# Project 4: Tom and Jerry in Reinforcement learning
## Introduction to Machine Learning

**Yash Narendra Saraf**
UB Person No. 50290453
MS Computer Science and Engineering
School of Engineering and Applied Science
ysaraf@buffalo.edu

## Abstract

In this project we are training an agent i.e. Tom to catch Jerry inside the environment, all this has been done using Reinforcement learning. The project introduces to interesting concepts like exploration and exploitation, Deep Q learning networks, Markov Decision Processes, and Bellman Equation. The problems of Reinforcement learning are solved in a way how we humans learn, i.e. a reward is given for performing a good action and negative reward for bad actions.

## 1 Introduction

Reinforcement learning refers to goal-oriented algorithms, which learn how to attain a complex objective (goal) or maximize along a particular dimension over many steps

A Markov decision process can be explained by the following diagram which illustrates that when an agent is in state $S_i$ , performs some action A, the environment rewards R to the agent based on the action and moves the agent to state $S_{i+1}$. Now this process keeps on repeating until and unless the agent reaches the final goal.

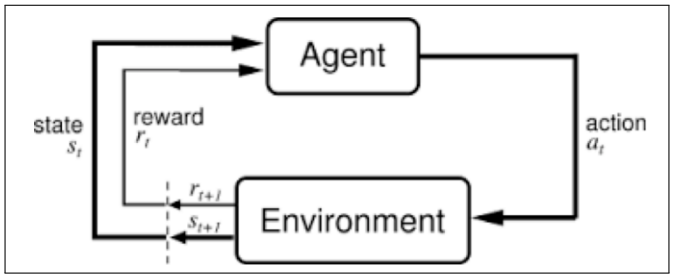


Figure 1: Markov Decision Process

Some important terms related to reinforcement learning are -

- **Agent**: An agent takes actions. The algorithm is the agent. In life, the agent is you. Over here the agent is Tom.
- **Action**: A is the set of all possible moves the agent can make. An action is almost self-explanatory, but it should be noted that agents choose among a list of possible actions. Over here out agent can take one of these 4 actions i.e move up, down, right, or left.
- **Discount factor**: The discount factor is multiplied by future rewards as discovered by the agent in order to dampen these rewards effect on the agents choice of action. It is designed

to make future rewards worth less than immediate rewards; i.e. it enforces a kind of short-term hedonism in the agent. Often expressed with the lower-case Greek letter gamma: . A discount factor of 1 would make future rewards worth just as much as immediate rewards. Were fighting against delayed gratification here.

- **Environment**: The world through which the agent moves. The environment takes the agents current state and action as input, and returns as output the agents reward and its next state. If you are the agent, the environment over here is a 5*5 Grid which returns the new coordinates of the agent.
- **State (S)**: A state is a concrete and immediate situation in which the agent finds itself; i.e. a specific instance of environment.
- **Reward (R)**: A reward is the feedback by which we measure the success or failure of an agents actions. This over here is is positive whenever the agent moves towards the goal. They effectively evaluate the agents action.
- **Policy** (): The policy is the strategy that the agent employs to determine the next action based on the current state. It maps states to actions, the actions that promise the highest reward.
- **Value (V)**: The expected long-term return with discount, as opposed to the short-term reward R. V(s) is defined as the expected long-term return of the current state under policy . We discount rewards, or lower their estimated value, the further into the future they occur. See discount factor.
- **Q-value or action-value (Q)**: Q-value is similar to Value, except that it takes an extra parameter, the current action a. Q(s, a) refers to the long-term return of the current state s, taking action a under policy . Q maps state-action pairs to rewards. Note the difference between Q and policy.

## 2 Problem Statement

The project combines reinforcement learning and deep learning. Task is to teach the agent to navigate in the grid-world environment. We have modeled Tom and Jerry cartoon, where Tom, a cat, is chasing Jerry, a mouse. In our modeled game the task for Tom (an agent) is to find the shortest path to Jerry (a goal), given that the initial positions of Tom and Jerry are deterministic. To solve the problem, we would apply deep reinforcement learning algorithm - DQN (Deep Q-Network), that was one of the first breakthrough successes in applying deep learning to reinforcement learning.

## 3 Code - Walk-through

- **Class environment** :Defining the environment i.e. the grid and the images for TOM, Jerry, and when tom catches jerry.
  - **_update_state** method updates the coordinates of tom based on action.
  - **_getreward** :Based on the action the _getreward function adds rewards based on manhattan distance.
    If the manhattan distance remains the same or increases then the reward is -1.
    If tom moves towards jerry i.e. the Manhattan distance decreases by 1 then the reward is +1.
    Also the reward is +1 when tom catches jerry i.e. the ultimate goal.
  - **_isover** method continuously monitors if the episode is over i.e. timeout or tom catches jerry.
  - **step** method calls all other methods and returns state, reward, and is the game over or not.
  - **render** function is used to create the image which can be later used to form the mp4 clip.
  - **reset** method is used to reset tom and jerry to their original positions.
    The deterministic variable which is by default True is used to reset the players at their original positions i.e. the top and bottom corner.
    By setting the variable as false both tom and jerry can be randomly spawned.

- Then to display the functionality of environment the Random actions division has been created.
- **Class Brain** :The brain of the network Brain class, Houses the Deep learning network.
  - Since we are working on Deep Q Networks the learning has been done using the Neural Network. For the above problem a 3 layer neural network has been built where the input parameters are the state_dim i.e. the positions of tom and jerry, i.e. the present state.
  - So here we have the **init_ method** which loads up the model for the network by calling the **_createmodel function**. Now we also have **_train. _predict and _predictOne functions** which have been called on the brain object later.
- **Class Memory** :Memory of the network - Stores the previous interactions with the environment.
  - Now all the agents experiences are saved as a tuple in samples. Now here we are saving 4 important parameters i.e.
    s: current state
    a: current action
    r: current reward
    s_: next state
  - Whenever the **sample method** is called the method returns any random n values stored in the samples array which stores the experiences as shown above.
- **Class Agent** :
  - On init_ the agent object has been initialized with all the parameters i.e. state_dim, action_dim, memory_capacity, batch_size, gamma, lambda, max_epsilon, and min_epsilon
  - Here also the brain object has been created and the memory object has been created using the above provided params.
  - The act method defines the policy of the agent.
  - Firstly based on the random number generation and comparison with current value of epsilon the agent decides whether he would exploit or explore.
    Now these two are important concepts.
    Exploration is taking random steps and going through the maze and learning the reward for that particular episode.
    But exploitation is using the currently trained model to see how well the model is performing and then updating weights.
  - So by common sense when the model is not at all trained we would like to explore and train the deep Q network. And as the model starts to learn exploitation should be more than exploration.
  - So to define that the epsilon greedy formula has been used to decrease the epsilon value exponentially. The exponential constant is a hyper parameter and can be adjusted.
  - So the act function returns any value 0-3 i.e. the action based on epsilon by either randomly predicting i.e. exploring or by predicting using the model i.e. exploiting.
  - The _observe method adds the sample to the memory and updates the epsilon value based on the exponential decay formula defined.
  - The replay function of the agent extracts some sample of data i.e. batch_size from the memory and then decides them again into 4 parts i.e.
    s: current state
    a: current action
    r: current reward
    s_: next state
  - The Q values for the current state and next state are predicted using the Brain classes predict function by passing both the current state and next state to the brain classs predict function.

- Then the Q values have been updated based on the Bellman equation i.e. defined below.
- This equation is an update of the current Q value based on given reward and the maximum Q for the next state.
- Now using the new Q values the model is trained.

- **main function**: So now in the main function we have an agent object which takes an action based on the current epsilon value and then updates the environment, which returns the next state and the reward, and whether the agent has succeeded or not. Based on this agent.observe function has been called which updates the epsilon and stores this instance in the memory.

  Then the agent.replays is called which trains the Deep Q network.

  Now every $1000^{th}$ episode has been saved and converted to mp4 to see how well the model gets trained.

# 4 Experiments

Here we have many hyper-parameters which can be tuned to obtain ideal results. For this we have experimented with the following parameters:

- number of episodes
- lambda
- gamma
- deterministic
- Deep neural network parameters

## 4.1 Number of episodes

Number of episodes is similar to the number of epochs parameter of the neural network, here each episode is one attempt by tom to catch jerry and the episode resets either when tom catches jerry or timeout.

So increasing the number of episodes increases the Rolling mean reward i.e. the average reward in last 100 episodes and then saturates based on on parameters.

The graph of how it effects the rolling mean has been showed. The other parameters are:

```
state_dim = 4
action_dim = 4 # left, right, up, down
MAX_EPSILON = 1 # the rate in which an agent randomly
                  decides its action
MIN_EPSILON = 0.05 # min rate in which an agent
                  randomly decides its action
LAMBDA = 0.001      # speed of decay for epsilon
num_episodes = Varied # number of games we
                    want the agent to play
gamma = 1 # discount factor
deterministic = True
```

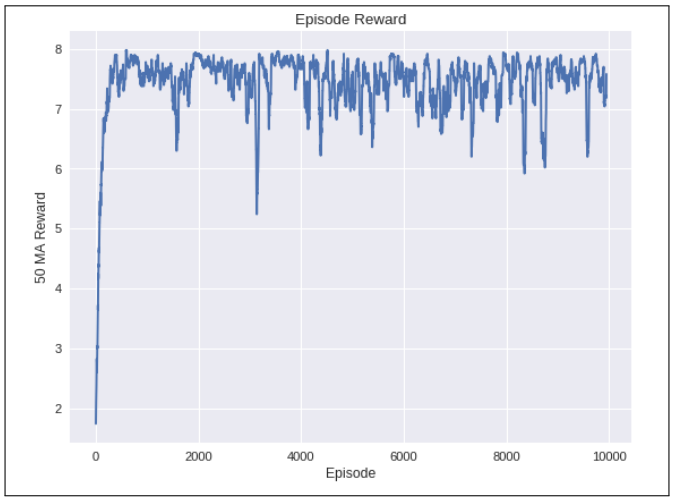


Figure 2: Tuning based on number of episodes

It is clear that with the above settings that after 2000 episodes the model saturates at rolling mean reward of 7.4.

### 4.2 lambda

The Lambda parameter is the exponential factor of the Epsilon equation, it essentially decides how steep the curve for epsilon decay should be. We start with a high epsilon value and keep on decreasing exponentially.

The decision whether to explore or exploit is taken using the random number and comparing it with epsilon. Initially as the model is not at all trained we choose to explore more than exploit and when the agent has explored a lot we will make agent exploit more than explore.

For this problem the agent learns the environment quickly as the environment is very small and having a very steep curve does not damage the rolling reward by a large factor. Equation for lambda decay is

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\lambda|S|}, \tag{1}$$

where$\epsilon \in (\epsilon_{min}, \epsilon_{max})$

i.e. $\epsilon \in [0.05, 1]$

The parameters used are following while varying lambda:

```
state_dim = 4
action_dim = 4 # left, right, up, down
MAX_EPSILON = 1 # the rate in which an agent randomly
                  decides its action
MIN_EPSILON = 0.05 # min rate in which an agent
                  randomly decides its action
LAMBDA = Varied       # speed of decay for epsilon
num_episodes = 1000 # number of games we
                    want the agent to play
gamma = 1 # discount factor
deterministic = True
```

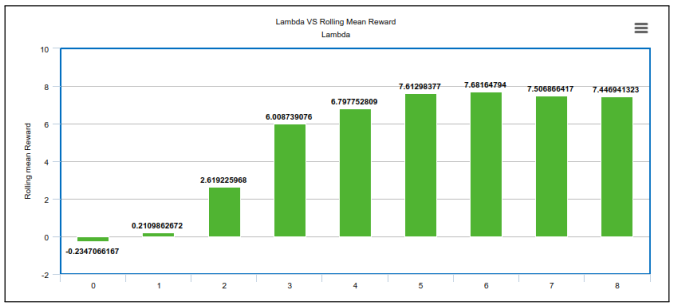


Figure 3: Tuning based on variation in lambda

For low number of episodes increasing lambda increases the rolling mean till a particular point and then drops as the model needs a proper balance between exploration and exploitation. This is a simple environment which does not require a lot of exploration. So having high gamma values do not harm as the minimum for epsilon has been set at 0.05. So even if the model explores for the initial few episodes then the model will perform well based on exploitation.

### 4.3 Gamma

The discount factor is multiplied by future rewards as discovered by the agent in order to dampen these rewards effect on the agents choice of action. It is designed to make future rewards worth less than immediate rewards; i.e. it enforces a kind of short-term hedonism in the agent. Often expressed with the lower-case Greek letter gamma: . A discount factor of 1 would make future rewards worth just as much as immediate rewards. Were fighting against delayed gratification here.

$$Q_t = r_t, \text{if episode terminates at step } t+1$$
$$Q_t = r_t + \gamma max_a Q(s_t, a_t; \Theta), \text{otherwise} \quad (2)$$

The parameters used are following while varying gamma:

```
state_dim = 4
action_dim = 4 # left, right, up, down
MAX_EPSILON = 1 # the rate in which an agent randomly
                  decides its action
MIN_EPSILON = 0.05 # min rate in which an agent
                  randomly decides its action
LAMBDA = 0.01      # speed of decay for epsilon
num_episodes = 1000 # number of games we
                    want the agent to play
gamma = Varied # discount factor
deterministic = True
```

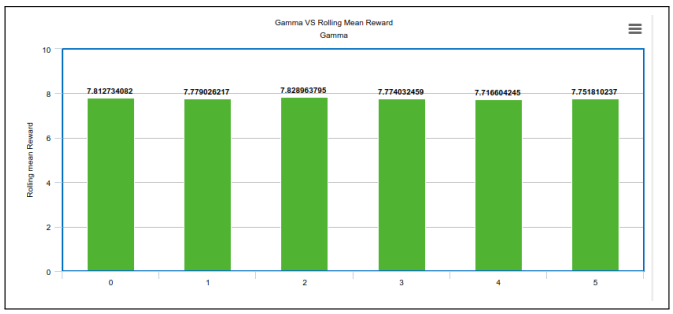


Figure 4: Tuning based on variation in gamma

For our model the varying gamma does not make any difference as there are no pitfalls i.e. no negative encounters, also the every short term goal lead to long term goal i.e. there is no short term goal which does not lead to long term goal. So having any gamma does not make any difference.

### 4.4 Deterministic

This provision has been added to make sure that the model is rigid and can be trained on the relative position of tom and jerry. When both of them are spawned at random positions still the model performs significantly well, which can be seen below.

The parameters used are following while varying gamma:

```
state_dim = 4
action_dim = 4 # left, right, up, down
MAX_EPSILON = 1 # the rate in which an agent randomly
                  decides its action
MIN_EPSILON = 0.05 # min rate in which an agent
                  randomly decides its action
LAMBDA = 0.01      # speed of decay for epsilon
num_episodes = 1000 # number of games we
                    want the agent to play
gamma = 0.99 # discount factor
deterministic = False
```

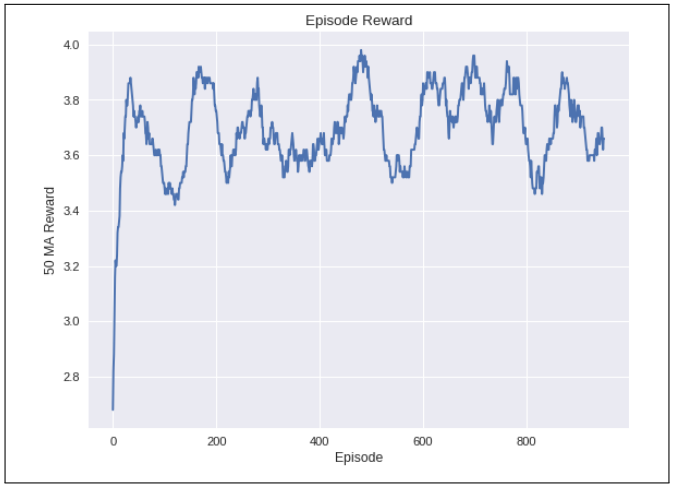


Figure 5: Rolling mean with random spawning

It can be seen that the rolling mean fluctuates between 3-4 as when both of them are randomly spawned the distance between them is relatively small and so the rewards agent can obtain also get limited so the rolling mean is less. This does not mean that the model is not performing well.

### 4.5 Deep learning network

The model needs to be complex enough to learn the relations between the state and the Q values. For this problem a simple neural network has been proposed with 2 hidden layers with relu and linear output layer. This is similar to a regression neural network which predicts the Q value for all four actions for that state.

The model has been defined like

```
model.add(Dense(units= 128, activation='relu', input_shape=(state_dim,)))
model.add(Dense(units= 128, activation='relu'))
```

```
model.add(Dense(units=action_dim, activation='linear'))
```

The parameters used are following while varying number of layers:

```
state_dim = 4
action_dim = 4 # left, right, up, down
MAX_EPSILON = 1 # the rate in which an agent randomly
                  decides its action
MIN_EPSILON = 0.05 # min rate in which an agent
                   randomly decides its action
LAMBDA = 0.01      # speed of decay for epsilon
num_episodes = 1000 # number of games we
                    want the agent to play
gamma = 0.99 # discount factor
deterministic = True
```

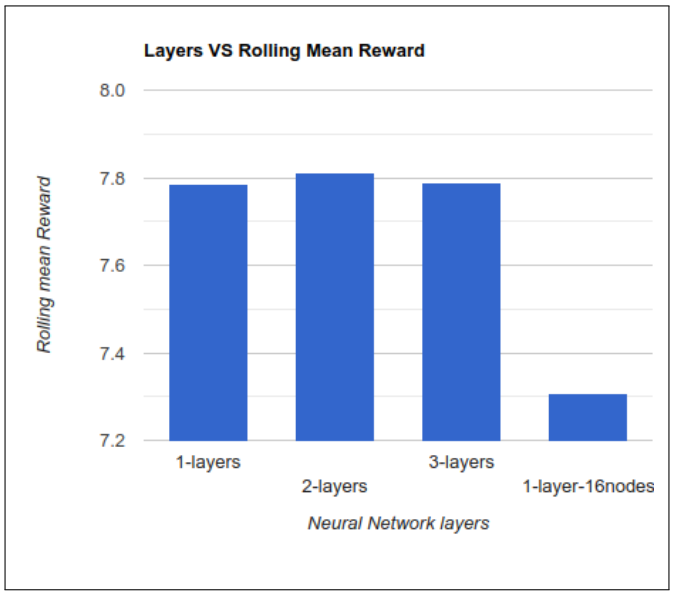


Figure 6: Rolling mean with different number of layers

Varying the number of layers does not make a lot of difference as the data is very simple and a single hidden layer model with 16 nodes also performs very well for the same data. As we are keeping deterministic as True we are fixing the jerry's position. So now we have only 25 spots for tom i.e. the number of states the environment has. So a simple NN is able to model that perfectly.

## 5 Report-deliverable

### 5.1 What parts have you implemented ?

For this project we have implemented 3 things.

**1.** The deep neural network using keras. The model takes the state i.e. the coordinates as the input and gives back the Q value. Now the selection of model has been analyzed in the above experimentation's section.

**2.** The Q updation equation i.e. how the value of Q gets updated. This equation is also known as the bellman equation and represents and important part of Q learning. This equation updates the Q values using dynamic programming, i.e. the Q value of a state action pair is the max over all Q values of the next state.

**3.** The epsilon decay curve. The epsilon decay curve has been used to create a balance between exploration and exploitation which are 2 important parts of model training. Initially the model needs to explore more than exploit so the epsilon value has been set to 1 i.e. maxima and then the value is exponentially decayed using the factor gamma. This allows the model to learn better.

### 5.2 What is their role in training the agent ?

First an agent object has been created which in turn creates a memory and brain objects and also interacts with the environment. Now the agent is interacting with the environment and then first checks whether it needs to explore or exploit based on the epsilon equation implemented.

If the model is going to explore the model takes a random action, updates it in the memory and samples a random set from the memory and first predicts the Q value using the deep neural network for current and next state, then the model updates the current Q values using the Q learning equation implemented. Then the model trains the network on new set of state -Q value pairs

If the model is going to exploit then then it uses the deep learning network implemented to predict the Q value and takes action based on that.

### 5.3 Can these snippets be improved and how will it influence the training of agent ?

The snippets can be modifies for example a complicated neural network can be used i.e. with more number of hidden layers. This over here does not seem necessary as the model gets trained perfectly using the provided neural networks model and there does not seem to be a lot of difference in the mean rolling reward.

Different epsilon decay curves can be used or some strategies like greedy epsilon can be implemented, but using the above provided epsilon decay works good as the number of states in the environment are very less the model does not require high complexity.

### 5.4 How quickly was your agent able to learn ?

After tuning the hyper-parameters the agent gives the rolling mean of more than 7.2 after 200 episodes and a rolling mean of about 7.85 after 1000 episodes.

## 6 Questions

### 6.1 Question-1

**Explain what happens in reinforcement learning if the agent always chooses the action that maximizes the Q-value. Suggest two ways to force the agent to explore. [20 points]**

**Part1:** In reinforcement learning two important concepts work together i.e. Q learning and Deep neural networks. The Q learning equation also represented by the bellman equation, the Q value for a particular (state,action) pair is the sum of reward based on that action and maximum Q value of the further state with respect to all actions.

In other words we can say that the agent would select an non-optimal policy and get stuck in the local minima. Now if we continuously just keep on increasing the Q value we will not allow of agent to explore. For exploration the agent picks up random action instead of picking up maximum Q value which allows the agent to explore the environment.

If the agent is not allowed to explore then the agent will be stuck on some path which it thinks is best but might not be the best given the environment.

So, it is important for agent to explore as well as exploit to get properly trained. This curve is adjusted using the epsilon equation.

**Part2:** One method to decide the curve between exploration and exploitation has been discussed in the code where we compare the epsilon to a random number and make a decision whether we have

to explore or exploit. Then epsilon here is decreased using the exponential decay formula. Based on the environment the decay formula can be set in many ways. A linear or step function can also be used. The most popular is exponential decay.

One of the other methodologies used is to maintain a count of how many times each state has been visited and deciding the path based on that. This can be massively large for large environments and might not be scalable.

Another way is to initialize the Q-table with random noise instead of 0's and then start the process as this will make the maximum value path to be random, it will in turn force the agent to explore. T

### 6.2 Question-2

**Calculate Q-value for the given states and provide all the calculation steps. [20 points]**

**Solution** Bellman equation used to update the **Q-table** is

$$Q(s_t, a_t) = r_t + \gamma * max_a Q(s_t + 1, a) \tag{3}$$

where $\gamma$ is 0.99

**Step 1** We will start calculating the **Q-values** from the last state.
Let's consider the $\mathbf{State} = \mathbf{S_4}$
We know that

$$Q(S_4, RIGHT) = 0$$

$$Q(S_4, LEFT) = 0$$

$$Q(S_4, UP) = 0$$

$$Q(S_4, DOWN) = 0$$

Consider the $\mathbf{State} = \mathbf{S_3}$

$$\begin{aligned} Q(S_3, RIGHT) &= 0 + 0.99 * max_Q(s_{23}, a) \\ &= 0 + 0.99 * 1 = 0.99 \end{aligned}$$

$$\begin{aligned} Q(S_3, LEFT) &= -1 + 0.99 * max_Q(s_{22}, a) \\ &= -1 + 0.99 * (max_Q(1 + 0.99 * max_Q(s_{23}, a)) \\ &= -1 + 0.99 * (1 + 0.99 * 1) \\ &= 0.9701 \end{aligned}$$

$$\begin{aligned} Q(S_3, UP) &= -1 + 0.99 * max_Q(s_{13}, a) \\ &= -1 + 0.99 * (max_Q(1 + 0.99 * max_Q(s_{23}, a)) \\ &= -1 + 0.99 * (1 + 0.99 * 1) \\ &= 0.9701 \end{aligned}$$

$$\begin{aligned} Q(S_3, DOWN) &= 1 + 0.99 * max_Q(s_{33}, a) \\ &= 1 + 0.99 * 0 \\ &= 1 \end{aligned}$$

Consider the $\mathbf{State} = \mathbf{S_2}$

$$\begin{aligned} Q(S_2, RIGHT) &= 1 + 0.99 * max_Q(s_{23}, a) \\ &= 1 + 0.99 * 1 = 1.99 \end{aligned}$$

$$
\begin{aligned}
Q(S_2, LEFT) &= -1 + 0.99 * max_Q(s_{21}, a) \\
&= -1 + 0.99 * (max_Q(1 + 0.99 * max_Q(s_{22}, a)) \\
&= -1 + 0.99 * (1 + 0.99 * 1.99) \\
&= 1.9403
\end{aligned}
$$

$$
\begin{aligned}
Q(S_2, UP) &= -1 + 0.99 * max_Q(s_{12}, a) \\
&= -1 + 0.99 * (max_Q(1 + 0.99 * max_Q(s_{13}, a)) \\
&= -1 + 0.99 * (max_Q(1 + 0.99 * (max_Q(1 + 0.99 * max_Q(s_{23}, a))) \\
&= -1 + 0.99 * (max_Q(1 + 0.99 * 1.99) \\
&= -1 + 0.99 * (2.9701) \\
&= 1.9403
\end{aligned}
$$

$$
\begin{aligned}
Q(S_2, DOWN) &= 1 + 0.99 * max_Q(s_{32}, a) \\
&= 1 + 0.99 * 1 = 1.99
\end{aligned}
$$

Consider the $\mathbf{State} = \mathbf{S_1}$

$$
\begin{aligned}
Q(S_1, RIGHT) &= 1 + 0.99 * max_Q(s_{13}, a) \\
&= 1 + 0.99 * 1.99 \\
&= 2.9701
\end{aligned}
$$

$$
\begin{aligned}
Q(S_1, LEFT) &= -1 + 0.99 * max_Q(s_{11}, a) \\
&= -1 + 0.99 * max_Q(1 + 0.99 * max_Q(s_{12}, a)) \\
&= -1 + 0.99 * max_Q(1 + 0.99 * max_Q(1 + 0.99 * max_Q(s_{13}, a))) \\
&= -1 + 0.99 * max_Q(1 + 0.99 * max_Q(1 + 0.99 * 1.99)) \\
&= -1 + 0.99 * max_Q(1 + 0.99 * 2.9701) \\
&= -1 + 0.99 * 3.940399 \\
&= 2.9009
\end{aligned}
$$

$$
\begin{aligned}
Q(S_1, UP) &= 0 + 0.99 * max_Q(s_{12}, a) \\
&= 0 + 0.99 * max_Q(1 + 0.99 * max_Q(s_{13}, a)) \\
&= 0 + 0.99 * max_Q(1 + 0.99 * 1.99) \\
&= 0 + 2.9403 \\
&= 2.9403
\end{aligned}
$$

$$
\begin{aligned}
Q(S_1, DOWN) &= 1 + 0.99 * max_Q(s_{22}, a) \\
&= 1 + 0.99 * 1.99 \\
&= 2.9701
\end{aligned}
$$

Consider the $\mathbf{State} = \mathbf{S_0}$

$$
\begin{aligned}
Q(S_0, RIGHT) &= 1 + 0.99 * max_Q(s_{12}, a) \\
&= 1 + 2.9403 \\
&= 3.9403
\end{aligned}
$$

$$
\begin{aligned}
Q(S_0, LEFT) &= 0 + 0.99 * max_Q(s_{11}, a) \\
&= 0 + 0.99 * max_Q(1 + 0.99 * max_Q(s_{12}, a)) \\
&= 0 + 0.99 * max_Q(1 + 2.9403) \\
&= 0 + 3.900897 \\
&= 3.9008
\end{aligned}
$$

$$
\begin{aligned}
Q(S_0, UP) &= 0 + 0.99 * max_Q(s_{11}, a) \\
&= 0 + 0.99 * max_Q(1 + 0.99 * max_Q(s_{12}, a)) \\
&= 0 + 0.99 * max_Q(1 + 2.9403) \\
&= 0 + 3.900897 \\
&= 3.9008
\end{aligned}
$$

$$
\begin{aligned}
Q(S_0, DOWN) &= 1 + 0.99 * max_Q(s_{21}, a) \\
&= 1 + 0.99 * max_Q(1 + 0.99 * max_Q(s_{22}, a)) \qquad = 1 + 0.99 * 2.9701 \\
&= 3.9403
\end{aligned}
$$

The Q-table would be:

| | RIGHT | LEFT | UP | DOWN |
|---|---|---|---|---|
| $\mathbf{S_0}$ | 3.9403 | 3.9008 | 3.9008 | 3.9403 |
| $\mathbf{S_1}$ | 2.9701 | 2.9009 | 2.9403 | 2.9701 |
| $\mathbf{S_2}$ | 1.99 | 1.9403 | 1.9403 | 1.99 |
| $\mathbf{S_3}$ | 0.99 | 0.9701 | 0.9701 | 1 |
| $\mathbf{S_4}$ | 0 | 0 | 0 | 0 |

**Table 1: Updated Q-Table**

## Conclusion

This project was the introduction to reinforcement learning where we understood how Markov decision processes work, the significance of bellman equation, Q learning and Deep Q learning.

In this a simple environment was trained and importance of each hyper-parameter was understood. How we use epsilon decay, designing of environment, agent(brain and memory), how they interact with each other was understood. For such a simple application we can simply use a single layer neural network, and exploit more than explore as the number of states are very less, and also ignore the gamma value as the long term goal and short term goal are both the same.

## References


[1] A Beginner's Guide to Deep Reinforcement Learning, *Skymind Article*
`https://skymind.ai/wiki/deep-reinforcement-learning`

[2] Keras Documentation
`https://keras.io/`

[3] Lecture 14 — Deep Reinforcement Learning, *Stanford University School of Engineering*
`https://www.youtube.com/watch?v=lvoHnicueoE&t=827s`