Tom and Jerry in Reinforcement Learning

Hrishikesh Nitturkar
University at Buffalo
hnitturk@buffalo.edu

Abstract

In this paper, we discuss an algorithm of Reinforcement Learning, 'Q-Learning', to train a machine to learn and solve a "problem" based on a finite set of states and actions and different rewards associated with each action. These rewards tell the machine which action is favourable and which action is not. Thus reinforcement learning is a machine learning techique that learns based on the interactions with the envrionment. In our example, we try to train our agent to follow an optimal path in a maze from source to reach the target. Every time the agent makes a move towards the target, it is rewarded and when it moves away from the target, the reward is subtracted.

1 Introduction

This project implements 'Q-Learning' that creates AI agents to learn by interacting with the environment. The AI agents learn using trail and error feedback from the environment. These feedbacks are rewards. They can be positive or negative, based on the action the agent chooses.

A 'Q-learning' algorithm consists of Environment, Brain, Memory and Agent. Environment is the base of the algorithm that helps in the training of the agents. It can be defined as the scope of the current algorithm being run. Environment provides feedback to the agents that try to solve a problem. Brain refers to the neural network that is used to process the inputs and calculate weights for each input and make precise predictions based on these weights. The brain is trained on the environment using different parameters so that it can make better predictions towards the solution. The brain makes the predictions based on trails on possible future values and also based on past experiences. It stores all of its past experiences in a memory which is used to predict the next steps. Finally, the agent is the player that executes the algorithm and trains the brain.

In our project, we solve a game where an agent tries to reach a target starting from a source. The source is Tom, the cat and the target is Jerry, the mouse. Firstly we start by defining the environment for our algorithm. We define a 5 X 5 grid. At each point, the agent has four actions to choose from - Up, Down, Left or Right. Before taking any step, the agent calculates the expected reward for each of the action and then chooses the action with the most optimal reward.

The brain of our model is a neural network. We implement it using keras library. The neural network consists of four layers. The input layer take 4 values for each sample, two describing the position of the source and two describing the position of the target. There are two hidden layers with 128 nodes in each layer. The first hidden layer gets 4 values as input and the second hidden layer get 128 inputs from the 128 nodes of the first layers. For each of the input layers we use the 'relu' as activation function. The output layer consists of the 'Q-values' for each of the action that can be selected or performed from the current state. The activation function used in the output layer is 'linear' since we have to get real values for each of the actions. We train on the input provided for only one epoch because the target values get updated as the agent explores new actions. We implement the nueral in our code as follows:

```
model.add(Dense(128, input_dim=self.state_dim, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(self.action_dim, activation='linear'))
```

We are using a (Root Mean Square Propogation) and a loss function function of mean squared error in our brain. Apart from functions that help the brain train and predict values, we have one more method that helps the brain make only one prediction which can be used later.

The memory of our model is a list that consists of all the experiences of the brain on the input data, which action produces what Q-value and what was the result of every choice made. We implement a method 'add()' in the Memory class to append every experience of the brain to the memory so that these can be used to make predictions. Each sample in the memory is an array of the form [s,a,r,s_] where s is the present state, a is the action performed in the state, r is the reward recieved for taking that action and s_ is the next state/step. The memory is of fixed size so that when it is full, to store the most recent episode of the game, it pops out the oldest experience. This is necessary to store the most recent data so that all the new possibilities that the brain explores can be stored successfully. These new possibilitites make the brain learn better than the initial ones.

The agent is a player that runs the algorithm and helps the brain learn all the possibiities of the game. Here a possibility is considered as the different actions that can be taken from a given state so that the reward from the state chosen will be the most optimal for the algorithm to complete successfully. The agent randomly selects an action based on the exploration rate, 'epsilon', to explore all possibilities at the beginning and start learning. If it is not selecting a random action, it will select an action that it predicts to give the maximum reward. We will decrease the number of random actions over the time by introducing a exponential decay formula for epsilon which is,

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

which is implemented as following in our code:

```
self.epsilon = self.min_epsilon + (self.max_epsilon -
self.min_epsilon) * math.exp(-self.lamb*abs(self.steps))
```

where λ is a hyperparameter and S is the number of steps. This is done so that the agent can take lots of random actions and explore the environment in the beginning. Then we decrease this factor to reduce the random actions and force agent to take actions based on training. The question that arises is why does the agent need exploration at all? In order for the agent to learn how to deal with all possible states in an environment optimally, it must be exposed to as many of those states as possible. The agent in a reinforcement learning problem only has access to the environment through its own actions. As a result, there emerges a chicken and egg problem: An agent needs the right experiences to learn a good policy, but it also needs a good policy to obtain those experiences. We have to balance this exploitation and exploration tradeoff.

To address this issue, we use epsilon-greedy method which states that when an action is selected in training, it is either chosen as the action with the highest q-value, or a random action. We start with epsilon at a very low value, such that there is a strong bias towards exploitation over exploration, favouring choosing the action with the highest q-value over a random action. However, random actions are still sometimes chosen.

The agent makes decisions on current state and predicts the future action to be taken from a set of four actions by using Q-values of each of the future states. The Q value of each state is the action-value quality function that helps determining the worth of an action in the long run. The Q-function is defined as follows:

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

which is implemented in the code as following:

```
if(st_next is None):
        t[act] = rew;
else:
        t[act] = rew + (self.gamma*np.amax(q_vals_next[i]))
```

where γ is the discounting factor that penalizes the rewards in the future. The Q-value is the quality value that is returned for the present state which is discounted based on γ and the maximum Q-values of the possible next states. It is only the reward if the algorithm ends in the next step. Here is γ is bound betweeen 0 and 1 so that it helps converge our algorithm. Reinforcement learning techniques can be used to solve Markov Decision Processes(MDP). An MDP provides a mathematical framework for modeling decision-making situations where outcomes are partly random and partly under the control of the decision maker. An MDP is defined via a state space S, an action space A, a function of transition probabilities between states (conditioned to the action taken by the decision maker), and a reward function.

In its basic setting, the agent takes an action, the environment changes its state and gets a reward. Then the agent senses the state of the environment, takes an action, gets a reward, and so on so forth. The state transitions are probabilistic and depend on the actual state and the action taken by the agent. The reward obtained by the agent depends on the action taken, and on both the original and the new state of the environment.

The objective is to find a policy such that the agent gets the maximum reward every time. Note that there is no limit on the time taken to complete the algorithm. Hence the algorithm has an infinite time horizon. This is usually called an MDP problem with a infinite horizon discounted reward criteria.

The reward is called discounted because $\gamma < 1$. If it was not a discounted problem i.e. γ=1, then the sum would not converge. All policies that have obtain on average a positive reward at each time instant would sum up to infinity. The would be a infinite horizon sum reward criteria, and is not a good optimization criteria. Hence the discount factor is always lies in between 0 and 1.

The agent has various methods that help the agent run the algorithm.

- act: The act method describes if the agent must select a random action based on epsilon. If not a random action, we predict the predict the Q-Values for the state, then take the action which maximizes those values.
- observe: In this method we pass a sample of (s, a, r, s_) format to this method and add it to the memory. Then modify the epsilon value using the exponential decay formula described above.
- replay: We take a batch from the memory and predict Q values for the states in the batch and Q values for the next states also. For every sample in the batch, we update the Q value of the action to be chosen using the Q value function descibed above. Then the state and the updated Q values are added to input vector and output vector respectively. Then the brain is trained on this input vector 'X' and output vector 'y'.

2 Algorithm

The agent runs the algorithm for a given number of episodes resetting the environment at beginning of every episodes and pass its return value to the agent's act() method. This returns an action and 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. If the episode is over, we break the loop and start over with another episode or else we continue with the next state as the current state.

First, we run a random simulation of 1 episode and we observe the behaviour and how the source moves towards the target to get a feel of the execution. The model does not converge but we see the movements of the source and how the agent explores the environment. After this, we start our main algorithm using all the hyperparamters.

We start the algorithm by setting the grid size as 5 X 5 and with the values of epsilon, γ, λ and number of episodes as 1,0.99, 0.00005 and 10000 respectively. We give a minimum rate,0.05, which the epsilon decays and a random action is selected. When we run the algorithm, we initialise the agent with these values and for every episode, we first reset the environment and get a initial state. We pass this state as an input to act method in agent and get an action for this step. Then we pass this action as an input to the step function of the environment, so we get next state, reward and a boolean to check if the episode is terminated. We give the current state, action, reward and next state (s, a, r, s_) as input to the observe method. Then the replay method of the agent is called so that the corresponding q values are calculated and the model is trained for each episode. The maximum reward for a 5 X 5 grid is 8. So, for 10000 epsiodes, we get the following observations.

Episode	Epsilon	Last Reward
1000	0.6265787811588737	4
2000	0.4028919729973172	6
3000	0.2695680011126189	8
4000	0.18833966264755125	4
5000	0.1377824842385145	8
6000	0.10630942819332628	8
7000	0.08624175780656329	8
8000	0.07342987657283959	8
9000	0.0651782272711976	8
10000	0.06028989128413769	8

If we plot a graph for epsilon change vs. number of episodes,

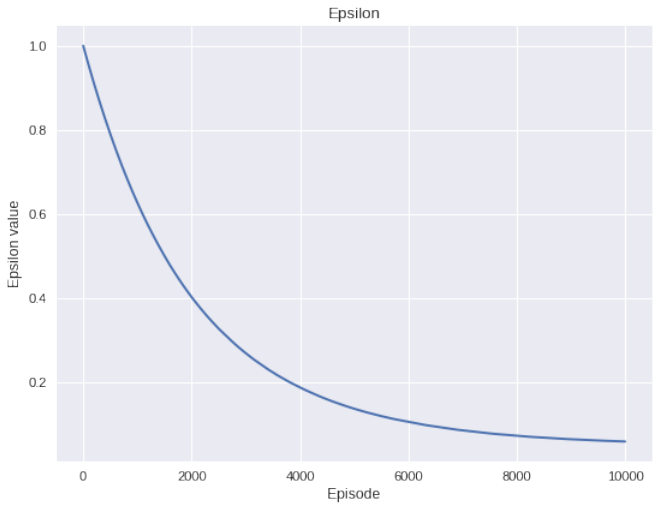


Figure 1: Episodes = 10000

Let us also plot the graph for moving average reward vs. number of episodes.

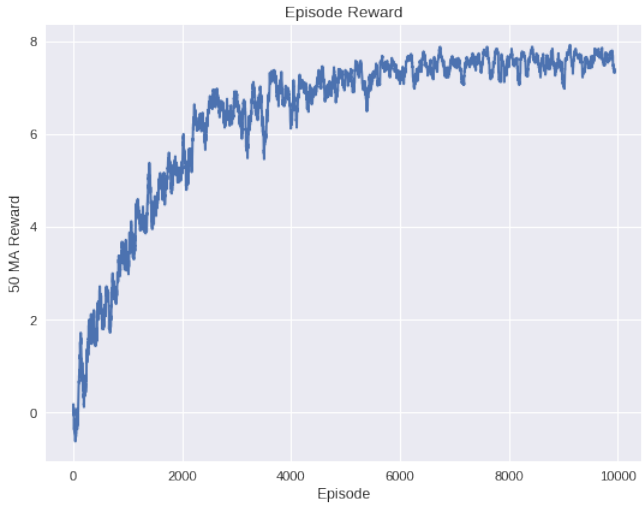


Figure 2: Reward vs Episodes

Our model learns after 8000 episodes and we get significant higher rolling reward i.e. above 6.0 after around 8000 episodes. We observe that epsilon decreases steadily and we get a final rolling average of 6.326905417814508 which is pretty good compared to the maximum reward the agent can get. This means that the model performed nicely according to the given constraints. Looking at the rewards graph, we can say that the rewards fluctuate but increase steadily throughout the run. The model converges i.e. the source reaches the target quite a few times in the 10000 episodes, that means the brain has learned a few paths to reach the target in optimal time and steps.

Let us change the number of episodes to 1000, we get the following values,

Episode	Epsilon	Last Reward
0	0.9995251187302109	0
100	0.9532613941793616	-2
200	0.9092088161589468	0
300	0.8673047077208147	2
400	0.8274442867669949	6
500	0.7896388189850636	4
600	0.7537069354354364	-2
700	0.7196210695762572	4
800	0.6871225253694981	4
900	0.6561406074160409	0

Graphically, we get

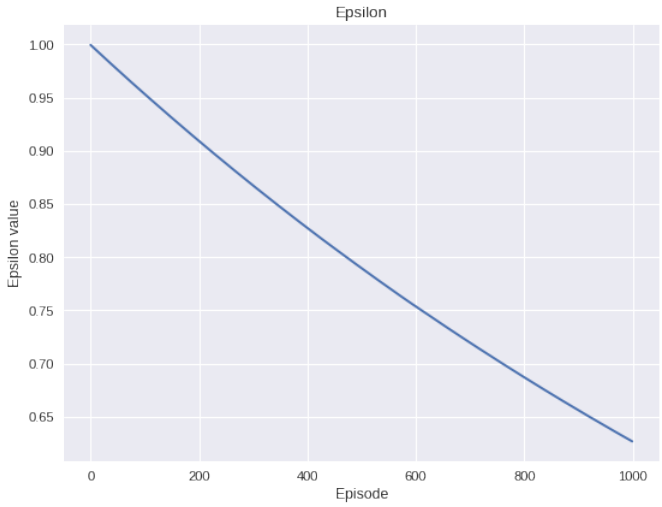


Figure 3: Episodes = 1000

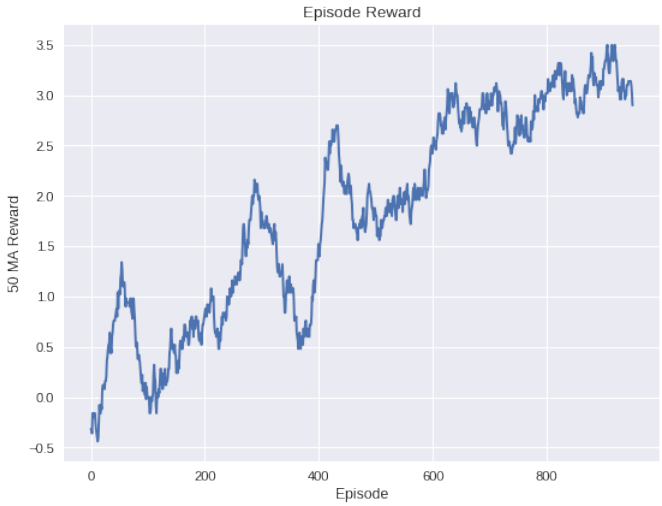


Figure 4: Reward vs Episodes

We observe that the model needs to be run more number of times to learn properly to make decision. In this scenario, the model makes more number of random actions. Hence it does not get sufficient time to train perfectly. The rewards fluctuate a lot which is evidence that the model cant guess which action to take. But the average of the rewards keep increasing indicating that the brain learns but not enough to make correct predictions everytime. More number of episodes might be able to fix this. So let us increase the number of episodes to 5000.

Episode	Epsilon	Last Reward
0	0.9995251187302109	2
500	0.7896388189850636	0
1000	0.6265499529405272	-2
1500	0.5001865194679347	4
2000	0.40211646402937645	6
2500	0.326652438749791	6
3000	0.26814543744590974	7
3500	0.22290840174456028	8
4000	0.18740901439608143	8
4500	0.15946030636570374	8
4900	0.1414380765897078	8

Graphically, we get

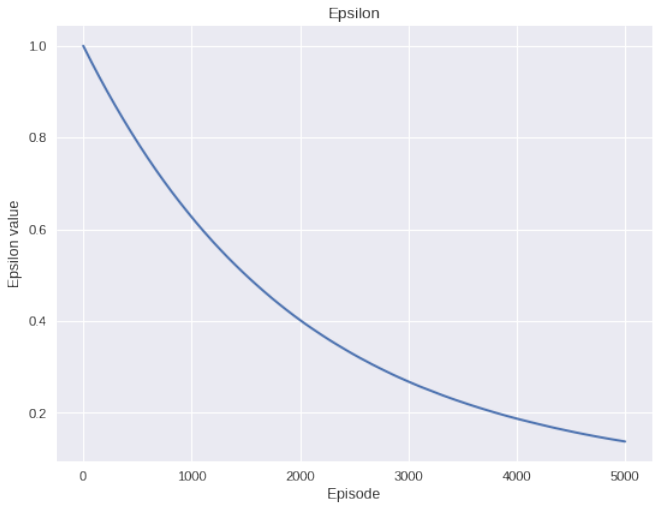


Figure 5: Episodes = 5000

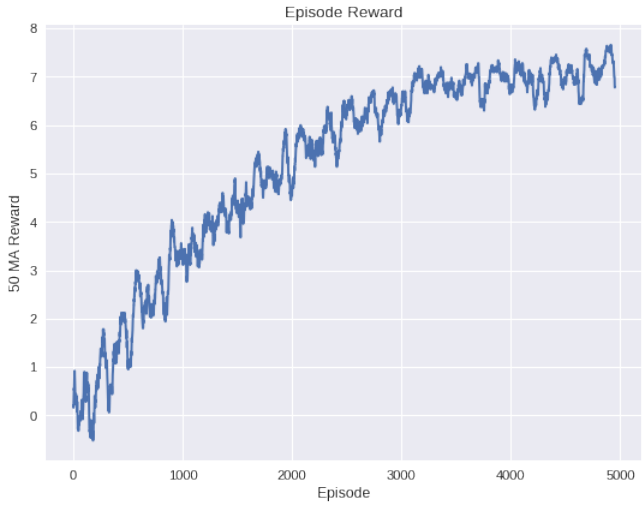


Figure 6: Reward vs Episodes

The model's brain learns better in this scenario and the epsilon value changes in a more better way. The fluctuations in the moving average of the rewards decreases indicating that the model is trained better and can make better predictions than before.

For number of episodes = 7000,

Episode	Epsilon	Last Reward
0	0.9995251187302109	0
700	0.719520633948681	6
1400	0.523268075894762	6
2100	0.38642054003222975	4
2800	0.29088315565010664	6
3500	0.2235493478500979	8
4200	0.1758023449751489	8
4900	0.1419791558558423	8
5600	0.11729338796145326	8
6300	0.0994425794235338	8
6900	0.08798768334147669	4

Graphically, we get

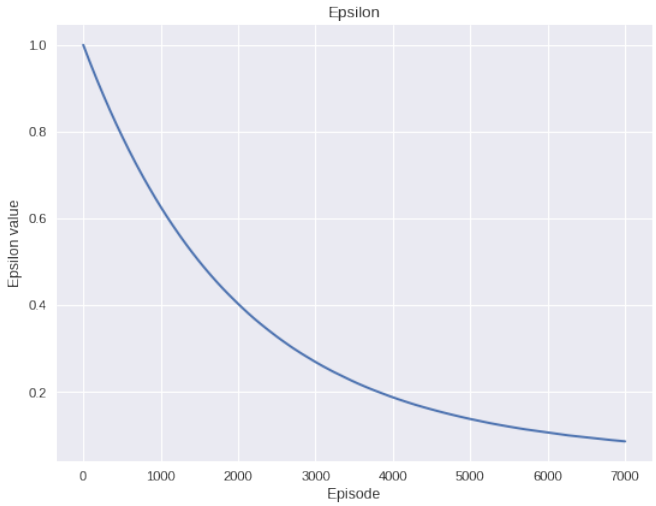


Figure 7: Episodes = 7000

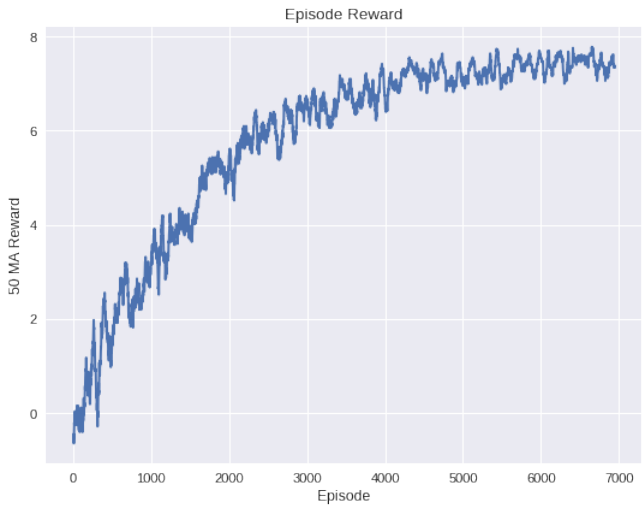


Figure 8: Reward vs Episodes

Let us increase the number of episodes to 12000, we get the following

Episode	Epsilon	Last Reward
0	0.9995251187302109	2
1200	0.5727545198654648	4
2400	0.3415501361324299	2
3600	0.21632804027691205	8
4800	0.14601572542799138	8
6000	0.10646731553138783	8
7200	0.08334513833236346	7
8400	0.06985412878194044	8
9600	0.06152031483814735	8
10800	0.056897175531886696	8
11900	0.05431527901717596	8

Graphically, we get

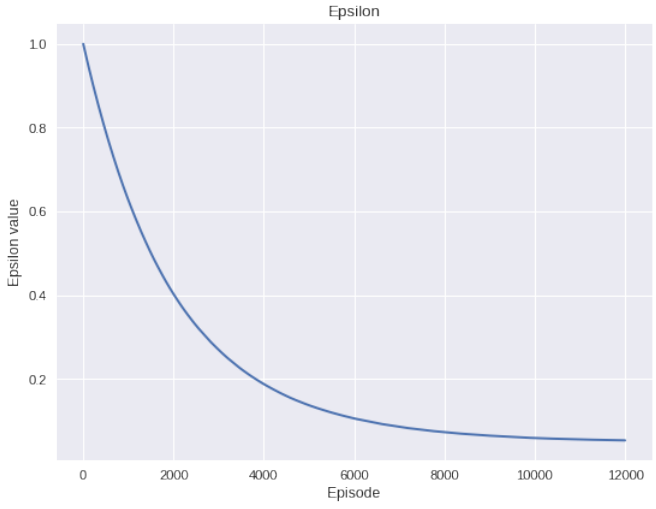


Figure 9: Episodes = 12000

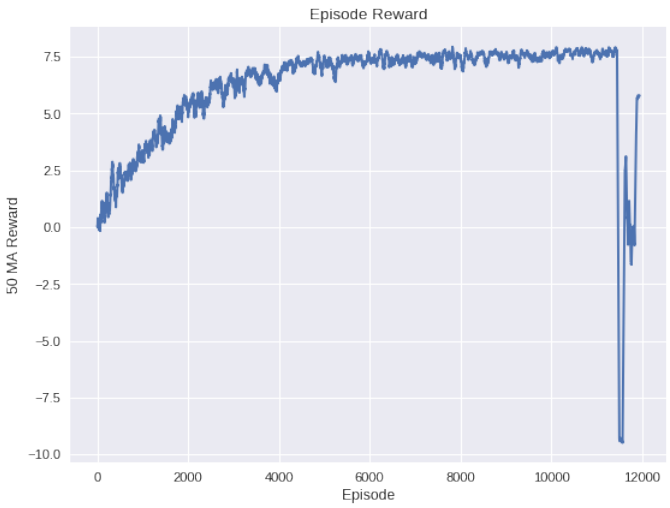


Figure 10: Reward vs Episodes

We observe that the moving average of the model falls sharply and then climbs back up. This maybe because the model took a random action which it did not take before. Hence it can be said that it was exploring the space. Additionally, the epsilon value decreases to the minimum epsilon value that we declare and begin to become a steady value for the rest of the run. Now let us explore the possibility that when we increase the number of episodes even more, the model makes better predictions take actions that always give him the maximum reward without random actions.

Now let us increase the number of episodes to 15000

Episode	Epsilon	Last Reward
0	0.9995251187302109	-4
1500	0.5002315403708891	6
3000	0.2692717843024517	8
4500	0.16000347812322546	7
6000	0.10647013896774972	8
7500	0.07928899899488771	8
9000	0.06534073923178615	8
10500	0.05807612343569021	8
12000	0.05426785766950317	8
13500	0.05218772454821631	8
14900	0.051204677606405964	8

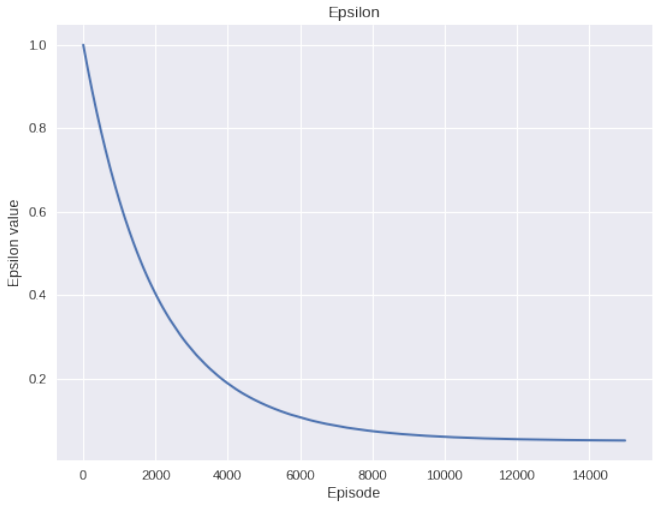


Figure 11: Episodes = 15000

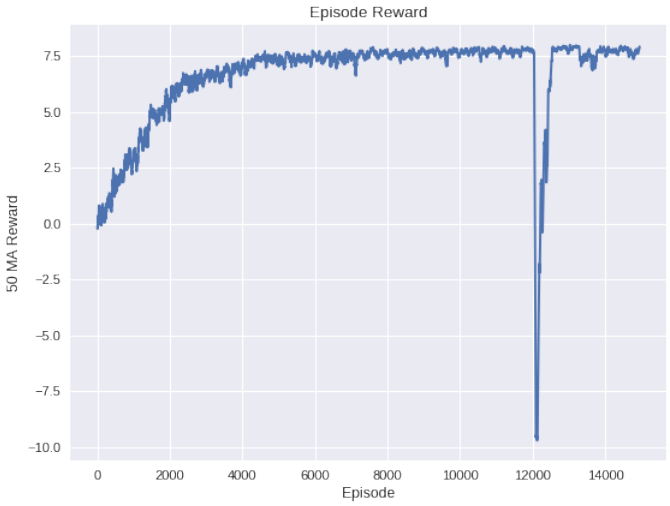


Figure 12: Reward vs Episodes

Here we observe that, our model has learned better that it makes a very good choice of action in every step so that it can always receive maximum reward of 8. The epsilon value decreases and remains constant near the end. The rolling average reaches around 6.52 which is a better case than any iteration before. Thus it can be said with increase in number of episodes, the rolling average of rewards will increase and the model learns better. As it reaches minimum ϵ value, we can conclude that the model takes every step according to what it has learned and the chance of a random selection of action decreases by a lot. The number of times the training converges is also increased.

Since we are using discounted reward, we use a discounting factor γ to predict the future actions. Let us decrease the gamma (γ) value to the values 0.89, 0.69, 0.39, 0.29 and observe the performance.

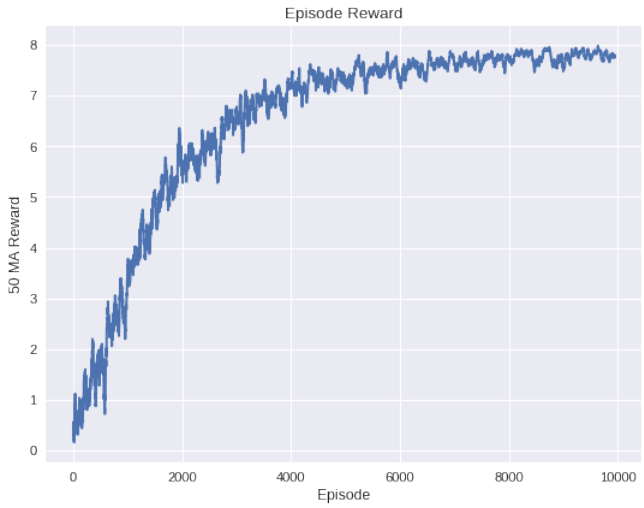


Figure 13: $\gamma = 0.89$

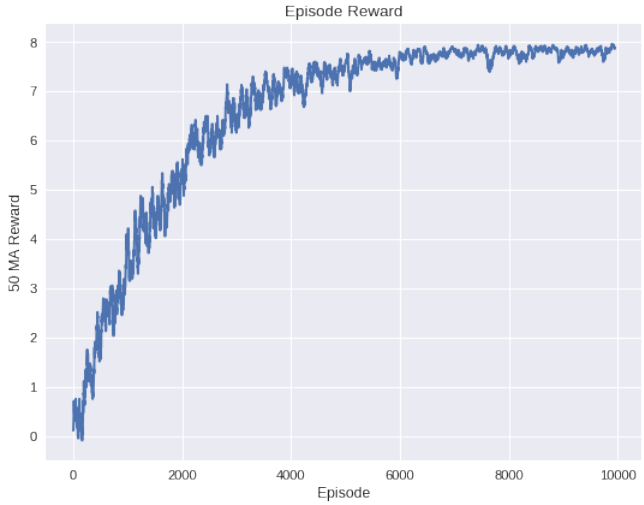


Figure 14: $\gamma = 0.69$

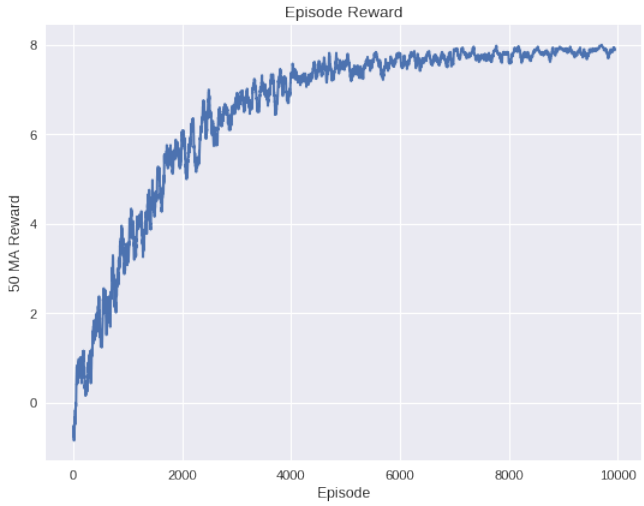


Figure 15: $\gamma = 0.39$

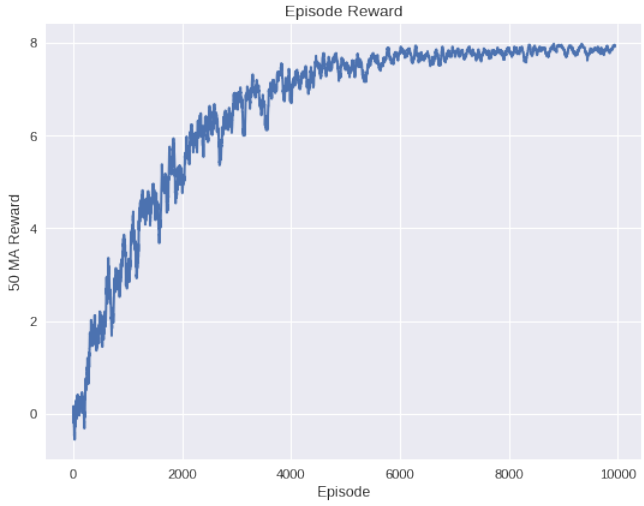


Figure 16: $\gamma = 0.29$

It can be observed that the value for rewards is changed just a little but there is a steady increase in moving average of the rewards. It can be seen that the rewards increase with stability from the beginning and follow a less fluctuating and increasing curve till a point where the agent makes the decisions that return the maxium reward. The rolling average of the model also increases significantly to around 6.48.

Evidently from the above graphs, it is clear that when the discount factor is decreased and brought closer to 0, the initial rewards are not discounted much and hence they are preferred. The model tends to give better weights to immediate rewards. This results in shallow lookahead meaning instead of having a better prediction that is beneficial over the entire run of the algorithm, the model only chooses the action that gives it best immediate reward and follows that action even if it turns out to be disadvantageous towards the end of the run. Hence it is important to keep the discount factor closer to 1 i.e. in the interval of [0.9,0.99] to prioritise the actions that result as a good choice for the algorithm to converge.

Now let us tune the number of nodes used in the neural network to study its effect on our model. Let us try changing the number of nodes to 512 and running our algorithm. We get the following results,

Episode	Epsilon	Last Reward
0	0.9995251187302109	4
2000	0.40333336377532736	2
4000	0.18728540191708432	8
6000	0.10564888182871807	6
8000	0.07303032865799379	8
10000	0.059660660151891684	8

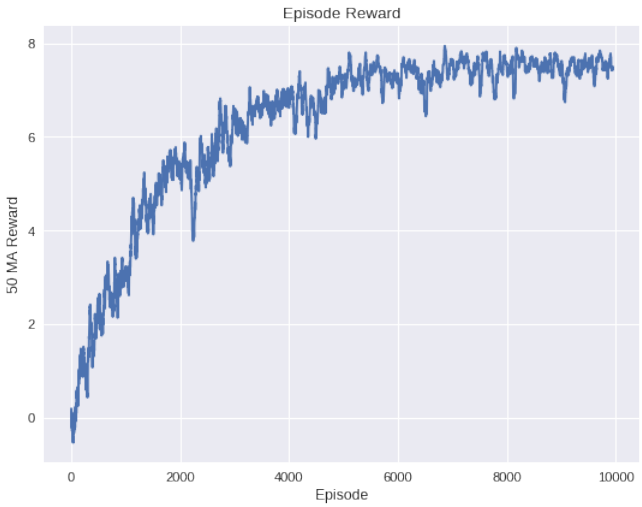


Figure 17: Reward vs Episodes

Let us change the number of nodes to 64 in each layer and we have

Episode	Epsilon	Last Reward
2000	0.4028214016600868	4
4000	0.18835349730553735	8
6000	0.10665963115549149	8
8000	0.07381134111448953	8
10000	0.060147343880530064	8

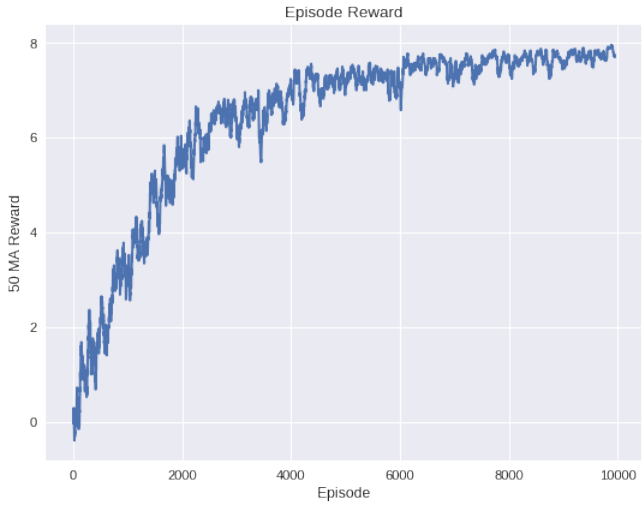


Figure 18: Reward vs Episodes

The rolling mean reward at the end of 10000 episodes is 6.412281587718413 which is greater than the previous case. Then lets change the number of nodes to 256,

Episode	Epsilon	Last Reward
0	0.9995251187302109	4
2000	0.40315674125275003	4
4000	0.18775296669330133	7
6000	0.10596978461057752	6
8000	0.07256188514644267	4
10000	0.059485003615754865	6

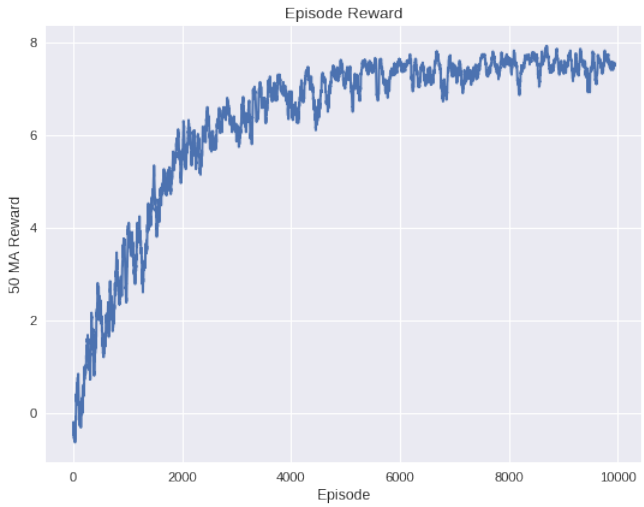


Figure 19: Reward vs Episodes

Now let us change the number of nodes in first and second hidden layer to 256 and 64 respectively and the study the behaviour. The model just learns a little quickly and makes better choices of actions.

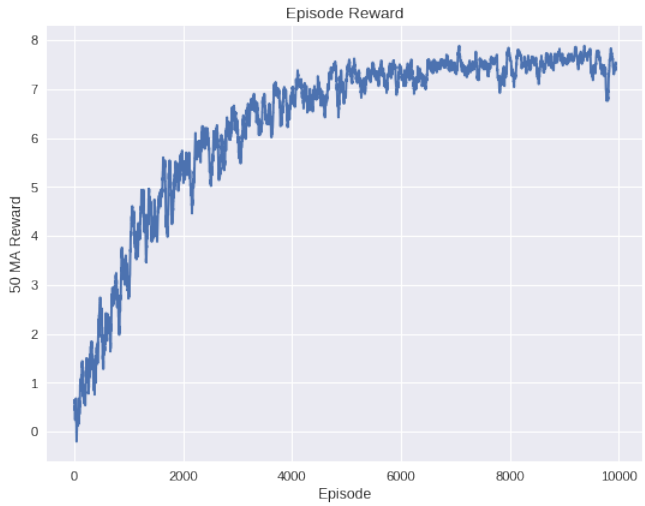


Figure 20: Reward vs Episodes

From the above results we can observe that the model learns better wth less number of nodes so that every input is processed through the network thoroughly which is an expected behaviour of the network. Hence the key to train the brain of the model, the neural network, properly is optimizing the number of nodes in our network so that the model learns best and as quickly as possible.

Now let us vary the parameters that determine the value of epsilon for every run. Let us change the speed of decay of epsilon i.e. lambda value from 0.00005 to 0.005. Let us keep the other parameters constant as before. We generate a random number in the range [0.0,1.0) and compare it to epsilon. If this random value is less than the epsilon value, the agent take random action otherwise the agent exploits its memory and predicts the next state. The random numbers generated are generally smaller than epsilon in the beginning resulting in more random actions but as the epsilon value decays, the number of random actions is reduced.

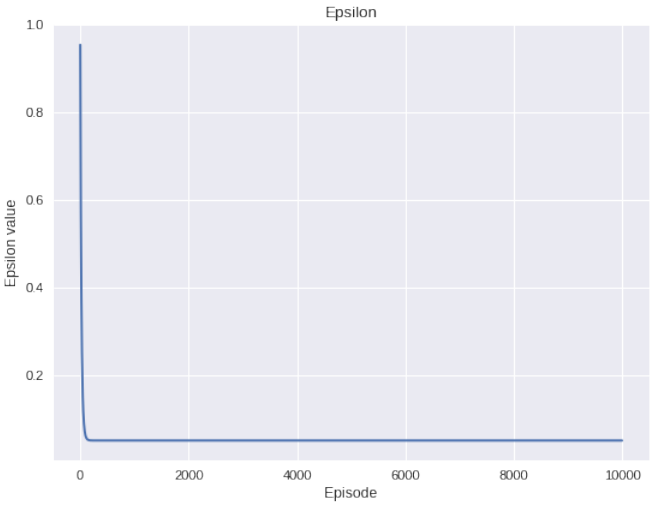


Figure 21: Epsilon vs Episodes

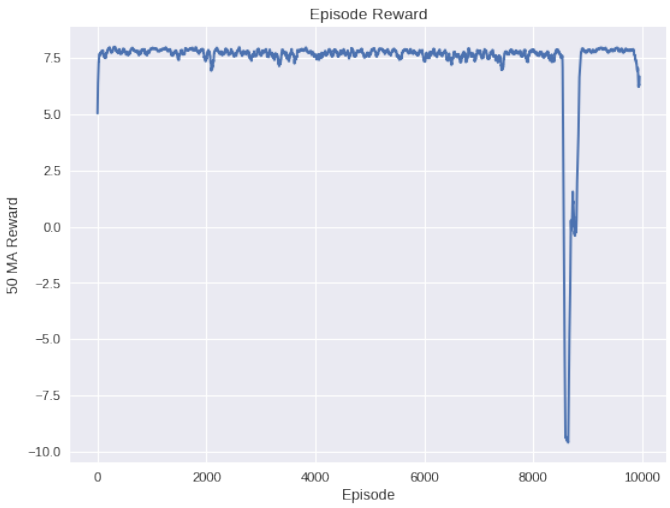


Figure 22: Reward vs Episodes

We can see that, when the epsilon decay speed value is increased, the model explores only for short amount of time and then it starts exploiting the memory or experiences. Hence, as a result, we can see that when the epsilon decays to the minimum value, all the actions are exploited and very less actions are randomized which results in a significant increase in the rolling average of the reward to greater than 7 very early in the training. This whole sceanrio may result in agent exploring less number of possible paths i.e states and actions that can lead him from source to target and the learning or training diminishes. Then it uses this learned knowledge to predict the future states. So it may not be aware of better optimal paths and use only the few paths that it has learned over its training.

Now let us consider decreasing the original epsilon value by a factor of 10 to 0.000005 and observe the behaviour.

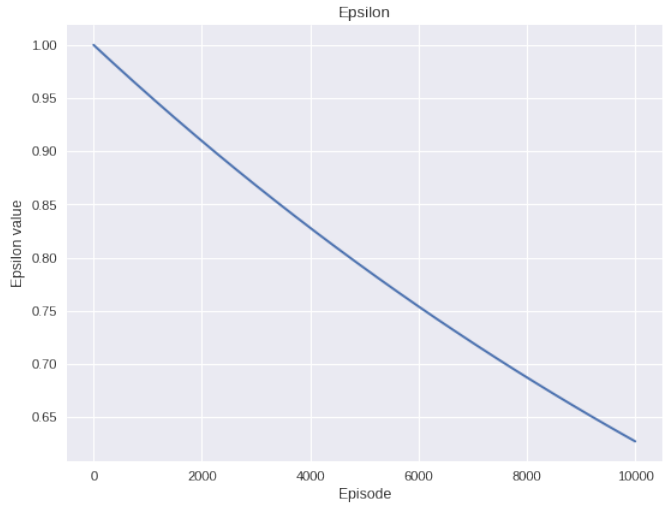


Figure 23: Epsilon vs Episodes

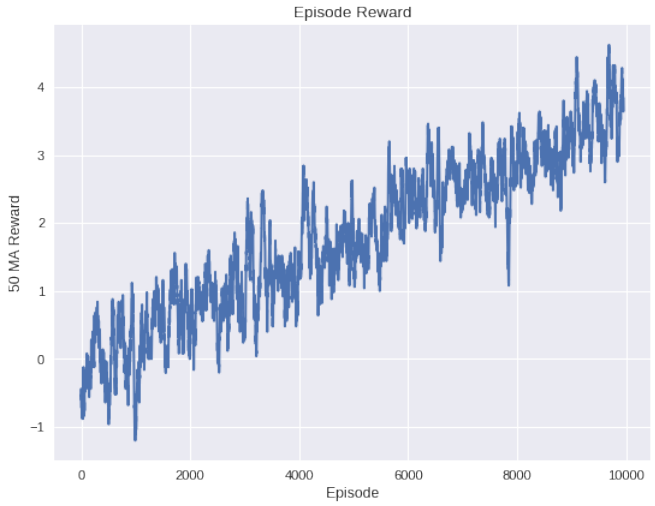


Figure 24: Reward vs Episodes

As clear from the graphs, when the epsilon is reduced by a huge factor, it requires relatively large number of episodes to learn from the training. This is because the epsilon decays at a much slower rate and this results in large number of random actions. As a result, the rolling mean does not increase to a significant value even after 10000 episodes.

Let us experiment with the maximum and minimum values of epsilon and observe its effects on the model. Let us set the minimum epsilon value to 0.01. We get the following observations.

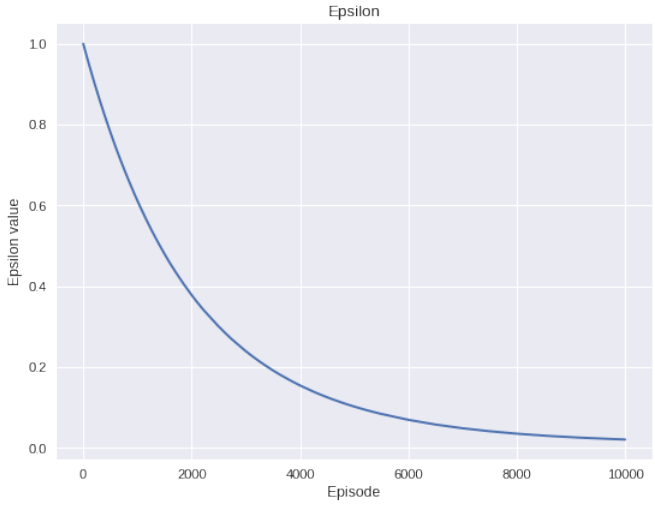


Figure 25: min_epsilon = 0.01

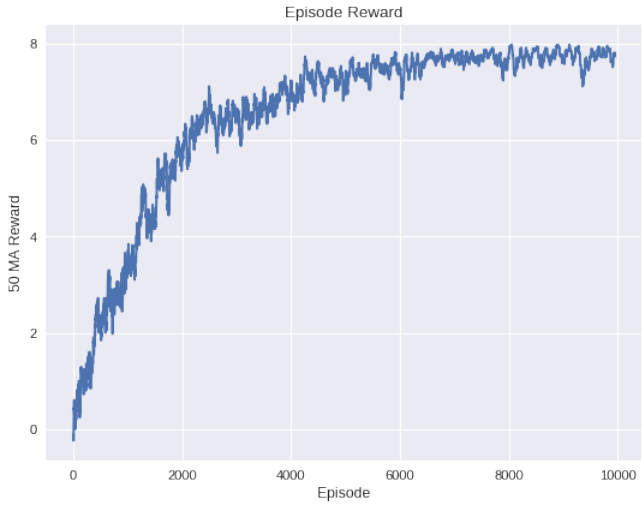


Figure 26: min_epsilon = 0.01

Now let us increase the minimum value of epsilon to 0.1, we observe,

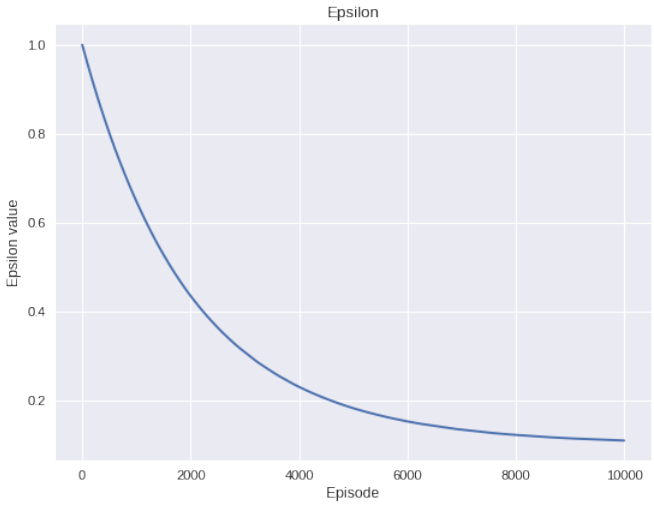


Figure 27: min_epsilon = 0.1

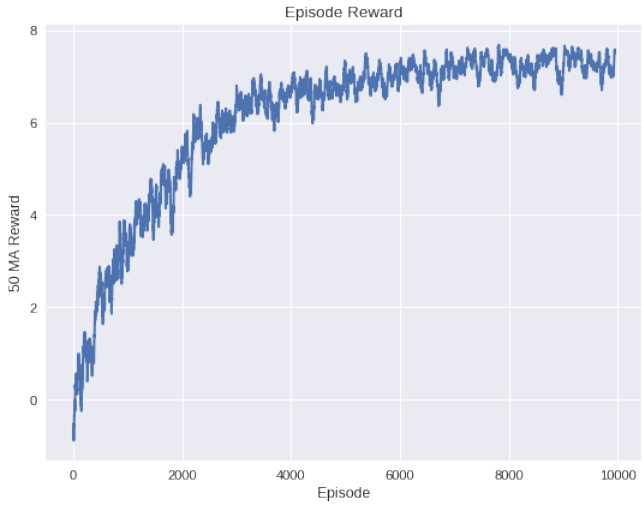


Figure 28: min_epsilon = 0.1

This means that the agent learns better when the epsilon decays more and the rolling average of 6+ is reached after 8000 episodes. But when we dont let the epsilon decay too much, the learning slows down and the rolling average of reward does not reach the best value after 8000 episodes. It takes upto 10000 episodes to reach the highest value.

Let us set the maximum epsilon value to 0.5 and observe the behaviour of the agent

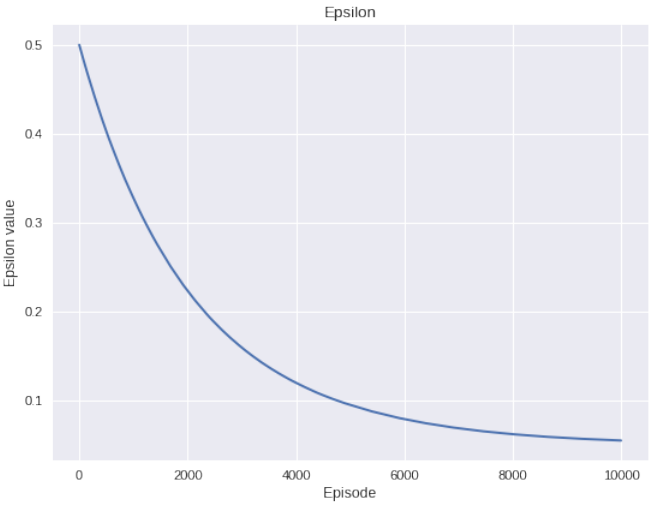


Figure 29: max_epsilon = 0.5

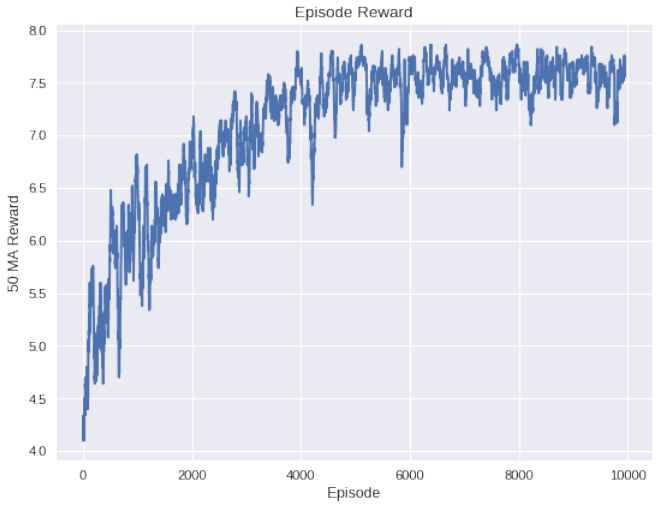


Figure 30: max_epsilon = 0.5

When we decrease the upper limit of the epsilon, the random values generated to check whether to take random action or predict states mostly is greater than the epsilon value, hence the agent predicts more values and we observe that the learning is improved by a significant factor and the rolling average increases to more than 7. Let us replicate these observations by setting the upper limit of epsilon to 0.4.

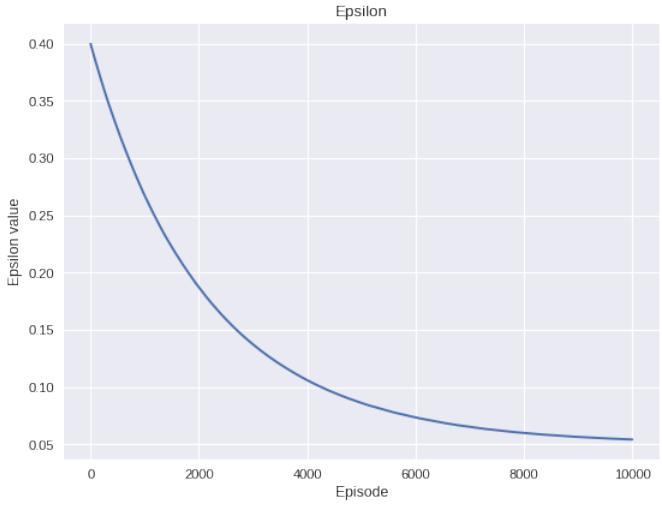


Figure 31: max_epsilon = 0.4

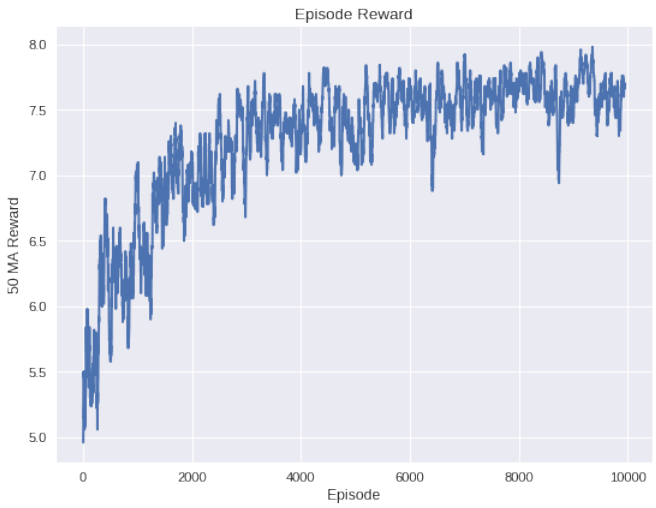


Figure 32: max_epsilon = 0.4

3 Observations

In this project, we have altered many hyperparameters that have a significant effect on the model's predictions. Each hyperparameter has a different effect on the predictions. It is important to configure the values of these hyperparameters for the model to make accurate predictions.

The number of episodes effect the model significantly. Our model learns better and makes When the model is trained on more number of episodes, it has the ability to explore more number of optimal solutions, some of which may be better than the rest. Hence more number of episodes contribute towards better learning of the model. But it is also important to regulate the number of episodes since after the agent discovers all the possible paths, the rest of the episodes are executed on the exploitation basis but few still may include random actions. We regulate this exploration vs exploitation situation based on the epsilon value.

For the model to learn and make as many less random choices as possible, it is really important to regulate the value of epsilon, that helps the agent make a random decision or predict the next step and we implement a decay algorithm for the same. We notice that when the speed of decay is increased, the algorithm starts predicting future decision from whatever it has learned very early in the algorithm. This means that the agent has less scope for exploring the environment. It will stick to whatever it has learned about the optimal path and use it to make future state and action predictions. This may not be favourable for any algorithm with many possible states and actions since the agent may not find all possible optimal states and actions. Also, if we decrease the speed of the decay, we observe the agent makes more random choices and keeps learning about the environment. As a result, we get a graph of fluctuating graph of rewards which is increasing but not to the point we want it to. Hence by running the algorithm relatively less number of times, the agent learns little about the environment. Hence it is important to regulate the number of the episodes the algorithm is run for. If we have very low decay speed, we use higher number of episodes abd vice versa.

The model uses the Q-value to predict the future action based on the expected discounted future reward. We discount the reward using γ value which lies between 0,1. It is observed that when the gamma value is decreased i.e. $\gamma- > 0$, the model starts favouring immediate rewards even if they are are useful in the long term. The discounting factor is used to help the algorithm converge. when the γ value decreases, the model starts exploiting whatever it has learned to maximize the rewards it recieves for each state and action. It does not care if this state or action will help it reach the solution along an optimal path. Since we are discounting the expected future rewards, the discounting factor lies between 0 and 1. So, to get optimal rewards, we try to keep the discounting factor, γ in the range of [0.90, 0.99]. This will help the agent increase the foresight of the model and consider the steps that will actually help in the long run.

When we decrease the minimum value of epsilon allowed for our algorithm, we observe that the model learns a little faster than before and the rolling average of 6 achieved just a little earlier than before but around the 8000 episodes. When we increase the minimum value of epsilon, we observe that the model learns slowly and that the highest reward of 8 is never recieved. So it is beneficial for the algorithm to let the epsilon decay more so that our agent makes nmore random actions and learn about the environment. If we alter the upper limit of the epsilon, the agent predicts more values and the rewards earned are also significant. But this may mean that the agent is learning little and making decisions based on whatever it has learned. It is to be noted that, the agent explores less when the room for exploration is reduced.

So, in conclusion, the neural network implemented for q-learning in this project requires optimal tuning of all the hyperparameters as mentioned above. Like in any other algorithm, we may have to find a balance between speed and accuracy of the model. This is important since trading one for another might be detrimental to the purpose of the algorithm. Hence fine tuning the algorithm hyperparamers like number of episodes, γ, λ, minimum and maximum values of ϵ, speed of decay of epsilon etc. is very important to optimize speed and accuracy of the algorithm.

We can improve the model by altering the number of nodes in each hidden layer in the neural network. Processing of data ensures getting better results for each weighted input. Number of hidden layers in the neural network can also be increased since the accuracy of the neural network increases when the input is passed through many hidden layers and weights for each input are altered more accurately. One other way to increase efficiency is to replace the exponential decay of epsilon with linear decay because if let the epsilon value decay slowly, the model makes more random choices and the chances of training of the model increase significantly.

4 Writing tasks of the project

4.1 Task 1

When the agent always chooses the action that maximizes the Q-value, the agent always chooses the optimal path to reach the target from the source. Q carries memory from the past and takes into account all future steps. As the agent uses up all the random actions and stores the path it took from that point on to reach the target in the memory, when it arrives at the same state or action again, it will follow the path that it has already stored to reach the target. Hence it wont try to randomize the future predictions to learn. It eventually learns the optimal Q-values for all possible state-action pairs. Then it can derive the action in every state that is optimal for the long term. Hence except for the initial state, all states are linked to each other based on the q-values. So, after the initial state, in whichever direction the agent proceeds, it tries to find only the optimal path to reach the target. As a result, the agent follows same fixed paths to reach the target. To force our agent to explore the space, we can do the following:

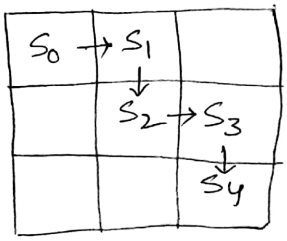
1. Use a epsilon-greedy approach: In this method, a decaying variable epsilon ranging between 1 and 0 is used, which decays over time while the agent learns. The agent will choose whether or not to predict an action from the training, based on the epsilon. For example, if epsilon value is 0.4 at a state, then the agent has a 0.4 probability of randomly selecting an action, and 0.6 probability of choosing the action by making predictions based on its training. This essentially forces the agent to explore the space by ignoring the policy. As a result, the agent explores the environment every time the random value produced is less than the epsilon value. As the epsilon decays over time, the random value generated gets larger than epsilon and then the agent starts predicting the values of the next states. If the agent makes a prediction, it will choose the action that maximizes the Q-value of the state.

2. Using Boltzmann Approach: In Boltzmann exploration approach, Instead of always taking the optimal action, or taking a random action, we choose an action with weighted probabilities. We use softmax over the estimated Q values of each action. In this case the action which the agent estimates to be optimal is most likely, but not guaranteed, to be chosen. Unlike epsilon greedy, actions are weighed by their relative value. This way the agent can ignore actions which it estimates to be largely sub-optimal and give more attention to potentially promising, but not necessarily ideal actions.

4.2 Task 2

WRITING TASK - 2

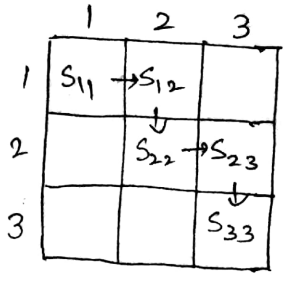
Given optimal Path:



Initialize a table as follows

	U	D	L	R
S_0				
S_1				
S_2				
S_3				
S_4				

Let us denote the states in the form of a matrix representation



	U	D	L	R
S_{11}				
S_{12}				
S_{22}				
S_{23}				
S_{33}				

Let us start calculating Q_1-values for the last state i.e. S_{33}.

As S_{33} is the last state, the goal is reached. Hence all Q-values for state S_{33} are zero - 0.

$Q(S_{33}, U) = Q_1(S_{33}, D) = Q_1(S_{33}, L) = Q(S_{33}, R) = 0$

Let us calculate the Q-values for state S_{23}.

$Q(S_{23}, a) = \text{Rew} + \gamma \cdot \max_{a'}(Q_1(\text{next\_state}, a))$

where $a = \{U, D, L, R\}$ and $\gamma = 0.99$

① $Q_1(S_{23}, U) = -1 + 0.99(\max(S_{13}, a'))$

Here Rew = -1, since the source is moving away from the target state S_{13} can have a maximum Q value when it is moving towards the target. so it gets rew = 1 for next step and discounted reward for the next step where rew = 1 and maximum of final state is '0'.

Hence $Q_1(S_{23}, U) = -1 + 0.99(1 + (0.99) \times 1)$

$= -1 + 0.99(1.99) = \underline{0.9701}$

② $Q_1(S_{23}, D) = 1 + 0.99(\max(S_{33}, a'))$

$= 1 + 0.99(0) = \underline{1}$

③ $Q(S_{23}, R) = 0 + 0.99(\max(S_{23}, a'))$

$= 0 + 0.99(1 + 0.99(\max(S_{33}, a')))$

$= 0.99(1 + 0.99(0))$

$= 0.99$

④ $Q(S_{23}, L) = -1 + 0.99(\max(S_{22}, a'))$

Here from state S_{22} we have two paths that can maximize the Q value. They are first go right and go down or go down and then go right.

Therefore $Q(23, L) = -1 + 0.99(1 + 0.99 \overset{\max}{\wedge}(S_{32}, a'))$

$= -1 + 0.99(1 + 0.99(1 + 0.99 \max(S_{33}, a')))$

$= -1 + 0.99(1 + 0.99(1 + 0.99(0)))$

$= -1 + 0.99(1 + 0.99(1))$

$= -1 + 0.99(1.99) = 0.9701$

Here we consider the state that the source can be in next step and intuitively predict what the maximum Q value of next state should be.

Now let us calculate the Q-values for the state S_{22}.

① $Q(S_{22}, U) = -1 + 0.99(\max(S_{12}, a'))$

$= -1 + 0.99(1 + 0.99 \overset{\max}{\wedge}(S_{13}, a'))$

$= -1 + 0.99(1 + 0.99(1 + 0.99 \max(S_{23}, a')))$

$= -1 + 0.99(1 + 0.99(1 + 0.99(1)))$

$= -1 + 0.99(1 + 0.99(1.99)) = -1 + 0.99(1 + 9701)$

$= 1.9403$

② $Q(S_{22}, L) = -1 + 0.99(\max(S_{21}, a))$

$= -1 + 0.99(1 + 0.99(\max(S_{31}, a)))$

$= -1 + 0.99(1 + 0.99(1 + 0.99(\max(S_{32}, a))))$

$= -1 + 0.99(1 + 0.99(1 + 0.99(1 + 0.99(\max(S_{33}, a)))))$

$= -1 + 0.99(1 + 0.99(1 + 0.99(1 + 0.99(0))))$

$= 1.9403$

For the above two states we get follow the action and reach a state. from this state, we make predictions as to what actions will get us closer to target and also in turn, maximize the reward and Q values. Then we proceed along that path predicting the same way for other states.

③ $Q(S_{22}, R) = 1 + 0.99 \max(Q(S_{23}, a'))$

$= 1 + 0.99 \max(Q(S_{13}, 0))$

$= 1 + 0.99(1) = 1.99$

④ $Q(S_{22}, D) = 1 + 0.99(1 + 0.99 \max Q(S_{32}, a'))$

$= 1 + 0.99(1 + 0.99(\max(S_{33}, a')))$

$= 1 + 0.99(1 + 0.99(0))$

$= 1 + 0.99(1 + 0.99(0)) =$

$= 1.99$

Now we calculate Q-values for the state S_{12}.

① $Q(S_{12}, L) = -1 + 0.99 (\max Q(S_{11}, a'))$

$= -1 + 0.99 (1 + 0.99 (1 + 0.99 \max (S_{31}, a')))$

$= -1 + 0.99 (1 + 0.99 (1 + 0.99 (1 + 0.99 \max (S_{32}, a'))))$

$= -1 + 0.99 (1 + 0.99 (1 + 0.99 (1 + 0.99 (1 + 0.99 (\max (S_{33}, a') + 1)))))$

$= -1 + 0.99 (1 + 0.99 (1 + 0.99 (1 + 0.99 (1 + 0.99 (1 + 0)))))$

$= 2.9008$

② $Q(S_{12}, R) = 1 + 0.99 (\max Q(S_{13}, a'))$

$= 1 + 0.99 (1 + 0.99 \max Q(S_{23}, a'))$

$= 1 + 0.99 (1 + 0.99 (1))$

$= 2.9701$

③ $Q(S_{12}, D) = 1 + 0.99 (\max Q(S_{22}, a'))$

$= 1 + 0.99 (1.99)$

$= 1 + 1.9701 = 2.9701$

④ $Q(S_{12}, U) = 0 + 0.99 (\max Q(S_{12}, a'))$

$= 0 + 0.99 (1 + 0.99 (\max Q(S_{22}, a'))$

$= 0.99 (1 + 0.99 (1.99))$

$= 0.99 (1 + 1.9701) = 0.99 (2.9701)$

$= 2.9403$

Now let us calculate the Q-values for state S_{11}

~~① $Q(S_{11}, R)$~~

① $Q_1(S_{11}, R) = 1 + 0.99 \max Q(S_{12}, a')$

$= 1 + 0.99 (2.9701)$

$= 3.940399 \Rightarrow 3.9404$

② $Q(S_{11}, D) = 1 + 0.99 \max Q(S_{21}, a')$

$= 1 + 0.99 \max (S_{22}, a')$

$= 1 + 0.99 (1.99)$

$= 1 + 1.9701 = 2.9701$

③ $Q_1(S_{11}, U) = 0 + 0.99 \max Q(S_{11}, a')$

$= 0 + 0.99 \max (1 + 0.99 (\max Q(S_{12}, a')))$

$= 0.99 (1 + 0.99 (2.9701))$

$= 0.99 (1 + 2.9403) = 3.9009$

④ $Q(S_{11}, L) = 0 + 0.99 \max Q(S_{11}, a')$

$= 0.99 (1 + 0.99 (\max Q(S_{12}, a')))$

$= 0.99 (3.9403) = 3.9009$

In each state, we recursively find Q values for each next state in the optimal path that we consider towards the target until we reach the target. we consider +1 reward for a step towards the target; -1 reward for a step away from the target and '0' reward for a step into an edge. For any state, if an action results in source going out of the boundary of environment, we consider that the state remains same, ~~but~~ so the reward changes to '0'. Hence the final Q-table would look like,

	U	D	L	R
S_{11}	3.9009	2.9701	3.9009	3.9404
S_{12}	2.9403	2.9701	2.9008	2.9701
S_{22}	1.9403	1.99	1.9403	1.99
S_{23}	0.9701	1	0.9701	0.99
S_{33}	0	0	0	0

Let us rename our states back to the following.

$S_{11} = S_0$

$S_{12} = S_1$

$S_{22} = S_2$

$S_{23} = S_3$

$S_{33} = S_4$. we get

The final Q-Table for the above problem is,

	U	D	L	R
S_0	3.9009	2.9701	3.9009	3.9404
S_1	2.9403	2.9701	2.9008	2.9701
S_2	1.9403	1.99	1.9403	1.99
S_3	0.9701	1	0.9701	0.99
S_4	0	0	0	0

Acknowledgments

This report is supported by the entire teaching faculty of CSE Department for the course CSE 574 - Introduction to Machine Learning for Fall 2018 semester

References

[1] Reinforcement Learning - *https://www.cse.unsw.edu.au/ cs9417ml/RL1/tdlearning.html*

[2] Diving Deeper into Reinforcement Learing with Q Learning, - *https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe*

[3] Understanding the role of discount factor - *https://stats.stackexchange.com/questions/221402/understanding-the-role-of-the-discount-factor-in-reinforcement-learning*