CSE 574: Project 4

Tom and Jerry in Reinforcement learning

11/30/2018

Mrudula Y
UB Person No: 50290843

I. PROBLEM STATEMENT

The project is intended to teach an agent to play a game using reinforcement learning and deep learning. Here, the game is a Tom and Jerry cartoon game where the agent Tom is chasing the goal Jerry in a grid-world environment. The aim is to make Tom reach Jerry in the shortest possible path when the initial positions of Tom and Jerry are given and are deterministic. The algorithm applied to solve this problem is Deep Q-Network algorithm (DQN) which applies deep learning to reinforcement learning and is a Deep Reinforcement Learning Algorithm.

II. UNDERSTANDING THE CODE

A. ENVIRONMENT

The environments which are in the form of a grid are called grid-world environments. The environment setup for this game is a grid of size 5*5. Each cell in the grid is a state, hence the total number of possible state is 25. Tom (agent) can make 4 actions in any state, which are namely– moving up, moving down, moving right, moving left. Tom(agent) cannot cross the borders of the grid that is go beyond 5*5 grid.

<u>IMPORTANT ASPECTS OF THE ENVIROMENT CLASS -</u>

1. The environment is class named Environment whose object is created by passing the grid size as the parameter.
2. _update_state function – The environment gives a new state after every action of agent. This function is used for that purpose of updating state. State is a tuple of 4 elements – the x and y coordinates of jerry and the x and y coordinates of Tom. Jerry 's(goal) coordinates always remain the same. Tom's coordinates gets updated based on the action it takes in the grid. If it takes an action that will make it go out of the grid, its state does not change at all hence such actions are equivalent to it not moving. Other than the coordinates of time, two other variables namely 'd' and 'time' are also maintained. 'd' represent the difference between the distance between agent and goal before and after the state of agent changes, whereas 'time' represents the maximum number of steps that can be taken by agent.
3. _get_reward function – specifies the immediate reward for an action based on variable 'd' which can take only three values 1, 0 and -1 which returns a reward of 1, - and -1 respectively.
4. _is_over function – specifies when the episode is over, either when goal is reached or the maximum number of steps (variable 'time') that can be taken is over.
5. render function – used to render the images of tom, jerry and the grid in a presentable manner.
6. reset function – After each and every episode, the environment needs to reach its starting form, that is all the variables of the environment needs to be re-initialized to their starting values, state of tom and jerry to the starting states etc so that each episode is the same game for the agent.

B. BRAIN

The Brain class is where the Deep Q-Network algorithm is implemented. An object of the Brain class is created by passing the state and action dimensions as parameters. The state dimension is 4 here as the state is a tuple of 4 elements as explained above. The action dimensions is also 4 here as 4 actions can be taken by an agent in a state as explained above.

The brain class has three more functions namely

1. train function– The train function takes training data (states and corresponding q values) to train the DQN model by implementing the fit function of Sequential Keras model so that the DQN model can learn the pattern and predict Q-values better.
2. predict function – The predict function implements the predict function of the Sequential Keras model that gives a predicted output of Q-values based on the input state.
3. predictOne function – This function implements the above predict function but reshapes the input parameter into a tuple of size equal to the state dimension (4).

SELF IMPLEMENTED – BUILDING A 3-LAYER NEURAL NETWORK, USING KERAS LIBRARY

4. _create_model function – As mentioned above, the Deep Q-network algorithm is implemented here by building a 3 layer neural network using Keras library. *It takes as input the state (which has a dimension of 4 as explained above) and returns a tuple of Q-values for each action taken by the agent from that inputted state* . This tuple is also of dimension 4. The DQN has 2 hidden layers. Both the hidden layers have 128 units and activation function for both layers is ‘relu’. The activation function for the final layer is “linear” and returns real values.

A Sequential Keras model is created. Three (3) “Dense” layers are “added” to the model using Dense and add functions of Keras library as follows -

1. The first one has 4 (state dimension) units, activation function is “relu” (activation function for first hidden layer) and the input shape is again a tuple of size 4.
2. The second one is the second hidden layer which has 128 units and an activation function “relu”.
3. The third one is the final output layer which has 4 units (dimension of actions) and an activation function “linear” and returns real values (which are Q-values).

ROLE IN TRAINING AGENT

The predictions given by the above DQN values are used to estimate Q-values for the actions taken by the agent in a given state. The Q-values in turn allow the agent to decide which action to take. Based on the predicted Q-values for each state the agent enters a tuple <current state, action, rewards, next state> is generated for each state. The tuple is stored in the memory of the agent. Some of these tuples are randomly chosen from the memory . The states and the updated q-values for those states (updating q-values is explained in Section D) are passed to this DQN above for training it (using the Brain class’s train function [explained above]) so that it generates better Q-values (which are as near as possible to correct Q-values so that the agent takes decisions corresponding to the optimal path).
Bottom-line: Its role in training the agent is to predict Q-values as correctly as possible, which in turn

helps the agent to take correct decisions as agent takes decision of which action to take based on Q-value. (detailed explanation in Section D).

C. MEMORY

The Memory class as the name suggests acts as the memory of the agent that is it has a list called "samples" where the experiences of the agent that is the tuple <current state, action, reward, next state> is stored. This tuple represents an experience of the agent because it tells the agent which action leads to which state from a state and that action gives what reward. A capacity variable is also defined which the maximum number of tuples/experiences it can store.

It has two functions namely –

1. add function – This is used to add experiences (tuples) to the memory ("samples" list).
2. sample function – This function returns some number (either the available number of tuples n the samples list or the whole samples list, whichever is minimum) of tuples in a randomized order which is further used to train the DQN so that it can learn from past experiences.

D. AGENT

The Agent class is Tom here. Here is where the objects of Brain and Memory class are created. It has three functions namely –

1. act function – this function takes as input the current state of agent and returns the action to be taken. It decides the action to be taken in either of the two ways – randomly (exploration) or choosing the one that has the maximum Q-value. The Q-values are predicted using the DQN above.
2. observe function – this function takes as input a sample i.e. an experience tuple and adds it to the memory of the agent. Apart from that it also increments the steps (steps taken by agent while playing the game) and also calculates the epsilon value.

SELF IMPLEMENTED – EXPONENTIAL - DECAY FORMULA FOR EPSILON

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

where $\epsilon_{\min}, \epsilon_{\max} \in [0,1]$
λ – hyper-parameter for epsilon
$|S|$ - total number of steps

Formula 1.1 – Exponential-Decay Formula for Epsilon

The code for implementing the exponential decay formula for epsilon was written based on Formula 1.1 shown above. The $\epsilon_{\min}$, $\epsilon_{\max}$ and λ are hyper-parameters which are defined in the Agent class. The ϵ is initialized to be $\epsilon_{\max}$. The python library "math" was used for using exponent.

ROLE IN TRAINING AGENT

The epsilon value acts as the control for implementing the trade-off between *"Exploration"* and *"Exploitation"*. A threshold for epsilon is defined which is also its maximum value. A number is chosen randomly, if the number is less than the current epsilon, the actions of the agent are chosen at random. Else, the Q-values (predicted using the DQN) are used to decide which action to be taken. The former method is called "*Exploration"* wherein the agent is simply exploring the environment and the latter is called "*Exploitation"* where the agent has already gained some knowledge about the environment and uses that knowledge to further predict actions that would lead to the goal in a path which is as optimal as possible. The DQN is trained on the basis of the past experience tuples to give better predictions on the Q-values. It is necessary to achieve a good trade-off between Exploration and Exploitation because if all actions are random there is high uncertainty on whether the model will ever converge and most cases it may never converge. If all actions are based on the Q-value prediction there is again the issue that the model never converges because the agent knows nothing about the environment to begin with and hence the DQN might keep predicting the same actions that give instant gratification that is positive instant reward and the agent never reaches the actual goal.
Bottom-line: Epsilon value is used to make the agent Explore as well as Exploit and to achieve a proper trade-off between these two.

3. replay function – this function implements the "Experience Replay" so that the DQN doesn't forget the past experiences. Here is also where the Q-value is being estimated based on Formula 1.2 below, for each observation in the batch. The batch has a certain number of randomly selected experiences from the memory of agent.

SELF IMPLEMENTED – Q-FUNCTION

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

Formula 1.2 – Q value calculation

The Formula 1.2 shown above was implemented using a simple if-and-else condition. If the episode terminates that is the next state value extracted from the experiences sample of the agent is "None" then the Q value for that (state, action) pair is the instant reward the agent receives else the Q-value is the sum of the instant reward and the product of discount factor (γ) and maximum value of the Q-values predicted for the next state and all actions taken from that state.

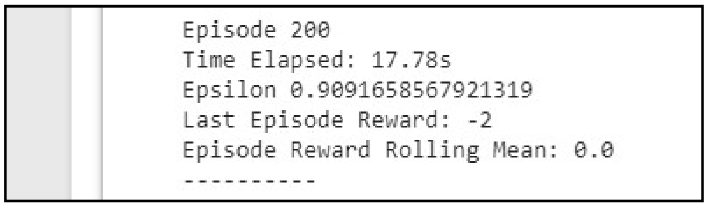
ROLE IN TRAINING AGENT

Q-value is the expected future reward. The$Q(s_{t+1}, a; \Theta)$, basically produces a tuple of 4 values, one for each action taken from the next state that will be reached from the current state of agent on taking a certain action. The maximum Q-value is taken from this tuple as that specifies the maximum expected future reward. The action that corresponds to this maximum value is the action that the agent decides to take next as it will give the maximum expected future reward as per DQN prediction. As this is the expected future reward, it is discounted by a factor of (γ). When the agent reaches the goal, it need not take any action further hence there is no need to find a expected future reward as there Is not future state hence the Q-value when the episode ends that is either the agent reaches the goal or the agent has taken maximum number of steps possible is the instant reward.

E. MAIN PROGRAM

The Environment and Agent class is objects are made. It defines the number of episodes that is the number of times the agent should play the game and keep track of it. It runs a loop until episodes end where in makes the agent decide an action, passes this action to the environment to receive the reward and the next state. If the next state is when the episode ends it makes the next state as ”None”. It adds the experiences to the memory of the agent and uses them to implement “Experience Replay”. It keeps a record of the reward for each episode, the number of episodes and epsilon values. All the functions in classes defined earlier come into play here to make the agent learn to play this game.

F. OUTPUT



```
Episode 200
Time Elapsed: 17.78s
Epsilon 0.9091658567921319
Last Episode Reward: -2
Episode Reward Rolling Mean: 0.0
----------
```

Fig 1.1

The main output of the program is formatted as shown in the Fig 1.1 above. Every 100 episodes this output is generated. It has the following 5 components –

1. Episode number – This specifies the number of episodes completed till now.
2. Time Elapsed – It shows how much time has elapsed since the training of the agent ahs begun.
3. Epsilon – The current epsilon value which is updated after every step.
4. Last episode Reward – The reward that the agent received at the end of the last episode that was run till now.
5. Episode Reward Rolling Mean – This is the mean of the rewards received by the agent in the last 100 episodes.

The above five values give us an idea as to how the program is performing as the number of episodes increase and whether the agent is receiving higher rewards over the time or not. If in any episode agent received the maximum reward possible it is known that the agent reached its goal in the optimal path in that episode which can be told by the Last episode reward value. The time elapsed will give us idea about how much time the model takes to converge.

Two graphs are generated which are –

1. Fig 1.2 shows the graph between the Episode number and Epsilon value.
2. Fig 1.3 shows the graph between the Episode number and Episode reward.

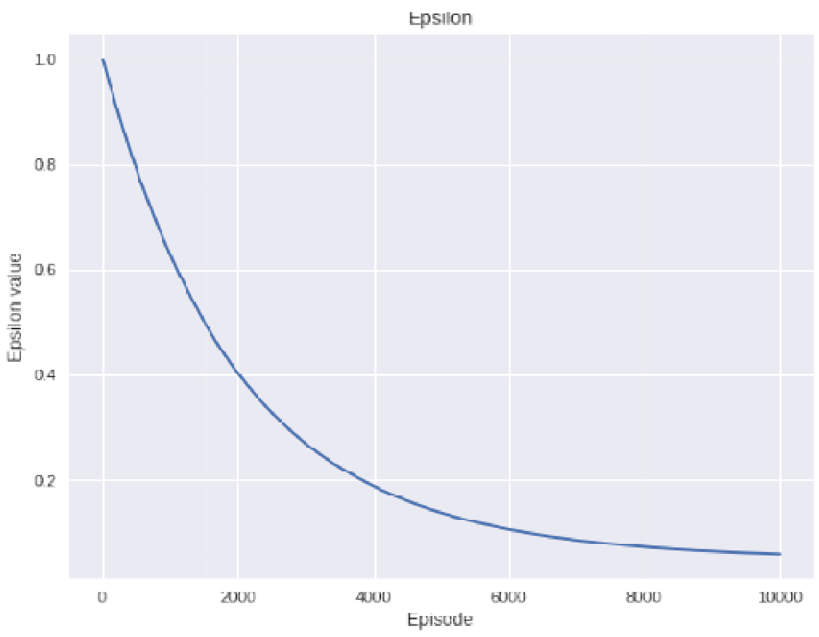


Fig 1.2 – Graph Episode number Vs Epsilon value

The graph in Fig 1.2 clearly shows that the epsilon value exponentially decreases (as per the Formula 1.1) and then becomes constant at its pre-defined minimum value (MIN_EPSILON, for this graph its value is 0.05).

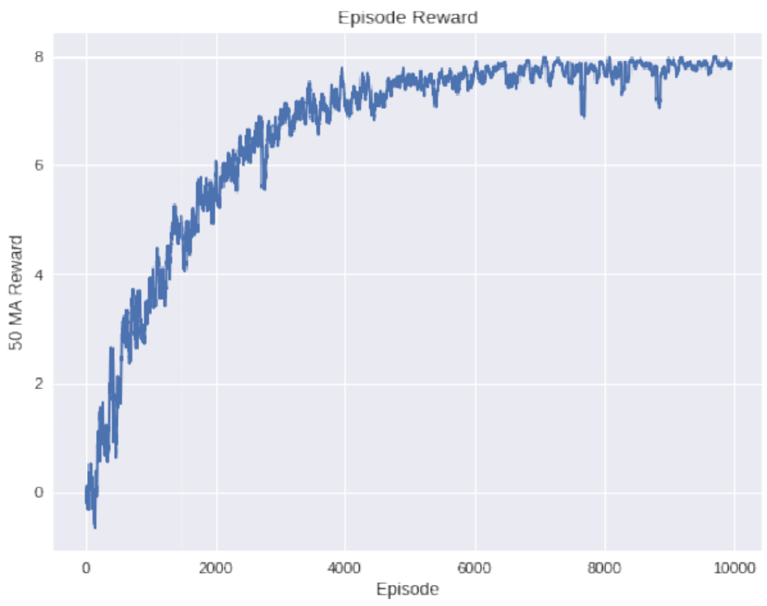


Fig 1.3 – Graph Episode number Vs Episode Reward

The graph in Fig 1.3 shows that the reward the agent receives gradually increases as the number of episodes it has played increases. It can also be seen that the reward starts to become constant at 8 after approximately 6000 episodes . It is important to notice here that 8 is the maximum reward possible for this game.

III. HOW TO IMPROVE THE SELF IMPLEMENTED COMPONENTS OF CODE AND HOW IT INFLUENCES TRAINING THE AGENT ?

1) Improving the Epsilon Decay Formula

Varying the epsilon decay function. All hyperparameters were kept same as that in the source code. These are the functions and their corresponding implementations :

1. Exponential Decay function – refer Formula 1.1 and Fig 1.2

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

2. Linear Decay Function

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

where $\epsilon_{\min}, \epsilon_{\max} \in [0,1]$
λ – hyper-parameter for epsilon
$|S|$ - total number of steps

Formula 1.2: Linear Decay of Epsilon Value

self.epsilon = self.min_epsilon + (self.max_epsilon - self.min_epsilon) * -self.lamb * self.steps

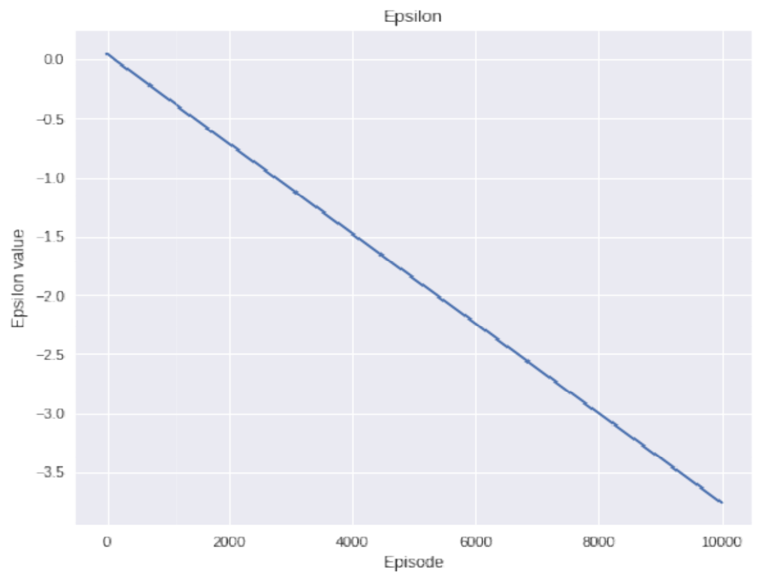


Fig 1.4: Linear decay of Epsilon Value

3. Product of Static Epsilon Value and decay$^{|S|}$

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

where $\epsilon_{\min}, \epsilon_{\max} \in [0,1]$
λ – decay $\in [0,1]$
$|S|$ - total number of steps

Formula 1.3: Product of Static Epsilon Value and decay$^{|S|}$

self.epsilon = self.min_epsilon + (self.max_epsilon) * math.pow(self.lamb, self.steps)

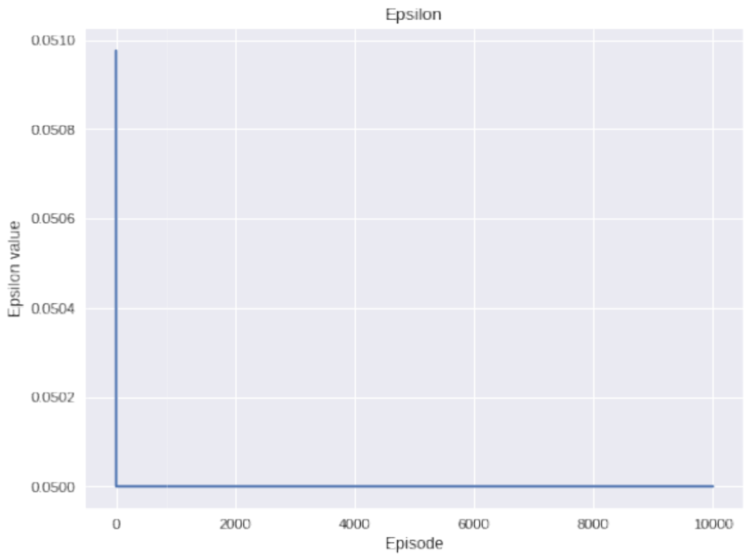


Fig 1.5: Product of Static Epsilon Value and decay$^{|S|}$

Here decay value was, Decay = 0.5. Very instantly it reaches the minimum value of epsilon and remains at it throughout all episodes.

Type of decay function	Mean Reward	Time taken (sec)
exponential decay	6.532802775	391.53
linear decay	7.974594429	721.36
Product of static epsilon value and decay^S	7.764717886	364.74

Table 1: Varying exponential decay functions , the mean reward and the total time taken to train the agent in seconds

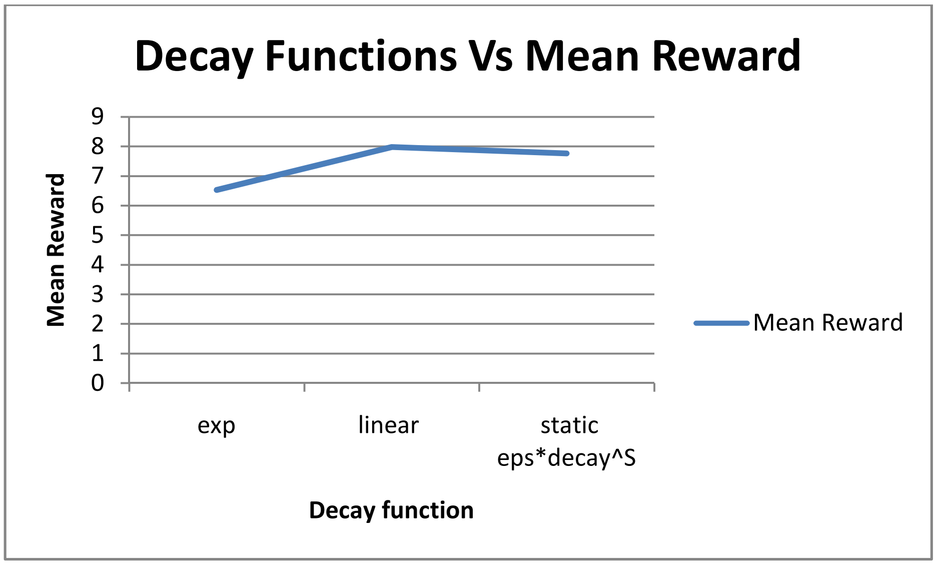


Fig 1.6: Mean Reward obtained on changing decay function

It can be seen from the Table 1 and Fig 1.6, that the linear decay function gives a near 8 mean reward (maximum possible reward). Even the product of static epsilon value with decay$^{|S|}$ gives pretty good mean reward. The least mean reward was obtained using the exponential decay function.

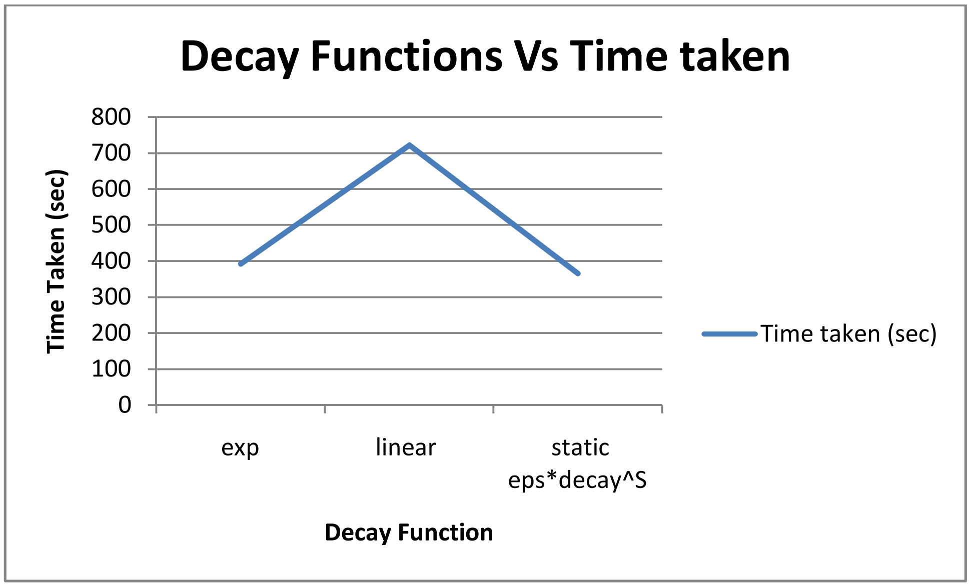


Fig 1.7: Time taken to train agent on changing decay function

It can be observed from the Fig 1.7 that the linear decay function takes the highest time and the static product of static epsilon value with decay$^{|S|}$ take the least amount of time.
On observing both figures Fig 1.6 and Fig 1.7 it can said that the implemented code can be improved by using a product of static epsilon value with decay$^{|S|}$ function for best results as it takes the least time and although doesn't give the highest mean reward but the mean reward obtained using it is pretty near to the highest.

2) Improving the Neural Network Implementation

List of hyper-parameters for Neural Networks :

1. Number of Epochs – While training the training data using the fit function we can specify this parameter. This is the number of times/iterations the entire data set needs to be re-run by the model. The parameter is called epochs.
2. Batch Size – While training the training data using the fit function we can specify this parameter . The batch size specifies the amount of data that needs to be passed to the model at a time while training.
3. Optimizer – The optimizer to be used for optimizing the model to minimize the loss. There are many optimizers which can be specified – 'sgd' which is stochastic gradient descent, 'RMSprop', 'Adagrad', 'Adadelta'.
4. Number of units in hidden layer – This is specified as a parameter to the Dense function. As the name suggests it is the number of neurons/units in the layer.
5. Number of Layers – in the neural network.

GRAPHS AND ANALYSIS

a. VARYING OPTIMIZER

Constant hyper-parameters and their values:

1. Batch Size : 64
2. Number of Units in hidden layer: 128
3. Number of Epochs : 1
4. Number of layers: 3

Optimizer	Mean Reward	Time Taken (sec)
sgd	6.593408836	446.44
RMSprop	6.547597184	460.32
Adagrad	6.553310887	451.22

Table 2 - Shows the mean reward and total time for agent training when the optimizer is varied

The Table 2 shows the mean reward and total time for training the agent when the optimizer is varied while keeping other hyper-parameters constant. The Fig 1.8 below graphically depicts the variation in mean reward when optimizer is varied. Fig 1.9 below illustrates the relationship between optimizer and the total time taken to train agent.

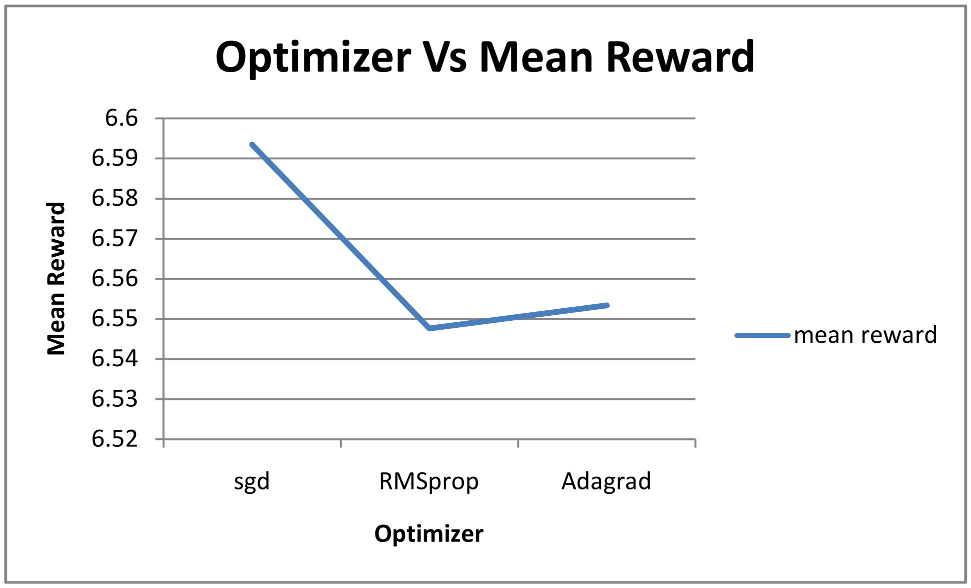


Fig 1.8 – Shows the change in mean reward when the optimizer is varied

It can be seen from Fig 1.8, that sgd optimizer gives the best mean reward and RMSprop gives the lease mean reward followed by Adagrad. But it can be seen from the Table 2, that the mean reward values don't vary significantly.

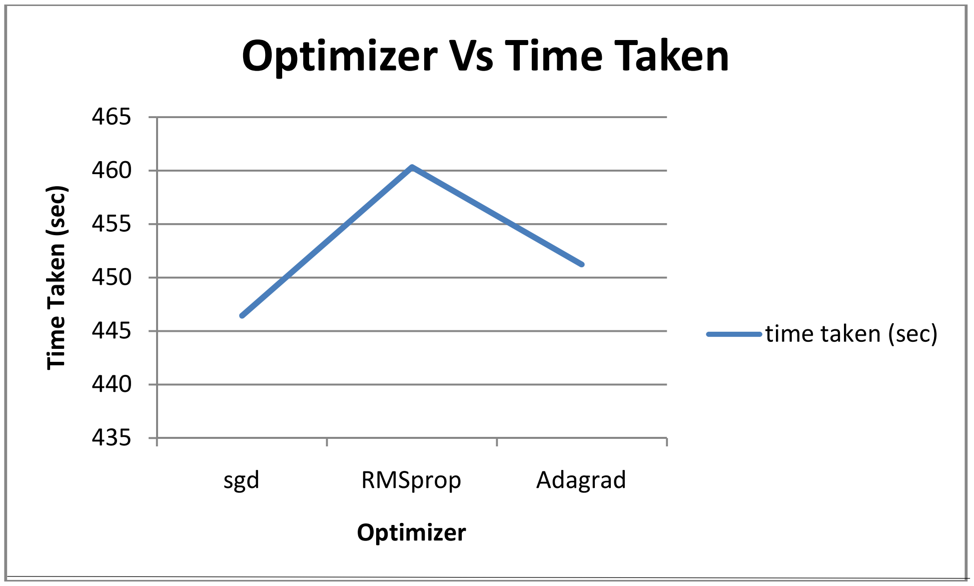


Fig 1.9 – Shows the variation in time taken to train agent when the optimizer is varied

It can be seen from Fig 1.9, that the RMSprop optimizer takes the highest time and the sgd optimizer takes the least time followed by Adagrad.

On observing Fig 1.8 and Fig 1.9, it can be said that the implemented code can be improved by using sgd optimizer as it takes the least time to train the agent and gives highest mean reward.

b. VARYING NUMBER OF EPOCHS

Constant hyper-parameters and their values:

1. Batch Size : 64
2. Number of Units in hidden layer: 128
3. Optimizer : RMSprop
4. Number of layers: 3

Number of Epochs	Mean reward	Time Taken (sec)
1	6.532802775	391.53
10	6.448831752	1276.12
15	6.334587346	5124.8

Table 3 – Shows the mean reward and total time for agent training when the number of epochs are varied

The Table 3 shows the mean reward and total time for training the agent when the number of epochs are varied while keeping other hyper-parameters constant. The Fig 2.0 below graphically depicts the variation in mean reward when number of epochs are varied. Fig 2.1 below illustrates the relationship between number of epochs and the total time taken to train agent.

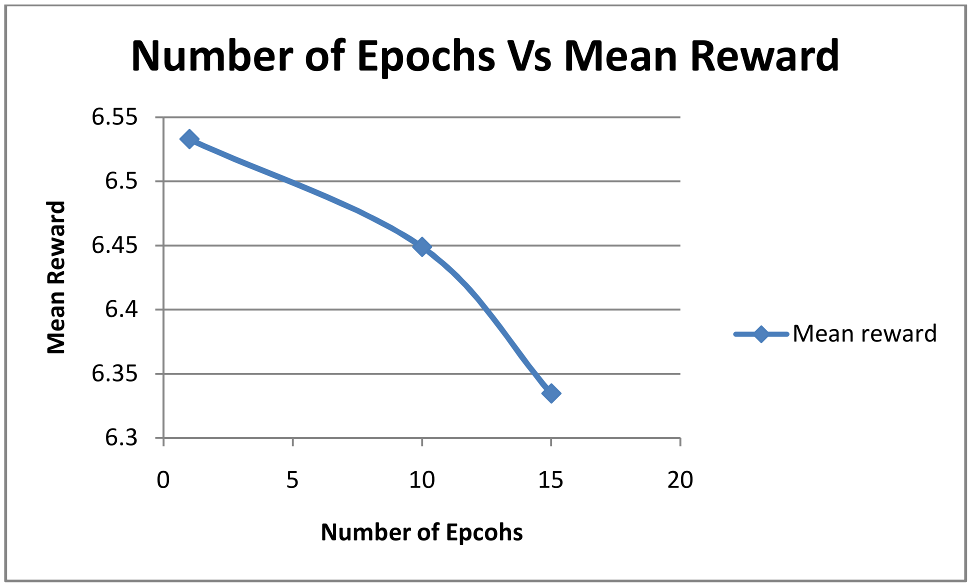


Fig 2.0 Shows the change in mean reward when the number of epochs are varied

It can be seen from Fig 2.0, the mean reward decreases as the number of epochs are increased. The best reward is obtained when number of epochs is 1 and worst when it is 15. As the neural network is predicting the expected future rewards for each action for that state, if the same states are passed again and again, the neural network might start thinking, the same state is being reached again and again and change its understanding of the environment hence decreasing the mean reward.

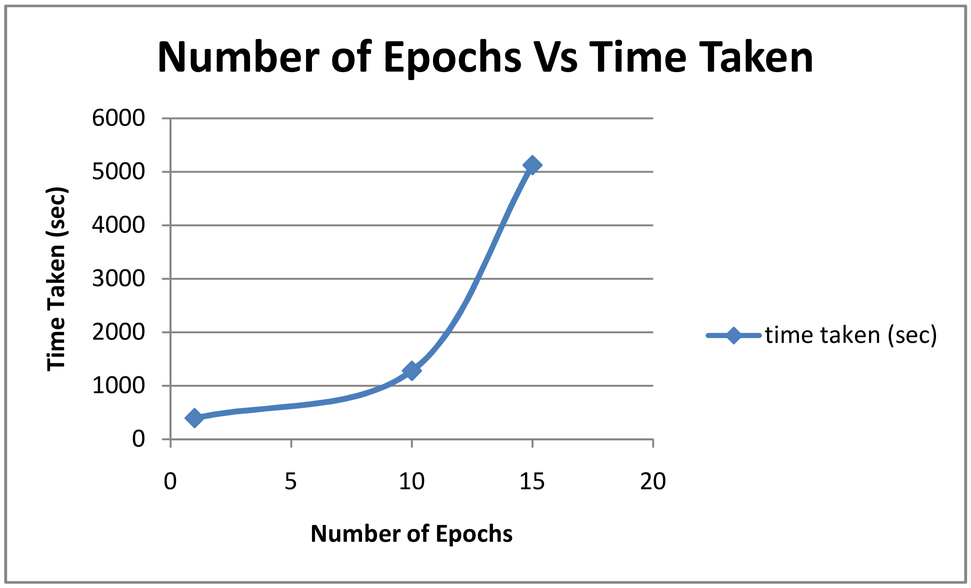


Fig 2.1 – Shows the variation in time taken to train agent when the number of Epochs are varied

It can be seen from Fig 2.1, the time taken to train the agent gradually decreases with increase in the number of epochs. This makes sense, the more the times the data will be passed ans run, the time taken for the agent to train will increase.

On observing Fig 2.0 and Fig 2.1, it can be said that the implemented code uses the best number of epochs which is 1 and there is no need to change anything for improvement.

c. VARYING BATCH SIZE

Constant hyper-parameters and their values:

1. Number of Epochs : 1
2. Number of Units in hidden layer: 128
3. Optimizer : RMSprop
4. Number of layers: 3

Batch Size	Mean Reward	Time Taken (sec)
32	6.293235384	521.87
64	6.532802775	391.53
100	6.498316498	441.98

Table 4 – Shows the mean reward and total time for agent training when the batch size is varied

The Table 4 shows the mean reward and total time for training the agent when the batch size is varied while keeping other hyper-parameters constant. The Fig 2.2 below graphically depicts the variation in mean reward when batch size is varied. Fig 2.3 below illustrates the relationship between batch size and the total time taken to train agent.

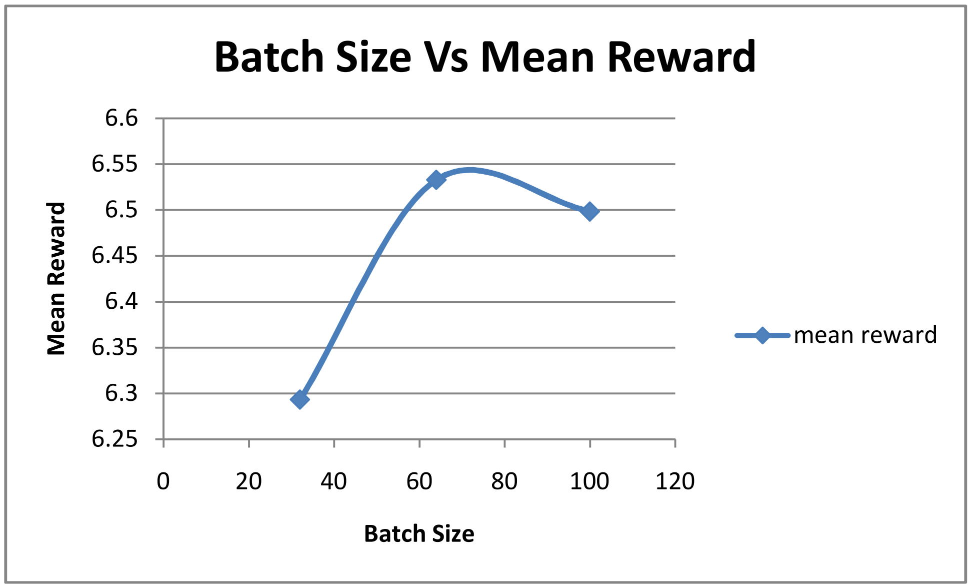


Fig 2.2 – Shows the change in mean reward when the batch size is varied

It can be seen from Fig 2.2, the mean reward increases from batch size 32 and reaches a peak at batch size 64 and then again decreases when batch size is further increased to 100. This means that for the neural network to know about the environment, if too less data is given at once it cannot gauge the situation well hence such low mean reward. When the neural network is given higher amount of data at one go, it does help it to understand the environment, but some unnecessary details that might not be required at that time, might confuse the neural network making it give less mean reward.

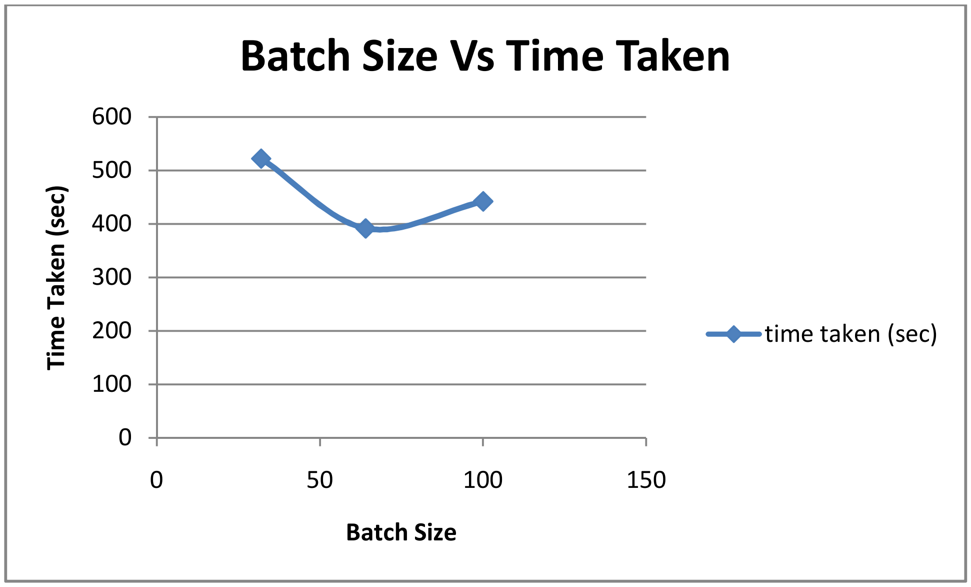


Fig 2.3 – Shows the variation in time taken to train agent when the batch size is varied

It can be seen from Fig 2.3, the time taken to train the agent doesn't vary that much with the batch size but gives the least time when batch size is 64.

On observing Fig 2.2 and Fig 2.3, it can be said that the implemented code uses the best batch size which is 64 and there is no need to change anything for improvement.

d. VARYING NUMBER OF UNITS IN HIDDEN LAYER

Constant hyper-parameters and their values:

1. Number of Epochs : 1
2. Batch Size: 64
3. Optimizer : RMSprop
4. Number of layers: 3

No of Units in Hidden Layer	Mean Reward	Time Taken (sec)
32	5.850321396	406.34
64	6.456484032	416.26
128	6.532802775	391.53
256	6.298438935	460.92

Table 5 - Shows the mean reward and total time for agent training when the no. of units in hidden layer are varied

The Table 5 shows the mean reward and total time for training the agent when the number of units in hidden layer are varied while keeping other hyper-parameters constant. The Fig 2.4 below graphically depicts the variation in mean reward when number of units in hidden layer are varied. Fig 2.5 below illustrates the relationship between number of units in hidden layer and the total time taken to train agent.

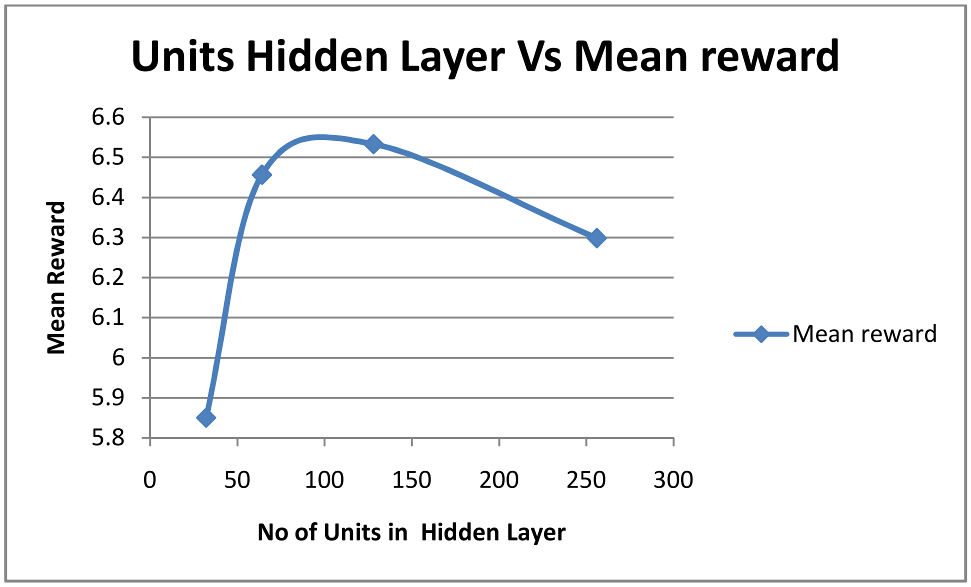


Fig 2.4 – Shows the change in mean reward when the no. of units in hidden layer are varied

It can be seen from Fig 2.4, the mean reward increases from number of units 32 and reaches a peak at number of units 128 and then again decreases when number of units is further increased to 256.This means 128 number of units in the hidden layer is optimal number that has been used in the source code.

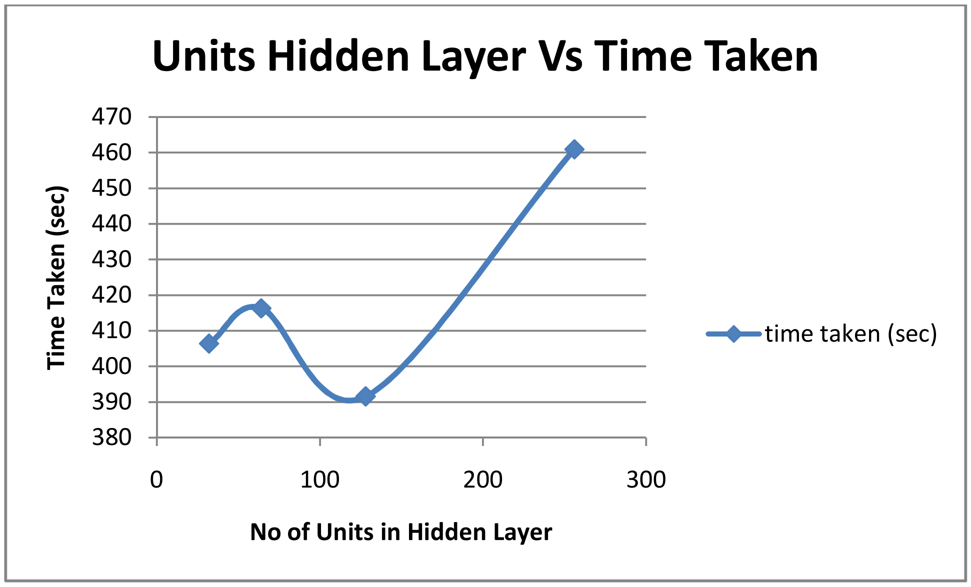


Fig 2.5 – Shows the variation in time taken to train agent when no. of units in hidden layer are varied

It can be seen from Fig 2.5, the time taken to train the agent varies irregularly when the number of units in hidden layer are less than 128 and surges when the number of units is increased to 256. It can be seen that 128 number of units take the least time.

On observing Fig 2.4 and Fig 2.5, it can be said that the implemented code uses the best number of units in hidden layer which is 128 and there is no need to change anything for improvement.

e. VARYING NUMBER OF LAYERS

Constant hyper-parameters and their values:

1. Number of Epochs : 1
2. Batch Size: 64
3. Optimizer : RMSprop
4. Number of units in hidden layer: 128

No of Layers	Mean Reward	Time Taken (sec)
3	6.532802775	391.53
4	6.290378533	459.09
5	6.015814713	524.76

Table 6 - Shows the mean reward and total time for agent training when the no. of layers are varied

The Table 6 shows the mean reward and total time for training the agent when the number of layers are varied while keeping other hyper-parameters constant. The Fig 2.6 below graphically depicts the variation in mean reward when number of layers are varied. Fig 2.7 below illustrates the relationship between number of layers and the total time taken to train agent.

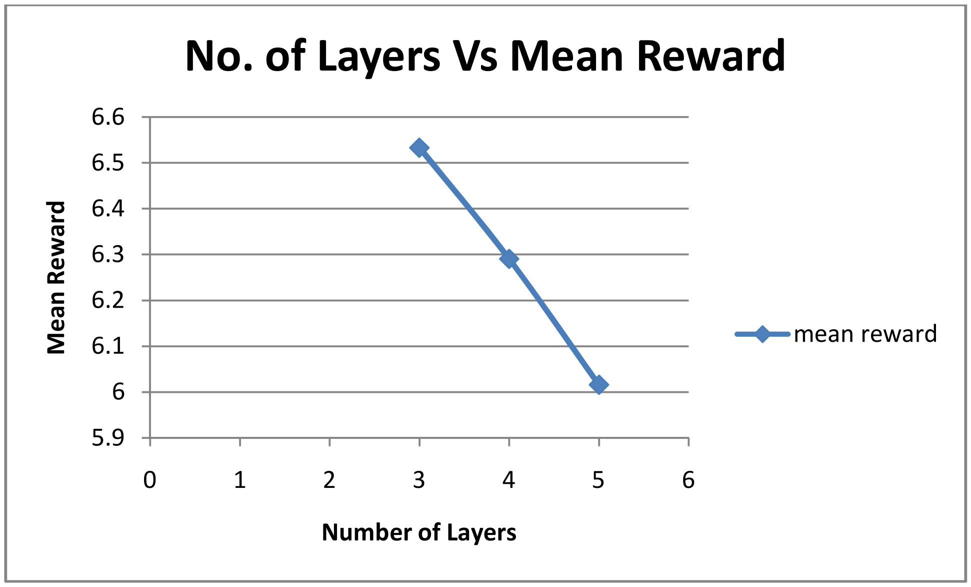


Fig 2.6 – Shows the change in mean reward when the no. of layers are varied

It can be seen from Fig 2.6, the mean reward decreases with the increase in the number of layers. The number of layers increase the complexity of the neural network and apply activation functions multiple times on the input values to produce the output. Here, the neural network is generating expected future reward values, which are based on the current state and only four possible actions. As the set up is so simple, using a complex neural network might be hindering from it working optimally hence producing low mean reward as the number of layers are increased.

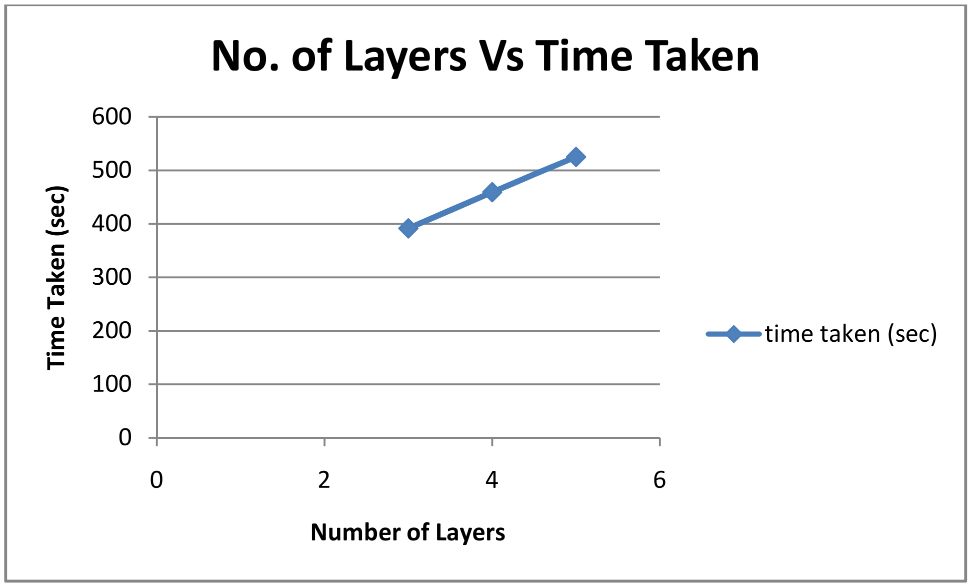


Fig 2.7 – Shows the variation in time taken to train agent when no. of layers are varied

It can be seen from Fig 2.7, the time taken to train the agent increases as the number of layers increases. This is logical since, more layers means more complexity, more computation hence more time taking.

On observing Fig 2.6 and Fig 2.7, it can be said that the implemented code uses the best number of layers which is 3 as it gives best reward and takes least time. There is no need to change anything for improvement.

IV. HOW QUICKLY YOUR AGENT WERE ABLE TO LEARN?

The code was run on Google Colab GPU. After implementation of the three parts, the code was run with the same hyperparameters given in the source code. The code was running in `798.77` seconds = 13.31 minutes.

Number of Episodes	Mean Reward (last 100 episodes)	Total Time (sec)	Time to reach goal (sec)
500	0.242524917	34.21	Never reaches
1000	1.494382022	74.64	Never reaches
6000	5.660058611	482.48	100.92
10000	6.532802775	798.77	111.02
15000	6.82920073	1157.56	91.12

Table 2: On Varying number of episodes, obtained mean reward, total time taken and the time taken by the agent to reach the goal with a reward of 8 for the first

As per the Table 2 and Graph 1.1 below, it can be seen that if the agent has to be run for 6000 or more episodes to learn how to navigate optimally. The agent gets better and better trained on increasing the episodes till 15000. After 15000, much change in the mean reward is not seen but it will take a very long time to run. So it can be said the agent has to be trained for any number of episodes between 6000 and

15000. The reason the agent has to be made to play these many games is that, it gives enough time for the agent to explore about the environment as well as the DQN to generate better and better predictions over time as it gets to see more information about the environment over the time. At 150000, episodes it has very well understood how to navigate optimally in the environment which can be seen by the increment in mean reward and more number of 8 rewards (maximum possible reward) in some episodes towards the end of the training. Another observation is that, when the agent is trained only for 500 or 1000 episodes, the agent never reaches the goal even once. Hence it should be run for more episodes. After 6000 episodes the agent reaches the goal on an average of 101.02 seconds = 1.68 minutes to first time reach the goal obtaining a reward of 8. This shows the agent takes 1.68 minutes to fully explore the environment to know the existence of the goal and an optimal path to it. Each episode takes an average of 0.08 seconds to run.

V. HYPERPARAMETER TUNING

The program was run in Google Colab GPU.

List of hyperparameters which can be tuned –

1. Number of episodes
2. Maximum value of Epsilon
3. Minimum value of Epsilon
4. Discount Factor (γ)
5. Lambda value (λ): hyperparameter for epsilon decay
6. Range of Epsilon: Maximum Epsilon - Minimum Epsilon

GRAPHS AND ANALYSIS

1) VARYING NUMBER OF EPISODES

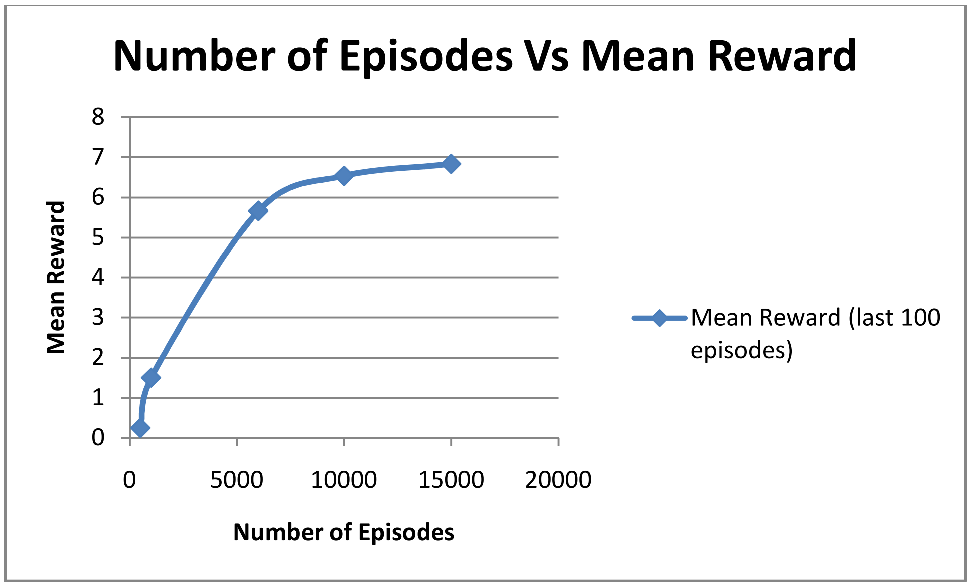
Constant hyperparameters and their values :

1. Maximum value of Epsilon : 1
2. Minimum value of Epsilon : 0.05
3. Discount Factor (γ) : 0.99
4. Lambda value (λ): 0.00005
5. Range of Epsilon: 0.95

Number of Episodes	Mean Reward (last 100 episodes)	Total Time (sec)
500	0.242524917	34.21
1000	1.494382022	74.64
6000	5.660058611	482.48
10000	6.532802775	798.77
15000	6.82920073	1157.56

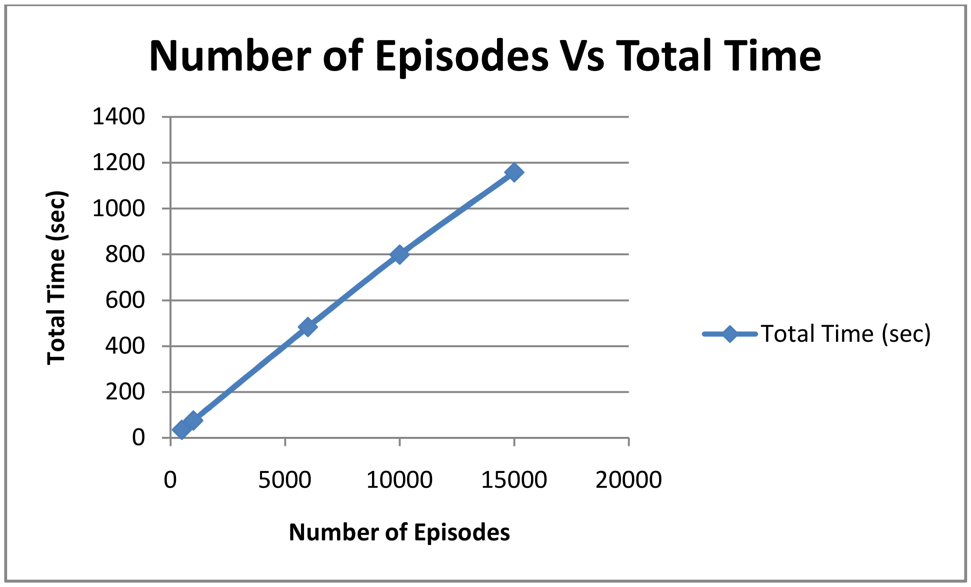
Table 1.1 – Shows the mean reward for the last 100 episodes and the total time for training when the number of episodes are varied

The Table 1.1 above shows the values obtained for the mean reward taken for last 100 episodes and the total time taken to train when the number of episodes were varied keeping all the other hyperparameters constant. The Graph 1.1 below graphically depicts the variation in mean reward when number of episodes is varied. Graph 1.2 below illustrates the relationship between number of episodes and the total time taken to train agent.



Graph 1.1 – Number of Episodes versus Mean Reward taken for last 100 episodes

It can be observed from the Graph 1.1, that the mean reward gradually increases with the number of episodes increase. It can also be seen that the increase in mean reward is less and less as the number of episodes increase. The change in mean reward is significantly less when the number of episodes go beyond 10000. It seems the mean reward starts to become constant. The more the number of games played the more rewarding it is. That said, after a certain number no matter how many games the agent plays the mean reward doesn't increase much.



Graph 1.2 – Number of Episodes versus Total Time taken for training

It can be observed clearly from Graph 1.2 , that the total time taken to train is linearly dependent on the number of episodes. Not only does the total time taken increase as the number of episodes increase, the visibly straight lined graph shows that there is proportional change in total time when number of episodes are changed.

2) *VARYING MAXIMUM VALUE OF EPSILON*

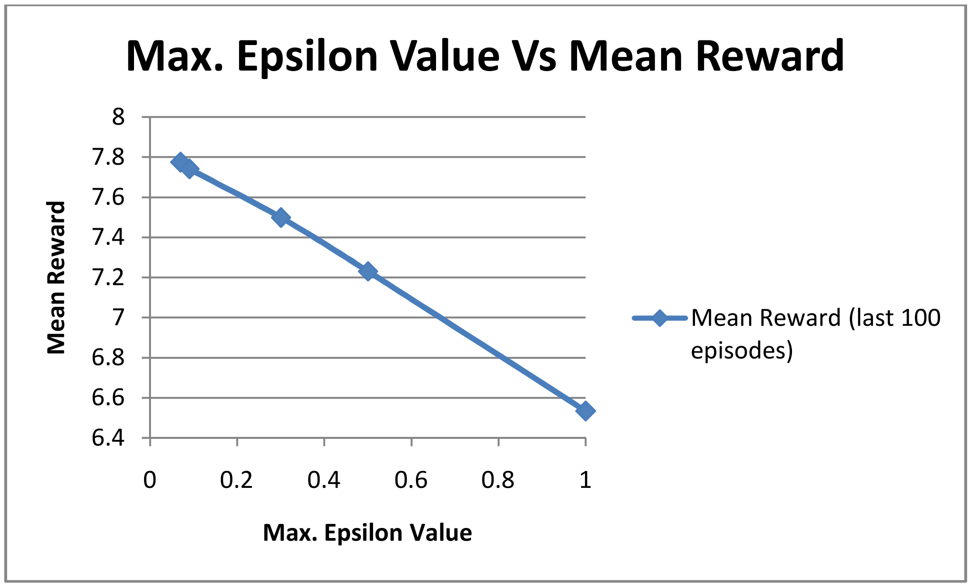
Constant hyperparameters and their values :

1. Number of Episodes : 10000
2. Minimum value of Epsilon : 0.05
3. Discount Factor (γ) : 0.99
4. Lambda value (λ): 0.00005
5. Range of Epsilon: 0.95

Max. Epsilon Value	Mean Reward (last 100 episodes)	Total Time (sec)
0.07	7.773900622	775.21
0.09	7.740332619	769.52
0.3	7.497398225	799.63
0.5	7.229874503	815.02
1	6.532802775	798.77

Table 2.1 – Shows the mean reward for the last 100 episodes and the total time for training when the maximum value of epsilon is varied

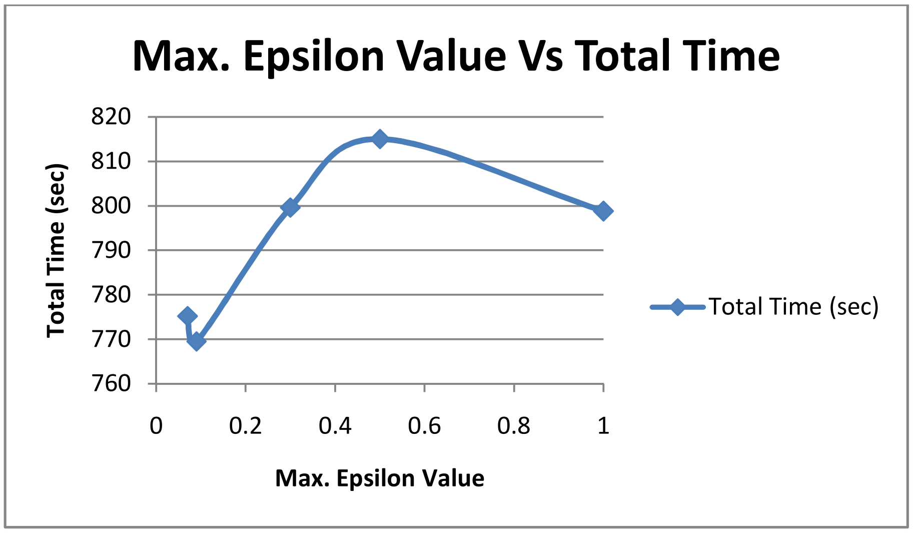
The Table 2.1 above shows the values obtained for the mean reward taken for last 100 episodes and the total time taken to train when the maximum value of epsilon is varied keeping all the other hyperparameters constant. The Graph 2.1 below graphically depicts the variation in mean reward when maximum value of epsilon is varied. Graph 2.2 below illustrates the relationship between the maximum value of epsilon and the total time taken to train agent.



Graph 2.1 – Maximum Value of Epsilon versus Mean Reward taken for last 100 episodes

The epsilon value governs the amount of exploration and exploitation the agent does. The lower the maximum value of epsilon, the faster the agent moves from more amount of exploration to more amount of exploitation. As lower max. value of epsilon is giving more mean reward, we can say that more amount of exploitation is being more rewarding. That means, actions taken based on Q values predicted by DQN are better than taking actions randomly.

In Graph 2.1, an approximately linear inverse relationship between mean reward and the maximum epsilon value. As the maximum value of epsilon is increased, the mean reward obtained decreases. The decrease in mean reward is nearly proportional to the increase in maximum value of epsilon. This shows that a lower maximum value of epsilon is more rewarding.



Graph 2.2 – Maximum Value of Epsilon versus Total Time taken for training

It can be observed clearly from Graph 2.2 , that there is no proper relationship between maximum value of epsilon and the total time taken to train the agent. Though, it can be seen that the time taken increases when maximum value of epsilon is increasing between 0.09 to 0.5. There is a slight decrease in the total time though not that significant when the maximum value of epsilon increases from 0.07 to 0.09. There is a significant decrease in total time when the maximum value of epsilon increases from 0.5 to 1.

3) VARYING MINIMUM VALUE OF EPSILON

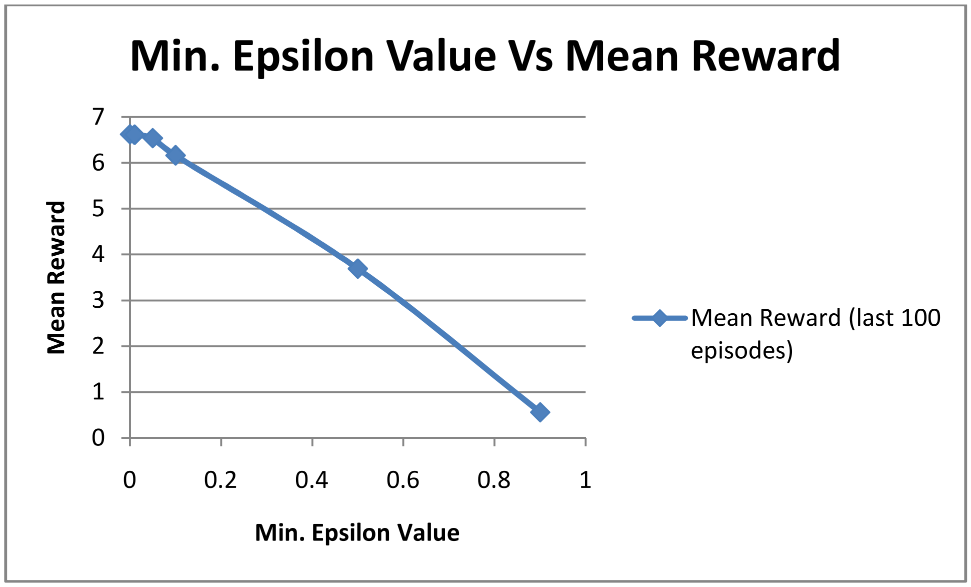
Constant hyperparameters and their values :

1. Number of Episodes : 10000
2. Maximum value of Epsilon : 1
3. Discount Factor (γ) : 0.99
4. Lambda value (λ): 0.00005
5. Range of Epsilon: 0.95

Min. Epsilon Value	Mean Reward (last 100 episodes)	Total Time (sec)
0	6.616467707	818.68
0.01	6.607489032	824.24
0.05	6.532802775	798.77
0.1	6.159167432	838.95
0.5	3.687888991	876.09
0.9	0.557290072	861.11

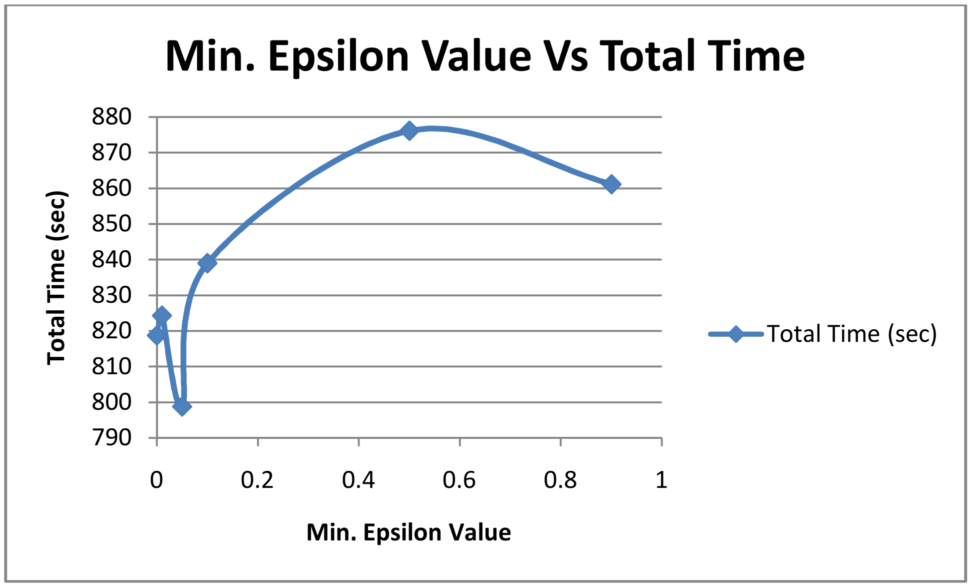
Table 3.1 – Shows the mean reward for the last 100 episodes and the total time for training when the minimum value of epsilon is varied

The Table3.1 above shows the values obtained for the mean reward taken for last 100 episodes and the total time taken to train when the minimum value of epsilon is varied keeping all the other hyperparameters constant. The Graph 3.1 below graphically depicts the variation in mean reward when minimum value of epsilon is varied. Graph 3.2 below illustrates the relationship between the minimum value of epsilon and the total time taken to train agent.



Graph 3.1 – Minimum Value of Epsilon versus Mean Reward taken for last 100 episodes

In Graph 3.1, an approximately linear inverse relationship between mean reward and the minimum epsilon value. As the minimum value of epsilon is increased, the mean reward obtained decreases. The decrease in mean reward is nearly proportional to the increase in minimum value of epsilon. This shows that a lower minimum value of epsilon is more rewarding. A lower minimum value of epsilon means higher chance of exploration for the agent and it is being more rewarding a higher chance of making decisions to take random actions.



Graph 3.2 – Minimum Value of Epsilon versus Total Time taken for training

It can be observed clearly from Graph 3.2 , that there is no proper relationship between minimum value of epsilon and the total time taken to train the agent. Though, it can be seen that the time taken increases when minimum value of epsilon is increasing between 0.05 to 0.5. There is a slight decrease in the total time though not that significant when the maximum value of epsilon increases from 0.01 to 0.05. There is a significant decrease in total time when the maximum value of epsilon increases from 0.5 to 0.9.

4) VARYING DISCOUNT FACTOR

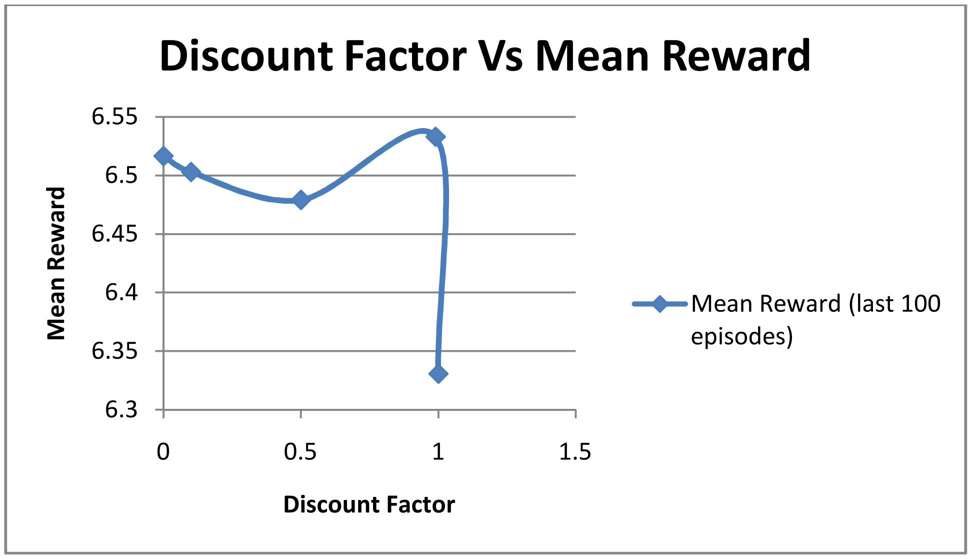
Constant hyperparameters and their values :

1. Number of Episodes : 10000
2. Maximum value of Epsilon : 1
3. Minimum value of Epsilon : 0.05
4. Lambda value (λ): 0.00005
5. Range of Epsilon: 0.95

Discount Factor	Mean Reward	Total Time (sec)
0	6.51637588	854.47
0.1	6.502805836	843.15
0.5	6.478930721	858.71
0.99	6.532802775	798.77
1	6.330578512	870.72

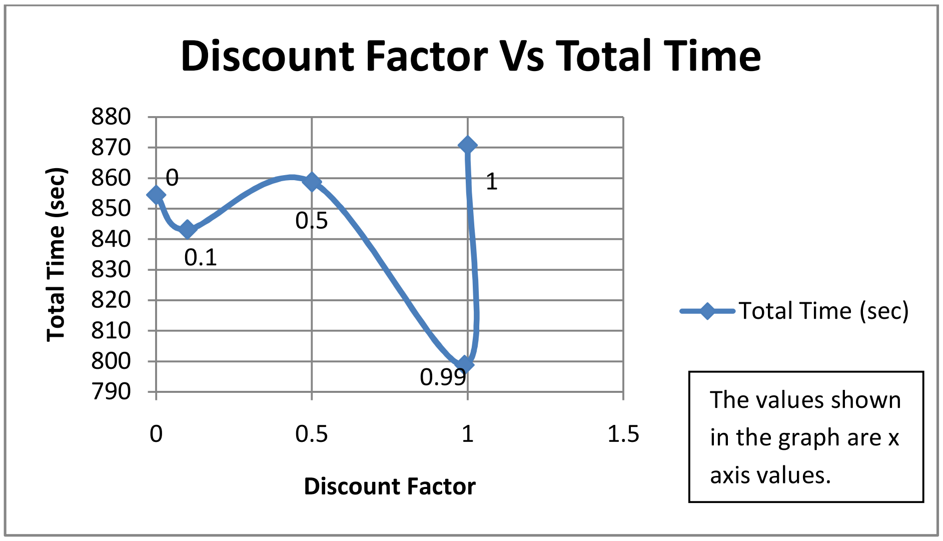
Table 4.1 – Shows the mean reward for the last 100 episodes and the total time for training when the discount factor is varied

The Table 4.1 above shows the values obtained for the mean reward taken for last 100 episodes and the total time taken to train when the discount factor is varied keeping all the other hyperparameters constant. The Graph 4.1 below graphically depicts the variation in mean reward when the discount factor is varied. Graph 4.2 below illustrates the relationship between the discount factor and the total time taken to train agent.



Graph 4.1 – Discount factor versus Mean Reward taken for last 100 episodes

In Graph 4.1, we can observe that when discount factor increases from 0 to 0.5 the mean reward decreases slightly. From $\gamma = 0.5$ to $\gamma = 0.99$, the mean reward increases and reaches a peak value it falls down significantly when the discount factor is made 1. The falling down of mean reward when discount factor is made 1, iterates the importance of discount factor which is discounting the future rewards as they are uncertain. If the present rewards and future rewards are given equal importance, then mea reward is less. The graph also shows that if we discount them heavily ($\gamma = 0.5$) even then high mean reward cannot be reached.



Graph 4.2 – Discount factor versus Total Time taken for training

It can be observed clearly from Graph 4.2 , that there is no proper relationship between discount factor and the total time taken to train the agent. Though, one thing is starkly visible and that is the total time taken for $\gamma = 0.99$, is significantly less as compared to any other discount factor value.
By the Graph 4.1 we can see that we get the best mean reward for $\gamma = 0.99$. From the above statement we can say that $\gamma = 0.99$, is the best discount factor that can be taken for this project.

5) *VARYING LAMBDA*

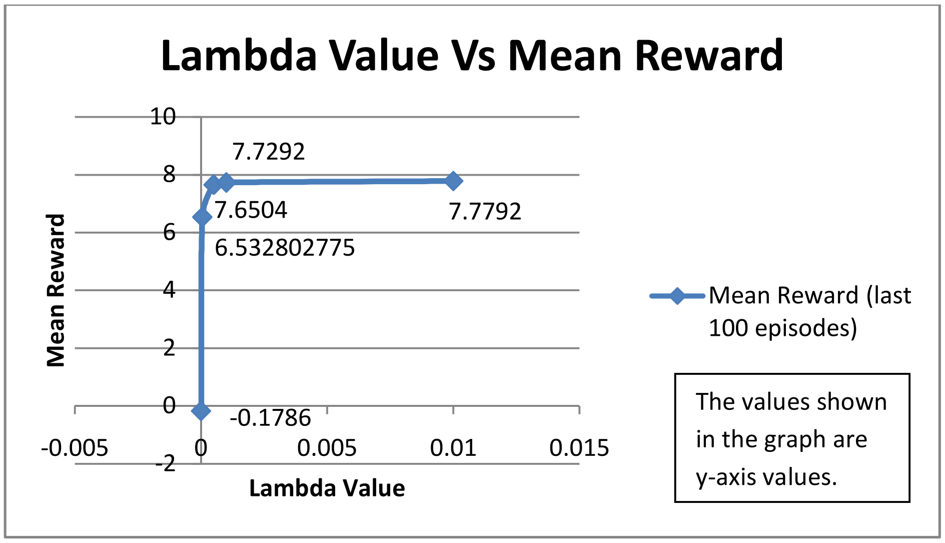
Constant hyperparameters and their values :

1. Number of Episodes : 10000
2. Maximum value of Epsilon : 1
3. Minimum value of Epsilon : 0.05
4. Discount Factor (γ) : 0.99
5. Range of Epsilon: 0.95

Lambda	Mean Reward	Total Time (sec)
0	-0.178655239	885.24
0.00005	6.532802775	798.77
0.0005	7.650443832	835.73
0.001	7.729211305	1025.11
0.01	7.779206203	1033.46

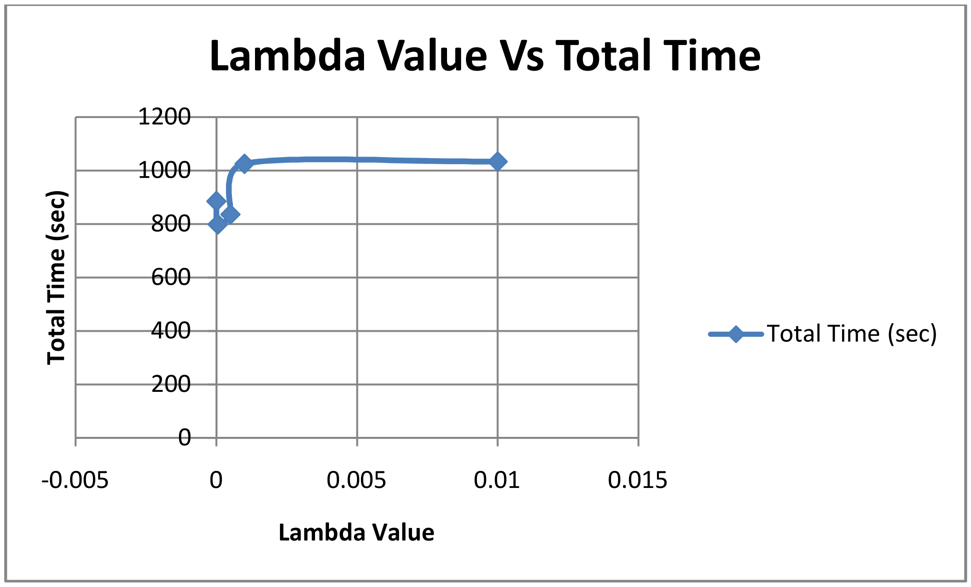
Table 5.1 – Shows the mean reward for the last 100 episodes and the total time for training when lambda is varied

The Table 5.1 above shows the values obtained for the mean reward taken for last 100 episodes and the total time taken to train when lambda is varied keeping all the other hyperparameters constant. The Graph 5.1 below graphically depicts the variation in mean reward when lambda is varied. Graph 5.2 below illustrates the relationship between lambda and the total time taken to train agent.



Graph 5.1 – Lambda Value versus Mean Reward taken for last 100 episodes

In Graph 5.1, we can observe that there is an instant surge in the mean reward when lambda is increased from 0 to 0.00005. It is intriguing to observe that when lambda is 0, the mean reward becomes a negative value. This seems logical. Notice the Formula 1.1, if lambda becomes 0, the exponential term becomes 1. Once it becomes one, the epsilon value remains at its maximum value which is one in this case and doesn't change throughout the program. This means that the agent always takes ransom actions with no opportunity for the DQN to facilitate in the prediction of its actions. With all random actions, the probability of reaching the goal decreases by a huge factor. Another interesting factor to observe here is that on increasing the lambda value by an order of 10 and 100 the mean reward not only increases, it reaches values extremely near to the maximum possible reward which is 8 which has never been reached before in the training. This means the larger the lambda value, the smaller the fractional decrement in the epsilon value. This shows the slower the epsilon value decreases the higher the mean reward. It gives more freedom for exploration and exploitation and sets the trade-off well.



Graph 5.2 – Lambda Value versus Total Time taken for training

It can be observed clearly from Graph 5.2 , though there is some unevenness in the graph when lambda values are too small that is below 0.001 but after that, there is no significant change in time despite the value of lambda increases. It can be said that, time doesn’t depend on lambda value. It is pretty logical, epsilon value helps in taking decisions, hence it effects the quality of the training which can be seen from Graph 5.1, but it surely doesn’t have any way of making the program go slower or faster which is clearly shown by Graph 5.2.

6) VARYING RANGE OF EPSILON VALUES

The range of epsilon values is an important hyperparameter because it controls the probability of taking random or Q-value of predicted values.

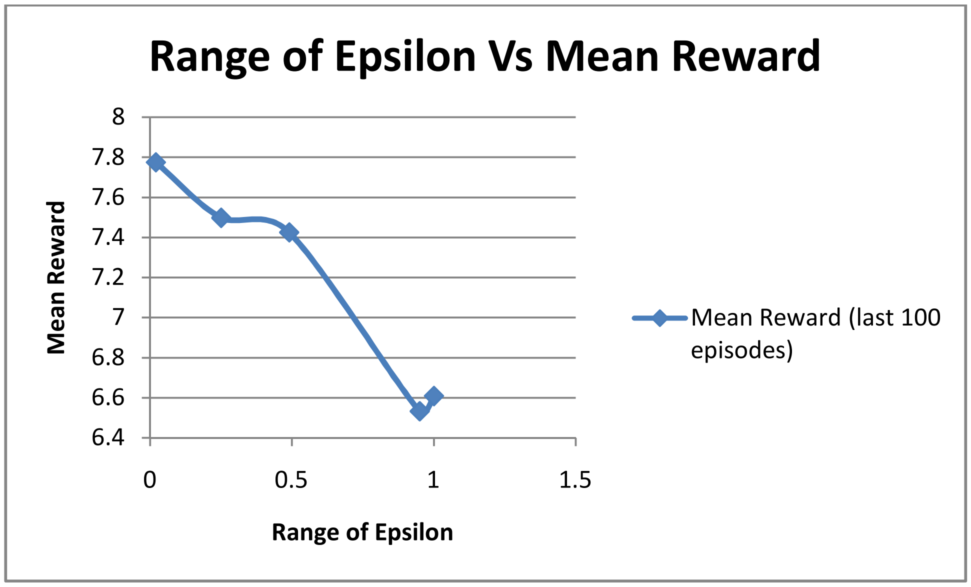
Constant hyperparameters and their values :

1. Number of Episodes : 10000
2. Lambda value (λ): 0.00005
3. Discount Factor (γ) : 0.99

Range of Epsilon (max. epsilon – min. epsilon)	**Max. Epsilon Value**	**Min. Epsilon Value**	**Mean Reward**	**Total Time (sec)**
0.02	0.07	0.05	7.773900622	775.21
0.25	0.3	0.05	7.497398225	799.63
0.49	0.5	0.01	7.424854607	752.35
0.95	1	0.05	6.532802775	798.77
1	1	0	6.608611366	784.28

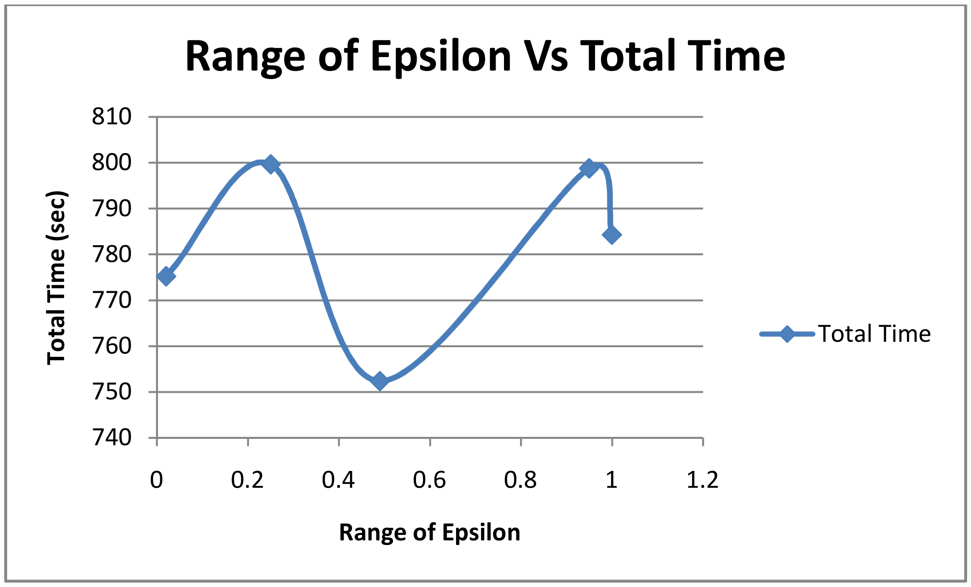
Table 6.1 – Shows the mean reward for the last 100 episodes and the total time for training when range of epsilon values is varied

The Table 6.1 above shows the values obtained for the mean reward taken for last 100 episodes and the total time taken to train when the range of epsilon values is varied keeping all the other hyperparameters constant. The Graph 6.1 below graphically depicts the variation in mean reward when the range of epsilon values is varied. Graph 6.2 below illustrates the relationship between range of epsilon values and the total time taken to train agent.



Graph 6.1 – Range of Epsilon values versus Mean Reward taken for last 100 episodes

In Graph 6.1, it can observed that a low range of epsilon values gives a very high mean reward value, very near to the highest mean reached in the hyperparameter tuning process of 7.77. It can be seen that the mean reward decreases gradually on increasing the range of epsilon values. When the range is less, there is a good trade-off between random actions and Q-value predicted actions that is because, as the range is less, the epsilon value very immediately reaches its minimum after which the probability of exploitation increases. But, as the maximum value is also small, it gives enough scope to the agent to take random actions too, that is there is also a good probability of the agent to take random actions and exploring the environment. The good trade-off shows in the high mean reward obtained. The mean reward is not too less but has decreased when the range is 1 (highest possible range for epsilon). It happens for the same opposite reasons as explained above.



Graph 6.2 – Range of Epsilon values versus Total Time taken for training

It can be observed clearly from Graph 6.2 , that there is no proper relationship between minimum value of epsilon and the total time taken to train the agent. It takes the least time when the range of epsilon values is 0.49. Range of values 0.25 and 0.95 take approximately same time.

VI. WRITING TASKS

Q1. In reinforcement learning if the agent always chooses the action that maximizes the Q-value it is always "Exploiting" and never "Exploring". "*Exploitation*" is where the agent has already gained some knowledge about the environment and uses that knowledge to further predict actions that would lead to the goal in a path which is as optimal as possible. "*Exploration*" is where the agent is simply exploring the environment and getting to know it. This is necessary as the agent does not have any information at all about the environment in which it is going to accomplish a certain goal. It has to take actions, see the results and know how the environment is through this. Hence, if the agent does not explore at all, and take actions based on its predictions, there is high possibility it will never reach the goal. IF it predicts a certain action, gets a positive result, there is high probability the NN keeps predicting the same actions and make the agent keep going in a loop never reaching the goal. This is because no matter how well the NN is designed, without exploration it doesn't know that in some other direction there is a chance of getting higher reward or reaching the goal. Taking another example, if there are two different rewards that agent can get in a maze. Reward 1 is +1 and Reward 2 is +100. The cells near the agent's starting position have a reward of +1, and the cells which are farther have reward of +100. If the agent never explores, the agent will keep moving in a loop in the cells near to it as it will keep getting positive results, but it missing on the major reward and hence cannot get the highest reward possible ever as it does not know about the +100 as there is no way other than exploration to know this information.

Two ways to force the agent to explore –

1. ϵ-Greedy Approach: This approach combines *"Exploration"* and *"Exploration"*. The ϵ value acts as the control for implementing the trade-off between *"Exploration"* and *"Exploitation"*. A threshold for ϵ is defined which is also its maximum value. A number is chosen randomly, if the number is less than the current ϵ, the actions of the agent are chosen at random. Else, the Q-values (predicted using the DQN) are used to decide which action to be taken. The former method is "*Exploration"* the latter is "*Exploitation"*.
2. Boltzmann Exploration: In this approach, the agent explores by learning everything that the estimated Q-values can tell it. Q-values are divided by a temperature parameter (τ) and a softmax function is taken over this value. What this essentially does is gives weights to each of the Q-values of a state rather than considering all the non-optimal actions (actions which do not have the maximum Q-value) equally, it gives weights to them and considers some actions more important than others. This way the agent is exploring to know about other non-optimal but important actions.
3. Random actions: The agent can just keep on taking random actions throughout, which is nothing but exploration. But, **this method is not good** because it doesn't allow the agent to exploit which is also important. **The key is to strike the correct balance between the two.**

Reference: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf

Q2. Using the following formula calculate Q-values , given $\gamma = 0.99$

$$\boldsymbol{Q(s_t\,, a_t) = r_t + \gamma * max_a Q(s_{t+1}\,, a)}$$

Start from the last state s_4, all Q values for this state will be 0 as in this state Tom (Agent) reaches its goal and will no longer move and there is no future expected reward.

1. $Q(s_4, up)$ = 0
2. $Q(s_4, down)$ = 0
3. $Q(s_4, left)$ = 0
4. $Q(s_4, right)$ = 0

$\max_a Q(s_4 , a) = 0$

Next move to state s_3 –

1. $Q(s_3, up)$ = -1 + 0.99*$\max_a$Q(s' , a) = -1 + 0.99*$\max_a$Q(s_2 , a) = -1 + 0.99*1.99 = -1 +1.9701 = 0.9701

On moving up from state s_3, agent reaches a state s' which is symmetric to state s2 as it takes optimally two steps to reach goal from s2 and same is the case with the state s'. As it is symmetric to s2, the q-values from state s' will be the same as the q values from s2. Hence, maximum of the q-values of state s2 can be taken in place of maximum of the q-values from state s'.
This value is substituted once state s2 max q-value is calculated further below.
Instant reward is -1 as moving away from goal.

2. $Q(s_3, down)$ = 1 + 0.99*max$_a$Q(s_4 , a) = 1 + 0.99*0 = 1 + 0 = **1**
On moving down from s3, agent reaches s4. So we take maximum of q-values of state s4. **This action is in the optimal path towards the goal hence this q-value is the maximum for state s3**. Instant reward is +1 as moving towards goal.
3. $Q(s_3, left)$ = -1 + 0.99*max$_a$Q(s_2 , a) = -1 + 0.99*1.99 = -1 +1.9701 = 0.9701
On moving left from s3, agent reaches state s2 so take maximum of q-values of state s2 which will be substituted once calculated further below. Instant reward is -1 as moving away from goal.
4. $Q(s_3, right)$ = 0 + 0.99*max$_a$Q(s_3 , a) = 0 + 0.99*1 = 0.99
On moving right from s3, agent is trying to go out of grid which means it is not moving at all hence future state remains s3 only. Hence, take maximum q-value of state s3 which is 1 as calculated in point number 2 above. Instant reward is 0 as not moving at all.

max$_a$Q(s_3 , a) = 1

Next move to state s2 –

1. $Q(s_2, up)$ = -1 + 0.99*max$_a$Q(s_1 , a) = -1 + 0.99*2.9701 = -1 +2.9404 = 1.9404
On moving up from state s2, the agent reaches state s1. Hence take maximum of the q-values of the state s1 which is calculated further below. Instant reward is -1 as moving away from goal.
2. $Q(s_2, down)$ = 1 + 0.99*max$_a$Q(s' , a) = 1 + 0.99*max$_a$Q(s_3 , a) = 1 + 0.99*1
= 1 +0.99 = 1.99
On moving down from state s_2, agent reaches a state s' which is symmetric to state s3 as it takes optimally one step to reach goal from s3 and same is the case with the state s'. As it is symmetric to s3, the q-values from state s' will be the same as the q values from s3. Hence, maximum of the q-values of state s3 can be taken in place of maximum of the q-values from state s'.
We know state s3 max q-value is 1 from above. Instant reward is 1 as moving towards goal.
3. $Q(s_2, left)$ = -1 + 0.99*max$_a$Q(s'' , a) = -1 + 0.99*max$_a$Q(s_1 , a) = -1 + 0.99*2.9701
= -1 +2.9404 = 1.9404
On moving left from state s_2, agent reaches a state s'' which is symmetric to state s1 as it takes optimally three steps to reach goal from s1 and same is the case with the state s''. As it is symmetric to s1, the q-values from state s'' will be the same as the q values from s1. Hence, maximum of the q-values of state s1 can be taken in place of maximum of the q-values from state s''.

This value is substituted once state s1 max q-value is calculated further below. Instant reward is -1 as moving away from goal.

4. $Q(s_2, right)$ = 1 + 0.99*$\max_a Q(s_3, a)$ = 1 + 0.99*1 = **1.99**
   On moving right from s2, agent reaches s3 hence take the maximum q-values for the state s3 which is 1 as calculated above. Instant reward is +1 as moving toward goal. **This action is in the optimal path towards the goal hence this q-value is the maximum for state s2**.

$\max_a Q(s_2, a)$ = 1.99

Next move to state s1 –

1. $Q(s_1, up)$ = 0 + 0.99*$\max_a Q(s_1, a)$ = 0 + 0.99*2.9701 = 2.9404
   On moving up from s1, agent is trying to go out of grid which means it is not moving at all hence future state remains s1 only. Hence, take maximum q-value of state s1 which will be calculated in point number 2 below. Instant reward is 0 as not moving at all.
2. $Q(s_1, down)$ = 1 + 0.99* $\max_a Q(s_2, a)$ = 1 + 0.99*1.99 = 1 + 1.9701 = **2.9701**
   On moving down from s1, the agent reaches state s2, hence take the maximum q-value of the state s2 which is 1.99 as per the calculation above. Instant reward is +1 as moving towards goal. **This action is in the optimal path towards the goal hence this q-value is the maximum for state s1**.
3. $Q(s_1, left)$ = -1 + 0.99* $\max_a Q(s_0, a)$ = -1 + 0.99*3.9404 = -1 + 3.9 = 2.9
   On moving left from state s1, the agent reaches state s0. Hence take maximum of the q-values of the state s0 which is calculated further below. Instant reward is -1 as moving away from goal.
4. $Q(s_1, right)$ = 1 + 0.99*$\max_a Q(s', a)$ = 1 + 0.99*$\max_a Q(s_2, a)$ = 1 + 0.99*1.99 = 1 +1.9701 = 2.9701
   On moving right from state s_1, agent reaches a state s' which is symmetric to state s2 as it takes optimally two steps to reach goal from s2 and same is the case with the state s'. As it is symmetric to s2, the q-values from state s' will be the same as the q values from s2. Hence, maximum of the q-values of state s2 can be taken in place of maximum of the q-values from state s'.
   We know state s2 max q-value is 1.99 from above. Instant reward is +1 as moving towards goal.

$\max_a Q(s_1, a)$ = 2.9701

Next move to state s0 –

1. $Q(s_0, up)$ = 0 + 0.99*$\max_a Q(s_0, a)$ = 0 + 0.99*3.9404 = 3.9
   On moving up from s0, agent is trying to go out of grid which means it is not moving at all hence future state remains s0 only. Hence, take maximum q-value of state s0 which will be calculated in point number 4 below. Instant reward is 0 as not moving at all.

2. $Q(s_0, down)$ = 1 + 0.99*max$_a$Q(s' , a) = 1 + 0.99*max$_a$Q(s$_1$, a) = 1 + 0.99*2.9701 = 1 + 2.9404 = 3.9404
   On moving down from state s$_0$, agent reaches a state s' which is symmetric to state s1 as it takes optimally three steps to reach goal from s3 and same is the case with the state s'. As it is symmetric to s1, the q-values from state s' will be the same as the q values from s1. Hence, maximum of the q-values of state s1 can be taken in place of maximum of the q-values from state s'. We know state s1 max q-value is 2.9701 from above. Instant reward is 1 as moving towards goal.
3. $Q(s_0, left)$ = 0 + 0.99*max$_a$Q(s$_0$, a) = 0 + 0.99*3.9404 = 3.9
   On moving left from s0, agent is trying to go out of grid which means it is not moving at all hence future state remains s0 only. Hence, take maximum q-value of state s0 which will be calculated in point number 4 below. Instant reward is 0 as not moving at all.
4. $Q(s_0, right)$ = 1 + 0.99* max$_a$Q(s$_1$, a) = 1 + 0.99*2.9701 = 1 + 2.9404 = **3.9404**
   On moving right from s0, the agent reaches state s1, hence take the maximum q-value of the state s1 which is 2.9701 as per the calculation above. Instant reward is +1 as moving towards goal. **This action is in the optimal path towards the goal hence this q-value is the maximum for state s0**.

max$_a$Q(s$_0$, a) = 3.9404

ACTIONS → / STATE ↓	Up	Down	Left	Right
0	3.9	3.94094	3.9	3.9404
1	2.9404	2.9701	2.9	2.9701
2	1.9404	1.99	1.9404	1.99
3	0.9701	1	0.9701	0.99
4	0	0	0	0

Table 7.1: Q-Table (for the states given in question 2)

****