Lab 4: Creating Simple MDP MATLAB Environment with a Q-Learning Agent

Abhishek M J - CS21B2018

21-02-2024

MDP Matlab Environment

```
MDP = createMDP(8, ["left"; "right"])
```

- This line creates a Markov Decision Process (MDP) object using the createMDP function.
- The first argument, 8, specifies the number of states in the MDP. Imagine eight distinct positions or situations in your environment.
- The second argument, ["left"; "right"], defines the possible actions that can be taken in each state. In this case, you can either move "left" or "right".

```
MDP =

  GenericMDP with properties:

              CurrentState: "s1"
                    States: [8×1 string]
                   Actions: [2×1 string]
                         T: [8×8×2 double]
                         R: [8×8×2 double]
            TerminalStates: [0×1 string]
      ProbabilityTolerance: 8.8818e-16
```

```
MDP.States
```

- It displays the numbers from 1 to 8, representing the unique identifiers for each state.

```
ans =

  8×1 string array

    "s1"
    "s2"
    "s3"
    "s4"
    "s5"
    "s6"
    "s7"
    "s8"
```

`MDP.Actions`

- It shows "left" and "right", the two actions you can take from any state.

```
ans =

  2×1 string array

    "left"
    "right"
```

`MDP.T`

- Each element in the array MDP.T(s, a, s') tells you the probability of transitioning from state s to state s' when action a is taken.

```
ans(:,:,1) =

     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0


ans(:,:,2) =

     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

`MDP.R`

- Each element in the array MDP.R(s, a) tells you the reward you receive immediately after taking action a in state s.

```
ans(:,:,1) =

    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10


ans(:,:,2) =

    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
    10    -1    -1    -1    -1    -1    -1    10
```

Defining Rewards and Transitions Probabilities

```
MDP.TerminalStates = ["s1";"s8"];
```

- This line specifies that states 1 and 8 are terminal states. Once the agent reaches one of these states, the episode ends.

```
nS = numel(MDP.States);
nA = numel(MDP.Actions);
```

- This line calculates the number of states and actions in the MDP.

```
MDP.R = -1*ones(nS,nA);
```

- This line initializes the reward array with -1 for all state-action pairs.

```
MDP.R(:, state2idx(MDP, MDP.TerminalStates), :) = 10;
```

- This line sets the reward to 10 for all state-action pairs that lead to a terminal state.

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

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

MDP.T(3, 2, 1) = 1;
MDP.T(3, 4, 2) = 1;

MDP.T(4, 3, 1) = 1;
```

```
MDP.T(4, 5, 2) = 1;

MDP.T(5, 4, 1) = 1;
MDP.T(5, 6, 2) = 1;

MDP.T(6, 5, 1) = 1;
MDP.T(6, 7, 2) = 1;

MDP.T(7, 6, 1) = 1;
MDP.T(7, 8, 2) = 1;

MDP.T(8, 7, 1) = 1;
MDP.T(8, 8, 2) = 1;
```

- This block of code sets the transition probabilities for each state-action pair. For example, MDP.T(1, 2, 2) = 1 sets the probability of transitioning from state 1 to state 2 when action 2 ("right") is taken to 1.
- In simpler terms, the agent moves to right from state 1 it will always (with probability 1) reach state 2.

```
MDP.T
MDP.R
```

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

- This line creates a reinforcement learning environment using the rlMDPEnv function. The environment is based on the MDP object we created earlier.

Define Q-Table and Initialize Agent

```
state_information = getObservationInfo(env)
action_information = getActionInfo(env)
```

- This line retrieves the observation and action information from the environment.

```
state_information = 

  rlFiniteSetSpec with properties:

       Elements: [8×1 double]
           Name: "MDP Observations"
    Description: [0×0 string]
      Dimension: [1 1]
       DataType: "double"


action_information = 

  rlFiniteSetSpec with properties:

       Elements: [2×1 double]
           Name: "MDP Actions"
    Description: [0×0 string]
      Dimension: [1 1]
       DataType: "double"
```

```
qTable = rlTable(state_information, action_information)
qTable.Table
```

- This line creates a Q-table using the rlTable function. The Q-table is a matrix that stores the Q-values for each state-action pair.

```
qTable = 

  rlTable with properties:

    Table: [8×2 double]


ans =

     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
```

```
qTable.Table = ones(size(qTable.Table))*5
qTable.Table
```

- This line initializes the Q-table with a constant value of 5.

```
qTable =

  rlTable with properties:

    Table: [8×2 double]


ans =

     5     5
     5     5
     5     5
     5     5
     5     5
     5     5
     5     5
     5     5
```

```
qRepresentation = rlQValueRepresentation(qTable, state_information, action_information)
qRepresentation.Options
qRepresentation.Options.L2RegularizationFactor = 0;
qRepresentation.Options.LearnRate = 0.01;
```

- This block of code creates a Q-value representation using the rlQValueRepresentation function. The Q-value representation is used to define the learning parameters for the Q-learning agent.
- The L2RegularizationFactor and LearnRate options are set to 0 and 0.01, respectively.

```
qRepresentation =

  rlQValueRepresentation with properties:

         ActionInfo: [1×1 rl.util.rlFiniteSetSpec]
    ObservationInfo: [1×1 rl.util.rlFiniteSetSpec]
            Options: [1×1 rl.option.rlRepresentationOptions]


ans =

  rlRepresentationOptions with properties:

                  LearnRate: 0.0100
          GradientThreshold: Inf
    GradientThresholdMethod: "l2norm"
     L2RegularizationFactor: 1.0000e-04
                  UseDevice: "cpu"
                  Optimizer: "adam"
        OptimizerParameters: [1×1 rl.option.OptimizerParameters]
```

```
agentOpts = rlQAgentOptions
agentOpts.EpsilonGreedyExploration
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 0.01
qAgent = rlQAgent(qRepresentation, agentOpts)
```

- This block of code creates a Q-learning agent using the rlQAgent function. The agent uses the Q-value representation and agent options we defined earlier.
- The EpsilonGreedyExploration.EpsilonDecay option is set to 0.01, which means the exploration rate decreases by 0.01 after each episode.
- The exploration rate determines the probability of the agent taking a random action instead of the action with the highest Q-value.

```
agentOpts = 

  rlQAgentOptions with properties:

                    SampleTime: 1
                DiscountFactor: 0.9900
    EpsilonGreedyExploration: [1×1 rl.option.EpsilonGreedyExploration]
      CriticOptimizerOptions: [1×1 rl.option.rlOptimizerOptions]
                    InfoToSave: [1×1 struct]


ans = 

  EpsilonGreedyExploration with properties:

    EpsilonDecay: 0.0050
         Epsilon: 1
      EpsilonMin: 0.0100


agentOpts = 

  rlQAgentOptions with properties:

                    SampleTime: 1
                DiscountFactor: 0.9900
    EpsilonGreedyExploration: [1×1 rl.option.EpsilonGreedyExploration]
      CriticOptimizerOptions: [1×1 rl.option.rlOptimizerOptions]
                    InfoToSave: [1×1 struct]


qAgent = 

  rlQAgent with properties:

            AgentOptions: [1×1 rl.option.rlQAgentOptions]
    UseExplorationPolicy: 0
         ObservationInfo: [1×1 rl.util.rlFiniteSetSpec]
              ActionInfo: [1×1 rl.util.rlFiniteSetSpec]
              SampleTime: 1
```

Train the Q-Learning Agent

```
trainOpts = rlTrainingOptions
trainOpts.MaxStepsPerEpisode = 10;
trainOpts.MaxEpisodes = 100;
trainOpts.StopTrainingCriteria = "AverageReward";
```

```
trainOpts.StopTrainingValue = 13;
trainOpts.ScoreAveragingWindowLength = 30;
```

- This block of code creates training options using the rlTrainingOptions function. The training options specify the maximum number of steps and episodes, as well as the stopping criteria for training.

```
trainOpts =

  rlTrainingOptions with properties:

                   MaxEpisodes: 500
            MaxStepsPerEpisode: 500
                   StopOnError: "on"
    ScoreAveragingWindowLength: 5
          StopTrainingCriteria: "AverageSteps"
             StopTrainingValue: 500
             SaveAgentCriteria: "none"
                SaveAgentValue: "none"
            SaveAgentDirectory: "savedAgents"
                       Verbose: 0
                         Plots: "training-progress"
                   UseParallel: 0
        ParallelizationOptions: [1×1 rl.option.ParallelTraining]

     5     5
     5     5
     5     5
     5     5
     5     5
     5     5
     5     5
     5     5
```

```
QTable0 = getLearnableParameters(getCritic(qAgent));
disp(QTable0{1})
```

- This line retrieves the initial Q-table from the Q-learning agent.

```
Data =

  struct with fields:

       Observation: [1×1 struct]
            Action: [1×1 struct]
            Reward: [1×1 timeseries]
            IsDone: [1×1 timeseries]
    SimulationInfo: [1×1 struct]


cumulativeReward =

     9
```

```
doTraining = true;
if doTraining
    trainingStats = train(qAgent, env, trainOpts);
else
    load('genericMDPQAgent.mat', 'qAgent')
end
```

- This block of code trains the Q-learning agent using the train function. The training process is controlled by the training options we defined earlier.
- The trainingStats variable stores the training statistics, such as the average reward per episode and the total number of steps taken.

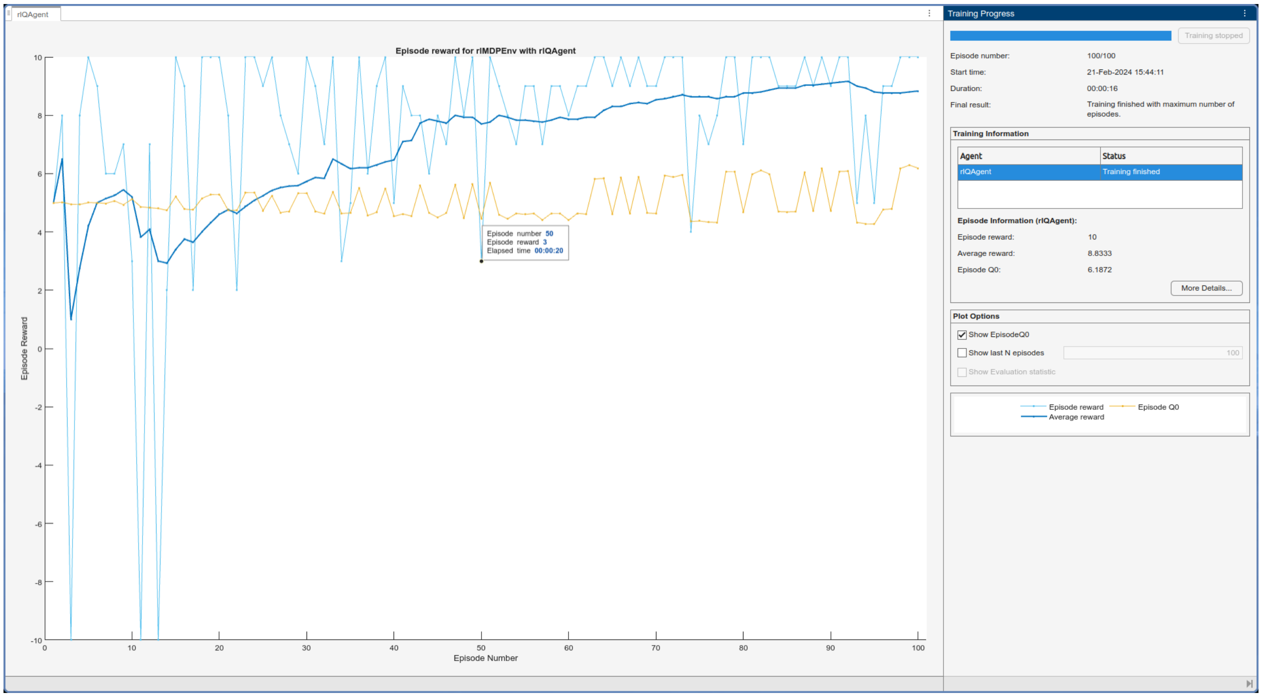
```
ans =

  8×2 single matrix

    5.0000    5.0000
    6.1872    4.6774
    4.7846    4.5259
    4.3065    4.2784
    4.2419    4.2593
    4.5215    4.8004
    4.6732    6.2947
    5.0000    5.0000
```

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

- This line simulates the Q-learning agent in the environment using the sim function. The Data variable stores the observations, actions, and rewards collected during the simulation.
- The cumulativeReward variable calculates the total reward obtained by the agent during the simulation.

```
QTable1 = getLearnableParameters(getCritic(qAgent));
disp(QTable1{1})
```

- This line retrieves the final Q-table from the Q-learning agent after training.


rlQAgent
Episode reward for rlMDPEnv with rlQAgent
Episode Reward
Episode Number
Episode number 50
Episode reward 3
Elapsed time 00:00:20
Training Progress
Training stopped
Episode number: 100/100
Start time: 21-Feb-2024 15:44:11
Duration: 00:00:16
Final result: Training finished with maximum number of episodes.
Training Information
Agent
Status
rlQAgent
Training finished
Episode Information (rlQAgent):
Episode reward: 10
Average reward: 8.8333
Episode Q0: 6.1872
More Details...
Plot Options
Show EpisodeQ0
Show last N episodes
Show Evaluation statistic
Episode reward
Episode Q0
Average reward