Project 4: Tom and Jerry in Reinforcement Learning

Emmanuel Johnson Gnana Chandra Johnson
Masters in Computer Science and Engineering
State University of New York at Buffalo
egnanach@buffalo.edu

1 Reinforcement Learning

Reinforcement Learning is an aspect of machine learning where an agent learns to behave in an environment, by performing certain actions and observing the rewards which it get from those actions. This results in a policy that maps the state of the agent to the actions which is used to maximize the long term total reward. The reinforcement learning process can be considered as an iterative loop where an agent receives a state S^0 from the environment and it takes an action A^0 and moves to a new state S^1 in the environment. For this, the environment gives a certain reward R^1 to the agent. The basic aim of the reinforcement learning agent is to maximize the reward. A sequence of states, actions and rewards, which ends with terminal state or goal is called as an Episode. A basic reinforcement learning is modeled as a Markov decision process. As seen in figure 1, the Markov decision process is a 5-tuple which consists of finite set of states S, finite set of actions A, the reward given to the agent R, a discount factor γ and the probability P that an action in a state will lead to a new state.

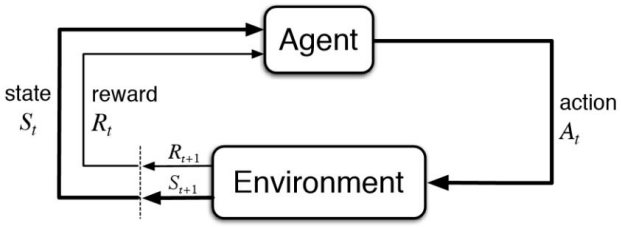


Figure 1: Basic Flow of Reinforcement Learning

2 Deep Q-Learning

Q-learning is a values-based learning algorithm in reinforcement learning. The Q-learning uses a Q-table which has all the maximum expected future rewards for action at each state calculated. Using this table as a reference we can take the best action for a given state. In the Q-table, the columns are the actions and the rows are the states. At first all the values are initialized to zero. The values in the Q-table are calculated using a Q-function that uses the bellman equation. It takes a state and action as the input. We can also use a neural network instead of the Q-table to determine the optimal action. The Q-function is used to calculate the Q-value and this value is used as a training target for our neural network along with the states as input. This is known as

Deep Q-Network. Here a neural network is fed with actions and state as input and it gives the best possible action that can be performed by the agent for the given state.

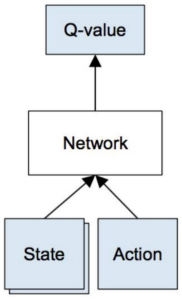


Figure 2: Deep Q-Network

3 Flow of Code

3.1 Environment

At first we define our environment. The environment that we set up is a 5X5 grid world. The environment has 25 possible states. There are four actions such as left, right, up and down. Tom (the cat) is our agent and jerry (the mouse) is our goal. Our goal is to make the agent reach the goal in the shortest path. There are two possible rewards that the agent can receive. The agent receives a reward 1 if it progresses towards the goal or if it had reached the goal. The agent receives a reward -1 if it does not move or if it moves in the opposite direction of the goal. The game is over if the agent has reached the goal or if the time becomes zero.

3.2 Brain and Memory

Here we create our neural network model for the Deep Q-Network. The network takes a stack of six tuple as the input and gives a vector of all the possible actions for the given state. The action with the highest Q-value is chosen. We have a memory that stores all the previous experiences of our agent. This is used to train our Deep Q-Network model and give better predictions. Initially our agent performs poor because of less experience, but as it progresses our agent becomes better and performs good.

3.3 Agent

At first our agent hasn't explored our environment. Hence it takes random actions to start exploring the environment. This random action is taken by a certain rate called epsilon or exploration rate. When the agent doesn't select an action by random, it will pick the action that will give the highest reward with the help of our Deep Q-Network. At first the number of random actions are more and we want this to eventually decrease. Hence the epsilon value is exponentially reduced using the exponential decay formula, $\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\lambda |S|}$.

4 Implementations

4.1 Neural Network

As we have seen in the previous section, the neural network serves as the brain of our agent. The neural network that we've built has 3 layers. Two of them are the hidden layers and the third is our output layer. The hidden layers have 128 as the default number of nodes. This is a hyperparameter and can be modified to produce better results. The activation function used here is relu. These first hidden network's input dimensions are equal to the number of states given as the input. The number of nodes in the output layer is four. This is because we have only four possible actions (up, down, left or right). Since these are real values, we use the linear activation function in the output layer. This linear activation function is nothing but an identity function.

```
model.add(Dense(128, input_dim=self.state_dim))
model.add(Activation('relu'))
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dense(4))
model.add(Activation('linear'))
```

4.1.1 Role in training the agent

The sequential neural network which acts as the Deep Q-Network in our algorithms serves as the brain of the agent. When the agent wants to take an action that maximizes the reward, it uses the model that we've trained to take an optimal action. The neural network will give all the possible Q-values for a given state. The highest Q-value represents the best action (up, down, left or right) to be taken. Usually a Q-table is used as a lookup to identify the best action. But when the number of states and actions are too many, the computation of Q-table becomes very high. The Deep Q-network reduces the computation of finding the best possible action for a given state even when there are thousands of possible actions.

4.1.2 Improving the approach

The three layer neural network that we've implemented performs well. The code snippet is simple and straightforward here. The only improvement that we can do here is to change the hyperparameters like the number of nodes, the number of hidden layers, the activation functions used; to see if there is any significant impact in our model. This hyperparameter tuning of the neural network is discussed in section 5.7. In our current algorithm, we re-train our Deep Q-network during each episode which could destabilize the learning process and magnifies the Q-value. To avoid this we can reduce the changes involved in the input and re-train the network for every 1000 episodes. This will result in a better generalization of our neural network.

4.2 Exponential Decay Formula

The exponential decay formula is given as,

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

Here $\epsilon_{min}, \epsilon_{max} \in [0,1]$, λ is the rate at which epsilon decays and S is the state of the agent. The python code for the above formula is given below,

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

4.2.1 Role in training the agent

The epsilon max value helps our agent to explore the environment. The agent decides to explore or exploit depending on the value of epsilon. Exploration is all about finding more information about an environment, whereas exploitation is exploiting already known information to maximize the rewards. For example if we do online shopping only at amazon.com, then its exploitation. But when we explore other ecommerce websites we might find better deals and this is exploration. Initially we want our agent to explore more and find patterns in our environment and later exploit to maximize the reward. An agent decides between exploration and exploitation depending on the present epsilon value and a random value that we generate. If the random value is less than the epsilon, then the agent will explore else it will exploit. The epsilon value is reduced during each iteration at the rate of lambda (λ) and when it reaches its minimum value, the agent will stop exploring and start exploiting the environment.

4.2.2 Improving the approach

In our algorithm we've implemented the greedy epsilon to decide whether to exploit or explore. In the exponential decay formula we can try altering the hyperparameter lambda. This will affect how fast our epsilon will reduce which in turn affect how fast our agent will start prefer exploiting over exploration. Another approach is to use softmax action selection to decide between exploration and exploitation. In softmax we bias exploration towards promising actions. Although greedy epsilon is an effective means of balancing exploration and exploitation, one drawback is that it chooses equally among all actions. This means that it is as likely to choose the worst-appearing action instead of the next-to-best action. This can be avoided if we try to vary the action probabilities as a graded function of estimated value. The greedy action is still given the highest selection probability, but all the others are ranked and weighted according to their value estimates. These are called softmax action selection rules. The most common softmax method uses a Gibbs, or Boltzmann, distribution. The softmax function is given as follows,

$$\pi(a|s) = \frac{e^{\frac{Q(s,a)}{\tau}}}{e^{\sum_{a' \in \mathcal{A}} \frac{Q(s,a')}{\tau}}}$$

t is known as the temperature and is always positive. If the value of t is high, it causes the actions to be all equiprobable. If the value of t is low, it causes a greater difference in selection probability for actions that differ in their value estimates.

4.3 Q-function

The Q-function is given as,

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

Here Q_t is the Q-value, r_t is the reward, γ (gamma) is the rate at which the reward is discounted, s_t is the state, a_t is the action and t represents the time. The python code for the above function is as follows,

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

There are two conditions given for the Q-function. If the current state is the last step ie., there is no next step/state, then the reward is returned as the Q-value. But if there is a next step, we calculate the discounted rewards that can be expected (reward plus the maximum reward in the next step discounted by the discount factor gamma).

4.3.1 Role in training the agent

The Q-function takes in state and action as a parameter. The Q-function gives the expected total reward for the agent when starting at a state 's' and perform an action 'a'. The Q-function tells the agent how good of a choice is an action 'a' for a given state 's'. This Q-function is used to draw a Q-table which is a simple lookup table where maximum expected future rewards for an action at each state is computed and kept. By using this table, out agent can decide which action to perform by picking the action which has the highest Q-value in the table. In our algorithm, we use the Q-function to train our Deep Q-network. The generated Q-values serves as the training target for our neural network.

4.3.2 Improving the approach

In the Q-function we can try modifying the gamma hyperparameter. The gamma is the reward discount factor which has effects on the long term and short term goals of the agent. When the gamma value is high, the agent will start focusing towards the long term goals which in our case Tom will try to catch Jerry. To make the agent focus on short term goals we can reduce gamma value. But it is desired for the gamma value to be high. Another improvement can be to try and use the value based approach instead of policy based approach.

5 Hyperparameter Tuning

There are a lot of hyperparameters that can be tuned to give better results. The hyperparameters include the maximum epsilon, minimum epsilon, lambda, number of episodes and gamma.

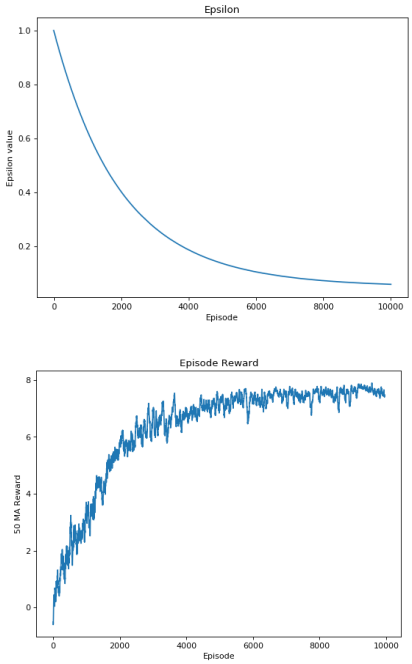
5.1 Model with default values

We start our model with some default hyper parameters. Later we'll be tuning these parameters to get an improved performance as well as learn how these parameters affect the performance of the model. The table 1 shows the default network settings that our model has.

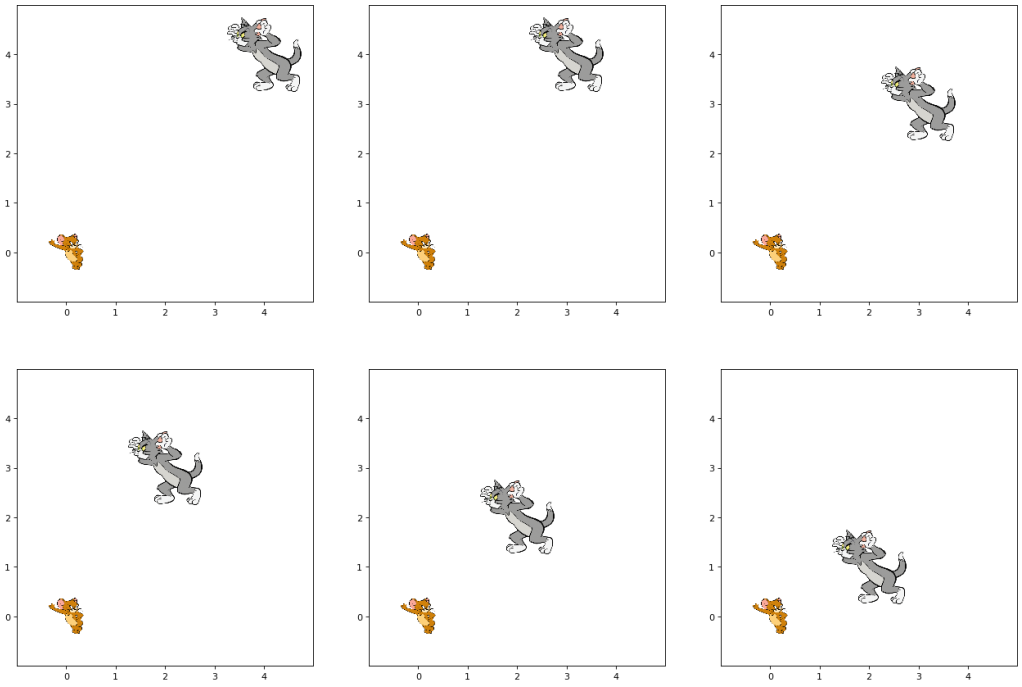
Table 1: Default Model Settings

Hyperparameter	**Value**
Episodes	10000
Max Epsilon	1
Min Epsilon	0.05
Lambda	0.00005
Gamma	0.99
First Layer Activation	Relu
Second Layer Activation	Relu
Hidden Layers	2

Figure 3: Default Model Performance



The default model takes about 7000 episodes to converge and for every 100 episodes, the model takes approximately 3.3 seconds to compute. And the mean rolling reward is 6.259. The graphical representation of our agent (Tom) to reach its goal (Jerry) is shown below. Each figure from left to right shows our agent's approach towards its goal at each step.



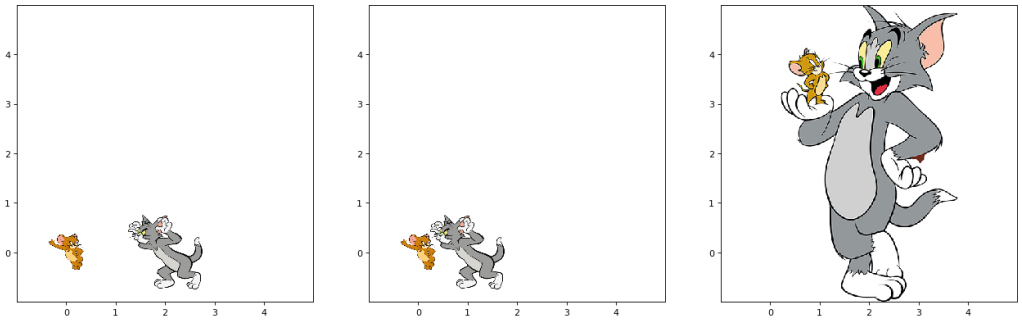


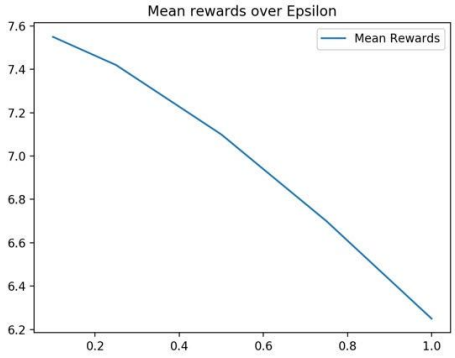
Figure 3: Steps taken by the agent to reach the goal

5.2 Changing Max Epsilon

Initially in our default setting the value of max epsilon is 1. We experimented the performance of the model when the value of max epsilon is changed to 0.75, 0.5, 0.25 and 0.1. The value of max epsilon is used by the agent to determine whether it should explore or exploit. Usually the max epsilon value starts at a higher value and gradually decays. When the random value generated by our algorithms is less than the max epsilon, the agent starts to explore. Changing the value of the max epsilon to a smaller value results in the agent to start exploiting the environment faster. From the graphs we can observe that the agent starts to receive higher reward in less episodes as the value of max epsilon is reduced. Ideally we would require this to be gradual and hence a much higher max epsilon value is recommended.

Table 2: Change in Episodes and Mean Reward when Max Epsilon value changes

Max Epsilon	0.75	0.5	0.25	0.1
Episodes	6000	4000	3000	500
Mean Reward	6.70	7.10	7.42	7.55



The above graph shows the relationship between the epsilon and the mean rewards. As the epsilon value increases, the value of mean rewards decreases.

When we change the value of the max epsilon, we reduce the range between the min and the max epsilon. This directly affects the agent's exploration and exploitation trade off. From the table 2 we can see that as the epsilon value is reduced, the mean reward that the agent receives increases. This means that the agent has exploited more than exploration. The number of episodes to converge also reduces drastically. This is observed in the graphs in figure 4.

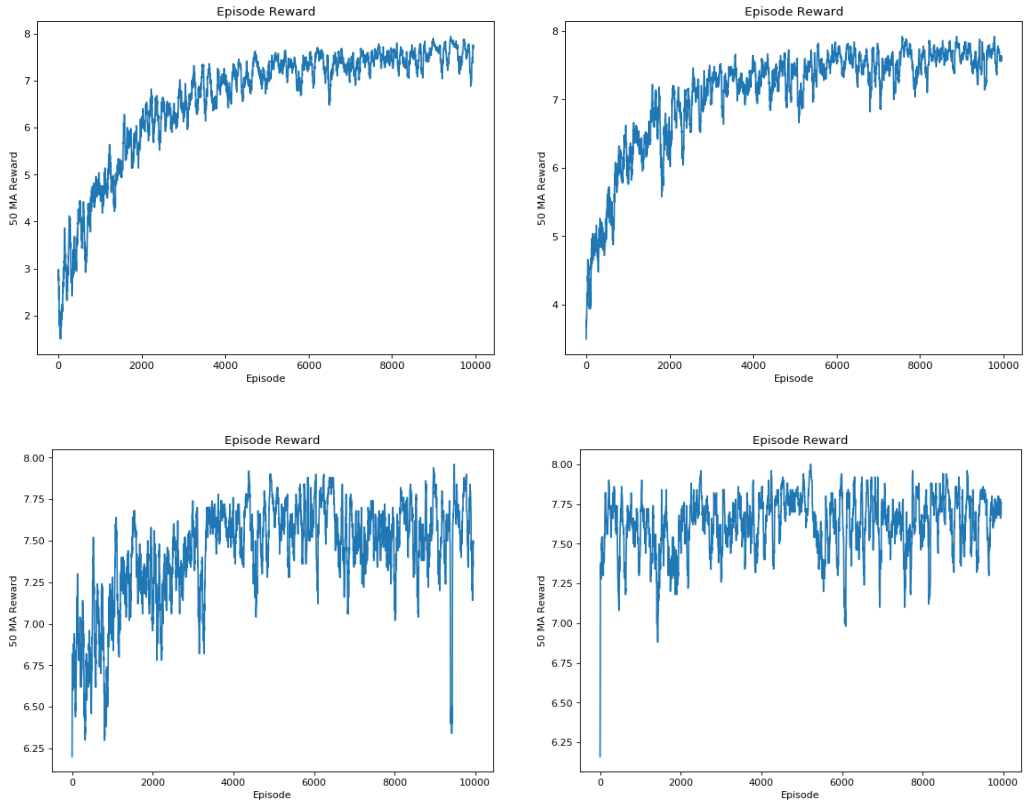


Figure 4: Reward vs Episode for various values of Max Epsilon 0.75, 0.5, 0.25, 0.1(left to right)

5.3 Changing Min Epsilon

In our default setting the min epsilon's initial value is 0.05. The min epsilon indicates the atmost minimum value that the epsilon can reach when it is decayed during each episodes. We experimented the performance of the model when the min epsilon value is changed to 0.5 and 0.85. Increasing the value of the min epsilon produces almost similar results to the scenario where we reduced the value of max epsilon. The random value generated by the algorithm should lie within the range of min and max epsilon. When the min epsilon value rises, the agent starts to exploit the environment and since it hasn't explored enough, the rewards that the agent receives is poor. Hence the min epsilon value should be much smaller compared to the value of max epsilon.

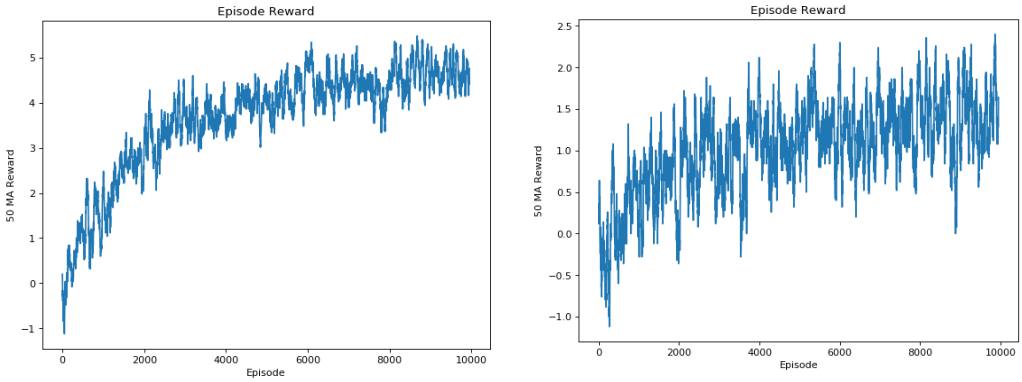


Figure 5: Reward vs Episode for various values of Min Epsilon 0.5, 0.85 (left to right)

5.4 Changing Lambda

Initially in our default setting the value of lambda is 0.00005. We experimented the performance of the model when the value of lambda is changed to 0.0005 and 0.05. As we've seen earlier lambda is the rate at which the epsilon value decays. By increasing the value of lambda we reduce the value of max epsilon much faster. This makes our agent to stop exploring the environment and start exploitation. From the graphs below we can see that the epsilon value reaches its minimum value before 1000 episodes when lambda is 0.0005 and even faster when 0.05. This results in the agent to stop exploring much faster and start exploitation. This can be seen in the reward graph were the we see a sudden increase in the rewards and it remains above 7 almost from the beginning. If the lambda value is much smaller, then our agent will take most of its time in exploration and this too is undesirable. Ideally we want our agent to explore the environment as well as exploit it in a balanced manner. Hence the default value of lambda works best for this model.

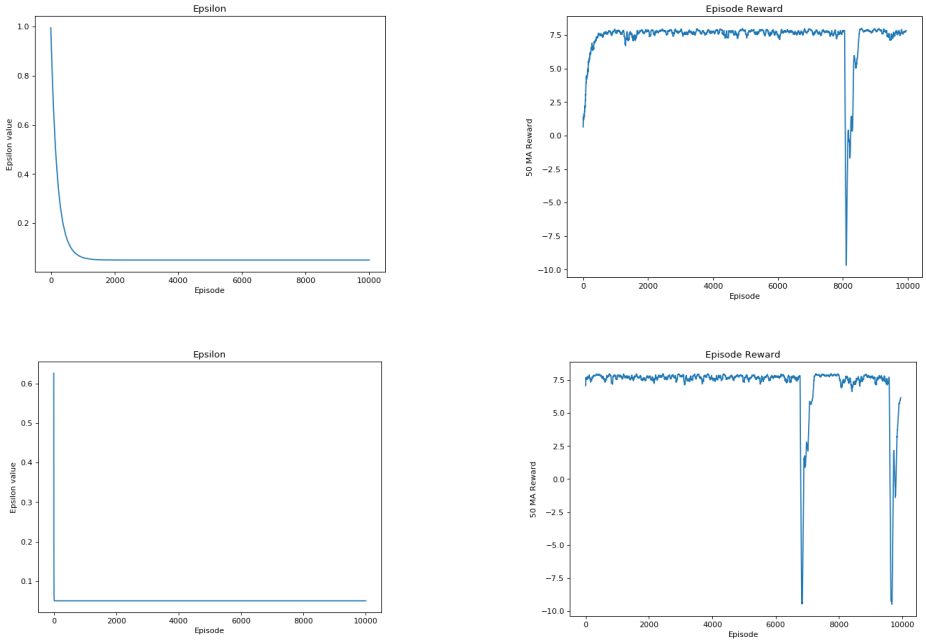


Figure 6: Epsilon vs Episode (left) and Reward vs Episode (right)
for Lambda 0.0005, 0.05 (top to bottom)

5.5 Changing number of episodes

Reducing the number of episodes has a direct impact in the performance of our model. Initially the value of episodes is 10000. We experimented the performance of the model by reducing the episodes to 5000 and 1000. From the graphs we can see that, as the number of episodes are reduces the epsilon value doesn't decay much and hence the agent doesn't exploit and get better rewards. When the episode is much smaller the agent has not been trained well and reaches to a maximum reward of 3. Hence we need more episodes to train our agent effectively.

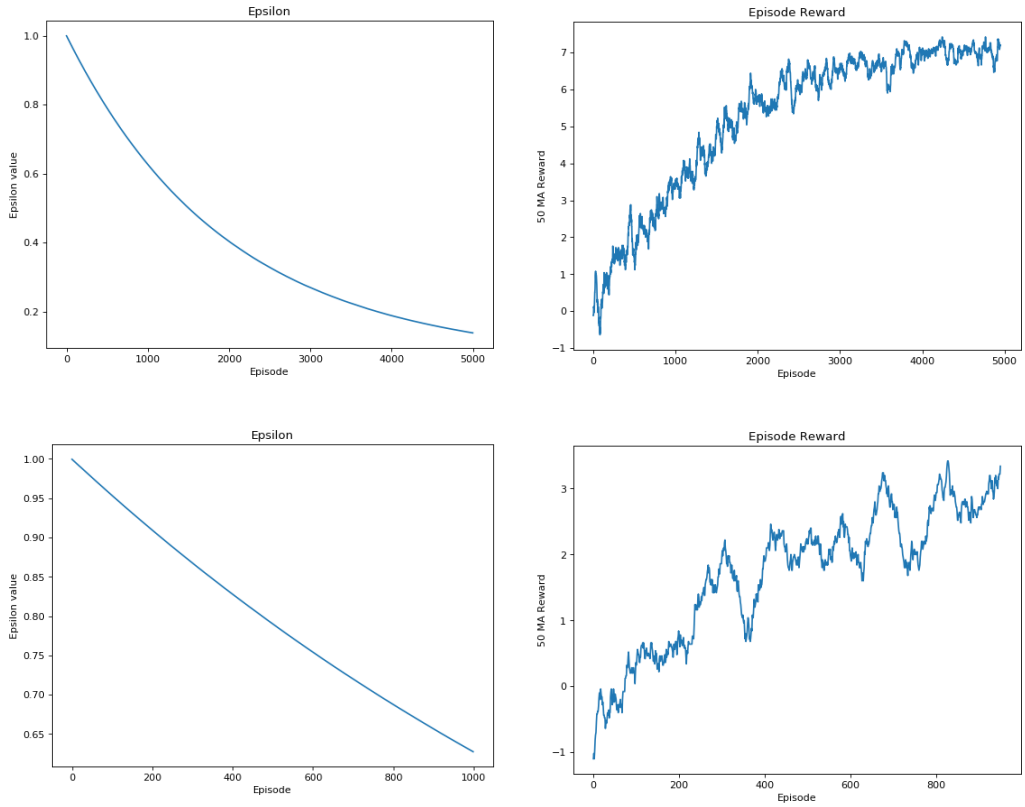


Figure 7: Epsilon vs Episode (left) and Reward vs Episode (right)
for Episodes 5000, 1000 (top to bottom)

5.6 Changing gamma value

The gamma value is the rate at which the future rewards are discounted. We experimented for different value of gamma and observed how it affected the performance of the model. The below graphs show the results for gamma values 0.99 (default), 0.9, 0.5 and 0.1. From the graphs we can see that there is no significant impact of the gamma value in our environment. This is because the max reward that our agent received in the given environment is 1. Thus it eventually reaches the goal even when the long term goal is uncertain. Ideally when the gamma value is much smaller, the agent starts to focus more on the short term rewards than the long term rewards. If our environment is much more complex like atari games, we would have seen an significance impact of the gamma value on the model. But in our environment the value of gamma doesn't have much of an impact and it can be left to its default value of 0.99.

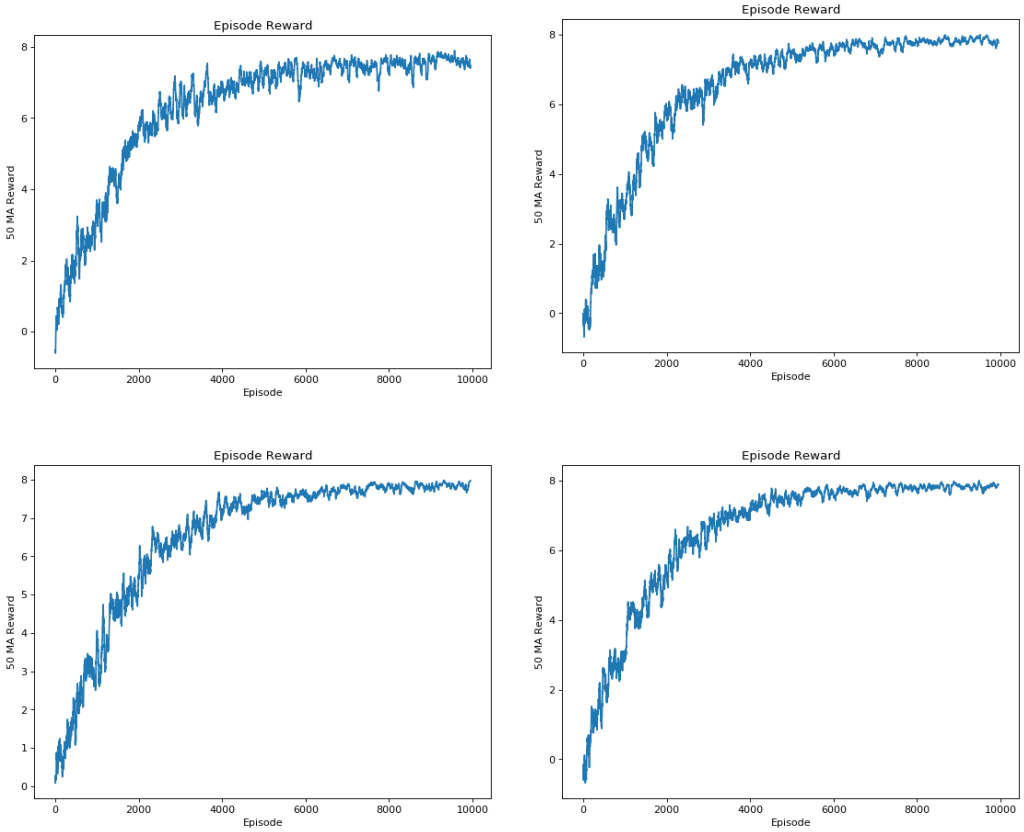


Figure 8: Reward vs Episode for various values of Gamma 0.99, 0.9, 0.5, 0.1 (left to right)

5.7 Changing number of hidden layers in neural networks

5.7.1 Three hidden layers

Initially our model has a three layer neural network with two hidden layers. The role of the neural network is to predict the best possible action for the agent given a state and time. When the model was experimented with three hidden layers, the third layer having 128 nodes and relu as its activation function, the below results are observed. The mean rolling reward is 6.47 and the model starts to converge around 3500 episodes.

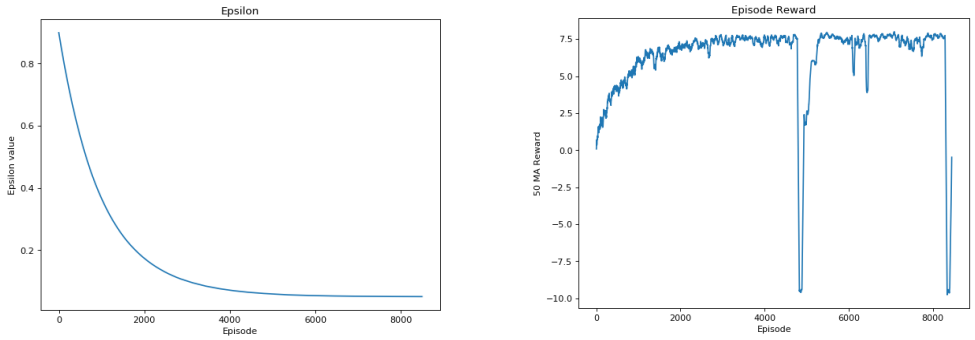


Figure 9: Four layer neural network with three hidden layers

5.7.2 Four hidden layers

Like the previous step, we add one more hidden layer with 128 nodes and relu as the activation function into our model. We observe the following results,

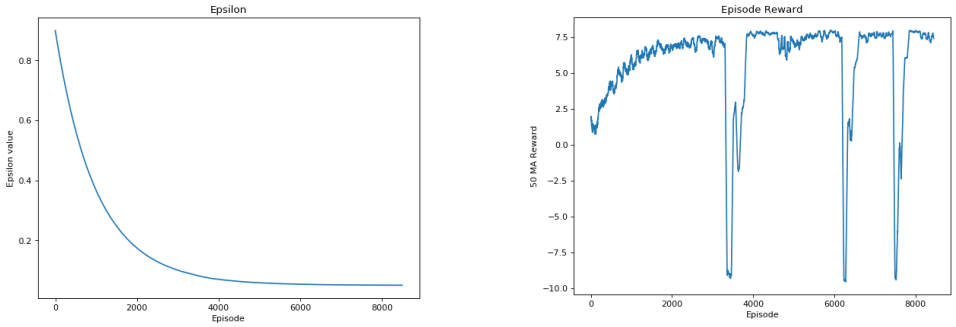


Figure 10: Five layer neural network with four hidden layers

The mean rolling reward is 5.59 and the model starts to converge around 2500 episodes.

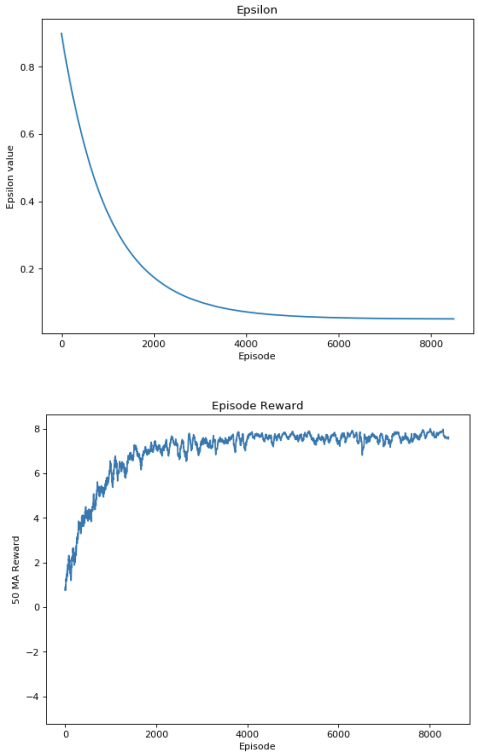
5.8 Optimal Modal

By observing the results of the hyperparameter tuning we conclude the below setting as optimal for the given environment.

Table 3: Optimal Model Settings

Hyperparameter	**Value**
Episodes	8000
Max Epsilon	0.9
Min Epsilon	0.1
Lambda	0.001
Gamma	0.99
First Layer Activation	Relu
Second Layer Activation	Relu
Hidden Layers	2

Figure 11: Optimal Model Performance

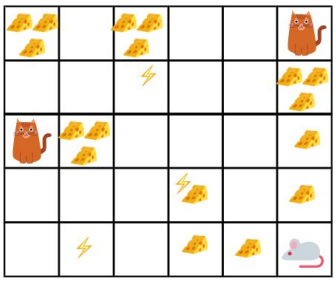


With these optimal settings we converge much faster at around 4000 episodes. The range between the max and min epsilon is also ideal to balance exploration and exploitation. The lambda value is not much lower or higher and is ideal. The two layer neural network also helps in the faster convergence and gives better predictions in shorter episodes. This optimal settings provides good results and we get a rolling mean reward of 6.97.

6 Writing Tasks

6.1 What happens when an agent always chooses actions that maximizes the Q-value?

When an agent always chooses the action that maximizes the Q-value the agents gets stuck in non-optimal policies and it does not explore the environment to find a more optimal path towards the goal or an optimal action from the given state. The Q-function corresponds to learning the optimal policy. So when an agent always chooses the maximum value and if the agent hasn't explored the environment completely, the action that the agent takes is not optimal. Hence it does not find the global optimal policy. Consider the below example,



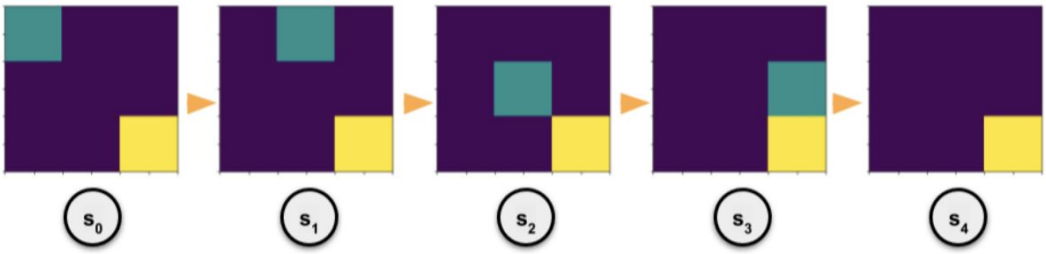
In this environment the mouse has to avoid the cat and get the cheese. When it starts picking the actions that maximizes the Q-value and when it hasn't explored the environment, it always tries to get the cheese which is near to this. But this is not optimal because if the agent had explored the environment, it would get better rewards where there are a lot of cheese in the top left corner of the environment.

6.2 Ways to force the agent to explore

There are few ways in which we can force the agent to explore the environment more rather than to exploit.

1. In the greedy epsilon approach, if we keep the value of the max epsilon to a higher value and if we keep the value of the min epsilon to a much smaller value, we can make the agent to explore the environment more. The decision between exploration and exploitation happens when the random value generated (between 0 to 1) is compared with the epsilon value. If the random value is less than the epsilon value, then the agent chooses to explore. Hence the much higher the epsilon value, the more probability that our agent will explore the environment more.
2. Another way is to make the value of lambda or the rate of decay to a much smaller value. This makes the value of max epsilon to reduce at a much slower rate and hence keeps the range between the min and max epsilon larger for longer episodes during the training process. Hence making lambda value much smaller will make the random value to lie within the epsilon range for longer steps and makes our agent to explore the environment more.

6.3 Calculate the Q-value for the given states



6.3.1 Final Solution

Table 4: Final Q-Table

	Up	Down	Left	Right
S_0	3.9008	3.9403	3.9008	3.9403
S_1	2.9404	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

6.3.2 Calculations

The handwritten scanned copy of the calculations are attached in this report.

WRITING TASK

QUESTION 2

Q TABLE CALCULATION:

STEP 1: In the initial step we fill the table in a bottom up approach. Totally there are five states S_0, S_1, S_2, S_3 and S_4.

	↑	↓	←	→
S_0	0		0	
S_1	0			
S_2				
S_3		1		0
S_4	0	0	0	0

State S_4: Since the goal is already reached is S_4, there is no action required. Hence everything is 0.

State S_3: Next we can go down and we will get a reward 1 Right will give a 0 reward because the next state is unknown.

State S_2 : All actions are possible in this state.

State S_1 : From this state ~~up~~, right, down & left actions are possible. Up will give a 0 reward since the next state is unknown.

State S_0 : Here up and left actions give 0 reward.

STEP 2:

	↑	↓	←	→
S_0	0	3.9403	0	3.9403
S_1	0	2.9701	2.9008	2.9701
S_2	1.9403	1.99	1.9403	1.99
S_3	0.9701	1	0.9701	0
S_4	0	0	0	0

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

+VE REWARD = 1 -VE REWARD = -1

NO REWARD = 0

State S_t: Since the goal is already reached, there is no action and everything is 0.

State S_3:

(i) $Q(S_3, a_\uparrow) = -1 + 0.99 \times \max Q(S_2, a)$

This is because $\max Q(S_2, a)$ is symmetric with $Q(S_3, a_\uparrow)$

(ii) $Q(S_3, a_\leftarrow) = -1 + 0.99 \times \max Q(S_2, a)$

State S_4:

(iii) $Q(S_2, a_\uparrow) = -1 + 0.99 \times \max_a Q(S_1, a)$
(This is because of symmetry)

(iv) $Q(S_2, a_\downarrow) = 1 + 0.99 \times \max_a Q(S_3, a)$

(v) $Q(S_2, a_\leftarrow) = -1 + 0.99 \times \max_a Q(S_1, a)$

(vi) $Q(S_2, a_\rightarrow) = 1 + 0.99 \times \max_a Q(S_3, a)$

State S_1:

(vii) $Q(S_1, a_\downarrow) = 1 + 0.99 \times \max_a Q(S_2, a)$

(viii) $Q(S_1, a_\leftarrow) = -1 + 0.99 \times \max_a Q(S_0, a)$

(ix) $Q(S_1, a_\rightarrow) = 1 + 0.99 \times \max_a Q(S_2, a)$

State S_0:

(x) $Q(S_0, a_\downarrow) = 1 + 0.99 \times \max_a Q(S_1, a)$

(xi) $Q(S_0, a_\rightarrow) = 1 + 0.99 \times \max_a Q(S_1, a)$

From (iv) & (vi)

$Q(S_2, a_\downarrow) = 1 + 0.99 \times 1$

$Q(S_2, a_\downarrow)$ & $Q(S_2, a_\downarrow) = 1.99$

$Q(S_3, a_\uparrow) = -1 \times 0.99 (1.99)$

$= 0.9701$

$Q(S_3, a_\leftarrow) = 0.9701$

$Q(S_1, a_\downarrow) = 1 + 0.99 \times 1.99$

$= 2.9701$

$Q(S_1, a_\downarrow) = 1 + 0.99 \times 1.99$
$= 2.9701$

$Q(S_2, a_\uparrow) = -1 + 0.99 \times 2.9701$
$= 1.9403$

$Q(S_2, a_\leftarrow) = -1 + 0.99 \times 2.9701$
$= 1.9403$

$Q(S_0, a_\rightarrow) = 1 + 0.99 \times 2.9701$
$= 3.9403$

$Q(S_0, a_\downarrow) = 1 + 0.99 \times 2.9701$
$= 3.9403$

$Q(S_1, a_\leftarrow) = 1 + 0.99 \times 3.9403$
$= 2.9008$

STEP 3:

	↑	↓	←	→
S_0		3.9403		3.9403
S_1		2.9701	2.9008	2.9701
S_2	1.9403	1.99	1.9403	1.99
S_3	0.9701	1	0.9701	
S_4	0	0	0	0

state S_3

$$Q(S_3, a_\rightarrow) = 0 + 0.99 \times 1$$

$$= 0.99$$

state S_1

$$Q(S_1, a_\rightarrow) = 0 + 0.99 \times 0.2701$$

$$= 2.9404$$

state S_0

$$Q(S_0, a_\rightarrow) = 0 + 0.99 \times 3.9403$$

$$= 3.9008$$

$$Q(S_0, a_\leftarrow) = 0 + 0.99 \times 3.9403$$

$$= 3.9008$$

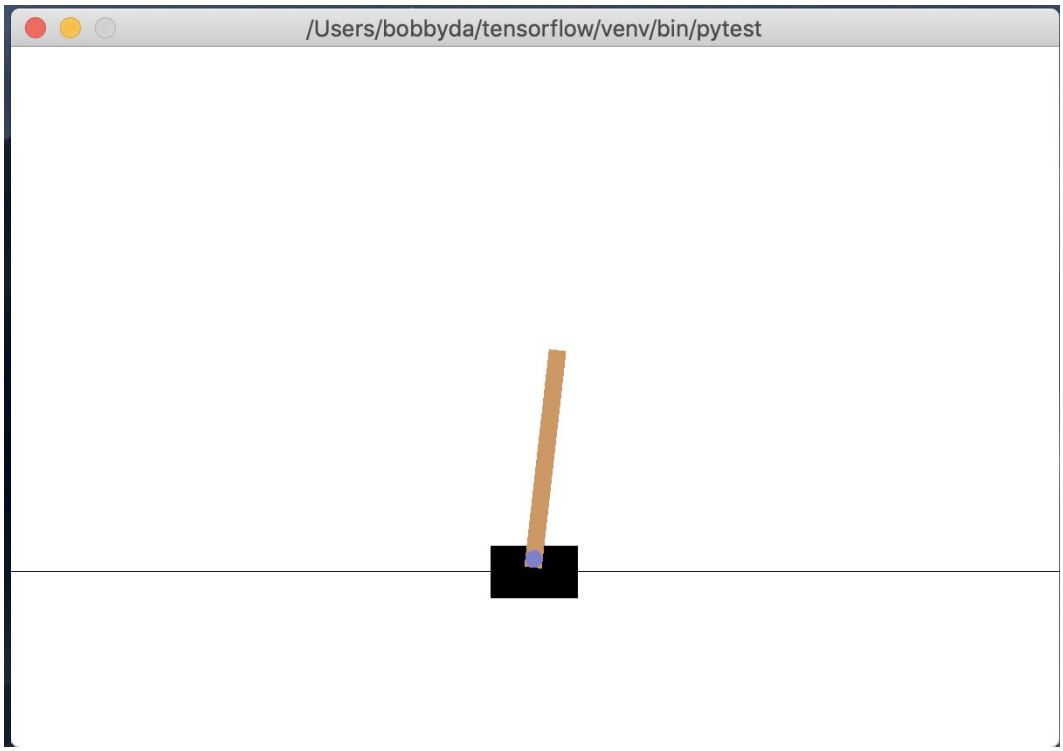
←	→	↓	↑	

FINAL RESULT:

	↑	↓	←	→
S_0	3.9008	3.9403	3.9008	3.9403
S_1	2.9404	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

7 Bonus

7.1 Cartpole Environment from Gym



The Cart-Pole environment consists of a cart that moves along the x axis and a pole that is anchored on the cart. At each step, you can observe its position in the x axis, its velocity, angle and angular velocity. These are the observable states of this environment. At any state, the cart only has two possible actions. It either moves to irs right or left to balance the pole. In this environment the agent gets a reward only when the pole stands on the cart within a certain degree or the agent is penalized if the pole falls beyond a certain angle. The problem is considered solved when it stays upright for 200 times consecutively.

7.2 Implementation

We use the stable baselines python library which is based on the gym library. We use the DQN to train our model. We use the mlp policy and log the results using the tensorboard. We run it for 100000 time steps. The reward graph is shown below.

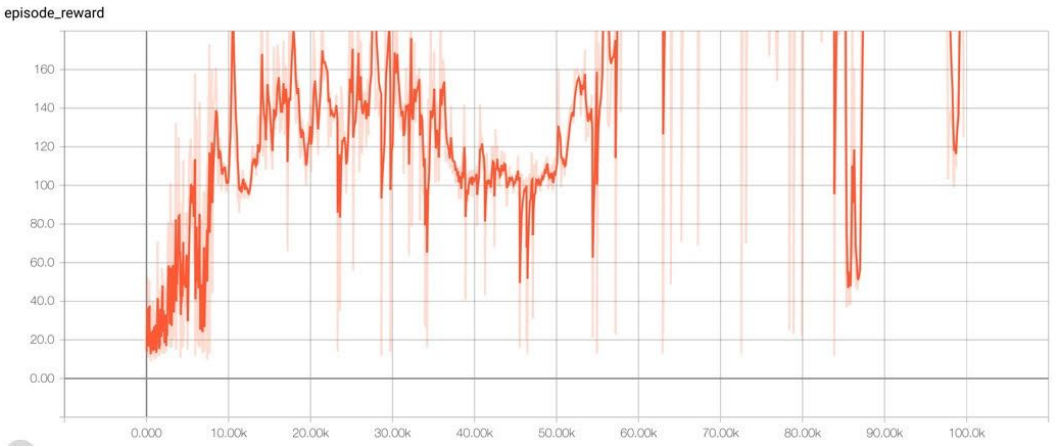


Figure 12: Reward Graph using tensorboard for Cartpole using gym for 100000 steps

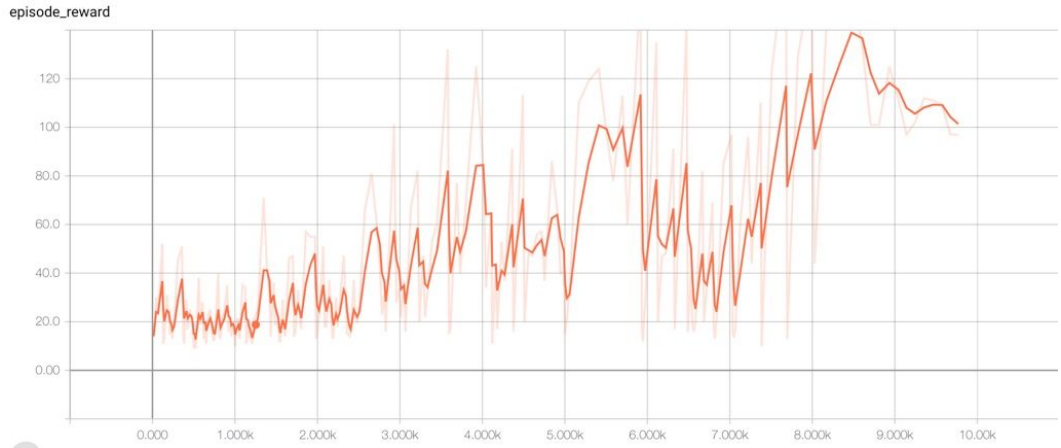


Figure 13: Reward Graph using tensorboard for Cartpole using gym for 50000 steps

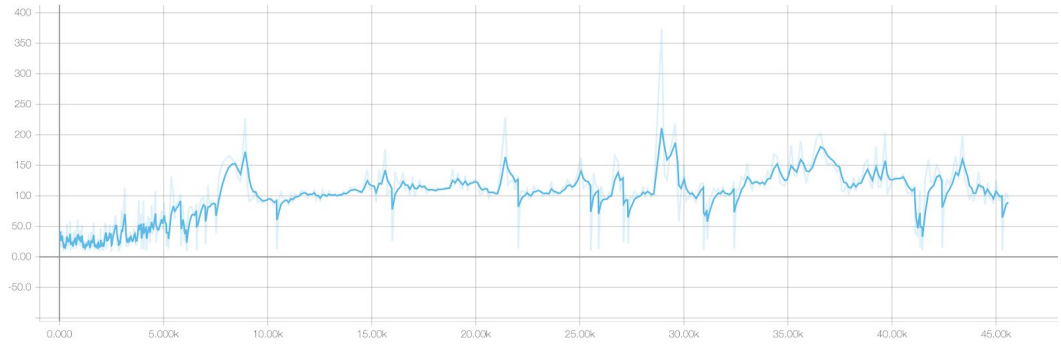


Figure 14: Optimal Reward Graph

References

[1]Lecture Slides

[2]https://medium.freecodecamp.org/a-brief-introduction-to-reinforcement-learning-7799af5840db

[3]https://towardsdatascience.com/stable-baselines-a-fork-of-openai-baselines-reinforcement-learning-made-easy-df87c4b2fc82

[4]http://home.deib.polimi.it/restelli/MyWebSite/pdf/rl5.pdf

[5]http://www.incompleteideas.net/book/ebook/node17.html

[6]https://towardsdatascience.com/stable-baselines-a-fork-of-openai-baselines-reinforcement-learning-made-easy-df87c4b2fc82