Reinforcement Learning

Raghvendra - 50289128

December 6, 2018

1 Introduction

We simulate a Reinforcement learning model with Deep Neural layers in order to understand how an agent learns to reach a static target without any supervised learning.

We break this report in following sections:

- Code Snippet Implementation - Description
- Training the Agent - Learning time and Improvement
- Hyper-parameter Tuning
- Writing Tasks

2 Code Snippet Implementation - Description

The code has been divided into following snippets:

- **Packages**
  Here we are simply importing all the packages required by our notebook for model compilation and training etc.
- **Environment**
  here we define our grid env. and render TOM and JERRY, additionally methods for resetting and updating the environment with each steps are compiled in this class
- **Random Actions**
  This is a block of code which helps in testing our environment,ent rendering and showcase a random set of actions taken by our Agent(TOM) on the grid. **Note** - The random actions do not guarantee any improvement and is not the basis for our model, rather a form of testing our env.
- **Brain**
  The Brain is where we introduce DEEP Q LEARNING aspect for our RL model, we create and compile our model with two hidden layers in this class. More details are present in 2.1
- **Memory**
  A simple class to store our (*current state*, *action*, *reward*, *next state*) tuple in order to allow our Deep-Q model to sample(learn) from later during training.

- **Agent**
  Agent is the class where we are basically teaching our moving entity (TOM) to navigate through the environment, we have methods such as *ACT*, *OBSERVE* and *REPLAY* to decide between exploration/exploitation, to add our tuple into memory and update exploration parameter(epsilon) and finally to train and update quality matrix of our learned decisions respectively. More on this in 2.2

- **Main Block**
  We are finally simulating our environment and agent for different number of episodes and with different epsilon , we also render our simulation per 1000 episode and print info about time elapsed, episode rolling mean. More info on this in (3) and 4

2.1 Initializing our Hidden layers

We use the following code to initialize our model with two hidden layers:

```
### START CODE HERE ### (≈ 3 lines of code)
model.add(Dense(output_dim=128, activation='relu', input_dim=self.state_dim))
model.add(Dense(output_dim=128, activation='relu'))
model.add(Dense(output_dim=self.action_dim, activation='linear'))
### END CODE HERE ###
```

Figure 1: Adding Two Hidden layers with activation function

Out first hidden layer has 128 nodes, it receives an input tuple of 4 values (*fruit y*, *fruit x*, *player y*, *player x*), activated on 'relu' activation function, the next hidden layer further processes on 128*128 weights before outputting 4 q values for each action to the output layer.

We further use RMSProp optimizer with a learning rate initially set to *0.00025*. The model is compiled with MSE (Mean square error)loss function.

As in all other deep neural networks we have implemented, the DNN model allow us to learn from the past plays of the agent by keeping a track of previous experiences(evaluated actions for each state). In addition, it allows the model to :

- Avoid forgetting previous experiences
- Reduce correlation between experiences
- Increase learning speed with min batches

The main advantage of training over agent's past experience is that we're able to train on randomly drawn batches of experiences, by keeping the training set random we prevent the network from learning only from what it is immediately doing in the environment and allow it to learn from a varied past experiences.

2.2 Updating Epsilon and Q-values

Below is the code snippet of updation done on Epsilon (Exploration parameter) and target Q-values:

```
### START CODE HERE ### (≈ 1 line of code)
self.epsilon = self.min_epsilon + np.exp(-self.lamb * self.steps) * (self.max_epsilon - self.min_epsilon)
### END CODE HERE ###
```

Figure 2: Decaying code snippet for EPSILON

```
### START CODE HERE ### (≈ 4 line of code)
if st_next is None:
  t[act] = rew
else:
  t[act] = rew + self.gamma * np.amax(q_vals_next[i])
### END CODE HERE ###
```

Figure 3: Updating our target action Q values based on Bellman equation

The epsilon is more prevalent during the initial phase of our agent's interaction with the environment, when we want out agent to explore around and gather data to be trained on and as the formula suggest with each episode we decay the epsilon to allow our model to then act on the best q-values obtained from the training data.

We tune the decay parameter lambda here in order to understand the optimum decay rate. More on this in 4

As for Q-VALUES, it is a measure of the quality of the steps that can be taken from each state.

3 Training the Agent - Learning time and Improvement

We are training the agent in a formal DNN method, at the beginning of an episode, we reset the environment, and pass it's return value, the initial state, to the agent's act method. This returns an action, which is then passed to the environment's step method. This returns the next state, the reward, and the Boolean indicating if the episode is over. We then save the observation tuple to the agent's memory via the agent's observe method, then run a round of training by calling the agent's replay method. We can then render the environment. If the episode is over, we break from this loop, otherwise we continue with the next state being passed to the agent as the (now) current state.

As is described earlier, the tendency of agent to act randomly is much more in the earlier episodes and once we start learning from previous episodes and have a fair amount of learned data in Q-matrix, the decayed epsilon will have much less effect and agent will act on best q-values for the corresponding state and actions.

Below is the list of training time and mean awards for different epochs:

Please note: We have not changed any other hyper-parameter here, we will explore improving our model in section 4

Learning time and Mean Award for different epochs				
Epochs	Learning Time	Mean Reward	Final epsilon val	Target Reached
1000	76.35s	1.36	0.656	No
5000	406.75s	5.18	0.142	Yes
1000	809.47s	6.25	0.060	Yes

```
Episode 900
Time Elapsed: 76.35s
Epsilon 0.6564740764456966
Last Episode Reward: 2
Episode Reward Rolling Mean: 1.3595505617977528
----------
```

```
Episode 4900
Time Elapsed: 406.75s
Epsilon 0.1421679065286844
Last Episode Reward: 8
Episode Reward Rolling Mean: 5.186627785877942
----------
```

```
Episode 9900
Time Elapsed: 809.47s
Epsilon 0.06024164226910593
Last Episode Reward: 8
Episode Reward Rolling Mean: 6.2541577390062235
----------
```

Figure 4: Learning time and Mean award for the above three configuration

Below are the screenshot from the execution for above described table:
Mean reward graph for the above configuration:

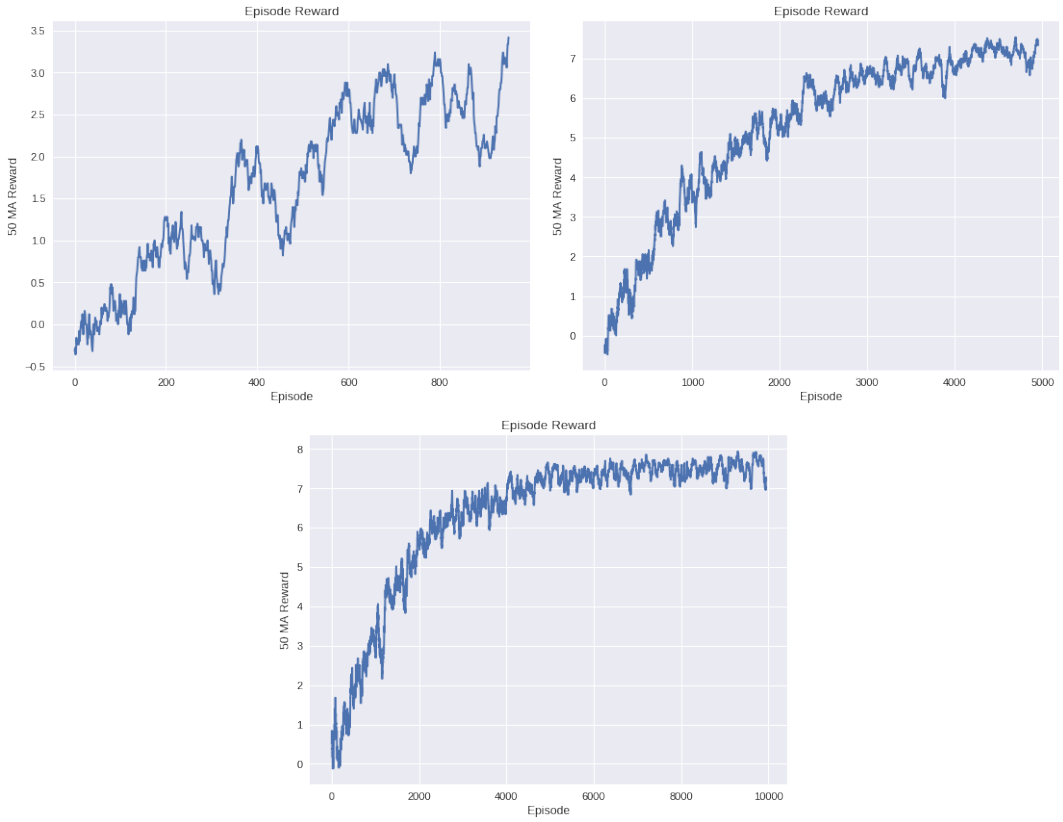

Figure 5: Mean Reward mapping for the above three configuration

4 Hyper-parameter Tuning

Here we tune different hyper-parameters and improve our model, achieving mean reward(of last 100 values) to 7.58 and learning time to 390s compared to the default metrics of 890s mean reward of 6.25, with our most optimum configuration.

Below are the configurations for which we tested our model:

4.0.1 Model-1

Changes made:

In below config, we're optimizing the exploration/exploitation trade-off by increasing the decay:

- Max-epsilon: **2** (default 1)
- Min-epsilon: **0.005** (default 0.05)
- Lambda: **0.0001** (default 0.00005)
- Learning rate for the DNN layers: **0.005** (default 0.00025)
- Epoch: 5000

Result: Epsilon decay, Mean reward and Learning time representation

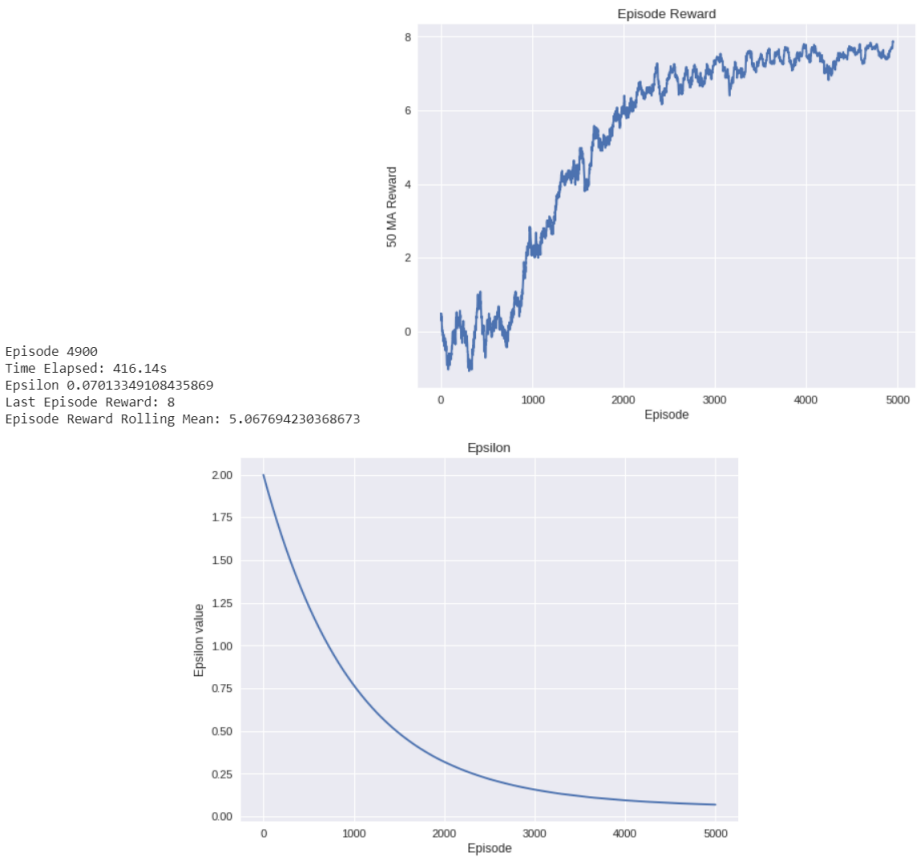

Figure 6: Epsilon Delay, Mean reward mapping and Learning time for Model 1

We observe the model has not improved and has rather performed worse from the default state, mostly due to bad balancing in exploration/exploitation rate

4.0.2 Model-2

Changes made:

In below config, we're optimizing the exploration/exploitation trade-off by increasing the decay:

- Max-epsilon: **1** (default 1)
- Min-epsilon: **0.01** (default 0.05)
- Lambda: **0.00055** (default 0.00005)
- Learning rate for the DNN layers: **0.1** (default 0.00025)
- Epoch: 5000

Result: Epsilon decay, Mean reward and Learning time representation

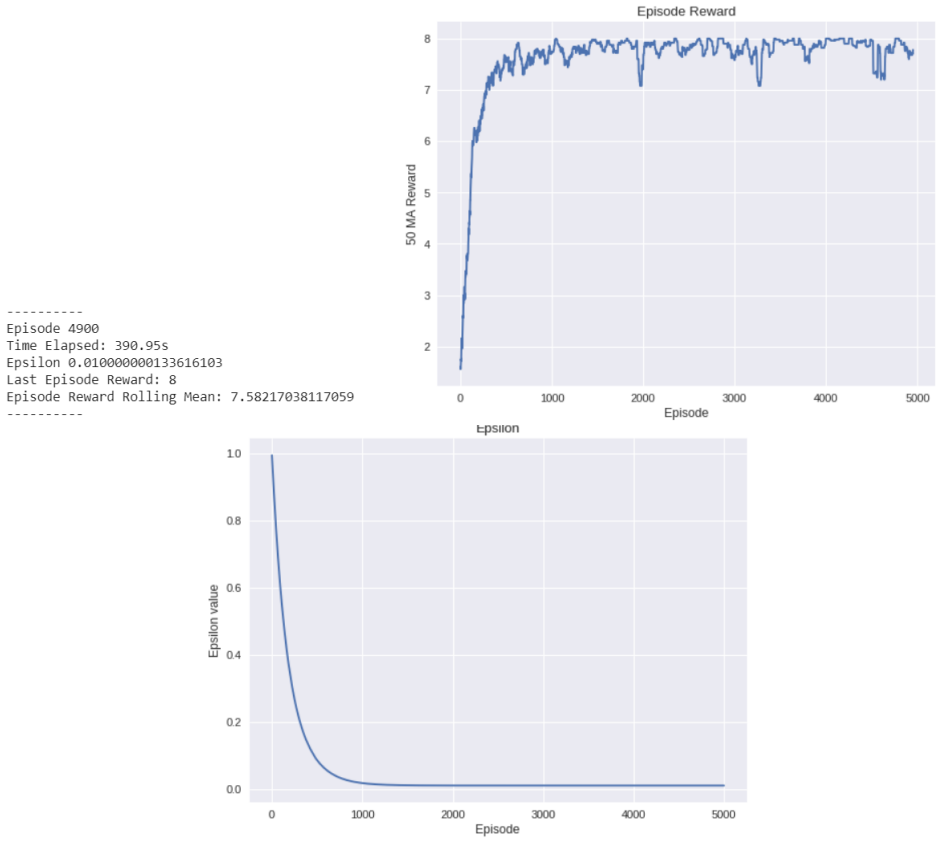

Figure 7: Epsilon Delay, Mean reward mapping and Learning time for Model 2

4.0.3 Model-3

Changes made:

In below config, we're optimizing the DNN part of the model, with added nodes and different activation function built over the Model 2 config:

- No. of Hidden node: 512 (default: 128)
- Activation function: tanh (default: relu)

Result: Epsilon decay, Mean reward and Learning time representation

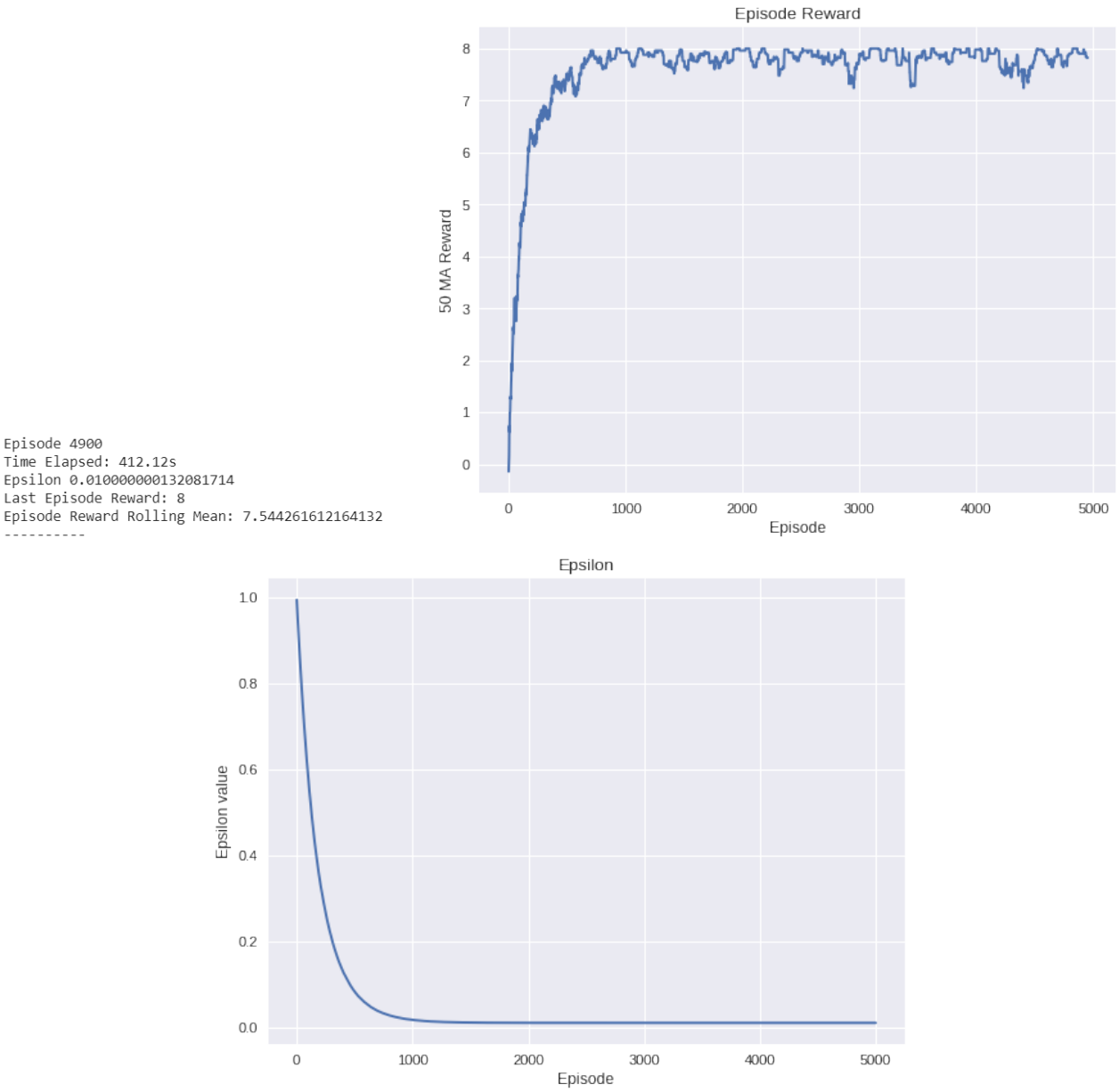

Figure 8: Epsilon Delay, Mean reward mapping and Learning time for Model 3

We see here that updating the activation function over the already optmized model of balanced exploration/exploitation model has given us comparable results but increase the learning time from 390s to 412s due to more number of hidden nodes.

Hence model 2 configuration has given us the best result

5 Writing Tasks

Now, we answer the questions part of the Writing task:

5.1 talks about the first question and 5.2 describes our step by step approach for populating the Q-values table.

5.1 Task-1

The Question asks us to explore the consequences of always choosing the action that maximizes the Q-value. If we look from a theoretical point of view, Q-learning requires that all the state-action pairs are (asymptotically) visited infinitely often. So it is possible that if from the start we force our agent to take only the actions leading to best Q-values, it get stuck in non optimal policies.
This is due to the fact that the agent has not explored enough states to optimally update the initial arbitrary Q-matrix i.e. find the best possible action from each state.

One e.g to showcase this point is the maze game where mouse is trying the end reward of cheese. It's end goal is to reach the ultimate cheese rewards.

Figure 9: The trade off with Exploration/Exploitation in a simple grid game

Right off the bat with an arbitrary Q matrix, the mouse might find the mini-paradise of three water sources clustered near the entrance, however with only choosing the best Q-values, it may spend all its time exploiting that discovery by continually racking up the small rewards of these water sources and never going further into the maze to pursue the larger prize.

Below we discuss two strategies to introduce more exploration in our model:

5.1.1 Introducing an exploration hyper-parameter ϵ

As have been done in our current DQN model, forcing an agent to explore can be achieved by simply introducing an exploration hyper-parameter.
We can choose to keep our */epsilon* constant which would be one method to force our agent to explore with every episodic iteration. However, the constant parameter will lead to delayed learning and on many occasions our agent might miss the best action. Another more robust approach for this is to **decay** our parameter with episodes: In the beginning, we'd set this parameter to 1 allowing our agent to explore all possible states and actions. We follow below formula to introduce the decay:

$$\epsilon = \epsilon_m in + (\epsilon_m ax - \epsilon_m in) * e^{-\lambda |s|} \tag{1}$$

As is evident from the above formula, the parameter is slowly decayed, and this rate is showcased by λ. The higher the lambda, greater will be the decay. Choosing an appropriate λ helps in balancing the exploration vs exploitation ratio.

5.1.2 Reward Curiosity

We reward our agent for not only moving towards the main target but also for exploring un-mapped territories. The key idea of this method is to store the agent's observations of the environment in an episodic memory, while also rewarding the agent for reaching observations not yet represented in memory. Being not in memory is the definition of novelty here seeking such observations means seeking the unfamiliar.
Such a drive to seek the unfamiliar will lead the artificial agent to new locations, thus keeping it from wandering in circles and ultimately help it stumble on the goal.

5.2 Task - 2

We present here a step by step calculation of the Q-matrix for the problem statement:

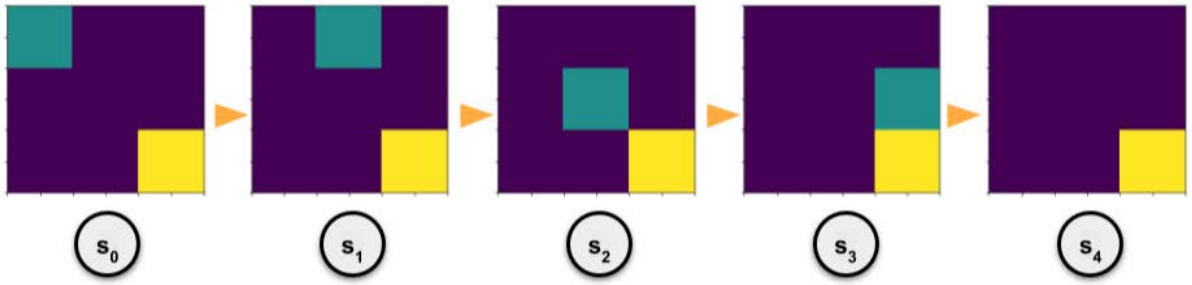

Figure 10: Different states and optimal path as stated in the problem statement

We begin from the last state:

State 4:

As stage 4 is the last stage, the Q-values will simply be 0 for all possible steps because we'll end our simulation upon reaching this stage and there would be no need for the agent to make any action.

State 3:

Calculation of Q-value for state 3 depends on the reward made for that step plus the maximum q-value for all other possible states as shown below: Since the action taken is $DOWN$ it suffices to calculate the Q-value for that action:

$$\begin{aligned}
Q(s_t = 3, a_t = DOWN) &= R_{s_3,Down} + \gamma * max(Q(s_4, UP), Q(s_4, DOWN), Q(s_4, LEFT), Q(s_4, RIGHT)) \\
&= 1 + 0.99 * max(0, 0, 0, 0) \\
&= 1 \\
Q(s_t = 3, a_t = UP) &= R_{s_3,UP} + \gamma * max(Q(s', UP), Q(s', DOWN), Q(s', LEFT), Q(s', RIGHT)) \\
&= -1 + 0.99 * max(0, 0, 0, 0) \\
&= -1 \\
Q(s_t = 3, a_t = LEFT) &= R_{s_3,LEFT} + \gamma * max(Q(s', UP), Q(s', DOWN), Q(s', LEFT), Q(s', RIGHT)) \\
&= -1 + 0.99 * max(0, 0, 0, 0) \\
&= -1 \\
Q(s_t = 3, a_t = RIGHT) &= 0 \qquad \text{(Since it's not possible to move RIGHT)}
\end{aligned}$$

State 2:

Calculation of Q-value for state 2 depends on the reward made for that step plus the maximum q-value for all other possible states (including state 3) as shown below:

$$\begin{aligned}
Q(s_t = 2, a_t = DOWN) &= R_{s_2,Down} + \gamma * max(Q(s', UP), Q(s', DOWN), Q(s', LEFT), Q(s', RIGHT)) \\
&= 1 + 0.99 * max(0, 0, 0, 0) \\
&= 1 \\
Q(s_t = 2, a_t = UP) &= R_{s_2,UP} + \gamma * max(Q(s', UP), Q(s', DOWN), Q(s', LEFT), Q(s', RIGHT)) \\
&= -1 + 0.99 * max(0, 0, 0, 0) \\
&= -1 \\
Q(s_t = 2, a_t = LEFT) &= R_{s_2,LEFT} + \gamma * max(Q(s', UP), Q(s', DOWN), Q(s', LEFT), Q(s', RIGHT)) \\
&= -1 + 0.99 * max(0, 0, 0, 0) \\
&= -1 \\
Q(s_t = 2, a_t = RIGHT) &= R_{s_2,RIGHT} + \gamma * max(Q(s_3, UP), Q(s_3, DOWN), Q(s_3, LEFT), Q(s_3, RIGHT)) \\
&= 1 + 0.99 * max(-1, 1, -1, 0) \\
&= 1.99
\end{aligned}$$

State 1:

Calculation of Q-value for state 1 depends on the reward made for that step plus the maximum q-value for all other possible states (including state 2) as shown below:

$$\begin{aligned}
Q(s_t = 1, a_t = DOWN) &= R_{s_1,Down} + \gamma * max(Q(s_2, UP), Q(s_2, DOWN), Q(s_2, LEFT), Q(s_2, RIGHT)) \\
&= 1 + 0.99 * max(-1, -1, 1, 1.99) \\
&= 1 + 0.99 * 1.99 \\
&= \mathbf{2.9701} \\
Q(s_t = 1, a_t = UP) &= 0 \qquad \text{(Since it's not possible to move UP)} \\
Q(s_t = 1, a_t = LEFT) &= R_{s_1,LEFT} + \gamma * max(Q(s_0, UP), Q(s_0, DOWN), Q(s_0, LEFT), Q(s_0, RIGHT)) \\
&= -1 + 0.99 * max(0, 0, 0, 0) \\
&= -1 \\
Q(s_t = 1, a_t = RIGHT) &= R_{s_1,RIGHT} + \gamma * max(Q(s', UP), Q(s', DOWN), Q(s', LEFT), Q(s', RIGHT)) \\
&= 1 + 0.99 * max(0, 0, 0, 0) \\
&= 1
\end{aligned}$$

State 0:

Calculation of Q-value for state 0 depends on the reward made for that step plus the maximum

q-value for all other possible states (including state 1) as shown below:

$$
\begin{aligned}
Q(s_t = 0, a_t = DOWN) &= R_{s_0,Down} + \gamma * max(Q(s_2, UP), Q(s_2, DOWN), Q(s_2, LEFT), Q(s_2, RIGHT)) \\
&= 1 + 0.99 * max(0, 0, 0, 0) \\
&= 1 \\
Q(s_t = 0, a_t = UP) &= 0 \qquad \text{(Since it's not possible to move UP)} \\
Q(s_t = 0, a_t = LEFT) &= 0 \qquad \text{(Since it's not possible to move UP)} \\
Q(s_t = 0, a_t = RIGHT) &= R_{s_0,RIGHT} + \gamma * max(Q(s_1, UP), Q(s_1, DOWN), Q(s_1, LEFT), Q(s_1, RIGHT)) \\
&= 1 + 0.99 * max(0, 2.97, -1, 1) \\
&= 1 + 2.94 \\
&= \mathbf{3.94}
\end{aligned}
$$

FINAL Q-TABLE

Please note that only values for the steps part of the optimal path have been included in the final Q-table

Q-Table for Optimal Path				
State	Up	DOWN	LEFT	RIGHT
0	0	0	0	**3.94**
1	0	**2.97**	0	0
2	0	0	0	**1.99**
3	0	**1**	0	0
4	0	0	0	0