 0;
Block1_StepRange = Block1_FinalValue - Block1_InitialValue;
Block1_MinRise = Block1_InitialValue + Block1_StepRange * 80/100;
Block1_MaxSettling = Block1_InitialValue + Block1_StepRange * (1+2/100);
Block1_MinSettling = Block1_InitialValue + Block1_StepRange * (1-2/100);
Block1_MaxOvershoot = Block1_InitialValue + Block1_StepRange * (1+10/100);
Block1_MinUndershoot = Block1_InitialValue - Block1_StepRange * 5/100;

if t >= Block1_StepTime
    if Block1_InitialValue <= Block1_FinalValue
        Block1_UpperBoundTimes = [0,5; 5,max(5+1,t+1)];
        Block1_UpperBoundAmplitudes = [Block1_MaxOvershoot,Block1_MaxOvershoot; Block1_MaxSettli
        Block1_LowerBoundTimes = [0,2; 2,5; 5,max(5+1,t+1)];
        Block1_LowerBoundAmplitudes = [Block1_MinUndershoot,Block1_MinUndershoot; Block1_MinRise
    else
        Block1_UpperBoundTimes = [0,2; 2,5; 5,max(5+1,t+1)];
        Block1_UpperBoundAmplitudes = [Block1_MinUndershoot,Block1_MinUndershoot; Block1_MinRise
        Block1_LowerBoundTimes = [0,5; 5,max(5+1,t+1)];
        Block1_LowerBoundAmplitudes = [Block1_MaxOvershoot,Block1_MaxOvershoot; Block1_MaxSettli
    end

    Block1_xmax = zeros(1,size(Block1_UpperBoundTimes,1));
    for idx = 1:numel(Block1_xmax)
        tseg = Block1_UpperBoundTimes(idx,:);
        xseg = Block1_UpperBoundAmplitudes(idx,:);
        Block1_xmax(idx) = interp1(tseg,xseg,t,'linear',NaN);
    end
    if all(isnan(Block1_xmax))
        Block1_xmax = Inf;
    else
        Block1_xmax = max(Block1_xmax,[],'omitnan');
    end

    Block1_xmin = zeros(1,size(Block1_LowerBoundTimes,1));
    for idx = 1:numel(Block1_xmin)
        tseg = Block1_LowerBoundTimes(idx,:);
        xseg = Block1_LowerBoundAmplitudes(idx,:);
        Block1_xmin(idx) = interp1(tseg,xseg,t,'linear',NaN);
    end
    if all(isnan(Block1_xmin))
        Block1_xmin = -Inf;
    else
        Block1_xmin = max(Block1_xmin,[],'omitnan');
    end
else
    Block1_xmin = -Inf;
```

```
    Block1_xmax = Inf;
end

%% Penalty function weight (specify nonnegative)
Weight = 10;

%% Compute penalty
% Penalty is computed for violation of linear bound constraints.
%
% To compute exterior bound penalty, use the exteriorPenalty function and
% specify the penalty method as 'step' or 'quadratic'.
%
% Alternaltely, use the hyperbolicPenalty or barrierPenalty function for
% computing hyperbolic and barrier penalties.
%
% For more information, see help for these functions.
Penalty = sum(exteriorPenalty(x,Block1_xmin,Block1_xmax,'quadratic'));

%% Compute reward
reward = -Weight * Penalty;
end
```

To integrate this reward function in the water tank model, open the MATLAB Function block under the Reward Subsystem.

```
open_system(mdl + "/Reward/Reward Function")
```

Append the function with the following line of code and save the model.

```
r = rewardFunctionVfb(x,t);
```

The MATLAB Function block will now execute `rewardFunctionVfb.m` for computing rewards.

For this example, the MATLAB Function block has already been modified and saved.

### Create a Reinforcement Learning Environment

The environment dynamics are modeled in the Water-Tank Subsystem. For this environment,

- The observations are the reference height `ref` from the last 5 time steps, and the height error is `err` = `ref` - `H`.
- The action is the voltage `V` applied to the pump.
- The sample time `Ts` is `0.1` s.

Create observation and action specifications for the environment.

```
numObs = 6;
numAct = 1;
oinfo = rlNumericSpec([numObs 1]);
ainfo = rlNumericSpec([numAct 1]);
```

Create the reinforcement learning environment using the `rlSimulinkEnv` function.

```
agentBlk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl, agentBlk, oinfo, ainfo);
```

### Create a Reinforcement Learning Agent

Fix the random seed for reproducibility.

```
rng(100)
```

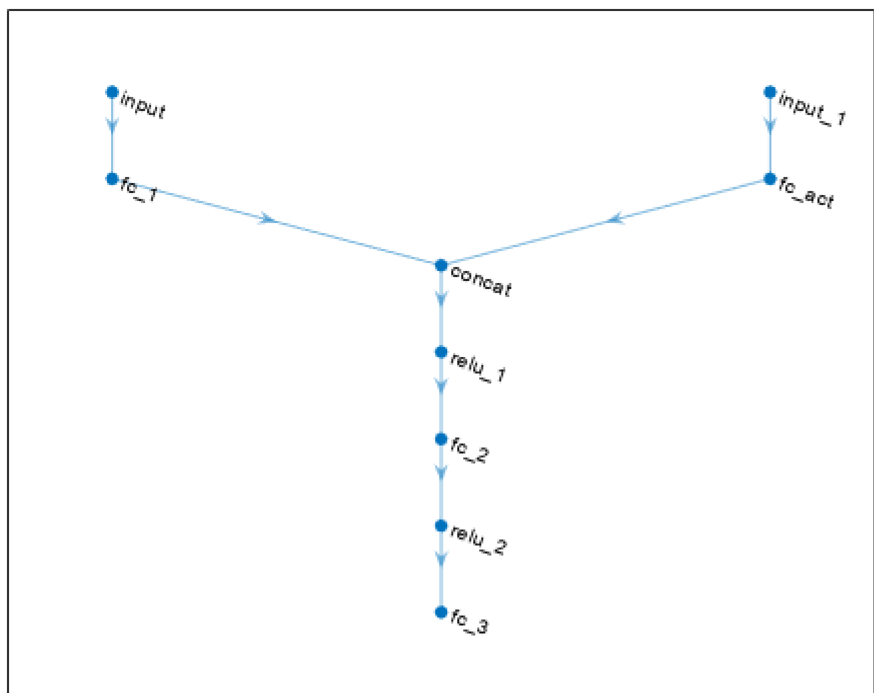
The agent used in this example is a twin-delayed deep deterministic policy gradient (TD3) agent. TD3 agents use two parametrized Q-value function approximators to estimate the value (that is the expected cumulative long-term reward) of the policy. To model the parametrized Q-value function within both critics, use a neural network with two inputs (the observation and action) and one output (the value of the policy when taking a given action from the state corresponding to a given observation). For more information on TD3 agents, see “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents” on page 3-44.

Define each network path as an array of layer objects. Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
mainPath = [
    featureInputLayer(numObs)
    fullyConnectedLayer(128)
    concatenationLayer(1, 2, Name="concat")
    reluLayer()
    fullyConnectedLayer(128)
    reluLayer()
    fullyConnectedLayer(1)];
actionPath = [
    featureInputLayer(numAct)
    fullyConnectedLayer(8, Name="fc_act")];

% Create layerGraph object and add layers
criticNet = layerGraph(mainPath);
criticNet = addLayers(criticNet, actionPath);

% Connect Layers
criticNet = connectLayers(criticNet, "fc_act", "concat/in2");
```

Plot the critic network structure.

```
plot(criticNet);
```




Create the critic function objects using `rlQValueFunction`. The critic function object encapsulates the critic by wrapping around the critic deep neural network. To make sure the critics have different initial weights, explicitly initialize each network before using them to create the critics.

```
% Convert the neural network to a dlnetwork object without initializing the networks
criticdlNet = dlnetwork(criticNet, Initialize=false);

% Create the two critic functions for the TD3 agent
critic1 = rlQValueFunction(initialize(criticdlNet), oinfo, ainfo);
critic2 = rlQValueFunction(initialize(criticdlNet), oinfo, ainfo);
```

TD3 agents learn a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by obsInfo) and one output layer (which returns the action to the environment action channel, as specified by `ainfo`).

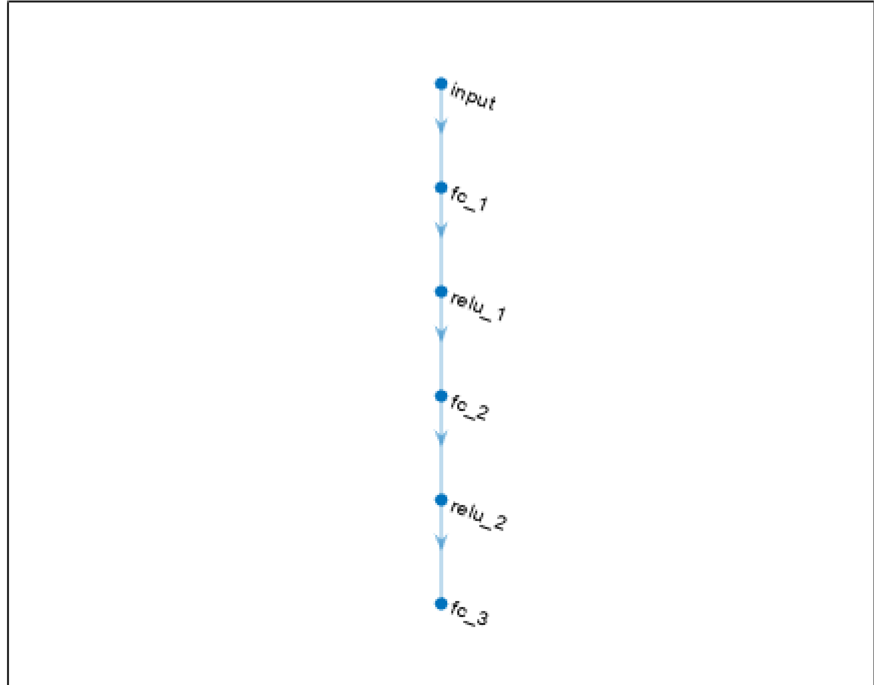
Define the network as an array of layer objects.

```
actorNet = [
    featureInputLayer(numObs)
    fullyConnectedLayer(128)
    reluLayer()
    fullyConnectedLayer(128)
```

```
    reluLayer()
    fullyConnectedLayer(numAct)];
```

Plot the actor network.

```
plot(layerGraph(actorNet));
```




Create a deterministic actor function that is responsible for modeling the policy of the agent. For more information, see `rlContinuousDeterministicActor`.

```
actordlNet = dlnetwork(actorNet);
actor = rlContinuousDeterministicActor(actordlNet, oinfo, ainfo);
```

Specify the agent options using `rlTD3AgentOptions`. The agent trains from an experience buffer of maximum capacity `1e6` by randomly selecting mini-batches of size `256`. The discount factor of `0.99` favors long-term rewards.

```
agentOpts = rlTD3AgentOptions( ...
    SampleTime=Ts, ...
    DiscountFactor=0.99, ...
    ExperienceBufferLength=1e6, ...
    MiniBatchSize=256);
```

The exploration model in this TD3 agent is Gaussian. The noise model adds a uniform random value to the action during training. Set the standard deviation of the noise to `0.5`. The standard deviation decays at the rate of `1e-5` every agent step until the minimum value of `0`.

```
agentOpts.ExplorationModel.StandardDeviation = 0.5;
agentOpts.ExplorationModel.StandardDeviationDecayRate = 1e-5;
agentOpts.ExplorationModel.StandardDeviationMin = 0;
```

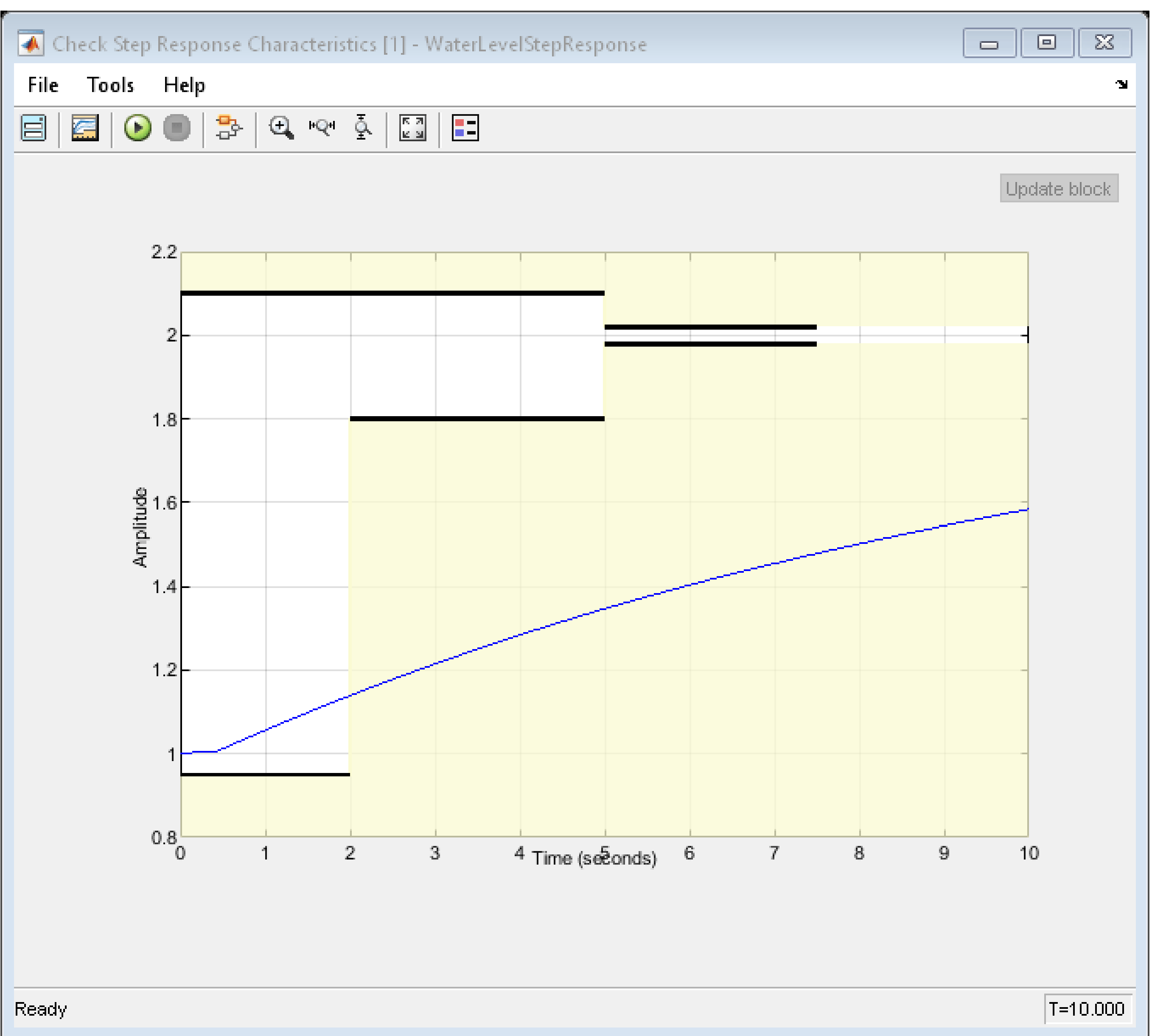
Create the TD3 agent using the actor and critic representations. For more information on TD3 agents, see `rlTD3Agent`.

```
agent = rlTD3Agent(actor, [critic1, critic2], agentOpts);
```

#### Closed Loop Response

Simulate the model to view the closed loop step response of the untrained agent.

```
sim(mdl);
```



### Train the Agent

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

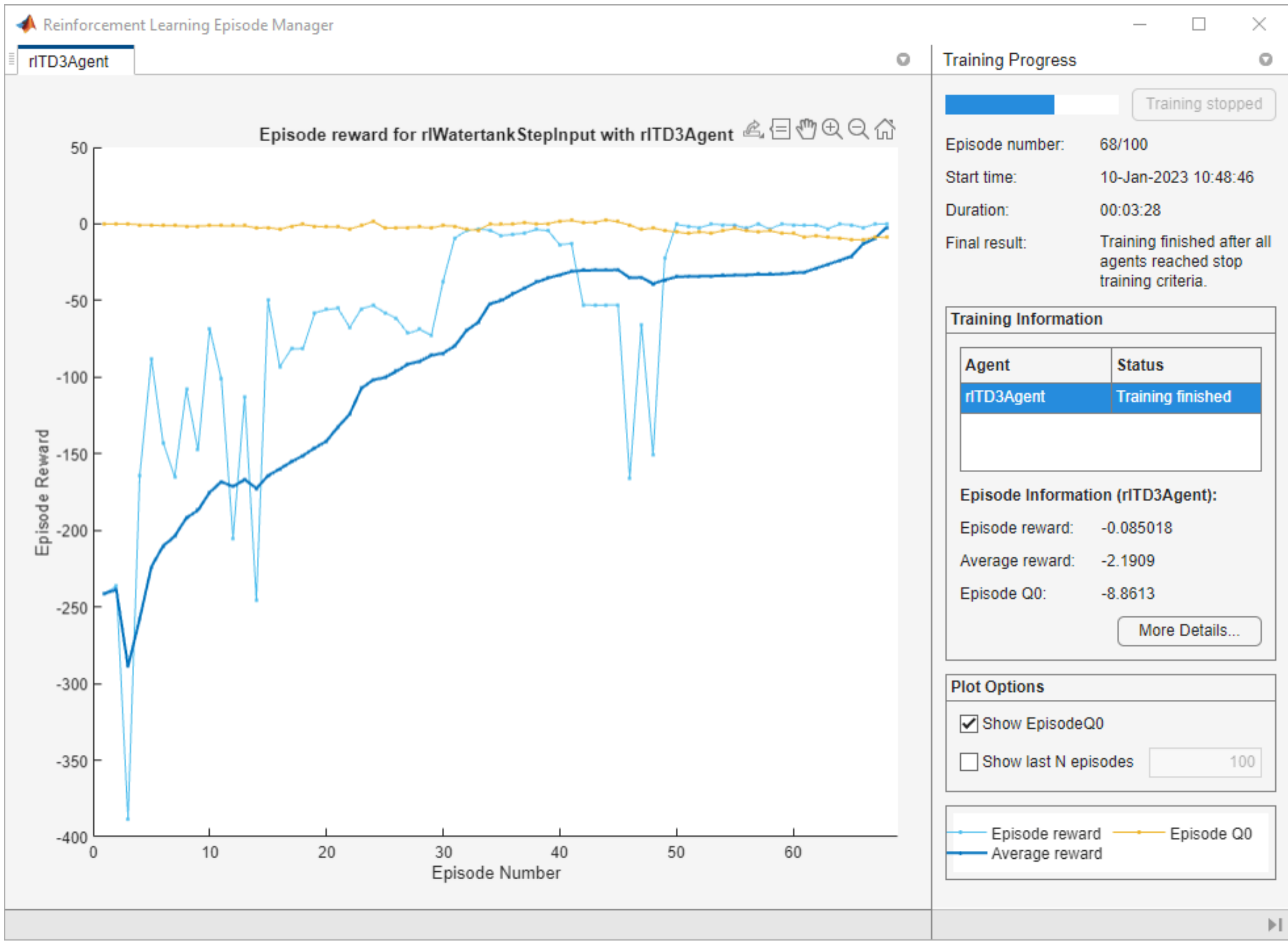
- Run training for at most 100 episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps, where the total simulation time `Tf` is `10` s.
- Stop the training when the agent receives an average cumulative reward greater than `-5` over `20` consecutive episodes. At this point, the agent can track the reference height.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=100, ...
    MaxStepsPerEpisode=ceil(Tf/Ts), ...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-5,...
    ScoreAveragingWindowLength=20);
```

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

```
doTraining = false;
if doTraining
    trainingStats = train(agent, env, trainOpts);
else
    load("rlWatertankTD3Agent.mat")
end
```

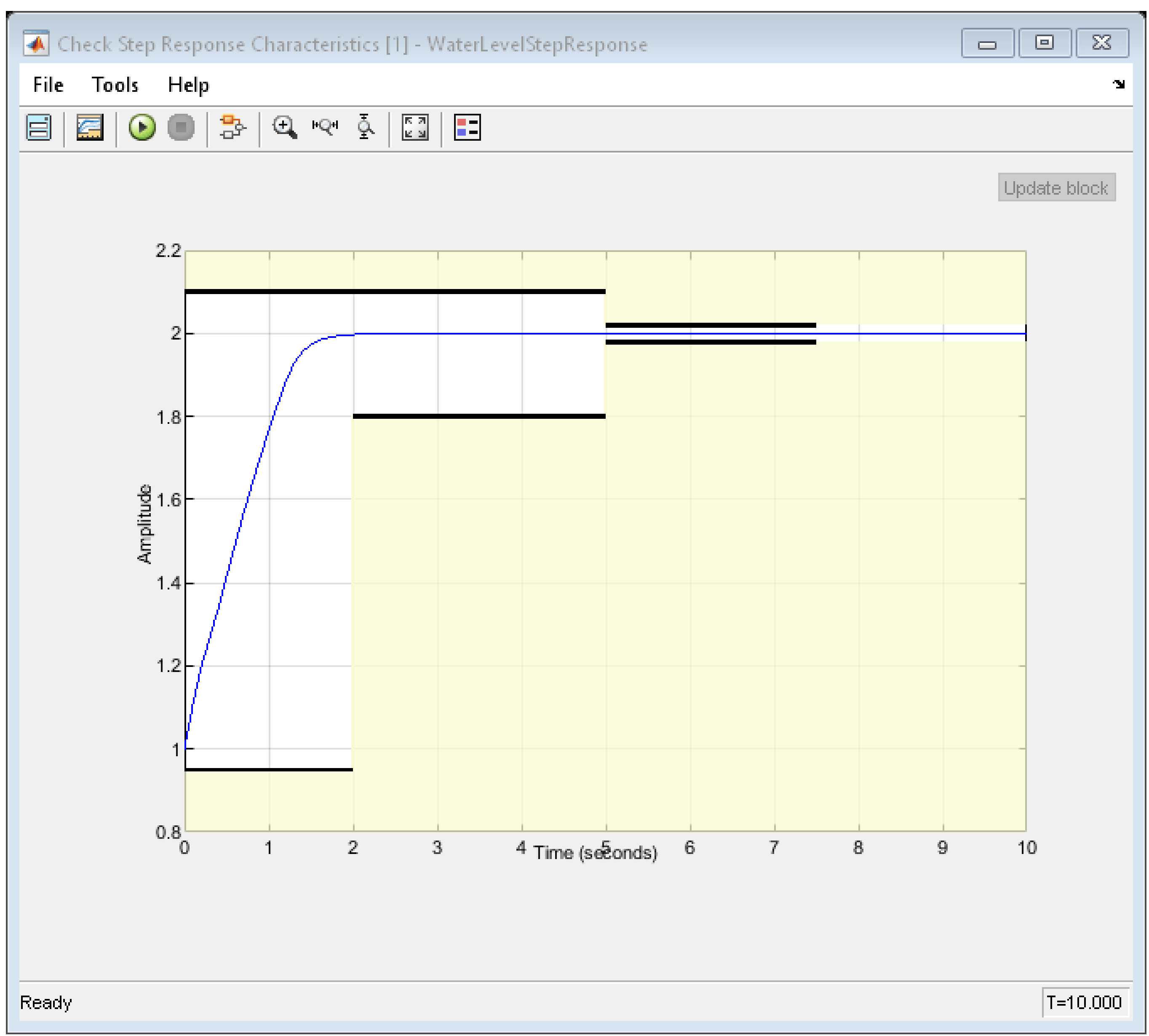
A snapshot of the training progress is shown in the following figure. You can expect different results due to inherent randomness in the training process.




## Closed Loop Response of Trained Agent

Simulate the model to view the closed loop step response. The reinforcement learning agent is able to track the reference height while satisfying the step response constraints.

```
sim(mdl);
```




## See Also

**Functions**
`generateRewardFunction` | `train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlTD3Agent` | `rlTD3AgentOptions` | `rlTrainingOptions`

**Blocks**
RL Agent

## Related Examples


- "Tune PI Controller Using Reinforcement Learning" on page 5-392
- "Train Biped Robot to Walk Using Reinforcement Learning Agents" on page 5-267
- "Generate Reward Function from a Model Predictive Controller for a Servomotor" on page 5-315


## More About


- "Define Reward Signals" on page 2-14
- "Water Tank Reinforcement Learning Environment Model" on page 2-55
- "Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents" on page 3-44
- "Train Reinforcement Learning Agents" on page 5-3

# Train TD3 Agent for PMSM Control

This example demonstrates speed control of a permanent magnet synchronous motor (PMSM) using a twin delayed deep deterministic policy gradient (TD3) agent.

The goal of this example is to show that you can use reinforcement learning as an alternative to linear controllers, such as PID controllers, to control the speed of PMSM systems. Linear controllers often do not produce good tracking performance outside the region in which the plant can be approximated with a linear system. In such cases, reinforcement learning provides a nonlinear control alternative.

Load the parameters for this example.

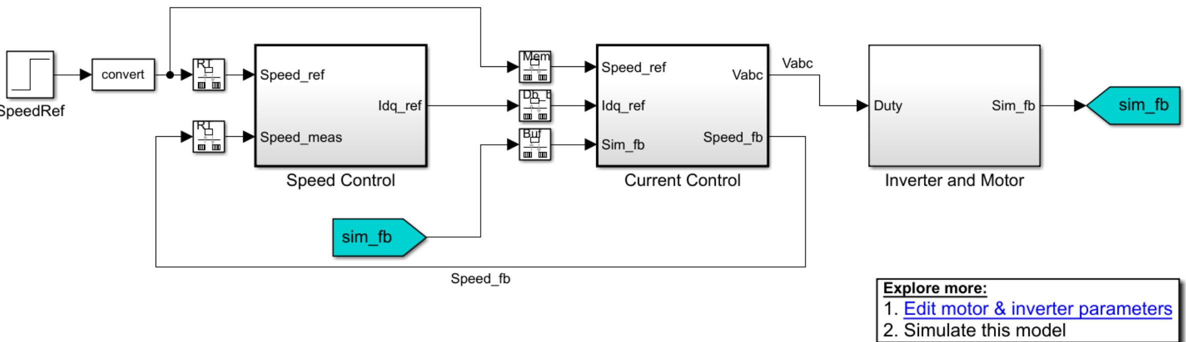
```
sim_data
```

```
### The Lq is observed to be lower than Ld. ###
### Using the lower of these two for the Ld (internal variable)  ###
### and higher of these two for the Lq (internal variable) for computations. ###
### The Lq is observed to be lower than Ld. ###
### Using the lower of these two for the Ld (internal variable)  ###
### and higher of these two for the Lq (internal variable) for computations. ###
             model: 'Maxon-645106'
                sn: '2295588'
                 p: 7
                Rs: 0.2930
                Ld: 8.7678e-05
                Lq: 7.7724e-05
                Ke: 5.7835
                 J: 8.3500e-05
                 B: 7.0095e-05
           I_rated: 7.2600
          QEPSlits: 4096
            N_base: 3476
             N_max: 4300
            FluxPM: 0.0046
           T_rated: 0.3471
    PositionOffset: 0.1650

               model: 'BoostXL-DRV8305'
                  sn: 'INV_XXXX'
                V_dc: 24
              I_trip: 10
              Rds_on: 0.0020
              Rshunt: 0.0070
       CtSensAOffset: 2295
       CtSensBOffset: 2286
       CtSensCOffset: 2295
             ADCGain: 1
         EnableLogic: 1
        invertingAmp: 1
          ISenseVref: 3.3000
    ISenseVoltPerAmp: 0.0700
           ISenseMax: 21.4286
             R_board: 0.0043
     CtSensOffsetMax: 2500
     CtSensOffsetMin: 1500
```

```
               model: 'LAUNCHXL-F28379D'
                  sn: '123456'
       CPU_frequency: 200000000
       PWM_frequency: 5000
  PWM_Counter_Period: 20000
            ADC_Vref: 3
        ADC_MaxCount: 4095
       SCI_baud_rate: 12000000

  V_base: 13.8564
  I_base: 21.4286
  N_base: 3476
  T_base: 1.0249
  P_base: 445.3845
```

Open the Simulink model.

```
mdl = "mcb_pmsm_foc_sim_RL";
open_system(mdl)
```

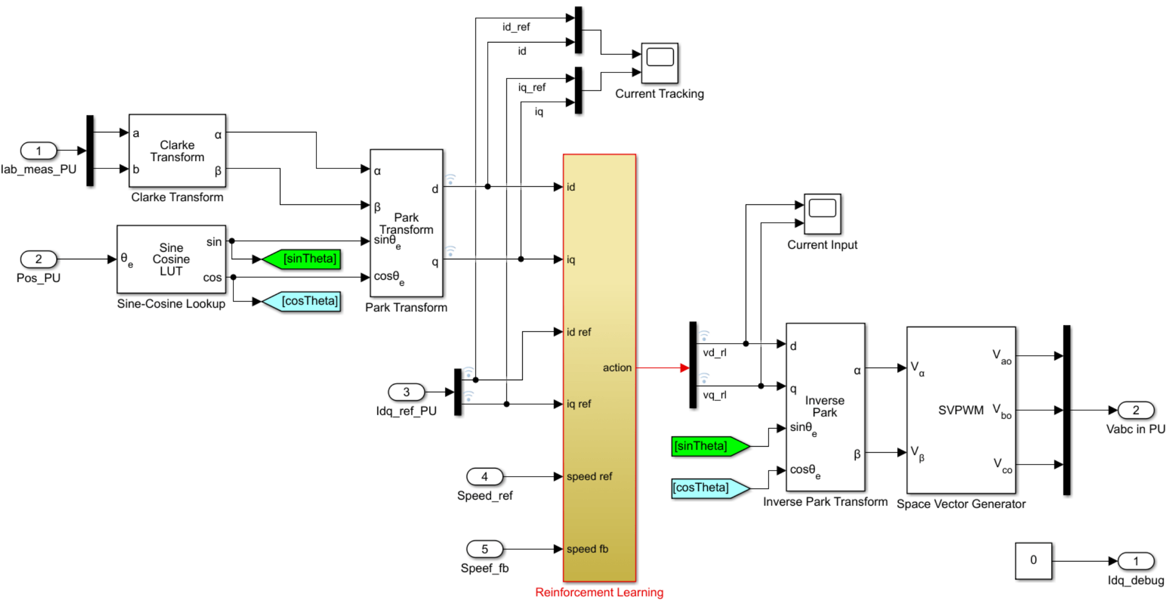



In a linear control version of this example, you can use PI controllers in both the speed and current control loops. An outer-loop PI controller can control the speed while two inner-loop PI controllers control the d-axis and q-axis currents. The overall goal is to track the reference speed in the `Speed_Ref` signal. This example uses a reinforcement learning agent to control the currents in the inner control loop while a PI controller controls the outer loop.

### Create Environment Interface

The environment in this example consists of the PMSM system, excluding the inner-loop current controller, which is the reinforcement learning agent. To view the interface between the reinforcement learning agent and the environment, open the Closed Loop Control subsystem.

```
open_system(mdl + ...
    "/Current Control/Control_System/Closed Loop Control")
```




The Reinforcement Learning subsystem contains an **RL Agent** block, the creation of the observation vector, and the reward calculation.

For this environment:

- The observations are the outer-loop reference speed `Speed_ref`, speed feedback `Speed_fb`, d-axis and q-axis currents and errors (id, iq, $\mathrm{id}_{\mathrm{error}}$ and $\mathrm{iq}_{\mathrm{error}}$), and the error integrals.
- The actions from the agent are the voltages `vd_rl` and `vq_rl`.
- The sample time of the agent is 2e-4 seconds. The inner-loop control occurs at a different sample time than the outer loop control.
- The simulation runs for 5000 time steps unless it is terminated early when the $\mathrm{iq}_{\mathrm{ref}}$ signal is saturated at 1.
- The reward at each time step is:

$$r_t = -\left(Q_1 * \mathrm{id}_{\mathrm{error}}{}^2 + Q_2 * \mathrm{iq}_{\mathrm{error}}{}^2 + R * \sum_j {u^j}_{t-1}{}^2\right) - 100d$$

Here, $Q_1 = Q_2 = 5$, and $R = 0.1$ are constants, $\mathrm{id}_{\mathrm{error}}$ is the d-axis current error, $\mathrm{iq}_{\mathrm{error}}$ is the q-axis current error, ${u^j}_{t-1}$ are the actions from the previous time step, and $d$ is a flag that is equal to 1 when the simulation is terminated early.

Create the observation and action specifications for the environment. For information on creating continuous specifications, see `rlNumericSpec`.

```
% Create observation specifications.
numObs = 8;
obsInfo = rlNumericSpec( ...
    [numObs 1], ...
    DataType=dataType);
obsInfo.Name = "observations";
obsInfo.Description = "Error and reference signal";
```

```
% Create action specifications.
numAct = 2;
actInfo = rlNumericSpec([numAct 1], "DataType", dataType);
actInfo.Name = "vqdRef";
```

Create the Simulink environment interface using the observation and action specifications. For more information on creating Simulink environments, see `rlSimulinkEnv`.

```
agentblk = "mcb_pmsm_foc_sim_RL/Current Control/" + ...
   "Control_System/Closed Loop Control/" + ...
   "Reinforcement Learning/RL Agent";

env = rlSimulinkEnv(mdl, agentblk, obsInfo, actInfo);
```

Provide a reset function for this environment using the `ResetFcn` parameter. At the beginning of each training episode, the `resetPMSM` function randomly initializes the final value of the reference speed in the **SpeedRef** block to `695.4` rpm (`0.2` pu), `1390.8` rpm (`0.4` pu), `2086.2` rpm (`0.6` pu), or `2781.6` rpm (`0.8` pu).

```
env.ResetFcn = @resetPMSM;
```

### Create Agent

The agent used in this example is a twin-delayed deep deterministic policy gradient (TD3) agent. TD3 agents use two parametrized Q-value function approximators to estimate the value (that is the expected cumulative long-term reward) of the policy.

To model the parametrized Q-value function within both critics, use a neural network with two inputs (the observation and action) and one output (the value of the policy when taking a given action from the state corresponding to a given observation). For more information on TD3 agents, see "Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents" on page 3-44.

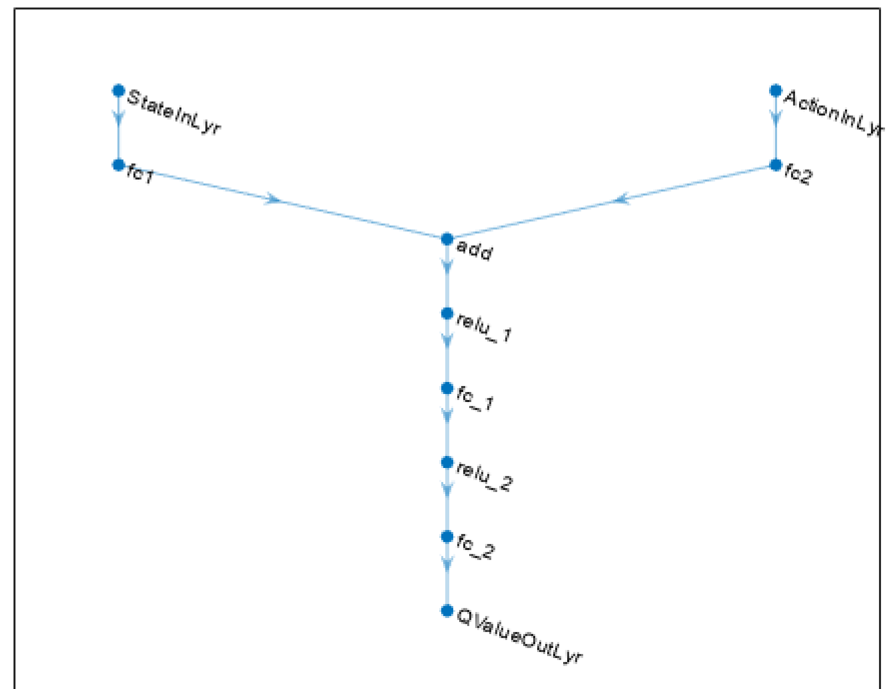
Define each network path as an array of layer objects. Assign names to the input and output layers of each path. These names allow you to connect the paths and then later explicitly associate the network input and output layers with the appropriate environment channel.

```
% State input path
statePath = [
    featureInputLayer(numObs, Name="StateInLyr")
    fullyConnectedLayer(64, Name="fc1")
    ];

% Action input path
actionPath = [
    featureInputLayer(numAct, Name="ActionInLyr")
    fullyConnectedLayer(64, Name="fc2")
    ];

% Common output path
commonPath = [additionLayer(2, Name="add")
    reluLayer
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(16)
    fullyConnectedLayer(1, Name="QValueOutLyr")
    ];
```

```
% Add layers to layergraph object
criticNet = layerGraph();
criticNet = addLayers(criticNet, statePath);
criticNet = addLayers(criticNet, actionPath);
criticNet = addLayers(criticNet, commonPath);

% Connect layers
criticNet = connectLayers(criticNet, "fc1", "add/in1");
criticNet = connectLayers(criticNet, "fc2", "add/in2");
```

Plot the critic network structure.

```
plot(criticNet);
```




Convert the neural network to a `dlnetwork` object without initializing the networks.

```
criticDLNet = dlnetwork(criticNet, Initialize=false);
```

The actor and critic networks are initialized randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

Create the critic objects using `rlQValueFunction`. The critics are function approximator objects that use the deep neural network as approximation model. To make sure the critics have different initial weights, explicitly initialize each network before using them to create a critic.

```
critic1 = rlQValueFunction(initialize(criticDLNet),obsInfo, actInfo);
critic2 = rlQValueFunction(initialize(criticDLNet),obsInfo, actInfo);
```

TD3 agents use a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor. This actor takes the current observation as input and returns as output an action that is a deterministic function of the observation.

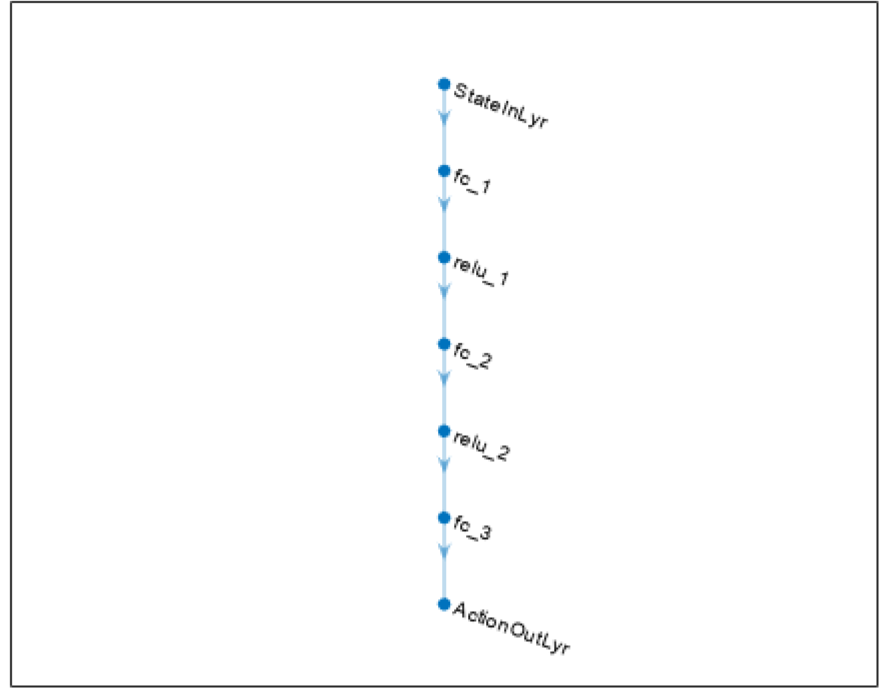
To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by obsInfo) and one output layer (which returns the action to the environment action channel, as specified by actInfo). For more information on creating approximator objects such as actors and critics, see “Create Policies and Value Functions” on page 4-2.

Define the network as an array of layer objects.

```
actorNet = [
    featureInputLayer(numObs, Name="StateInLyr")
    fullyConnectedLayer(64)
    reluLayer
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(numAct)
    tanhLayer(Name="ActionOutLyr")
    ];
```

Plot the actor network.

```
plot(layerGraph(actorNet));
```



Convert the network to a `dlnetwork` object.

```
actordlNet = dlnetwork(actorNet);
```

Display the number of parameters.

```
summary(actordlNet)
```

```
   Initialized: true

   Number of learnables: 2.7k

   Inputs:
      1   'StateInLyr'   8 features
```

Create the actor using `rlContinuousDeterministicActor`, passing the actor network and the environment specifications as input arguments.

```
actor = rlContinuousDeterministicActor(actordlNet,obsInfo,actInfo);
```

To create the TD3 agent, first specify the agent options using an `rlTD3AgentOptions` object. Alternatively, you can create the agent first, and then access its option object and modify all the options using dot notation.

The agent trains from an experience buffer of maximum capacity 2e6 by randomly selecting mini-batches of size 512. Use a discount factor of 0.995 to favor long-term rewards. TD3 agents maintain time-delayed copies of the actor and critics known as the *target actor and critics*. Configure the targets to update every 10 agent steps during training with a smoothing factor of 0.005.

```
Ts_agent = Ts;
agentOpts = rlTD3AgentOptions( ...
    SampleTime=Ts_agent, ...
    DiscountFactor=0.995, ...
    ExperienceBufferLength=2e6, ...
    MiniBatchSize=512, ...
    NumStepsToLookAhead=1, ...
    TargetSmoothFactor=0.005, ...
    TargetUpdateFrequency=10);
```

Specify training options for the critics and the actor using `rlOptimizerOptions`. For this example, set a learning rate of 1e-3 and 1e-4 for the actor and critics respectively.

```
% Critic optimizer options
for idx = 1:2
    agentOpts.CriticOptimizerOptions(idx).LearnRate = 1e-4;
    agentOpts.CriticOptimizerOptions(idx).GradientThreshold = 1;
end

% Actor optimizer options
agentOpts.ActorOptimizerOptions.LearnRate = 1e-3;
agentOpts.ActorOptimizerOptions.GradientThreshold = 1;
agentOpts.ActorOptimizerOptions.L2RegularizationFactor = 1e-3;
```

During training, the agent explores the action space using a Gaussian action noise model. Set the noise variance and decay rate using the `ExplorationModel` property. The noise variance decays at the rate of `2e-4`, which favors exploration towards the beginning of training and exploitation in later stages. For more information on the noise model, see `rlTD3AgentOptions`.

```
agentOpts.ExplorationModel.Variance = 0.05;
agentOpts.ExplorationModel.VarianceDecayRate = 2e-4;
agentOpts.ExplorationModel.VarianceMin = 0.001;
```

The agent also uses a Gaussian action noise model for smoothing the target policy updates. Specify the variance and decay rate for this model using the `TargetPolicySmoothModel` property.

```
agentOpts.TargetPolicySmoothModel.Variance = 0.1;
agentOpts.TargetPolicySmoothModel.VarianceDecayRate = 1e-4;
```

Create the agent using the specified actor, critics, and options.

```
agent = rlTD3Agent(actor, [critic1,critic2], agentOpts);
```

#### Train Agent

To train the agent, first specify the training options using `rlTrainingOptions`. For this example, use the following options.

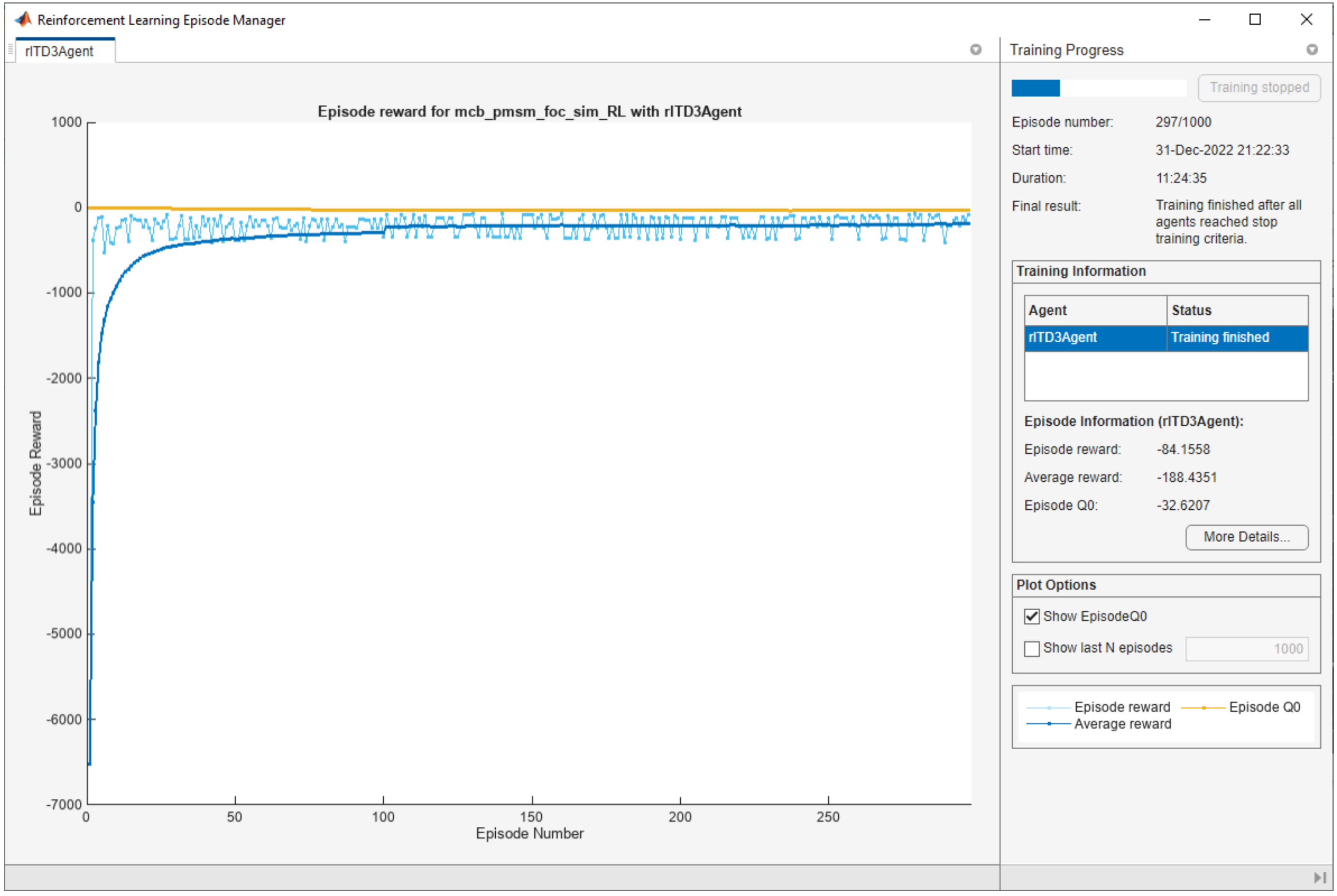
- Run each training for at most 1000 episodes, with each episode lasting at most `ceil(T/Ts_agent)` time steps.
- Stop training when the agent receives an average cumulative reward greater than -190 over 100 consecutive episodes. At this point, the agent can track the reference speeds.

```
T = 1.0;
maxepisodes = 1000;
maxsteps = ceil(T/Ts_agent);
trainOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes, ...
    MaxStepsPerEpisode=maxsteps, ...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-190,...
    ScoreAveragingWindowLength=100);
```

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

```
doTraining = false;
if doTraining
    trainResult = train(agent, env, trainOpts);
else
    load("rlPMSMAgent.mat","agent")
end
```

A snapshot of the training progress is shown in the following figure. You can expect different results due to randomness in the training process.




### Simulate Agent

To validate the performance of the trained agent, simulate the model and view the closed-loop performance through the **Speed Tracking Scope** block.

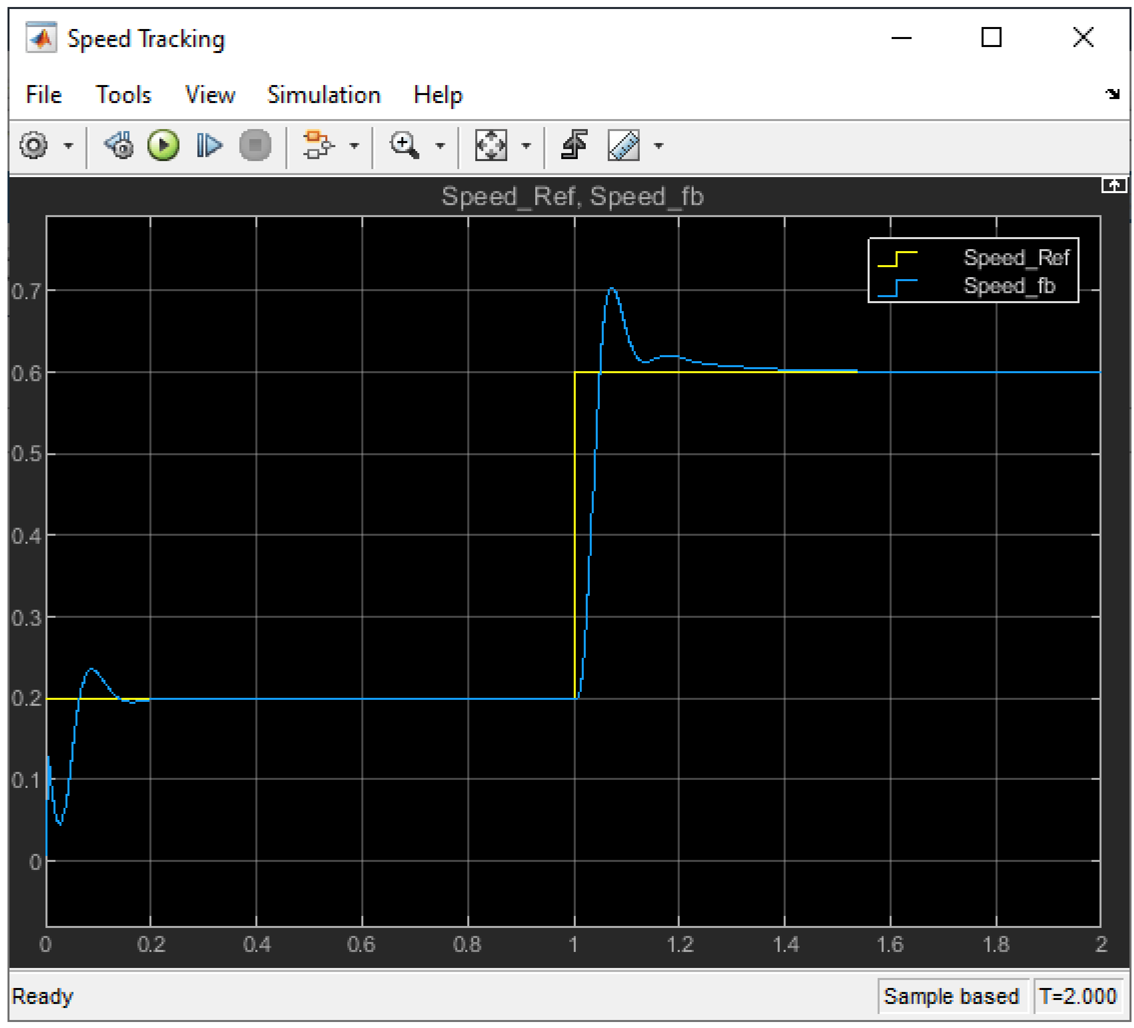
```
sim(mdl);
```

You can also simulate the model at different reference speeds. Set the reference speed in the **SpeedRef** block to a different value between `0.2` and `1.0` per-unit and simulate the model again.

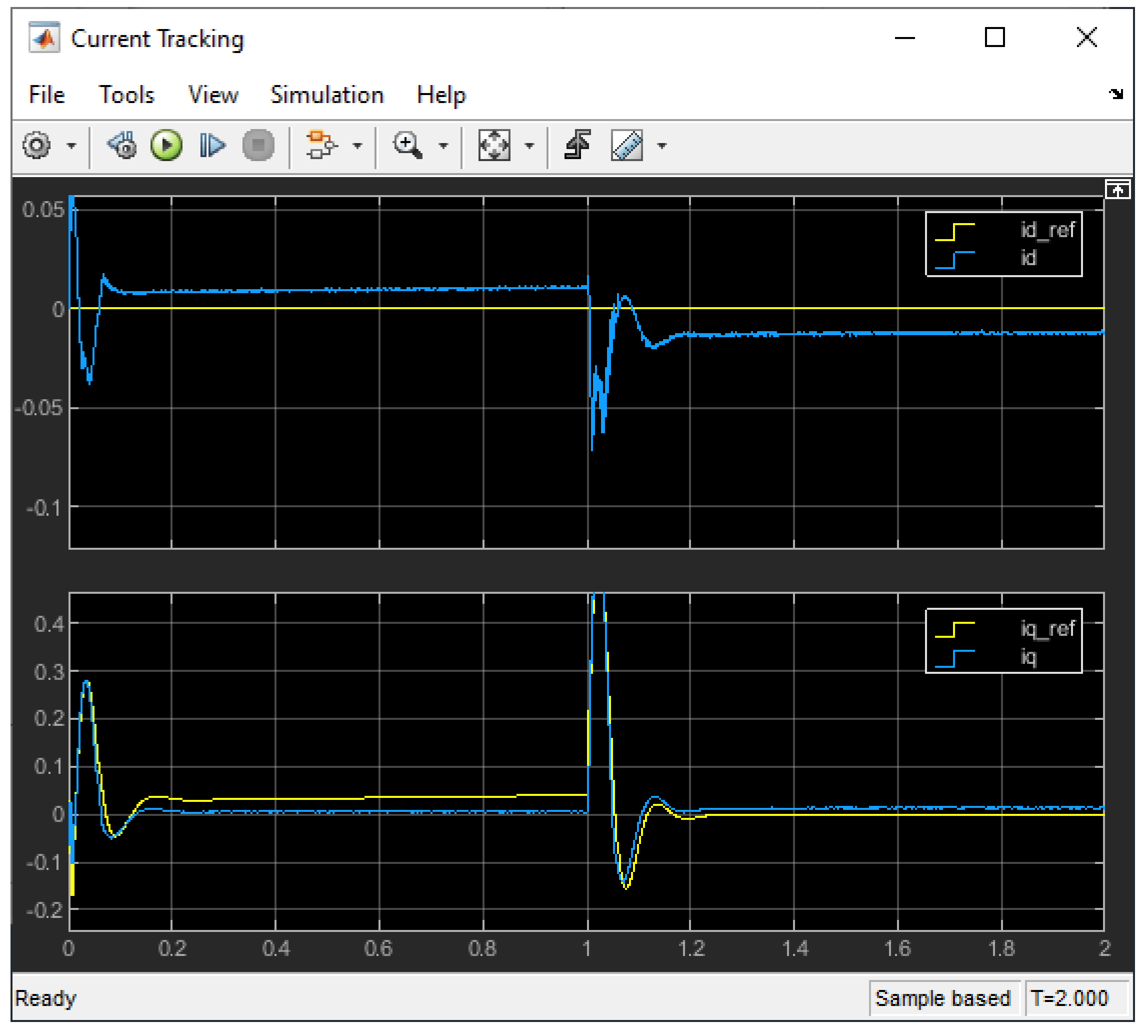
```
set_param("mcb_pmsm_foc_sim_RL/SpeedRef", "After", "0.6")
sim(mdl);
```

The following figure shows an example of closed-loop tracking performance. In this simulation, the reference speed steps through values of `695.4` rpm (`0.2` per-unit) and `1738.5` rpm (`0.5` pu). The PI and reinforcement learning controllers track the reference signal changes within `0.5` seconds.

Although the agent was trained to track the reference speed of `0.2` per-unit and not `0.5` per-unit, it was able to generalize well.




The following figure shows the corresponding current tracking performance. The agent was able to track the id and iq current references with steady-state error less than 2%.




## See Also

**Functions**
train | rlSimulinkEnv

**Objects**
rlTD3Agent | rlTD3AgentOptions | rlOptimizerOptions | rlTrainingOptions

**Blocks**
RL Agent

## Related Examples

- “Tune PI Controller Using Reinforcement Learning” on page 5-392
- “Train Biped Robot to Walk Using Reinforcement Learning Agents” on page 5-267

## More About

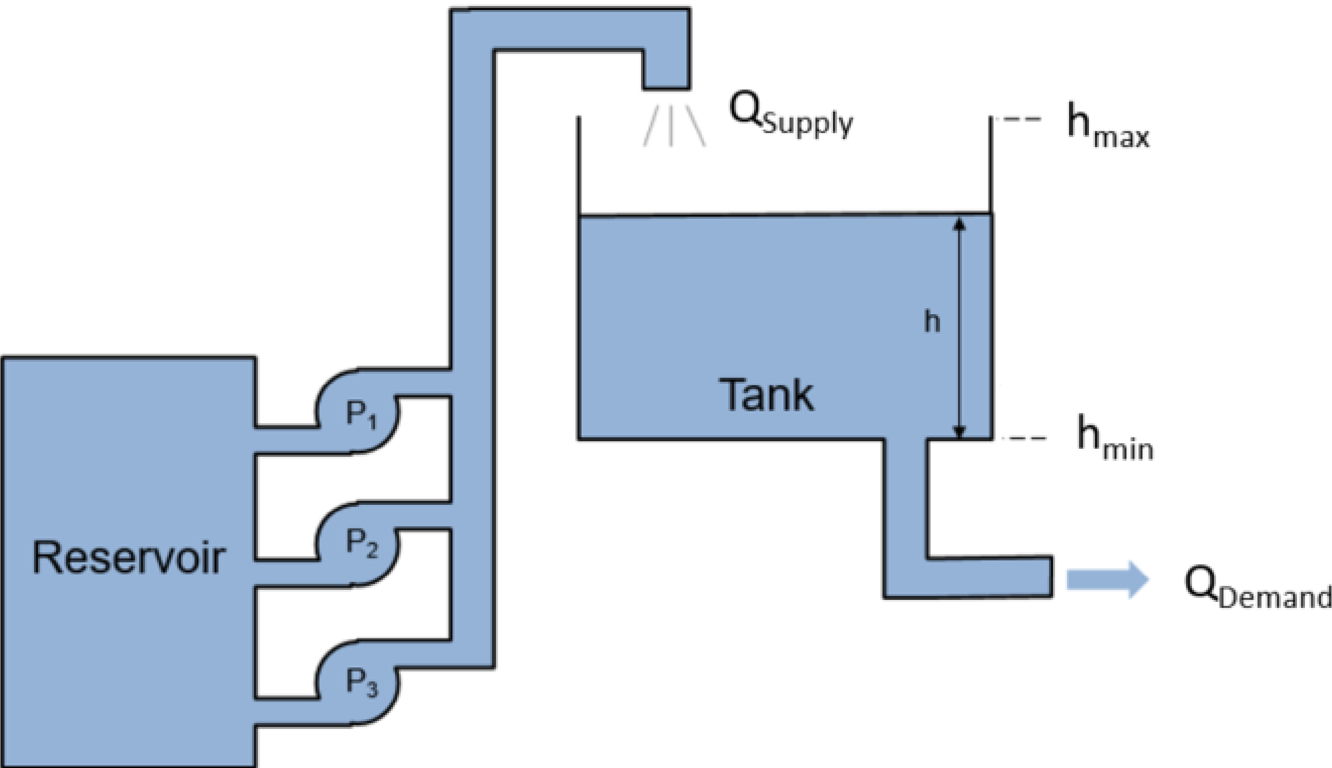
- "Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents" on page 3-44
- "Train Reinforcement Learning Agents" on page 5-3

# Water Distribution System Scheduling Using Reinforcement Learning

This example shows how to learn an optimal pump scheduling policy for a water distribution system using reinforcement learning (RL).

### Water Distribution System

The following figure shows a water distribution system.




Here:

- $Q_{\text{Supply}}$ is the amount of water supplied to the water tank from a reservoir.
- $Q_{\text{Demand}}$ is the amount of water flowing out of the tank to satisfy usage demand.

The objective of the reinforcement learning agent is to schedule the number of pumps running to both minimize the energy usage of the system and satisfy the usage demand ($h > 0$). The dynamics of the tank system are governed by the following equation.

$$A\frac{dh}{dt} = Q_{\text{Supply}}(t) - Q_{\text{Demand}}(t)$$

Here, $A = 40\, m^2$ and $h_{\max} = 7m$. The demand over a 24 hour period is a function of time given as

$$Q_{\text{Demand}}(t) = \mu(t) + \eta(t)$$

where $\mu(t)$is the expected demand and $\eta(t)$ represents the demand uncertainty, which is sampled from a uniform random distribution.

The supply is determined by the number of pumps running, $a \in \{0, 1, 2, 3\}$according to following mapping.

$$Q_{\text{Supply}}(t) = Q(a) = \begin{cases} 0 & a = 0 \\ 164 & a = 1 \\ 279 & a = 2 \\ 344 & a = 3 \end{cases} \frac{\text{cm}}{h}$$

To simplify the problem, power consumption is defined as the number of pumps running, $a$.

The following function is the reward for this environment. To avoid overflowing or emptying the tank, an additional cost is added if the water height is close to the maximum or minimum water levels, $h_{\text{max}}$ or $h_{\text{min}}$, respectively.

$$r(h, a) = -10(h \geq (h_{\text{max}} - 0.1)) - 10(h \leq 0.1) - a$$

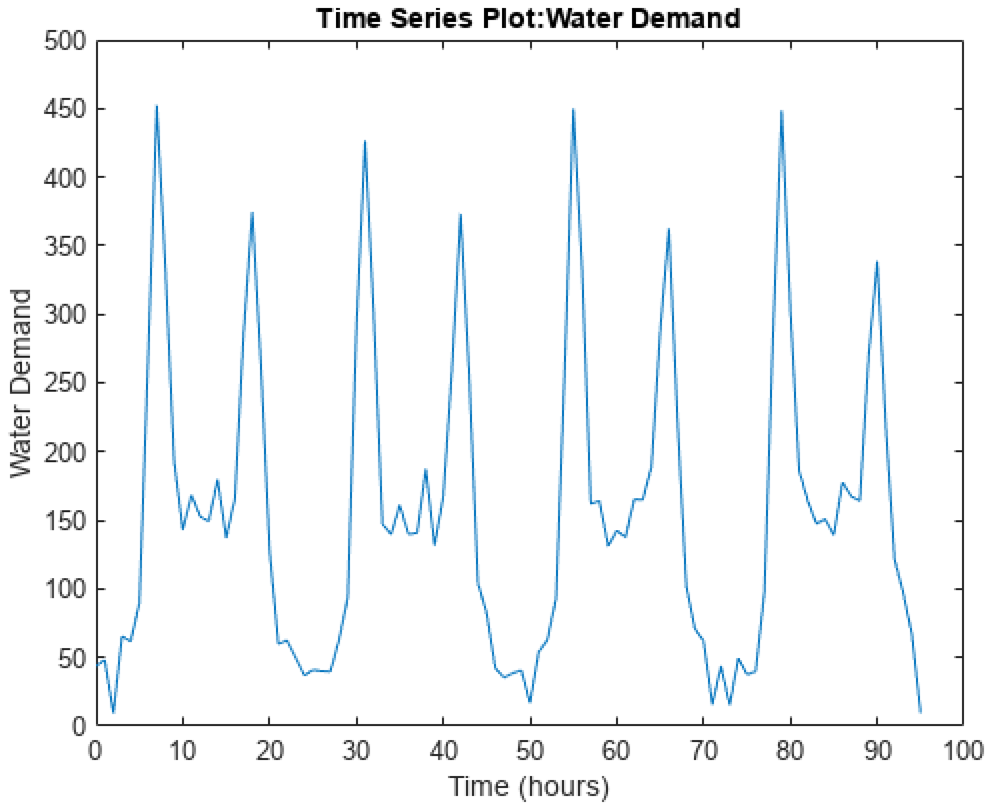
**Generate Demand Profile**

To generate and a water demand profile based on the number of days considered, use the `generateWaterDemand` function defined at the end of this example.

```
num_days = 4; % Number of days
[WaterDemand,T_max] = generateWaterDemand(num_days);
```

View the demand profile.

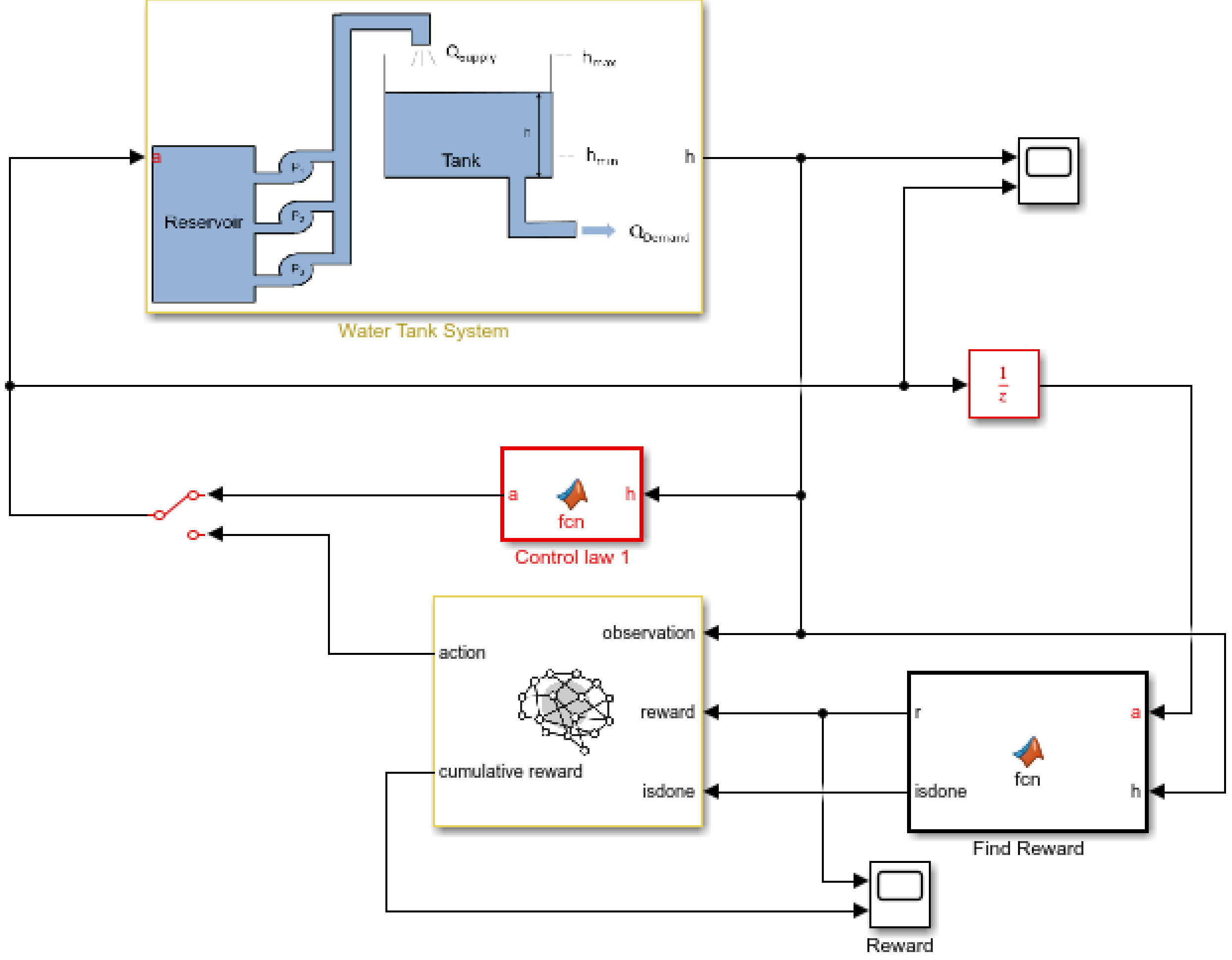
```
plot(WaterDemand)
```




#### Open and Configure Model

Open the distribution system Simulink model.

```
mdl = "watertankscheduling";
open_system(mdl)
```




In addition to the reinforcement learning agent, a simple baseline controller is defined in the Control law MATLAB Function block. This controller activates a certain number of pumps depending on the water level.

Specify the initial water height.

```
h0 = 3; % m
```

Specify the model parameters.

```
SampleTime = 0.2;
H_max = 7; % Max tank height (m)
A_tank = 40; % Area of tank (m^2)
```

### Create Environment Interface for RL Agent

To create an environment interface for the Simulink model, first define the action and observation specifications, `actInfo` and `obsInfo`, respectively. The agent action is the selected number of pumps. The agent observation is the water height, which is measured as a continuous-time signal.

```
actInfo = rlFiniteSetSpec([0,1,2,3]);
obsInfo = rlNumericSpec([1,1]);
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,mdl+"/RL Agent",obsInfo,actInfo);
```

Specify a custom reset function, which is defined at the end of this example, that randomizes the initial water height and the water demand. Doing so allows the agent to be trained on different initial water levels and water demand functions for each episode.

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

The actor and critic networks are initialized randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0);
```

#### Create DQN Agent

DQN agents use a parametrized Q-value function approximator to estimate the value of the policy. Since DQN agents have a discrete action space, you have the option to create a vector (that is multi-output) Q-value function critic, which is generally more efficient than a comparable single-output critic.

A vector Q-value function takes only the observation as input and returns as output a single vector with as many elements as the number of possible actions. The value of each output element represents the expected discounted cumulative long-term reward when an agent starts from the state corresponding to the given observation and executes the action corresponding to the element number (and follows a given policy afterwards)

To model the parametrized Q-value function within the critic, use a non-recurrent neural network. To use a recurrent neural network, set `useLSTM` to `true`. Note that `prod(obsInfo.Dimension)` returns the number of dimensions of the observation space (regardless of whether they are arranged as a row vector, column vector, or matrix, while `numel(actInfo.Elements)` returns the number of elements of the discrete action space.

```
useLSTM = false;
if useLSTM
    layers = [
        sequenceInputLayer(prod(obsInfo.Dimension))
        fullyConnectedLayer(32)
        reluLayer
        lstmLayer(32)
        fullyConnectedLayer(32)
        reluLayer
        fullyConnectedLayer(numel(actInfo.Elements))
        ];
else
    layers = [
        featureInputLayer(obsInfo.Dimension(1))
        fullyConnectedLayer(32)
        reluLayer
        fullyConnectedLayer(32)
        reluLayer
        fullyConnectedLayer(32)
        reluLayer
```

```
        fullyConnectedLayer(numel(actInfo.Elements))
        ];
end
```

Convert to a `dlnetwork` object and display the number of parameters.

```
dnn = dlnetwork(layers);
summary(dnn)
```

```
   Initialized: true

   Number of learnables: 2.3k

   Inputs:
      1   'input'   1 features
```

Create a critic using `rlVectorQValueFunction` using the defined deep neural network and the environment specifications.

```
critic = rlVectorQValueFunction(dnn,obsInfo,actInfo);
```

### Create DQN Agent

Specify training options for the critic and the actor using `rlOptimizerOptions`.

```
criticOpts = rlOptimizerOptions(LearnRate=0.001,GradientThreshold=1);
```

Specify the DDPG agent options using `rlDQNAgentOptions`. If you are using an LSTM network, set the sequence length to `20`.

```
opt = rlDQNAgentOptions(SampleTime=SampleTime);
if useLSTM
    opt.SequenceLength = 20;
else
    opt.SequenceLength = 1;
end
opt.DiscountFactor = 0.995;
opt.ExperienceBufferLength = 1e6;
opt.EpsilonGreedyExploration.EpsilonDecay = 1e-5;
opt.EpsilonGreedyExploration.EpsilonMin = .02;
```

Include the training options for the critic.

```
opt.CriticOptimizerOptions = criticOpts;
```

Create the agent using the critic and the options object.

```
agent = rlDQNAgent(critic,opt);
```

### Train Agent

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

- Run training for 1000 episodes, with each episode lasting at `ceil(T_max/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option)

Specify the options for training using an `rlTrainingOptions` object.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=1000, ...
    MaxStepsPerEpisode=ceil(T_max/SampleTime), ...
    Verbose=false, ...
    Plots="training-progress",...
    StopTrainingCriteria="EpisodeCount",...
    StopTrainingValue=1000,...
    ScoreAveragingWindowLength=100);
```

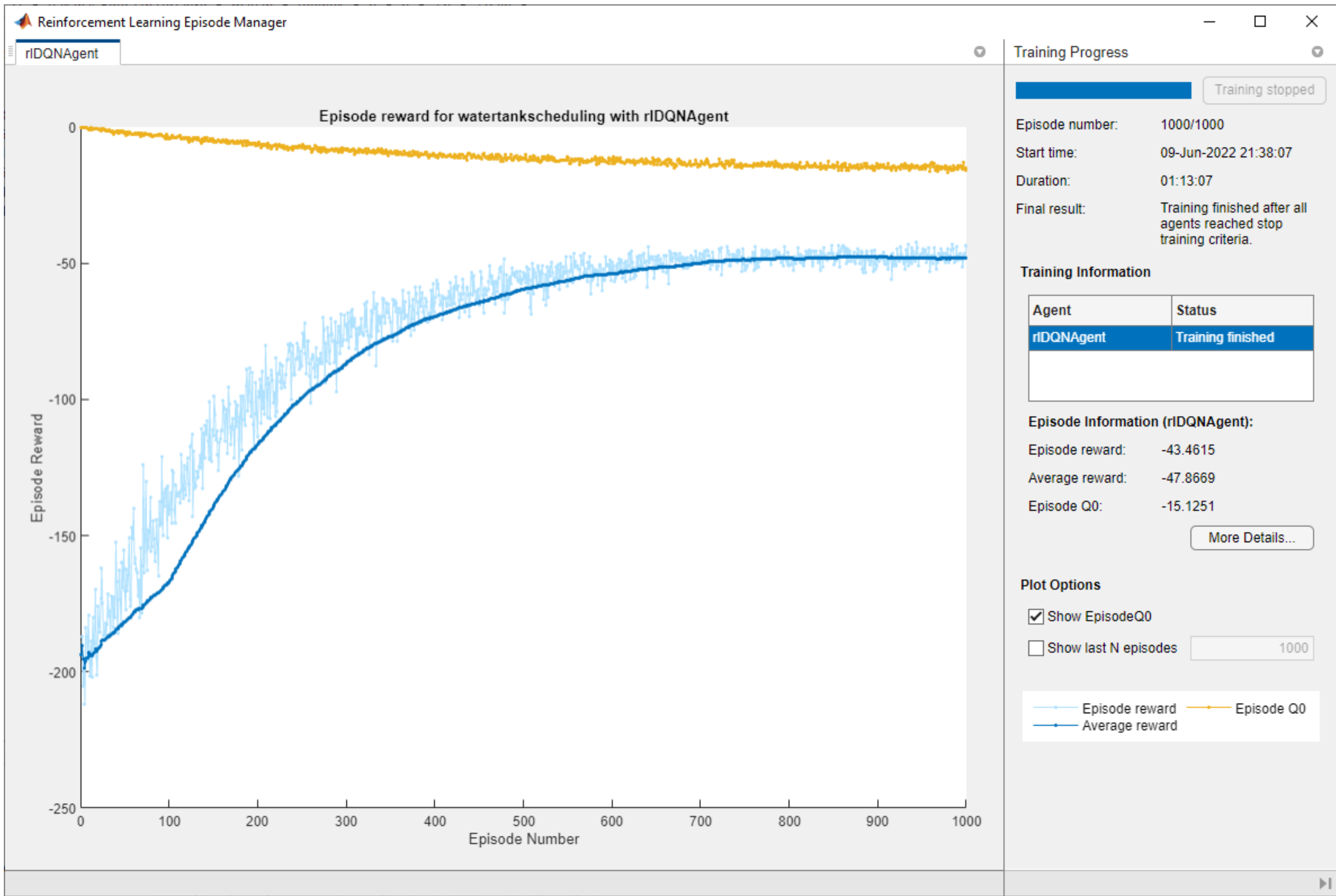
While you do not do it for this example, you can save agents during the training process. For example, the following options save every agent with a reward value greater than or equal to `-42`.

```
Save agents using SaveAgentCriteria if necessary
trainOpts.SaveAgentCriteria = "EpisodeReward";
trainOpts.SaveAgentValue = -42;
```

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

```
doTraining = false;
if doTraining

    % Connect the RL Agent block by toggling
    % the Manual Switch block
    set_param(mdl+"/Manual Switch","sw","0");

    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load("SimulinkWaterDistributionDQN.mat","agent")
end
```

The following figure shows the training progress.




## Simulate DQN Agent

To validate the performance of the trained agent, simulate it in the water-tank environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

To simulate the agent performance, connect the RL Agent block by toggling the manual switch block.

```
set_param(mdl+"/Manual Switch","sw","0");
```

Set the maximum number of steps for each simulation and the number of simulations. For this example, run 30 simulations. The environment reset function sets a different initial water height and water demand are different in each simulation.

```
NumSimulations = 30;
simOptions = rlSimulationOptions(MaxSteps=T_max/SampleTime,...
    NumSimulations= NumSimulations);
```

To compare the agent with the baseline controller under the same conditions, reset the initial random seed used in the environment reset function.

```
env.ResetFcn("Reset seed");
```

Simulate the agent against the environment.

```
experienceDQN = sim(env,agent,simOptions);
```

#### Simulate Baseline Controller

To compare DQN agent with the baseline controller, you must simulate the baseline controller using the same simulation options and initial random seed for the reset function.

Enable the baseline controller.

```
set_param(mdl+"/Manual Switch","sw","1");
```

To compare the agent with the baseline controller under the same conditions, reset the random seed used in the environment reset function.

```
env.ResetFcn("Reset seed");
```

Simulate the baseline controller against the environment.

```
experienceBaseline = sim(env,agent,simOptions);
```

#### Compare DQN Agent with Baseline Controller

Initialize cumulative reward result vectors for both the agent and baseline controller.

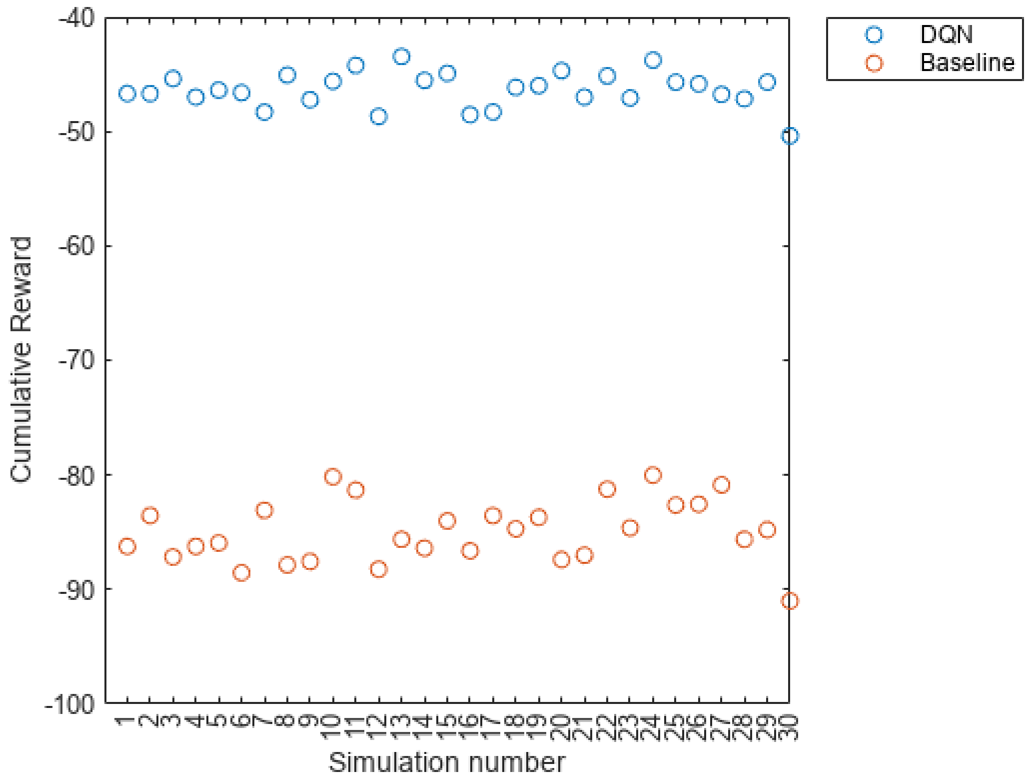
```
resultVectorDQN = zeros(NumSimulations, 1);
resultVectorBaseline = zeros(NumSimulations,1);
```

Calculate cumulative rewards for both the agent and the baseline controller.

```
for ct = 1:NumSimulations
    resultVectorDQN(ct) = sum(experienceDQN(ct).Reward);
    resultVectorBaseline(ct) = sum(experienceBaseline(ct).Reward);
end
```

Plot the cumulative rewards.

```
plot([resultVectorDQN resultVectorBaseline],"o")
set(gca,"xtick",1:NumSimulations)
xlabel("Simulation number")
ylabel("Cumulative Reward")
legend("DQN","Baseline","Location","NorthEastOutside")
```




The cumulative reward obtained by the agent is consistently around –40. This value is much greater than the average reward obtained by the baseline controller. Therefore, the DQN agent consistently outperforms the baseline controller in terms of energy savings.

## Local Functions

### Water Demand Function

```
function [WaterDemand,T_max] = generateWaterDemand(num_days)

    t = 0:(num_days*24)-1; % hr
    T_max = t(end);

    Demand_mean = [28, 28, 28, 45, 55, 110, ...
                   280, 450, 310, 170, 160, 145, ...
                   130, 150, 165, 155, 170, 265, ...
                   360, 240, 120, 83, 45, 28 ]'; % m^3/hr

    Demand = repmat(Demand_mean,1,num_days);
    Demand = Demand(:);

    % Add noise to demand
    a = -25; % m^3/hr
    b = 25; % m^3/hr
    Demand_noise = a + (b-a).*rand(numel(Demand),1);

    WaterDemand = timeseries(Demand + Demand_noise,t);
```

```
    WaterDemand.Name = "Water Demand";
    WaterDemand.TimeInfo.Units = "hours";
end
```

**Reset Function**

```
function in = localResetFcn(in)

    % Use a persistent random seed value to evaluate the agent
    % and the baseline controller under the same conditions.
    persistent randomSeed
    if isempty(randomSeed)
        randomSeed = 0;
    end
    if strcmp(in,"Reset seed")
        randomSeed = 0;
        return
    end
    randomSeed = randomSeed + 1;
    rng(randomSeed)

    % Randomize water demand.
    num_days = 4;
    H_max = 7;
    [WaterDemand,~] = generateWaterDemand(num_days);
    assignin("base","WaterDemand",WaterDemand)

    % Randomize initial height.
    h0 = 3*randn;
    while h0 <= 0 || h0 >= H_max
        h0 = 3*randn;
    end

    blk = "watertankscheduling/Water Tank System/Initial Water Height";
    in = setBlockParameter(in,blk,"Value",num2str(h0));

end
```

## See Also

**Functions**
train | sim | rlSimulinkEnv

**Objects**
rlDQNAgent | rlDQNAgentOptions | rlOptimizerOptions | rlTrainingOptions | rlSimulationOptions

**Blocks**
RL Agent

## Related Examples

- “Train DQN Agent to Balance Cart-Pole System” on page 5-50

## More About


- “Create Policies and Value Functions” on page 4-2
- “Deep Q-Network (DQN) Agents” on page 3-23
- “Train Reinforcement Learning Agents” on page 5-3

# Imitate MPC Controller for Lane Keeping Assist

This example shows how to train, validate, and test a deep neural network that imitates the behavior of a model predictive controller for an automotive lane keeping assist system. In the example, you also compare the behavior of the deep neural network with that of the original controller.

Model predictive control (MPC) solves a constrained quadratic-programming (QP) optimization problem in real time based on the current state of the plant (for more information, see "What is Model Predictive Control?" (Model Predictive Control Toolbox)). Because MPC solves its optimization problem in an open-loop fashion, you can potentially replace the controller with a deep neural network. Evaluating a deep neural network can be more computationally efficient than solving a QP problem in real time.

If the training of the network sufficiently traverses the state-space for the application, you can create a reasonable approximation of the controller behavior. You can then deploy the network for your control application. You can also use the network as a warm starting point for training the actor network of a reinforcement learning agent. For an example, see "Train DDPG Agent with Pretrained Actor Network" on page 5-373.

### Design MPC Controller

Design an MPC controller for lane keeping assist. To do so, first create a dynamic model for the vehicle.

```
[sys,Vx] = createModelForMPCImLKA;
```

Create and design the MPC controller object `mpcobj`. Also, create an `mpcstate` object for setting the initial controller state. For details on the controller design, type `edit createMPCobjImLKA`.

```
[mpcobj,initialState] = createMPCobjImLKA(sys);
```

For more information on designing model predictive controllers for lane keeping assist applications, see "Lane Keeping Assist System Using Model Predictive Control" (Model Predictive Control Toolbox) and "Lane Keeping Assist with Lane Detection" (Model Predictive Control Toolbox).

### Prepare Input Data

The data in `InputDataFileImLKA.mat` was created by computing the MPC control actions for randomly generated states, previous control actions, and measured disturbances. To generate your own training data, use the `collectDataImLKA` function.

For this example, load the input data from `InputDataFileImLKA.mat`.

```
dataStruct = load("InputDataFileImLKA.mat");
data = dataStruct.Data;
```

The columns of the data set are ordered as follows:

1. Lateral velocity $V_y$
2. Yaw angle rate $r$
3. Lateral deviation $e_1$
4. Relative yaw angle $e_2$

5. Previous steering angle (control variable) $u$
6. Measured disturbance (road yaw rate: longitudinal velocity * curvature ($\rho$))
7. Cost function value
8. MPC iterations
9. Steering angle computed by MPC controller: $u^*$

Divide the input data into training, validation, and testing data. First, determine the number of validation data rows based on a given percentage.

```
totalRows = size(data,1);
validationSplitPercent = 0.1;
numValidationDataRows = floor(validationSplitPercent*totalRows);
```

Determine the number of test data rows based on a given percentage.

```
testSplitPercent = 0.05;
numTestDataRows = floor(testSplitPercent*totalRows);
```

Randomly extract validation and testing data from the input data set. To do so, first randomly extract enough rows for both data sets.

```
randomIdx = randperm(totalRows,numValidationDataRows + numTestDataRows);
randomData = data(randomIdx,:);
```

Divide the random data into validation and testing data.

```
validationData = randomData(1:numValidationDataRows,:);
testData = randomData(numValidationDataRows + 1:end,:);
```

Extract the remaining rows as training data.

```
trainDataIdx = setdiff(1:totalRows,randomIdx);
trainData = data(trainDataIdx,:);
```

Randomize the training data.

```
numTrainDataRows = size(trainData,1);
shuffleIdx = randperm(numTrainDataRows);
shuffledTrainData = trainData(shuffleIdx,:);
```

Reshape the training and validation data into 4-D matrices for use with `trainNetwork`.

```
numObs = 6;
numActions = 1;

trainInput = shuffledTrainData(:,1:6);
trainOutput = shuffledTrainData(:,9);

validationInput = validationData(:,1:6);
validationOutput = validationData(:,9);
validationCellArray = {validationInput,validationOutput};
```

Reshape the testing data for use with `predict`.

```
testDataInput = testData(:,1:6);
testDataOutput = testData(:,9);
```

### Create Deep Neural Network

The deep neural network architecture uses the following layers.

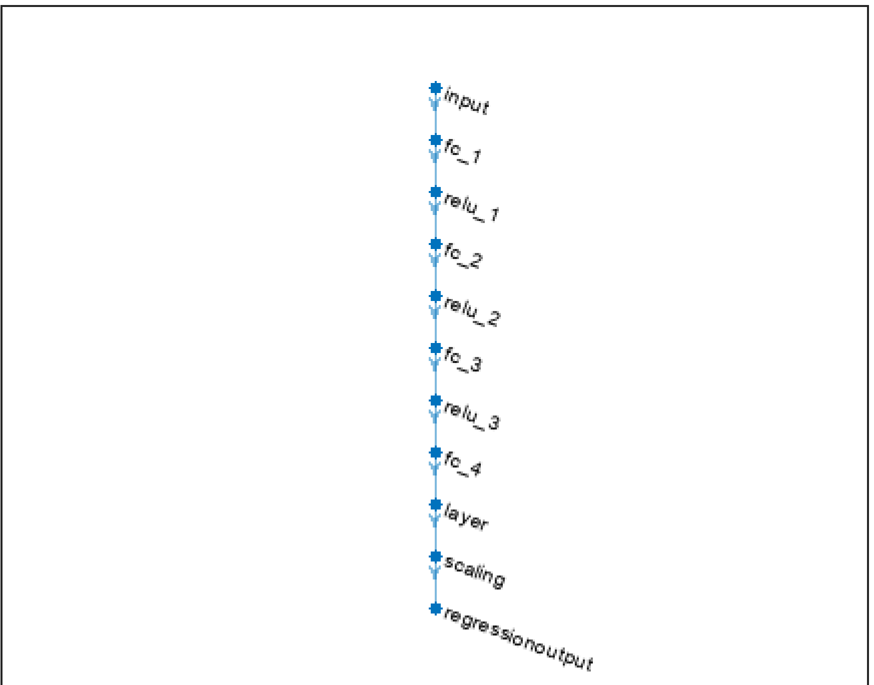
- `imageInputLayer` is the input layer of the neural network.
- `fullyConnectedLayer` multiplies the input by a weight matrix and then adds a bias vector.
- `reluLayer` is the activation function of the neural network.
- `tanhLayer` constrains the value to the range to [-1,1].
- `scalingLayer` scales the value to the range to [-1.04,1.04], this constrains the steering angle to the range [-60,60].
- `regressionLayer` defines the loss function of the neural network.

Create the deep neural network that will imitate the MPC controller after training.

```
imitateMPCLayers = [
    featureInputLayer(numObs)
    fullyConnectedLayer(45)
    reluLayer
    fullyConnectedLayer(45)
    reluLayer
    fullyConnectedLayer(45)
    reluLayer
    fullyConnectedLayer(numActions)
    tanhLayer
    scalingLayer(Scale=1.04)
    regressionLayer
];
```

Plot the network.

```
plot(layerGraph(imitateMPCLayers))
```



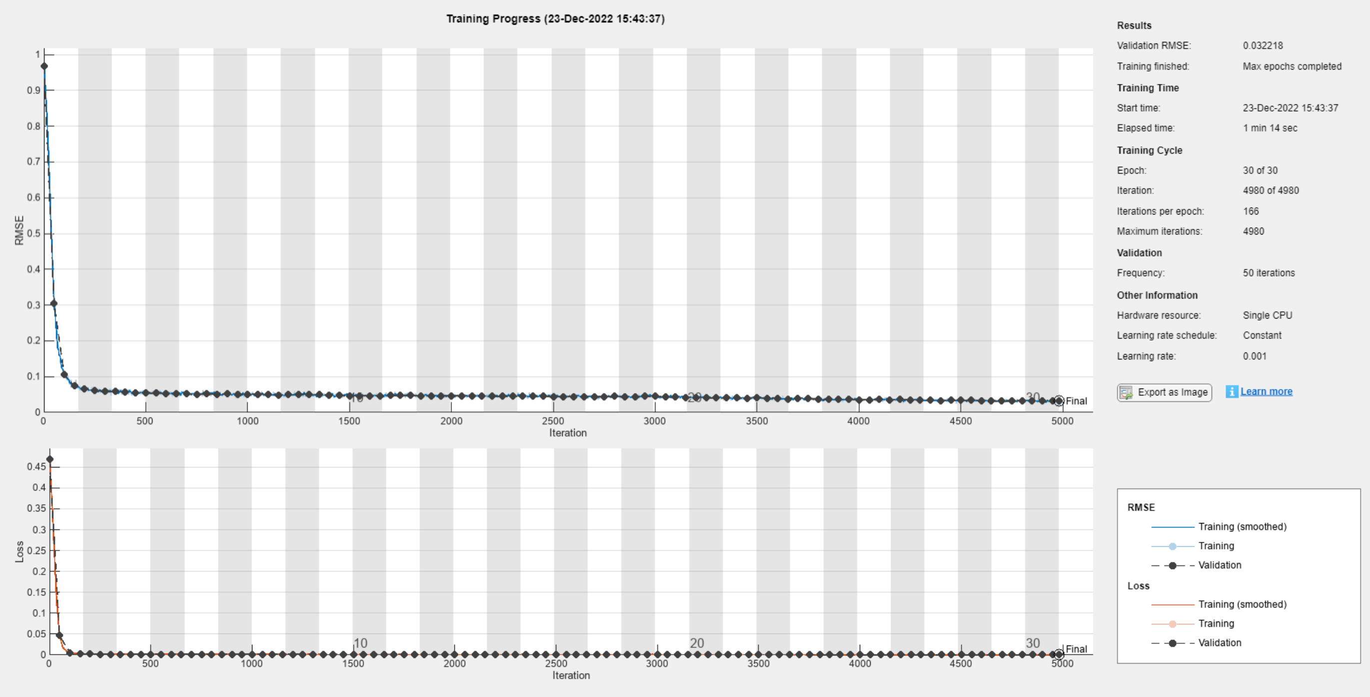

#### Train Deep Neural Network

Specify training options.

```
options = trainingOptions("adam", ...
    Verbose=false, ...
    Plots="training-progress", ...
    Shuffle="every-epoch", ...
    MaxEpochs=30, ...
    MiniBatchSize=512, ...
    ValidationData=validationCellArray, ...
    InitialLearnRate=1e-3, ...
    GradientThresholdMethod="absolute-value", ...
    ExecutionEnvironment="cpu", ...
    GradientThreshold=10, ...
    Epsilon=1e-8);
```

Train the deep neural network. To view detailed training information in the Command Window, set the `Verbose` training option to `true`.

```
imitateMPCNetwork = trainNetwork( ...
    trainInput, ...
    trainOutput, ...
    imitateMPCLayers, ...
    options);
```




Training of the deep neural network stops after the final iteration.

The training and validation loss are nearly the same for each mini-batch, which indicates that the trained network does not overfit.

### Test Trained Network

Check that the trained deep neural network returns steering angles similar to the MPC controller control actions given the test input data. Compute the network output using the `predict` function.

```
predictedTestDataOutput = predict(imitateMPCNetwork,testDataInput);
```

Calculate the root mean squared error (RMSE) between the network output and the testing data.

```
testRMSE = sqrt(mean((testDataOutput - predictedTestDataOutput).^2));
fprintf("Test Data RMSE = %d\n", testRMSE);
```

```
Test Data RMSE = 3.251544e-02
```

The small RMSE value indicates that the network outputs closely reproduce the MPC controller outputs.

### Compare Trained Network with MPC Controller

To compare the performance of the MPC controller and the trained deep neural network, run closed-loop simulations using the vehicle plant model.

Generate random initial conditions for the vehicle that are not part of the original input data set, with values selected from the following ranges:

1. Lateral velocity $V_y$ — Range (-2,2) m/s

2. Yaw angle rate $r$ — Range (-1.04,1.04) rad/s
3. Lateral deviation $e_1$ — Range (-1,1) m
4. Relative yaw angle $e_2$ — Range (-0.8,0.8) rad
5. Last steering angle (control variable) $u$ — Range (-1.04,1.04) rad
6. Measured disturbance (road yaw rate, defined as longitudinal velocity * curvature ($\rho$)) — Range (-0.01,0.01) with a minimum road radius of 100 m

```
rng(5e7)
[x0,u0,rho] = generateRandomDataImLKA(data);
```

Set the initial plant state and control action in the `mpcstate` object.

```
initialState.Plant = x0;
initialState.LastMove = u0;
```

Extract the sample time from the MPC controller. Also, set the number of simulation steps.

```
Ts = mpcobj.Ts;
Tsteps = 30;
```

Obtain the `A` and `B` state-space matrices for the vehicle model.

```
A = sys.A;
B = sys.B;
```

Initialize the state and input trajectories for the MPC controller simulation.

```
xHistoryMPC = repmat(x0',Tsteps+1,1);
uHistoryMPC = repmat(u0',Tsteps,1);
```

Run a closed-loop simulation of the MPC controller and the plant using the `mpcmove` function.

```
for k = 1:Tsteps

    % Obtain plant outputs
    xk = xHistoryMPC(k,:)';

    % Compute control action using the MPC controller
    uk = mpcmove(mpcobj,initialState,xk,zeros(1,4),Vx*rho);

    % Store the control action
    uHistoryMPC(k,:) = uk;

    % Update plant state using the control action
    xHistoryMPC(k+1,:) = (A*xk + B*[uk;Vx*rho])';
end
```

Initialize the state and input trajectories for the deep neural network simulation.

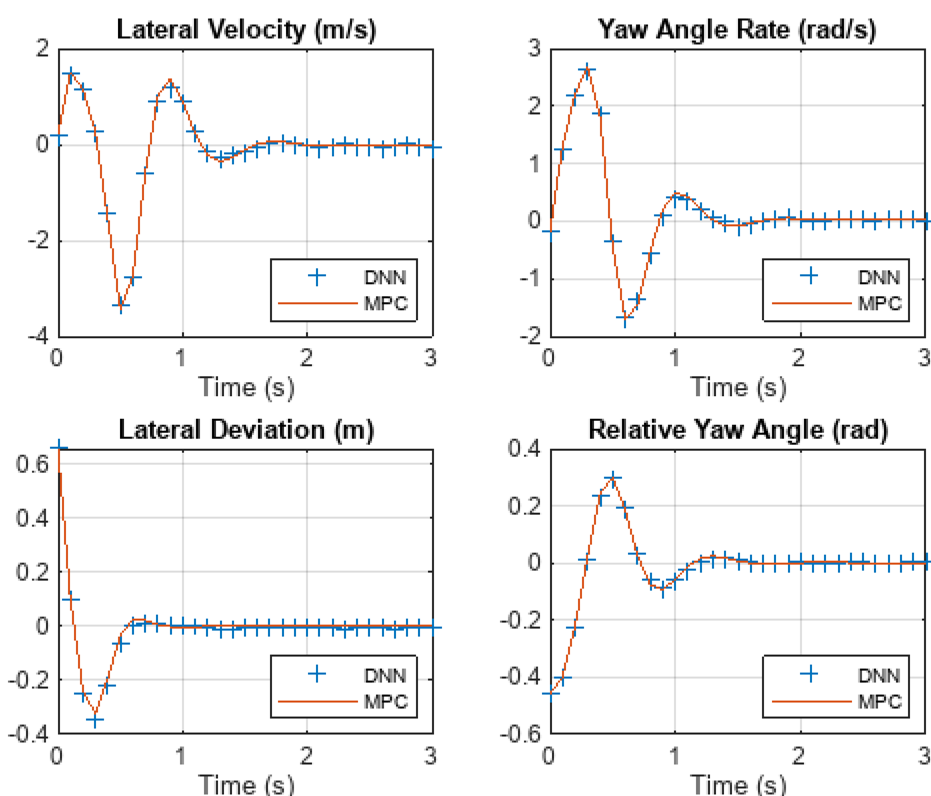
```
xHistoryDNN = repmat(x0',Tsteps+1,1);
uHistoryDNN = repmat(u0',Tsteps,1);
lastMV = u0;
```

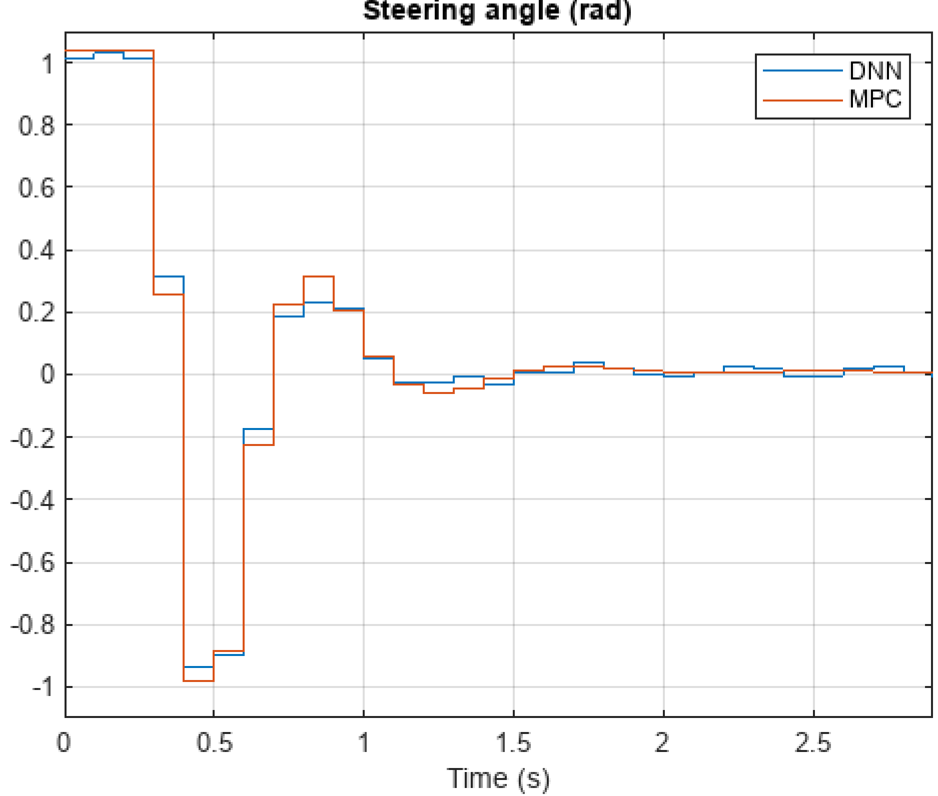
Run a closed-loop simulation of the trained network and the plant. The `neuralnetLKAmove` function computes the deep neural network output using the `predict` function.

```
for k = 1:Tsteps

    % Obtain plant outputs
    xk = xHistoryDNN(k,:)';

    % Predict the next move using trained network
    uk = neuralnetLKAmove(imitateMPCNetwork,xk,lastMV,rho);

    % Store control action
    uHistoryDNN(k,:) = uk;

    % Update the last MV for the next step
    lastMV = uk;

    % Update plant state using the control action
    xHistoryDNN(k+1,:) = (A*xk + B*[uk;Vx*rho])';
end
```

Plot the results to compare the MPC controller and trained deep neural network (DNN) trajectories.

```
plotValidationResultsImLKA(Ts, ...
    xHistoryDNN,uHistoryDNN, ...
    xHistoryMPC,uHistoryMPC);
```






The deep neural network successfully imitates the behavior of the MPC controller. The vehicle state and control action trajectories for the controller and the deep neural network closely align.

## See Also

**Functions**
`trainNetwork` | `predict` | `mpcmove`

**Objects**
`SeriesNetwork` | `mpc`

## Related Examples

- “Train DDPG Agent with Pretrained Actor Network” on page 5-373
- “Lane Keeping Assist System Using Model Predictive Control” (Model Predictive Control Toolbox)
- “Lane Keeping Assist with Lane Detection” (Model Predictive Control Toolbox)

## More About

- “What is Model Predictive Control?” (Model Predictive Control Toolbox)

# Train DDPG Agent with Pretrained Actor Network

This example shows how to train a deep deterministic policy gradient (DDPG) agent for lane keeping assist (LKA) in Simulink. To make training more efficient, the actor of the DDPG agent is initialized with a deep neural network that was previously trained using supervised learning. This actor trained is trained in the “Imitate MPC Controller for Lane Keeping Assist” on page 5-365 example.

For more information on DDPG agents, see “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40.

### Simulink Model

The training goal for the lane-keeping application is to keep the ego vehicle traveling along the centerline of the a lane by adjusting the front steering angle. This example uses the same ego vehicle dynamics and sensor dynamics as the “Train DQN Agent for Lane Keeping Assist” on page 5-226 example.

```
m = 1575;   % total vehicle mass (kg)
Iz = 2875;  % yaw moment of inertia (mNs^2)
lf = 1.2;   % long. distance from center of gravity to front tires (m)
lr = 1.6;   % long. distance from center of gravity to rear tires (m)
Cf = 19000; % cornering stiffness of front tires (N/rad)
Cr = 33000; % cornering stiffness of rear tires (N/rad)
Vx = 15;    % longitudinal velocity (m/s)
```

Define the sample time, `Ts`, and simulation duration, `T`, in seconds.

```
Ts = 0.1;
T = 15;
```

The output of the LKA system is the front steering angle of the ego vehicle. Considering the physical limitations of the ego vehicle, constrain its steering angle to the range [-60,60] degrees. Specify the constraints in radians.

```
u_min = -1.04;
u_max = 1.04;
```

Define the curvature of the road as a constant $0.001(m^{-1})$.

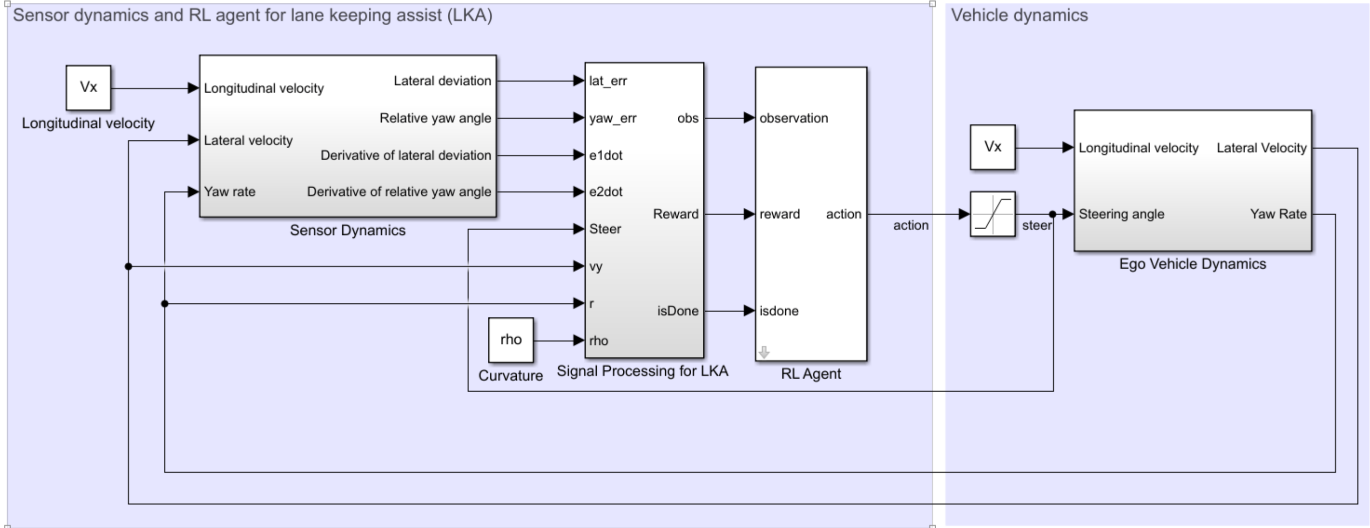
```
rho = 0.001;
```

Set initial values for the lateral deviation (`e1_initial`) and the relative yaw angle (`e2_initial`). During training, these initial conditions are set to random values for each training episode.

```
e1_initial = 0.2;
e2_initial = -0.1;
```

Open the model.

```
mdl = "rlActorLKAMdl";
open_system(mdl)
```




Copyright 2018 The MathWorks, Inc.

Define the path to the RL Agent block within the model.

```
agentblk = mdl + "/RL Agent";
```

**Create Environment**

Create a reinforcement learning environment interface for the ego vehicle. To do so, first define the observation and action specifications. These observations and actions are the same as the features for supervised learning used in “Imitate MPC Controller for Lane Keeping Assist” on page 5-365.

The six observations for the environment are the lateral velocity $v_y$, yaw rate $\dot{\psi}$, lateral deviation $e_1$, relative yaw angle $e_2$, steering angle at previous step $u_0$, and curvature $\rho$.

```
obsInfo = rlNumericSpec([6 1],...
    LowerLimit=-inf*ones(6,1), ...
    UpperLimit= inf*ones(6,1))
```

```
obsInfo = 
  rlNumericSpec with properties:

     LowerLimit: [6x1 double]
     UpperLimit: [6x1 double]
           Name: [0x0 string]
    Description: [0x0 string]
      Dimension: [6 1]
       DataType: "double"
```

```
obsInfo.Name = "observations";
```

The action for the environment is the front steering angle. Specify the steering angle constraints when creating the action specification object.

```
actInfo = rlNumericSpec([1 1], ...
    LowerLimit=u_min, ...
    UpperLimit=u_max);
actInfo.Name = "steering";
```

In the model, the Signal Processing for LKA block creates the observation vector signal, computes the reward function, and calculates the stop signal.

The reward $r_t$, provided at every time step $t$, is as follows, where $u$ is the control input from the previous time step $t-1$.

$$r_t = -(10e_1^2 + 5e_2^2 + 2u^2 + 5\dot{e}_1^2 + 5\dot{e}_2^2)$$

The simulation stops when $|e_1| > 1$.

Create the reinforcement learning environment.

```
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
```

To define the initial condition for lateral deviation and relative yaw angle, specify an environment reset function using an anonymous function handle. The `localResetFcn` function, which is defined at the end of the example, sets the initial lateral deviation and relative yaw angle to random values.

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

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create DDPG Agent

DDPG agents use a parametrized deterministic policy over continuous action spaces, which is learned by a continuous deterministic actor, and a parametrized Q-value function approximator to estimate the value of the policy. Use use neural networks to model both the parametrized policy within the actor and the Q-value function within the critic. For this example, use the helper functions `createLaneKeepingCritic` and `createLaneKeepingActor` to create the critic and the actor, along with their training options sets.

```
[critic,criticOpts] = createLaneKeepingCritic(obsInfo,actInfo);
[actor,actorOpts] = createLaneKeepingActor(obsInfo,actInfo);
```

These initial actor and critic networks have random initial parameter values.

Specify the DDPG agent options using `rlDDPGAgentOptions`, include the training options for the actor and critic.

```
agentOptions = rlDDPGAgentOptions(...
    SampleTime=Ts,...
    ActorOptimizerOptions=actorOpts,...
    CriticOptimizerOptions=criticOpts,...
    ExperienceBufferLength=1e6);
```

You can also set or modify the agent options using dot notation.

```
agentOptions.NoiseOptions.Variance = 0.3;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Create the DDPG agent using the specified actor and critic objects and the agent options. For more information, see `rlDDPGAgent`.

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

Alternatively, you can also create the agent first, and then access its option object, and modify the options, using dot notation.

### Train Agent

As a baseline, train the agent with an actor that has random initial parameters. To train the agent, first specify the training options. For this example, use the following options.

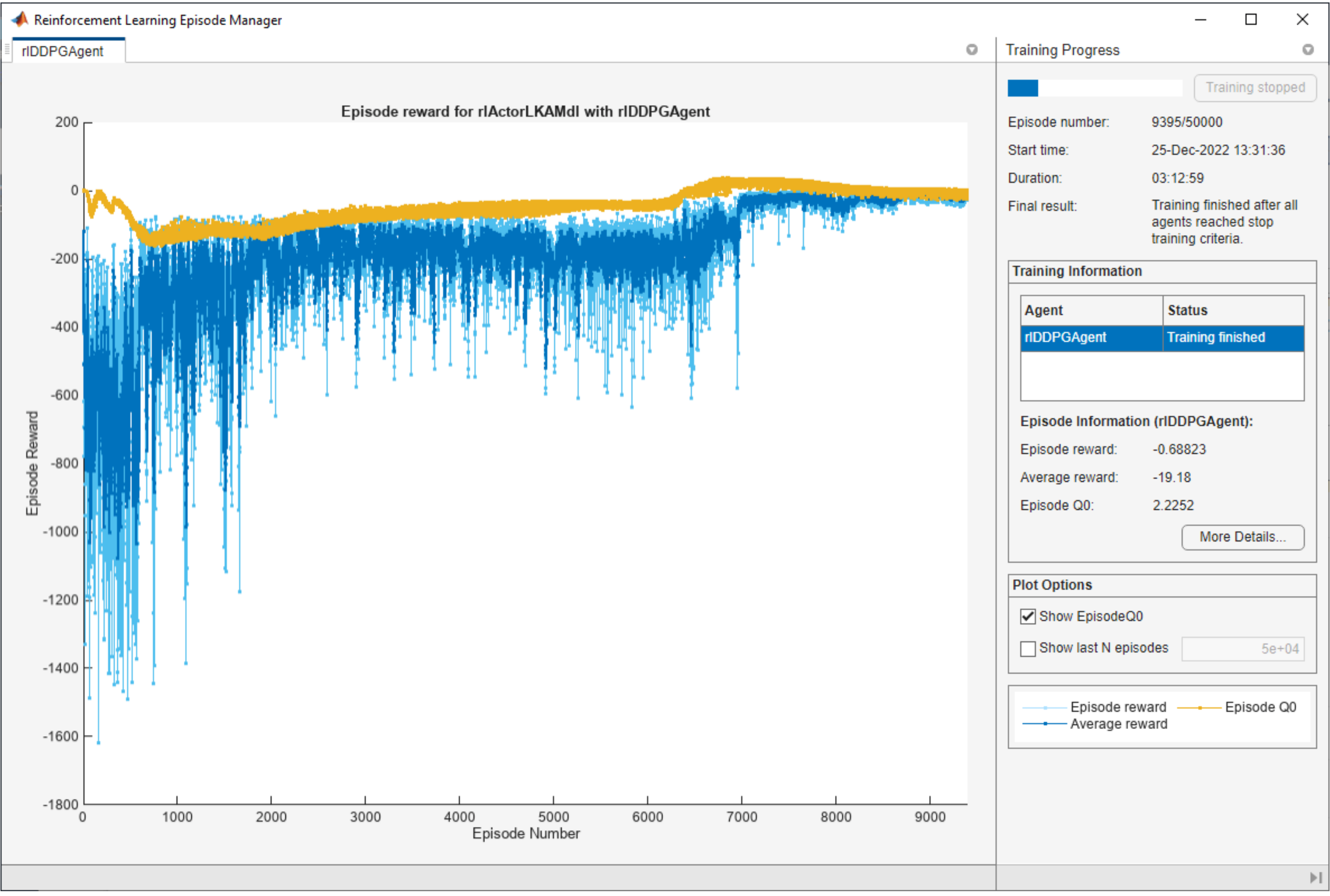
- Run training for at most 50000 episodes, with each episode lasting at most 150 time steps.
- Display the training progress in the Episode Manager dialog box.
- Stop training when the episode reward reaches –1.
- Save a copy of the agent for each episode where the cumulative reward is greater than `–2.5`.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 50000;
maxsteps = T/Ts;
trainingOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes,...
    MaxStepsPerEpisode=maxsteps,...
    Verbose=false,...
    Plots="training-progress",...
    StopTrainingCriteria="EpisodeReward",...
    StopTrainingValue=-1,...
    SaveAgentCriteria="EpisodeReward",...
    SaveAgentValue=-2.5);
```

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

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load pretrained agent for the example.
    load("ddpgFromScratch.mat");
end
```




## Train Agent with Pretrained Actor

You can set the actor network of your agent to a deep neural network that has been previously trained. For this example, use the deep neural network from the “Imitate MPC Controller for Lane Keeping Assist” on page 5-365 example. This network was trained to imitate a model predictive controller using supervised learning.

Load the pretrained actor network.

```
load("imitateMPCNetActorObj.mat","imitateMPCNetObj");
```

Create an actor representation using the pretrained actor.

```
supervisedActor = rlContinuousDeterministicActor( ...
    imitateMPCNetObj, ...
    obsInfo, ...
    actInfo);
```

Check that the network used by `supervisedActor` is the same one that was loaded. To do so, evaluate both the network and the agent using the same random input observation.

```
testData = rand(6,1);
```

Evaluate the deep neural network.

```
predictImNN = predict(imitateMPCNetObj,testData');
```

Evaluate the actor.

```
evaluateRLRep = getAction(supervisedActor,{testData});
```

Compare the results.

```
error = evaluateRLRep{:} - predictImNN
```

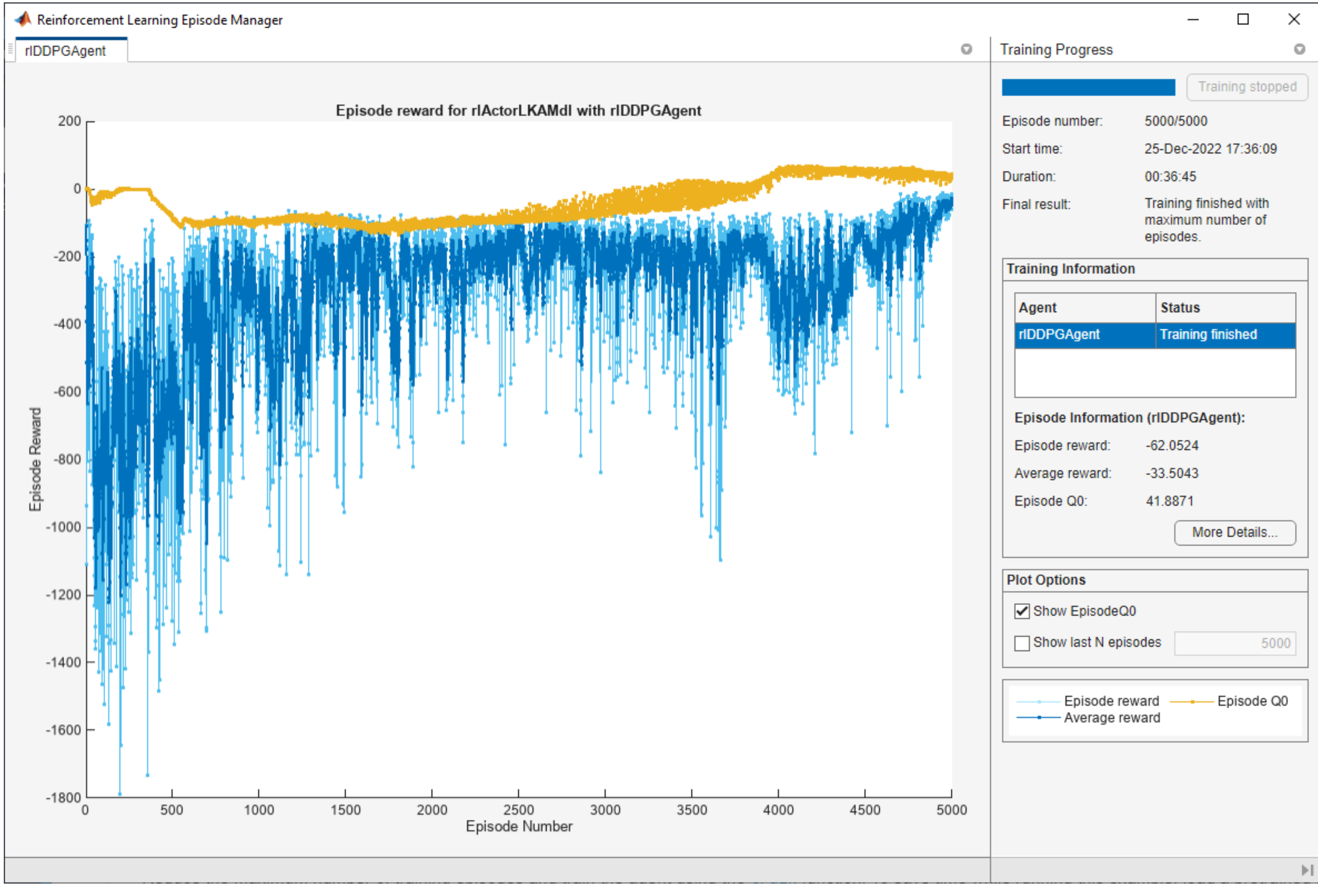
```
error = single
    1.4901e-08
```

Create a DDPG agent using the pretrained actor.

```
agent = rlDDPGAgent(supervisedActor,critic,agentOptions);
```

Reduce the maximum number of training episodes and train the agent using the `train` function. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
trainingOpts.MaxEpisodes = 5000;
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load pretrained agent for the example.
    load("ddpgFromPretrained.mat");
end
```




By using the pretrained actor network, the training of the DDPG agent is more efficient. Both the total training time and the total number of training steps have improved by approximately 20%. Also, the number of episodes for the training to approach the neighborhood of the optimal result decreased from approximately 4500 to approximately 3500.

### Simulate DDPG Agent

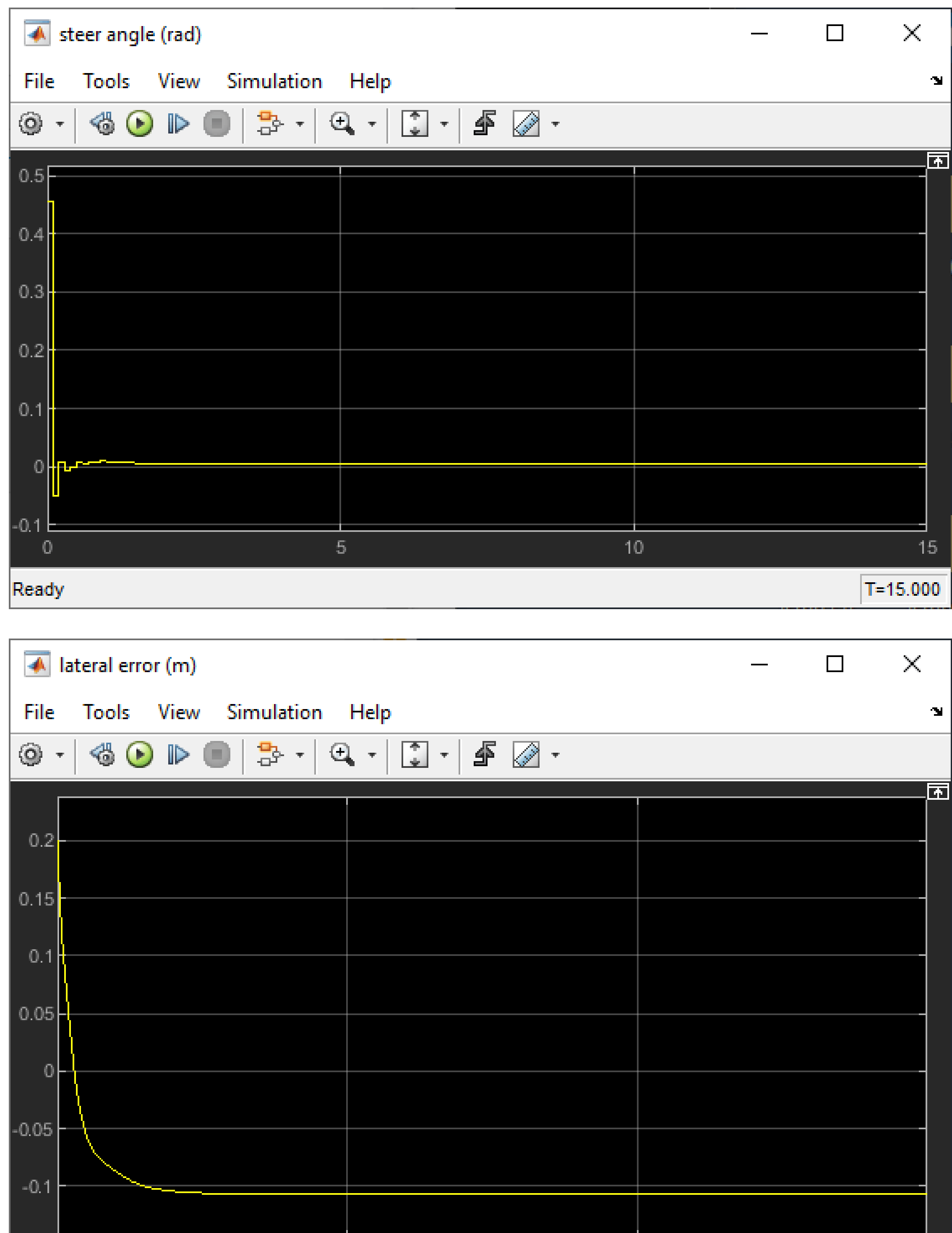
To validate the performance of the trained agent, uncomment the following two lines and simulate it within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

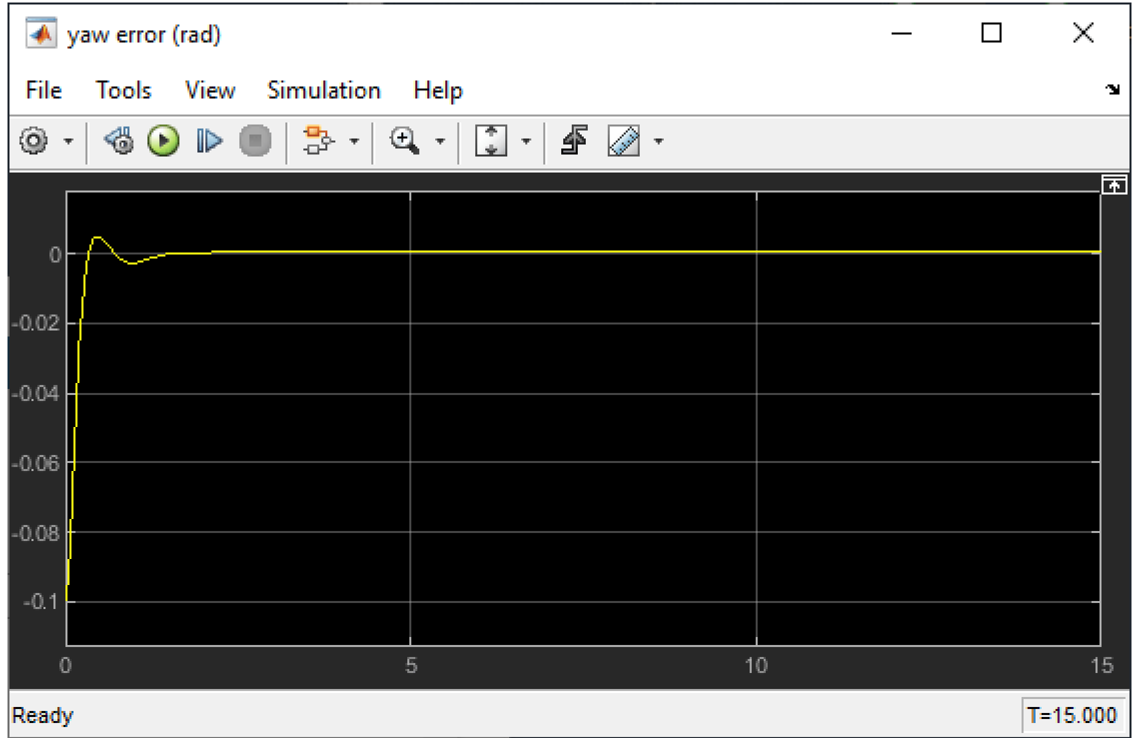
```
% simOptions = rlSimulationOptions(MaxSteps=maxsteps);
% experience = sim(env,agent,simOptions);
```

To check the performance of the trained agent within the Simulink model, simulate the model using the previously defined initial conditions (`e1_initial = 0.2` and `e2_initial = -0.1`).

```
sim(mdl)
```

As shown below, the lateral error (middle plot) and relative yaw angle (bottom plot) are both driven to zero. The vehicle starts with a lateral deviation from the centerline (0.2 m) and a nonzero yaw angle error (-0.1 rad). The lane-keeping controller makes the ego vehicle travel along the centerline after around two seconds. The steering angle (top plot) shows that the controller reaches steady state after about two seconds.


steer angle (rad)
File Tools View Simulation Help
0.5
0.4
0.3
0.2
0.1
0
-0.1
0
5
10
15
Ready
T=15.000
lateral error (m)
File Tools View Simulation Help
0.2
0.15
0.1
0.05
0
-0.05
-0.1
0
5
10
15
Ready
T=15.000




Close the Simulink model without saving any changes.

```
bdclose(mdl)
```

### Local Functions

```
function in = localResetFcn(in)
% Set random value for lateral deviation.
in = setVariable(in,"e1_initial", 0.5*(-1+2*rand));

% Set random value for relative yaw angle.
in = setVariable(in,"e2_initial", 0.1*(-1+2*rand));
end
```

## See Also

**Functions**
train | sim | rlSimulinkEnv

**Objects**
rlDDPGAgent | rlDDPGAgentOptions | rlQValueFunction | rlContinuousDeterministicActor | rlTrainingOptions | rlSimulationOptions | rlOptimizerOptions

**Blocks**
RL Agent

## Related Examples


- “Imitate MPC Controller for Lane Keeping Assist” on page 5-365
- “Train DQN Agent for Lane Keeping Assist” on page 5-226
- “Train DDPG Agent to Control Double Integrator System” on page 5-77
- “Lane Keeping Assist System Using Model Predictive Control” (Model Predictive Control Toolbox)


## More About


- “Define Reward Signals” on page 2-14
- “Deep Deterministic Policy Gradient (DDPG) Agents” on page 3-40
- “Train Reinforcement Learning Agents” on page 5-3

# Imitate Nonlinear MPC Controller for Flying Robot

This example shows how to train, validate, and test a deep neural network (DNN) that imitates the behavior of a nonlinear model predictive controller for a flying robot. It then compares the behavior of the deep neural network with that of the original controller. To train the deep neural network, this example uses the data aggregation (DAgger) approach as in [1].

Nonlinear model predictive control (NLMPC) solves a constrained nonlinear optimization problem in real time based on the current state of the plant. For more information on NLMPC, see “Nonlinear MPC” (Model Predictive Control Toolbox).

Since NLMPC solves its optimization problem in an open-loop fashion, there is the potential to replace the controller with a trained DNN. Doing so is an appealing option, since evaluating a DNN can be more computationally efficient than solving a nonlinear optimization problem in real-time.

If the trained DNN reasonably approximates the controller behavior, you can then deploy the network for your control application. You can also use the network as a warm starting point for training the actor network of a reinforcement learning agent. For an example that does so with a DNN trained for an MPC application, see “Train DDPG Agent with Pretrained Actor Network” on page 5-373.

### Design Nonlinear MPC Controller

Design a nonlinear MPC controller for a flying robot. The dynamics for the flying robot are the same as in “Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC” (Model Predictive Control Toolbox) example. First, define the limit for the control variables, which are the robot thrust levels.

```
umax = 3;
```

Create the nonlinear MPC controller object `nlobj`. To reduce command-window output, disable the MPC update messages. For more information, see `nlmpc` (Model Predictive Control Toolbox).

```
mpcverbosity off;
nlobj = createMPCobjImFlyingRobot(umax);
```

### Prepare Input Data

Load the input data from `DAggerInputDataFileImFlyingRobot.mat`. The columns of the data set contain:

1. $x$ is the position of the robot along the x-axis.
2. $y$ is the position of the robot along the y-axis.
3. $\theta$ is the orientation of the robot.
4. $\dot{x}$ is the velocity of the robot along the x-axis.
5. $\dot{y}$ is the velocity of the robot along the y-axis.
6. $\dot{\theta}$ is the angular velocity of the robot.
7. $u_l$ is the thrust on the left side of the flying robot
8. $u_r$ is the thrust on the right side of the flying robot
9. $u_l^*$ is the thrust on the left side computed by NLMPC

**10** $u_r^*$ is the thrust on the right side computed by NLMPC

The data in `DAggerInputDataFileImFlyingRobot.mat` is created by computing the NLMPC control action for randomly generated states ($x$, $y$, $\theta$, $\dot{x}$, $\dot{y}$, $\dot{\theta}$), and previous control actions ($u_l$, $u_r$). To generate your own training data, use the `collectDataImFlyingRobot` function.

Load the input data.

```
fileName = "DAggerInputDataFileImFlyingRobot.mat";
DAggerData = load(fileName);
data = DAggerData.data;
existingData = data;
numCol = size(data,2);
```

#### Create Deep Neural Network

Create the deep neural network that will imitate the NLMPC controller after training. The network architecture uses the following types of layers.

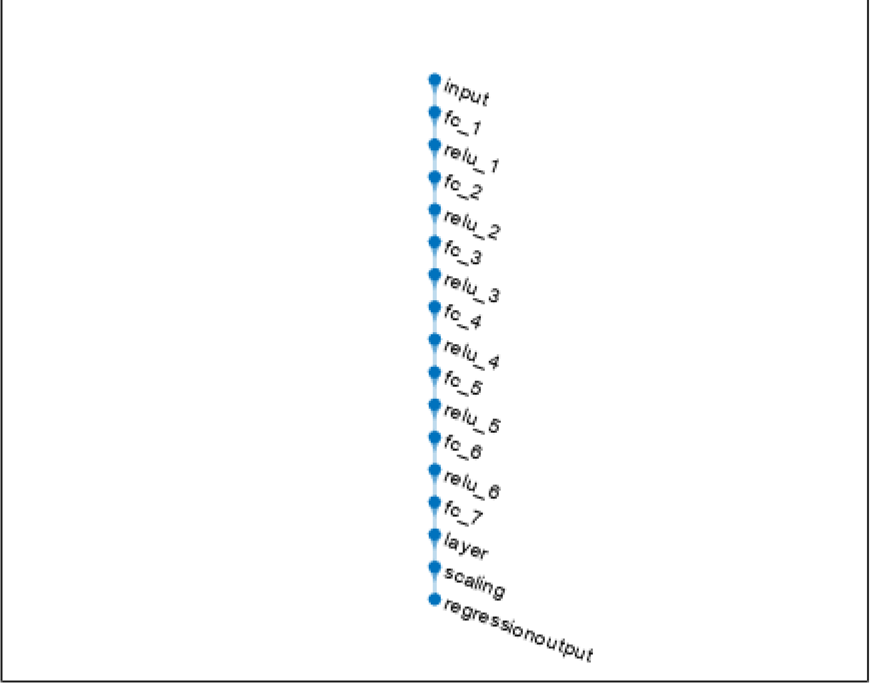
- `imageInputLayer` is the input layer of the neural network.
- `fullyConnectedLayer` multiplies the input by a weight matrix and then adds a bias vector.
- `reluLayer` is the activation function of the neural network.
- `tanhLayer` constrains the value to the range to [-1,1].
- `scalingLayer` scales the value to the range to [-3,3].
- `regressionLayer` defines the loss function of the neural network.

Define the network as an array of layer objects.

```
numObs = numCol-2;
numAct = 2;
hiddenLayerSize = 256;

imitateMPCNetwork = [
    featureInputLayer(numObs)
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(hiddenLayerSize)
    reluLayer
    fullyConnectedLayer(numAct)
    tanhLayer
    scalingLayer(Scale=umax)
    regressionLayer
    ];
```

Plot the network.

```
plot(layerGraph(imitateMPCNetwork))
```




### Behavior Cloning Approach

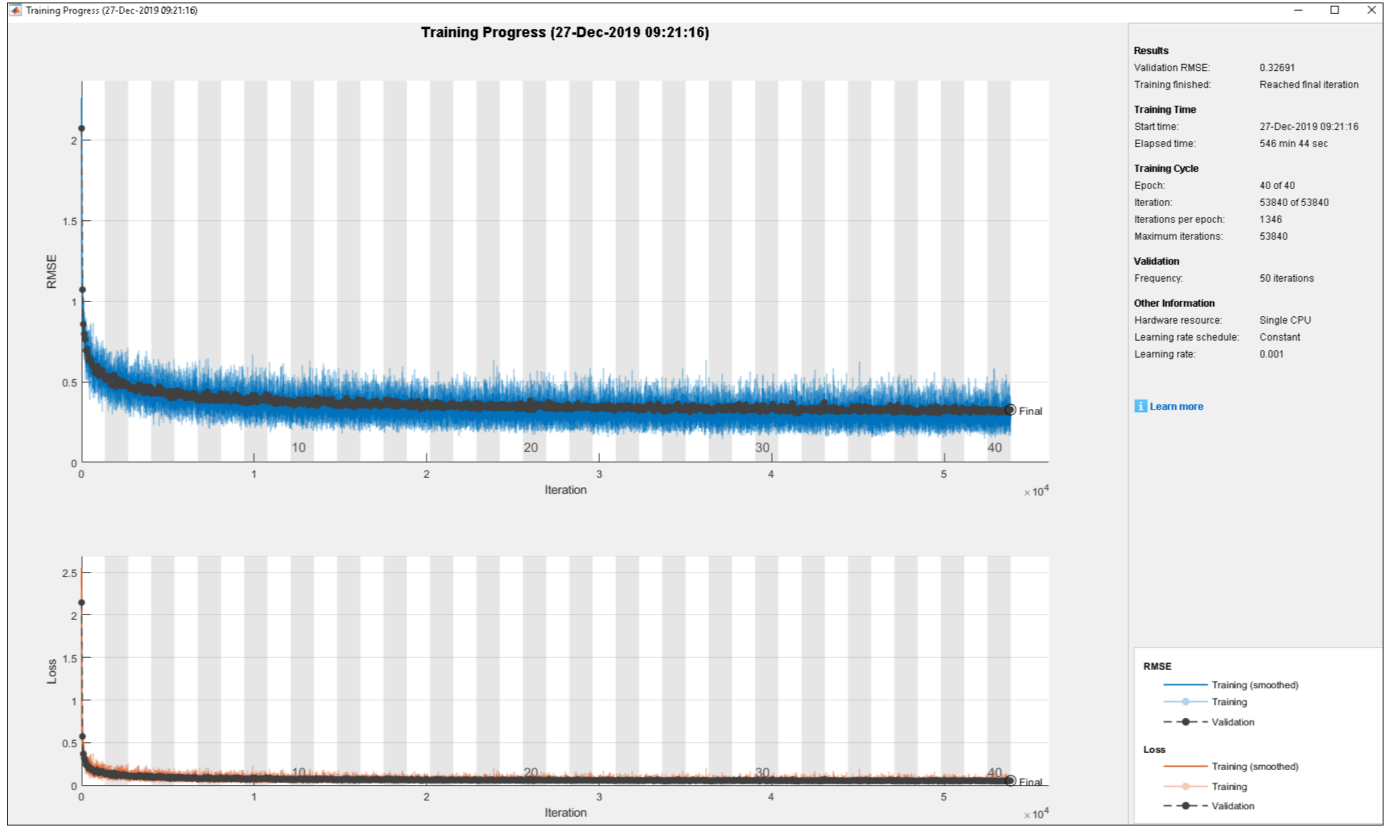
One approach to learning an expert policy using supervised learning is the behavior cloning method. This method divides the expert demonstrations (NLMPC control actions in response to observations) into state-action pairs and applies supervised learning to train the network.

Specify training options.

```
% Intialize validation cell array
validationCellArray = {0,0};

options = trainingOptions("adam", ...
    Verbose=false, ...
    Plots="training-progress", ...
    Shuffle="every-epoch", ...
    MiniBatchSize=512, ...
    ValidationData=validationCellArray, ...
    InitialLearnRate=1e-3, ...
    ExecutionEnvironment="cpu", ...
    GradientThreshold=10, ...
    MaxEpochs=40 ...
    );
```

You can train the behavior cloning neural network by following below steps

1. Collect data using the `collectDataImFlyingRobot` function.
2. Train the behavior cloning network using the `behaviorCloningTrainNetwork` function.




The training of the DNN using behavior cloning reduces the gap between the DNN and NLMPC performance. However, the behavior cloning neural network fails to imitate the behavior of the NLMPC controller correctly on some randomly generated data.

Training a DNN is a computationally intensive process. For this example, to save time, load a pretrained neural network object.

```
load("behaviorCloningMPCImDNNObject.mat");
```

### Data Aggregation Approach

To improve the performance of the DNN, you can learn the policy using an interactive demonstrator method. DAgger is an iterative method where the DNN is run in the closed-loop environment. The expert, in this case the NLMPC controller, outputs actions based on the states visited by the DNN. In this manner, more training data is aggregated and the DNN is retrained for improved performance. For more information, see [1].

Train the deep neural network using the `DAggerTrainNetwork` function. It creates `DAggerImFlyingRobotDNNObj.mat` file that contains the following information.

- `DatasetPath:` path where the dataset corresponding to each iteration is stored
- `policyObjs:` policies that were trained in each iteration
- `finalData:` total training data collected till final iteration
- `finalPolicy:` best policy among all the collected policies

First, create and initialize the parameters for training. Use the network trained using behavior cloning (`behaviorCloningNNObj.imitateMPCNetObj`) as the starting point for the DAgger training.

```
[dataStruct,nlmpcStruct,tuningParamsStruct,neuralNetStruct] = ...
    loadDAggerParameters(existingData,numCol,nlobj,umax, ...
    options, behaviorCloningNNObj.imitateMPCNetObj);
```

To save time, load a pretrained neural network by setting `doTraining` to `false`. To train the DAgger yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    DAgger = DAggerTrainNetwork( ...
        nlmpcStruct, ...
        dataStruct, ...
        neuralNetStruct, ...
        tuningParamsStruct);
else
    load("DAggerImFlyingRobotDNNObj.mat");
end
DNN = DAgger.finalPolicy;
```

As an alternative, you can train the neural network with a modified policy update rule using the `DAggerModifiedTrainNetwork` function. In this function, after every 20 training iterations, the DNN is set to the most optimal configuration from the previous 20 iterations. To run this example with a neural network object with the modified DAgger approach, use the `DAggerModifiedImFlyingRobotDNNObj.mat` file.

#### Compare Trained DAgger Network with NLMPC Controller

To compare the performance of the NLMPC controller and the trained DNN, run closed-loop simulations with the flying robot model.

Set initial condition for the states of the flying robot ($x$, $y$, $\theta$, $\dot{x}$, $\dot{y}$, $\dot{\theta}$) and the control variables of flying robot ($u_l$, $u_r$).

```
x0 = [-1.8200 0.5300 -2.3500 1.1700 -1.0400 0.3100]';
u0 = [-2.1800 -2.6200]';
```

Define simulation duration, sample time and number of simulation steps.

```
% Duration
Tf = 15;

% Sample time
Ts = nlobj.Ts;

% Simulation steps
Tsteps = Tf/Ts+1;
```

Run a closed-loop simulation of the NLMPC controller.

```
tic
[xHistoryMPC,uHistoryMPC] = ...
    simModelMPCImFlyingRobot(x0,u0,nlobj,Tf);
toc
```

```
Elapsed time is 24.483770 seconds.
```

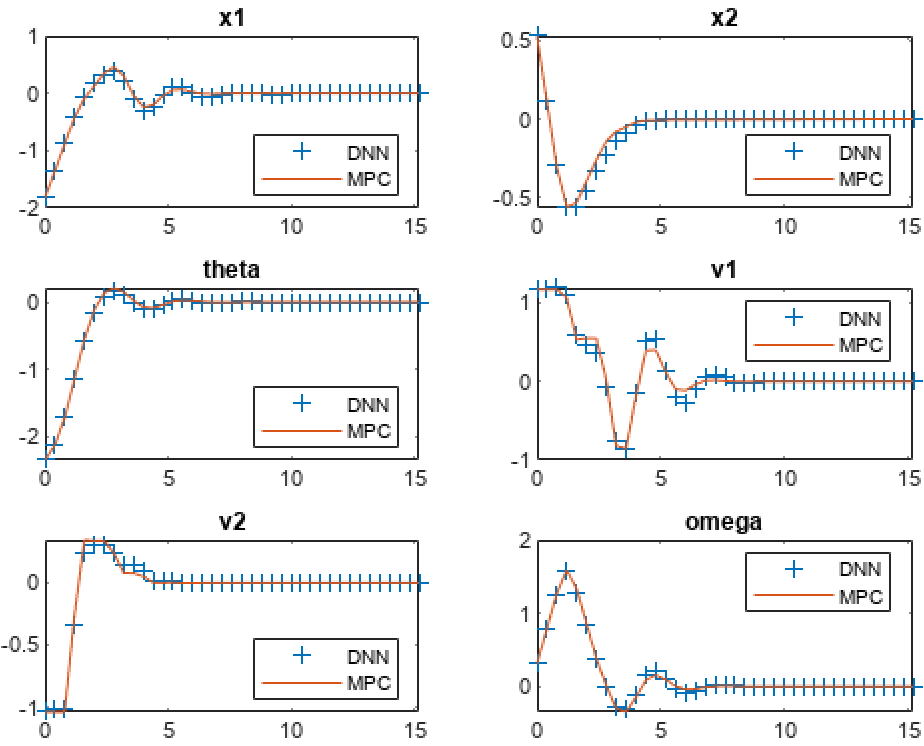
Run a closed-loop simulation of the trained DAgger network.

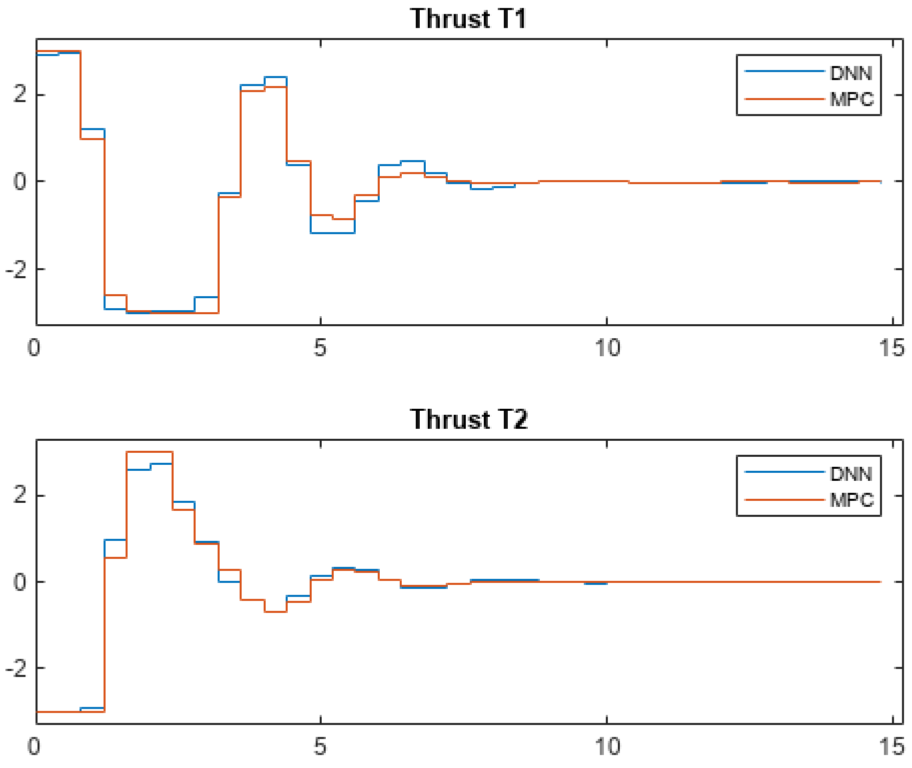
```
tic
[xHistoryDNN,uHistoryDNN] = ...
    simModelDAggerImFlyingRobot(x0,u0,DNN,Ts,Tf);
toc
```

```
Elapsed time is 2.444482 seconds.
```

Plot the results, and compare the NLMPC and trained DNN trajectories.

```
plotSimResultsImFlyingRobot(nlobj, ...
    xHistoryMPC,uHistoryMPC,xHistoryDNN,uHistoryDNN,umax,Tf);
```



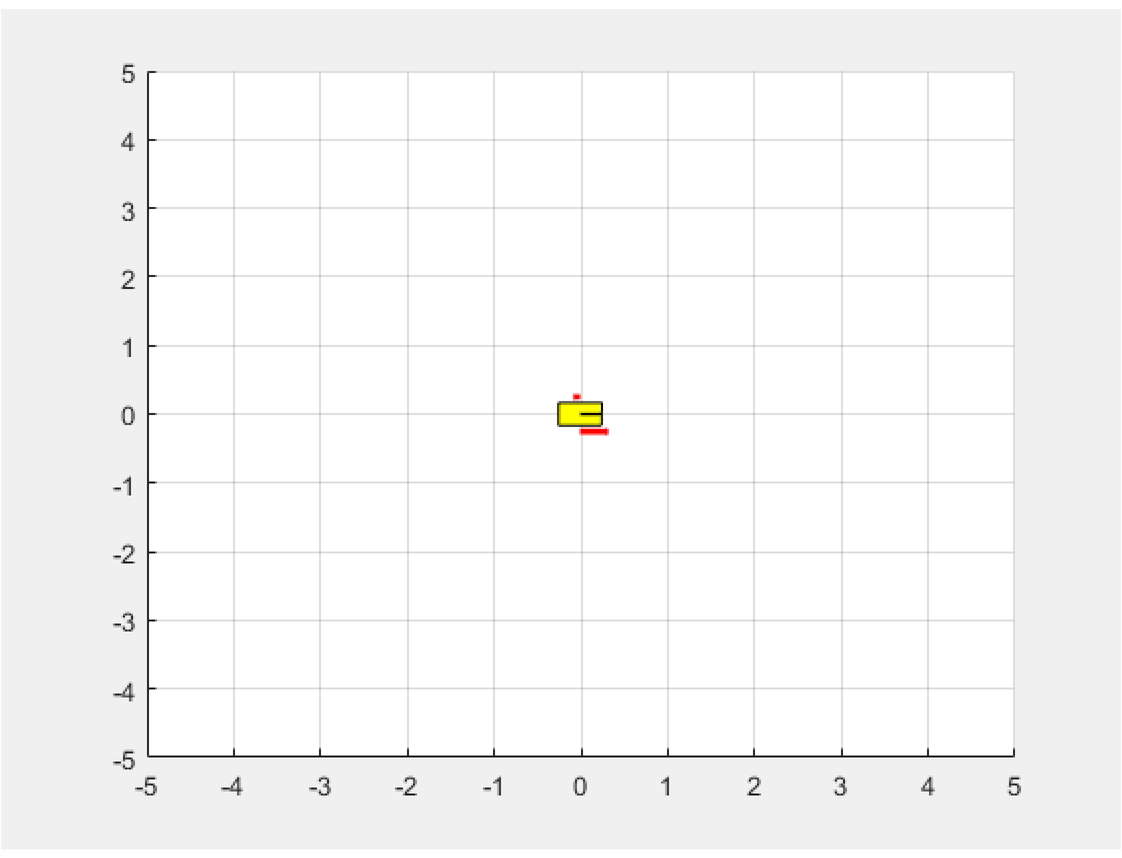



The DAgger neural network successfully imitates the behavior of the NLMPC controller. The flying robot states and control action trajectories for the controller and the DAgger deep neural network closely align. The closed-loop simulation time for the DNN is significantly less than that of the NLMPC controller.

### Animate the Flying Robot with Trained DAgger Network

To validate the performance of the trained DNN, animate the flying robot with data from the DNN closed-loop simulation. The flying robot lands at the origin successfully.

```
Lx = 5;
Ly = 5;
for ct = 1:Tsteps
    x = xHistoryDNN(ct,1);
    y = xHistoryDNN(ct,2);
    theta = xHistoryDNN(ct,3);
    tL = uHistoryDNN(ct,1);
    tR = uHistoryDNN(ct,2);
    rl.env.viz.plotFlyingRobot(x,y,theta,tL,tR,Lx,Ly);
    pause(0.05);
end
```




```
% Turn on MPC messages
mpcverbosity on;
```

### References

[1] Osa, Takayuki, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. 'An Algorithmic Perspective on Imitation Learning'. *Foundations and Trends in Robotics* 7, no. 1–2 (2018): 1–179. https://doi.org/10.1561/2300000053.

## See Also

**Functions**
`trainNetwork` | `predict` | `nlmpcmove`

**Objects**
`SeriesNetwork` | `nlmpc`

## Related Examples

- "Imitate MPC Controller for Lane Keeping Assist" on page 5-365
- "Train DQN Agent for Lane Keeping Assist" on page 5-226
- "Train DDPG Agent with Pretrained Actor Network" on page 5-373
- "Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC" (Model Predictive Control Toolbox)

## More About

- "Nonlinear MPC" (Model Predictive Control Toolbox)

# Tune PI Controller Using Reinforcement Learning

This example shows how to tune the two gains of a PI controller using the twin-delayed deep deterministic policy gradient (TD3) reinforcement learning algorithm. The performance of the tuned controller is compared with that of a controller tuned using the **Control System Tuner** app. Using the **Control System Tuner** app to tune controllers in Simulink® requires Simulink Control Design™ software.

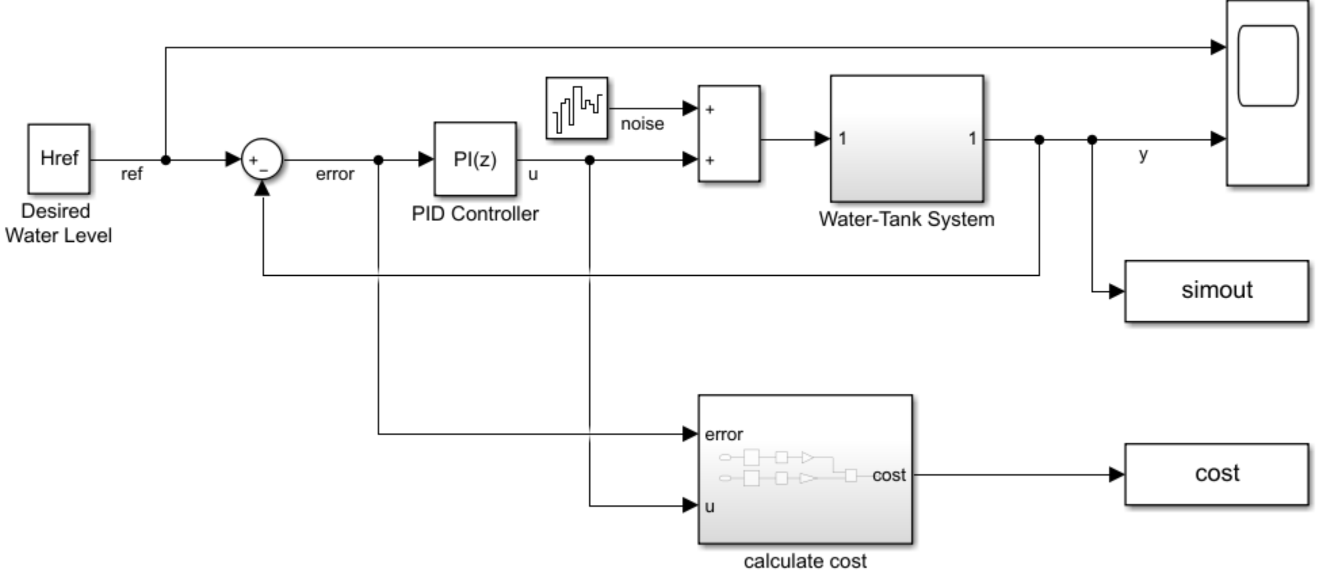
For relatively simple control tasks with a small number of tunable parameters, model-based tuning techniques can get good results with a faster tuning process compared to model-free RL-based methods. However, RL methods can be more suitable for highly nonlinear systems or adaptive controller tuning. To facilitate the controller comparison, both tuning methods use a linear quadratic Gaussian (LQG) objective function. For an example that uses a DDPG agent to implement an LQR controller, see “Train DDPG Agent to Control Double Integrator System” on page 5-77.

This example uses a reinforcement learning (RL) agent to compute the gains for a PI controller. For an example that replaces the PI controller with a neural network controller, see “Create Simulink Environment and Train Agent” on page 1-20.

### Environment Model

The environment model for this example is a water tank model. The goal of this control system is to maintain the level of water in a tank to match a reference value.

```
open_system('watertankLQG')
```




The model includes process noise with variance $E\left(n^2(t)\right) = 1$.

To maintain the water level while minimizing control effort u, the controllers in this example use the following LQG criterion.

$$J = \lim_{T \Rightarrow \infty} E\left(\frac{1}{T}\int_0^T \left((Href - y(t))^2 + 0.01u^2(t)\right)\mathrm{dt}\right)$$

To simulate the controller in this model, you must specify the simulation time `Tf` and the controller sample time `Ts` in seconds.

```
Ts = 0.1;
Tf = 10;
```

For more information about the water tank model, see “watertank Simulink Model” (Simulink Control Design).

### Tune PI Controller Using Control System Tuner

To tune a controller in Simulink using **Control System Tuner**, you must specify the controller block as a tuned block and define the goals for the tuning process. For more information on using **Control System Tuner**, see “Tune a Control System Using Control System Tuner” (Simulink Control Design).

For this example, open the saved session `ControlSystemTunerSession.mat` using **Control System Tuner**. This session specifies the PID Controller block in the `watertankLQG` model as a tuned block and contains an LQG tuning goal.

```
controlSystemTuner("ControlSystemTunerSession")
```

To tune the controller, on the **Tuning** tab, click **Tune**.

The tuned proportional and integral gains are approximately 9.8 and 1e-6, respectively.

```
Kp_CST = 9.80199999804512;
Ki_CST = 1.00019996230706e-06;
```

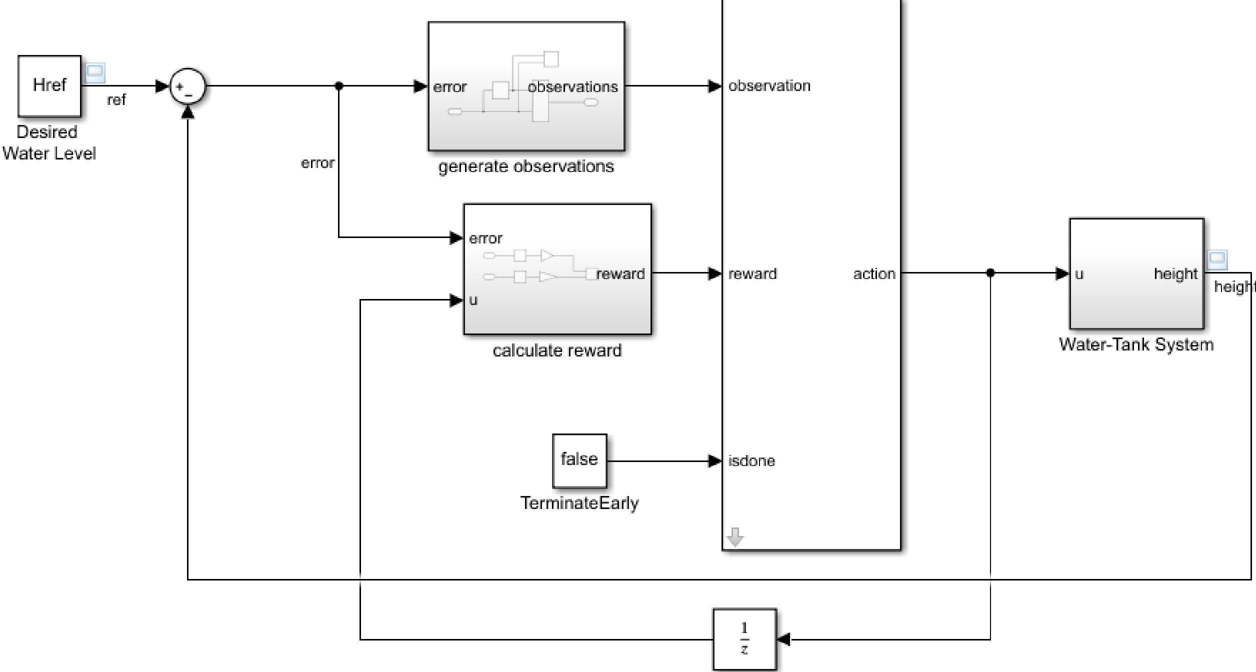
### Create Environment for Training Agent

To define the model for training the RL agent, modify the water tank model using the following steps.

1. Delete the PID Controller.
2. Insert an RL Agent block.
3. Create the observation vector $[\int e\,\mathrm{dt}\;\; e\;]^T$ where $e = Href - h$, $h$ is the height of the water in the tank, and $Href$ is the reference water height. Connect the observation signal to the RL Agent block.
4. Define the reward function for the RL agent as the **negative** of the LQG cost , that is, $\mathrm{Reward} = -\left((Href - h(t))^2 + 0.01u^2(t)\right)$. The RL agent maximizes this reward, thus minimizing the LQG cost.

The resulting model is `rlwatertankPIDTune.slx`.

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




Create the environment interface object. To do so, use the `localCreatePIDEnv` function defined at the end of this example.

```
[env,obsInfo,actInfo] = localCreatePIDEnv(mdl);
```

Extract the observation and action dimensions for this environment. Use `prod(obsInfo.Dimension)` and `prod(actInfo.Dimension)` to return the number of dimensions of the observation and action spaces, respectively, regardless of whether they are arranged as row vectors, column vectors, or matrices.

```
numObs = prod(obsInfo.Dimension);
numAct = prod(actInfo.Dimension);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create TD3 Agent

To create the actor, first create a deep neural network with the observation input and the action output. For more information, see `rlContinuousDeterministicActor`.

You can model a PI controller as a neural network with one fully-connected layer with error and error integral observations.

$$u = \left[ \int e \, \mathrm{dt} \;\; e \right] * [K_i \; K_p]^T$$

Here:

- u is the output of the actor neural network.

- `Kp` and `Ki` are the absolute values of the neural network weights.
- $e = Href - h(t)$, $h(t)$ is the height of the tank, and $Href$ is the reference height.

Gradient descent optimization can drive the weights to negative values. To avoid negative weights, replace normal `fullyConnectedLayer` with a `fullyConnectedPILayer`. This layer ensures that the weights are positive by implementing the function $Y = \text{abs}(\text{WEIGHTS}) * X$. This layer is defined in `fullyConnectedPILayer.m`. For more information on defining custom layers, see "Define Custom Deep Learning Layers".

```
initialGain = single([1e-3 2]);

actorNet = [
    featureInputLayer(numObs)
    fullyConnectedPILayer(initialGain,'ActOutLyr')
    ];

actorNet = dlnetwork(actorNet);


actor = rlContinuousDeterministicActor(actorNet,obsInfo,actInfo);
```

The agent in this example is a twin-delayed deep deterministic policy gradient (TD3) agent. TD3 agents rely on actor and critic approximator objects to learn the optimal policy.

A TD3 agent approximates the long-term reward given observations and actions using two critic value-function representations. To create the critics, first create a deep neural network with two inputs, the observation and action, and one output.

To create the critics, use the `localCreateCriticNetwork` function defined at the end of this example. Use the same network structure for both critic representations.

```
criticNet = localCreateCriticNetwork(numObs,numAct);
```

Create the critic objects using the specified neural network and the environment action and observation specifications. Pass as additional arguments also the names of the network layers to be connected with the observation and action channels.

```
critic1 = rlQValueFunction(dlnetwork(criticNet), ...
    obsInfo,actInfo,...
    ObservationInputNames='stateInLyr', ...
    ActionInputNames='actionInLyr');

critic2 = rlQValueFunction(dlnetwork(criticNet), ...
    obsInfo,actInfo,...
    ObservationInputNames='stateInLyr', ...
    ActionInputNames='actionInLyr');

critic = [critic1 critic2];
```

Configure the agent using the following options.

- Set the agent to use the controller sample time `Ts`.
- Set the mini-batch size to 128 experience samples.
- Set the experience buffer length to 1e6.
- Set the exploration model and target policy smoothing model to use Gaussian noise with variance of 0.1.

Specify training options for the actor and critic.

```
actorOpts = rlOptimizerOptions( ...
    LearnRate=1e-3, ...
    GradientThreshold=1);

criticOpts = rlOptimizerOptions( ...
    LearnRate=1e-3, ...
    GradientThreshold=1);
```

Specify the TD3 agent options using `rlTD3AgentOptions`. Include the training options for the actor and critic.

```
agentOpts = rlTD3AgentOptions(...
    SampleTime=Ts,...
    MiniBatchSize=128, ...
    ExperienceBufferLength=1e6,...
    ActorOptimizerOptions=actorOpts,...
    CriticOptimizerOptions=criticOpts);
```

You can also set or modify the agent options using dot notation.

```
agentOpts.TargetPolicySmoothModel.StandardDeviation = sqrt(0.1);
```

Create the TD3 agent using the specified actor representation, critic representation, and agent options. For more information, see `rlTD3AgentOptions`.

```
agent = rlTD3Agent(actor,critic,agentOpts);
```

### Train Agent

To train the agent, first specify the following training options.

- Run each training for at most 1000 episodes, with each episode lasting at most 100 time steps.
- Display the training progress in the Episode Manager (set the `Plots` option) and disable the command-line display (set the `Verbose` option).
- Stop training when the agent receives an average cumulative reward greater than -355 over 100 consecutive episodes. At this point, the agent can control the level of water in the tank.

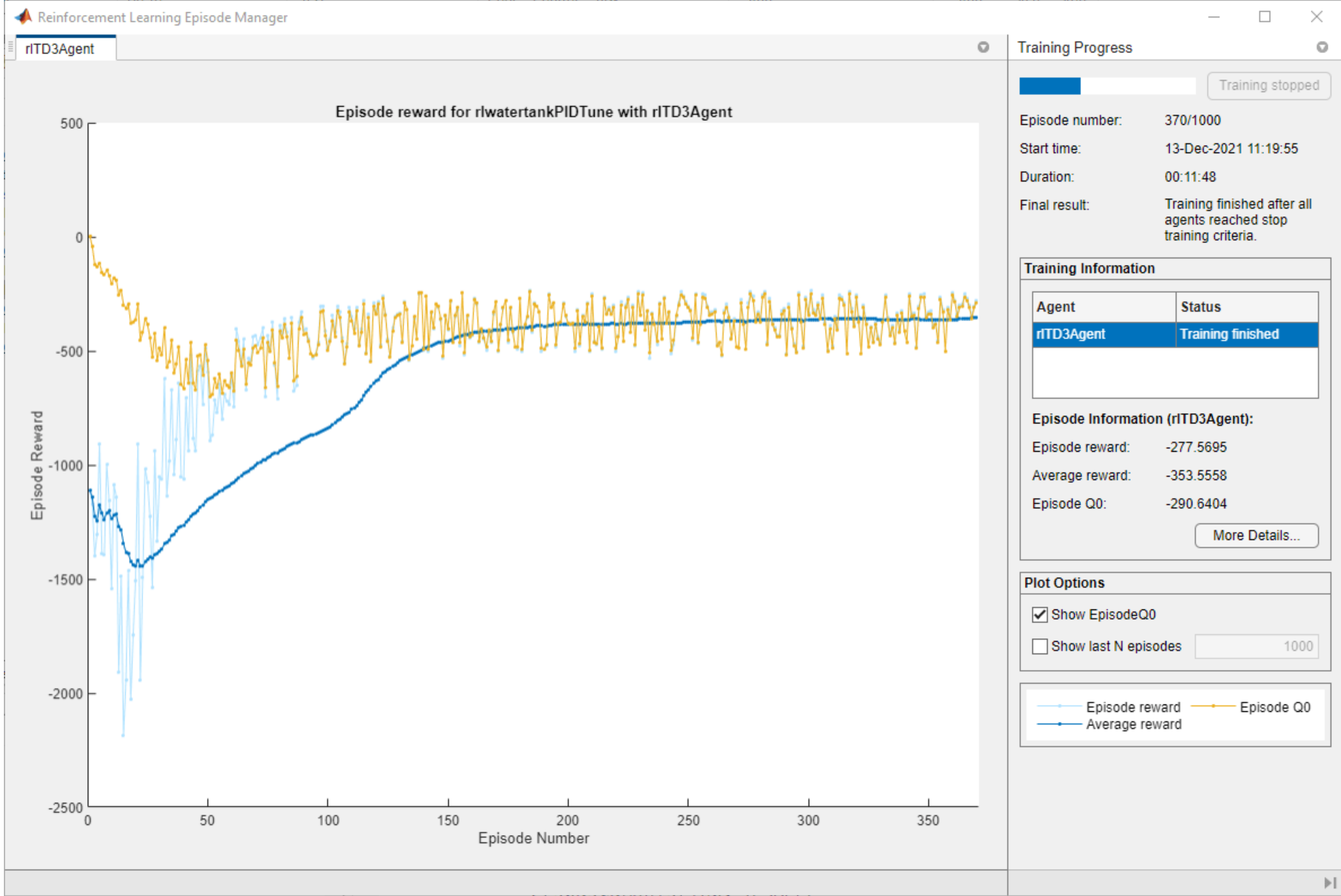
For more information, see `rlTrainingOptions`.

```
maxepisodes = 1000;
maxsteps = ceil(Tf/Ts);
trainOpts = rlTrainingOptions(...
    MaxEpisodes=maxepisodes, ...
    MaxStepsPerEpisode=maxsteps, ...
    ScoreAveragingWindowLength=100, ...
    Verbose=false, ...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=-355);
```

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

```
doTraining = false;
```

```
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load pretrained agent for the example.
    load("WaterTankPIDtd3.mat","agent")
end
```




## Validate Trained Agent

Validate the learned agent against the model by simulation.

```
simOpts = rlSimulationOptions(MaxSteps=maxsteps);
experiences = sim(env,agent,simOpts);
```

The integral and proportional gains of the PI controller are the absolute weights of the actor representation. To obtain the weights, first extract the learnable parameters from the actor.

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

Obtain the controller gains.

```
Ki = abs(parameters{1}(1))
```

```
Ki = single
    0.3958
```

```
Kp = abs(parameters{1}(2))
```

```
Kp = single
    8.0822
```

Apply the gains obtained from the RL agent to the original PI controller block and run a step-response simulation.

```
mdlTest = 'watertankLQG';
open_system(mdlTest);
set_param([mdlTest '/PID Controller'],'P',num2str(Kp))
set_param([mdlTest '/PID Controller'],'I',num2str(Ki))
sim(mdlTest)
```

Extract the step response information, LQG cost, and stability margin for the simulation. To compute the stability margin, use the `localStabilityAnalysis` function defined at the end of this example.

```
rlStep = simout;
rlCost = cost;
rlStabilityMargin = localStabilityAnalysis(mdlTest);
```

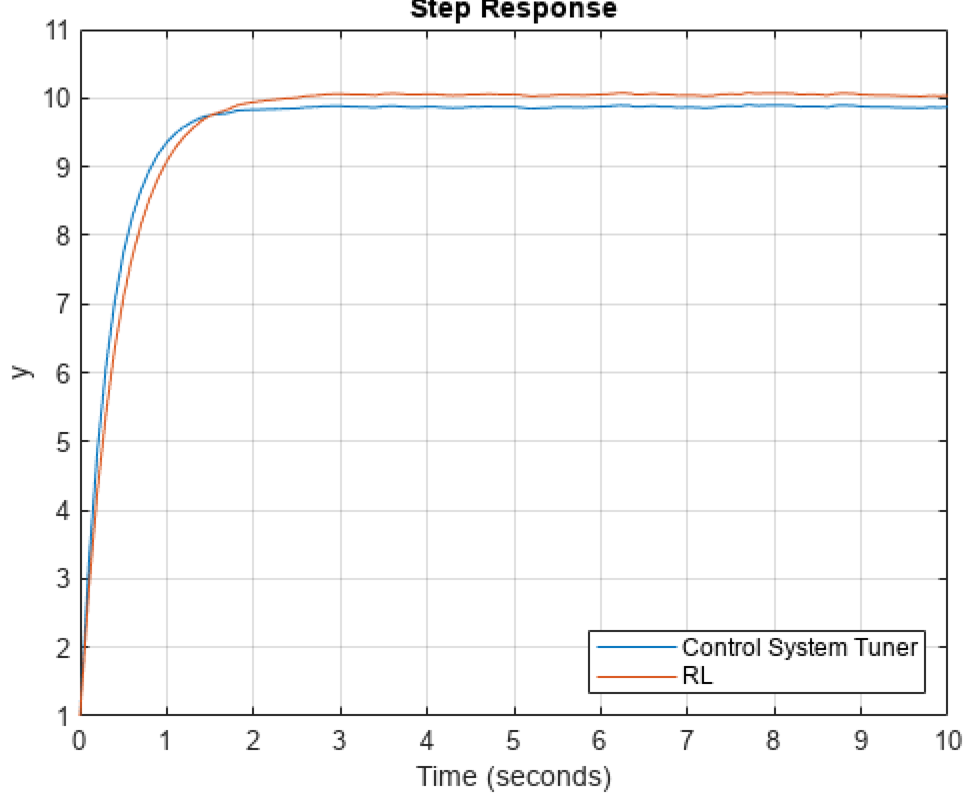
Apply the gains obtained using **Control System Tuner** to the original PI controller block and run a step-response simulation.

```
set_param([mdlTest '/PID Controller'],'P',num2str(Kp_CST))
set_param([mdlTest '/PID Controller'],'I',num2str(Ki_CST))
sim(mdlTest)
cstStep = simout;
cstCost = cost;
cstStabilityMargin = localStabilityAnalysis(mdlTest);
```

#### Compare Controller Performance

Plot the step response for each system.

```
figure
plot(cstStep)
hold on
plot(rlStep)
grid on
legend('Control System Tuner','RL',Location="southeast")
title('Step Response')
```




Analyze the step response for both simulations.

```
rlStepInfo = stepinfo(rlStep.Data,rlStep.Time);
cstStepInfo = stepinfo(cstStep.Data,cstStep.Time);
stepInfoTable = struct2table([cstStepInfo rlStepInfo]);
stepInfoTable = removevars(stepInfoTable,{'SettlingMin', ...
    'TransientTime','SettlingMax','Undershoot','PeakTime'});
stepInfoTable.Properties.RowNames = {'CST','RL'};
stepInfoTable
```

```
stepInfoTable=2×4 table
           RiseTime    SettlingTime    Overshoot     Peak 
           ________    ____________    _________    ______

    CST    0.77737        1.3278        0.33125     9.9023
    RL     0.98024        1.7073        0.40451     10.077
```

Analyze the stability for both simulations.

```
stabilityMarginTable = struct2table( ...
    [cstStabilityMargin rlStabilityMargin]);
stabilityMarginTable = removevars(stabilityMarginTable,{...
    'GMFrequency','PMFrequency','DelayMargin','DMFrequency'});
stabilityMarginTable.Properties.RowNames = {'CST','RL'};
stabilityMarginTable
```

```
stabilityMarginTable=2×3 table
           GainMargin    PhaseMargin    Stable
```

```
              __________    ___________    ______

    CST       8.1616        84.124         true
    RL        9.9226        84.241         true
```

Compare the cumulative LQG cost for the two controllers. The RL-tuned controller produces a slightly more optimal solution.

```
rlCumulativeCost  = sum(rlCost.Data)
```

```
rlCumulativeCost = -375.9135
```

```
cstCumulativeCost = sum(cstCost.Data)
```

```
cstCumulativeCost = -376.9373
```

Both controllers produce stable responses, with the controller tuned using **Control System Tuner** producing a faster response. However, the RL tuning method produces a higher gain margin and a more optimal solution.

### Local Functions

Function to create the water tank RL environment.

```
function [env,obsInfo,actInfo] = localCreatePIDEnv(mdl)

% Define the observation specification obsInfo
% and the action specification actInfo.
obsInfo = rlNumericSpec([2 1]);
obsInfo.Name = 'observations';
obsInfo.Description = 'integrated error and error';

actInfo = rlNumericSpec([1 1]);
actInfo.Name = 'PID output';

% Build the environment interface object.
env = rlSimulinkEnv(mdl,[mdl '/RL Agent'],obsInfo,actInfo);

% Set a cutom reset function that randomizes
% the reference values for the model.
env.ResetFcn = @(in)localResetFcn(in,mdl);
end
```

Function to randomize the reference signal and initial height of the water tank at the beginning of each episode.

```
function in = localResetFcn(in,mdl)

% Randomize reference signal
blk = sprintf([mdl '/Desired \nWater Level']);
hRef = 10 + 4*(rand-0.5);
in = setBlockParameter(in,blk,'Value',num2str(hRef));

% Randomize initial height
hInit = 0;
blk = [mdl '/Water-Tank System/H'];
in = setBlockParameter(in,blk,'InitialCondition',num2str(hInit));
```

```
end
```

Function to linearize and compute stability margins of the SISO water tank system.

```
function margin = localStabilityAnalysis(mdl)

io(1) = linio([mdl '/Sum1'],1,'input');
io(2) = linio([mdl '/Water-Tank System'],1,'openoutput');
op = operpoint(mdl);
op.Time = 5;
linsys = linearize(mdl,io,op);

margin = allmargin(linsys);
end
```

Function to create critic network.

```
function criticNet = localCreateCriticNetwork(numObs,numAct)
statePath = [
    featureInputLayer(numObs,Name='stateInLyr')
    fullyConnectedLayer(32,Name='fc1')];

actionPath = [
    featureInputLayer(numAct,Name='actionInLyr')
    fullyConnectedLayer(32,Name='fc2')];

commonPath = [
    concatenationLayer(1,2,Name='concat')
    reluLayer
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(1,Name='qvalOutLyr')];

criticNet = layerGraph();
criticNet = addLayers(criticNet,statePath);
criticNet = addLayers(criticNet,actionPath);
criticNet = addLayers(criticNet,commonPath);

criticNet = connectLayers(criticNet,'fc1','concat/in1');
criticNet = connectLayers(criticNet,'fc2','concat/in2');
end
```

## See Also

**Functions**
train | sim | rlSimulinkEnv

**Objects**
rlTD3Agent | rlTD3AgentOptions | rlTrainingOptions

## Related Examples

- “Train DDPG Agent to Control Double Integrator System” on page 5-77
- “Train TD3 Agent for PMSM Control” on page 5-341

- “Train Custom LQR Agent” on page 5-466

## More About

- “Twin-Delayed Deep Deterministic (TD3) Policy Gradient Agents” on page 3-44
- “Define Custom Deep Learning Layers”
- “Train Reinforcement Learning Agents” on page 5-3

# Train Reinforcement Learning Agent with Constraint Enforcement

This example shows how to train a reinforcement learning (RL) agent with actions constrained using the Constraint Enforcement block. This block computes modified control actions that are closest to the actions output by the agent subject to constraints and action bounds. Training reinforcement learning agents requires Reinforcement Learning Toolbox™ software.

In this example, the goal of the agent is to bring a green ball as close as possible to the changing target position of a red ball [1].

The dynamics for the green ball from velocity $v$ to position $x$ are governed by Newton's law with a small damping coefficient $\tau$:

$$\frac{1}{s(\tau s + 1)}.$$

The feasible region for the ball position $0 \leq x \leq 1$ and the velocity of the green ball is limited to the range $[-1, 1]$.

The position of the target red ball is uniformly random across the range $[0, 1]$. The agent can observe only a noisy estimate of this target position.

Set the random seed to ensure reproducibility.

```
rng("default")
```

Configure model parameters.

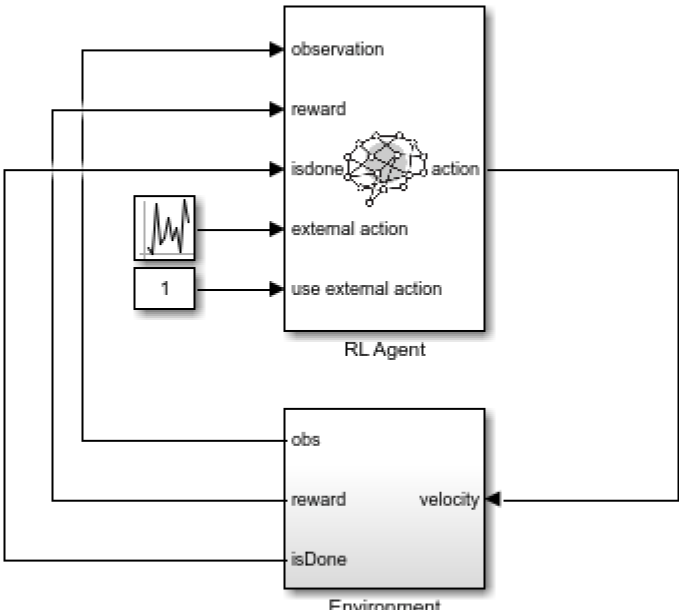
```
Tv = 0.8;            % sample time for visualizer
Ts = 0.1;            % sample time for controller
tau = 0.01;          % damping constant for green ball
velLimit = 1;        % maximum speed for green ball
s0 = 200;            % random seed
s1 = 100;            % random seed
x0 = 0.2;            % initial position for ball
```

### Create Environment and Agent for Collecting Data

In this example, a constraint function is represented using a trained deep neural network. To train the network, you must first collect training data from the environment.

To do so, first create an RL environment using the `rlBallOneDim` model. This model applies random external actions through an RL Agent block to the environment.

```
mdl = "rlBallOneDim";
open_system(mdl)
```




The Environment subsystem performs the following steps.

- Applies the input velocity to the environment model and generates the resulting output observations
- Computes the training reward $r = \left[1 - 10(x - x_r)^2\right]^+$, where $x_r$ denotes the position of the red ball
- Sets the termination signal `isDone` to `true` if the ball position violates the constraint $0 \le x \le 1$

For this model, the observations from the environment include the position and velocity of the green ball and the noisy measurement of the red ball position. Define a continuous observation space for these three values.

```
obsInfo = rlNumericSpec([3 1]);
```

The action that the agent applies to the green ball is its velocity. Create a continuous action space and apply the required velocity limits.

```
actInfo = rlNumericSpec([1 1], ...
    LowerLimit=-velLimit, ...
    UpperLimit=velLimit);
```

Create an RL environment for this model.

```
agentblk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
```

Specify a reset function, which randomly initializes the environment at the start of each training episode or simulation.

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

Next, create a DDPG reinforcement learning agent, which supports continuous actions and observations, using the `createDDPGAgentBall` helper function. This function creates critic and

actor representations based on the action and observation specifications and uses the representations to create a DDPG agent.

```
agent = createDDPGAgentBall(Ts,obsInfo,actInfo);
```

In the `rlBallOneDim` model, the RL Agent block does not generate actions. Instead, it is configured to pass a random external action to the environment. The purpose for using a data-collection model with an inactive RL Agent block is to ensure that the environment model, action and observation signal configurations, and model reset function used during data collection match those used during subsequent agent training.

### Learn Constraint Function

In this example, the ball position signal $x_{k+1}$ must satisfy $0 \le x_{k+1} \le 1$. To allow for some slack, the constraint is set to be $0.1 \le x_{k+1} \le 0.9$. The dynamic model from velocity to position has a very small damping constant, thus it can be approximated by $x_{k+1} \approx x_k + h(x_k)u_k$. Therefore, the constraints for green ball are given by the following equation.

$$\begin{bmatrix} x_k \\ -x_k \end{bmatrix} + \begin{bmatrix} h(x_k) \\ -h(x_k) \end{bmatrix} u_k \le \begin{bmatrix} 0.9 \\ -0.1 \end{bmatrix}$$

The Constraint Enforcement block accepts constraints of the form $f_x + g_x u \le c$. For the above equation, the coefficients of this constraint function are as follows.

$$f_x = \begin{bmatrix} x_k \\ -x_k \end{bmatrix}, g_x = \begin{bmatrix} h(x_k) \\ -h(x_k) \end{bmatrix}, c = \begin{bmatrix} 0.9 \\ -0.1 \end{bmatrix}$$

The function $h(x_k)$ is approximated by a deep neural network that is trained on the data collected by simulating the RL agent within the environment. To learn the unknown function $h(x_k)$, the RL agent passes a random external action to the environment that is uniformly distributed in the range $[-1, 1]$.

To collect data, use the `collectDataBall` helper function. This function simulates the environment and agent and collects the resulting input and output data. The resulting training data has three columns: $x_k$, $u_k$, and $x_{k+1}$.

For this example, load precollected training data. To collect the data yourself, set `collectData` to `true`.

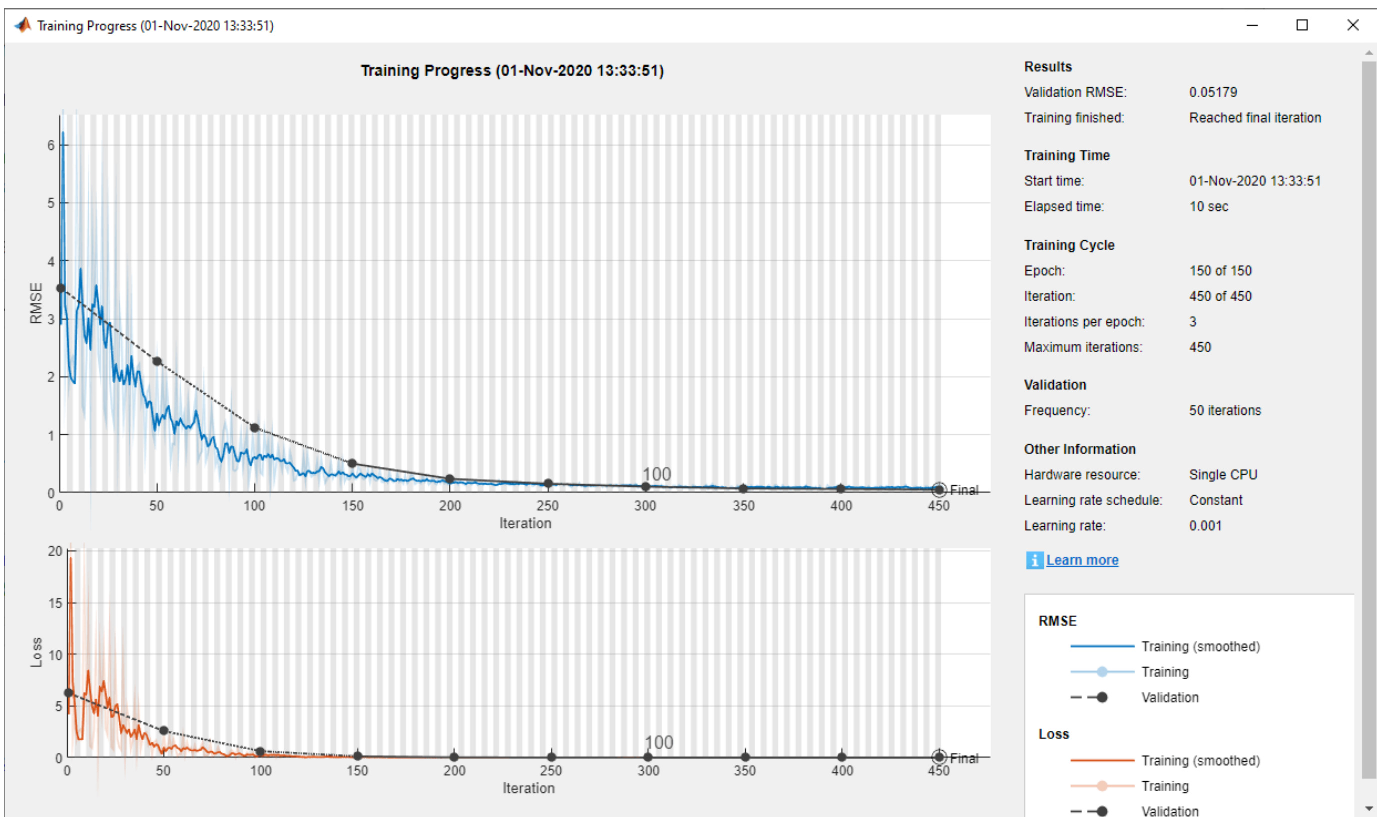
```
collectData = false;
if collectData
    count = 1050;
    data = collectDataBall(env,agent,count);
else
    load trainingDataBall data
end
```

Train a deep neural network to approximate the constraint function using the `trainConstraintBall` helper function. This function formats the data for training then creates and trains a deep neural network. Training a deep neural network requires Deep Learning Toolbox™ software.

For this example, to ensure reproducibility, load a pretrained network. To train the network yourself, set `trainConstraint` to `true`.

```
trainConstraint = false;
if trainConstraint
    network = trainConstraintBall(data);
else
    load trainedNetworkBall network
end
```

The following figure shows an example of the training progress.




Validate the trained neural network using the `validateNetworkBall` helper function. This function processes the input training data using the trained deep neural network. It then compares the network output with the training output and computes the root mean-squared error (RMSE).

```
validateNetworkBall(data,network)
```

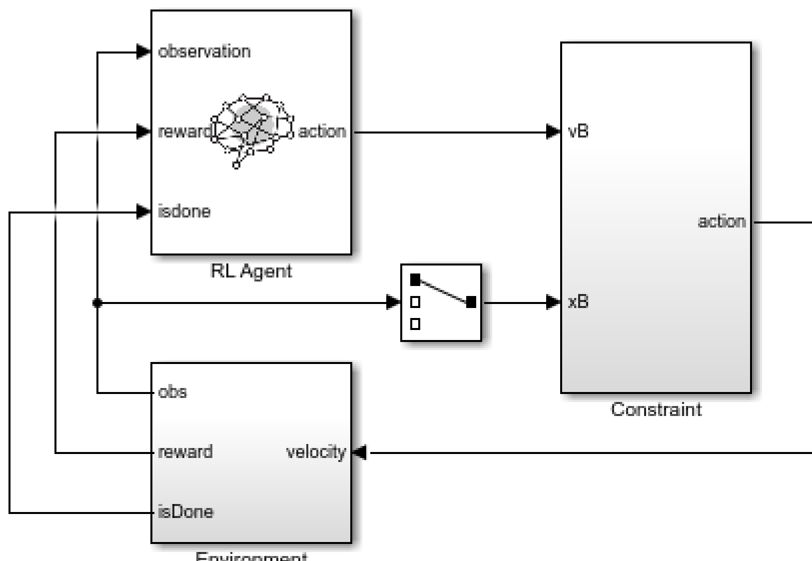
```
Test Data RMSE = 9.996700e-02
```

The small RMSE value indicates that the network successfully learned the constraint function.

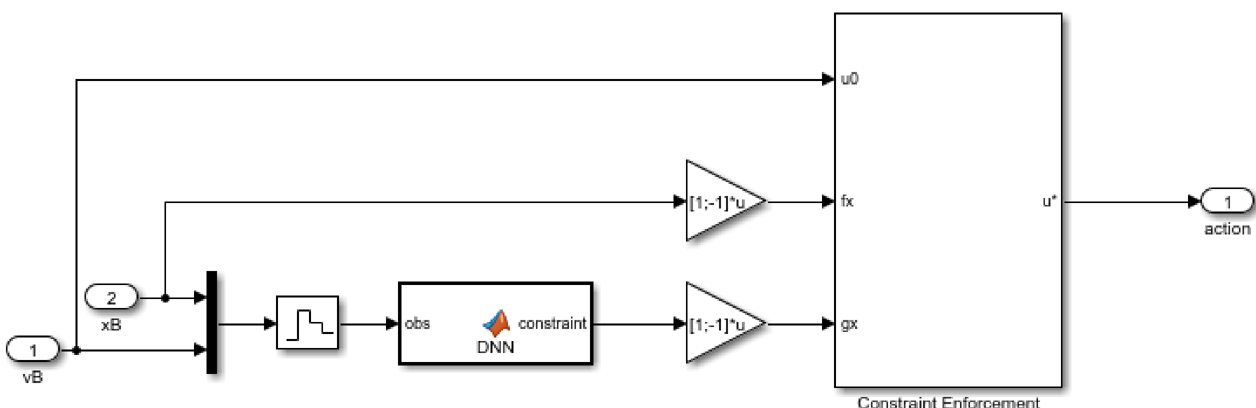
### Train Agent with Constraint Enforcement

To train the agent with constraint enforcement, use the `rlBallOneDimWithConstraint` model. This model constrains the actions from the agent before applying them to the environment.

```
mdl = "rlBallOneDimWithConstraint";
open_system(mdl)
```




To view the constraint implementation, open the Constraint subsystem. Here, the trained deep neural network approximates $h(x_k)$, and the Constraint Enforcement block enforces the constraint function and velocity bounds.




For this example the following Constraint Enforcement block parameter settings are used.

- **Number of constraints** — 2
- **Number of actions** — 1
- **Constraint bound** — `[0.9;-0.1]`

Create an RL environment using this model. The observation and action specifications match those used for the previous data collection environment.

```
agentblk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
env.ResetFcn = @(in)localResetFcn(in);
```

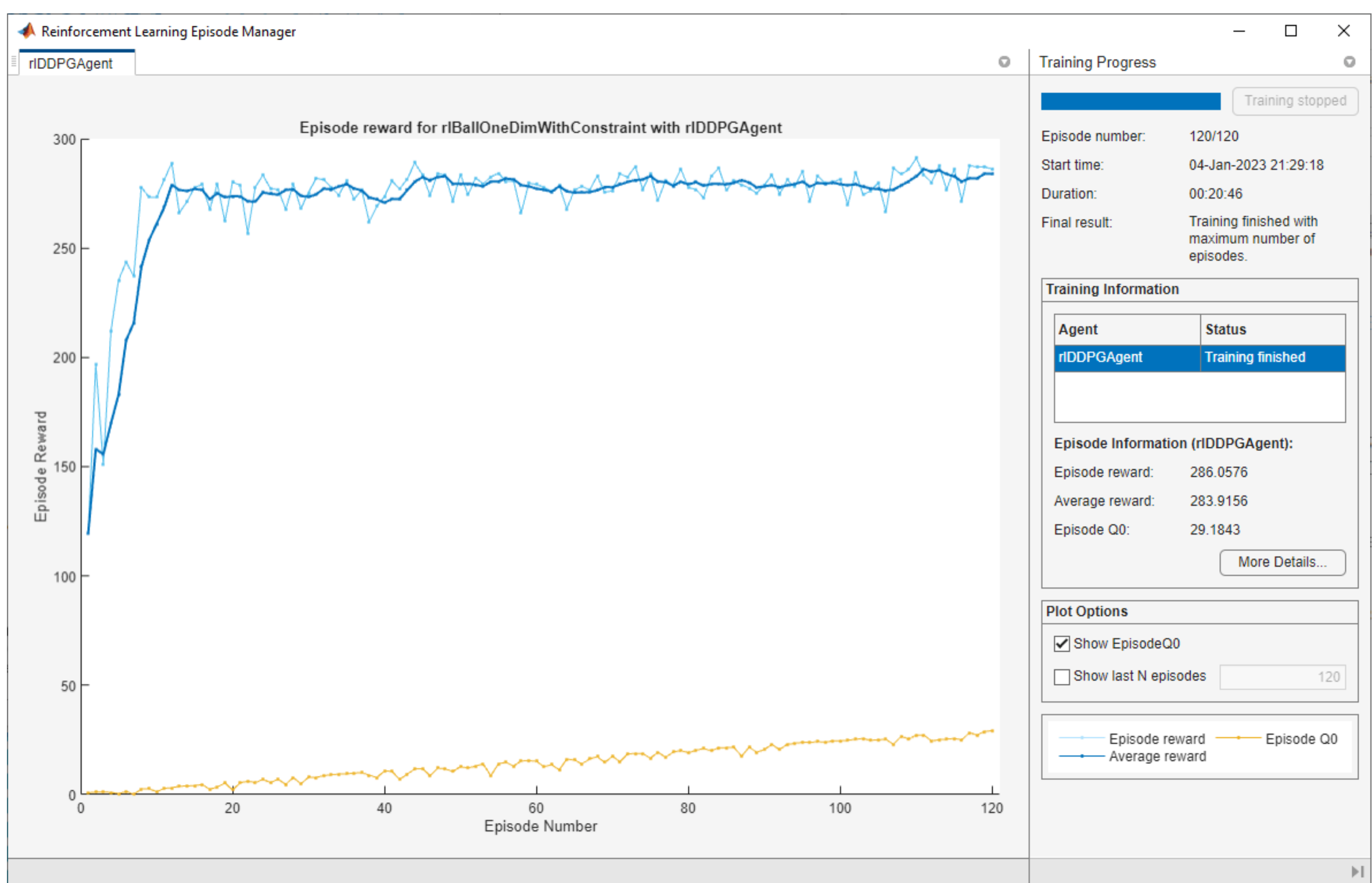
Specify options for training the agent. Train the RL agent for 300 episodes with 300 steps per episode.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=120, ...
    MaxStepsPerEpisode=300, ...
    Verbose=false, ...
    Plots="training-progress");
```

Train the agent. Training is a time-consuming process. For this example, load a pretrained agent. To train the agent yourself, set `trainAgent` to `true`.

```
trainAgent = false;
if trainAgent
    trainingStats = train(agent,env,trainOpts);
else
    load("rlAgentBallParams.mat","agent")
end
```

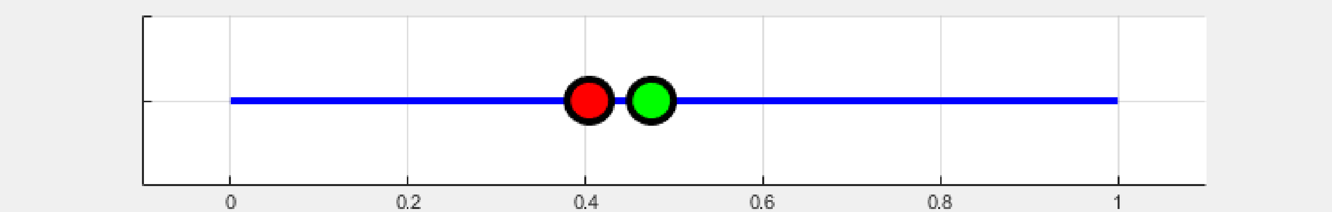
The following figure shows the training results. The training process converges to a good agent within 20 episodes.

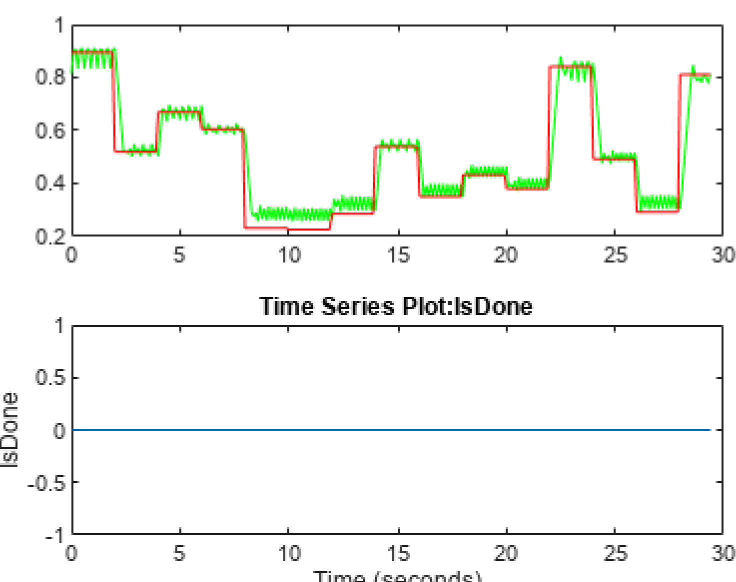



Since **Total Number of Steps** equals the product of **Episode Number** and **Episode Steps**, each training episode runs to the end without early termination. Therefore, the Constraint Enforcement block ensures that the ball position $x$ never violates the constraint $0 \leq x \leq 1$.

Simulate the trained agent using the `simWithTrainedAgentBall` helper function.

```
simWithTrainedAgentBall(env,agent)
```



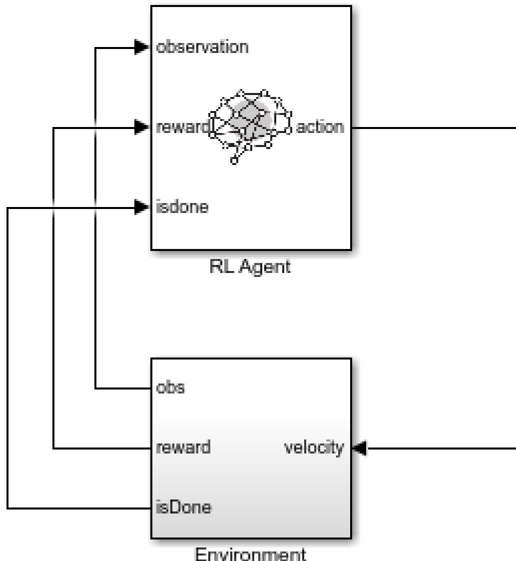

The agent successfully tracks the position of the red ball.

### Train Agent Without Constraint Enforcement

To see the benefit of training an agent with constraint enforcement, you can train the agent without constraints and compare the training results to the constraint enforcement case.

To train the agent without constraints, use the `rlBallOneDimWithoutConstraint` model. This model applies the actions from the agent directly to the environment.

```
mdl = "rlBallOneDimWithoutConstraint";
open_system(mdl)
```




Create an RL environment using this model.

```
agentblk = mdl + "/RL Agent";
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
env.ResetFcn = @(in)localResetFcn(in);
```

Create a new DDPG agent to train. This agent has the same configuration as the agent used in the previous training.

```
agent = createDDPGAgentBall(Ts,obsInfo,actInfo);
```

Train the agent using the same training options as in the constraint enforcement case. For this example, as with the previous training, load a pretrained agent. To train the agent yourself, set `trainAgent` to true.

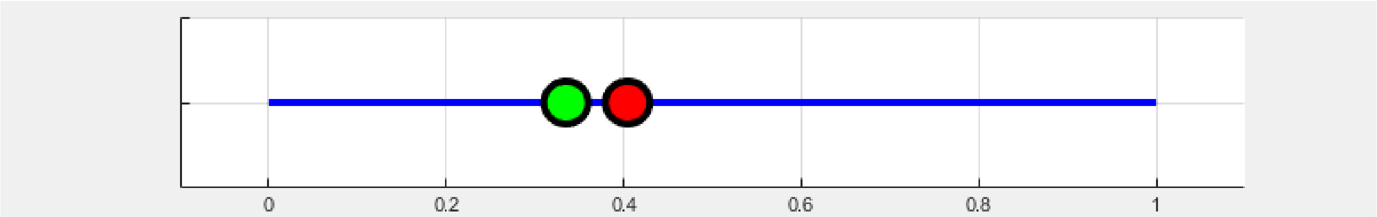
```
trainAgent = false;
if trainAgent
    trainingStats2 = train(agent,env,trainOpts);
else
    load("rlAgentBallCompParams.mat","agent")
end
```

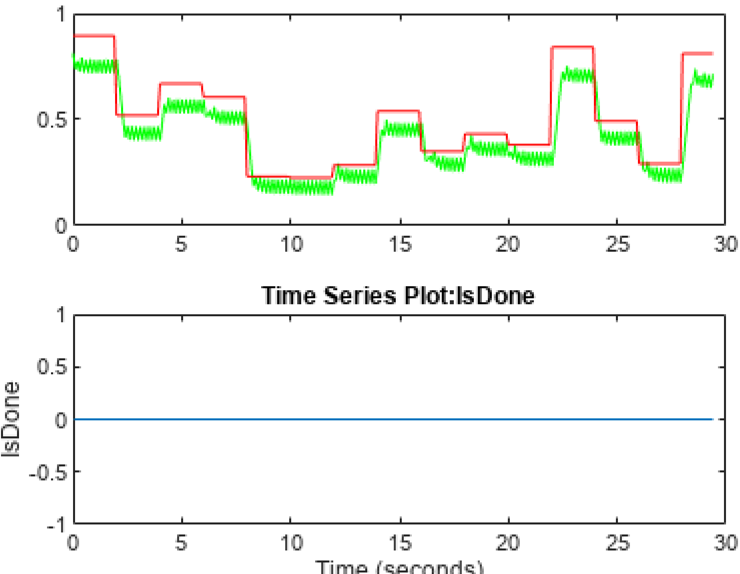
The following figure shows the training results. The training process converges to a good agent after 50 episodes.

Since **Total Number of Steps** is less than the product of **Episode Number** and **Episode Steps**, the training includes episodes that terminated early due to constraint violations.

Simulate the trained agent.

```
simWithTrainedAgentBall(env,agent)
```






The agent tracks the position of the red ball with more steady-state offset than the agent trained with constraints.

### Conclusion

In this example, training an RL agent with the Constraint Enforcement block ensures that the actions applied to the environment never produce a constraint violation. As a result, the training process converges to a good agent quickly. Training the same agent without constraints produces slower convergence and poorer performance.

```
bdclose("rlBallOneDim")
bdclose("rlBallOneDimWithConstraint")
bdclose("rlBallOneDimWithoutConstraint")
close("Ball One Dim")
```

### Local Reset Function

```
function in = localResetFcn(in)
% Reset function
in = setVariable(in,"x0",rand);
in = setVariable(in,"s0",randi(5000));
in = setVariable(in,"s1",randi(5000));
end
```

### References

[1] Dalal, Gal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. "Safe Exploration in Continuous Action Spaces." Preprint, submitted January 26, 2018. https://arxiv.org/abs/1801.08757

## See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlDDPGAgent` | `rlDDPGAgentOptions` | `rlTrainingOptions`

**Blocks**
RL Agent | Constraint Enforcement

## Related Examples

- "Train RL Agent for Adaptive Cruise Control with Constraint Enforcement" (Simulink Control Design)
- "Train RL Agent for Lane Keeping Assist with Constraint Enforcement" (Simulink Control Design)
- "Train DDPG Agent for Path-Following Control" on page 5-247

## More About

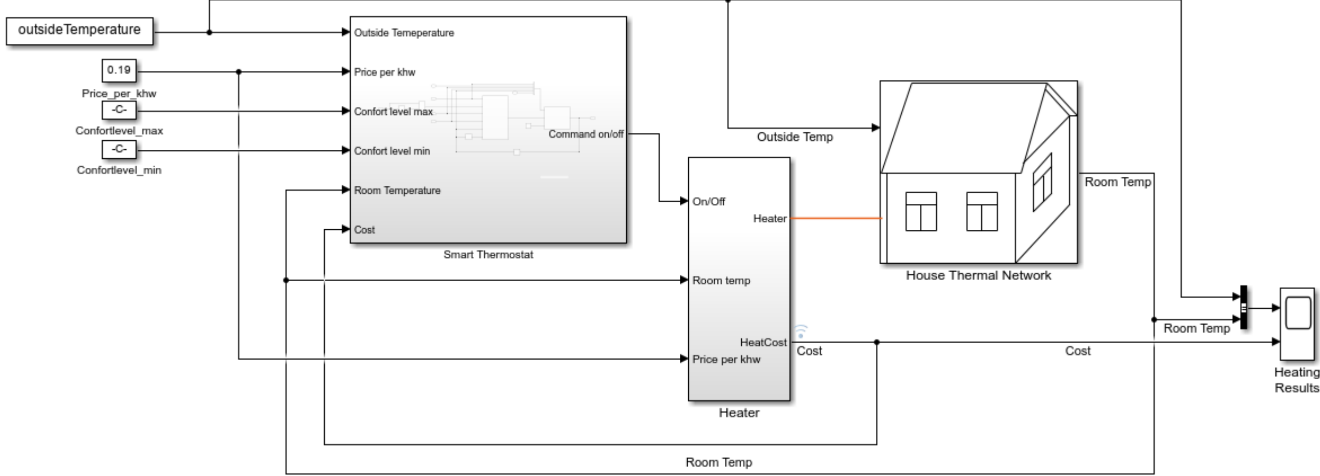
- "Constraint Enforcement for Control Design" (Simulink Control Design)
- "Deep Deterministic Policy Gradient (DDPG) Agents" on page 3-40
- "Train Reinforcement Learning Agents" on page 5-3

# Train DQN Agent with LSTM Network to Control House Heating System

This example shows how to train a deep Q-learning network (DQN) agent with a Long Short-Term Memory (LSTM) network to control a house heating system modeled in Simscape®. For more information on DQN agents, see “Deep Q-Network (DQN) Agents” on page 3-23.

### House Heating Model

The reinforcement learning (RL) environment for this example uses a model from the “House Heating System” (Simscape) example. The model in this example contains a heater, a thermostat controlled by an RL agent, a house, outside temperatures, and a reward function. Heat is transferred between the outside environment and the interior of the home through the walls, windows, and roof. Weather station data between March 21 and April 15, 2022, from the MathWorks® campus in Natick, MA, is used to simulate the outside temperature. ThingSpeak™ was used to obtain this data. The data file `temperatureMar21toApr15_20022.mat`, is located in this example folder. For more information about the data acquisition, see “Compare Temperature Data from Three Different Days” (ThingSpeak).




The training goal for the agent is to minimize the energy cost and maximize the comfort of the room by turning on/off the heater. The house is comfortable when the room temperature $T_{\text{room}}$is between $T_{\text{comfortMin}}$ and $T_{\text{comfortMax}}$.

The reinforcement learning problem is defined as follows:

- The observation is a 6-dimensional column vector that consists of the room temperature $(^\circ C)$ , outside temperature $(^\circ C)$ , maximum comfort temperature $(^\circ C)$ , minimum comfort temperature $(^\circ C)$ , last action, and price per kWh (USD). The maximum and minimum comfort temperatures and price per kWh in this example do not change over time, and you do not need to use them to train the agent. However, you can extend this example by varying these values over time.
- The action is discrete, and represents either turning the heater off (action is 0) or turning it on (action is 1).

- The total reward $R$ is calculated as the sum of the comfort reward $Rc$ and a penalty for switching $Ps$, to which an energy cost $Ce$ is subtracted: $R = Rc + Ps - Ce$. These three components are defines as follows.

$$Rc = \begin{cases} 0.1 & \text{if } T_{\text{comfortMin}} \leq T_{\text{room}} \leq T_{\text{comfortMax}} \\ -w|T_{\text{room}} - T_{\text{comfortMin}}| & \text{if } T_{\text{room}} < T_{\text{comfortMin}} \\ -w|T_{\text{room}} - T_{\text{comfortMax}}| & \text{if } T_{\text{room}} > T_{\text{comfortMax}} \end{cases}$$

where $w = 0.1$, $T_{\text{comfortMin}} = 18$, and $T_{\text{comfortMax}} = 23$.

$$Ps = \begin{cases} -0.01 & \text{if } a_t \neq a_{t-1} \text{ where } a_t \text{ is the current action and } a_{t-1} \text{ is the previous action} \\ 0 & \text{otherwise} \end{cases}$$

$$Ce = \text{CostPerStep} = \text{PricePerKwh} * \text{ElectricityUsed}$$

The reward function is inspired by [1].

- **Is-Done signal** is always `0`, indicating that there is no early termination condition.

Open the model.

```
mdl = "rlHouseHeatingSystem";
open_system(mdl)
```

Define a sample time and the maximum number of steps per episode. Both variables are needed in the script and in the reset function.

```
sampleTime = 120; % seconds
maxStepsPerEpisode = 1000;
```

Assign the agent block path information to the `agentBlk` variable, for later use.

```
agentBlk = mdl + "/Smart Thermostat/RL Agent";
```

Load the outside temperature data to simulate the environment temperature.

```
data = load('temperatureMar21toApr15_2022.mat');
temperatureData = data.temperatureData;
```

Extract validation data.

```
temperatureMarch21 = temperatureData(1:60*24,:);
temperatureApril15 = temperatureData(end-60*24+1:end,:);
```

Extract training data.

```
 temperatureData = temperatureData(60*24+1:end-60*24,:);
```

The Simulink® model loads the following variables as a part of observations.

```
outsideTemperature = temperatureData;
comfortMax = 23;
comfortMin = 18;
```

Define observation and action specifications.

```
obsInfo = rlNumericSpec([6,1]);
actInfo = rlFiniteSetSpec([0,1]); % (0=off,1=on)
```

The actor and critic networks of the DQN agent are initialized randomly. Ensure reproducibility by fixing the seed of the random generator.

```
rng(0)
```

#### Create DQN Agent with LSTM Network

DQN agents use a parametrized Q-value function approximator to estimate the value of the policy. Since DQN agents have a discrete action space, you have the option to create a vector (that is multi-output) Q-value function critic, which is generally more efficient than a comparable single-output critic.

A vector Q-value function takes only the observation as input and returns as output a single vector with as many elements as the number of possible actions. The value of each output element represents the expected discounted cumulative long-term reward when an agent starts from the state corresponding to the given observation and executes the action corresponding to the element number (and follows a given policy afterwards).

Specify training options for the critic and the actor using `rlOptimizerOptions`.

```
criticOpts = rlOptimizerOptions( ...
    LearnRate=0.001, ...
    GradientThreshold=1);
```

Specify the DQN agent options using `rlDQNAgentOptions`, include the training options for the critic.

```
agentOpts = rlDQNAgentOptions(...
    UseDoubleDQN = false, ...
    TargetSmoothFactor = 1, ...
    TargetUpdateFrequency = 4, ...
    ExperienceBufferLength = 1e6, ...
    CriticOptimizerOptions = criticOpts, ...
    MiniBatchSize = 64);
```

You can also set or modify the agent options using dot notation.

```
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 0.0001;
```

To model the parametrized Q-value function within the critic, use a recurrent neural network, which can capture the effect of previous observations. By setting the `UseRNN` option in `rlAgentInitializationOptions`, you can create a default DQN agent with an LSTM network. Alternatively, you can manually configure the LSTM network. See “Water Distribution System Scheduling Using Reinforcement Learning” on page 5-353 to create an LSTM network for the DQN agent manually.

```
useRNN = true;
initOpts = rlAgentInitializationOptions( ...
    UseRNN=useRNN, ...
    NumHiddenUnit=64);
```

When using a recurrent neural network, you must set `SequenceLength` to be greater than `1` in `rlDQNAgentOptions`. This option is used in training to determine the length of the minibatch used

to calculate the gradient. For more information about LSTM layers, see "Long Short-Term Memory Neural Networks".

```
if useRNN
    agentOpts.SequenceLength = 20;
end
```

Create a DQN agent with default critic network using `rlDQNAgent`, the agent initalization options object (which specify that a recurrent network is needed), and the agent options object.

```
agent = rlDQNAgent(obsInfo, actInfo, initOpts, agentOpts);
```

You can also modify the agent options, including the critic options, using dot notation. For example, specify the agent sample time.

```
agent.SampleTime = sampleTime;
```

### Define Simulink Environment

Create an environment interface for the house heating environment.

```
env = rlSimulinkEnv(mdl,agentBlk,obsInfo,actInfo);
```

Use `hRLHeatingSystemResetFcn` to reset the environment at the beginning of each episode. This function randomly selects the time between March 22nd and April 14th. The environment uses this time as the initial time for the outside temperatures.

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

Validate the environment.

```
validateEnvironment(env)
```

### Train Agent

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

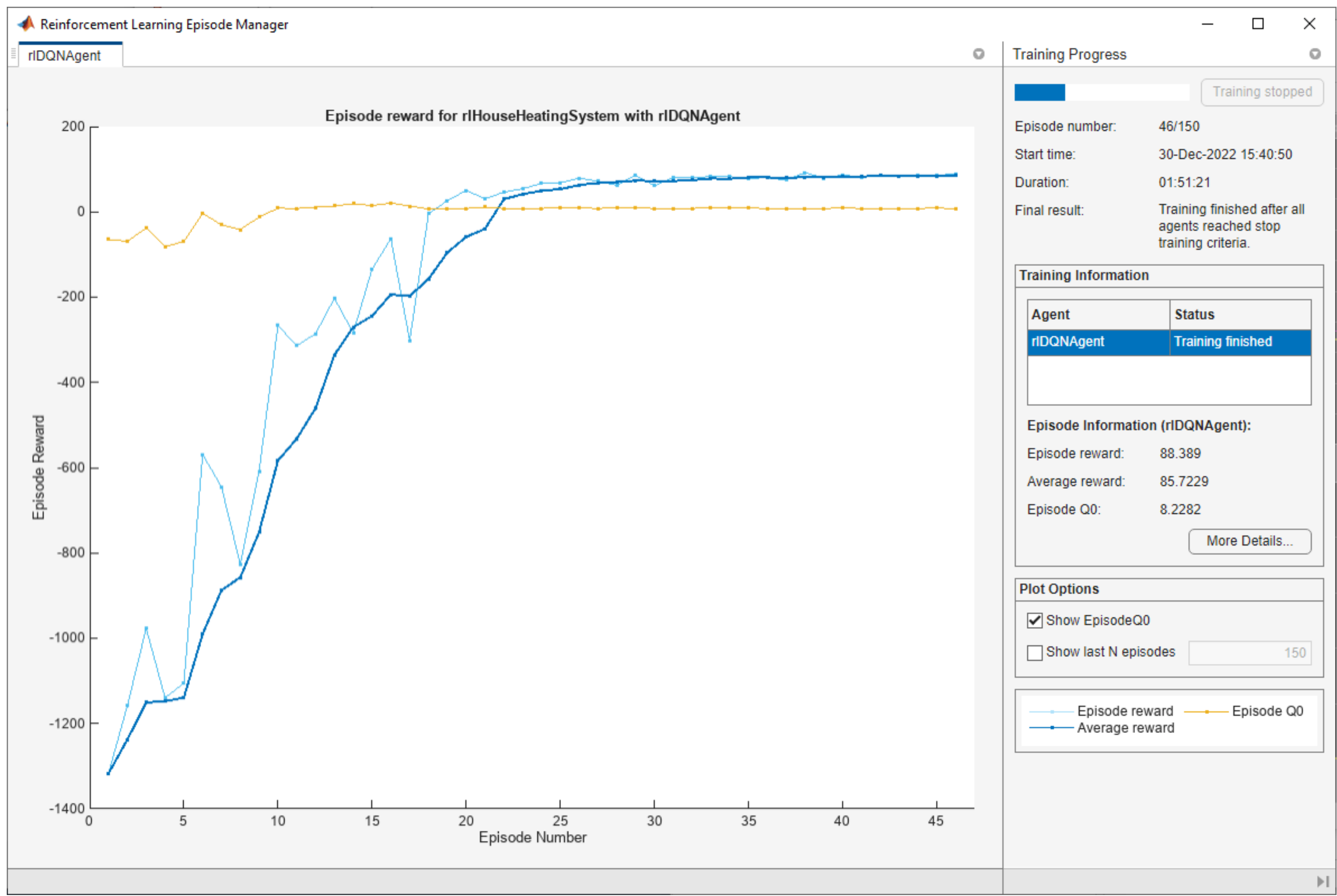
- Run training for at most 150 episodes, with each episode lasting 1000 time steps.
- Set the `Plots` option to `"training-progress"`, which displays training progress in the Reinforcement Learning Episode Manager.
- Set the `Verbose` option to `false` to disable the command line display
- Stop training when the agent receives an average cumulative reward greater than 85 over 5 consecutive episodes.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes = 150, ...
    MaxStepsPerEpisode = maxStepsPerEpisode, ...
    ScoreAveragingWindowLength = 5,...
    Verbose = false, ...
    Plots = "training-progress",...
    StopTrainingCriteria = "AverageReward",...
    StopTrainingValue = 85);
```

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

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




## Simulate DQN Agent

To validate the performance of the trained agent, simulate it within the house heating system. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

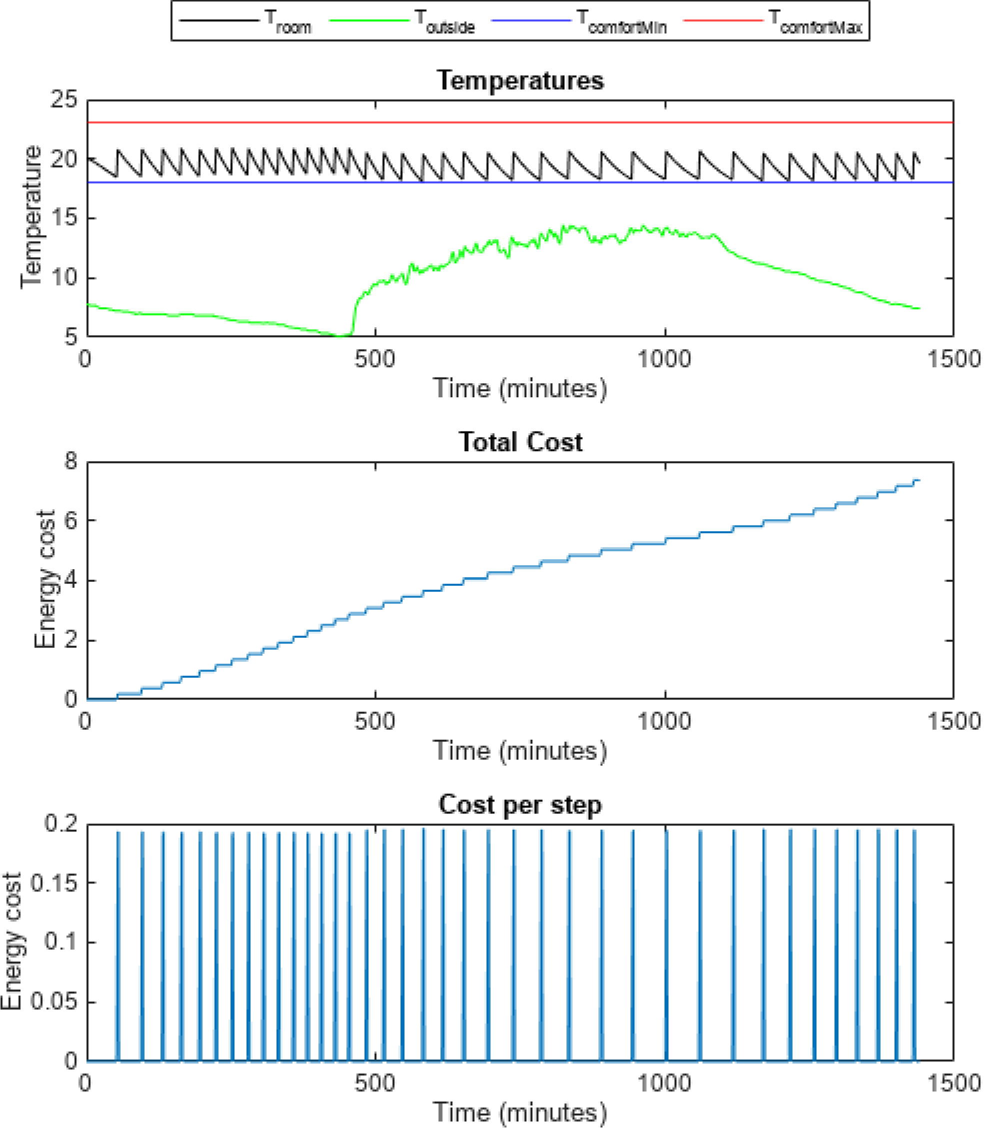
First, evaluate the agent's performance using the temperature data from March 21st, 2022. The agent did not use this temperature data for training.

```
maxSteps= 720;
validationTemperature = temperatureMarch21;
env.ResetFcn = @(in) hRLHeatingSystemValidateResetFcn(in);
```

```
simOptions = rlSimulationOptions(MaxSteps = maxSteps);
experience1 = sim(env,agent,simOptions);
```

Use the `localPlotResults` function, provided at the end of the example, to analyze the performance.

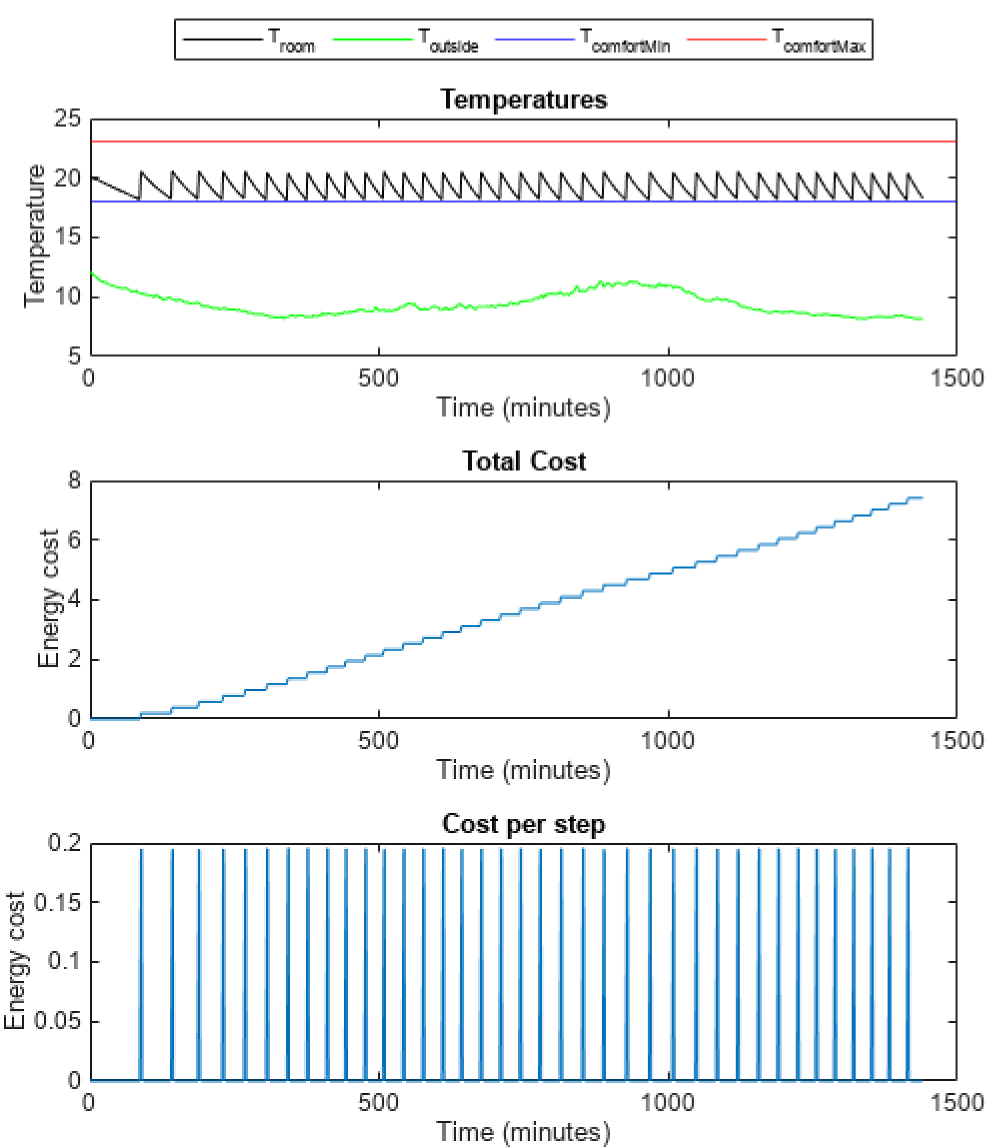
```
localPlotResults(experience1, maxSteps, ...
    comfortMax, comfortMin, sampleTime,1)
```




```
Comfort Temperature violation: 0/1440 minutes, cost: 7.368132 dollars
```

Next, evaluate the agent's performance using the temperature data from April 15th, 2022. The agent did not use this temperature data during training either.

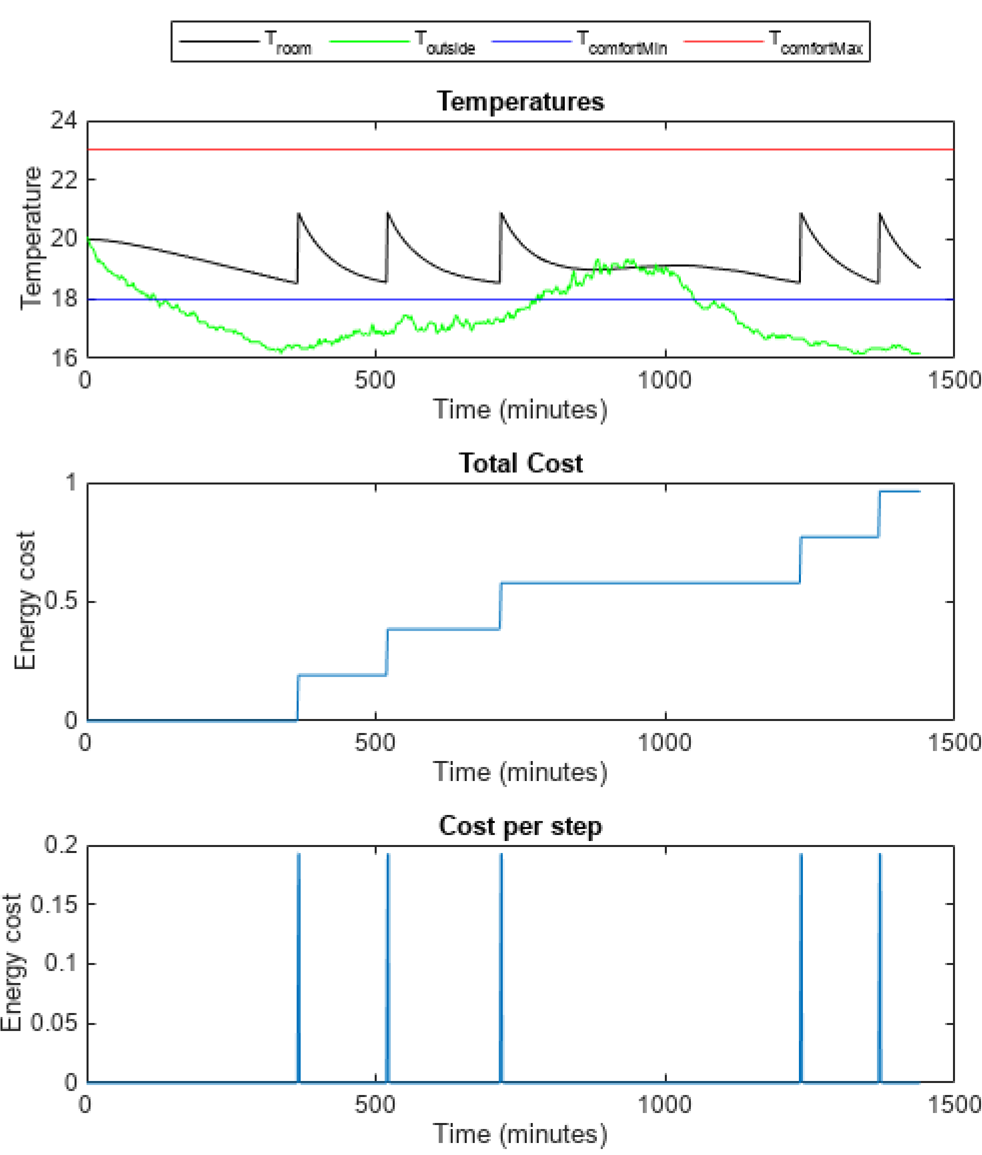
```
% Validate agent using the data from April 15
validationTemperature = temperatureApril15;
experience2 = sim(env,agent,simOptions);
localPlotResults( ...
    experience2, ...
    maxSteps, ...
    comfortMax, ...
    comfortMin, ...
    sampleTime,2)
```




```
Comfort Temperature violation: 0/1440 minutes, cost: 7.413613 dollars
```

Evaluate the agent's performance when the temperature is mild. Add eight degrees to the temperature from April 15th to create data for mild temperatures.

```
% Validate agent using the data from April 15 + 8 degrees
validationTemperature = temperatureApril15;
validationTemperature(:,2) = validationTemperature(:,2) + 8;
experience3 = sim(env,agent,simOptions);
localPlotResults(experience3, ...
    maxSteps, ...
    comfortMax, ...
    comfortMin, ...
    sampleTime, ...
    3)
```




```
Comfort Temperature violation: 0/1440 minutes, cost: 0.963522 dollars
```

### Local Function

```
function localPlotResults(experience, maxSteps, comfortMax, comfortMin, sampleTime, figNum)
    % localPlotResults plots results of validation

    % Compute comfort temperature violation
    minutesViolateComfort = ...
    sum(experience.Observation.obs1.Data(1,:,1:maxSteps) < comfortMin) ...
    + sum(experience.Observation.obs1.Data(1,:,1:maxSteps) > comfortMax);

    % Cost of energy
    totalCosts = experience.SimulationInfo(1).househeat_output{1}.Values;
    totalCosts.Time = totalCosts.Time/60;
    totalCosts.TimeInfo.Units="minutes";
    totalCosts.Name = "Total Energy Cost";
    finalCost = experience.SimulationInfo(1).househeat_output{1}.Values.Data(end);

    % Cost of energy per step
    costPerStep = experience.SimulationInfo(1).househeat_output{2}.Values;
    costPerStep.Time = costPerStep.Time/60;
    costPerStep.TimeInfo.Units="minutes";
    costPerStep.Name = "Energy Cost per Step";
    minutes = (0:maxSteps)*sampleTime/60;

    % Plot results

    fig = figure(figNum);
    % Change the size of the figure
    fig.Position = fig.Position + [0, 0, 0, 200];

    % Temperatures
    layoutResult = tiledlayout(3,1);
    nexttile
    plot(minutes, ...
        reshape(experience.Observation.obs1.Data(1,:,:), ...
        [1,length(experience.Observation.obs1.Data)]),"k")
    hold on
    plot(minutes, ...
        reshape(experience.Observation.obs1.Data(2,:,:), ...
        [1,length(experience.Observation.obs1.Data)]),"g")
    yline(comfortMin,'b')
    yline(comfortMax,'r')
    lgd = legend("T_{room}", "T_{outside}","T_{comfortMin}", ...
        "T_{comfortMax}","location","northoutside");
    lgd.NumColumns = 4;
    title("Temperatures")
    ylabel("Temperature")
    xlabel("Time (minutes)")
    hold off

    % Total cost
    nexttile
    plot(totalCosts)
    title("Total Cost")
    ylabel("Energy cost")

    % Cost per step
    nexttile
```

```
    plot(costPerStep)
    title("Cost per step")
    ylabel("Energy cost")
    fprintf("Comfort Temperature violation:" + ...
        " %d/1440 minutes, cost: %f dollars\n", ...
        minutesViolateComfort, finalCost);
end
```

#### Reference

[1] Du, Yan, Fangxing Li, Kuldeep Kurte, Jeffrey Munk, and Helia Zandi. “Demonstration of Intelligent HVAC Load Management With Deep Reinforcement Learning: Real-World Experience of Machine Learning in Demand Control.” *IEEE Power and Energy Magazine* 20, no. 3 (May 2022): 42–53. https://doi.org/10.1109/MPE.2022.3150825.

## See Also

**Functions**
`train` | `sim` | `rlSimulinkEnv`

**Objects**
`rlDQNAgent` | `rlDQNAgentOptions` | `rlTrainingOptions`

## Related Examples

- “House Heating System” (Simscape)
- “Train DQN Agent to Balance Cart-Pole System” on page 5-50
- “Water Distribution System Scheduling Using Reinforcement Learning” on page 5-353
- “Compare Temperature Data from Three Different Days” (ThingSpeak)

## More About

- “Deep Q-Network (DQN) Agents” on page 3-23
- “Train Reinforcement Learning Agents” on page 5-3
- “Long Short-Term Memory Neural Networks”

# Generate Policy Block for Deployment

This example shows how to generate a Policy block ready for deployment from an agent object. You generate the policy from the agent in “Train TD3 Agent for PMSM Control” on page 5-341, then simulate it to validate its performance. The policy is simulated to validate performance. If Embedded Coder® is installed, a software-in-the-loop (SIL) simulation is run to validate the generated code of the policy.

In general, the workflow for the deployment of a reinforcement learning policy via a Simulink® model is:

1. Train the agent (see “Train TD3 Agent for PMSM Control” on page 5-341).
2. Generate a Policy block from the trained agent.
3. Replace the RL Agent block with the Policy block.
4. Configure the model for code generation.
5. Simulate the policy and verify policy performance.
6. Generate code for the policy, simulate the generated code, and verify policy performance.
7. Deploy to hardware for testing.

In this example, you do steps 2 through 6.

Load the motor parameters along with the trained TD3 agent.

```
sim_data;
```

```
### The Lq is observed to be lower than Ld. ###
### Using the lower of these two for the Ld (internal variable)  ###
### and higher of these two for the Lq (internal variable) for computations. ###
### The Lq is observed to be lower than Ld. ###
### Using the lower of these two for the Ld (internal variable)  ###
### and higher of these two for the Lq (internal variable) for computations. ###
             model: 'Maxon-645106'
                sn: '2295588'
                 p: 7
                Rs: 0.2930
                Ld: 8.7678e-05
                Lq: 7.7724e-05
                Ke: 5.7835
                 J: 8.3500e-05
                 B: 7.0095e-05
           I_rated: 7.2600
          QEPSlits: 4096
            N_base: 3476
             N_max: 4300
            FluxPM: 0.0046
           T_rated: 0.3471
    PositionOffset: 0.1650

               model: 'BoostXL-DRV8305'
                  sn: 'INV_XXXX'
                V_dc: 24
              I_trip: 10
              Rds_on: 0.0020
```

```
             Rshunt: 0.0070
      CtSensAOffset: 2295
      CtSensBOffset: 2286
      CtSensCOffset: 2295
            ADCGain: 1
        EnableLogic: 1
       invertingAmp: 1
         ISenseVref: 3.3000
    ISenseVoltPerAmp: 0.0700
          ISenseMax: 21.4286
            R_board: 0.0043
     CtSensOffsetMax: 2500
     CtSensOffsetMin: 1500

                   model: 'LAUNCHXL-F28379D'
                      sn: '123456'
           CPU_frequency: 200000000
           PWM_frequency: 5000
      PWM_Counter_Period: 20000
                ADC_Vref: 3
            ADC_MaxCount: 4095
           SCI_baud_rate: 12000000

    V_base: 13.8564
    I_base: 21.4286
    N_base: 3476
    T_base: 1.0249
    P_base: 445.3845
```

```
load("rlPMSMAgent.mat","agent");
```

### Generate Policy Block

Open the Simulink model used for training the TD3 agent.

```
mdl_rl = "mcb_pmsm_foc_sim_RL";
open_system(mdl_rl);
```

Open the subsystem containing the **RL Agent** block.

```
agentblk = mdl_rl + ...
    "/Current Control/Control_System" + ...
    "/Closed Loop Control/Reinforcement Learning/RL Agent";
open_system(get_param(agentblk,"Parent"));
```

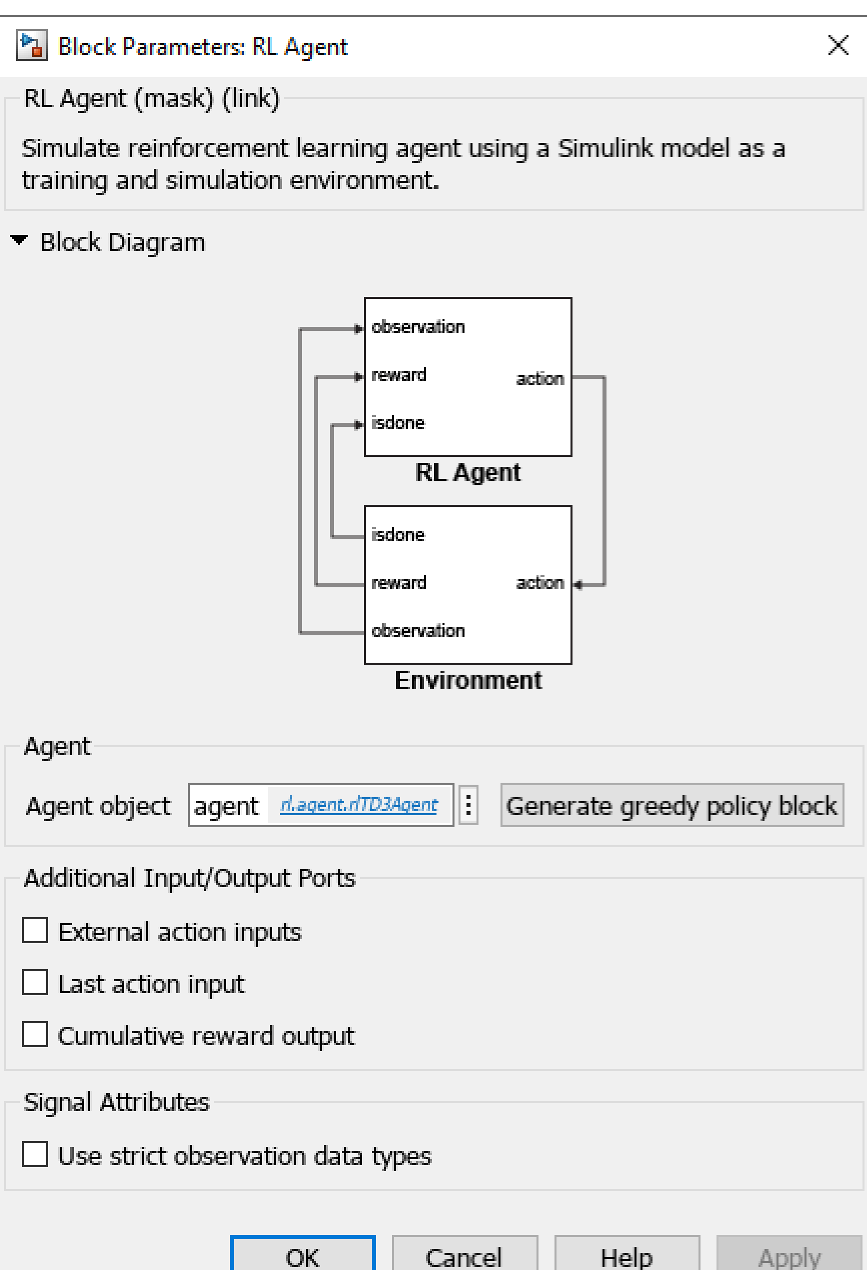
To create a deployable model, you replace the **RL Agent** block with a **Policy** block.

Set the agent's `UseExplorationPolicy` property to false so the generated policy takes the greedy action at each time step. Generate the policy block using `generatePolicyBlock` and specify the name of the MAT-file containing the policy data for the block.

```
% To ensure that the generated policy is greedy,
% set UseExplorationPolicy to false
agent.UseExplorationPolicy = false;

% Specify the MAT-file name for the policy data
fname = "PMSMPolicyBlockData.mat";
```

```
% Delete the file if it already exists
if isfile(fname)
    delete(fname);
end

% Generate the block and the policy data
generatePolicyBlock(agent,MATFileName=fname)
```

Alternatively, you can generate the policy block can be generated by clicking **Generate greedy policy block** from the block mask. Use `open_system(agentblk)` to open the **RL Agent** block mask, or simply double-click the block.



### Simulate the Policy

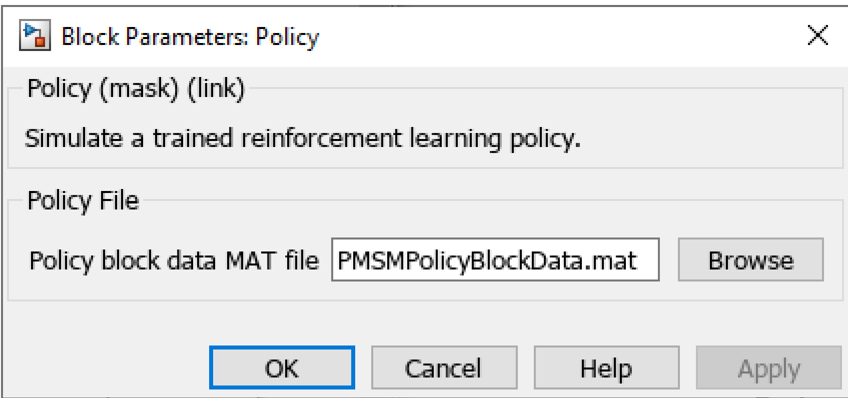
For this example, the **Policy** block has already replaced the **RL Agent** block inside of the `pmsm_current_control` model. This model has been configured for code generation, and the **Policy** block loads the trained policy from `PMSMPolicyBlockData.mat`.

```
mdl_current_ctrl = "pmsm_current_control";
open_system(mdl_current_ctrl);
```

Obtain the path of the policy block in the Simulink model.

```
policyblk = mdl_current_ctrl + ...
    "/Current Control/Control_System" + ...
    "/Closed Loop Control/Reinforcement Learning/Policy";
```

Use `open_system(policyblk)` to open the **Policy** block mask, or simply double-click the block.




The model **mcb_pmsm_foc_sim_policy** references **pmsm_current_control** using a model reference block. Simulate the top-level model and plot the responses for the inner and outer control loops.

Setup the **Simulation Data Inspector** (SDI).

```
Simulink.sdi.clear;
Simulink.sdi.setSubPlotLayout(3,1);
```

Open the model and get the path to the `Current Control` block.

```
mdl_policy = "mcb_pmsm_foc_sim_policy";
open_system(mdl_policy);
current_ctrl_blk = mdl_policy + "/Current Control";
```

To temporarily change the model and easily run multiple simulations with such changes, `Simulink.SimulationInput` (Simulink) object.

```
in = Simulink.SimulationInput(mdl_policy);
```

Simulate the model with the current controller, run the simulation in normal mode

```
in = setBlockParameter(in,current_ctrl_blk, ...
    "SimulationMode","Normal");
out_sim = sim(in);
```

Get results from the latest SDI run.

```
runSim = Simulink.sdi.Run.getLatest;
```

Extract the outer control loop signals.

```
speedSim    = getSignalsByName(runSim,"Speed_fb" );
speedRefSim = getSignalsByName(runSim,"Speed_Ref");
```

Plot both signals.

```
plotOnSubPlot(speedSim   ,1,1,true);
plotOnSubPlot(speedRefSim,1,1,true);
```

Extract the inner control loop signals.

```
idSim    = getSignalsByName(runSim,"id" );
iqSim    = getSignalsByName(runSim,"iq" );
idRefSim = getSignalsByName(runSim,"id_ref");
iqRefSim = getSignalsByName(runSim,"iq_ref");
```

Plot the extracted signals.

```
plotOnSubPlot(idSim   ,2,1,true);
plotOnSubPlot(idRefSim,2,1,true);
plotOnSubPlot(iqSim   ,3,1,true);
plotOnSubPlot(iqRefSim,3,1,true);
```

Open the **Simulation Data Inspector**.

```
Simulink.sdi.view;
```

### Validate Generated Code for Policy

If Embedded Coder is installed, the current controller model reference can be run in SIL mode. Running the current controller in SIL mode generates code for the current controller model, including the policy block.

The Simulink model is configured only on Windows, so display a message and stop execution when attempting to run on a different operating system.

```
if ~ispc
    disp("The model ""pmsm_current_control"" is configured " + ...
         "for SIL simulations only on Windows system.")
```

```
    return
end
```

Simulate the model with the current controller.

Enable SIL mode.

```
in = setBlockParameter(in,current_ctrl_blk, ...
    "SimulationMode","Software-in-the-loop");
```

Simulate the model. Use `evalc` to capture the text output from code generation, for possible later inspection.

```
txt_out = evalc("out_sil = sim(in)");
```

Get results from the latest SDI run.

```
runSIL = Simulink.sdi.Run.getLatest;
```

Extract the speed response when run in SIL mode.

```
speedSIL = getSignalsByName(runSIL,"Speed_fb");
```

Compare the SIL response to the simulated response. The SIL response should be close to the response simulated in normal mode.

```
speedSim.AbsTol = 1e-3;
cr = Simulink.sdi.compareSignals(speedSim.ID,speedSIL.ID);
```

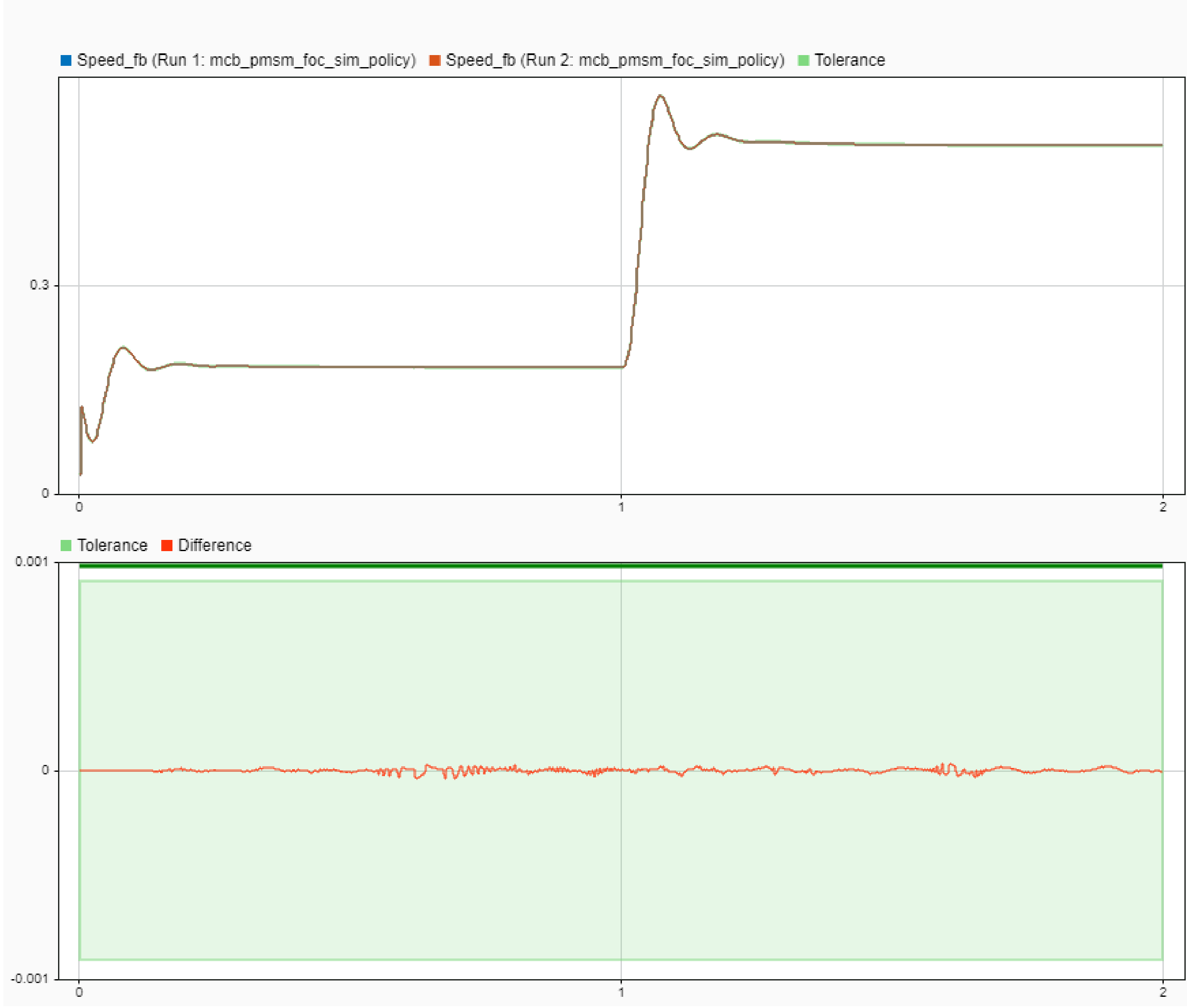
Display the largest signal difference.

```
cr.MaxDifference
```

```
ans = 5.0485e-05
```

Open the **Simulation Data Inspector**.

```
Simulink.sdi.view;
```




Once you are satisfied with the performance of the policy in simulation, you can use an appropriate target/hardware support package to deploy the policy to hardware.

## See Also

**Functions**
`generatePolicyBlock` | `generatePolicyFunction` | `train`

**Objects**
`dlnetwork`

**Blocks**
Policy

## Related Examples


- “Train Reinforcement Learning Agents” on page 5-3


## More About


- “Create Policies and Value Functions” on page 4-2
- “Reinforcement Learning Agents” on page 3-2
- “Deploy Trained Reinforcement Learning Policies” on page 6-2

# Train Reinforcement Learning Policy Using Custom Training Loop

This example shows how to define a custom training loop for a reinforcement learning policy. You can use this workflow to train reinforcement learning policies with your own custom training algorithms rather than using one of the built-in agents from the Reinforcement Learning Toolbox™ software.

Using this workflow, you can train policies that use any of the following policy and value function approximators.

- `rlValueFunction` - State value function approximator
- `rlQValueFunction` - State-action value function approximator with scalar output
- `rlVectorQValueFunction` - State-action function approximator with vector output
- `rlContinuousDeterministicActor` - Continuous deterministic actor
- `rlDiscreteCategoricalActor` - Discrete stochastic actor
- `rlContinuousGaussianActor` - Continuous Gaussian actor (stochastic)

In this example, a discrete actor policy with a discrete action space is trained using the REINFORCE algorithm (with no baseline). For more information on the REINFORCE algorithm, see “Policy Gradient (PG) Agents” on page 3-27.

Fix the random generator seed for reproducibility.

```
rng(0)
```

For more information on the functions you can use for custom training, see Functions for Custom Training on page 5-438.

#### Environment

For this example, a reinforcement learning policy is trained in a discrete cart-pole environment. The objective in this environment is to balance the pole by applying forces (actions) on the cart. Create the environment using the `rlPredefinedEnv` function.

```
env = rlPredefinedEnv("CartPole-Discrete");
```

Extract the observation and action specifications from the environment.

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

Obtain the dimension of the observation space (`numObs`) and the number of possible actions (`numAct`).

```
numObs = obsInfo.Dimension(1);
numAct = actInfo.Dimension(1);
```

For more information on this environment, see “Load Predefined Control System Environments” on page 2-23.

#### Policy

The reinforcement learning policy in this example is a discrete-action stochastic policy. It is modeled by a deep neural network that contains `fullyConnectedLayer`, `reluLayer`, and `softmaxLayer`

layers. This network outputs probabilities for each discrete action given the current observations. The `softmaxLayer` ensures that the actor outputs probability values in the range [0 1] and that all probabilities sum to 1.

Create the deep neural network for the actor.

```
actorNetwork = [
    featureInputLayer(numObs)
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(2)
    softmaxLayer
    ];
```

Convert to `dlnetwork`.

```
actorNetwork = dlnetwork(actorNetwork);
```

Create the actor using an `rlDiscreteCategoricalActor` object.

```
actor = rlDiscreteCategoricalActor(actorNetwork,obsInfo,actInfo);
```

Accelerate the gradient computation of the actor.

```
actor = accelerate(actor,true);
```

Evaluate the policy with a random observation as input.

```
policyEvalOutCell = evaluate(actor,{rand(obsInfo.Dimension)});
policyEvalOut = policyEvalOutCell{1}
```

```
policyEvalOut = 2x1 single column vector

    0.4682
    0.5318
```

Create the optimizer using `rlOptimizer` and `rlOptimizerOptions` function.

```
actorOpts = rlOptimizerOptions(LearnRate=1e-2);
actorOptimizer = rlOptimizer(actorOpts);
```

### Training Setup

Configure the training to use the following options:

- Set up the training to last at most 5000 episodes, with each episode lasting at most 250 steps.
- To calculate the discounted reward, choose a discount factor of 0.995.
- Terminate the training after the maximum number of episodes is reached or when the average reward across 100 episodes reaches the value of 220.

```
numEpisodes = 5000;
maxStepsPerEpisode = 250;
discountFactor = 0.995;
avgWindowSize = 100;
trainingTerminationValue = 220;
```

Create a vector to store the cumulative reward for each training episode.

```
episodeCumulativeRewardVector = [];
```

Create a figure for training visualization using the `hBuildFigure` on page 5-440 helper function.

```
[trainingPlot,lineReward,lineAveReward] = hBuildFigure;
```

### Custom Training Loop

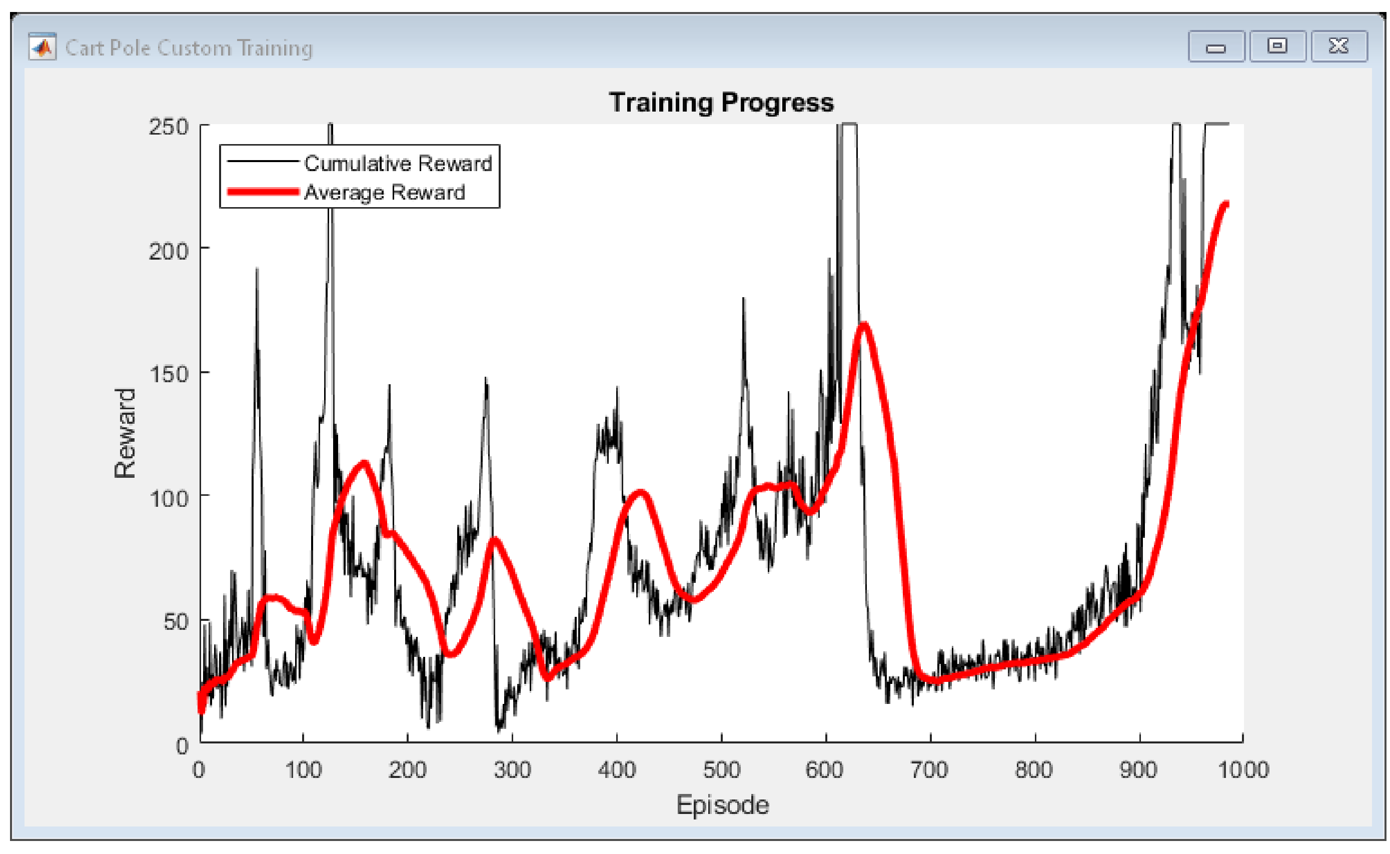
The algorithm for the custom training loop is as follows. For each episode:

1. Reset the environment.
2. Create buffers for storing experience information: observations, actions, and rewards.
3. Generate experiences until a terminal condition occurs. To do so, evaluate the policy to get actions, apply those actions to the environment, and obtain the resulting observations and rewards. Store the actions, observations, and rewards in buffers.
4. Collect the training data as a batch of experiences.
5. Compute the episode Monte Carlo return, which is the discounted future reward.
6. Compute the gradient of the loss function with respect to the policy parameters.
7. Update the policy using the computed gradients.
8. Update the training visualization.
9. Terminate training if the policy is sufficiently trained.

```
% Enable the training visualization plot.
set(trainingPlot,Visible="on");

% Train the policy for the maximum number of episodes
% or until the average reward indicates that the policy
% is sufficiently trained.
for episodeCt = 1:numEpisodes

    % 1. Reset the environment at the start of the episode

    obs = reset(env);
    episodeReward = zeros(maxStepsPerEpisode,1);

    % 2. Create buffers to store experiences.
    % The dimensions for each buffer must be as follows.
    %
    % For the observation buffer:
    %  numberOfObservations x ...
    %  numberOfObservationChannels x ...
    %  batchSize
    %
    % For action buffer:
    %     numberOfActions x ...
    %     numberOfActionChannels x ...
    %     batchSize
    %
    % For reward buffer:
    %     1 x batchSize
    %
    observationBuffer = zeros(numObs,1,maxStepsPerEpisode);
```

```
actionBuffer = zeros(numAct,1,maxStepsPerEpisode);
rewardBuffer = zeros(1,maxStepsPerEpisode);

% 3. Generate experiences
%    for the maximum number of steps per episode
%    or until a terminal condition is reached.
for stepCt = 1:maxStepsPerEpisode

    % Compute an action using the policy
    % based on the current observation.
    action = getAction(actor,{obs});

    % Apply the action to the environment
    % and obtain the resulting observation and reward.
    [nextObs,reward,isdone] = step(env,action{1});

    % Store the action, observation,
    % and reward experiences in their buffers.
    observationBuffer(:,:,stepCt) = obs;
    actionBuffer(:,:,stepCt) = action{1};
    rewardBuffer(:,stepCt) = reward;

    episodeReward(stepCt) = reward;
    obs = nextObs;

    % Stop if a terminal condition is reached.
    if isdone
        break;
    end

end

% 4. Create training data.
% Training is performed using batch data.
% The batch size cannot exceed the length of the episode.
batchSize = min(stepCt,maxStepsPerEpisode);
observationBatch = observationBuffer(:,:,1:batchSize);
actionBatch = actionBuffer(:,:,1:batchSize);
rewardBatch = rewardBuffer(:,1:batchSize);

% Compute the discounted future reward.
discountedReturn = zeros(1,batchSize);
for t = 1:batchSize
    G = 0;
    for k = t:batchSize
        G = G + discountFactor ^ (k-t) * rewardBatch(k);
    end
    discountedReturn(t) = G;
end

% 5. Organize data to pass to the loss function.
lossData.batchSize = batchSize;
lossData.actInfo = actInfo;
lossData.actionBatch = actionBatch;
lossData.discountedReturn = discountedReturn;

% 6. Compute the gradient of the loss
%    with respect to the policy parameters.
```

```
        actorGradient = gradient(actor,@actorLossFunction,...
            {observationBatch},lossData);

        % 7. Update the actor network using the computed gradients.
        % for more information, at the command line, type:
        % help rl.optimizer.AbstractOptimizer/update
        [actor,actorOptimizer] = update( ...
            actorOptimizer, ...
            actor, ...
            actorGradient);

        % 8. Update the training visualization.
        episodeCumulativeReward = sum(episodeReward);
        episodeCumulativeRewardVector = cat(2,...
            episodeCumulativeRewardVector,episodeCumulativeReward);
        movingAvgReward = movmean(episodeCumulativeRewardVector,...
            avgWindowSize,2);
        addpoints(lineReward,episodeCt,episodeCumulativeReward);
        addpoints(lineAveReward,episodeCt,movingAvgReward(end));
        drawnow;

        % 9. Terminate training if the network is sufficiently trained.
        if max(movingAvgReward) > trainingTerminationValue
            break
        end

    end
```



### Simulation

After training, simulate the trained policy.

Before simulation, reset the environment.

```
obs = reset(env);
```

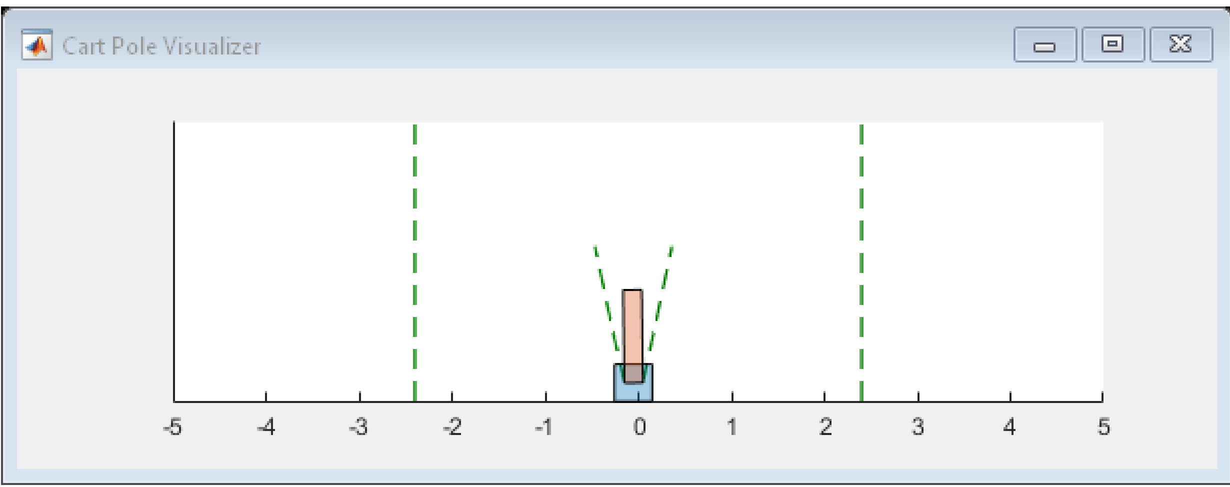
Enable the environment visualization, which is updated each time the environment step function is called.

```
plot(env)
```

For each simulation step, perform the following actions.

1. Get the action by sampling from the policy using the `getAction` function.
2. Step the environment using the obtained action value.
3. Terminate if a terminal condition is reached.

```
for stepCt = 1:maxStepsPerEpisode

    % Select action according to trained policy
    action = getAction(actor,{obs});

    % Step the environment
    [nextObs,reward,isdone] = step(env,action{1});

    % Check for terminal condition
    if isdone
        break
    end

    obs = nextObs;

end
```




### Functions for Custom Training

To obtain actions and value functions for given observations from Reinforcement Learning Toolbox policy and value function approximators, you can use the following functions.

- `getValue` — Obtain the estimated state value or state-action value function.
- `getAction` — Obtain the action from an actor based on the current observation.
- `getMaxQValue` — Obtain the estimated maximum state-action value function for a discrete Q-value approximator.

If your policy or value function approximator is a recurrent neural network, that is, a neural network with at least one layer that has hidden state information, the preceding functions can return the current network state. You can use the following function syntaxes to get and set the state of your approximator.

- `state = getState(critic)` — Obtain the state of approximator `critic`.
- `newCritic = setState(oldCritic,state)` — Set the state of approximator `newCritic`, and return the result in `oldCritic`.
- `newCritic = resetState(oldCritic)` — Reset all state values of `oldCritic` to zero and return the result in `newCritic`.

You can get and set the learnable parameters of your approximator using the `getLearnableParameters` and `setLearnableParameters` function, respectively.

In addition to these functions, you can use the `gradient`, `optimize`, and `syncParameters` functions to set parameters and compute gradients for your policy and value function approximators.

#### gradient

The `gradient` function computes the gradients of the approximator loss function. You can compute several different gradients. For example, to compute the gradient of the sum of the approximator outputs with respect to its inputs, use the following syntax.

```
grad = gradient(actor,"output-input",inputData)
```

Here:

- `actor` is a policy or value function approximator object.
- `inputData` contains values for the input channels to the approximator (e.g. an observation).
- `grad` contains the computed gradients.

For more information, see `gradient`.

#### syncParameters

The `syncParameters` function updates the learnable parameters of one policy or value function approximator based on those of another approximator. This function is useful for updating a target actor or critic approximator, as is done for DDPG agents. To synchronize parameters values between two approximators, use the following syntax.

```
newTargetApproximator = syncParameters(
   oldTargetApproximator, ...
   sourceApproximator, ...
   smoothFactor)
```

Here:

- `oldTargetApproximator` is a policy or value function approximator object with parameters $\theta_{\text{old}}$.

- `sourceApproximator` is a policy or value function approximator object with the same structure as `oldTargetRep`, but with parameters $\theta_{\text{source}}$.
- `smoothFactor` is a smoothing factor ($\tau$) for the update.
- `newTargetApproximator` has the same structure as `oldRep`, but its parameters are $\theta_{\text{new}} = \tau\theta_{\text{source}} + (1 - \tau)\theta_{\text{old}}$.

For more information, at the MATLAB command line, type `help rl.function.AbstractFunction.syncParameters`

#### Loss Function

The loss function in the REINFORCE algorithm the product between the discounted reward and the logarithm of the probability distribution of the action (coming from the policy evaluation for a given observation), summed across all time steps. The discounted reward calculated in the custom training loop must be resized to be multiplied with the logarithm of the action probability distribution.

The function first input parameter must be a cell array like the one returned from the evaluation of a function approximator object. For more information, see the description of `outData` in `evaluate`. The second, optional, input argument contains additional data that might be needed by the loss calculation function. For more information, see `gradient`.

```
function loss = actorLossFunction(ActProbCell,lossFcnStruct)

    % Extract the matrix resulting from the policy evaluation
    ActProb = ActProbCell{1};

    % Create the action indication matrix.
    batchSize = lossFcnStruct.batchSize;
    Z = repmat(lossFcnStruct.actInfo.Elements',1,batchSize);
    actionIndicationMatrix = (lossFcnStruct.actionBatch(:,:)==Z);

    % Resize the discounted return to the size of ActProb.
    G = actionIndicationMatrix .* lossFcnStruct.discountedReturn;
    G = reshape(G,size(ActProb));

    % Round any action probability values less than eps to eps.
    ActProb(ActProb < eps) = eps;

    % Compute the loss.
    loss = -sum(G .* log(ActProb),"all");

end
```

#### Helper Function

The following helper function creates a figure for training visualization.

```
function [trainingPlt, lineRewd, lineAvgRwd] = hBuildFigure()
    plotRatio = 16/9;
    trainingPlt = figure(...
                Visible="off",...
                HandleVisibility="off", ...
                NumberTitle="off",...
                Name="Cart Pole Custom Training");
```

```
    trainingPlt.Position(3) = ...
         plotRatio * trainingPlt.Position(4);

    ax = gca(trainingPlt);

    lineRewd = animatedline(ax);
    lineAvgRwd = animatedline(ax,Color="r",LineWidth=3);
    xlabel(ax,"Episode");
    ylabel(ax,"Reward");
    legend(ax,"Cumulative Reward","Average Reward", ...
           Location="northwest")
    title(ax,"Training Progress");
end
```

## See Also

**Functions**
accelerate | gradient | evaluate | rlOptimizer | getLearnableParameters | setLearnableParameters

**Objects**
rlDiscreteCategoricalActor | rlOptimizerOptions

## Related Examples

- "Custom Training Loop with Simulink Action Noise" on page 5-442
- "Train Custom LQR Agent" on page 5-466
- "Create Agent for Custom Reinforcement Learning Algorithm" on page 5-456

## More About

- "Train Reinforcement Learning Agents" on page 5-3
- "Create Custom Reinforcement Learning Agents" on page 3-68
- "Load Predefined Control System Environments" on page 2-23

# Custom Training Loop with Simulink Action Noise

This example shows how to tune a controller for vehicle platooning applications using a custom reinforcement learning (RL) training loop. For this application, action noise is generated in the Simulink® model to promote exploration during training.

For an example on tuning a PID-based vehicle platooning system, see “Design Controller for Vehicle Platooning” (Simulink Control Design).

Platooning has the following control objectives [1].

- Individual vehicle stability — Spacing error for each following vehicle converges to zero if the preceding vehicle is traveling at constant speed.
- String stability — Spacing errors do not amplify as they propagate towards the tail of the vehicle string.

### Platooning Environment Model

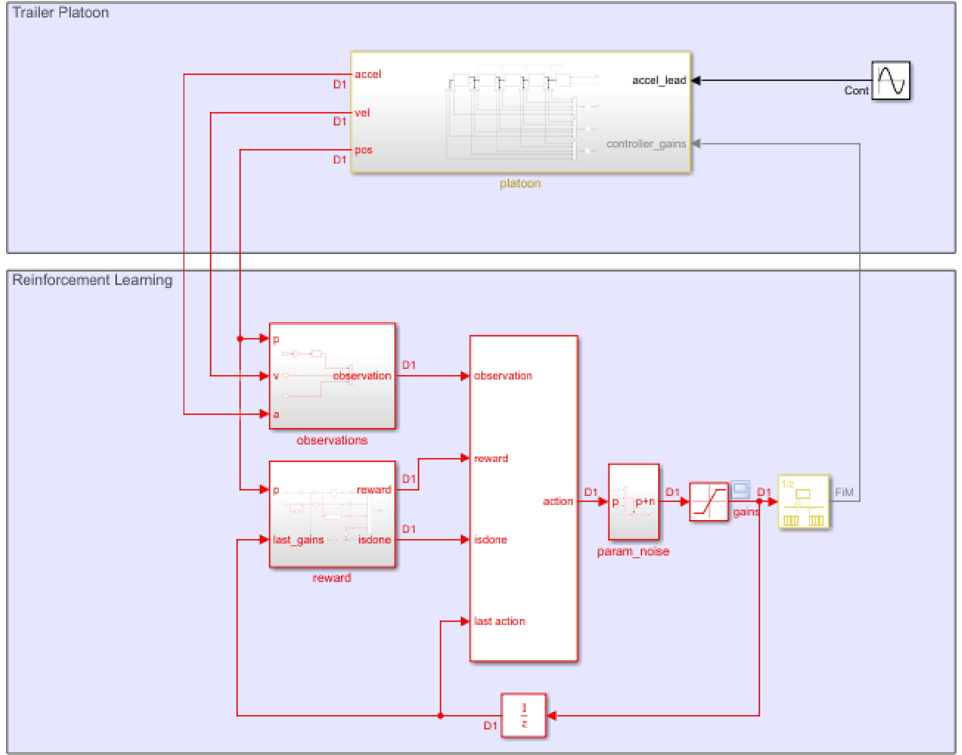
In this example, there are five vehicles in the platoon. Every vehicle is modeled as a truck-trailer system with the following parameters. All lengths are in meters.

```
L1 = 6;               % Truck length
L2 = 10;              % Trailer length
M1 = 1;               % Hitch length
L = L1 + L2 + M1 + 5; % Desired front-to-front vehicle spacing
```

The lead vehicle follows a given acceleration profile. Each trailing vehicle has a controller that controls its acceleration.

Open the Simulink® model.

```
mdl = "fiveVehiclePlatoonEnv";
open_system(mdl)
```




The model contains an RL Agent block with its **last action** input port enabled. This input port allows the specification of custom noise in the Simulink model for off-policy RL agents, such as deep deterministic policy gradient (DDPG) agents.

Specify the path to the RL Agent block.

```
agentBlk = mdl + "/RL Agent";
```

### Controller Structure

In this example, each trailing vehicle (ego vehicle) has the same continuous-time controller structure and parameterization.

$$a_{ego} = K_1 a_{front} - K_2\left(v_{ego} - v_{front}\right) - K_3\left(x_{ego} - x_{front} + L\right)$$

Here:

- $a_{ego}$, $v_{ego}$, and $x_{ego}$ are the respective acceleration, velocity, and position of the ego vehicle.
- $a_{front}$, $v_{front}$, and $x_{front}$ are the respective acceleration, velocity, and position of the vehicle directly in front of the ego vehicle.

Each vehicle has full access to its own velocity and position states but can only access the acceleration, velocity, and position of the vehicle directly in front using wireless communication.

The controller minimizes the velocity error $v_{front} - v_{ego}$ using the velocity gain $K_2$ and minimizes the spacing error $x_{ego} - x_{front} + L$ using the spacing gain $K_3$. The feedforward gain $K_1$ is used to improve tracking of the front vehicle.

The lead vehicle' acceleration is assumed to be a sine wave

$$a_{\text{lead}} = A\sin(\text{wt})$$

Here:

- $a_{\text{lead}}$ is the lead vehicle acceleration (m/s^2).
- $A$ is the amplitude of the sine wave (m/s).
- $w$ is the frequency of the sine wave (rad/s).
- $t$ is the simulation time (s).

### Reinforcement Learning Agent Design

The objective of the agent is to compute adaptive gains so that each vehicle can track the desired spacing with respect to the vehicle immediately in front. Therefore, the model is configured such that:

- The action signal consists of the gains $\boldsymbol{K} = [K_1\ K_2\ K_3]$ shared by all vehicles except the lead vehicle. Each gain has a lower bound of 0 and upper bounds of 1, 20, and 20, respectively. The agent calculates new gains once per second. To encourage exploration during training, the gains are perturbed by random noise with a zero-mean normal distribution: $K_{\text{noisy}} = K + N\left(0, \sigma^2\right)$ where the variance $\sigma^2 = [0.02, 0.1, 0.1]$.
- The observation signal consists of the vehicle spacing ($s_t = \text{diff}(\boldsymbol{x})$) minus the target spacing ($L$), the vehicle velocities ($\boldsymbol{v}$), and the vehicle accelerations ($\boldsymbol{a}$).
- The reward calculated at every time step $t$ is

$$r_t = \sum \frac{1}{1 + (\boldsymbol{s_t} - L)^2} - 0.2 \sum (\boldsymbol{K_{t-1}} - \boldsymbol{K_{t-2}})^2 - 1.0\,\text{maxOvershoot}(\boldsymbol{s_t}, L) - 10c_t$$

where:

- $\boldsymbol{s_t}$ is the vehicle spacing ($\text{diff}(\boldsymbol{x_t})$) at time step $t$.
- $\text{maxOvershoot}(\boldsymbol{s_t}, L)$ calculates the max overshoot of all vehicles given the actual vehicle spacing $s_t$ and the desired spacing $L$. In this case, overshoot is defined as when the vehicle spacing is less than the desired spacing $\boldsymbol{s_t} < L$.
- $c_t = \text{any}((\boldsymbol{s_t} + L_1 + L_2 + M_1 - L) \le 0))$ indicates if a vehicle collision occurs. The simulation will terminate if $c_t$ is true.

The first term in the reward function encourages the vehicle spacing to match $L$. The second term penalizes large changes in gain between time steps. The third term penalizes overshooting the target spacing (getting too close to the front vehicle). Finally, the fourth term penalizes collisions.

For this example, to accommodate the custom noise specified in the model, you implement a custom DDPG training loop.

### Define Model Parameters

Define the training and simulation parameters that remain fixed during training.

```
Ts = 1;                         % Sample time (seconds)
Tf = 100;                       % Simulation length (seconds)
```

```
AccelNoiseV = ones(1,5)*0.01; % Acceleration input noise variance
VelNoiseV = ones(1,5)*0.01;   % Velocity sensor noise variance
PosNoiseV = ones(1,5)*0.01;   % Position sensor noise variance
ParamLowerLimit = [0 0 0]';   % Lower limits for the controller gains
ParamUpperLimit = [1 20 20]'; % Upper limits for the controller gains
UseParamNoise = 1;            % Option to indicate noise injection
```

Define the parameters that change every training episode. The values for these parameters are updated in the environment reset function `resetFunction`.

```
LeadA = 2;                    % Lead vehicle acceleration amplitude
LeadF = 1;                    % Lead vehicle acceleration frequency
ParamNoiseV = [0.02 0.1 0.1]; % Variance for controller gains

% Random noise seeds
ParamNoiseSeed = 1:3;       % Controller gain noise seed
AccelNoiseSeed = 1:5 + 100; % Acceleration input noise seed
VelNoiseSeed = 1:5 + 200;   % Velocity sensor noise seed
PosNoiseSeed = 1:5 + 300;   % Position sensor noise seed

% Initial position and velocity of each vehicle
InitialPositions  = [200 150 100 50 0] + 50; % Positions
InitialVelocities = [10 10 10 10 10];        % Velocities
```

### Create Environment

Create an environment using `rlSimulinkEnv`.

To do so, first define the observation and action specifications for the environment.

```
obsInfo = rlNumericSpec([14 1]);
actInfo = rlNumericSpec([3  1],...
    LowerLimit=ParamLowerLimit,...
    UpperLimit=ParamUpperLimit);

obsInfo.Name = "measurements";
actInfo.Name = "control_gains";
```

Next, create the environment object.

```
env = rlSimulinkEnv(mdl,agentBlk,obsInfo,actInfo);
```

Set the environment reset function to the local function `resetFunction` included with this example. This function varies the training conditions for each episode.

```
env.ResetFcn = @resetFunction;
```

Noise in the model is specified using the Random Number (Simulink) block. Each block has its own random number generator and thus its own starting seed parameter. To ensure the noise stream varies across episodes, the seed variables are updated using `resetFunction` .

### Create Actor, Critic, and Policy

Create actor and critic function approximators for the agent using the local function `createNetworks` included with this example.

```
[critic,actor] = createNetworks(obsInfo,actInfo);
```

Create optimizer objects for updating the actor and critic. Use the same options for both optimizers.

```
optimizerOpt = rlOptimizerOptions(...
    LearnRate=1e-3, ...
    GradientThreshold=1, ...
    L2RegularizationFactor=1e-3);
criticOptimizer = rlOptimizer(optimizerOpt);
actorOptimizer  = rlOptimizer(optimizerOpt);
```

Create a deterministic policy for the actor approximator.

```
policy = rlDeterministicActorPolicy(actor);
```

Specify the policy sample time.

```
policy.SampleTime = Ts;
```

### Create Experience Buffer

Create an experience buffer for the agent with a maximum length of `1e6`.

```
replayMemory = rlReplayMemory(obsInfo,actInfo,1e6);
```

### Data Required for Learning

To update the actor and critic during training, the `runEpisode` function processes each experience as it is received from the environment. For this example, the processing function is the `processExperienceFcn` local function.

This function requires additional data to perform its processing. Create a structure to store this additional data.

```
processExpData.Critic = critic;
processExpData.TargetCritic = critic;
processExpData.Actor = actor;
processExpData.TargetActor = actor;
processExpData.ReplayMemory = replayMemory;
processExpData.CriticOptimizer = criticOptimizer;
processExpData.ActorOptimizer = actorOptimizer;

processExpData.MiniBatchSize = 128;
processExpData.DiscountFactor = 0.99;
processExpData.TargetSmoothFactor = 1e-3;
```

Each episode, the `processExperienceFcn` function updates the critics, actors, replay memory, and optimizers. The updated data is used as the input for the next episode.

### Training Loop

To train the agent, the custom training loop simulates the agent in the environment for a maximum of `maxEpisodes` episodes.

```
maxEpisodes = 1000;
```

Compute the maximum number of steps per episode using the simulation time and sample time.

```
maxSteps = ceil(Tf/Ts);
```

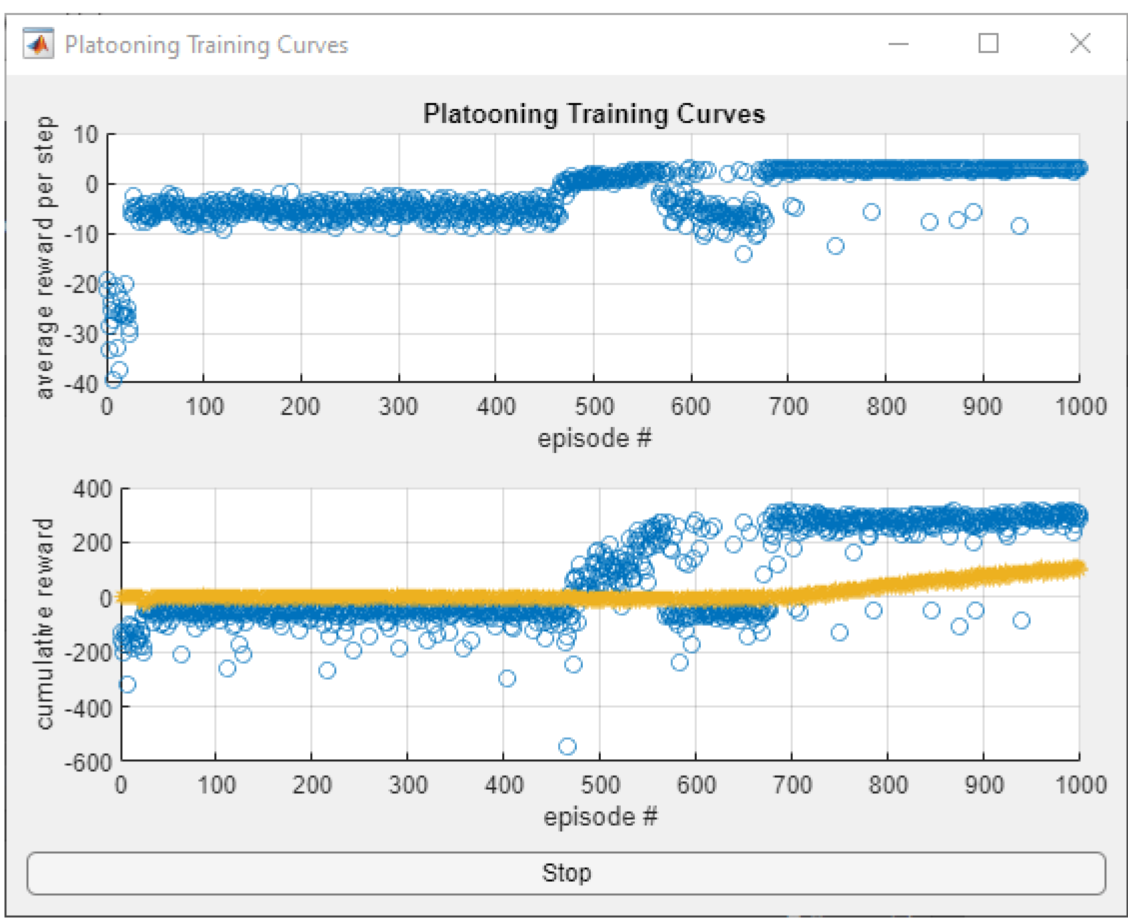
For this custom training loop:

- The `runEpisode` function simulates the agent in the environment for one episode.
- Experiences are processed as they are received from the environment using the `processExperienceFcn` function.
- Experiences are not logged by `runEpisode` since the experiences are processed as they are received.
- To speed up training, when calling `runEpisode`, the `CleanupPostSim` option is set to `false`. Doing so keeps the model compiled between episodes.
- The `PlatooningTrainingCurvePlotter` object is a helper object to plot training data while the training is running.
- You can stop the training using a **Stop** button in the training plot.
- After all the episodes are complete, the `cleanup` function cleans up the environment and terminates the model compilation.

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

```
doTraining = false;
if doTraining
    % Create plotting helper object.
    plotObj = PlatooningTrainingCurvePlotter();

    % Training loop
    for i = 1:maxEpisodes

        % Run the episode.
        out = runEpisode(...
            env,policy,...
            MaxSteps=maxSteps,...
            ProcessExperienceFcn=@processExperienceFcn,...
            ProcessExperienceData=processExpData,...
            LogExperiences=false,...
            CleanupPostSim=false);

        % Extract episode information
        % to update the training curves.
        episodeInfo = out.AgentData.EpisodeInfo;

        % Extract updated processExpData for the next episode.
        processExpData = out.AgentData.ProcessExperienceData;
        % Extract the updated policy for the next episode.
        policy = out.AgentData.Agent;

        % Extract critic and actor networks from processExpData.
        critic = processExpData.Critic;
        actor  = processExpData.Actor;

        % Extract the cumulative reward and calculate
        % average reward per step for this episode.
        cumulativeRwd = episodeInfo.CumulativeReward;
        avgRwdPerStep = cumulativeRwd/episodeInfo.StepsTaken;

        % Evaluate q0 from the initial episode observation.
        obs0 = episodeInfo.InitialObservation;
```

```
        q0 = evaluate(critic,[obs0,evaluate(actor,obs0)]);
        q0 = double(q0{1});

        % Update the plot.
        update(plotObj,i,avgRwdPerStep,cumulativeRwd,q0);

        % Exit training if button pushed.
        if plotObj.StopTraining
            break;
        end
    end

    % Clean up the environment.
    cleanup(env);

    % Save the policy.
    save("PlatooningDDPGPolicy.mat","policy");
else
    % Load the pretrained policy.
    load("PlatooningDDPGPolicy.mat");
end
```



### Validate Trained Policy

Validate the learned policy by running five simulations with random initial conditions specified by the reset function.

First, turn off parameter noise in the model.

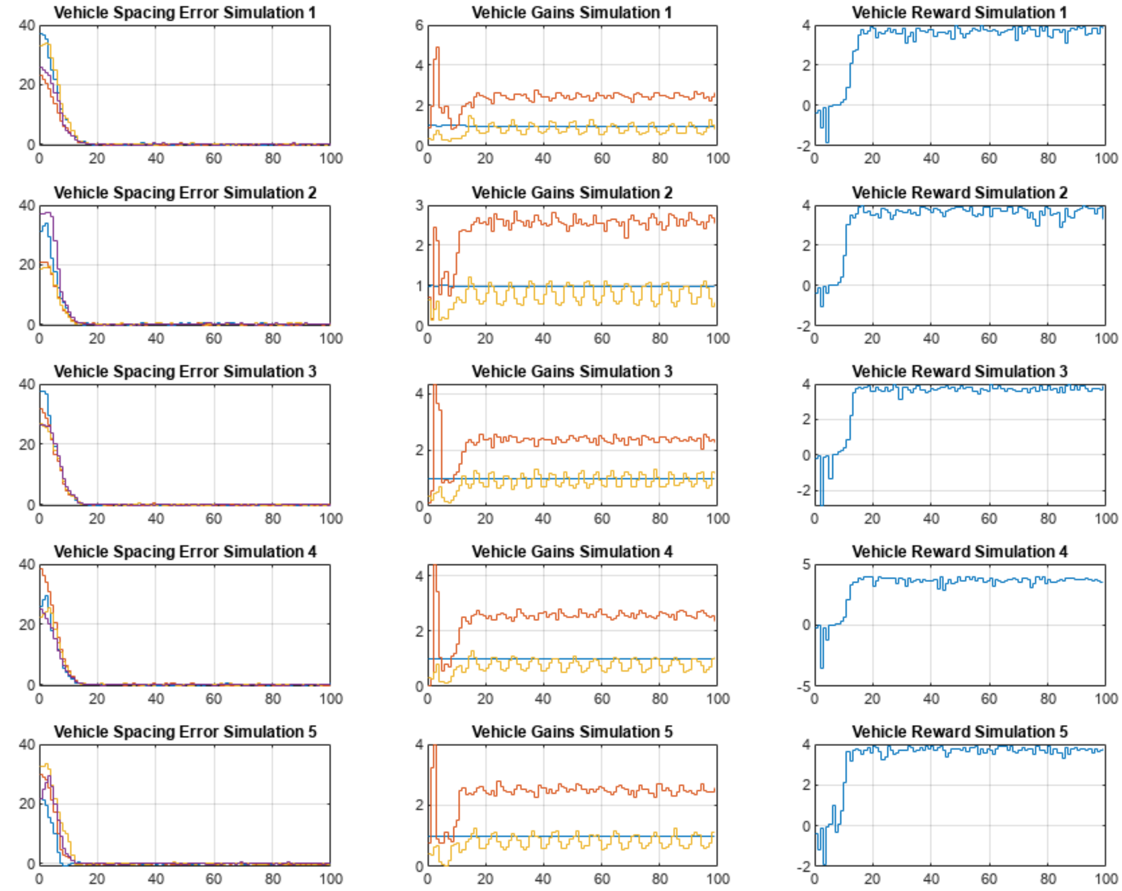
```
UseParamNoise = 0;
```

Simulate the model against the trained policy five times.

```
N = 5;
simOpts = rlSimulationOptions(...
    MaxSteps=maxSteps, ...
    NumSimulations=N);
experiences = sim(env,policy,simOpts);
```

Plot the vehicle spacing error, gains, and reward from the `experiences` output structure.

```
f = figure(Position=[100 100 1024 768]);
tiledlayout(f,N,3);
for i = 1:N

    % Get the spacing.
    tspacing = experiences(i).Observation.measurements.Time;
    spacing  = ...
    squeeze(experiences(i).Observation.measurements.Data(1:4,:,:));

    % Get the gains.
    tgains = experiences(i).Action.control_gains.Time;
    gains  = squeeze(experiences(i).Action.control_gains.Data);

    % Get the reward.
    trwd = experiences(i).Reward.Time;
    rwd  = experiences(i).Reward.Data;

    % Plot the spacing.
    nexttile
    stairs(tspacing,spacing');
    title(sprintf("Vehicle Spacing Error Simulation %u",i))
    grid on

    % plot the gains.
    nexttile
    stairs(tgains,gains');
    title(sprintf("Vehicle Gains Simulation %u",i))
    grid on

    % Plot the reward.
    nexttile
    stairs(trwd,rwd);
    title(sprintf("Vehicle Reward Simulation %u",i))
    grid on
end
```




From the plots, you can see that the trained policy generates adaptive gains that adequately track the desired spacing for all vehicles.

### Local Functions

The process experience function is called every time an experience is processed by the RL Agent block. Here, `processExperienceFcn` appends the experience to the replay memory, samples a mini-batch of experiences from the replay memory, and updates the critic, actor, and target networks.

```
function [policy,procExpData] = processExperienceFcn(...
    exp,episodeInfo,policy,procExpData)

% Append experience to replay memory buffer.
append(procExpData.ReplayMemory,exp);

% Sample a mini-batch of experiences from replay memory.
miniBatch = sample(procExpData.ReplayMemory, ...
    procExpData.MiniBatchSize,...
    DiscountFactor=procExpData.DiscountFactor);

if ~isempty(miniBatch)
```

```
    % Update network parameters using the mini-batch.
    [procExpData,actorParams] = learnFcn(procExpData,miniBatch);

    % Update the policy parameters using the actor parameters.
    policy = setLearnableParameters(policy,actorParams);
end
end
```

The `learnFcn` function updates the critic, actor, and target networks given a sampled mini-batch

```
function [processExpData,actorParams] = learnFcn( ...
    processExpData,miniBatch)

% Find the terminal experiences.
doneidx = (miniBatch.IsDone == 1);

% Compute target next actions against the next observations.
nextAction = evaluate( ...
    processExpData.TargetActor,miniBatch.NextObservation);

% compute qtarget = reward + gamma*Q(nextObservation,nextAction)
%                 = reward + gamma*expectedFutureReturn
targetq = miniBatch.Reward;

% Bootstrap the target at nonterminal experiences.
expectedFutureReturn = getValue(processExpData.TargetCritic, ...
    miniBatch.NextObservation,nextAction);
targetq(~doneidx) = targetq(~doneidx) + ...
    processExpData.DiscountFactor.*expectedFutureReturn(~doneidx);

% Compute critic gradient using deepCriticLoss function.
criticGradient = gradient(processExpData.Critic,@deepCriticLoss,...
    [miniBatch.Observation,miniBatch.Action],targetq);

% Update the critic parameters.
[processExpData.Critic,processExpData.CriticOptimizer] = update(...
    processExpData.CriticOptimizer,processExpData.Critic,...
    criticGradient);

% Compute the actor gradient using the deepActorGradient function.
% To accelerate the deepActorGradient function, the critic network
% is extracted outside the function and is passed in as a field
% to the actorGradData input struct.
actorGradData.CriticNet = getModel(processExpData.Critic);
actorGradData.MiniBatchSize = processExpData.MiniBatchSize;
actorGradient = customGradient(processExpData.Actor, ...
    @deepActorGradient,miniBatch.Observation,actorGradData);

% Update the actor parameters.
[processExpData.Actor,processExpData.ActorOptimizer] = update(...
    processExpData.ActorOptimizer,processExpData.Actor,...
    actorGradient);
actorParams = getLearnableParameters(processExpData.Actor);

% Update targets using the TargetSmoothFactor hyperparameter.
processExpData.TargetCritic = syncParameters( ...
    processExpData.TargetCritic,...
    processExpData.Critic, ...
```

```
    processExpData.TargetSmoothFactor);
processExpData.TargetActor  = syncParameters( ...
    processExpData.TargetActor,...
    processExpData.Actor, ...
    processExpData.TargetSmoothFactor);
end
```

The critic gradient is computed against the `deepCriticLoss` function.

```
function loss = deepCriticLoss(q,targetq)

% Extract value from cell
q = q{1};

% Loss is the half mean-square error of
% q = Q(observation,action) against  qtarget
loss = mse(q,reshape(targetq,size(q)));

end
```

The actor gradient is computed to maximize the expected value of an observation-action pair given the policy parameters. Here, the negative sign is used to maximize $Q$ with respect to $\theta$.

$$\frac{d}{d\theta}\left(-\frac{1}{N}\sum Q_\phi(\boldsymbol{s},\boldsymbol{a})\right) = \frac{d}{d\theta}\left(-\frac{1}{N}\sum Q_\phi(\boldsymbol{s},\pi_\theta(\boldsymbol{s})\,)\right)$$

Here:

- $s$ is the batch observations
- $a$ is the batch actions
- $Q_\phi$ is the critic network parameterized by $\phi$
- $\pi_\theta$ is the actor network parameterized by $\theta$
- $N$ is the mini batch size

```
function dQdTheta = deepActorGradient( ...
    actorNet,observation,gradData)

% Evaluate actions from current observations.
action = forward(actorNet,observation{:});

% Compute: q = Q(s,a)
q = predict(gradData.CriticNet,observation{:},action);

% Compute: qsum = -sum(q)/N to maximize q
qsum = -sum(q,"all")/gradData.MiniBatchSize;

% Compute: d(-sum(q)/N)/dActorParams
dQdTheta = dlgradient(qsum,actorNet.Learnables);
end
```

The environment reset function varies the initial conditions, reference trajectory, and noise seeds for every episode.

```
function in = resetFunction(in)
% Perturb the nominal reference amplitude and frequency.
LeadA = max(2 + 0.1*randn,0.1);
```

```
LeadF = max(1 + 0.1*randn,0.1);

% Perturb the nominal spacing.
L = 22 + 3*randn;

% Perturb the initial states.
InitialPositions  = [250 200 150 100 50] + 5*randn(1,5);
InitialVelocities = [10  10  10  10  10] + 1*randn(1,5);

% Update the noise seeds.
ParamNoiseSeed  = randi(100,1,3);
AccelNoiseSeed  = randi(100,1,5) + 100;
VelNoiseSeed    = randi(100,1,5) + 200;
PosNoiseSeed    = randi(100,1,5) + 300;

% Update the model variables.
in = setVariable(in,"L",L);
in = setVariable(in,"LeadA",LeadA);
in = setVariable(in,"LeadF",LeadF);
in = setVariable(in,"InitialPositions",InitialPositions);
in = setVariable(in,"InitialVelocities",InitialVelocities);
in = setVariable(in,"ParamNoiseSeed",ParamNoiseSeed);
in = setVariable(in,"AccelNoiseSeed",AccelNoiseSeed);
in = setVariable(in,"VelNoiseSeed",VelNoiseSeed);
in = setVariable(in,"PosNoiseSeed",PosNoiseSeed);
end
```

Create the critic and actor networks.

```
function [critic,actor] = createNetworks(obsInfo,actInfo)

% The actor and critic networks are initialized randomly.
% Ensure reproducibility by fixing random generator seed.
rng(0);

% Number of neurons in hidden layers
hiddenLayerSize = 64;

% Extract dimensions of observation and action spaces.
numObs = prod(obsInfo.Dimension);
numAct = prod(actInfo.Dimension);

% Use a Q-value function critic. This critic takes the current
% observation and an action as inputs and returns a single
% scalar as output (the estimated discounted cumulative long-term
% reward given the action from the state corresponding to the
% current observation, and following the policy thereafter).

% To model the parametrized Q-value function within the critic,
% use a neural network with two input layers (one for the
% observation channel and the other for the action channel),
% and one output layer (which returns the scalar value).

% Create the critic network.
% Define each network path as an array of layer objects.
obsInput = featureInputLayer(numObs, ...
    Normalization="none", ...
    Name=obsInfo.Name);
```

```
actInput = featureInputLayer(numAct, ...
    Normalization="none", ...
    Name=actInfo.Name);
catPath = [
    concatenationLayer(1,2,Name="concat")
    fullyConnectedLayer(hiddenLayerSize,Name="fc1")
    reluLayer(Name="relu1")
    fullyConnectedLayer(hiddenLayerSize,Name="fc2")
    reluLayer(Name="relu2")
    fullyConnectedLayer(1,Name="q")
    ];

% Add layers to layergraph object.
net = layerGraph();
net = addLayers(net,obsInput);
net = addLayers(net,actInput);
net = addLayers(net,catPath);

% Connect layers.
net = connectLayers(net,obsInfo.Name,"concat/in1");
net = connectLayers(net,actInfo.Name,"concat/in2");

% Convert to dlnetwork object.
net = dlnetwork(net);

% Create the critic object.
critic = rlQValueFunction(net,obsInfo,actInfo);

% Set critic to accelerate gradient computation.
critic = accelerate(critic,true);

% Use a continuous deterministic actor.
% This actor learns a parametrized deterministic policy
% for a continuous action space. It takes the current
% observation as input and returns as output an action
% that is a deterministic function of the observation.
%
% To model the parametrized policy within the actor, use a
% neural network with one input layer (which receives the
% content of the environment observation channel)
% and one output layer (which returns the action to the
% environment action channel).

% Define scale and bias for the output layer
scale = (actInfo.UpperLimit - actInfo.LowerLimit)/2;
bias  = actInfo.LowerLimit + scale;

% Create the actor network as an array of layer objects.
% Use tanhLayer to scale the signal to the (-1,1) range,
% and scalingLayer to scale the output to the action range.
obsPath = [
    featureInputLayer(numObs, ...
        Normalization="none", ...
        Name=obsInfo.Name)
    fullyConnectedLayer(hiddenLayerSize,Name="fc1")
    reluLayer(Name="relu1")
    fullyConnectedLayer(numAct,Name="fc2")
    reluLayer(Name="relu2")
```

```
    fullyConnectedLayer(numAct,Name="fc3")
    tanhLayer(Name="tanh1")
    scalingLayer(Scale=scale,...
        Bias=bias,...
        Name=actInfo.Name)
    ];

% Add layers to layergraph object.
net = layerGraph;
net = addLayers(net,obsPath);

% Convert to dlnetwork object.
net = dlnetwork(net);

% Create the actor object.
actor = rlContinuousDeterministicActor(net,obsInfo,actInfo);

% Set actor to accelerate gradient computation.
actor = accelerate(actor,true);

end
```

**References**

[1] Rajamani, Rajesh. *Vehicle Dynamics and Control*. 2. ed. Mechanical Engineering Series. New York, NY Heidelberg: Springer, 2012.

## See Also

**Functions**
runEpisode | setup | cleanup | gradient | rlOptimizer | rlSimulinkEnv

**Objects**
rlReplayMemory | rlQValueFunction | rlContinuousDeterministicActor

**Blocks**
RL Agent

## Related Examples

- “Design Controller for Vehicle Platooning” (Simulink Control Design)
- “Train Reinforcement Learning Policy Using Custom Training Loop” on page 5-433
- “Train Custom LQR Agent” on page 5-466
- “Create Agent for Custom Reinforcement Learning Algorithm” on page 5-456

## More About

- “Create Custom Reinforcement Learning Agents” on page 3-68
- “Train Reinforcement Learning Agents” on page 5-3

# Create Agent for Custom Reinforcement Learning Algorithm

This example shows how to create a custom agent for your own custom reinforcement learning algorithm. Doing so allows you to leverage the following built-in functionality from the Reinforcement Learning Toolbox™ software.

- Access to all agent functions, including `train` and `sim`
- Visualize training progress using the Episode Manager
- Train agents within a Simulink® environment

In this example, you convert a custom REINFORCE training loop into a custom agent class. For more information on the REINFORCE custom train loop, see “Train Reinforcement Learning Policy Using Custom Training Loop” on page 5-433. For more information on writing custom agent classes, see “Create Custom Reinforcement Learning Agents” on page 3-68.

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create Environment

Create the same training environment used in the “Train Reinforcement Learning Policy Using Custom Training Loop” on page 5-433 example. The environment is a cart-pole balancing environment with a discrete action space. Create the environment using the `rlPredefinedEnv` function.

```
env = rlPredefinedEnv("CartPole-Discrete");
```

Extract the observation and action specifications from the environment.

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

Obtain the dimension of the observation space (`numObs`) and the number of possible actions (`numAct`).

```
numObs = obsInfo.Dimension(1);
numAct = numel(actInfo.Elements);
```

For more information on this environment, see “Load Predefined Control System Environments” on page 2-23.

### Define Policy

The reinforcement learning policy in this example is a parametrized discrete-action stochastic policy, which is learned by a discrete categorical actor. This actor takes an observation as input and returns as output a random action sampled (among the finite number of possible actions) from a categorical probability distribution.

To model the parametrized policy within the actor, use a neural network with one input layer (which receives the content of the environment observation channel, as specified by `obsInfo`) and one output layer. The output layer must return a vector of probabilities for each possible action, as specified by `actInfo`.

Define the network as an array of layer objects, using `fullyConnectedLayer`, `reluLayer`, and `softmaxLayer` layers. The `softmaxLayer` ensures that the policy outputs probability values in the range [0 1] and that all probabilities sum to 1.

```
actorNetwork = [
    featureInputLayer(numObs)
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(2)
    softmaxLayer
    ];
```

Convert to a `dlnetwork` object and summarize properties.

```
actorNetwork = dlnetwork(actorNetwork)
```

```
actorNetwork =
  dlnetwork with properties:

         Layers: [7x1 nnet.cnn.layer.Layer]
    Connections: [6x2 table]
     Learnables: [6x3 table]
          State: [0x3 table]
     InputNames: {'input'}
    OutputNames: {'softmax'}
    Initialized: 1

  View summary with summary.
```

```
summary(actorNetwork)
```

```
   Initialized: true

   Number of learnables: 770

   Inputs:
      1   'input'   4 features
```

Create the actor using an `rlDiscreteCategoricalActor` object.

```
actor = rlDiscreteCategoricalActor(actorNetwork,obsInfo,actInfo);
```

Accelerate the gradient computation of the actor.

```
actor = accelerate(actor,true);
```

Create the optimizer options and `rlOptimizerOptions` function.

```
actorOpts = rlOptimizerOptions(LearnRate=1e-3);
```

### Custom Agent Class

To define your custom agent, first create a class that is a subclass of the `rl.agent.CustomAgent` class. The custom agent class for this example is defined in `CustomReinforceAgent.m`.

The `CustomReinforceAgent` class has the following class definition, which indicates the agent class name and the associated abstract agent.

```
classdef CustomReinforceAgent < rl.agent.CustomAgent
```

To define your agent you must specify the following:

- Agent properties
- Constructor function
- Critic approximator, to estimate the value of the policy (if required)
- Actor, to learn the policy (if required)
- Required agent methods
- Optional agent methods

### Agent Properties

In the `properties` section of the class file, specify any parameters necessary for creating and training the agent.

The `rl.Agent.CustomAgent` class already includes properties for the agent sample time (`SampleTime`) and the action and observation specifications (`ActionInfo` and `ObservationInfo`, respectively).

The custom REINFORCE agent defines the following additional agent properties.

```
properties
    % Actor
    Actor
    ActorOptimizer

    % Agent options
    Options

    % Experience buffer
    ObservationBuffer
    ActionBuffer
    RewardBuffer
end

properties (Access = private)
    % Training utilities
    Counter
    NumObservation
    NumAction
end
```

### Constructor Function

To create your custom agent, you must define a constructor function. The constructor function performs the following actions.

- Defines the action and observation specifications. For more information about creating these specifications, see `rlNumericSpec` and `rlFiniteSetSpec`.
- Sets the agent properties.

- Calls the constructor of the base abstract class.
- Defines the sample time (required for training in Simulink environments).

For example, the `CustomREINFORCEAgent` constructor defines action and observation spaces based on the input actor.

```
function obj = CustomReinforceAgent(Actor,Options)
    %CUSTOMREINFORCEAGENT Construct custom agent
    %   AGENT = CUSTOMREINFORCEAGENT(ACTOR,OPTIONS) creates custom
    %   REINFORCE AGENT from rlStochasticActorRepresentation ACTOR
    %   and structure OPTIONS. OPTIONS has fields:
    %       - DiscountFactor
    %       - MaxStepsPerEpisode

    % (required) Call the abstract class constructor.
    obj = obj@rl.agent.CustomAgent();
    obj.ObservationInfo = Actor.ObservationInfo;
    obj.ActionInfo = Actor.ActionInfo;

    % (required for Simulink environment) Register sample time.
    % For MATLAB environment, use -1.
    obj.SampleTime = -1;

    % (optional) Register actor and agent options.
    obj.Actor = Actor;
    obj.Options = Options;
    obj.ActorOptimizer = rlOptimizer(Options.OptimizerOptions);

    % (optional) Cache the number of observations and actions.
    obj.NumObservation = prod(obj.ObservationInfo.Dimension);
    obj.NumAction = prod(obj.ActionInfo.Dimension);

    % (optional) Initialize buffer and counter.
    resetImpl(obj);
end
```

#### Required Functions

To create a custom reinforcement learning agent you must define the following implementation functions.

- `getActionImpl` — Evaluates agent policy and selects an action during simulation.
- `getActionWithExplorationImpl` — Evaluates policy and selects an action with exploration during training.
- `learnImpl` — Updates learnable parameters, therefore allowing the agent to learn from the current experience.

To call these functions in your own code, use the wrapper methods from the abstract base class. For example, to call `getActionImpl`, use `getAction`. The wrapper methods have the same input and output arguments as the implementation methods.

#### `getActionImpl` Function

The `getActionImpl` function is used to evaluate the policy of your agent and select an action when simulating the agent using the `sim` function. This function must have the following signature, where `obj` is the agent object, `Observation` is the current observation, and `Action` is the selected action.

```
function Action = getActionImpl(obj,Observation)
```

For the custom REINFORCE agent, you select an action by calling the `getAction` function for the actor. The `rlDiscreteCategoricalActor` object generates a discrete distribution from an observation and then samples an action from that distribution.

```
function Action = getActionImpl(obj,Observation)
    % Compute an action using the policy given the current
    % observation.

    Action = getAction(obj.Actor,Observation);
end
```

#### `getActionWithExplorationImpl` Function

The `getActionWithExplorationImpl` function selects an action using the exploration model of your agent when training the agent using the `train` function. Using this function you can implement exploration techniques such as epsilon-greedy exploration or the addition of Gaussian noise. This function must have the following signature, where `obj` is the agent object, `Observation` is the current observation, and `Action` is the selected action.

```
function Action = getActionWithExplorationImpl(obj,Observation)
```

For the custom REINFORCE agent, the `getActionWithExplorationImpl` function is the same as `getActionImpl`. By default, stochastic actors always explore, that is, they always select an action based on a probability distribution.

```
function Action = getActionWithExplorationImpl(obj,Observation)
    % Compute an action using the exploration policy given the
    % current observation.

    % REINFORCE: Stochastic actors always explore by default
    % (sample from a probability distribution)
    Action = getAction(obj.Actor,Observation);
end
```

#### `learnImpl` Function

The `learnImpl` function defines how the agent learns from the current experience. This function implements the custom learning algorithm of your agent by updating the policy parameters and selecting an action with exploration for the next state. This function must have the following signature, where `obj` is the agent object, `Experience` is the current agent experience, and `Action` is the selected action.

```
function Action = learnImpl(obj,Experience)
```

The agent experience is the cell array `Experience = {observation,action,reward,nextstate,isdone}`. Here:

- `observation` is the current observation.
- `action` is the current action. This is different from the output argument `Action`, which is an action for the next state.
- `reward` is the current reward.
- `nextState` is the next observation.
- `isDone` is a logical flag indicating that the training episode is complete.

```
function Action = learnImpl(obj,Experience)
    % Define how the agent learns from an Experience, which is a
    % cell array with the following format.
    %   Experience = ...
    %   {observation,action,reward,nextObservation,isDone}

    % Reset buffer at the beginning of the episode.
    if obj.Counter < 2
        resetBuffer(obj);
    end

    % Extract data from experience.
    Obs = Experience{1};
    Action = Experience{2};
    Reward = Experience{3};
    NextObs = Experience{4};
    IsDone = Experience{5};

    % Save data to buffer.
    obj.ObservationBuffer(:,:,obj.Counter) = Obs{1};
    obj.ActionBuffer(:,:,obj.Counter) = Action{1};
    obj.RewardBuffer(:,obj.Counter) = Reward;

    if ~IsDone
        % Choose an action for the next state.

        Action = getActionWithExplorationImpl(obj, NextObs);
        obj.Counter = obj.Counter + 1;
    else
        % Learn from episodic data.

        % Collect data from the buffer.
        BatchSize = min(obj.Counter,obj.Options.MaxStepsPerEpisode);
        ObservationBatch = obj.ObservationBuffer(:,:,1:BatchSize);
        ActionBatch = obj.ActionBuffer(:,:,1:BatchSize);
        RewardBatch = obj.RewardBuffer(:,1:BatchSize);

        % Compute the discounted future reward.
        DiscountedReturn = zeros(1,BatchSize);
        for t = 1:BatchSize
            G = 0;
            for k = t:BatchSize
                G = G + ...
                obj.Options.DiscountFactor ^ (k-t) * RewardBatch(k);
            end
            DiscountedReturn(t) = G;
        end

        % Organize data to pass to the loss function.
        LossData.batchSize = BatchSize;
        LossData.actInfo = obj.ActionInfo;
        LossData.actionBatch = ActionBatch;
        LossData.discountedReturn = DiscountedReturn;

        % Compute the gradient of the loss with respect to the
        % actor parameters.
        ActorGradient = gradient(obj.Actor,@lossFunction,...
            {ObservationBatch},LossData);
```

```
        % Update the actor parameters using the computed gradients.
        [obj.Actor,obj.ActorOptimizer] = update( ...
            obj.ActorOptimizer,obj.Actor,ActorGradient);

        % Reset the counter.
        obj.Counter = 1;
    end
end
```

For the custom REINFORCE agent, replicate steps 2 through 7 of the custom training loop in “Train Reinforcement Learning Policy Using Custom Training Loop” on page 5-433. You omit steps 1, 8, and 9 since you will use the built-in `train` function to train your agent.

The actor computes the gradient of the loss function with respect to the parameters. The loss function in the REINFORCE algorithm the product between the discounted reward and the logarithm of the probability distribution of the action (coming from the policy evaluation for a given observation), summed across all time steps.

This computation is implemented in the local function `lossFunction` in `CustomREINFORCEAgent.m`. The function first input parameter must be a cell array like the one returned from the evaluation of a function approximator object. For more information, see the description of `outData` in `evaluate`. The second, optional, input argument contains additional data that might be needed by the loss calculation function. For more information, see `gradient`.

```
function loss = lossFunction(ActProbCell,lossData)

    ActProb = ActProbCell{1};
    % Create the action indication matrix.
    batchSize = lossData.batchSize;
    Z = repmat(lossData.actInfo.Elements',1,batchSize);
    actionIndicationMatrix = lossData.actionBatch(:,:) == Z;

    % Resize the discounted return to the size of ActProb.
    G = actionIndicationMatrix .* lossData.discountedReturn;
    G = reshape(G,size(ActProb));

    % Round any action probability values less than eps to eps.
    ActProb(ActProb < eps) = eps;

    % Compute the loss.
    loss = -sum(G .* log(ActProb),'all');

end
```

### Optional Functions

Optionally, you can define how your agent is reset at the start of training by specifying a `resetImpl` function with the following function signature, where `obj` is the agent object.

```
function resetImpl(obj)
```

Using this function, you can set the agent into a know or random condition before training.

```
function resetImpl(obj)
    % (Optional) Define how the agent is reset before training/
```

```
    resetBuffer(obj);
    obj.Counter = 1;
end
```

Also, you can define any other helper functions in your custom agent class as required. For example, the custom REINFORCE agent defines a `resetBuffer` function for reinitializing the experience buffer at the beginning of each training episode.

```
function resetBuffer(obj)

    % Reinitialize observation buffer.
    obj.ObservationBuffer = zeros( ...
        obj.NumObservation, ...
        1, ...
        obj.Options.MaxStepsPerEpisode);

    % Reinitialize action buffer.
    obj.ActionBuffer = zeros(obj.NumAction, ...
        1, ...
        obj.Options.MaxStepsPerEpisode);

    % Reinitialize reward buffer.
    obj.RewardBuffer = zeros(1,obj.Options.MaxStepsPerEpisode);

end
```

#### Create Custom Agent

Once you have defined your custom agent class, create an instance of it in the MATLAB workspace. To create the custom REINFORCE agent, first specify the agent options.

```
options.MaxStepsPerEpisode = 250;
options.DiscountFactor = 0.995;
options.OptimizerOptions = actorOpts;
```

Then, using the options and the previously defined actor, call the constructor function of the custom agent.

```
agent = CustomReinforceAgent(actor,options);
```

#### Train Custom Agent

Configure the training to use the following options.

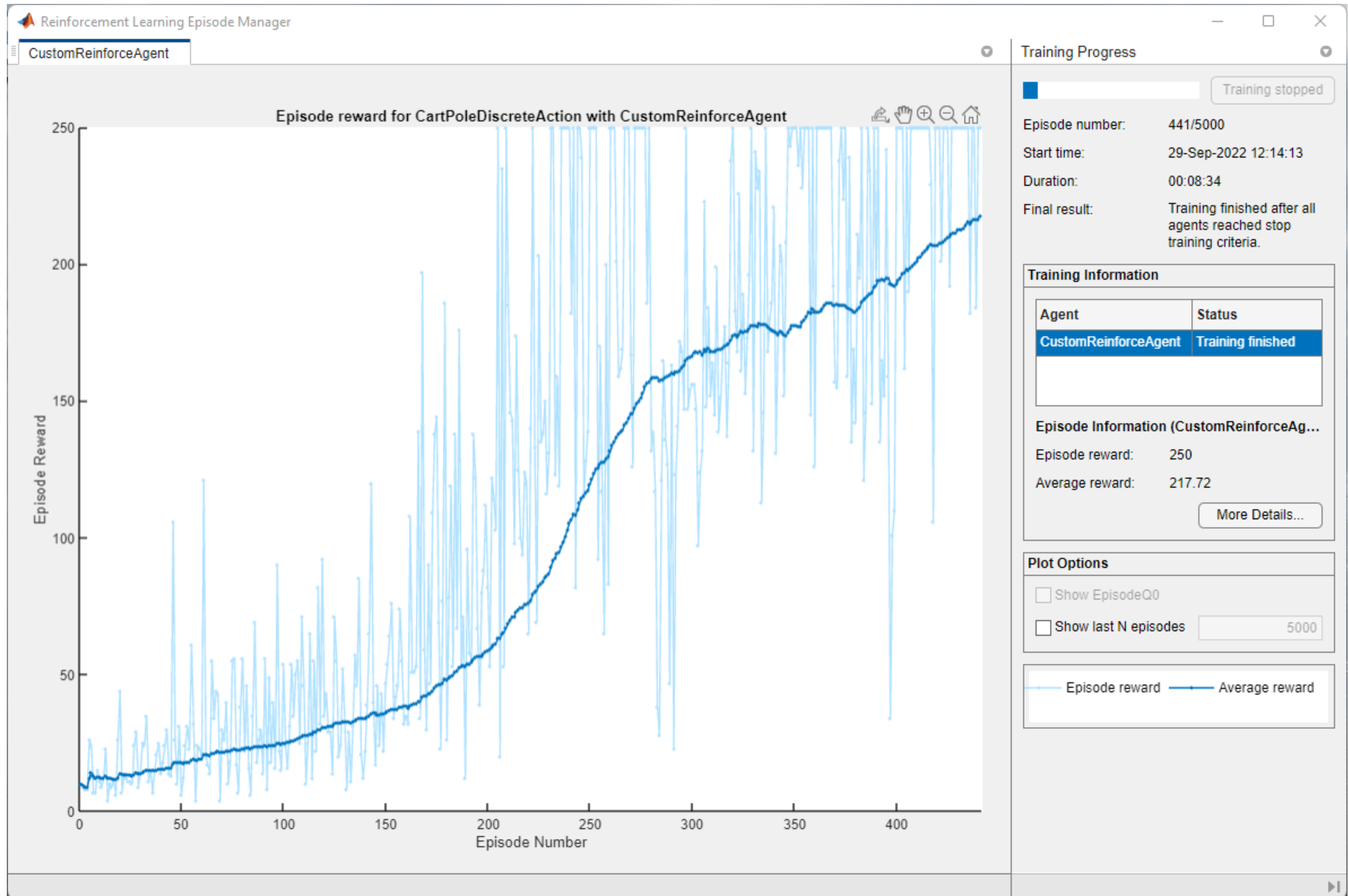
- Set up the training to last at most 5000 episodes, with each episode lasting at most 250 steps.
- Terminate the training after the maximum number of episodes is reached or when the average reward across 100 episodes reaches a value of 220.

For more information, see `rlTrainingOptions`.

```
numEpisodes = 5000;
aveWindowSize = 100;
trainingTerminationValue = 220;
trainOpts = rlTrainingOptions(...
    MaxEpisodes=numEpisodes,...
    MaxStepsPerEpisode=options.MaxStepsPerEpisode,...
    ScoreAveragingWindowLength=aveWindowSize,...
    StopTrainingValue=trainingTerminationValue);
```

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

```
doTraining = false;
if doTraining
    % Train the agent.
    trainStats = train(agent,env,trainOpts);
else
    % Load pretrained agent for the example.
    load("CustomReinforce.mat","agent");
end
```



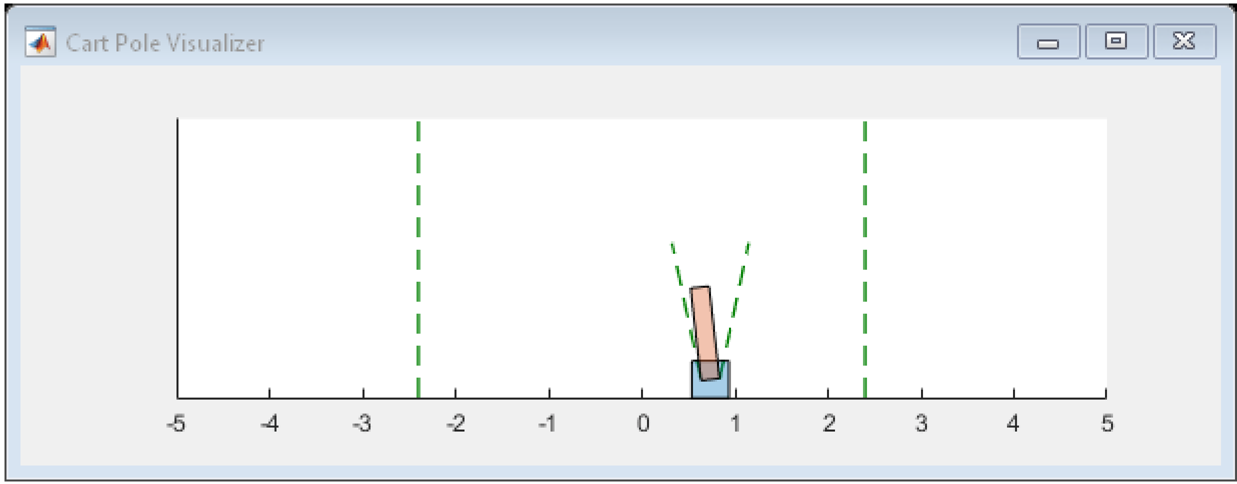

## Simulate Custom Agent

Enable the environment visualization, which is updated each time the environment `step` function is called.

```
plot(env)
```

To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOpts = rlSimulationOptions(MaxSteps=options.MaxStepsPerEpisode);
experience = sim(env,agent,simOpts);
```




## See Also

**Functions**
rlPredefinedEnv | train | evaluate | gradient | accelerate | getAction | sim

**Objects**
rlDiscreteCategoricalActor | rlNumericSpec | rlFiniteSetSpec | rlTrainingOptions | rlSimulationOptions

## Related Examples

- “Train Custom LQR Agent” on page 5-466
- “Train Reinforcement Learning Policy Using Custom Training Loop” on page 5-433

## More About

- “Train Reinforcement Learning Agents” on page 5-3
- “Create Custom Reinforcement Learning Agents” on page 3-68

# Train Custom LQR Agent

This example shows how to train a custom linear quadratic regulation (LQR) agent to control a discrete-time linear system modeled in MATLAB®. For an example of how a DDPG agent can be used as an optimal controller for a discrete-time system, see “Train DDPG Agent to Control Double Integrator System” on page 5-77.

### Create Linear System Environment

The reinforcement learning environment for this example is a discrete-time linear system. The dynamics for the system are given by

$$x_{t+1} = Ax_t + Bu_t$$

The feedback control law is

$$u_t = -Kx_t$$

The control objective is to minimize the quadratic cost: $J = \sum_{t=0}^{\infty}(x_t'Qx_t + u_t'Ru_t)$.

In this example, the system matrices are

$$A = \begin{bmatrix} 1.05 & 0.05 & 0.05 \\ 0.05 & 1.05 & 0.05 \\ 0 & 0.05 & 1.05 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.1 & 0 & 0.2 \\ 0.1 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$$

```
A = [1.05,0.05,0.05;0.05,1.05,0.05;0,0.05,1.05];
B = [0.1,0,0.2;0.1,0.5,0;0,0,0.5];
```

The quadratic cost matrices are:

$$Q = \begin{bmatrix} 10 & 3 & 1 \\ 3 & 5 & 4 \\ 1 & 4 & 9 \end{bmatrix}$$

$$R = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$$

```
Q = [10,3,1;3,5,4;1,4,9];
R = 0.5*eye(3);
```

For this environment, the reward at time $t$ is given by $r_t = -x_t"Qx_t - u_t"Ru_t$, which is the negative of the quadratic cost. Therefore, maximizing the reward minimizes the cost. The initial conditions are set randomly by the reset function.

Create the MATLAB environment interface for this linear system and reward. The `myDiscreteEnv` function creates an environment by defining custom `step` and `reset` functions. For more information

on creating such a custom environment, see “Create MATLAB Environment Using Custom Functions” on page 2-41.

```
env = myDiscreteEnv(A,B,Q,R);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create Custom LQR Agent

For the LQR problem, the Q-value function for a given control gain $K$ is quadratic and can be defined as $Q_K(x,u) = \begin{bmatrix} x \\ u \end{bmatrix}' H_K \begin{bmatrix} x \\ u \end{bmatrix}$, where $H_K = \begin{bmatrix} H_{\mathrm{xx}} & H_{\mathrm{xu}} \\ H_{\mathrm{ux}} & H_{\mathrm{uu}} \end{bmatrix}$ is a symmetric, positive definite matrix.

The control law that maximizes $Q_K$ is $u = -(H_{\mathrm{uu}})^{-1} H_{\mathrm{ux}}\, x$, so the feedback gain is $K = -(H_{\mathrm{uu}})^{-1} H_{\mathrm{ux}}$.

The matrix $H_K$ contains $m = \frac{1}{2}n(n+1)$ distinct element values, where $n$ is the sum of the number of states and number of inputs. Denote $\theta$ as the vector containing these $m$ elements, in which the off-diagonal elements in $H_K$ are multiplied by two. The elements of $\theta$ are the parameters that the custom agent needs to learn.

You can express the Q-value function as the inner product of the vectors $\theta$ and $\phi(x,u)$: $Q_K(x,u) = \theta' \phi(x,u)$, where $\phi(x,u)$ is a vector of quadratic monomials built from the combination of all the elements in $x$ and $u$. For an example, see the $Q(x,u)$ matrix in “Train DDPG Agent to Control Double Integrator System” on page 5-77.

The LQR agent starts with a stabilizing controller $K_0$. To get an initial stabilizing controller, place the poles of the closed-loop system $A - BK_0$ inside the unit circle.

```
K0 = place(A,B,[0.4,0.8,0.5]);
```

To create a custom agent, you must create a subclass of the `rl.agent.CustomAgent` abstract class. For the custom LQR agent, the defined custom subclass is `LQRCustomAgent`. For more information, see “Create Custom Reinforcement Learning Agents” on page 3-68. Create the custom LQR agent using $Q$, $R$, and $K_0$. The agent does not require information on the system matrices $A$ and $B$.

```
agent = LQRCustomAgent(Q,R,K0);
```

For this example, set the agent discount factor to one. To use a discounted future reward, set the discount factor to a value less than one.

```
agent.Gamma = 1;
```

Because the linear system has three states and three inputs, the total number of learnable parameters is $m = 21$. To ensure satisfactory performance of the agent, set the number of parameter estimates $N_p$ (the number of data point to be collected before updating the critic) to be greater than twice the number of learnable parameters. In this example, the value is $N_p = 45$.

```
agent.EstimateNum = 45;
```

To get good estimation results for $\theta$, you must apply a persistently excited exploration model to the system. In this example, encourage model exploration by adding white noise to the controller output: $u_t = -Kx_t + e_t$. In general, the exploration model depends on the system models.

### Train Agent

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

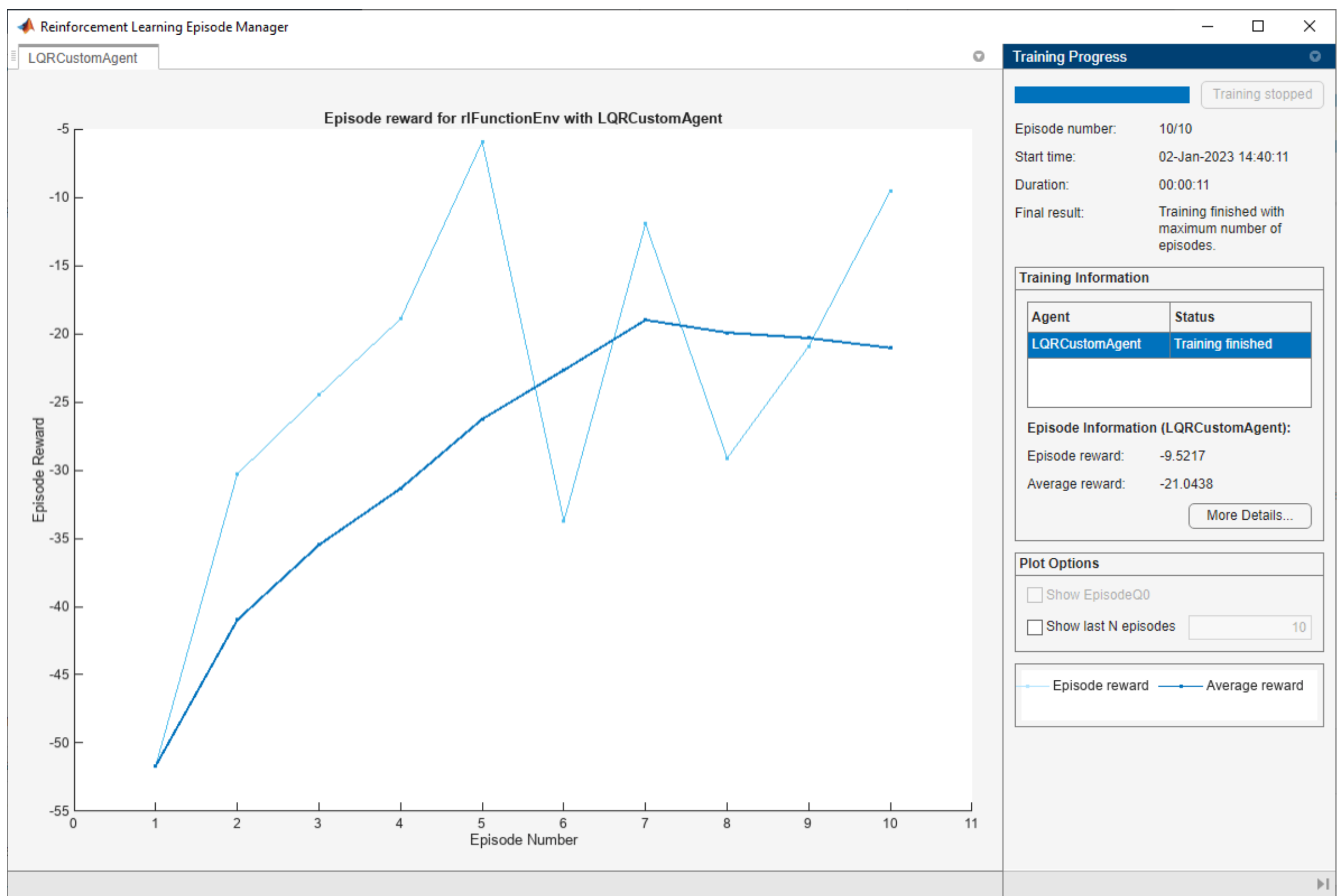
- Run each training episode for at most 10 episodes, with each episode lasting at most 50 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable command line display (set the `Verbose` option).

For more information, see `rlTrainingOptions`.

```
trainingOpts = rlTrainingOptions(...
    MaxEpisodes=10, ...
    MaxStepsPerEpisode=50, ...
    Verbose=false, ...
    Plots="training-progress");
```

Train the agent using the `train` function.

```
trainingStats = train(agent,env,trainingOpts);
```




### Simulate Agent and Compare with Optimal Solution

To validate the performance of the trained agent, simulate it within the MATLAB environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(MaxSteps=20);
experience = sim(env,agent,simOptions);
totalReward = sum(experience.Reward)
```

```
totalReward = -30.6482
```

You can compute the optimal solution for the LQR problem using the `dlqr` function.

```
[Koptimal,P] = dlqr(A,B,Q,R);
```

The optimal reward is given by $J_{\text{optimal}} = -x_0{}'Px_0$.

```
x0 = experience.Observation.obs1.getdatasamples(1);
Joptimal = -x0'*P*x0;
```

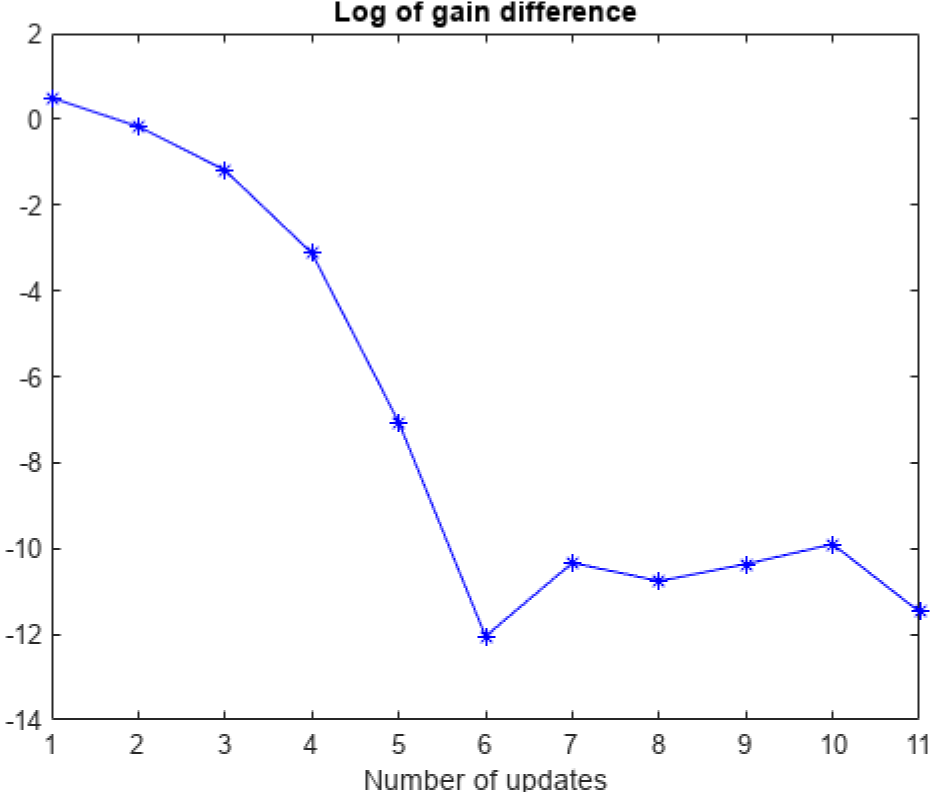
Compute the error in the reward between the trained LQR agent and the optimal LQR solution.

```
rewardError = totalReward - Joptimal
```

```
rewardError = 5.0439e-07
```

View the history of the norm of the difference between the gains between the trained LQR agent and the optimal LQR solution.

```
% Number of gain updates
len = agent.KUpdate;

% Initialize error vector
err = zeros(len,1);

% Fill elements
for i = 1:len
    err(i) = norm(agent.KBuffer{i}-Koptimal);
end

% Plot logarithm of the error vector
plot(log10(err),'b*-')
title("Log of gain difference")
xlabel("Number of updates")
```




Compute the norm of final error for the feedback gain.

```
gainError = norm(agent.K - Koptimal)
```

```
gainError = 3.3210e-12
```

Overall, the trained agent finds a solution that is very close to the true optimal LQR solution.

## See Also

**Functions**
`train` | `sim` | `lqr`

**Objects**
`rlTrainingOptions` | `rlSimulationOptions`

## Related Examples

- “Create Agent for Custom Reinforcement Learning Algorithm” on page 5-456
- “Train DDPG Agent to Control Double Integrator System” on page 5-77
- “Create MATLAB Environment Using Custom Functions” on page 2-41
- “Train Reinforcement Learning Policy Using Custom Training Loop” on page 5-433

## More About


- “Train Reinforcement Learning Agents” on page 5-3
- “Create Custom Reinforcement Learning Agents” on page 3-68

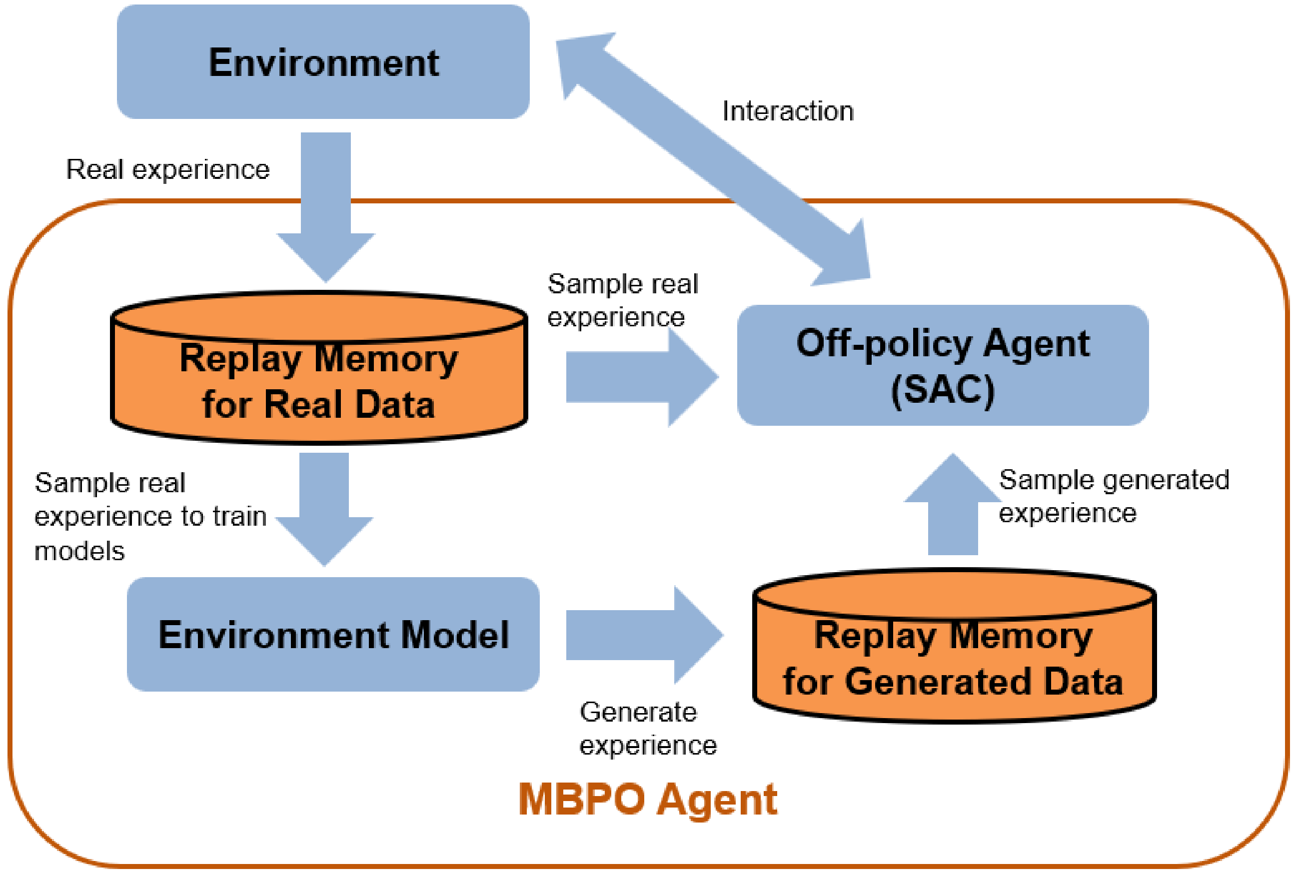
# Train MBPO Agent to Balance Cart-Pole System

This example shows how to train a model-based policy optimization (MBPO) agent to balance a cart-pole system modeled in MATLAB®. For more information on MBPO agents, see “Model-Based Policy Optimization (MBPO) Agents” on page 3-62.

MBPO agents use an environment model to generate more experiences while training a base agent. In this example, the base agent is a soft actor-critic (SAC) agent.

The build-int MBPO agent is based on a model-based policy optimization algorithm in [1]. The original MBPO algorithm trains an ensemble of stochastic models. In contrast, this example trains an ensemble of deterministic models.

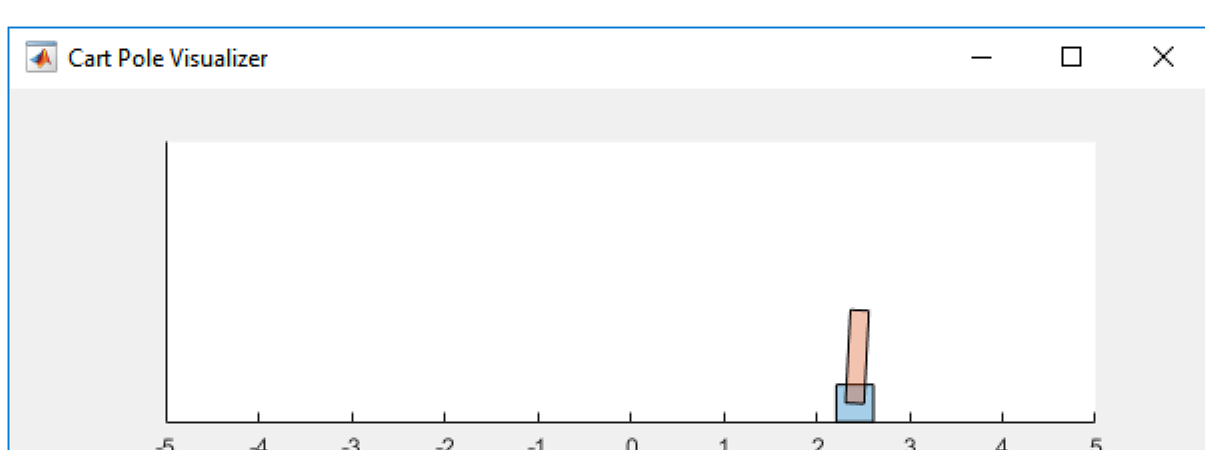
The following figure summarizes the algorithm used in this example. During training the MBPO agent collects real experiences resulting from interactions with the environment. The MBPO agent uses these experiences to train its internal environment model. Then, it uses this model to generate experiences without interacting with the actual environment Finally, the MBPO agent uses the real experiences and generated experiences to train the SAC base agent.




### Cart-Pole MATLAB Environment

For this example, the reinforcement learning environment is a pole attached to an unactuated revolutionary joint on a cart. The cart has an actuated prismatic joint connected to a one-dimensional

frictionless track. The training goal in this environment is to balance the pole by applying forces (actions) to the prismatic joint.




For this environment:

- The upward balanced pendulum position is `0` radians and the downward hanging position is `pi` radians.
- The pendulum starts upright with an initial angle between –0.05 radians and 0.05 radians.
- The force action signal from the agent to the environment is from –10 N to 10 N.
- The observations from the environment are the position and velocity of the cart, the pendulum angle, and the pendulum angle derivative.
- The episode terminates if the pole is more than 12 degrees from vertical or if the cart moves more than 2.4 m from the original position.
- A reward of +0.5 is provided for every time-step that the pole remains upright. An additional reward is provided based on the distance between the cart and the origin. A penalty of –50 is applied when the pendulum falls.

For more information on this model, see “Load Predefined Control System Environments” on page 2-23.

Create a predefined environment interface for the cart-pole system.

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

The interface has a continuous action space where the agent can apply one force value ranging from –10 N to 10 N.

Obtain the observation and action specifications from the environment interface.

```
obsInfo = getObservationInfo(env);
numObservations = obsInfo.Dimension(1);
actInfo = getActionInfo(env);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

### Create MBPO Agent

An MBPO agent decides which action to take given observations using a base off-policy agent. The MBPO agent trains both the base agent and an environmental model. The environmental model consists of transition functions, a reward function, and an is-done function. This model is used to create more samples without interacting with an environment. This example uses the following steps to construct an MBPO agent.

1. Define model-free off-policy agent.
2. Define transition models.
3. Define reward model.
4. Define is-done model.
5. Create neural network environment.
6. Create MBPO agent.

#### 1. Define Model-Free Off-Policy Agent

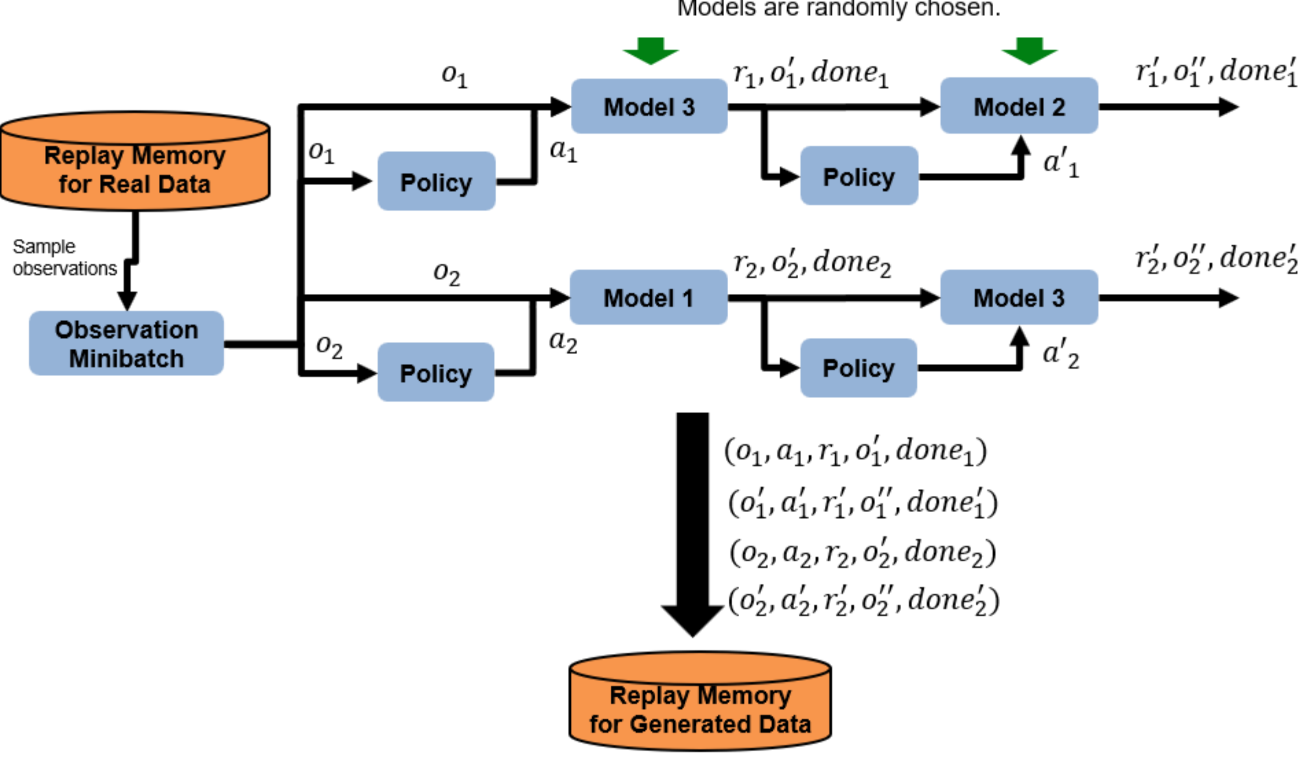
Create a SAC base agent with a default network structure. For more information on SAC agents, see “Soft Actor-Critic (SAC) Agents” on page 3-35. For an environment with a continuous action space, you can also use a DDPG or TD3 base agent. For discrete environments, you can use a DQN base agent.

```
agentOpts = rlSACAgentOptions;
agentOpts.MiniBatchSize = 256;
initOpts = rlAgentInitializationOptions(NumHiddenUnit=64);
baseagent = rlSACAgent(obsInfo,actInfo,initOpts,agentOpts);
baseagent.AgentOptions.ActorOptimizerOptions.LearnRate = 1e-4;
baseagent.AgentOptions.CriticOptimizerOptions(1).LearnRate = 1e-4;
baseagent.AgentOptions.CriticOptimizerOptions(2).LearnRate = 1e-4;
baseagent.AgentOptions.NumGradientStepsPerUpdate = 5;
```

#### 2. Define Transition Models

To model the environment, an MBPO agent trains one or more transition models. To model an environment effectively, you must consider two kinds of uncertainty: statistical uncertainty and modeling uncertainty. A stochastic transition function can model the statistical uncertainty better than a deterministic transition function. In this example, since the cart-pole environment is deterministic, you use deterministic transition functions.

It is challenging to have a perfect model, and a trained model usually has modeling uncertainty. One common approach to overcoming modeling uncertainty is to use multiple transition models. The original MBPO paper uses seven models [1]. For this example, to reduce computational cost, you use three models. The MBPO agent generates experiences using all three transition models. The following figure shows how an ensemble of transition models generates samples without interacting with the environment. In this figure, the models generate two trajectories with horizon = 2.




Create three deterministic transition functions. To do so, create a deep neural network using the `createDeterministicTransitionNetwork` helper function. Then, use the neural network to create an `rlContinuousDeterministicTransitionFunction` object. When creating a transition function object, you must specify the action and observation input/output names for the neural network.

```
net1 = createDeterministicTransitionNetwork(4,1);
transitionFcn = rlContinuousDeterministicTransitionFunction(net1,...
    obsInfo,...
    actInfo,...
    ObservationInputNames="state",...
    ActionInputNames="action",...
    NextObservationOutputNames="nextObservation");

net2 = createDeterministicTransitionNetwork(4,1);
transitionFcn2 = rlContinuousDeterministicTransitionFunction(net2,...
    obsInfo,...
    actInfo,...
    ObservationInputNames="state",...
    ActionInputNames="action",...
    NextObservationOutputNames="nextObservation");

net3 = createDeterministicTransitionNetwork(4,1);
transitionFcn3 = rlContinuousDeterministicTransitionFunction(net3,...
    obsInfo,...
    actInfo,...
    ObservationInputNames="state",...
    ActionInputNames="action",...
    NextObservationOutputNames="nextObservation");
```

#### 3. Define Reward Model

An MBPO agent also contains a reward model for the environment. If you know a ground-truth reward function, you can specify it using a custom function. In this example, the ground-truth reward function is defined in the `cartPoleRewardFunction` helper function. To use this reward function set `useGroundTruthReward` to `true`.

You can also specify a neural-network-based reward function that the MBPO agent can train. In this example, you can use such a reward function by setting `useGroundTruthReward` to `false`. The deep neural network for the reward function is defined in the `createRewardNetworkActionNextObs` helper function. To define an is-done function using the neural network, create an `rlContinuousDeterministicRewardFunction` object.

```
useGroundTruthReward = true;
if useGroundTruthReward
    rewardFcn = @cartPoleRewardFunction;
else
    % This neural network uses action and next observation as inputs.
    rewardnet = createRewardNetworkActionNextObs(4,1);
    rewardFcn = rlContinuousDeterministicRewardFunction(rewardnet,...
        obsInfo,...
        actInfo, ...
        ActionInputNames="action",...
        NextObservationInputNames="nextState");
end
```

#### 4. Define Is-Done Model

An MBPO agent also contains an is-done model for computing the termination signal for the environment. If you know a ground-truth termination signal, you can specify it using a custom function. In this example, the ground-truth termination signal is defined in the `cartPoleIsDoneFunction` helper function. To use this reward function set `useGroundTruthIsDone` to `true`.

You can also specify a neural-network-based is-done function that the MBPO agent can train. In this example, you can use such an is-done function by setting `useGroundTruthIsDone` to `false`. The deep neural network for the is-done function is defined in the `createIsDoneNetwork` helper function. To define an is-done function using the neural network, create an `rlIsDoneFunction` object.

```
useGroundTruthIsDone = true;
if useGroundTruthIsDone
    isdoneFcn = @cartPoleIsDoneFunction;
else
    % This neural network uses only next obesrvation as inputs.
    isdoneNet = createIsDoneNetwork(4);
    isdoneFcn = rlIsDoneFunction(isdoneNet,...
        obsInfo,...
        actInfo,...
        NextObservationInputNames="nextState");
end
```

#### 5. Create Neural Network Environment

Define a neural network environment using the transition, reward, and is-done functions. To do so, create an `rlNeuralNetworkEnvironment` object.

```
generativeEnv = rlNeuralNetworkEnvironment(obsInfo,actInfo, ...
    [transitionFcn,transitionFcn2,transitionFcn3],rewardFcn,isdoneFcn);
% Reset model environment.
reset(generativeEnv);
```

#### 6. Create MBPO Agent

Define an MBPO agent using the base off-policy agent and environment model. To do so, first create an MBPO agent options object.

```
MBPOAgentOpts = rlMBPOAgentOptions;
```

Specify options for training the environment model. Train the model for 1 epoch at the beginning of each episode and use 15 mini-batches of size 256.

```
MBPOAgentOpts.NumEpochForTrainingModel = 1;
MBPOAgentOpts.NumMiniBatches = 15;
MBPOAgentOpts.MiniBatchSize = 256;
```

Specify the size of the model experience buffer.

```
MBPOAgentOpts.ModelExperienceBufferLength = 60000;
```

Specify the ratio of real and generated experience used to train the base SAC agent. For this example, 20% of samples are from the real experience buffer and 80% of samples are from model experience buffer.

```
MBPOAgentOpts.RealSampleRatio = 0.2;
```

Specify options for generating samples using the environment model.

- Generate 20000 trajectories at the beginning of each epoch.
- Use a `piecewise` roll-out horizon schedule, which increases the horizon length gradually..
- Increase the horizon length every 100 epochs.
- Use an initial horizon length of 1.
- Use a maximum horizon length of 3.

```
MBPOAgentOpts.ModelRolloutOptions.NumRollout = 20000;
MBPOAgentOpts.ModelRolloutOptions.HorizonUpdateSchedule = "piecewise";
MBPOAgentOpts.ModelRolloutOptions.HorizonUpdateFrequency = 100;
MBPOAgentOpts.ModelRolloutOptions.Horizon = 1;
MBPOAgentOpts.ModelRolloutOptions.HorizonMax = 3;
```

Specify optimizer options for training the transition models. Use the same optimizer options for all three transition models.

```
transitionOptimizerOptions1 = rlOptimizerOptions(...
    LearnRate=1e-4,...
    GradientThreshold=1.0);
transitionOptimizerOptions2 = rlOptimizerOptions(...
    LearnRate=1e-4,...
    GradientThreshold=1.0);
transitionOptimizerOptions3 = rlOptimizerOptions(...
    LearnRate=1e-4,...
    GradientThreshold=1.0);
MBPOAgentOpts.TransitionOptimizerOptions = ...
```

```
    [transitionOptimizerOptions1,...
    transitionOptimizerOptions2,...
    transitionOptimizerOptions3];
```

Specify optimizer options for training the reward model. If you use a custom ground-truth reward function, the agent ignores these options.

```
rewardOptimizerOptions = rlOptimizerOptions(...
    LearnRate=1e-4,...
    GradientThreshold=1.0);
MBPOAgentOpts.RewardOptimizerOptions = rewardOptimizerOptions;
```

Specify optimizer options for training the is-done model. If you use a custom ground-truth reward function, the agent ignores these options.

```
isdoneOptimizerOptions = rlOptimizerOptions(...
    LearnRate=1e-4,...
    GradientThreshold=1.0);
MBPOAgentOpts.IsDoneOptimizerOptions = isdoneOptimizerOptions;
```

Create the MBPO agent, specifying the base agent, environment model, and options.

```
agent = rlMBPOAgent(baseagent,generativeEnv,MBPOAgentOpts);
```

### Train Agent

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

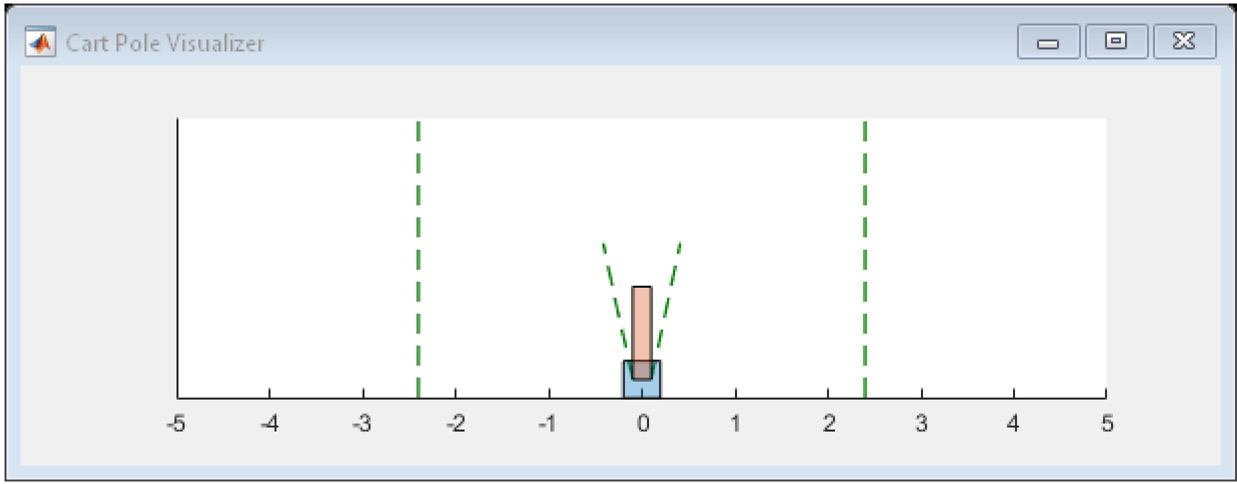
- Run each training episode for at most 500 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Save the agent when the average episode reward is greater than or equal to 470.
- Stop training when the agent receives an average cumulative reward greater than 470 over 5 consecutive episodes. At this point, the agent can balance the pendulum in the upright position.

For more information, see `rlTrainingOptions`.

```
trainOpts = rlTrainingOptions(...
    MaxEpisodes=500, ...
    MaxStepsPerEpisode=500, ...
    Verbose=false, ...
    Plots="training-progress",...
    StopTrainingCriteria="AverageReward",...
    StopTrainingValue=470,...
    ScoreAveragingWindowLength=5,...
    SaveAgentCriteria="EpisodeReward",...
    SaveAgentValue=470,...
    ScoreAveragingWindowLength=5);
```

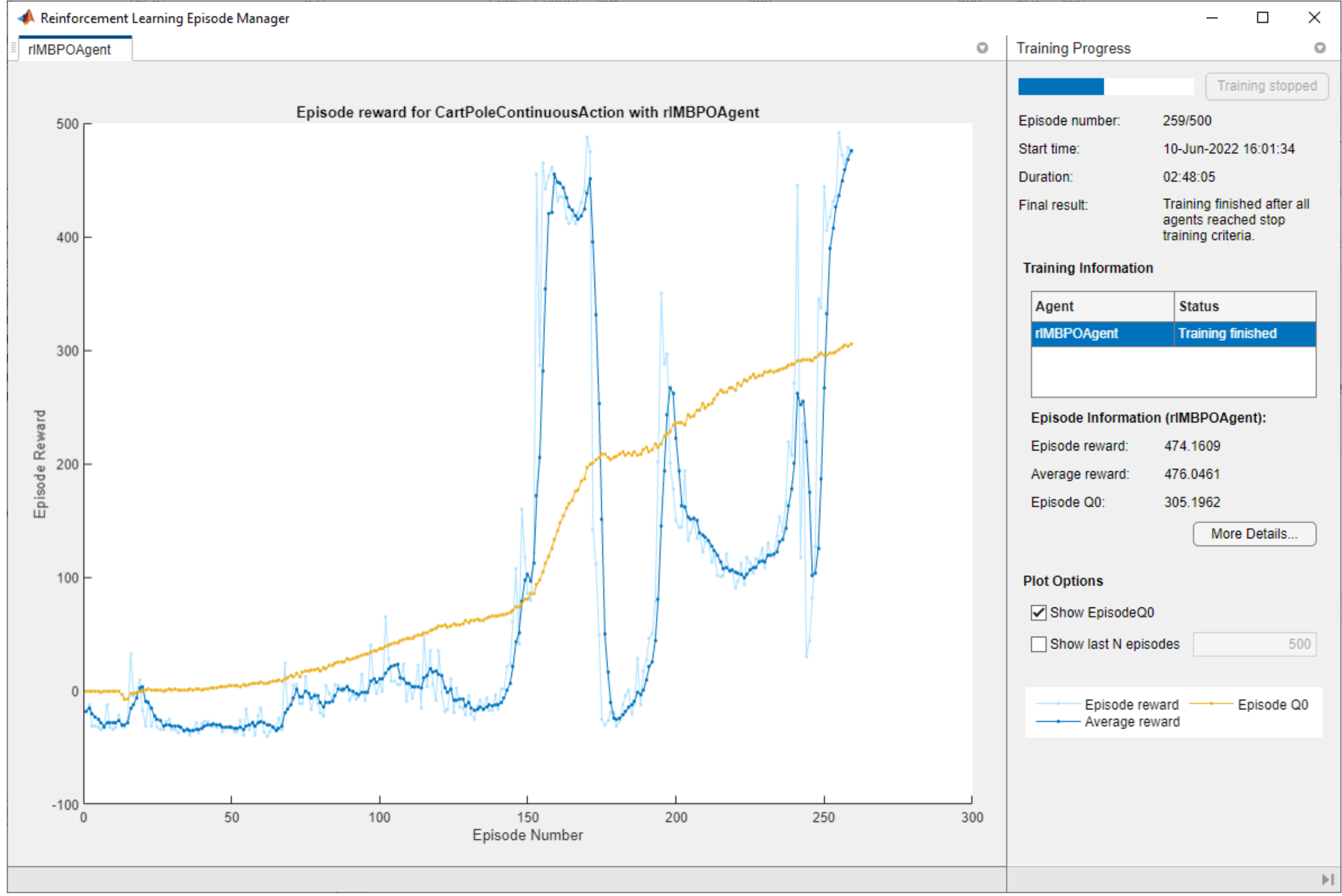
You can visualize the cart-pole system by using the `plot` function during training or simulation.

```
plot(env)
```




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

```
doTraining = false;

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




### Simulate MBPO Agent

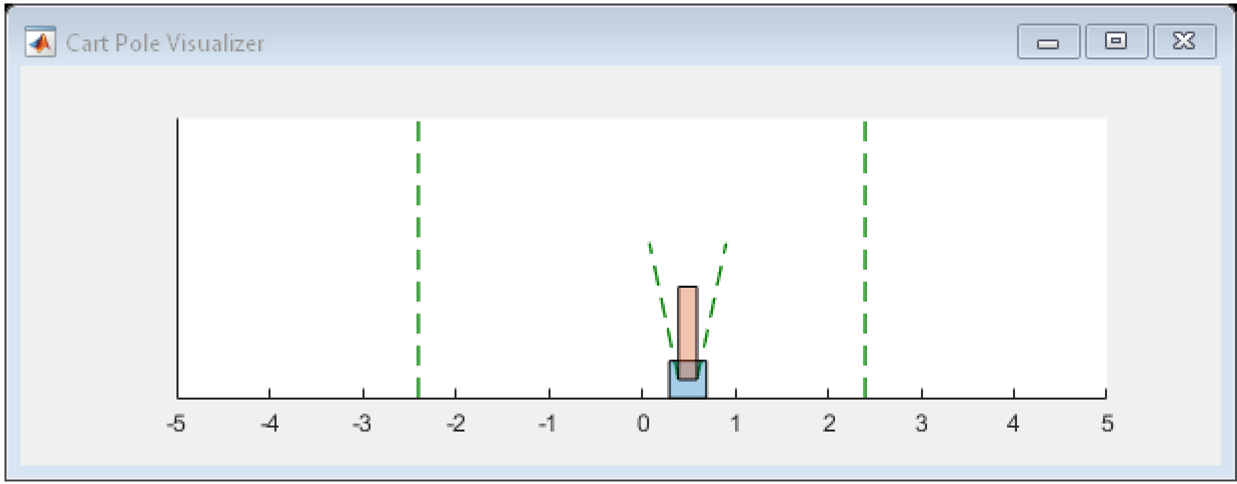
To validate the performance of the trained agent, simulate it within the cart-pole environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`. Exploration during validation is not necessary in this example. Therefore, to use deterministic actions during the simulation, set the `UseExplorationPolicy` agent property to agent to be `false`.

```
rng(1)

% Disable exploration during sim
agent.UseExplorationPolicy = false;

simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
totalReward_MBPO = sum(experience.Reward)
```

```
totalReward_MBPO = 460.7233
```

Instead of simulating the MBPO agent, you can simulate the base agent. If you use the same random seed, you get the same result as simulating the MBPO agent.

```
rng(1)
experience = sim(env,agent.BaseAgent,simOptions);
```




```
totalReward_SAC = sum(experience.Reward)
```

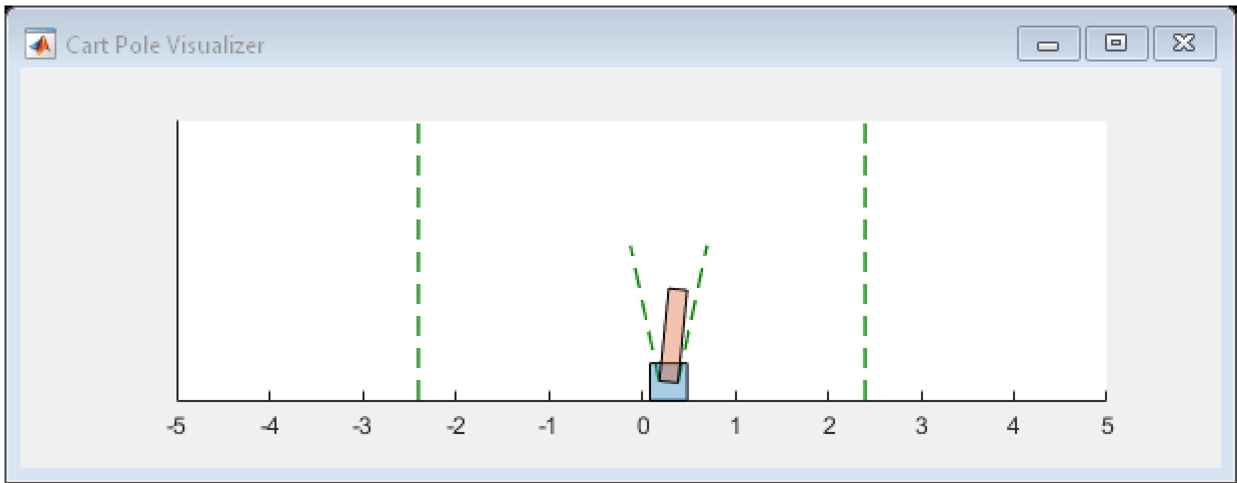
```
totalReward_SAC = 460.7233
```

**Evaluate Learned Environment Model**

To validate the trained environment transition models, you can check whether they are able to correctly predict the next observations. Similarly, you can validate the performance of the reward and is-done functions. To make a prediction based on the environment model, use the `step` function.

Collect data for learned model evaluation

```
rng(1)

% Enable exploration during sim to create
% diverse data for model evaluation
agent.UseExplorationPolicy = true;

simOptions = rlSimulationOptions(MaxSteps=500);
experience = sim(env,agent,simOptions);
```




For this example, evaluate the performance of the first transition model.

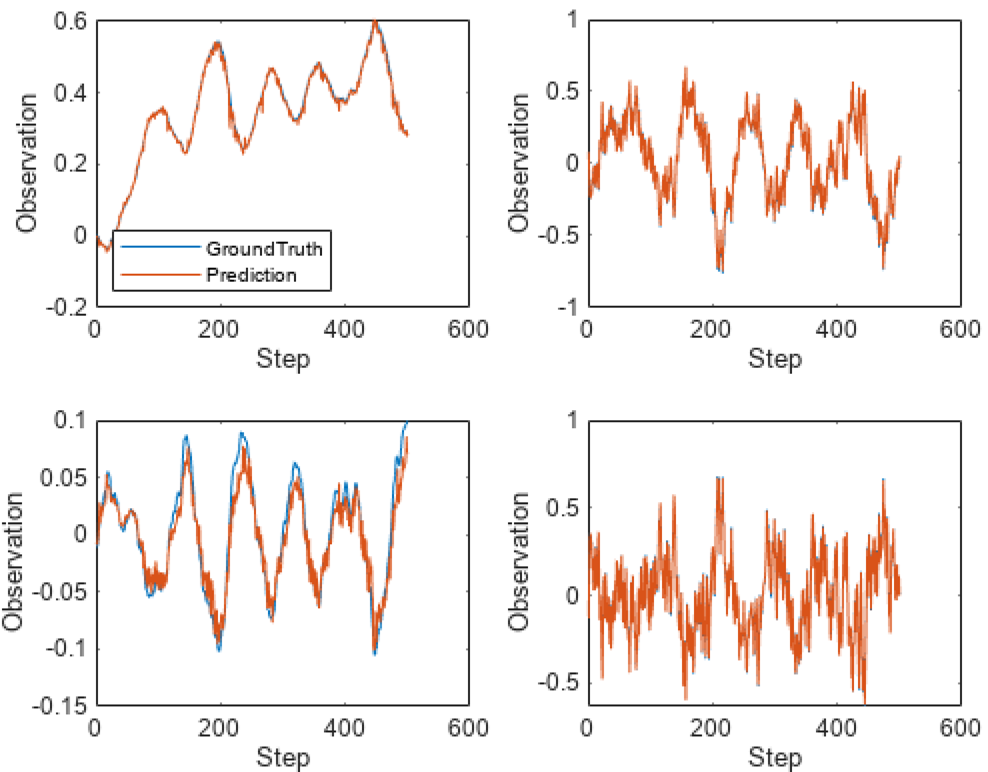
```
agent.EnvModel.TransitionModelNum = 1;
```

For each simulation step, extract the actual next observation.

```
numSteps = length(experience.Reward.Data);
nextObsPrediction = zeros(4,1,numSteps);
rewardPrediction = zeros(1,numSteps);
isdonePrediction = zeros(1,numSteps);
nextObsGroundTruth = zeros(4,1,numSteps);
rewardGroundTruth = zeros(1,numSteps);
isdoneGroundTruth = zeros(1,numSteps);
for stepCt = 1:numSteps
    % Extract the actual next observation, reward, and is-done value.
    nextObsGroundTruth(:,:,stepCt) = ...
        experience.Observation.CartPoleStates.Data(:,:,stepCt+1);
    rewardGroundTruth(:, stepCt) = experience.Reward.Data(stepCt);
    isdoneGroundTruth(:, stepCt) = experience.IsDone.Data(stepCt);

    % Predict the next observation, reward, and is-done value
    % using the environment model.
    obs = experience.Observation.CartPoleStates.Data(:,:,stepCt);
    agent.EnvModel.Observation = {obs};
    action = experience.Action.CartPoleAction.Data(:,:,stepCt);
    [nextObs,reward,isdone] = step(agent.EnvModel,{action});

    nextObsPrediction(:,:,stepCt) = nextObs{1};
    rewardPrediction(:,stepCt) = reward;
    isdonePrediction(:,stepCt) = isdone;
end
```

Plot the ground truth and prediction of each dimension of the observations.

```
figure
for obsDimensionIndex = 1:4
    subplot(2,2,obsDimensionIndex)
    plot(reshape(nextObsGroundTruth(obsDimensionIndex,:,:),1,numSteps))
    hold on
    plot(reshape(nextObsPrediction(obsDimensionIndex,:,:),1,numSteps))
    hold off
    xlabel("Step")
    ylabel("Observation")
    if obsDimensionIndex == 1
        legend("GroundTruth","Prediction","Location","southwest")
    end
end
```




### References

[1] Janner, Michael, Justin Fu, Marvin Zhang, and Sergey Levine. “When to Trust Your Model: Model-Based Policy Optimization.” In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 12519–30. 1122. Red Hook, NY, USA: Curran Associates Inc., 2019.

## See Also

### Functions
`train` | `sim`

### Objects
`rlMBPOAgent` | `rlNeuralNetworkEnvironment` | `rlMBPOAgentOptions` | `rlTrainingOptions` | `rlSimulationOptions` | `rlSACAgent`

## Related Examples

- “Model-Based Reinforcement Learning Using Custom Training Loop” on page 5-484

## More About

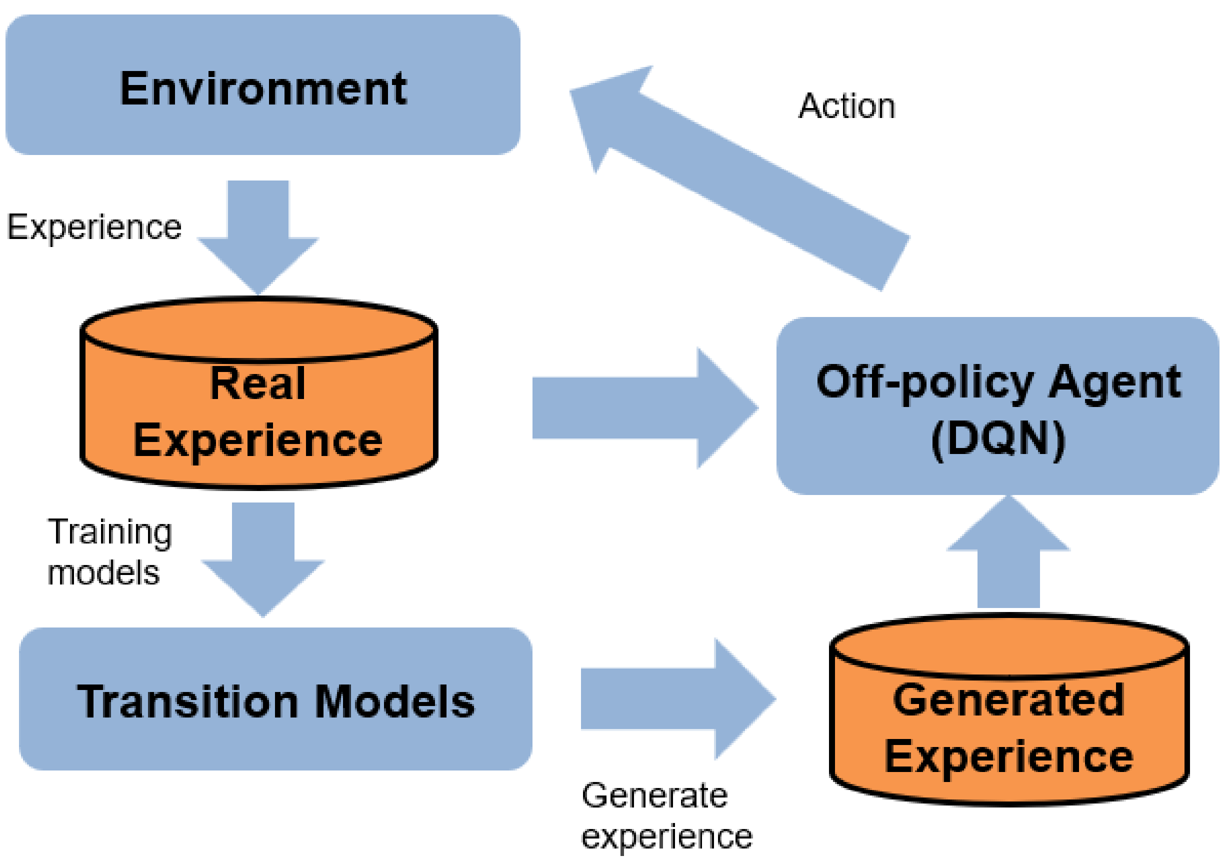
- “Load Predefined Control System Environments” on page 2-23
- “Soft Actor-Critic (SAC) Agents” on page 3-35
- “Model-Based Policy Optimization (MBPO) Agents” on page 3-62

# Model-Based Reinforcement Learning Using Custom Training Loop

This example shows how to define a custom training loop for a model-based reinforcement learning (MBRL) algorithm. You can use this workflow to train an MBRL policy with your custom training algorithm using policy and value function representations from Reinforcement Learning Toolbox™ software.

For an example on how to use the built in model-based policy optimization (MBPO) agent, see "Train MBPO Agent to Balance Cart-Pole System" on page 5-472. For an overview of built-in MBPO agents, see "Model-Based Policy Optimization (MBPO) Agents" on page 3-62.

In this example, you use transition models to generate more experiences while training a custom DQN [2] agent in a cart-pole environment. The algorithm used in this example is based on an MBPO algorithm [1]. The original MBPO algorithm trains an ensemble of stochastic models and a soft actor-critic (SAC) agent in tasks with continuous actions. In contrast, this example trains three deterministic models and a DQN agent in a task with discrete actions. The following figure summarizes the algorithm used in this example.



The agent generates real experiences by interacting with the environment. These experiences are used to train a set of transition models, which are used to generate additional experiences. The training algorithm then uses both the real and generated experiences to update the agent policy.

### Create Environment

For this example, a reinforcement learning policy is trained in a discrete cart-pole environment. The objective in this environment is to balance the pole by applying forces (actions) on the cart. Create the environment using the `rlPredefinedEnv` function. Fix the random generator seed for reproducibility. For more information on this environment, see “Load Predefined Control System Environments” on page 2-23.

```
clear
clc
rngSeed = 1;
rng(rngSeed);
env = rlPredefinedEnv("CartPole-Discrete");
```

Extract the observation and action specifications from the environment.

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

Obtain the number of observations (`numObservations`) and actions (`numActions`).

```
numObservations = obsInfo.Dimension(1);
% Number of discrete actions, -10 or 10
numActions = numel(actInfo.Elements);
numContinuousActions = 1; % force
```

### Critic Construction

DQN is a value-based reinforcement learning algorithm that estimates the discounted cumulative reward using a critic. In this example, the critic network contains `fullyConnectedLayer`, and `reluLayer` layers.

```
qNetwork = [
    featureInputLayer(obsInfo.Dimension(1))
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(24)
    reluLayer
    fullyConnectedLayer(length(actInfo.Elements))];
qNetwork = dlnetwork(qNetwork);
```

Create the critic representation using the specified neural network and options. For more information, see `rlQValueFunction`.

```
critic = rlVectorQValueFunction(qNetwork,obsInfo,actInfo);
```

Create optimizer objects for updating the critic. For more information, see `rlOptimizerOptions`.

```
optimizerOpt = rlOptimizerOptions(...
    LearnRate=1e-3, ...
    GradientThreshold=1);
criticOptimizer = rlOptimizer(optimizerOpt);
```

Create a max Q value policy

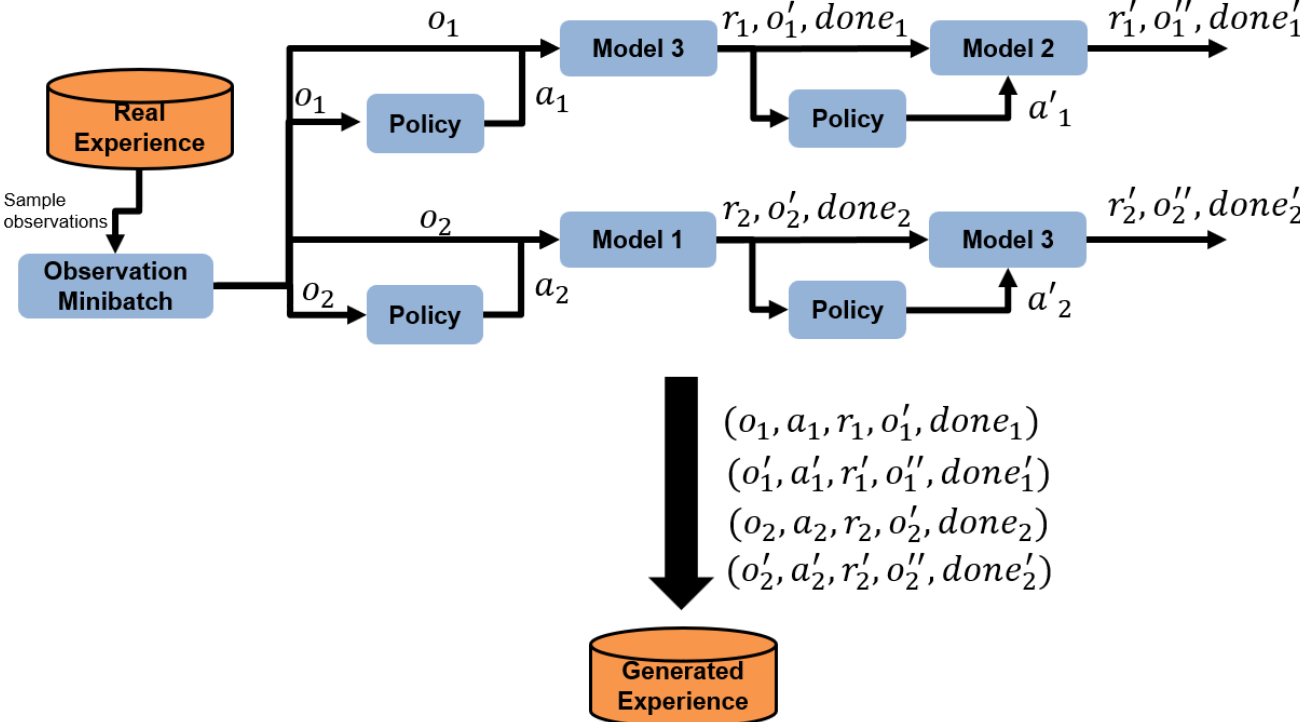
```
policy = rlMaxQPolicy(critic);
```

### Create Transition Models

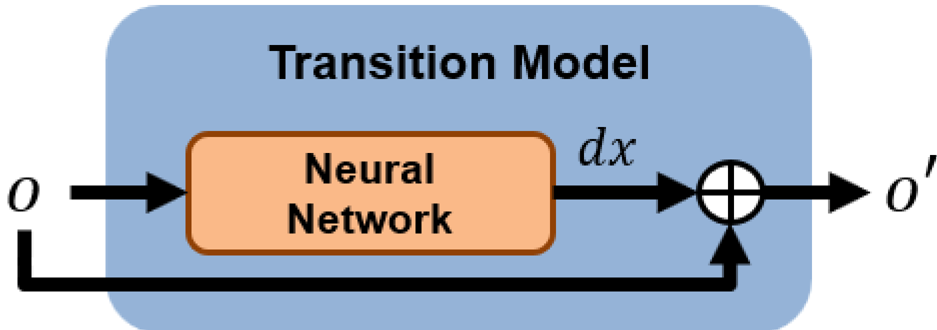
Model-based reinforcement learning uses transition models of the environment. The model usually consists of a transition function, a reward function, and a terminal state function.

- The transition function predicts the next observation given the current observation and the action.
- The reward function predicts the reward given the current observation, the action, and the next observation.
- The terminal state function predicts the terminal state given the observation.

As shown in the following figure, this example uses three transition functions as an ensemble of transition models to generate samples without interacting with the environment. The true reward function and the true terminal state function are given in this example.




Define three neural networks for transition models. The neural network predicts the difference between the next observation and the current observation




```
numModels = 3;
transitionNetwork1 = ...
   createTransitionNetwork(numObservations, numContinuousActions);

transitionNetwork2 = ...
   createTransitionNetwork(numObservations, numContinuousActions);

transitionNetwork3 = ...
   createTransitionNetwork(numObservations, numContinuousActions);

transitionNetworkVector =   [transitionNetwork1, ...
                             transitionNetwork2, ...
                             transitionNetwork3];
```

### Create Experience Buffers

Create an experience buffer for storing agent experiences (observation, action, next observation, reward, and isDone).

```
myBuffer.bufferSize = 1e5;
myBuffer.bufferIndex = 0;
myBuffer.currentBufferLength = 0;
myBuffer.observation = zeros(numObservations,myBuffer.bufferSize);
myBuffer.nextObservation = ...
                       zeros(numObservations,myBuffer.bufferSize);
myBuffer.action = ...
    zeros(numContinuousActions,1,myBuffer.bufferSize);
myBuffer.reward = zeros(1,myBuffer.bufferSize);
myBuffer.isDone = zeros(1,myBuffer.bufferSize);
```

Create a model experience buffer for storing the experiences generated by the models.

```
myModelBuffer.bufferSize = 1e5;
myModelBuffer.bufferIndex = 0;
myModelBuffer.currentBufferLength = 0;
myModelBuffer.observation =...
    zeros(numObservations,myModelBuffer.bufferSize);
myModelBuffer.nextObservation =...
    zeros(numObservations,myModelBuffer.bufferSize);
myModelBuffer.action = ...
    zeros(numContinuousActions,myModelBuffer.bufferSize);
myModelBuffer.reward = zeros(1,myModelBuffer.bufferSize);
myModelBuffer.isDone = zeros(1,myModelBuffer.bufferSize);
```

### Configure Training

Configure the training to use the following options.

- Maximum number of training episodes — 250
- Maximum steps per training episode — 500
- Discount factor — 0.99
- Training termination condition — Average reward across 10 episodes reaches the value of 480

```
numEpisodes = 250;
maxStepsPerEpisode = 500;
discountFactor = 0.99;
aveWindowSize = 10;
trainingTerminationValue = 480;
```

Configure the model options.

- Train transition models only after 2000 samples are collected.
- Train the models using all experiences in the real experience buffer in each episode. Use a mini-batch size of 256.
- The models generate trajectories with a length of 2 at the beginning of each episode.
- The number of generated trajectories is `numGenerateSampleIteration` x `numModels` x `miniBatchSize` = 20 x 3 x 256 = 15360.
- Use the same epsilon-greedy parameters as the DQN agent, except for the minimum epsilon value.
- Use a minimum epsilon value of 0.1, which is higher than the value used for interacting with the environment. Doing so allows the model to generate more diverse data.

```
warmStartSamples = 2000;
numEpochs = 1;
miniBatchSize = 256;
horizonLength = 2;
epsilonMinModel = 0.1;
numGenerateSampleIteration = 20;
sampleGenerationOptions.horizonLength = horizonLength;
sampleGenerationOptions.numGenerateSampleIteration = ...
                             numGenerateSampleIteration;
sampleGenerationOptions.miniBatchSize = miniBatchSize;
sampleGenerationOptions.numObservations = numObservations;
sampleGenerationOptions.epsilonMinModel = epsilonMinModel;

% optimizer options
velocity1 = [];
velocity2 = [];
velocity3 = [];
decay = 0.01;
momentum = 0.9;
learnRate = 0.0005;
```

Configure the DQN training options.

- Use the epsilon greedy algorithm with an initial epsilon value is 1, a minimum value of 0.01, and a decay rate of 0.005.
- Update the target network every 4 steps.

- Set the ratio of the real experiences to generated experiences to 0.2:0.8 by setting `RealRatio` to 0.2. Setting `RealRatio` to 1.0 is the same as the model-free DQN.
- Take 5 gradient steps at each environment step.

```
epsilon = 1;
epsilonMin = 0.01;
epsilonDecay = 0.005;
targetUpdateFrequency = 4;
realRatio = 0.2; % Set to 1 to run a standard DQN
numGradientSteps = 5;
```

Create a vector for storing the cumulative reward for each training episode.

```
episodeCumulativeRewardVector = [];
```

Create a figure for model training visualization using the `hBuildFigureModel` helper function.

```
[trainingPlotModel, ...
    lineLossTrain1, ...
    lineLossTrain2, ...
    lineLossTrain3, ...
    axModel] = hBuildFigureModel();
```

Create a figure for model validation visualization using the `hBuildFigureModelTest` helper function.

```
[testPlotModel, lineLossTest1, axModelTest] ...
                                 = hBuildFigureModelTest();
```

Create a figure for DQN agent training visualization using the `hBuildFigure` helper function.

```
[trainingPlot,lineReward,lineAveReward, ax] = hBuildFigure;
```

#### Train Agent

Train the agent using a custom training loop. The training loop uses the following algorithm. For each episode:

1. Train the transition models.
2. Generate experiences using the transition models and store the samples in the model experience buffer.
3. Generate a real experience. To do so, generate an action using the policy, apply the action to the environment, and obtain the resulting observation, reward, and is-done values.
4. Create a mini-batch by sampling experiences from both the experience buffer and the model experience buffer.
5. Compute the target Q value.
6. Compute the gradient of the loss function with respect to the critic representation parameters.
7. Update the critic representation using the computed gradients.
8. Update the training visualization.
9. Terminate training if the critic is sufficiently trained.

Training the policy is a computationally intensive process. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the policy yourself, set `doTraining` to `true`.

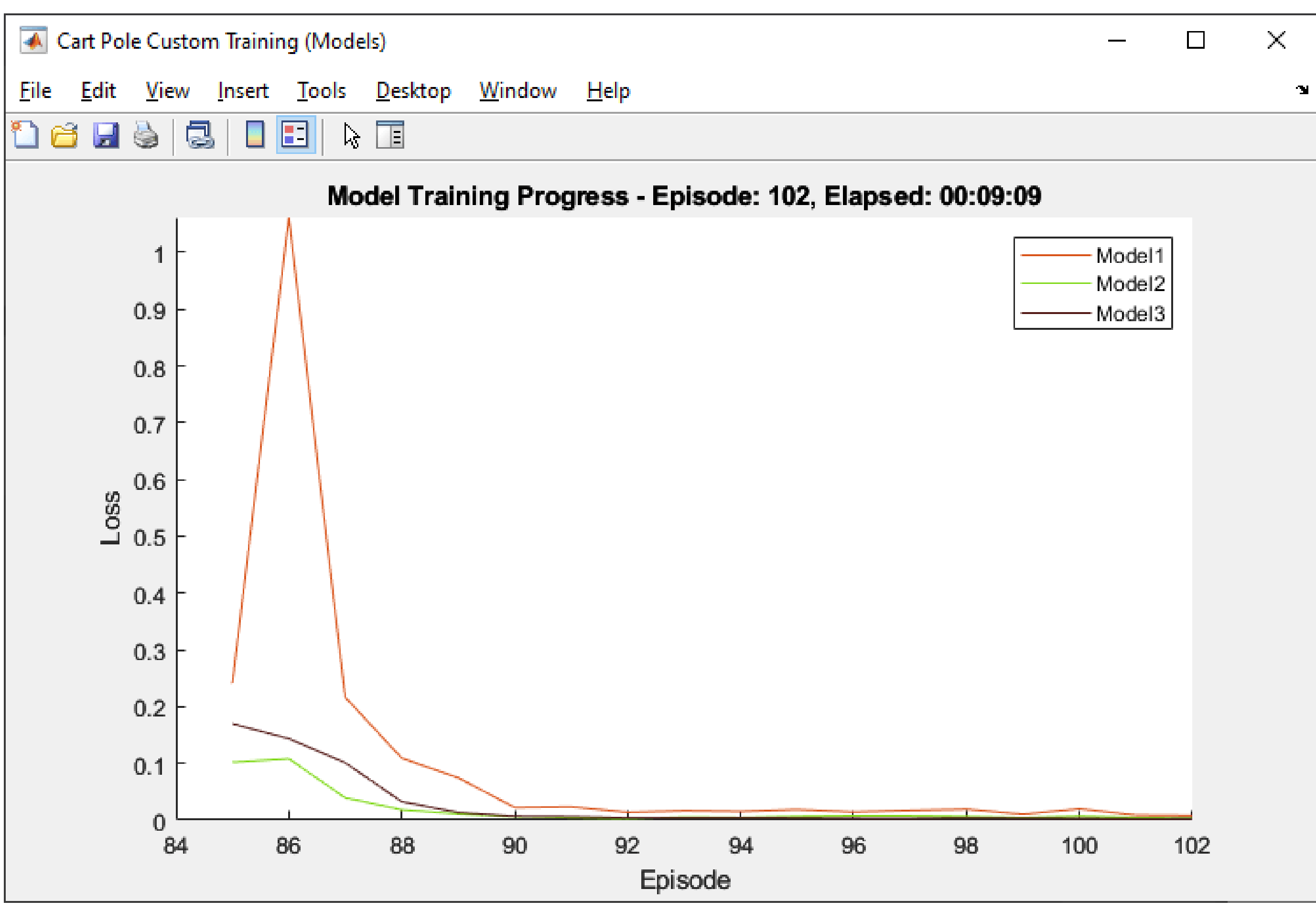
```
doTrianing = false;
if doTrianing
    targetCritic = critic;
    modelTrainedAtleastOnce = false;
    totalStepCt = 0;
    start = tic;

    set(trainingPlotModel,Visible = "on");
    set(testPlotModel,Visible = "on");
    set(trainingPlot,Visible = "on");

    for episodeCt = 1:numEpisodes
        if myBuffer.currentBufferLength > miniBatchSize && ...
                totalStepCt > warmStartSamples
            if realRatio < 1.0
                %---------------------------------------------
                % 1. Train transition models.
                %---------------------------------------------
                % Training three transition models
                [transitionNetworkVector(1),loss1,velocity1] = ...
                        trainTransitionModel(...
                        transitionNetworkVector(1),...
                        myBuffer,velocity1,miniBatchSize,...
                        numEpochs,momentum,learnRate);
                [transitionNetworkVector(2),loss2,velocity2] = ...
                        trainTransitionModel(...
                        transitionNetworkVector(2),...
                        myBuffer,velocity2,miniBatchSize,...
                        numEpochs,momentum,learnRate);
                [transitionNetworkVector(3),loss3,velocity3] = ...
                        trainTransitionModel(...
                        transitionNetworkVector(3),...
                        myBuffer,velocity3,miniBatchSize,...
                        numEpochs,momentum,learnRate);
                modelTrainedAtleastOnce = true;

                % Display the training progress
                d = duration(0,0,toc(start),"Format",'hh:mm:ss');
                addpoints(lineLossTrain1,episodeCt,loss1)
                addpoints(lineLossTrain2,episodeCt,loss2)
                addpoints(lineLossTrain3,episodeCt,loss3)
                legend(axModel,"Model1","Model2","Model3");
                title(axModel, ...
                    "Model Training Progress - Episode: "...
                    + episodeCt + ", Elapsed: " + string(d))
                drawnow

                %---------------------------------------------
                % 2. Generate experience using models.
                %---------------------------------------------
                % Create numGenerateSampleIteration x
                % horizonLength xnumModels x miniBatchSize
                % ex) 20 x 2 x 3 x 256 = 30720 samples
                myModelBuffer = generateSamples(myBuffer,...
                    myModelBuffer,...
                    transitionNetworkVector,policy,actInfo,...
                    epsilon,sampleGenerationOptions);
            end
```

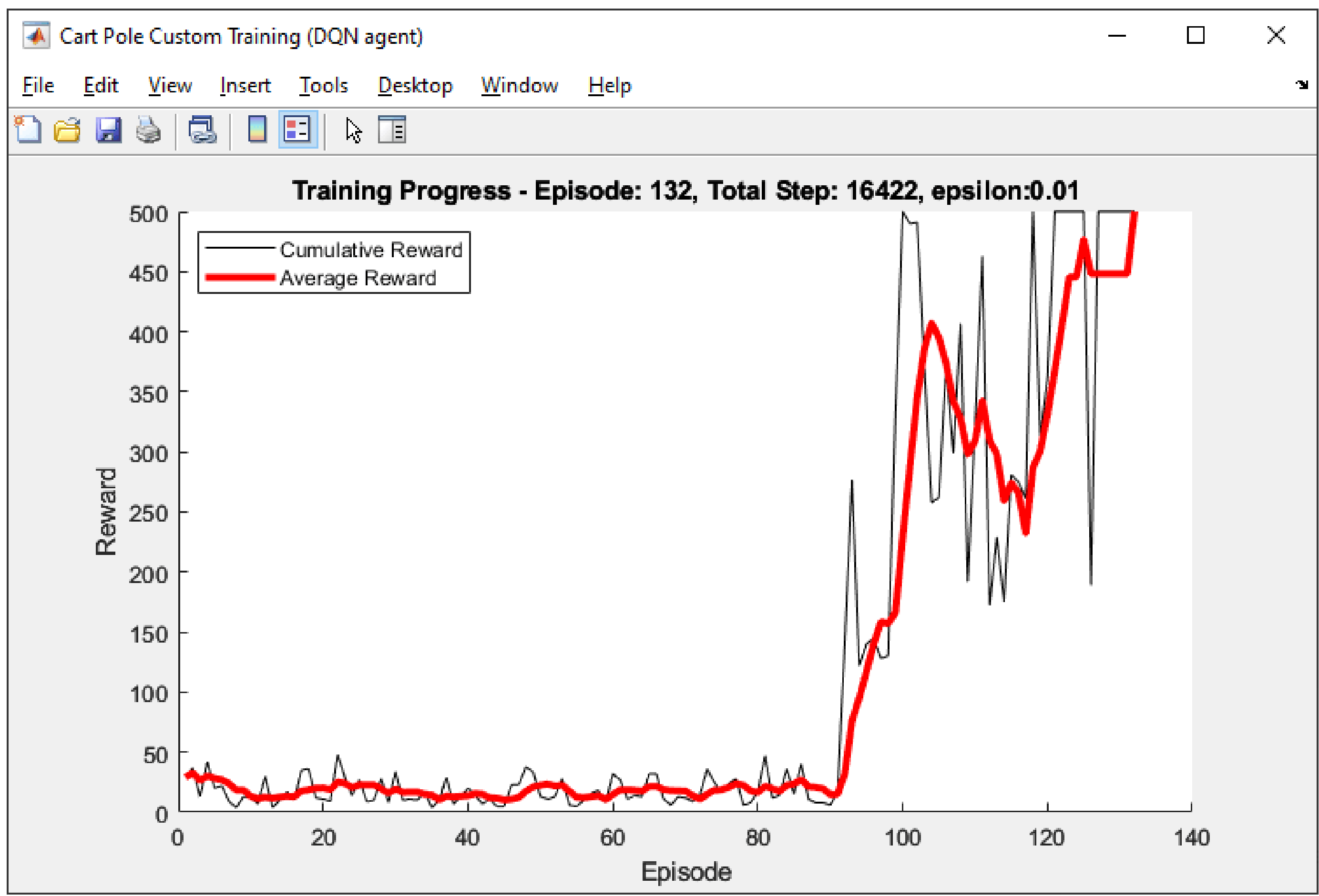
```
end

%-----------------------------------------------
% Interact with environment and train agent.
%-----------------------------------------------
% Reset the environment at the start of the episode
observation = reset(env);
episodeReward = zeros(maxStepsPerEpisode,1);
errorPreddiction = zeros(maxStepsPerEpisode,1);

for stepCt = 1:maxStepsPerEpisode
    %-----------------------------------------------
    % 3. Generate an experience.
    %-----------------------------------------------
    totalStepCt = totalStepCt + 1;

    % Compute an action using the policy based on
    % the current observation.
    if rand() < epsilon
        action = actInfo.usample;
    else
        action = getAction(policy,{observation});
    end
    action = action{1};
    % Udpate epsilon
    if totalStepCt > warmStartSamples
        epsilon = max(epsilon*(1-epsilonDecay),...
                      epsilonMin);
    end

    % Apply the action to the environment and obtain the
    % resulting observation and reward.
    [nextObservation,reward,isDone] = step(env,action);

    % Check prediction
    dx = predict(transitionNetworkVector(1),...
        dlarray(observation,"CB"),dlarray(action,"CB"));
    predictedNextObservation = observation + dx;
    errorPreddiction(stepCt) = ...
        sqrt(sum((nextObservation - ...
                    predictedNextObservation).^2));

    % Store the action, observation, reward and is-done
    % experience
    myBuffer = storeExperience(myBuffer,...
                    observation,...
                    action,...
                    nextObservation,reward,isDone);

    episodeReward(stepCt) = reward;
    observation = nextObservation;

    % Train DQN agent
    for gradientCt = 1:numGradientSteps
        if myBuffer.currentBufferLength >= miniBatchSize ...
                && totalStepCt>warmStartSamples
            %-----------------------------------------------
            % 4. Sample minibatch from experience buffers.
```

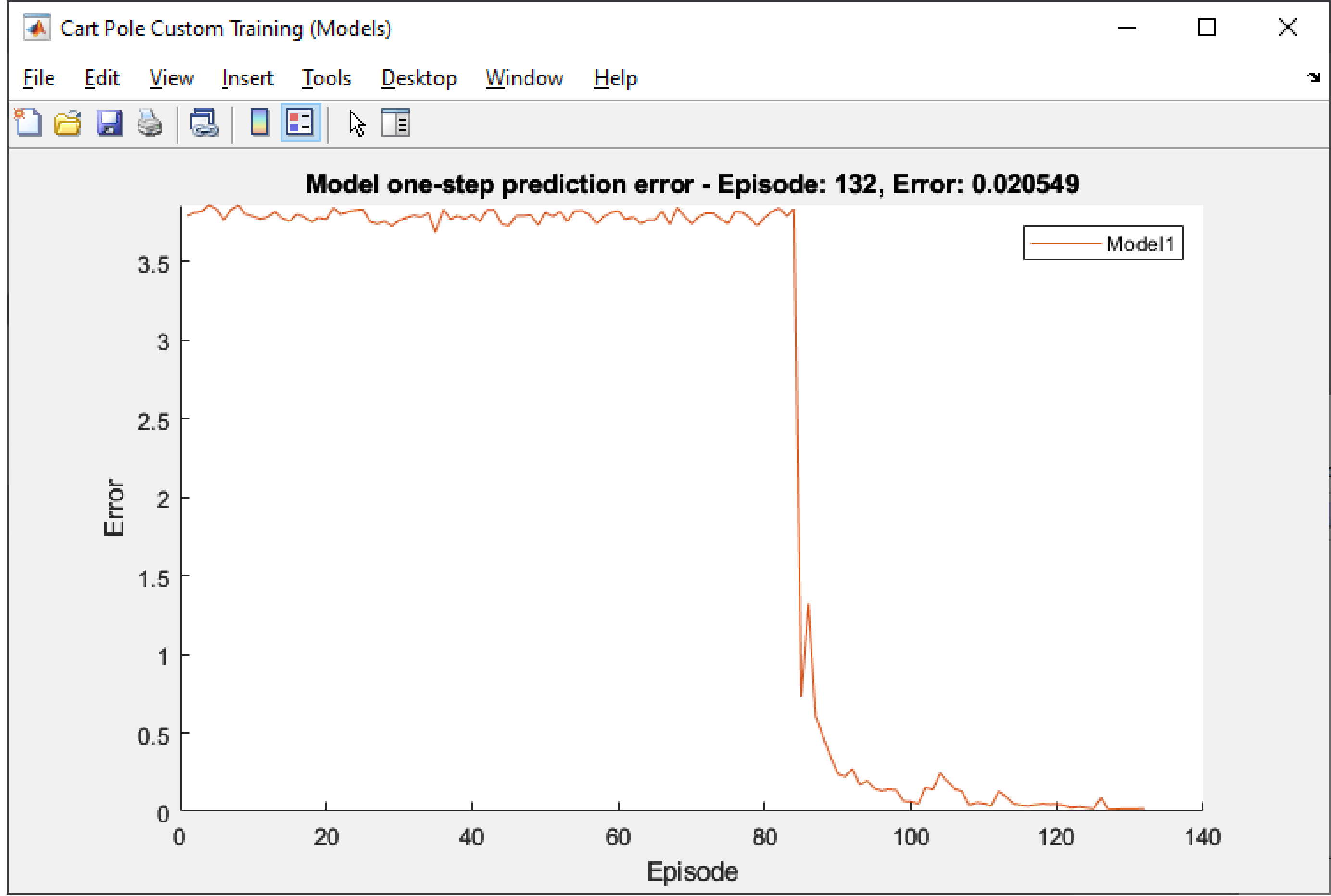
```
        %-----------------------------------------------
        [sampledObservation,...
            sampledAction,...
            sampledNextObservation,...
            sampledReward,...
            sampledIsdone] ...
                  = sampleMinibatch(...
                        modelTrainedAtleastOnce,...
                        realRatio,...
                        miniBatchSize,...
                        myBuffer,myModelBuffer);

        %-----------------------------------------------
        % 5. Compute target Q value.
        %-----------------------------------------------
        % Compute target Q value
        [targetQValues, MaxActionIndices] = ...
                getMaxQValue(targetCritic, ...
                {reshape(sampledNextObservation,...
                [numObservations,1,miniBatchSize])});

        % Compute target for nonterminal states
        targetQValues(~logical(sampledIsdone)) = ...
            sampledReward(~logical(sampledIsdone)) + ...
            discountFactor.*...
            targetQValues(~logical(sampledIsdone));
        % Compute target for terminal states
        targetQValues(logical(sampledIsdone)) = ...
            sampledReward(logical(sampledIsdone));

        lossData.batchSize = miniBatchSize;
        lossData.actInfo = actInfo;
        lossData.actionBatch = sampledAction;
        lossData.targetQValues = targetQValues;

        %-----------------------------------------------
        % 6. Compute gradients.
        %-----------------------------------------------
        criticGradient = ...
            gradient(critic,...
                 @criticLossFunction, ...
                {reshape(sampledObservation,...
                [numObservations,1,miniBatchSize])},...
                lossData);

        %-----------------------------------------------
        % 7. Update the critic network using gradients.
        %-----------------------------------------------
        [critic, criticOptimizer] = update(...
            criticOptimizer, critic,...
            criticGradient);

        % Update the policy parameters using the critic
        % parameters.
        policy = setLearnableParameters(...
                        policy,...
                        getLearnableParameters(critic));
    end
```

```
    end
    % Update target critic periodically
    if mod(totalStepCt, targetUpdateFrequency)==0
        targetCritic = critic;
    end

    % Stop if a terminal condition is reached.
    if isDone
        break;
    end
end % End of episode

%---------------------------------------------------------
% 8. Update the training visualization.
%---------------------------------------------------------
episodeCumulativeReward = sum(episodeReward);
episodeCumulativeRewardVector = cat(2,...
    episodeCumulativeRewardVector,episodeCumulativeReward);
movingAveReward = movmean(episodeCumulativeRewardVector,...
    aveWindowSize,2);
addpoints(lineReward,episodeCt,episodeCumulativeReward);
addpoints(lineAveReward,episodeCt,movingAveReward(end));
title(ax, "Training Progress - Episode: " + episodeCt + ...
    ", Total Step: " + string(totalStepCt) + ...
    ", epsilon:" + string(epsilon))
drawnow;

errorPreddiction = errorPreddiction(1:stepCt);

% Display one step prediction error.
addpoints(lineLossTest1,episodeCt,mean(errorPreddiction))
legend(axModelTest,"Model1");
title(axModelTest, ...
    "Model one-step prediction error - Episode: " + ...
    episodeCt + ", Error: " + ...
    string(mean(errorPreddiction)))
drawnow

% Display training progress every 10th episode
if (mod(episodeCt,10) == 0)
    fprintf("EP:%d, Reward:%.4f, AveReward:%.4f, " + ...
            "Steps:%d, TotalSteps:%d, epsilon:%f," + ...
            "error model:%f\n",...
                episodeCt, ...
                episodeCumulativeReward,...
                movingAveReward(end),...
                stepCt,totalStepCt,...
                epsilon,...
                mean(errorPreddiction))
end

%---------------------------------------------------------
% 9. Terminate training
%    if the network is sufficiently trained.
%---------------------------------------------------------
if max(movingAveReward) > trainingTerminationValue
    break
end
```

```
        end
    else
        load("cartPoleModelBasedCustomLoopPolicy.mat");
    end
```




Cart Pole Custom Training (DQN agent)
File Edit View Insert Tools Desktop Window Help
Training Progress - Episode: 132, Total Step: 16422, epsilon:0.01
Cumulative Reward
Average Reward
Reward
Episode
0
50
100
150
200
250
300
350
400
450
500
0
20
40
60
80
100
120
140




### Simulate Agent

To simulate the trained agent, first reset the environment.

```
obs0 = reset(env);
obs = obs0;
```

Enable the environment visualization, which is updated each time the environment step function is called.

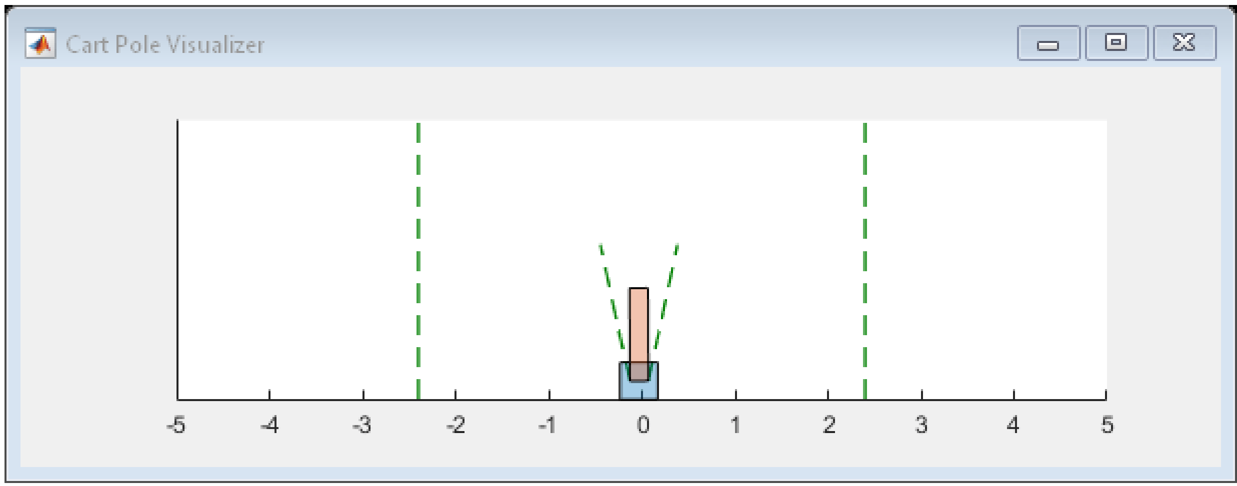
```
plot(env)
```

For each simulation step, perform the following actions.

1. Get the action by sampling from the policy using the `getAction` function.
2. Step the environment using the obtained action value.
3. Terminate if a terminal condition is reached.

```
actionVector = zeros(1,maxStepsPerEpisode);
obsVector = zeros(numObservations,maxStepsPerEpisode+1);
obsVector(:,1) = obs0;
for stepCt = 1:maxStepsPerEpisode

    % Select action according to trained policy.
```

```
    action = getAction(policy,{obs});
    action= action{1};

    % Step the environment.
    [nextObs,reward,isDone] = step(env,action);

    obsVector(:,stepCt+1) = nextObs;
    actionVector(1,stepCt) = action;

    % Check for terminal condition.
    if isDone
        break
    end

    obs = nextObs;
end
```



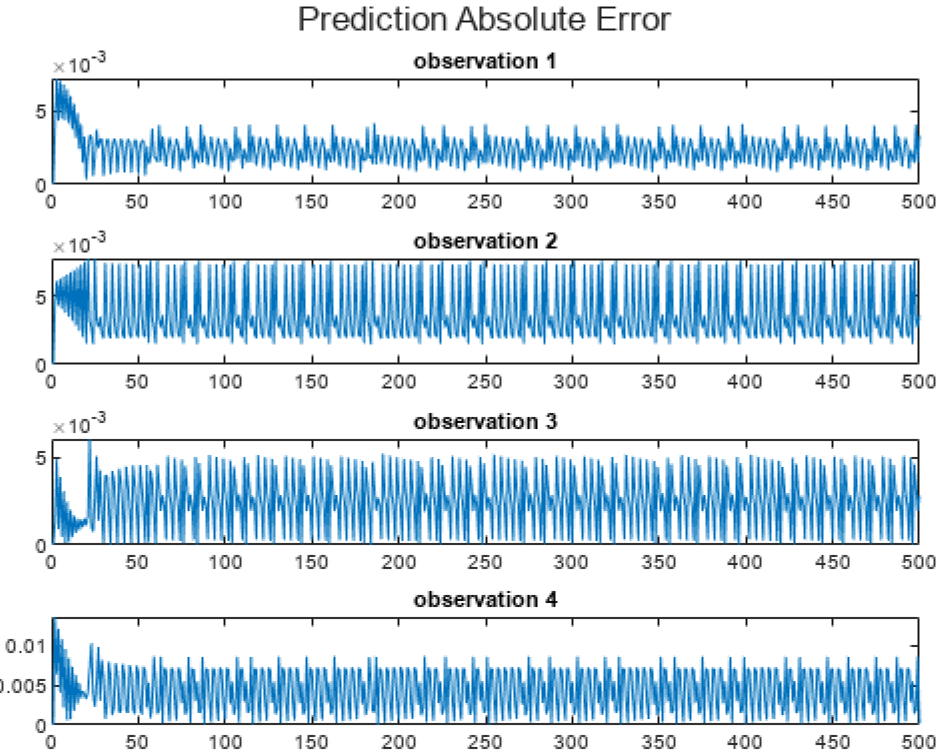

```
lastStepCt = stepCt;
```

### Test Model

Test one of the models by predicting a next observation given a current observation and an action.

```
modelID = 3;
predictedObsVector = zeros(numObservations,lastStepCt);
obs = dlarray(obsVector(:,1),"CB");
predictedObsVector(:,1) = obs;
for stepCt = 1:lastStepCt
    obs = dlarray(obsVector(:,stepCt),"CB");
    action = dlarray(actionVector(1,stepCt),"CB");

    dx = predict(transitionNetworkVector(modelID),obs, action);
    predictedObs = obs + dx;
    predictedObsVector(:,stepCt+1) = predictedObs;
end
predictedObsVector = predictedObsVector(:, 1:lastStepCt);
figure(5)
layOut = tiledlayout(4,1, "TileSpacing", "compact");
for i = 1:4
    nexttile;
    errorPrediction = abs(predictedObsVector(i,1:lastStepCt) - ...
```

```
                                                obsVector(i,1:lastStepCt));
    line1 = plot(errorPrediction,"DisplayName", "Absolute Error");
    title("observation "+num2str(i));
end
title(layOut,"Prediction Absolute Error")
```




The small absolute prediction error shows that the model is successfully trained to predict the next observation.

**References**


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

[2] Janner, Michael, Justin Fu, Marvin Zhang, and Sergey Levine. "When to trust your model: Model-based policy optimization." *ArXiv:1907.08253 [Cs, Stat]*, November 5, 2019. https://arxiv.org/abs/1906.08253.


## See Also

**Functions**
`rlPredefinedEnv` | `rlOptimizer` | `gradient`

**Objects**
`rlQValueFunction` | `rlOptimizerOptions`

## Related Examples


- "Train Reinforcement Learning Agents" on page 5-3
- "Create Agent for Custom Reinforcement Learning Algorithm" on page 5-456
- "Train MBPO Agent to Balance Cart-Pole System" on page 5-472


## More About


- "Load Predefined Control System Environments" on page 2-23
- "Train Reinforcement Learning Agents" on page 5-3
- "Model-Based Policy Optimization (MBPO) Agents" on page 3-62
- "Create Custom Reinforcement Learning Agents" on page 3-68

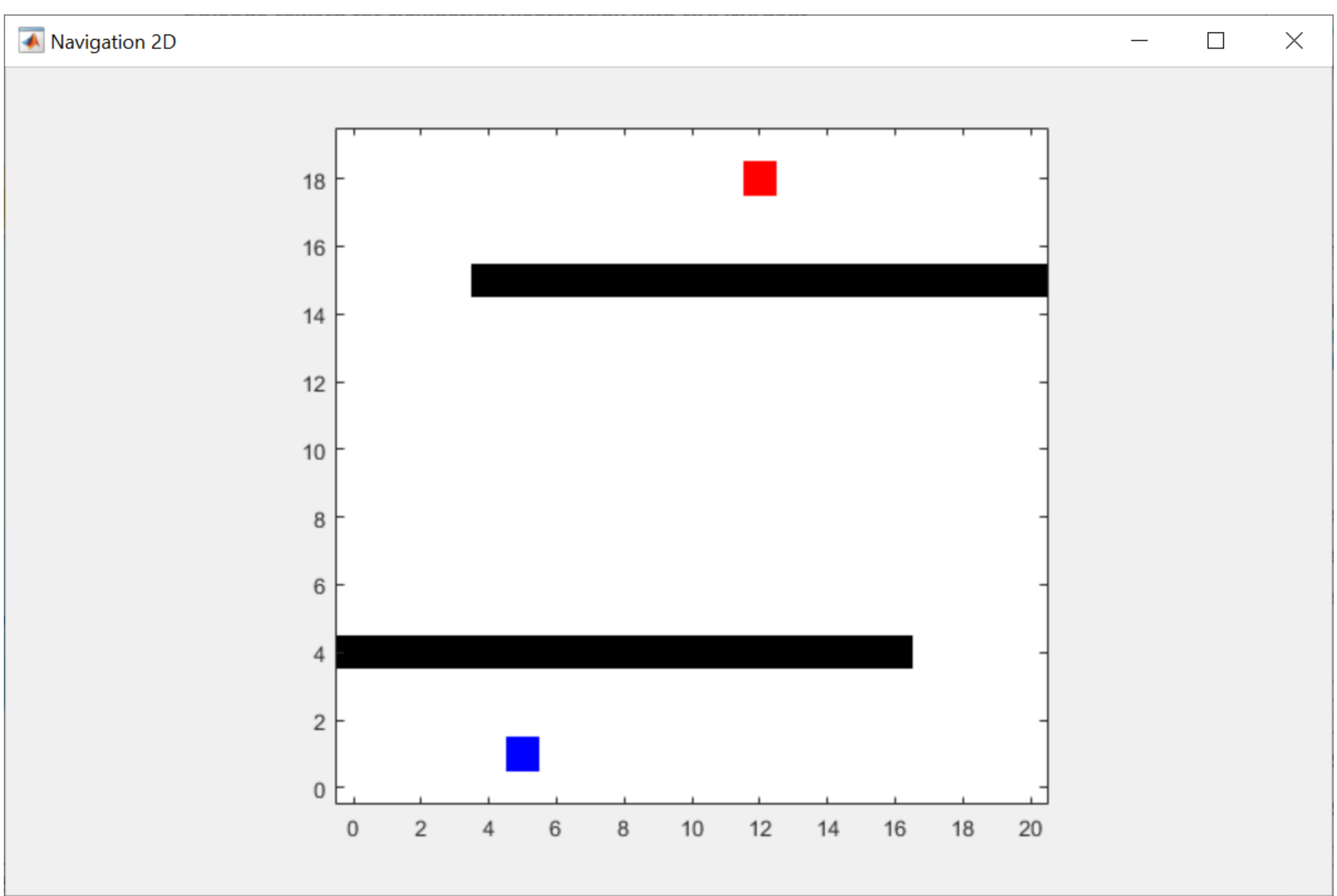
# Train DQN Agent Using Hindsight Experience Replay

This example shows how to train a deep Q-learning network (DQN) agent using hindsight experience replay in a discrete navigation problem.

### Discrete Navigation Environment

The reinforcement learning environment for this example is a simple discrete 2D navigation problem. The training goal is to make a robot reach the goal state.

The environment is a 20-by-20 grid containing a robot (blue), a goal (red), and obstacles (black).




The navigation problem in this example is a goal-conditioned task for which the observation from the environment contains both the goal (navigation target) and the goal measurement (robot position). The observation column vector contains the robot position $(x_{\text{robot}}, y_{\text{robot}})$ and the navigation target $(x_{\text{goal}}, y_{\text{goal}})$.

$$O = \left[x_{\text{robot}}, y_{\text{robot}}, x_{\text{goal}}, y_{\text{goal}}\right]^T$$

The initial position of the robot and the goal are sampled as follows:

$$x_{\text{robot}} \sim U\{0, 19\}$$
$$y_{\text{robot}} \sim U\{0, 3\}$$
$$x_{\text{goal}} \sim U\{0, 19\}$$
$$y_{\text{goal}} \sim U\{16, 19\}$$

where $U\{a, b\}$ is a discrete uniform distribution between $a$ and $b$.

The discrete actions for this environment are defined as follows:

$$A = \begin{cases} 1 & y_{\text{robot}} = y_{\text{robot}} + 1 \quad \text{(Go up)} \\ 2 & y_{\text{robot}} = y_{\text{robot}} - 1 \quad \text{(Go down)} \\ 3 & x_{\text{robot}} = x_{\text{robot}} - 1 \quad \text{(Go left)} \\ 4 & x_{\text{robot}} = x_{\text{robot}} + 1 \quad \text{(Go right)} \end{cases}$$

If the action leads to the obstacle location or the map's boundary, the robot doesn't move.

The reward signal is defined as follows:

$$R = \begin{cases} 2 & \text{if } x_{\text{robot}} = x_{\text{goal}} \text{ and } y_{\text{robot}} = y_{\text{goal}} \\ -0.01 - 0.01 I(O, O') & \text{otherwise} \end{cases}$$

where $O'$ is the next observation, and

$$I(O, O') = \begin{cases} 1 & \text{if the robot doesn't move due to obstacles or boundaries} . \\ 0 & \text{otherwise} \end{cases}$$

Terminal condition is defined as follows:

$$\text{IsDone} = \begin{cases} 1 & \text{if } x_{\text{robot}} = x_{\text{goal}} \text{ and } y_{\text{robot}} = y_{\text{goal}} \\ 0 & \text{otherwise} \end{cases}$$

**Create Environment Interface**

Fix the random generator seed for reproducibility.

```
rng(1)
```

Create a discrete navigation environment. `NavigationDiscreteEnv.m` is located in this example folder.

```
env = NavigationDiscreteEnv;
```

**Create DQN Agent**

For a given observation and action, a DQN agent estimates the value of the policy (the cumulative discounted long-term reward) using a parametrized Q-value function critic.

Create a default DQN agent from the environment specifications. For more information on DQN agents, see `rlDQNAgent`.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
agent = rlDQNAgent(obsInfo,actInfo);
```

Specify the DQN agent options, including training options for the critic. Alternatively, you can use `rlDQNAgentOptions` object.

```
agent.AgentOptions.EpsilonGreedyExploration.EpsilonDecay = 1e-5;
agent.AgentOptions.EpsilonGreedyExploration.Epsilon = 0.2;
agent.AgentOptions.MiniBatchSize = 128;
agent.AgentOptions.CriticOptimizerOptions.LearnRate = 5e-4;
agent.AgentOptions.CriticOptimizerOptions.GradientThreshold = 1;
```

#### Construct Hindsight Replay Memory

The DQN agent is unable to efficiently learn a good policy with a default replay memory because of the sparse rewards. You can improve the sample efficiency by using a hindsight replay memory for a goal-conditioned task with sparse rewards. The hindsight experience replay augments the acquired experiences by replacing the goal with the goal measurement so that agent can use the data that reaches the replaced goal. Thus, the agent can be trained with meaningful rewards even if the agent does not reach the goal.

To use a hindsight replay memory, set `ExperienceBuffer` of the agent to `rlHindsightReplayMemory`. You need to specify the following.

- A reward function: The reward function, `myNavigationGoalRewardFcn`, computes the true reward given observation, action, and next observation.
- An is-done function: The is-done function, `myNavigationGoalIsDoneFcn`, computes the true is-done signal given observation, action, and the next observation.
- Goal condition information: `goalConditionInfo` specifies the channel and indices of goal measurements and goals. In this example, there is only one observation channel; the first and second observation elements correspond to the goal measurements (the robot position), and the third and fourth observation elements correspond to the goal position.
- Experience buffer length (optional): This examples uses 1e5.

`myNavigationGoalRewardFcn.m` and `myNavigationGoalIsDoneFcn.m` are located in this example folder. For more information, type `help rlHindsightReplayMemory.`

To use a standard replay memory, set `useHER` to `false.`

```
useHER = true;
if useHER
    rewardFcn = @myNavigationGoalRewardFcn;
    isDoneFcn = @myNavigationGoalIsDoneFcn;
    % observation is [robot_x, robot_y, goal_x, goal_y]'
    % goal measurement: channel --- 1, indices --- 1,2
    % goal: channel --- 1, indices --- 3,4
    goalConditionInfo = {{1,[1,2], 1, [3,4]}};
    bufferLength = 1e5;
    agent.ExperienceBuffer = rlHindsightReplayMemory(obsInfo,actInfo,...
        rewardFcn,isDoneFcn,goalConditionInfo,bufferLength);
else
    agent.AgentOptions.ExperienceBufferLength = 1e5;
end
```

#### Train Agent

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

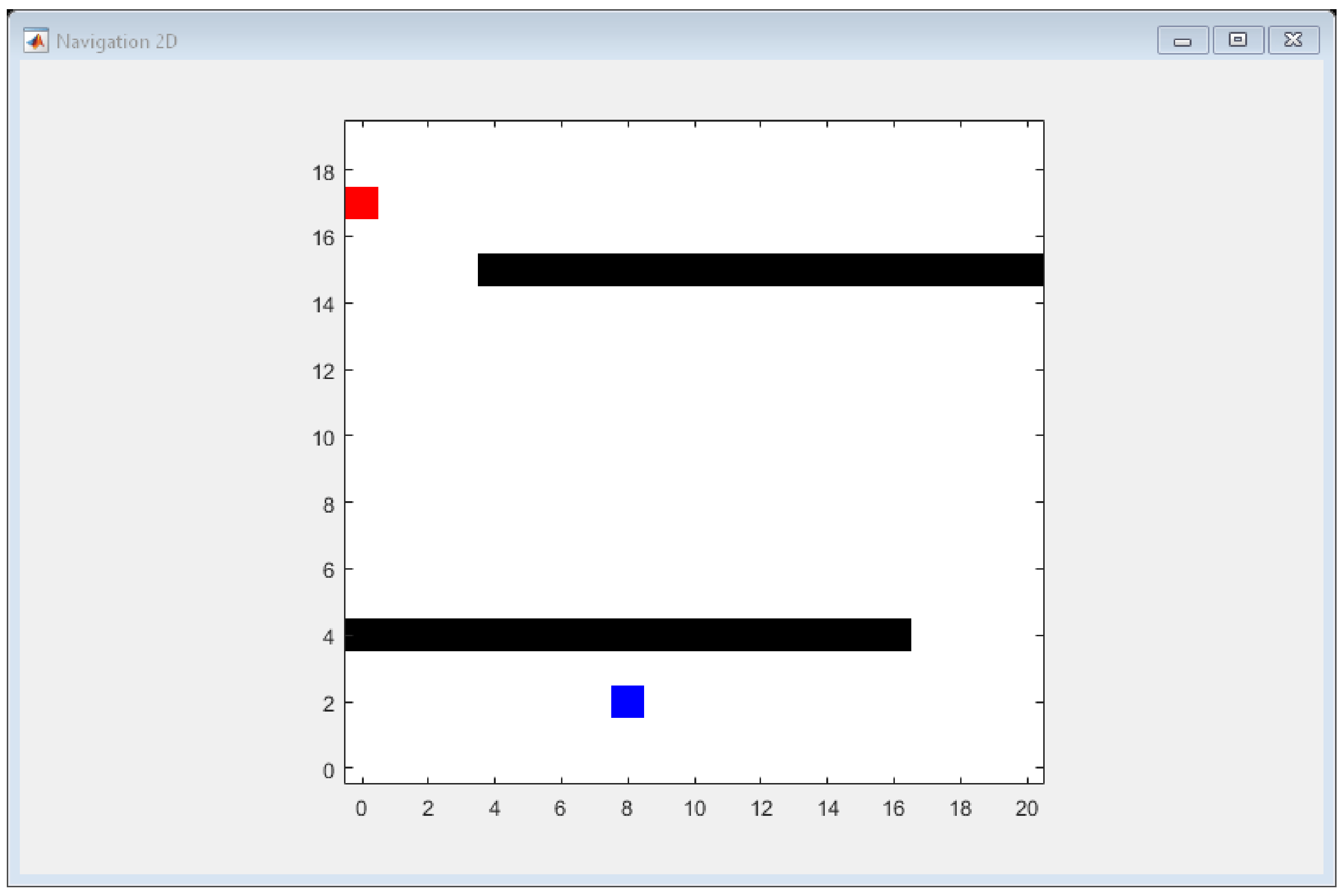
- Run one training session containing 3000 episodes, with each episode lasting at most 120 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Set the Plots options to "training-progress"

For more information, see `rlTrainingOptions`.

```
maxEpisodes = 3000;
maxStepsPerEpisode = 120;
trainOpts = rlTrainingOptions(...
    MaxEpisodes=maxEpisodes, ...
    MaxStepsPerEpisode=maxStepsPerEpisode, ...
    Verbose=false, ...
    ScoreAveragingWindowLength=100,...
    Plots="training-progress",...
    StopTrainingCriteria="EpisodeCount",...
    StopTrainingValue=maxEpisodes);
```

You can visualize the environment by using the `plot` function during training or simulation.

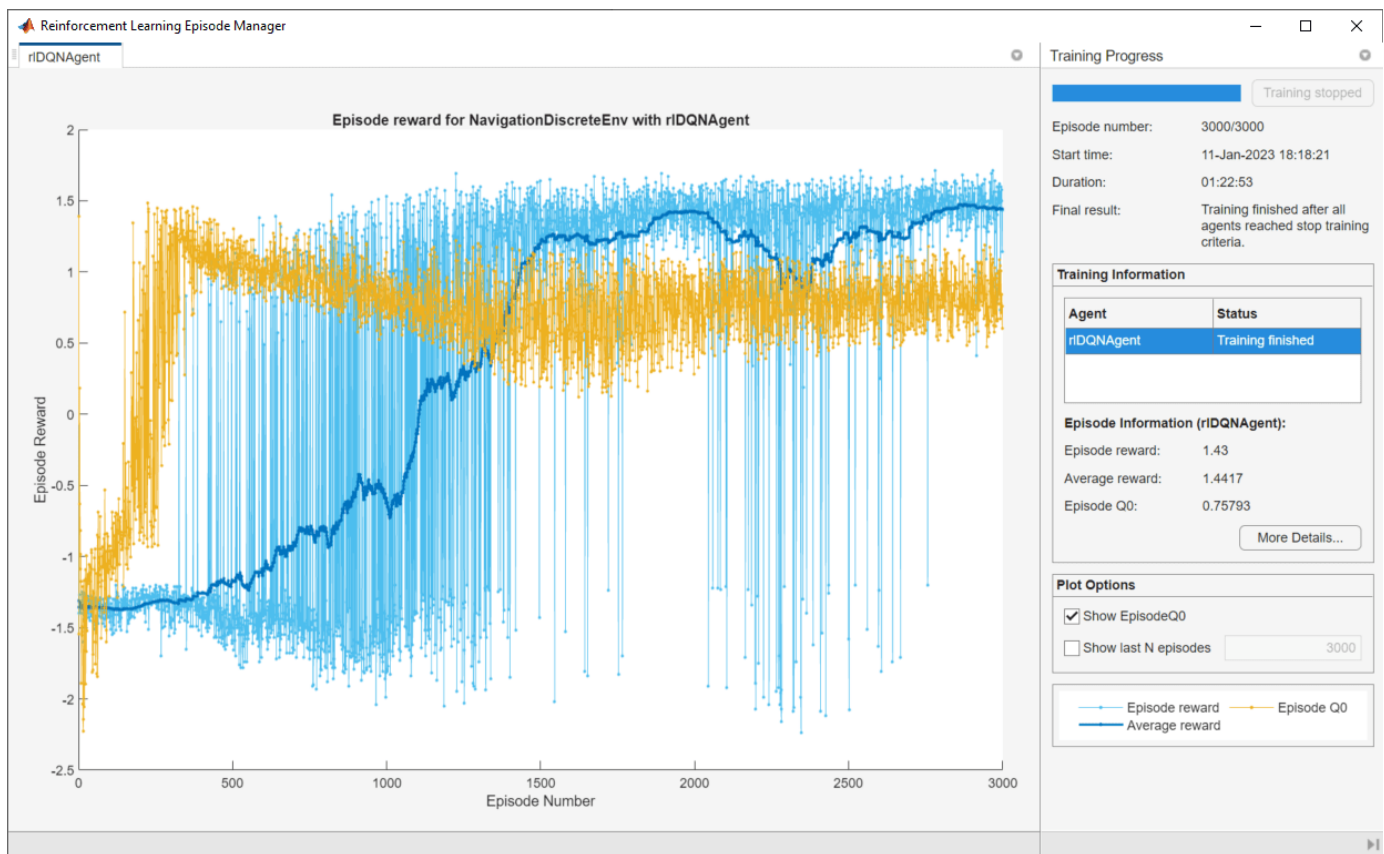
```
plot(env)
```



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

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

The following figure shows a snapshot of training progress.




### Simulate DQN Agent

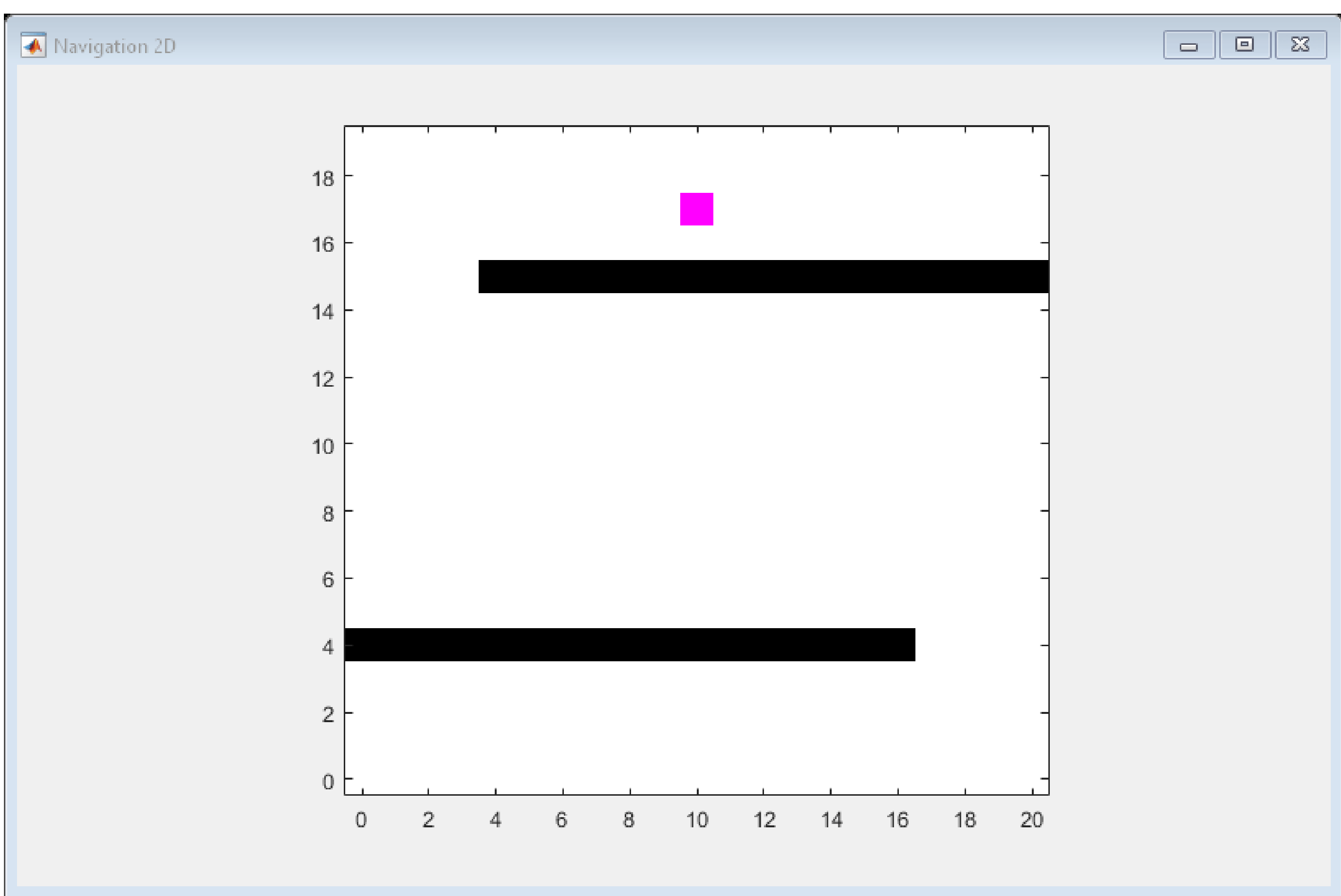
Since the reset function randomizes the reference values, fix the random generator seed to ensure simulation reproducibility.

```
rng(2)
```

To validate the performance of the trained agent, simulate it within the navigation environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
simOptions = rlSimulationOptions(...
    NumSimulations=1,...
```

```
    MaxSteps=maxStepsPerEpisode);
simResults = sim(env,agent,simOptions);
```




The trained agent successfully reached the goal.

### Reference

Andrychowicz, Marcin, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. "Hindsight experience replay." *Advances in neural information processing systems* 30 (2017).

## See Also

**Objects**
`rlHindsightReplayMemory` | `rlHindsightPrioritizedReplayMemory`

6

# Deploy Trained Policies

# Deploy Trained Reinforcement Learning Policies

Once you train a reinforcement learning agent, you can generate code to deploy the optimal policy. You can generate:

- CUDA code for deep neural network policies using GPU Coder
- C/C++ code for table, deep neural network, or linear basis function policies using MATLAB Coder

Code generation is supported for agents using feedforward neural networks in any of the input paths, provided that all the used layers are supported. Code generation is not supported for continuous actions PG, AC, PPO, and SAC agents using a recurrent neural network (RNN).

For more information on training reinforcement learning agents, see “Train Reinforcement Learning Agents” on page 5-3.

To generate a policy evaluation function that selects an action based on a given observation, use `generatePolicyFunction`. You can generate code to deploy this policy function using GPU Coder or MATLAB Coder.

To generate a Simulink policy evaluation block that selects an action based on a given observation, use `generatePolicyBlock`. You can generate code to deploy this policy block using Simulink Coder.

## Generate Code Using GPU Coder

If your trained optimal policy uses a deep neural network, you can generate CUDA code for the policy using GPU Coder. For more information on supported GPUs see “GPU Computing Requirements” (Parallel Computing Toolbox). There are several required and recommended prerequisite products for generating CUDA code for deep neural networks. For more information, see “Installing Prerequisite Products” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Not all deep neural network layers support GPU code generation. For a list of supported layers, see “Supported Networks, Layers, and Classes” (GPU Coder). For more information and examples on GPU code generation, see “Deep Learning with GPU Coder” (GPU Coder).

### Generate CUDA Code for Deep Neural Network Policy

As an example, generate GPU code for the policy gradient agent trained in “Train PG Agent to Balance Cart-Pole System” on page 5-57.

Load the trained agent.

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

Create a policy evaluation function for this agent.

```
generatePolicyFunction(agent)
```

This command creates the `evaluatePolicy.m` file, which contains the policy function, and the `agentData.mat` file, which contains the trained deep neural network actor. For a given observation, the policy function evaluates a probability for each potential action using the actor network. Then, the policy function randomly selects an action based on these probabilities.

You can generate code for this network using GPU Coder. For example, you can generate a CUDA compatible MEX function.

Configure the `codegen` function to create a CUDA compatible C++ MEX function.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
```

Set an example input value for the policy evaluation function. To find the observation dimension, use the `getObservationInfo` function. In this case, the observations are in a four-element vector.

```
argstr = '{ones(4,1)}';
```

Generate code using the `codegen` function.

```
codegen('-config','cfg','evaluatePolicy','-args',argstr,'-report');
```

This command generates the MEX function `evaluatePolicy_mex`.

## Generate Code Using MATLAB Coder

You can generate C/C++ code for table, deep neural network, or linear basis function policies using MATLAB Coder.

Using MATLAB Coder, you can generate:

- C/C++ code for policies that use Q tables, value tables, or linear basis functions. For more information on general C/C++ code generation, see “Generating Code” (MATLAB Coder).
- C++ code for policies that use deep neural networks. Note that code generation is not supported for continuous actions PG, AC, PPO, and SAC agents using a recurrent neural network (RNN). For a list of supported layers, see “Networks and Layers Supported for Code Generation” (MATLAB Coder). For more information, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder) and “Deep Learning with MATLAB Coder” (MATLAB Coder).

### Generate C Code for Deep Neural Network Policy without using any Third-Party Library

As an example, generate C code without dependencies on third-party libraries for the policy gradient agent trained in “Train PG Agent to Balance Cart-Pole System” on page 5-57.

Load the trained agent.

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

Create a policy evaluation function for this agent.

```
generatePolicyFunction(agent)
```

This command creates the `evaluatePolicy.m` file, which contains the policy function, and the `agentData.mat` file, which contains the trained deep neural network actor. For a given observation, the policy function evaluates a probability for each potential action using the actor network. Then, the policy function randomly selects an action based on these probabilities.

Configure the `codegen` function to generate code suitable for building a MEX file.

```
cfg = coder.config('mex');
```

On the configuration object, set the target language to C++, and set `DeepLearningConfig` to '`none`'. This option generates code without using any third-party library.

```
cfg.TargetLang = 'C';
cfg.DeepLearningConfig = coder.DeepLearningConfig('none');
```

Set an example input value for the policy evaluation function. To find the observation dimension, use the `getObservationInfo` function. In this case, the observations are in a four-element vector.

```
argstr = '{ones(4,1)}';
```

Generate code using the `codegen` function.

```
codegen('-config','cfg','evaluatePolicy','-args',argstr,'-report');
```

This command generates the C++ code for the policy gradient agent containing the deep neural network actor.

### Generate C++ Code for Deep Neural Network Policy using Third-Party Libraries

As an example, generate C++ code for the policy gradient agent trained in "Train PG Agent to Balance Cart-Pole System" on page 5-57 using the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN).

Load the trained agent.

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

Create a policy evaluation function for this agent.

```
generatePolicyFunction(agent)
```

This command creates the `evaluatePolicy.m` file, which contains the policy function, and the `agentData.mat` file, which contains the trained deep neural network actor. For a given observation, the policy function evaluates a probability for each potential action using the actor network. Then, the policy function randomly selects an action based on these probabilities.

Configure the `codegen` function to generate code suitable for building a MEX file.

```
cfg = coder.config('mex');
```

On the configuration object, set the target language to C++, and set `DeepLearningConfig` to the target library '`mkldnn`'. This option generates code using the Intel Math Kernel Library for Deep Neural Networks (Intel MKL-DNN).

```
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

Set an example input value for the policy evaluation function. To find the observation dimension, use the `getObservationInfo` function. In this case, the observations are in a four-element vector.

```
argstr = '{ones(4,1)}';
```

Generate code using the `codegen` function.

```
codegen('-config','cfg','evaluatePolicy','-args',argstr,'-report');
```

This command generates the C++ code for the policy gradient agent containing the deep neural network actor.

#### Generate C Code for Q Table Policy

As an example, generate C code for the Q-learning agent trained in “Train Reinforcement Learning Agent in Basic Grid World” on page 1-14.

Load the trained agent.

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

Create a policy evaluation function for this agent.

```
generatePolicyFunction(qAgent)
```

This command creates the `evaluatePolicy.m` file, which contains the policy function, and the `agentData.mat` file, which contains the trained Q table value function. For a given observation, the policy function looks up the value function for each potential action using the Q table. Then, the policy function selects the action for which the value function is greatest.

Set an example input value for the policy evaluation function. To find the observation dimension, use the `getObservationInfo` function. In this case, there is a single one dimensional observation (belonging to a discrete set of possible values).

```
argstr = '{[1]}';
```

Configure the `codegen` function to generate embeddable C code suitable for targeting a static library, and set the output folder to `buildFolder`.

```
cfg = coder.config('lib');
outFolder = 'buildFolder';
```

Generate C code using the `codegen` function.

```
codegen('-c','-d',outFolder,'-config','cfg',...
    'evaluatePolicy','-args',argstr,'-report');
```

## See Also

**Functions**
`generatePolicyFunction` | `generatePolicyBlock`

## Related Examples

- “Train Reinforcement Learning Agents” on page 5-3

## More About

- “Reinforcement Learning Agents” on page 3-2