RL: Assignment 2: Final

Name: K Naga Kiran Reddy UB ID: 50432242

"I certify that the code and data in this assignment were generated independently, using only the tools and resources defined in the course and that I did not receive any external help, coaching or contributions during the production of this work."

Part I:

Cart pole v1:

##Cart pole v1 main details:

Goal: To keep the pole upright.

Action:

0: Left

1: Right

Observation space:

(Observation, Min, Max):

0. (Cart Position, -4.8, 4.8)

1. (Cart Velocity, -Inf, Inf)

2. (Pole Angle, ~ -0.418 rad (-24°), ~ 0.418 rad (24°))

3. (Pole Angular Velocity, -Inf, Inf)

Rewards:

Reward of +1 for every step taken including the termination step, is allotted. The threshold for rewards is 475 for v1.

Initial State:

All observations are assigned a uniformly random value in (-0.05, 0.05)

Episode Termination Conditions:

1. Pole Angle is greater than ±12°

2. Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)

3. Episode length is greater than 500 (200 for v0)

Mountain Car v0:

Goal: The goal is to reach the flag placed on top of the right hill as quickly as possible.

Actions:

0. Accelerate to the left

1. Don't accelerate

2. Accelerate to the right

Observation space:

0. Position of the car along the x-axis

1. Velocity of the car

Reward:

0: if it takes step towards the goal.

-1: if it takes step that isn't towards the goal

Initial State:

The position of the car is assigned a uniform random value in [-0.6, -0.4]. The starting velocity of the car is always 0.

Episode Termination Conditions:

1. The position of the car is greater than or equal to 0.5 (the goal position on top of the right hill)

2. The length of the episode is 200.

Part II:

1.

a) Using experience replay in DQN and how its size can influence the results?

It breaks the correlation in data and using larger chunks of experience values randomly gives efficient results.

b) Introducing the target network

By using a separate network, which is used to freeze the Q-value target, so we don't have moving target, we can avoid oscillations and breaks correlations between Q-network and target.

Using one deep network (i.e., target network) to retrieve Q values while the second one (i.e., policy network) includes all updates in the training and w- and w are synchronized periodically.

c) Representing the Q function as qˆ(s, w)

Using Q function in above format helps to determine Q values using function approximation, a neural network can be used to learn the mapping (i.e., weight parameters) from states to the Q-value for real world scenarios, which usually involves a larger set of state and action space.

2. Grid Environment details:

Actions: {down, up, right, left}

States: {s1, s2, s3, s4, s5....s16} -> 4x4 grid

Rewards: {-2, -1, 0, 1, 2, 150}

Starting position: [0,0]

Target position: [3,3] #If agent reaches target position: Reward = 150

Danger1 position: [1,1] #If agent reaches danger1 position: Reward = -1

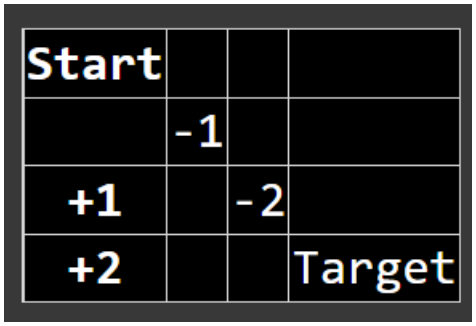
Danger2 position: [2,2] #If agent reaches danger2 position: Reward = -2

Gold1 position: [2,0] #If agent reaches gold1 position: Reward = +1

Gold2 position: [3,0] #If agent reaches gold2 position: Reward = +2

A reward of **zero** in all other cases. #Reward = 0

Grid-world Environment:



3) Model and Results:

Model:

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_6 (Dense)             (None, 64)                1088

 dense_7 (Dense)             (None, 32)                2080

 dense_8 (Dense)             (None, 4)                 132

=================================================================
Total params: 3,300
Trainable params: 3,300
Non-trainable params: 0
_________________________________________________________________
```

Model:

Number of Episodes: 2500

Batch Size: 128

Replay memory size: 2000

C value: 5

Discount rate: 0.99

I have used 3 NN layers:

model.add(Dense(64, activation='relu', input_dim = 16)) #first hidden layer with 64 nodes taking input from 16 - one hot vector of a state

model.add(Dense(32, activation='relu')) #first hidden layer with 32 nodes taking input from first dense layer

model.add(Dense(4, activation='linear')) #output layers has q value for 4 actions

I have used a grid world of 4x4, so I have used 16 one hot vector as an input to first dense layer.

I have penalized my agent whenever it stays in same place (i.e., picking small rewards so many times) or whenever it picks a small reward.

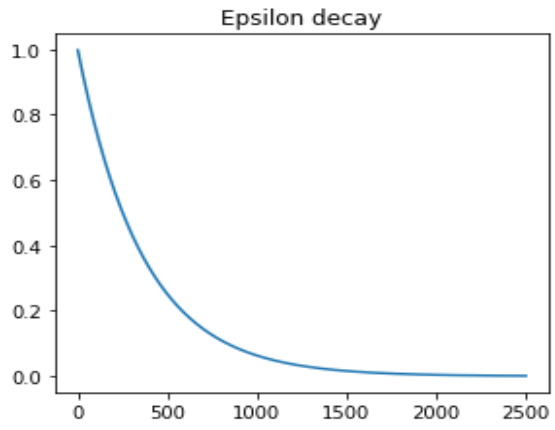
```
if list(state[0]).index(1) == list(next_state[0]).index(1) or reward <=
 1:
        reward = reward - 2
```

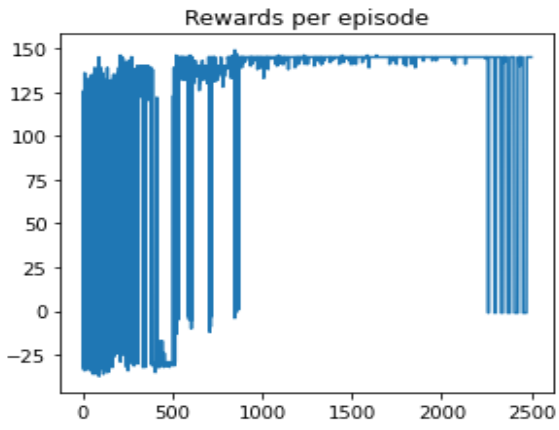
Results:

```
Episode: 1, total_reward: -33
Episode: 125, total_reward: -4
Episode: 250, total_reward: 134
Episode: 375, total_reward: 130
Episode: 500, total_reward: -31
Episode: 625, total_reward: 140
Episode: 750, total_reward: 132
Episode: 875, total_reward: 145
Episode: 1000, total_reward: 145
Episode: 1125, total_reward: 143
Episode: 1250, total_reward: 145
Episode: 1375, total_reward: 143
Episode: 1500, total_reward: 145
Episode: 1625, total_reward: 145
Episode: 1750, total_reward: 145
Episode: 1875, total_reward: 145
Episode: 2000, total_reward: 145
Episode: 2125, total_reward: 145
Episode: 2250, total_reward: 145
Episode: 2375, total_reward: -1
Episode: 2500, total_reward: 145
Text(0.5, 1.0, 'Rewards per episode')
```

The above total_reward is the reward obtained after penalizing the agent. While evaluating I have printed the actual rewards.

Plots showing episodes vs epsilon decay, episodes vs rewards:

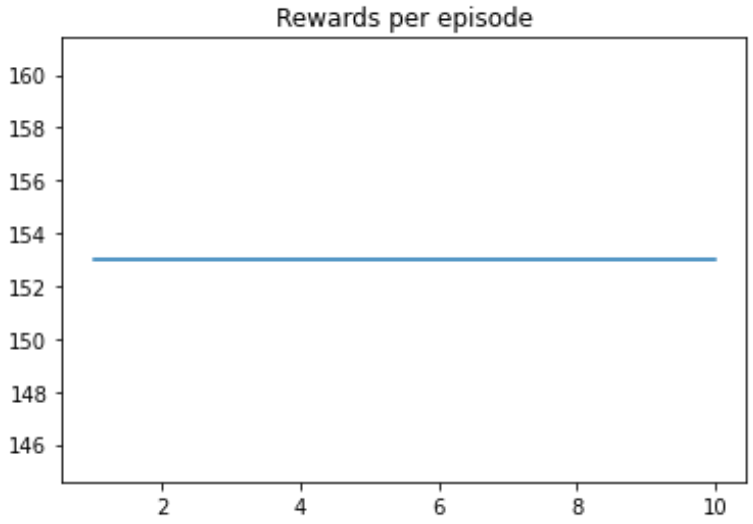




From the above results it is evident that the epsilon decays smoothly whereas reward oscillates at first and reaches a stable state afterwards and the oscillation at the end shows agent choosing a random action and picking a different total reward other than the optimal reward. But in final episodes the agent is picking optimal path to reach the target.

4) Evaluation results: I have run the agent on 10 more episodes choosing greedy actions from learnt policy, below figure shows that the agent has reached an optimal path with a constant reward across all 10 episodes.

```
Episode: 1, Total Reward: 153
Episode: 2, Total Reward: 153
Episode: 3, Total Reward: 153
Episode: 4, Total Reward: 153
Episode: 5, Total Reward: 153
Episode: 6, Total Reward: 153
Episode: 7, Total Reward: 153
Episode: 8, Total Reward: 153
Episode: 9, Total Reward: 153
Episode: 10, Total Reward: 153
Optimal Path:
1 -> 5 -> 9 -> 13 -> 14 -> 15 -> 16 -> Text(0.5, 1.0, 'Rewards per episode')
```



Part III:

1) For the improved version of Deep Q-network (i.e., DQN), I have implemented Double DQN, which reduces the overestimation problem. For this I have changed my target value calculation and syncing weights strategy.

The difference is in using target network weights to calculate the y_target, to which the policy network output has to be adjusted.

```
y_target = reward + self.discount_rate*target_qvals[index][action_val]
```

There is also a change in how we sync the weights. In vanilla version DQN, we just sync weights directly every 5 episodes, but in Double DQN, the sync happens as shown below.

```
for target_param, param in zip(self.target_model.trainable_variables, s
elf.policy_model.trainable_variables):
    target_param.assign(self.t * param + (1 - self.t) * target_param)
```

2. Double DQN on Grid World:

Model:

```
Model: "sequential_18"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_54 (Dense)            (None, 64)                1088

 dense_55 (Dense)            (None, 32)                2080

 dense_56 (Dense)            (None, 4)                 132

=================================================================
Total params: 3,300
Trainable params: 3,300
Non-trainable params: 0
_________________________________________________________________
```

Number of Episodes: 500

Batch Size: 128

Replay memory size: 2000

C value: 5

Discount rate: 0.99

I have used a grid world of 4x4 so 16 one hot vector as an input to first dense layer and I have penalized my agent whenever it stays in same place (i.e., picking small rewards so many times) and gave reward of 5 whenever it reaches the surroundings of the target position.

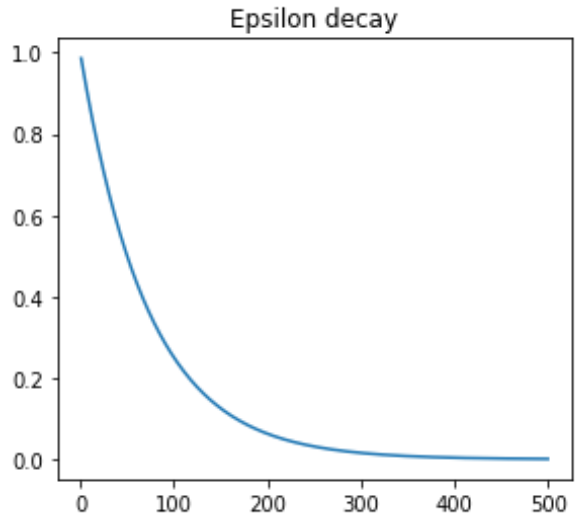
```
if list(state[0]).index(1) == list(next_state[0]).index(1): #same state
        reward = reward - 2

if np.linalg.norm(target - np.array(next_state_pos)) <= 1:
        reward = reward + 5
```

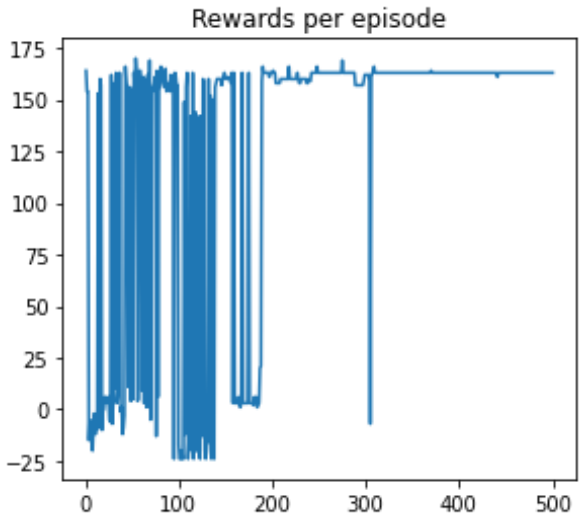
3) Results:

```
Episode: 1, total_reward: 164
Episode: 25, total_reward: 6
Episode: 50, total_reward: 155
Episode: 75, total_reward: 161
Episode: 100, total_reward: -17
Episode: 125, total_reward: -13
Episode: 150, total_reward: 160
Episode: 175, total_reward: 163
Episode: 200, total_reward: 163
Episode: 225, total_reward: 160
Episode: 250, total_reward: 163
Episode: 275, total_reward: 169
Episode: 300, total_reward: 162
Episode: 325, total_reward: 163
Episode: 350, total_reward: 163
Episode: 375, total_reward: 163
Episode: 400, total_reward: 163
Episode: 425, total_reward: 163
Episode: 450, total_reward: 163
Episode: 475, total_reward: 163
Episode: 500, total_reward: 163
```

The above reward shows the reward obtained after penalizing and rewarding the agent. While evaluating I have printed the actual rewards.

Plots showing episodes vs epsilon decay, episodes vs rewards:

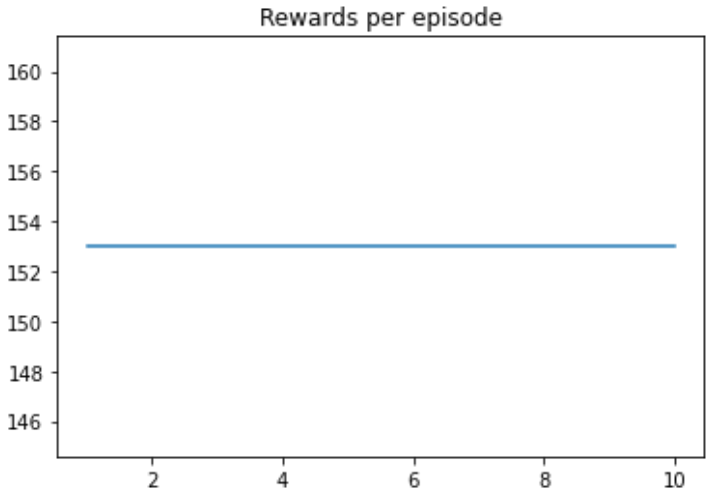




From the above results it is evident that the epsilon decays smoothly whereas reward oscillates at first and reaches a stable state afterwards, the agent is picking maximum reward while following optimal path to reach the target.

4) Evaluation results: I have run the agent on 10 more episodes choosing greedy actions from learnt policy, below figure shows that the agent has reached an optimal path with a constant reward across all 10 episodes.

```
Episode: 1, Total Reward: 153
Episode: 2, Total Reward: 153
Episode: 3, Total Reward: 153
Episode: 4, Total Reward: 153
Episode: 5, Total Reward: 153
Episode: 6, Total Reward: 153
Episode: 7, Total Reward: 153
Episode: 8, Total Reward: 153
Episode: 9, Total Reward: 153
Episode: 10, Total Reward: 153
Optimal Path:
1 -> 5 -> 9 -> 13 -> 14 -> 15 -> 16 -> Text(0.5, 1.0, 'Rewards per episode')
```



5) Separate section for each model:

DQN on CartPole-v1:

Model:

```
Model: "sequential_14"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_42 (Dense)            (None, 32)                160

 dense_43 (Dense)            (None, 24)                792

 dense_44 (Dense)            (None, 2)                 50

=================================================================
Total params: 1,002
Trainable params: 1,002
Non-trainable params: 0
```

Number of Episodes: 100

Batch Size: 128

Replay memory size: 5000

C value: 3

Discount rate: 0.99

I have used a different reward structure to make the algorithm faster, as +1 and -1 will take a lot of time to converge

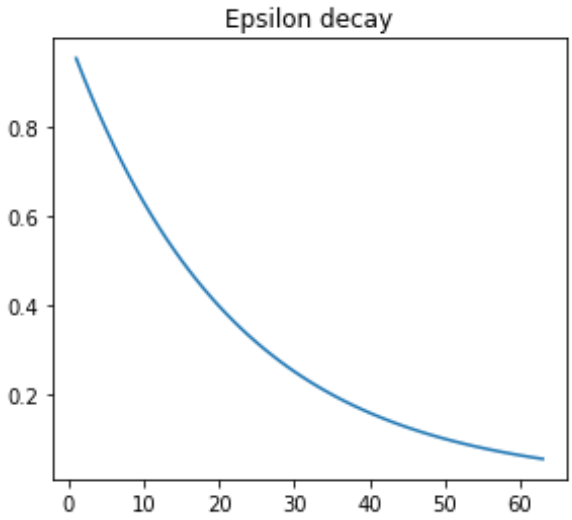
```
reward = -100*(abs(next_state[0,2]) - abs(state[0,2]))
```

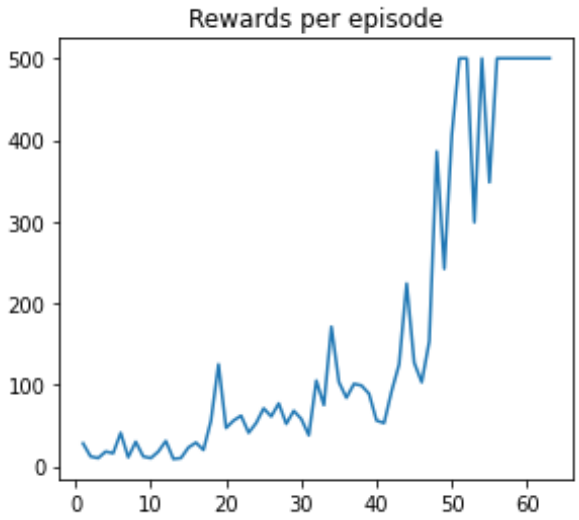
The above reward is based on pole angle, where it gets a positive reward if it maintains the pole closer to equilibrium state.

I have stopped my algorithm when it reaches an average of 470 in last 10 episodes

Results:

```
Episode: 1, Total reward: 28, Average reward: 0
Episode: 5, Total reward: 16, Average reward: 0
Episode: 10, Total reward: 10, Average reward: 18.8
Episode: 15, Total reward: 23, Average reward: 19.5
Episode: 20, Total reward: 47, Average reward: 36.8
Episode: 25, Total reward: 71, Average reward: 56.0
Episode: 30, Total reward: 58, Average reward: 59.9
Episode: 35, Total reward: 103, Average reward: 80.8
Episode: 40, Total reward: 56, Average reward: 92.1
Episode: 45, Total reward: 127, Average reward: 105.0
Episode: 50, Total reward: 406, Average reward: 191.1
Episode: 55, Total reward: 348, Average reward: 343.7
Episode: 60, Total reward: 500, Average reward: 464.7
Average above 470 Reached!!
Text(0.5, 1.0, 'Rewards per episode')
```

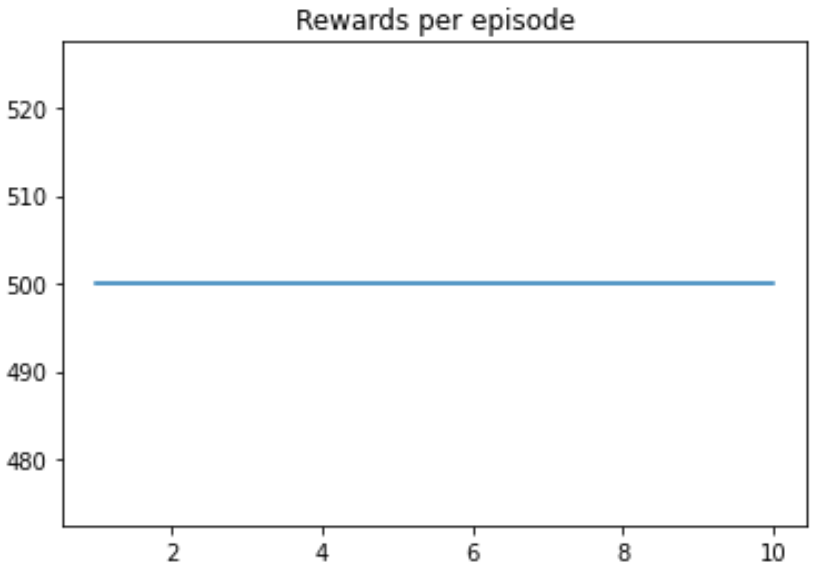




The above plots show that the cartpole is learning, is able to balance long time.

Evaluation results:

```
Episode: 1, Total Reward: 500
Episode: 2, Total Reward: 500
Episode: 3, Total Reward: 500
Episode: 4, Total Reward: 500
Episode: 5, Total Reward: 500
Episode: 6, Total Reward: 500
Episode: 7, Total Reward: 500
Episode: 8, Total Reward: 500
Episode: 9, Total Reward: 500
Episode: 10, Total Reward: 500
Text(0.5, 1.0, 'Rewards per episode')
```



The above plot shows how cartpole is balancing for all 500 timesteps and getting a reward of 500 while taking greedy actions in all 10 episodes

DQN on Mountain Car-v0:

Model:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 32)                96

 dense_1 (Dense)             (None, 24)                792

 dense_2 (Dense)             (None, 3)                 75

=================================================================
Total params: 963
Trainable params: 963
Non-trainable params: 0
```

Number of Episodes: 1000

Batch Size: 128

Replay memory size: 5000

C value: 3

Discount rate: 0.99

I have used a different reward structure to make the algorithm faster, as +1 and -1 if it reaches the top of the hill will take a lot of time to converge.

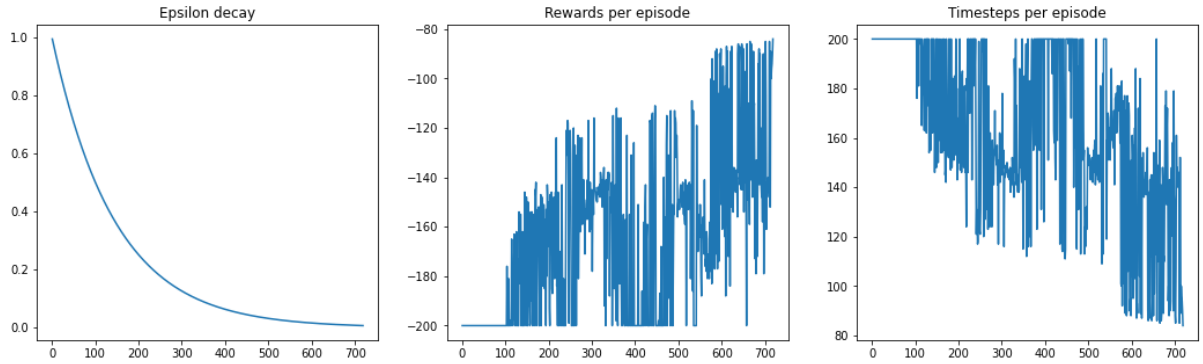
```
reward = 100*((math.sin(3*next_state[0,0]) * 0.0025 + 0.5 * next_state[
0,1] * next_state[0,1]) - (math.sin(3*state[0,0]) * 0.0025 + 0.5 * stat
e[0,1] * state[0,1]))
```

Results:

The mountain car environment is considered solved if it gets a reward of -110 in 100 runs or 100 timesteps. I have used a soft condition, I have considered average over last 10 episodes, so if it collects -110 in last episodes then the mountain car is solved, below results shows this intuition.

```
Episode: 1, Average reward: -200, Average timesteps: 200
Episode: 50, Average reward: -200.0, Average timesteps: 200.0
Episode: 100, Average reward: -200.0, Average timesteps: 200.0
Episode: 150, Average reward: -174.1, Average timesteps: 174.1
Episode: 200, Average reward: -165.7, Average timesteps: 165.7
Episode: 250, Average reward: -132.3, Average timesteps: 132.3
Episode: 300, Average reward: -148.0, Average timesteps: 148.0
Episode: 350, Average reward: -155.1, Average timesteps: 155.1
Episode: 400, Average reward: -185.7, Average timesteps: 185.7
Episode: 450, Average reward: -169.4, Average timesteps: 169.4
Episode: 500, Average reward: -131.5, Average timesteps: 131.5
Episode: 550, Average reward: -159.4, Average timesteps: 159.4
Episode: 600, Average reward: -123.3, Average timesteps: 123.3
Episode: 650, Average reward: -113.2, Average timesteps: 113.2
Episode: 700, Average reward: -131.1, Average timesteps: 131.1
Car reached the top of the hill!!
```

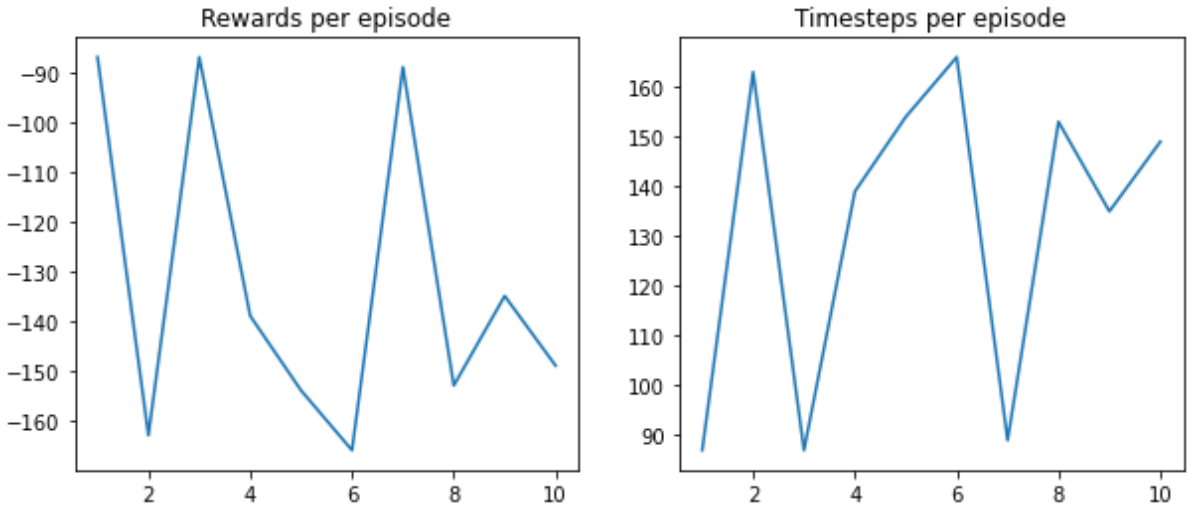
Plots showing epsilon decay and rewards, timesteps over the episodes



The above plots show the car is learning, as it is reaching top of the hill in less timesteps (i.e., decreasing timesteps) while collecting more reward (i.e., increasing reward).

Evaluation results:

```
Episode: 1, Total Reward: -87.0, Timesteps: 87
Episode: 2, Total Reward: -163.0, Timesteps: 163
Episode: 3, Total Reward: -87.0, Timesteps: 87
Episode: 4, Total Reward: -139.0, Timesteps: 139
Episode: 5, Total Reward: -154.0, Timesteps: 154
Episode: 6, Total Reward: -166.0, Timesteps: 166
Episode: 7, Total Reward: -89.0, Timesteps: 89
Episode: 8, Total Reward: -153.0, Timesteps: 153
Episode: 9, Total Reward: -135.0, Timesteps: 135
Episode: 10, Total Reward: -149.0, Timesteps: 149
Text(0.5, 1.0, 'Timesteps per episode')
```



The above results are not stable because I have used a soft condition but still car learnt to reach the top of the hill in less timesteps, which can be inferred from 3 episodes where agent is able to collect < -89 rewards in just < 89 timesteps.

Double DQN on CartPole-v1:

Model:

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_18 (Dense)            (None, 32)                160

 dense_19 (Dense)            (None, 24)                792

 dense_20 (Dense)            (None, 2)                 50

=================================================================
Total params: 1,002
Trainable params: 1,002
Non-trainable params: 0
```

Number of Episodes: 1000

Batch Size: 128

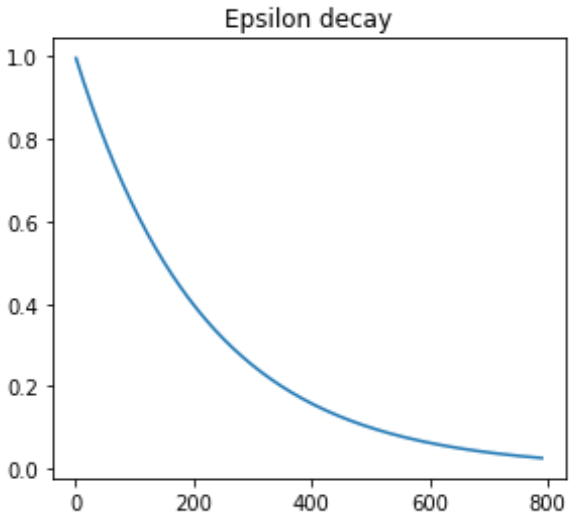
Replay memory size: 5000

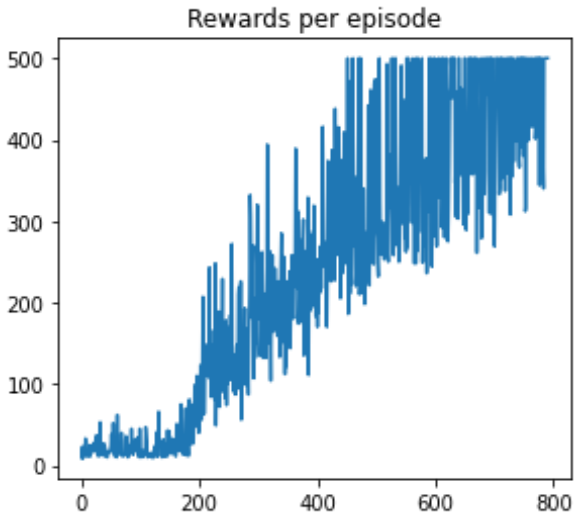
C value: 3

Discount rate: 0.99

Results

```
Episode: 1, Total reward: 9, Average reward: 0
Episode: 50, Total reward: 23, Average reward: 16.0
Episode: 100, Total reward: 18, Average reward: 22.7
Episode: 150, Total reward: 22, Average reward: 23.9
Episode: 200, Total reward: 104, Average reward: 76.8
Episode: 250, Total reward: 155, Average reward: 120.3
Episode: 300, Total reward: 181, Average reward: 197.2
Episode: 350, Total reward: 148, Average reward: 182.0
Episode: 400, Total reward: 170, Average reward: 223.2
Episode: 450, Total reward: 289, Average reward: 296.7
Episode: 500, Total reward: 288, Average reward: 359.4
Episode: 550, Total reward: 262, Average reward: 362.6
Episode: 600, Total reward: 281, Average reward: 401.4
Episode: 650, Total reward: 500, Average reward: 393.8
Episode: 700, Total reward: 269, Average reward: 401.8
Episode: 750, Total reward: 500, Average reward: 448.8
Average above 470 Reached!!
CPU times: user 3h 31min 17s, sys: 11min 2s, total: 3h 42min 20s
Wall time: 3h 40min 27s
```

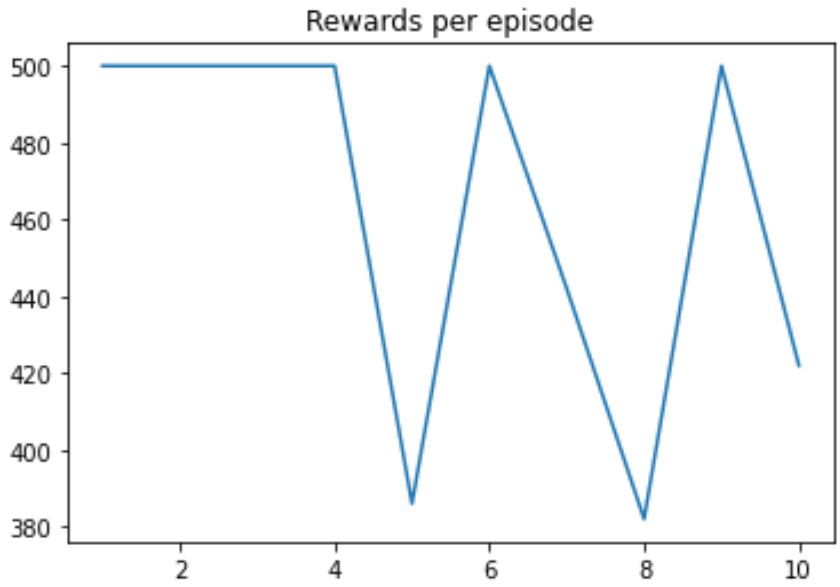




The above clears show the cartpole is learning, as it is collecting more reward by staying balanced for more time in each episode.

Evaluation Results:

```
Episode: 1, Total Reward: 500
Episode: 2, Total Reward: 500
Episode: 3, Total Reward: 500
Episode: 4, Total Reward: 500
Episode: 5, Total Reward: 386
Episode: 6, Total Reward: 500
Episode: 7, Total Reward: 442
Episode: 8, Total Reward: 382
Episode: 9, Total Reward: 500
Episode: 10, Total Reward: 422
Text(0.5, 1.0, 'Rewards per episode')
```



The above plot clearly shows that the cart pole has learnt to take semi optimal actions and is balancing itself for 500 timesteps for 6 out of 10 episodes and in other episodes also it is able to balance for more than 350 timesteps.

Double DQN on MountainCar-v0:

Model:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 32)                96
_________________________________________________________________
dense_1 (Dense)              (None, 24)                792
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 75
=================================================================
Total params: 963
Trainable params: 963
Non-trainable params: 0
```

Number of Episodes: 4000

Batch Size: 128

Replay memory size: 5000

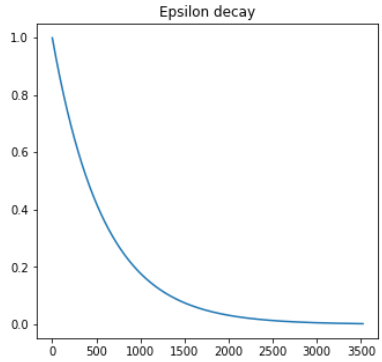
C value: 3

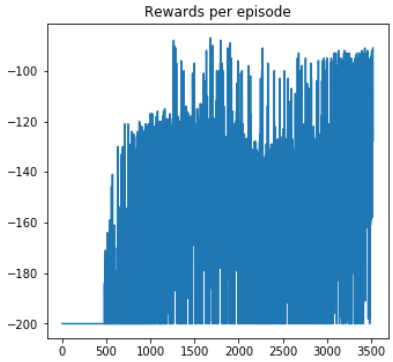
Discount rate: 0.99

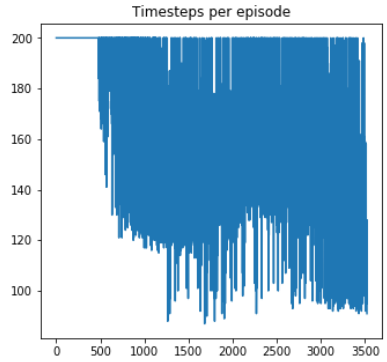
Results:

```
Episode: 1, Average reward: -200, Average timesteps: 200
Episode: 50, Average reward: -200.0, Average timesteps: 200.0
Episode: 100, Average reward: -200.0, Average timesteps: 200.0
Episode: 150, Average reward: -200.0, Average timesteps: 200.0
Episode: 200, Average reward: -200.0, Average timesteps: 200.0
Episode: 250, Average reward: -200.0, Average timesteps: 200.0
Episode: 300, Average reward: -200.0, Average timesteps: 200.0
Episode: 350, Average reward: -200.0, Average timesteps: 200.0
Episode: 400, Average reward: -200.0, Average timesteps: 200.0
Episode: 450, Average reward: -200.0, Average timesteps: 200.0
Episode: 500, Average reward: -197.7, Average timesteps: 197.7
Episode: 550, Average reward: -198.7, Average timesteps: 198.7
Episode: 600, Average reward: -198.3, Average timesteps: 198.3
Episode: 650, Average reward: -195.4, Average timesteps: 195.4
Episode: 700, Average reward: -199.5, Average timesteps: 199.5
Episode: 750, Average reward: -181.2, Average timesteps: 181.2
Episode: 800, Average reward: -190.1, Average timesteps: 190.1
Episode: 850, Average reward: -170.7, Average timesteps: 170.7
Episode: 900, Average reward: -165.7, Average timesteps: 165.7
Episode: 950, Average reward: -150.2, Average timesteps: 150.2
Episode: 1000, Average reward: -151.3, Average timesteps: 151.3
Episode: 1050, Average reward: -143.8, Average timesteps: 143.8
Episode: 1100, Average reward: -168.8, Average timesteps: 168.8
Episode: 1150, Average reward: -132.6, Average timesteps: 132.6
Episode: 1200, Average reward: -147.5, Average timesteps: 147.5
Episode: 1250, Average reward: -162.7, Average timesteps: 162.7
```

```
Episode: 1350, Average reward: -147.8, Average timesteps: 147.8
Episode: 1400, Average reward: -143.0, Average timesteps: 143.0
Episode: 1450, Average reward: -135.2, Average timesteps: 135.2
Episode: 1500, Average reward: -128.2, Average timesteps: 128.2
Episode: 1550, Average reward: -135.2, Average timesteps: 135.2
Episode: 1600, Average reward: -140.3, Average timesteps: 140.3
Episode: 1650, Average reward: -137.2, Average timesteps: 137.2
Episode: 1700, Average reward: -141.8, Average timesteps: 141.8
Episode: 1750, Average reward: -147.6, Average timesteps: 147.6
Episode: 1800, Average reward: -132.4, Average timesteps: 132.4
Episode: 1850, Average reward: -150.2, Average timesteps: 150.2
Episode: 1900, Average reward: -144.7, Average timesteps: 144.7
Episode: 1950, Average reward: -149.4, Average timesteps: 149.4
Episode: 2000, Average reward: -141.4, Average timesteps: 141.4
Episode: 2050, Average reward: -154.0, Average timesteps: 154.0
Episode: 2100, Average reward: -157.9, Average timesteps: 157.9
Episode: 2150, Average reward: -162.6, Average timesteps: 162.6
Episode: 2200, Average reward: -148.1, Average timesteps: 148.1
Episode: 2250, Average reward: -152.7, Average timesteps: 152.7
Episode: 2300, Average reward: -165.0, Average timesteps: 165.0
Episode: 2350, Average reward: -150.7, Average timesteps: 150.7
Episode: 2400, Average reward: -165.4, Average timesteps: 165.4
Episode: 2450, Average reward: -150.9, Average timesteps: 150.9
Episode: 2500, Average reward: -148.4, Average timesteps: 148.4
Episode: 2550, Average reward: -157.6, Average timesteps: 157.6
Episode: 2600, Average reward: -149.5, Average timesteps: 149.5
Episode: 2650, Average reward: -151.2, Average timesteps: 151.2
Episode: 2700, Average reward: -132.2, Average timesteps: 132.2
Episode: 2750, Average reward: -152.3, Average timesteps: 152.3
Episode: 2800, Average reward: -156.1, Average timesteps: 156.1
Episode: 2850, Average reward: -153.7, Average timesteps: 153.7
Episode: 2900, Average reward: -143.1, Average timesteps: 143.1
Episode: 2950, Average reward: -144.4, Average timesteps: 144.4
Episode: 3000, Average reward: -135.9, Average timesteps: 135.9
Episode: 3050, Average reward: -135.9, Average timesteps: 135.9
Episode: 3100, Average reward: -130.2, Average timesteps: 130.2
Episode: 3150, Average reward: -143.9, Average timesteps: 143.9
Episode: 3200, Average reward: -144.6, Average timesteps: 144.6
Episode: 3250, Average reward: -153.4, Average timesteps: 153.4
Episode: 3300, Average reward: -130.0, Average timesteps: 130.0
Episode: 3350, Average reward: -139.4, Average timesteps: 139.4
Episode: 3400, Average reward: -127.9, Average timesteps: 127.9
Episode: 3450, Average reward: -123.6, Average timesteps: 123.6
Episode: 3500, Average reward: -144.0, Average timesteps: 144.0
Car reached the top of the hill!!
```



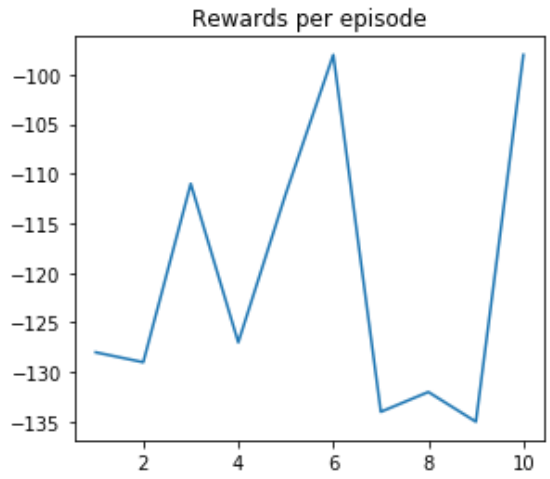


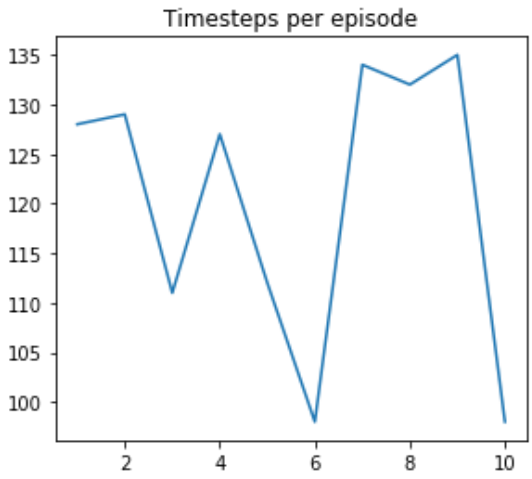


Having experimented for 3 days just tuning hyperparameters only on this Double DQN Mountain Car-v0, I learnt choosing number of episodes and minibatch size played a crucial part in final efficiency. At last, for an episode count of 4000 and minibatch size of 128 worked best for this, as the reward keeps on increasing, though oscillating a bit. I considered mountain car is solved if its reward over the last 10 episodes reaches above -110. After 3500 episodes the car is able to reach the top of the hill consecutively over last 10 episodes in less than 110 timesteps.

Evaluation results:

```
Episode: 1, Total Reward: -128.0, Timesteps: 128
Episode: 2, Total Reward: -129.0, Timesteps: 129
Episode: 3, Total Reward: -111.0, Timesteps: 111
Episode: 4, Total Reward: -127.0, Timesteps: 127
Episode: 5, Total Reward: -112.0, Timesteps: 112
Episode: 6, Total Reward: -98.0, Timesteps: 98
Episode: 7, Total Reward: -134.0, Timesteps: 134
Episode: 8, Total Reward: -132.0, Timesteps: 132
Episode: 9, Total Reward: -135.0, Timesteps: 135
Episode: 10, Total Reward: -98.0, Timesteps: 98
```



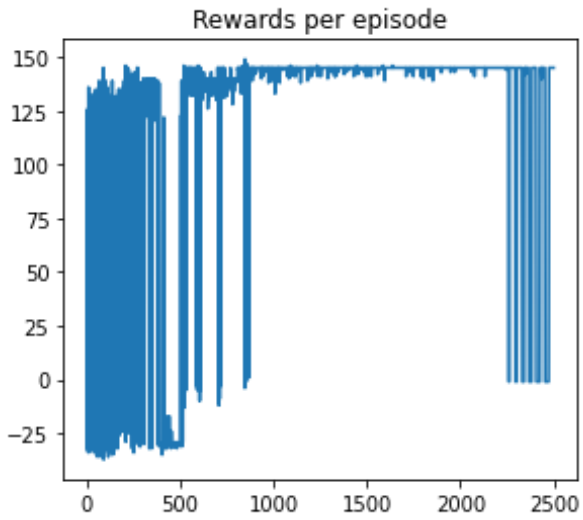


Though the mountain car is not stable, but the car is able to reach the top of the hill in less that 135 timesteps and even in < 110 in few episodes, so car did learn a way to accomplish its goal sooner.

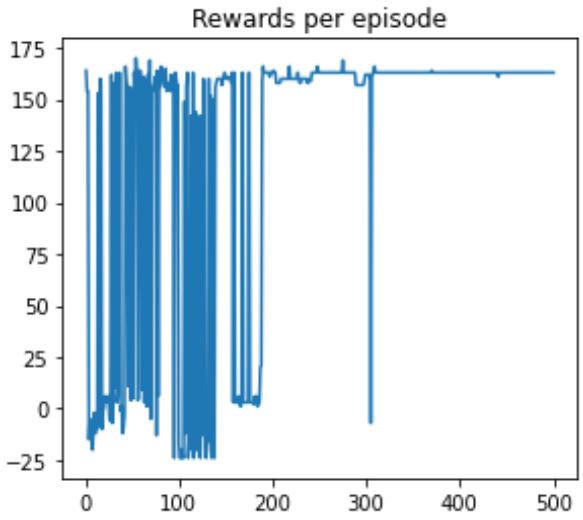
5) Comparison:

DQN vs Double DQN on Grid-World environment

DQN

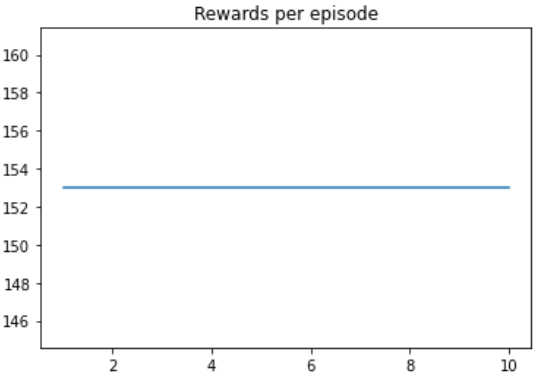


Double DQN

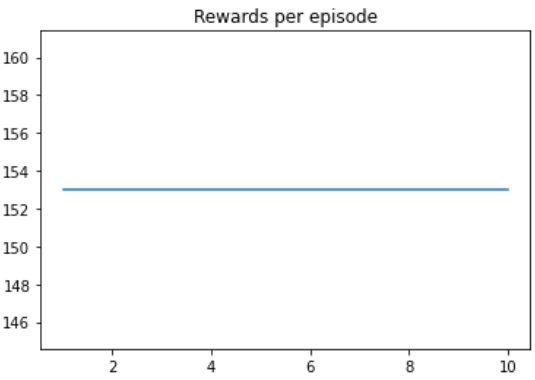


From the above two results we can infer that Double DQN is more stable while DQN varies a lot, the problem of overestimating, which can be reduced by using Double DQN where the policy networks outputs are adjusted based on the target values calculated from the target network, instead of using policy network values like in DQN.

DQN evaluation results



Double DQN evaluation results

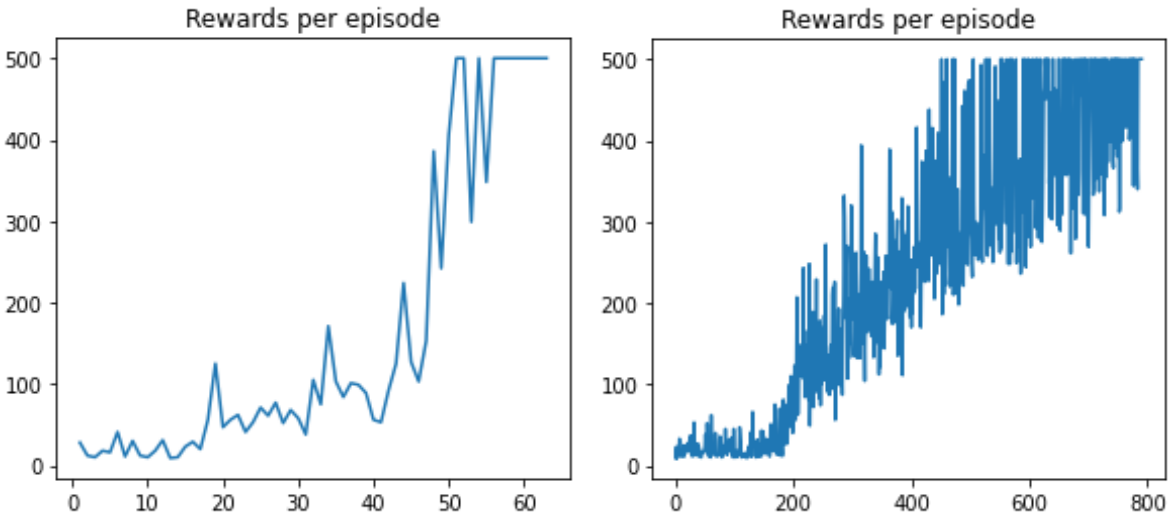


Both algorithms performed well on grid environment, picking maximum rewards following optimal path to the target.

Comparison:

DQN vs Double DQN on cartpole-v1:

DQN **Double DQN**

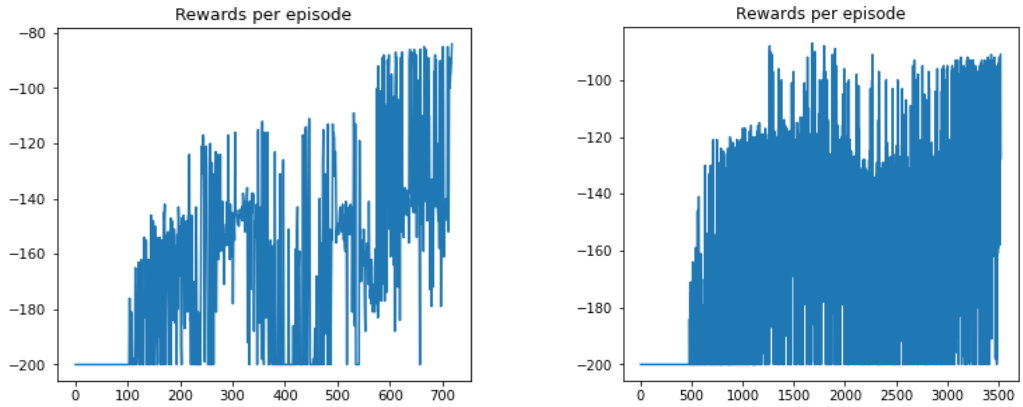


From the above two plots, we can infer that Double DQN takes more episodes to converge. But both algorithms are making the cartpole to balance for more timesteps with increase in episodes.

Comparison:

DQN vs Double DQN on MountainCar-v1:

DQN **Double DQN**



From the above two plots, we can infer that Double DQN takes more episodes to converge and even the reward is oscillating more in Double DQN. But both algorithms are making the mountain car to reach the top of the hill in less timesteps with increase in episodes.

6) Interpretation: From experimenting a lot, I understood that Double DQN requires a greater number of episodes as it takes slower steps and minibatch size also has major impact on time it takes to converge. DQN performed well on all the environments converging sooner and Double DQN did well on grid world while on cartpole and mountain car the rewards were oscillating until the end, as it the average over the last episodes. Overall, both algorithms performed well while double DQN took a lot of time to fine tune.

References:

https://gym.openai.com/envs/CartPole-v1/

https://gym.openai.com/envs/MountainCar-v0/

https://towardsdatascience.com/open-ai-gym-classic-control-problems-rl-dqn-reward-functions-16a1bc2b007