

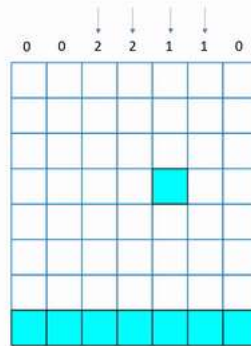
## Stochastic Gridworld Environment with Deep Q-Learning Agent

In this module we cover stochastic environments and how a Deep Q learning agent is generated and trained. In the previous module we saw how to use and update the pre defined environment functions in MATLAB. This module is more focused on using and exploring Reinforcement Learning designer App for creating and training a DQN agent.

1. Create a Stochastic Gridworld environment using an edited MATLAB function
2. Model, define and initialize a Deep Q-Learning Agent
3. Using Reinforcement Learning Designer App

Please go through the supplementary file named 'Reason to use Deep Learning Agents' before you start with this module.

### Problem Formulation



**Goal:** Reach the nearest **positive** terminal states as quickly as possible

**Actions:** The agent can move in 4 possible directions

**States:** There are 56 states with 7 Positive Terminal (8th Row) and 1 Negative Terminal state (4,5)

**Reward:** All nonterminal states have a small negative reward (-1) and terminal states have a large positive reward (10)

**Assumptions:**

- Agent has to move around, cannot stop at one place.
- The stochasticity in the environment affects the agents movement.
- The environment pushes the agent towards the bottom of the grid with a given intensity
- If the agent goes up from state [4,2], it will land in state [6,2]

### Creating the Environment

**TASK:** Go through the pre defined MATLAB environment 'WaterFallGridWorld-Stochastic'

Use function `rlPredefinedEnv(keyword)`, which takes an input that is keyword for environment defined.

```
%Start you code ~ one line
env= rlPredefinedEnv('WaterFallGridWorld-Stochastic')
```

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

```
%Code ends here
```

Visualize the environment using plot function

```
%Start you code ~ one line
plot(env)
```

```
%Code ends here
```

Access pre-defined properties of the in-built environment using dot notation

```
%Start you code ~ one line
env.Model
```

```
ans =
  GridWorld with properties:

    GridSize: [8 7]
    CurrentState: "[5,1]"
    States: [56x1 string]
    Actions: [4x1 string]
    T: [56x56x4 double]
    R: [56x56x4 double]
    ObstacleStates: [0x1 string]
    TerminalStates: [8x1 string]
```

```
%Code ends here
```

- Show the Transition Probabilities
- Show the Reward
- Show the Actions

```
%Start you code ~ two lines
env.Model.T
```

[illegible]

```
env.Model.R
```

[illegible]

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



env.Model.Actions

```
ans = 4x1 string  
"N"  
"S"  
"E"  
"W"
```

%Code ends here

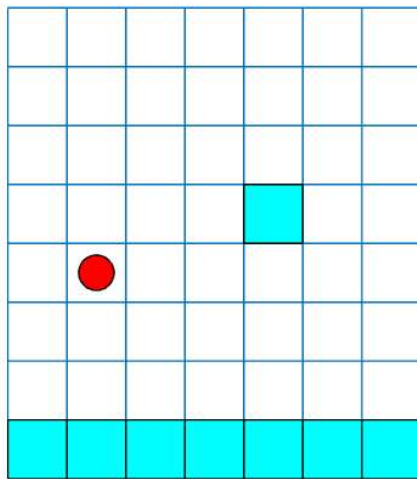
Use the step function to visualize behavior of an agent in the environment

Move the agent from position to (5,1) to (5,2). What do you notice regarding the selected action and landed state?

Use the dot notation to access the step function of the environment. step function takes in an Input of the action that the agent needs to perform.

Here the action E indexed at position 3 will move the agent towards left to state (5,2)

```
%Start your code here  
env.step(3)
```

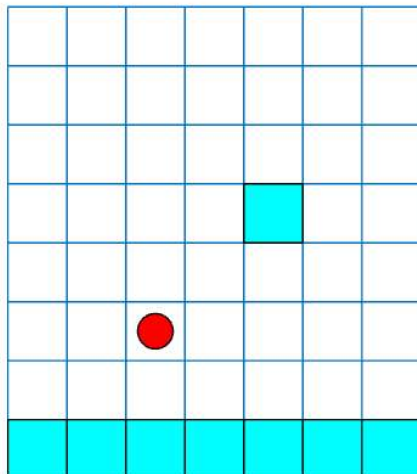


ans = 13

%end your code here

Move the agent from position to (5,2) by taking action East. What do you notice regarding the selected action and landed state?

```
%Start your code here  
env.step(3)
```

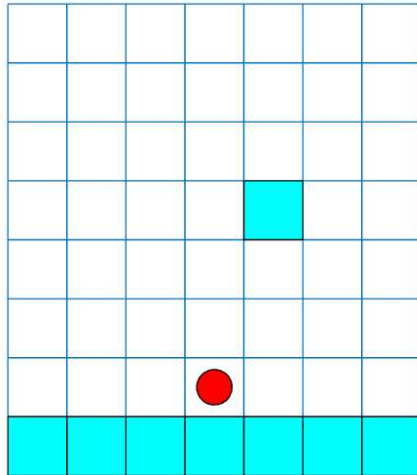


ans = 22

%Code ends here

Move the agent from position to (5,3) by taking action East. What do you notice regarding the selected action and landed state?

%Start your code here  
env.step(3)



ans = 31

%Code ends here

## Creating an Agent

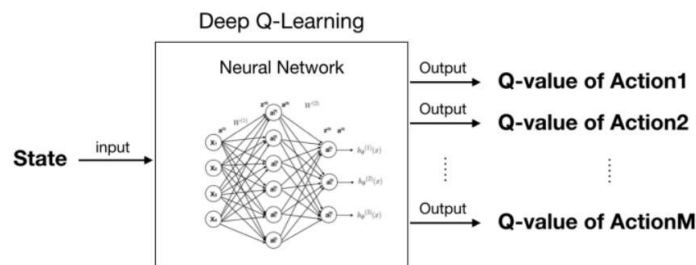
We have now seen how our environments behaves, let us move on to see how to construct and model a Deep Q learning agent.

This can be done in following ways -

1. Using Reinforcement Learning Designer app.
2. Using RL toolbox agent functions (Like module 1)
3. Modeling our own agent using Deep Designer App.

In this module we use method 1 : Using Reinforcement Learning Designer app for initiaizing agent

### Method 1: Using Reinforcement Learning Designer app for initiaizing agent.



Like a Q leaning agent, Deep Q Learning agent is a model-free, online, off-policy reinforcement learning agent. Instead of table, we have a neural network as function approximator to map relation between input (States) and output (State- Action Values).

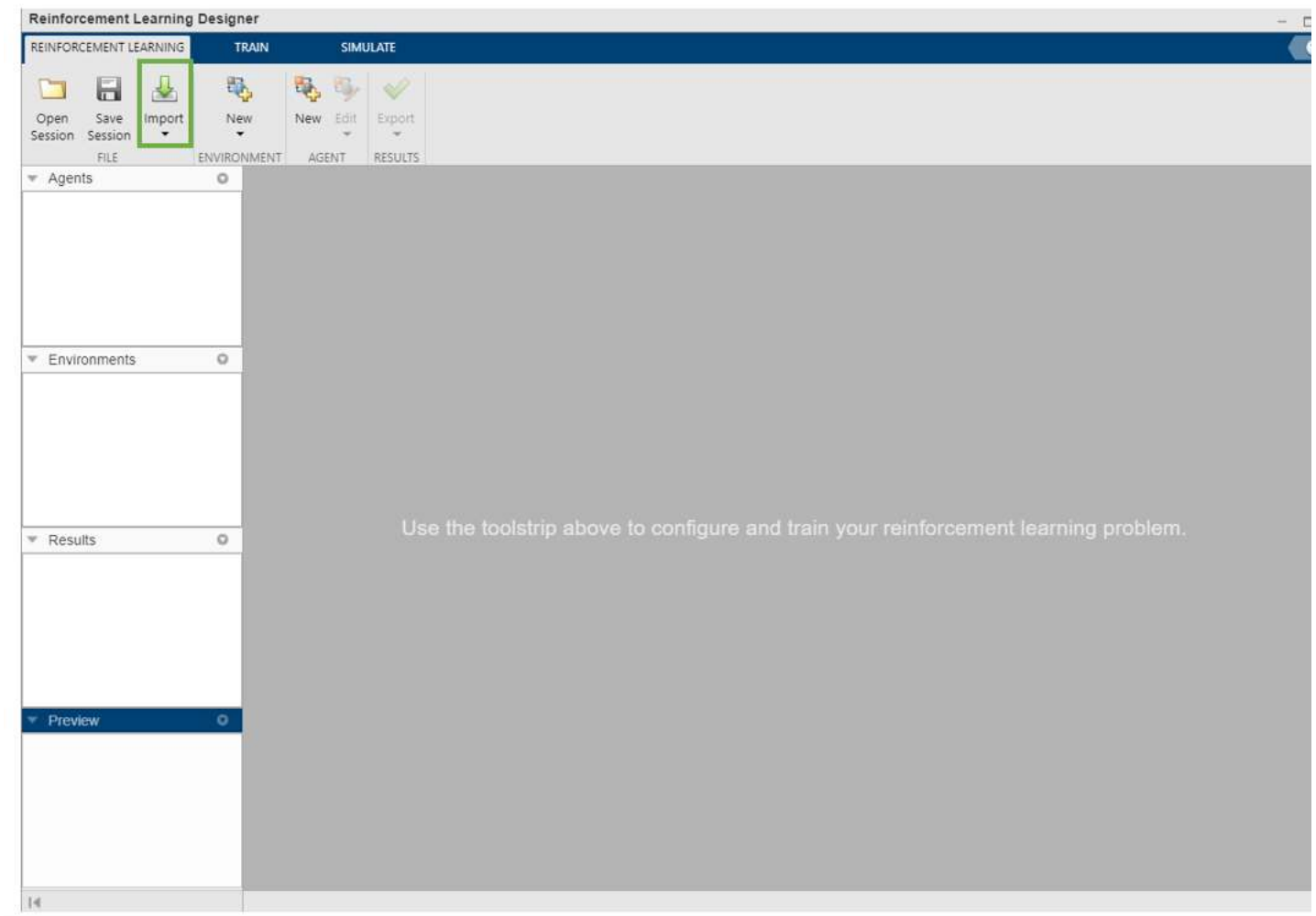
Let us open the Reinforcement Learning Designer App. Simply type in ' reinforcementLearningDesigner '. Or select Apps tab from the ribbon above.

Please visit [this](#) page for more information

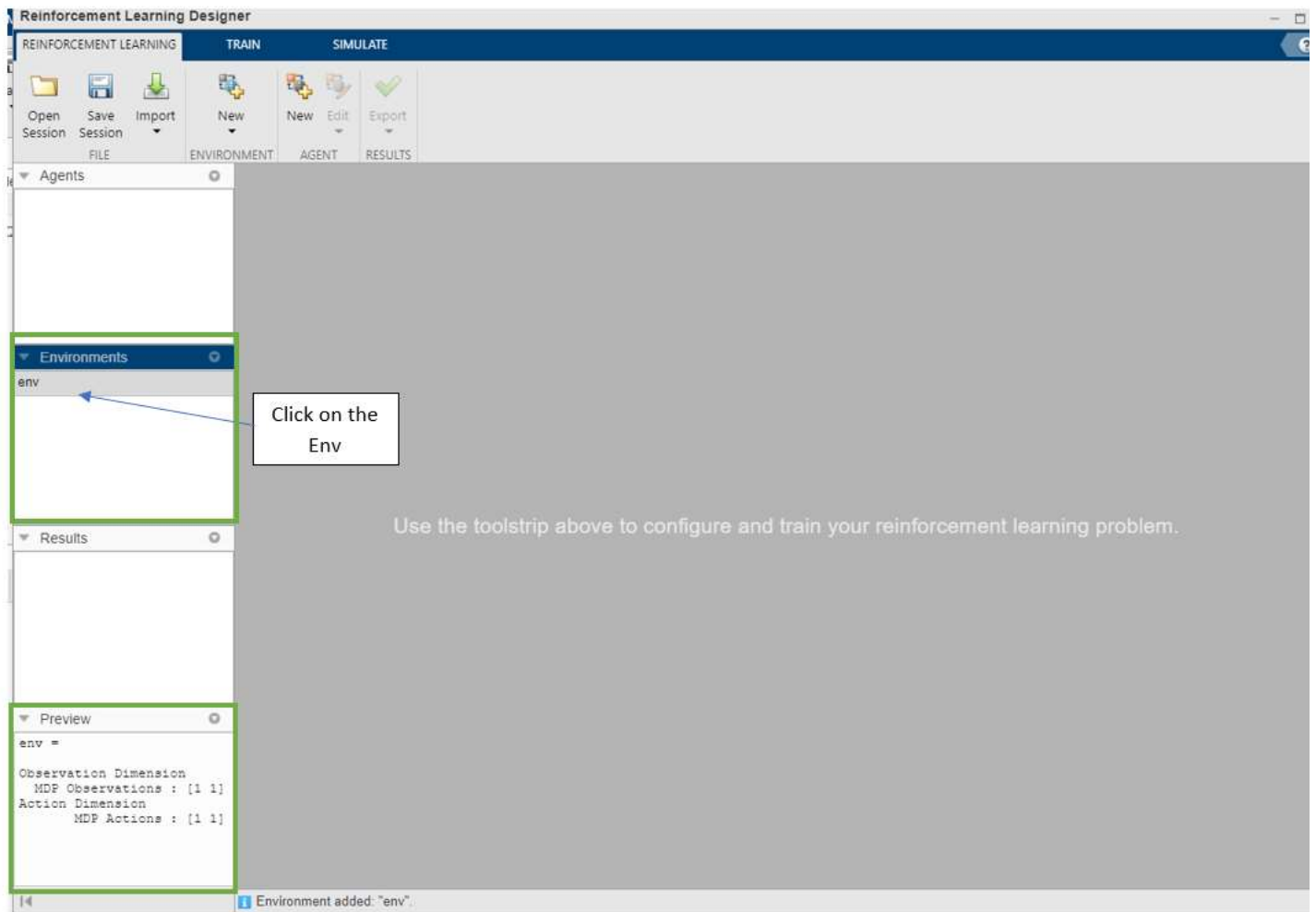
%Start your code here  
reinforcementLearningDesigner  
%Code ends here

Follow the images and steps to train your agent:

**Step 1:** Click on the Import button to load the environment stored in the variable env

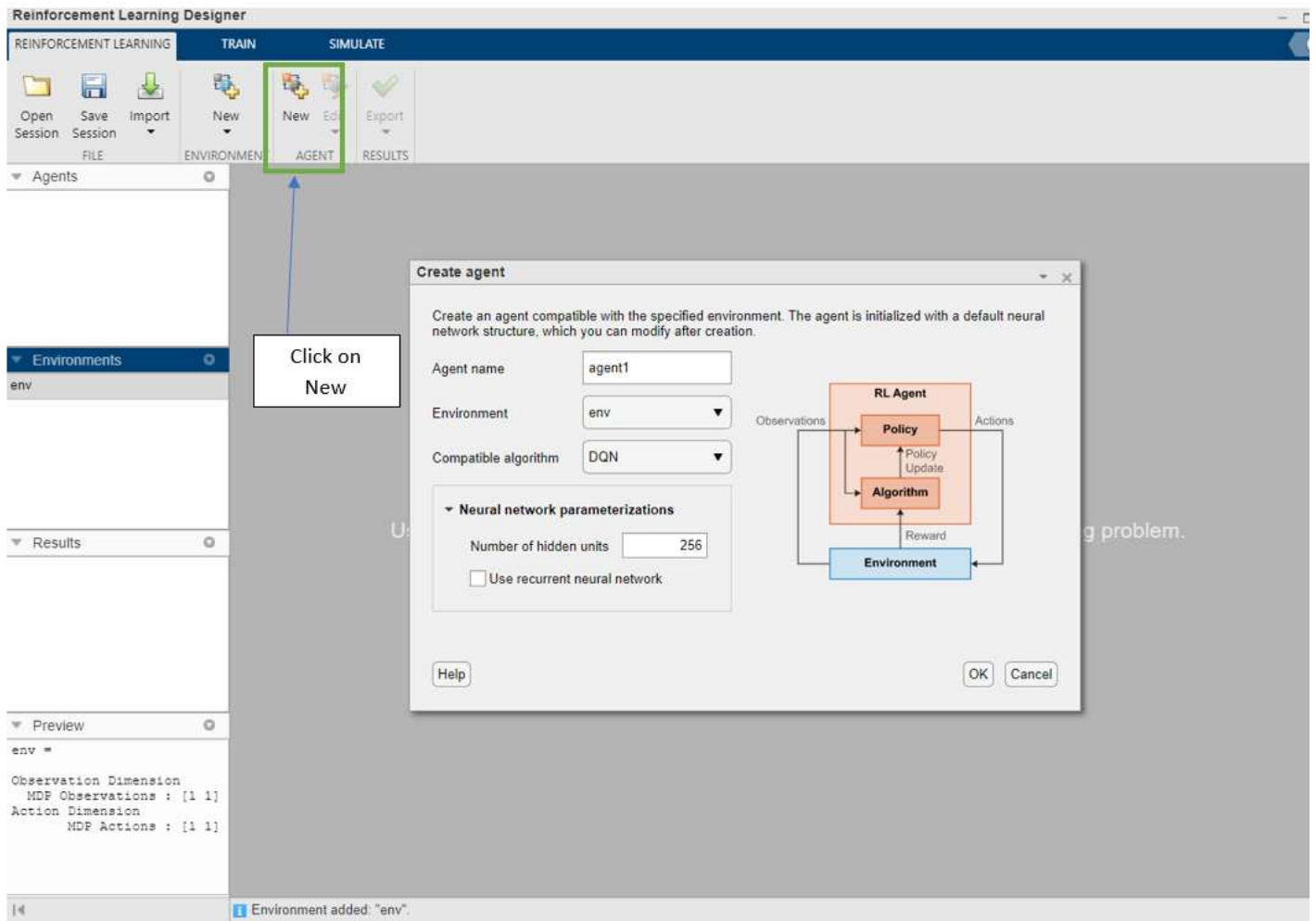


**Step 2:** Click on the env loaded in the Environment pane. This will populate the Preview Pane as well.



**Step 3:** Creating an Agent.

This is similar to using RL Toolbox function - [rlDDQNAgent\(observationInfo,actionInfo\)](#).



In the Create Agent window following options can be modified:

Agent Name- Name your agent

Environment - Select the environment on which you want to train your agent

Compatible Algorithm - Select a training algorithm that will be used for updating the agent's learning process

Hit **OK** once the editing is done.

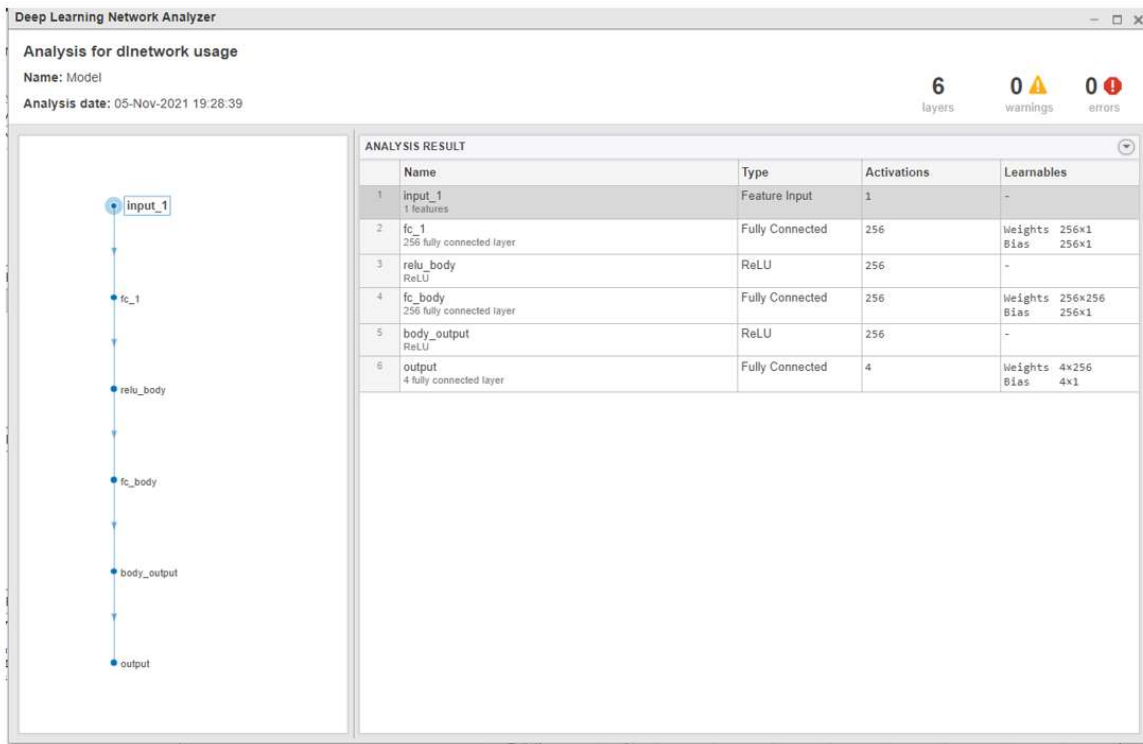
**Step 4:** Visualizing the neural network of agent created

Select the agent in Agent pane

Click on the View Critic Model

Close the Window once done





1. Input Layer - Feature Input Layer -- with one input. With every step the agent lands in next state (grid location denoted by s1, s2, s3,...s56)
2. Two Fully connected hidden layer with 256 neurons each. These layers are then fitted with ReLU activation function
3. Output Layer - Fully connected with neurons equal to number of action i.e 4

**Step 5:** Click on the Agent name in Agent Pane(Window) then click on Train button in tabs at the top.

Reinforcement Learning Designer

REINFORCEMENT LEARNING    TRAIN    SIMULATE    DQN AGENT

DQN\_Agent    Import    View Critic Model    Train    Simulate    Export

NAME    IMPORT    REPRESENTATION    NEXT STEPS

Agents

DQN\_Agent

Environments

env

Results

Preview

env =

Observation Dimension

MDF Observations : [ 1 1]

Action Dimension

MDF Actions : [ 1 1]

Discount factor    0.95

Execution environment    ☒ CPU    ☐ GPU

Batch size    64

Experience buffer length    1e+04

More Options

Gradient threshold    Inf

More Options

Optimizer    adam

Denominator offset    1e-08

Gradient decay    0.9

Squared gradient decay    0.999

Gradient threshold method    l2norm

L2 regularization    0.0001

Exploration

Epsilon Greedy Exploration Options

Initial epsilon    1

Epsilon decay    0.005

Epsilon min    0.01

Plot Options

X-axis limit    1000

Epsilon decay

Value

Steps

Epsilon

Epsilon min

Agent opened: "DQN\_Agent"



**Step 6:** In this step we specify -

1. Agent Learning Parameters such as (Learn Rate, exploration rate, Epsilon decay)
2. Training parameters such as (Episode, Stopping criteria, etc)

This is similar to using RL toolbox functions - [rlDQNAgentOptions](#), [rlQValueRepresentation](#), [rlTrainingOptions](#).

The screenshot displays the Reinforcement Learning Designer window. The top tabs are REINFORCEMENT LEARNING, TRAIN, SIMULATE, and DQN AGENT. The TRAIN tab is active, showing various configuration fields. A green box highlights the top section containing Environment (env), Agent (DQN\_Agent), Max Episodes (700), Max Episode Length (700), Average Window Length (5), Stopping Criteria (EpisodeCount), and Stopping Value (700). Another green box highlights the bottom section containing Exploration options: Initial epsilon (1), Epsilon decay (0.01), Epsilon min (0.01), and X-axis limit (700). The TRAIN button is highlighted with a blue arrow and a text box that says "Click on the Train after putting in the training options". The bottom right shows an "Epsilon decay" plot with a solid blue line for Epsilon and a dashed red line for Epsilon min, both decaying over 700 steps.

**Reinforcement Learning Designer**

**TRAIN**

Environment: env  
Agent: DQN\_Agent

Max Episodes: 700  
Max Episode Length: 700  
Average Window Length: 5

Stopping Criteria: EpisodeCount  
Stopping Value: 700

**TRAINING OPTIONS**

Discount factor: 0.95  
Execution environment: CPU (selected) GPU  
Batch size: 64  
Experience buffer length: 1e+04

**More Options**

Gradient threshold: Inf  
Optimizer: adam  
Denominator offset: 1e-08  
Gradient decay: 0.9  
Squared gradient decay: 0.999  
Gradient threshold method: l2norm  
L2 regularization: 0.0001

**Exploration**

**Epsilon Greedy Exploration Options**

Initial epsilon: 1  
Epsilon decay: 0.01  
Epsilon min: 0.01

**Plot Options**

X-axis limit: 700

**Epsilon decay plot**

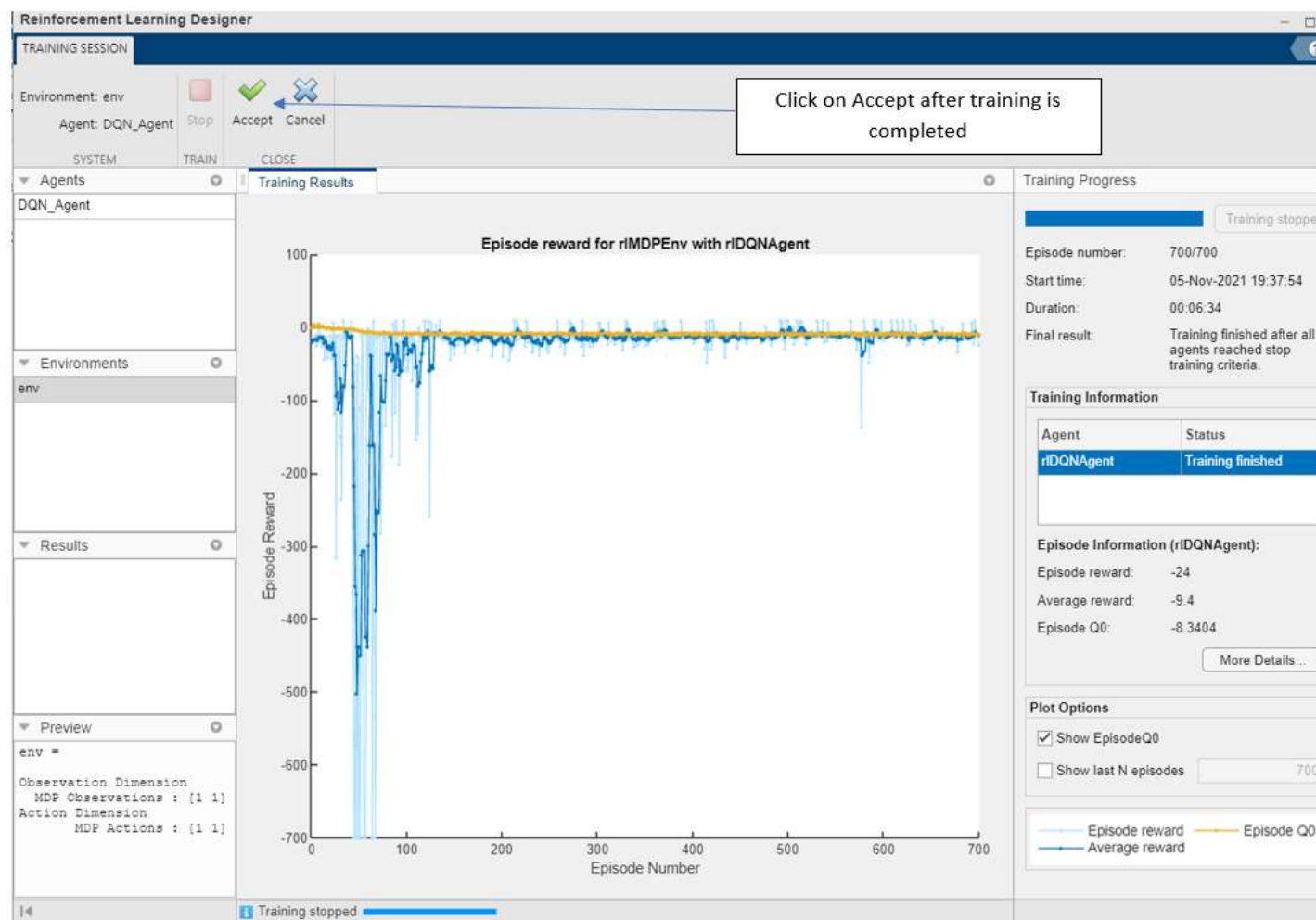
Value vs Steps (0 to 700). The plot shows Epsilon (solid blue line) and Epsilon min (dashed red line) decaying over time.

Click on the Train after putting in the training options

**Step 7:** Analyzing Training Process

The training information is seen on the right side of the window.

Click on the Accept result button to store the training hyper parameters. This trained agent can now be used to simulate in an environment or if necessary put into re-training process.



**Step 8:** Simulate the trained agent

This is similar to using RL Toolbox functions [rlSimulationOptions](#) and [sim](#).

DQN AGENT

DQN\_Agent\_Trained

Import

View  
Critic Model

Train

Simulate

Export

NAME

IMPORT

REPRESENTATION

NEXT STEPS

Click on Simulate to see how the trained agent performs

Agents

DQN\_Agent

DQN\_Agent\_Trained

Environments

env

Results

trainStats1

Preview

DQN\_Agent\_Trained =

Type DQN

Observation Dimension

MDP Observations : [1 1]

Action Dimension

MDP Actions : [1 1]

Critic Learn Rate 0.01

Overview

Hyperparameters

Agent Options

Sample time

Discount factor

Execution environment ☒ CPU ☐ GPU

Batch size

Experience buffer length

More Options

Critic Options

Learn rate

Gradient threshold

More Options

Exploration

Epsilon Greedy Exploration Options

Initial epsilon

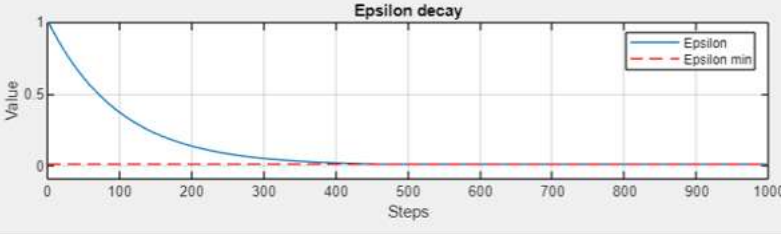
Epsilon decay

Epsilon min

Plot Options

X-axis limit

Epsilon decay



Steps	Epsilon	Epsilon min
0	1.0	0.01
100	0.37	0.01
200	0.14	0.01
300	0.05	0.01
400	0.02	0.01
500	0.01	0.01
600	0.01	0.01
700	0.01	0.01
800	0.01	0.01
900	0.01	0.01
1000	0.01	0.01

14 Opening Agent Editor

Reinforcement Learning Designer

REINFORCEMENT LEARNING

TRAIN

SIMULATE

DQN AGENT

Environment

env

Agent

DQN\_Agent\_Trained

Number of Episodes

20

Max Episode Length

100

Stop on Error

☒

Use Parallel

Simulate

SYSTEM

SIMULATION OPTIONS

SIMULATE

Agents

DQN\_Agent

DQN\_Agent\_Trained

Environments

env

Results

trainStats1

Preview

DQN\_Agent\_Trained =  
Type DQN  
Observation Dimension  
MDP Observations : [1 1]  
Action Dimension  
MDP Actions : [1 1]  
Critic Learn Rate 0.01

Overview

A DQN agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards.  
[Learn more](#)

Observation Specification

Observation Name	Domain	Dimension

Action Specification

Action Name	Domain	Dimension

Hyperparameters

Agent Options

Sample time

1

Discount factor

0.95

Execution environment

☒ CPU ☐ GPU

Batch size

64

Experience buffer length

1e+04

Critic Options

Learn rate

0.01

Gradient threshold

Inf

More Options

Optimizer

adam

Denominator offset

1e-08

Gradient decay

0.9

Squared gradient decay

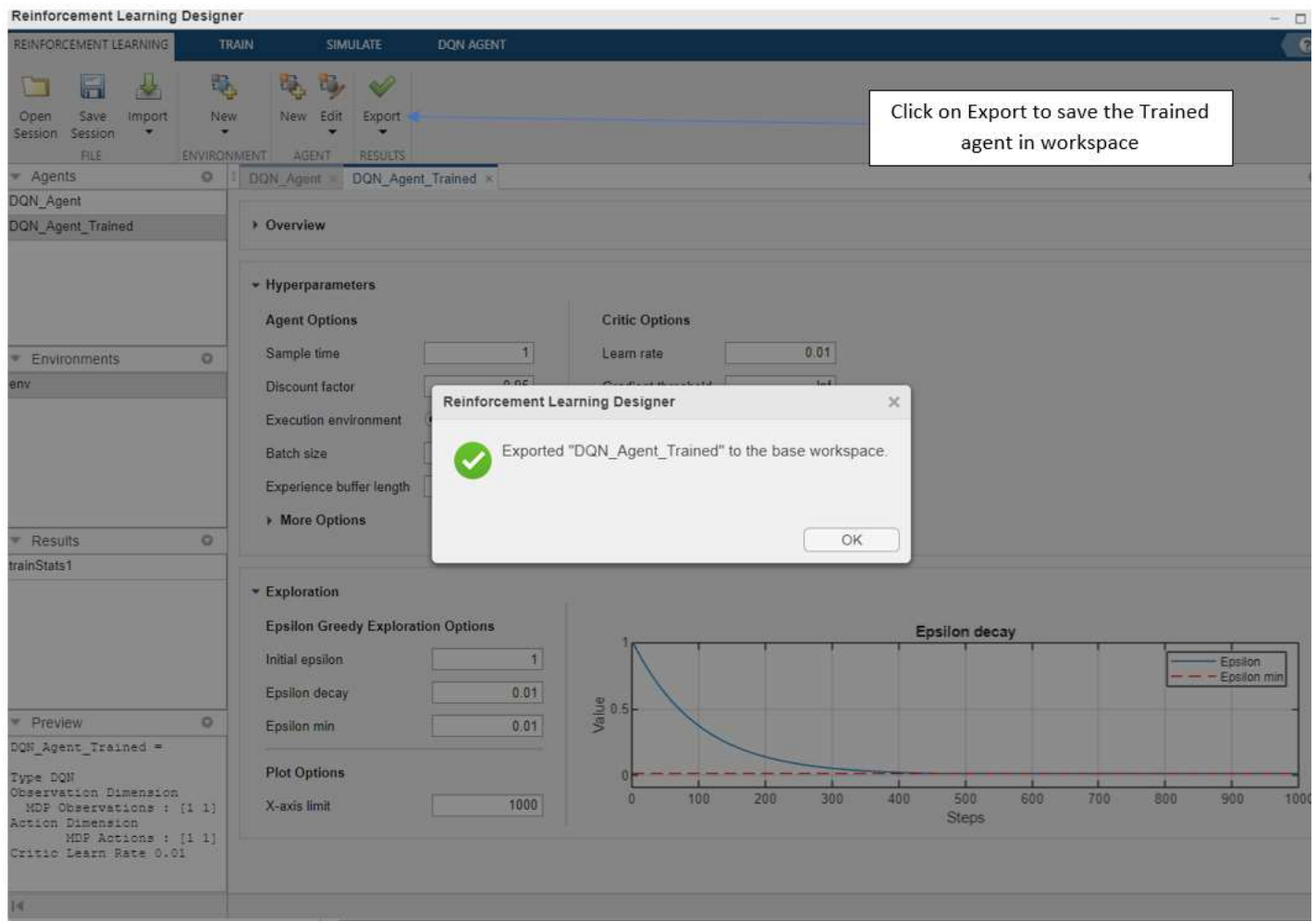
0.999

Gradient threshold method

l2norm

Agent opened: "DQN\_Agent\_Trained"

**Step 9:** Saving the trained agent in Workspace



Let us analyze the trained network and extract the deep neural network from the critic.

Save the trained DQN\_Agent\_Trained in a variable named 'agent'

```
%Start code here ~ one line
agent = DQN_Agent_Trained
```

```
agent =
    rlDQNAgent with properties:
        AgentOptions: [1x1 rl.option.rlDQNAgentOptions]
        ExperienceBuffer: [1x1 rl.util.ExperienceBuffer]
```

```
%Code ends here
```

Use function `getCritic(agent)`. This function returns the critic representation object for the specified reinforcement learning agent.

```
%Start your code here ~ one line
critic = getCritic(agent)
```

```
critic =
    rlQValueRepresentation with properties:
        ActionInfo: [1x1 rl.util.rlFiniteSetSpec]
        ObservationInfo: [1x1 rl.util.rlFiniteSetSpec]
        Options: [1x1 rl.option.rlRepresentationOptions]
```

```
%Code ends here
```

Now we obtain the hyper parameter of the trained using function `getLearnableParameters(critic)`

```
%Start your code here ~ one line
parameters = getLearnableParameters(critic)
```

```
parameters = 6x1 cell
```

	1
1	256×1 single
2	256×1 single
3	256×256 si...

	1
4	256×1 single
5	4×256 single
6	[-1.8119;-1....

```
%Code ends here
```

In this module we learnt

- Limitations of tabular RL agents
- How Deep Neural Networks can be applied as a function approximator for RL agents
- How to use Reinforcement Learning Designer App to create and train an agent

In the next module to Custom build MATLAB environments