CSE474/574: Introduction to Machine Learning (Fall 2018)

Project 4: Tom and Jerry in Reinforcement learning

Kishan Dhamotharan
Person# 50287619

Introduction

Reinforcement learning is an approach where an agent interacts with an environment by taking actions in which it tries to maximize an accumulated reward. We have used this approach for teach the agent(Tom) to navigate and reach Jerry. To solve this we have use deep reinforcement learning algorithm with DQN (Deep Q-Network).

Description on the parts which were implemented:

(a) Neural Networks

Role in Reinforcement learning
Neural networks here acts as the brain of the system. Neural networks plays a very crucial role in reinforcement learning, it helps us deduce the mapping between state-action pairs to rewards. Like all neural networks, they use coefficients to approximate the function relating inputs to outputs, and their learning consists to finding the right weights, by iteratively adjusting those weights along gradients that promise less error.

Expected Setup
We were asked to implement the neural network with the setup:
LINEAR → RELU → LINEAR → RELU → LINEAR
Both the hidden layers have 128 hidden nodes, First hidden layer equals to the size of your observation space and the output size to be equal to action size.

Things which can be enhanced
A number of changes can be bought in the neural network:
Number of hidden layers can be adjusted, No. of hidden nodes per layer can be adjusted doing these two might make the model learn better but the computation time might increase as well. This might some time also lead to over fitting. Thereby we will have adjust add dropouts after the dense layers, to overcome overfitting.
Other changes that can be brought in are on the learning rate of the neural network.

Current system is a deep neural network, as here we deal with the images/frames of the game, we can implement a **convolution neural network**. This will most likely enhance the performance.

(b) Exponential-decay formula for epsilon

Role in Reinforcement learning
Following is the formula which was implemented for calculation of epsilon at each stage.

$$\varepsilon = \varepsilon_{min} + (\varepsilon_{max} - \varepsilon_{min})e^{-\lambda|S|}$$

This plays a very important role between choosing whether to explore or exploit. We start of with a high epsilon as that the beginning our knowledge is empty, exploiting will not be a good choice to get good results, So we explore more. Hence we have higher epsilon at the start and as we progress we decay(reduce) our epsilon, as we can make meaning full exploiting after some episodes.

Expected Setup
We are asked to setup the epsilon as given in the above formula.

Things which can be enhanced
There can be a lot of queking done around the lambda which is a hyperparameter for the exponential-decay. Lambda controls the rate at which the **epsilon** decays. If this is tuned properly we can expect better results. I have performed and experimented with tuning this hyper parameter, written my observations around it below.

(c) Experience Replay

Role in Reinforcement learning
Important reason for using experience replay in that we record the experiences when we perform any steps, which will be useful in future decision making. Our goal is to maximize the value function Q (expected future reward given a state and action). Q table helps us to find the best action for each state. Before we explore the environment: Q table gives the some arbitrary fixed value → but as we explore the environment → Q gives us a better and better approximation. I had to implement the below:

Expected Setup

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

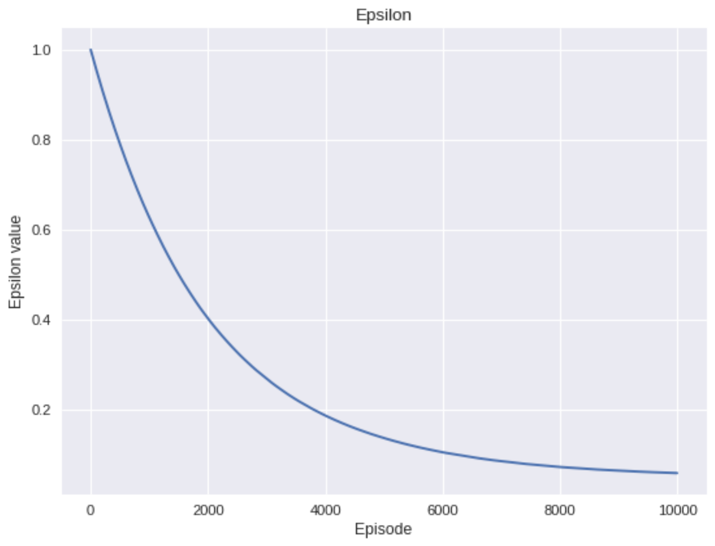
Things which can be enhanced
Our implementation of Q function is a simplified version of the **Bellman equation.** We can enhance our setup by implementing the full version of the bellman equation. Which is as below

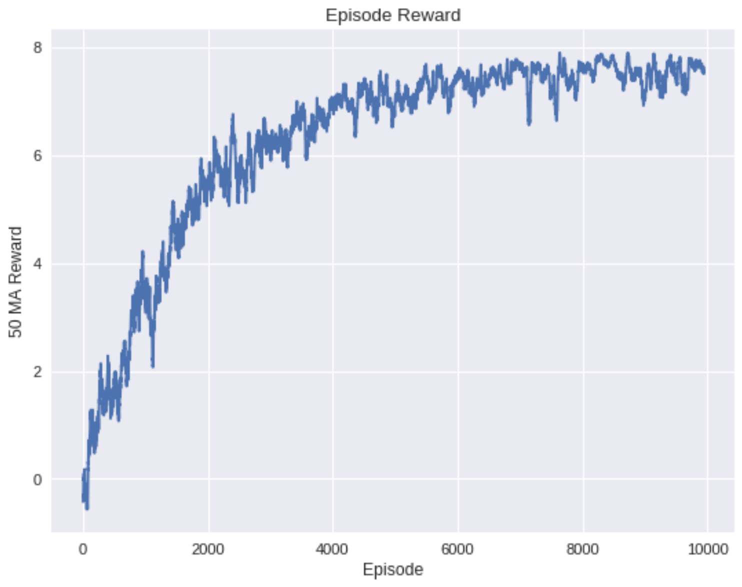
```
New Q value = Current Q value +
              lr * [Reward + discount_rate * (highest Q value between possible
actions from the new state s' ) - Current Q value ]
```

Also we can tune the hyperparameter gamma(discount factor) is a measure of how far ahead in time the algorithm looks. To prioritise rewards in the distant future, keep the value closer to one. A discount factor closer to zero on the other hand indicates that only rewards in the immediate future are being considered.

We can see from the below graph that the episode reward saturates around 6K episodes at near 8 reward points. Therefore we can tell that the agent(Tom) was able to find Jerry around the **6K episode** at a **reward point of 8**.

Graphs with default configuration (Working Algorithm):



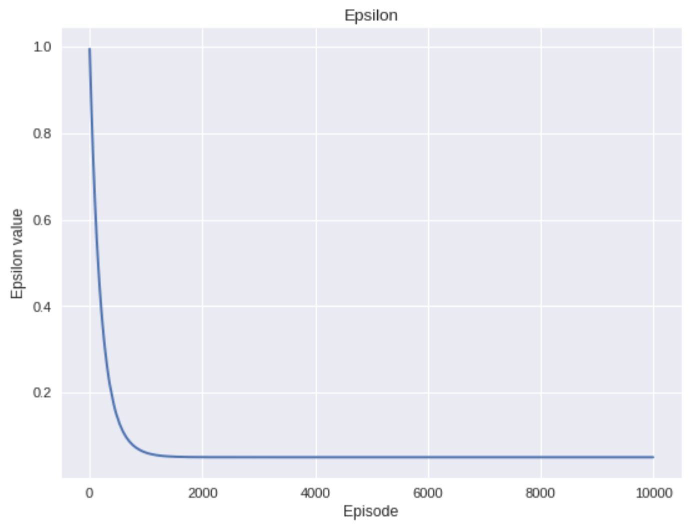


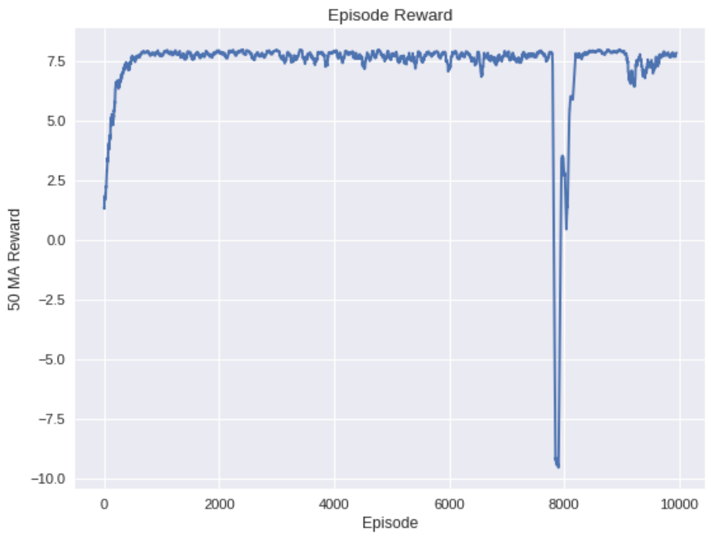
Hyperparameter Tuning:

LAMBDA

λ = 0.0005

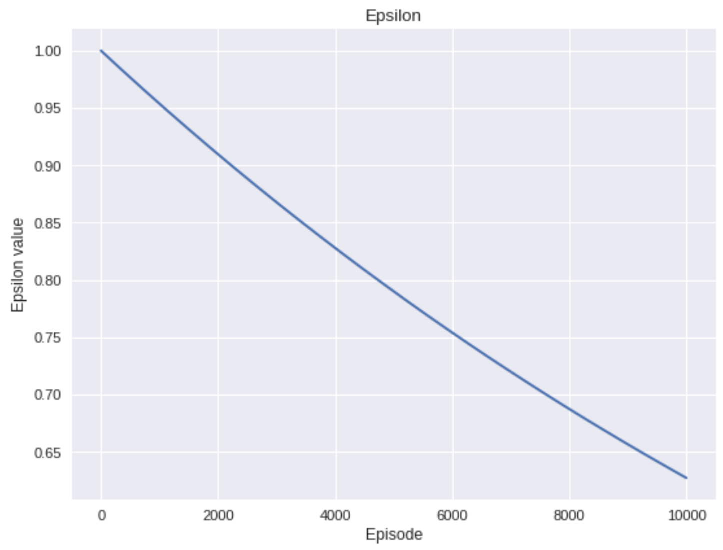
When we increase the lambda the epsilon reaches the min epsilon faster. Which means that it stops to explore faster than with a higher value of lambda. One other interesting observation is that here we reach the saturation point of rewards at around 1000 episode, which is faster than the default config, but as lambda decreases the system stops to explore, our reward is slightly lesser than the default config.

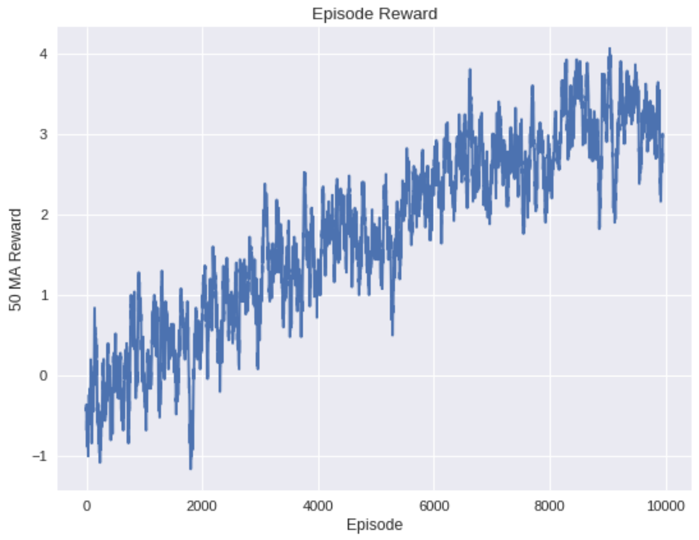




λ = 0.000005

When we decrease the lambda furthermore, it takes much longer to reduce the epsilon, hence more time to explore, hence the graph has much more noise. On top which it will take lots more episode to improve on the rewards. As we can see that it has just reached a max reward of just below 4.

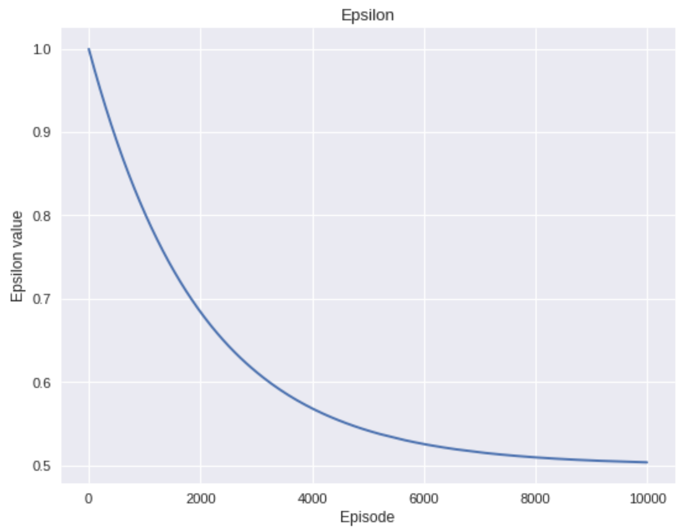


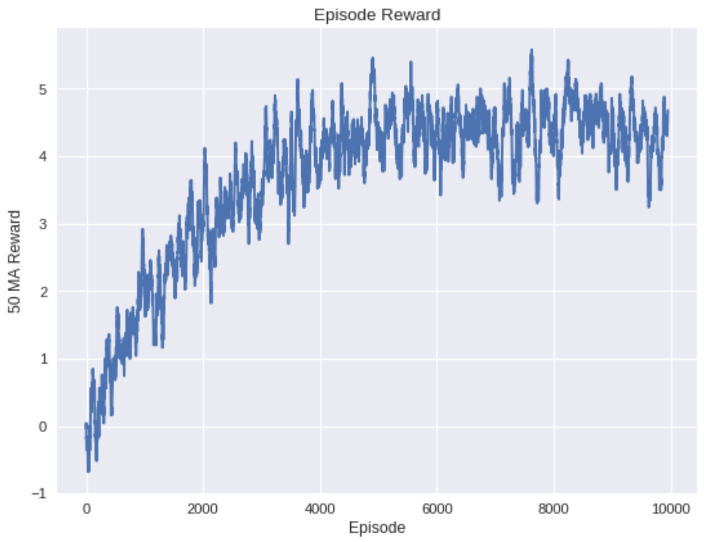


EPSILON

Increasing the min ε = 0.5

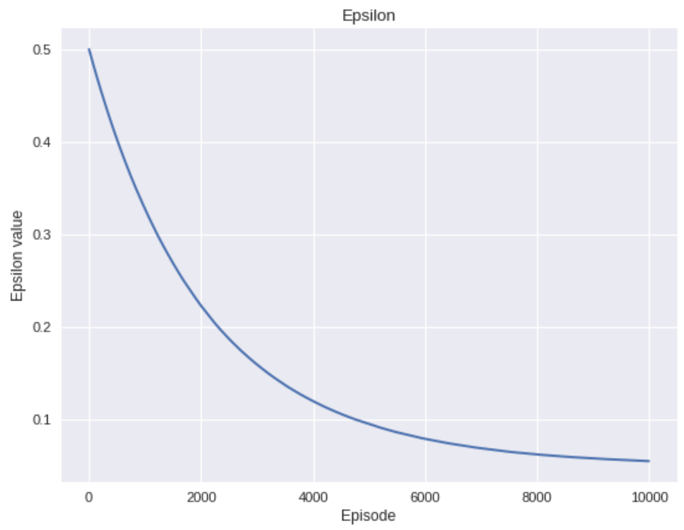
Has a significant impact on the reward as it saturates around 4.5-5, which quite less compared to the default config. Reason behind this is that, we do not let the system to exploit more , as we have increased the min epsilon. From this we can understand that as our q table gets filled up, we need to start to exploit more than explore.

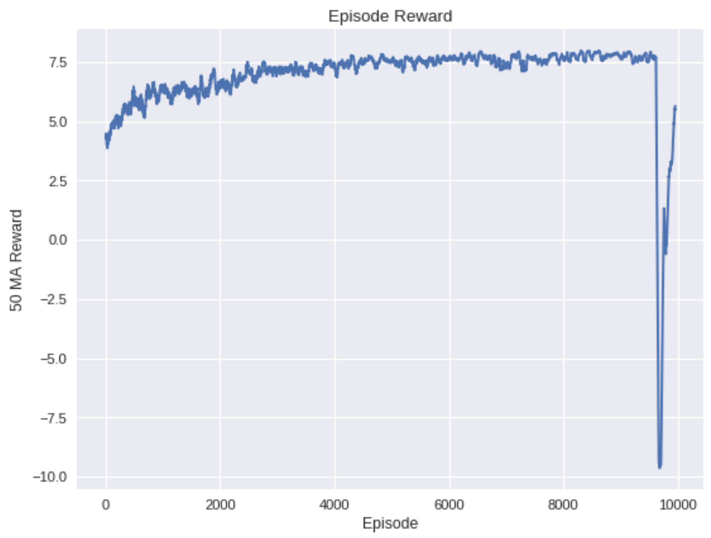




Decreasing the max ε = 0.5

Now that we have reduced the max epsilon, now the system gets much lesser time to explore. System will be highly greedy, this need not always produce good result. Maybe work good for simple situations, but not for complex ones.

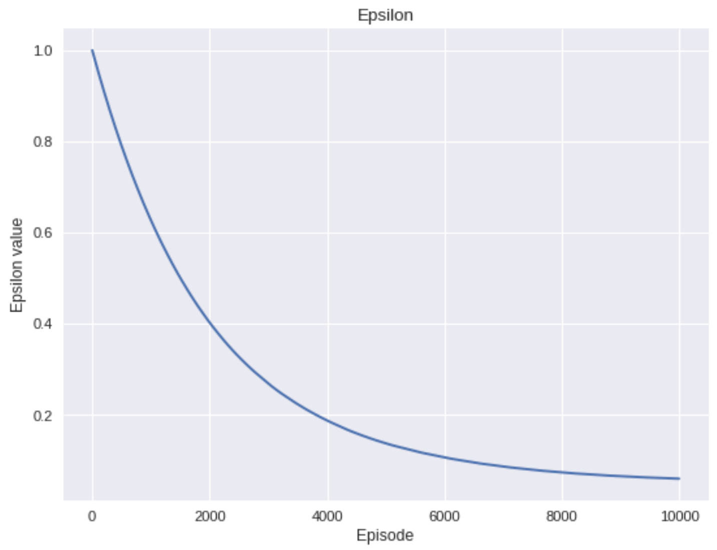


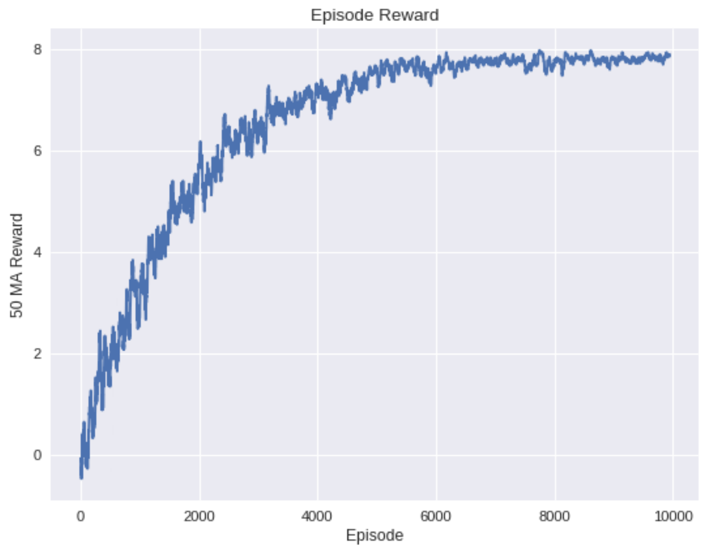


GAMMA

ɣ = 0.69

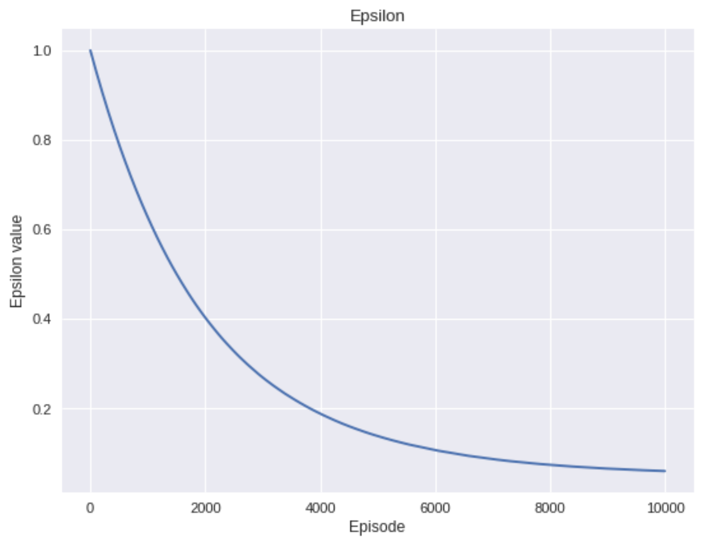
As now we have reduced the gamma from 0.99 to 0.69, we can see that it has not impacted the reward much, but we can totally observe that the graph has become much more smooth, less amount of noise in the system. As with reduced gamma, the system tries to exploit more. Here as the gamma was 0.99 there

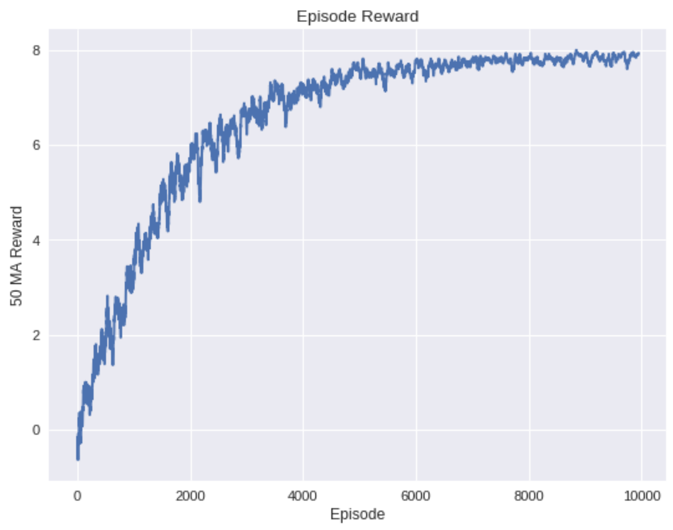




ɣ = 0.39

Has a very similar graph as that of the gamma = 0.69. This look even more smoother, and rewards is almost the same as the above. Hence does not have too much impact on the system, as we have greedy approach where we take the max of state+1. Will have much more impact of we move out of the gamma

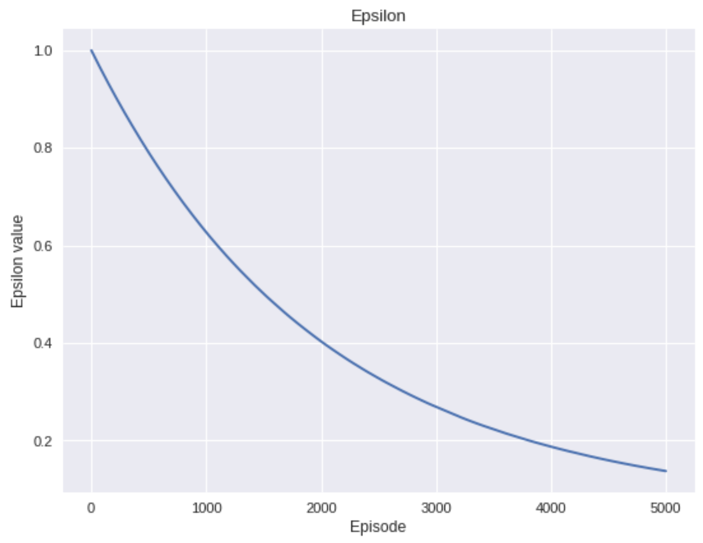


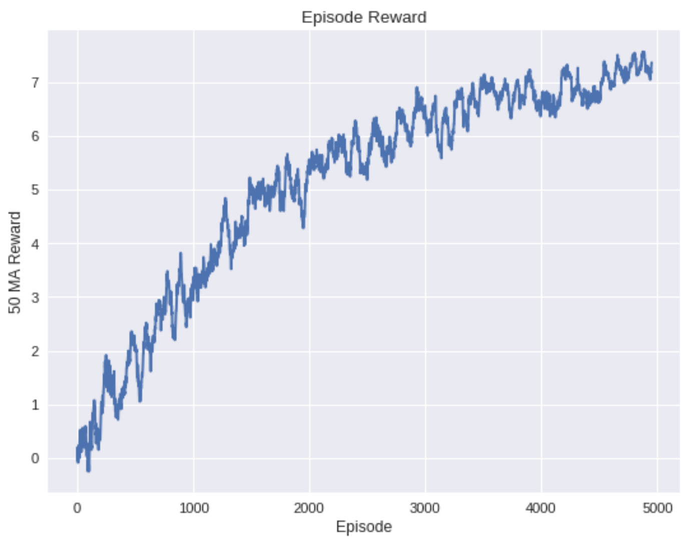


Number of episodes

5000

We can observe the normal





2 Writing part

1)

If the agent always chooses the action that maximizes the Q-value it only exploits from the known knowledge around it, it will does not look at the bigger picture. Always performing a action which will maximize the Q-value is highly a greedy approach, which will not always work. Hence we need to bring in some randomness/explore different paths to get the best results. This can be understood by an example,

A	-1	-1
1	-1	10
1	1	B

Suppose our starting point is A and our destination is B and the score is the sum of all the values along the path. If our agent always looks to maximize the Q values around it it will never be able to achieve the max score.

When tries to choose to always maximizes the Q-value

A	-1	-1
1	-1	10
1	1	B

Actual maximum path, which can be achieved only by exploring, not such exploiting.

A	-1	-1
1	-1	10
1	1	B

Hence if our agent always performs the action that maximizes the Q-value, we might not always get the ideal result. And most the real world problems will require exploring not just exploiting.

Following are the approaches for forcing the agent to explore are :

- We can make the agent explore more by just letting it make few random actions now and then, which will allow it to explore more, rather than just exploit.
- As we know that the major draw of just pure exploiting is that, they never pick the paths which do not give them immediate results. Hence if possible we need to start with giving good values to all action, which by itself induce exploration. In case, it picks a bad path, it will eventually learn that it is bad, rather than leaving a path which might not have local benefit, but at the end might give better results.
- **Using Gamma** : Also we can tune the hyperparameter gamma(discount factor) is a measure of how far ahead in time the algorithm looks. To prioritise rewards in the distant future, keep the value closer to one. A discount factor closer to zero on the other hand indicates that only rewards in the immediate future are being considered.
- **Using epsilon :** As we know that epsilon is the exploration rate, having high values of epsilon forces the agent to explore. Our general strategy is that we start of with a high epsilon as that the beginning our knowledge (Q table) is empty, exploiting will not be a good choice to get good results, hence we have higher epsilon at the start and as we progress we decay(reduce) our epsilon, as we can make meaning full exploiting after some iterations. The strategy which we can apply which will force the agent to always explore is having a high value for min epsilon, which will ensure that the epsilon never goes below a certain limit.

2)

Q Table Calculation:

Known:

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

ɣ (gamma) = 0.99

S_t = State

A_t = Action

r_t = Rewards

Rewards Table

Towards goal state	**+1**
Same state	**0**
Away from goal state	**-1**

- We start with the final state, as the final state being the terminal state (where we have reached the destination) we do not move any further, hence initialize all the actions to 0.

First iteration:
- We fill the Q table in a bottom up manner starting from state 4 to state 0.
- In the first iteration we skip the (state, action) of which we do not know the max (state + 1, a).

Calculations:

Q(s3, Down) = 1
Q(s3, Right) = 0 + (0.99) (1) = 0.99
Q(s2, Right) = 1 + (0.99)(1) = 1.99
Q(s1, Bottom) = 1 + (0.99)(1.99) = 2.9701
Q(s1, Up) = 0 + (0.99)(2.97) = 2.940
Q(s0, Right) = 1 + (0.99)(2.9701) = 3.9403
Q(s0, Up) = 0 + (0.99)(3.9403) = 3.9
Q(s0, Left) = 0 + (0.99)(3.9403) = 3.9

State	**Up**	**Down**	**Left**	**Right**
s0	**3.9**	**?**	**3.9**	**3.9403**
s1	**2.940**	**2.9701**	**?**	**?**
s2	**?**	**?**	**?**	**1.99**

s3	**?**	**1**	**?**	**0.99**
s4	**0**	**0**	**0**	**0**

Second iteration:

- Now we fill out all the cases of which did not have their maximums in the last iteration.
- Also we check for elements which are symmetric and fill them as well

Calculations:
Q(s3, Left) = -1 + (0.99) (1.99) = 0.97
Q(s3,Up) = Q(s3, Left), By symmetry
Q(s2, Up) = -1 + (0.99)(2.9701) = 1.9403
Q(s2, Down) = Q(s2, Right), By symmetry
Q(s2, Left) = Q(s2, Up), By symmetry
Q(s1, Left) = -1 + (0.99)(3.9403) = 2.9008
Q(s1, Right) = Q(s1, Down), By symmetry
Q(s0, Down) = Q(s0, Right), By symmetry

State	**Up**	**Down**	**Left**	**Right**
s0	**3.90**	**3.9403**	**3.90**	**3.9403**
s1	**2.940**	**2.9701**	**2.9008**	**2.9701**
s2	**1.9403**	**1.99**	**1.9403**	**1.99**
s3	**0.97**	**1**	**0.97**	**0.99**
s4	**0**	**0**	**0**	**0**