RL Assignment 2 Deep Q-Networks Checkpoint

Part 1: Exploring OpenAI Gym Environments

CartPole-v1

In CartPole environment a pole is attached to a cart via an un-actuated joint and the cart moves on a frictionless surface. The goal of the agent is to maintain the pole attached to it from falling. The agent can apply a force of +1 or -1 to the cart to keep the pole upright. For each timestep the pole is upright then the agent gets the reward of +1. The episode ends when the pole is more than 15 degrees from the vertical or the cart on which the pole is mounted moves more than 2.4 units from the starting position.

State / Observation Space:

The state space is of type box which has 4 observations in it:

1. Cart Position
2. Cart Velocity
3. Pole Angle
4. Pole Angular Velocity at the tip

The cart position can take values from -4.8 to +4.8. The velocity of the cart can take values from -infinity to +infinity. The pole angle can take value from -24 degree to +24 degree. The angular velocity of the pole can range from -infinity to +infinity. As we can see the state space of the cart pole environment is continuous.

Action Space:

There are two possible actions that the agent can do in the given state:

1. 0 (push the cart to the left)
2. 1 (push the cart to the right)

Rewards:

The reward space for this environment consists of only one reward:

1. +1

The agent gets a reward of +1 for every timestep. This also includes the terminal step taken by the agent.

Starting State:

For the starting position of the environment all observations of the environment are assigned a uniform random value in -0.05 to +0.05

Termination:

The episode is terminated when one of the following conditions occurs:

1. Pole angle is more than 12 degrees.
2. Cart position is more than 2.4 units from the center.
3. Episode length is greater than 500.

If the pole angle is 12 degrees from the starting vertical position, then the episode terminates. The episode also terminates if the cart moves 2.4 units from the center. At this timestep the cart moves out of the screen. Therefore, the episode terminates because the cart moves out of the display. If the total number of timesteps in the episodes exceeds 500 then episode terminates.

Solved Requirements:

The agent is said to be learnt if the average returns of the rewards during 100 consecutive episodes is more than or equal to 475.

BreakoutDeterministic-v4

Atari breakout is a classic arcade game in which we have to clear all the bricks in a given screen. There are 8 rows of bricks on top of the screen/image, and a paddle at the bottom that deflects a ball to remove the bricks without losing its track/missing the rebound. The agent must toggle the paddle such that all the bricks break by directly deflecting the balls to the bricks or via the side wall. If the agent misses to deflect the ball beyond a specific number of tries (3 generally), the episode ends.

State / Observation Space:

We are given an image in with dimensions {210,160,3}, which is an RGB image. We sample k frames (4 in general), stack them sequentially to get the exact information about the state in which the agent is in. For example, the ball is going up and gets deflected after breaking a brick, it goes down. We cannot represent this in a single image. It must be represented as a series of images so that the agent understands the exact space.

Action Space:

There are four possible actions that the agent can do in the given state:

1. Left (shift paddle to the left)
2. Right (shift paddle to the right)
3. Noop (don't shift paddle)
4. Fire (for starting and ending episodes)

Rewards:

The reward space for this environment consists of one reward:

1. +1 for each step in $(0, \infty)$.

Starting State:

The agent starts with all bricks intact.

Termination:

The episode is terminated when one of the following conditions occurs:

1. All bricks are broken.
2. The agent misses to deflect the paddle beyond a specified limit.

Solved Requirements:

The agent is said to be learnt if manages to break all the bricks once. There is no specific reward threshold for actual definition of the "Solved Atari".

MountainCar-v0

The MountainCar environment is an environment where the objective is that the car reaches the top of the mountain on its right. It must do so in less than 200 timesteps otherwise it must start again. At every point in time, we know two things, the cars position, and its velocity. If the cars position is > 0.5, the car is said to have reached the flag (or considered to have conquered the environment).

State / Observation Space:

We are given a vector of numbers which indicate the cars position and velocity.

Action Space:

There are 3 possible actions that the agent can do in the given state:

0. Accelerate to the left
1. Don't Accelerate
2. Accelerate to the right

Rewards:

The reward space for this environment consists of two rewards:

1. -1 for each step.
2. 0 for reaching the flag.

Starting State:

The agent starts at the bottom of the valley, in the middle of the left and right mountains (assigned value between -0.6 and -0.4).

Termination:

The episode is terminated when one of the following conditions occurs:

1. The car reaches the flag.
2. The episode length crosses 200 timesteps

Solved Requirements:

The agent has said to be learnt the environment if it reaches the flag in less than 110 timesteps.

Part 2: Implementing DQN & Solving grid-world environment

ROBOT GRID WORLD using DQN

Main Objective

The environment used is a grid world in which the Robot must explore and reach the bag of gold while collecting intermediate gold rewards. While exploring for the gold in the world the robot must avoid the monster and avoid falling into the pit.

States

The environment is a grid of 5x5 matrix. All the states can be represented using X and Y co-ordinates

S = { [4,0], [4,1], [4,2], [4,3], [4,4],

[3,0], [3,1], [3,2], [3,3], [3,4],

[2,0], [2,1], [2,2], [2,3], [2,4],

[1,0], [1,1], [1,2], [1,3], [1,4],

[0,0], [0,1], [0,2], [0,3], [0,4],

}

Actions

There are four possible actions that the robot can take.

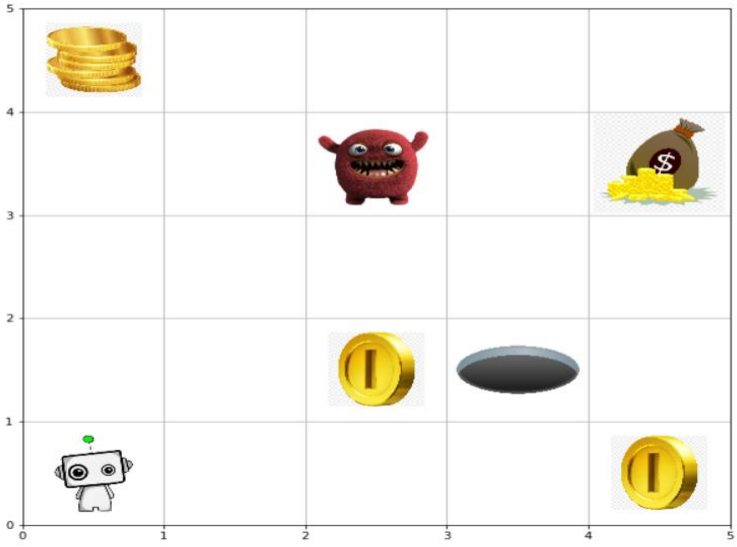
$$A \in \{Up, Down, Left, Right\}$$

Rewards

There are 4 positive rewards that the robot can receive. Out of 4, 3 rewards (+2, +2, +0.5) are intermediate rewards and final reward is for reaching the goal (+25) i.e. collecting the bag of coins. If the robot falls in the pit (-5) or gets eaten by the monster (-10), then the robot receives a negative reward. Additionally, for each action step taken by the robot it receives a negative reward (-0.5) and if the robot tries to go outside the grid then it receives a negative reward (-1).

$$R \in \{-10, -5, -1 - 0.5, +0.5, +2, +25\}$$

Fig 1: Visual Representation of the environment



- Agent starts at s = [0,0]
- Final goal position (bags of coins +10) is at s = [4,3]
- Intermediate Rewards 1 and 3 (+2) are at s = {[2,1], [4,0]}
- Intermediate Reward 2 (+0.5) is at s = [0,4]
- Pit (-5) is present at state s = [3,1]
- Monster (-10) is present at state s = [2,3]

Deep Q-Network:

In Deep Q-Network (DQN), we use deep learning for finding the optimal policy for the reinforcement learning agent. Instead of using a fixed representation of the policy (the Q-table in Q Learning), we let the neural network approximate the optimal policy for us. The input to the neural networks is observation of the current state in the environment. The neural network gives us the approximate Q-values for each action. The purpose of introducing deep learning is that the naïve Q learning is computationally expensive while having many issues such as over-estimating the values for each action. Neural network is fairly robust function approximation mechanism, that overcomes many issues faced by naïve Q learning. But as with every other thing, it has a few requirements that we need to tackle before implementing it for finding the optimal policy. One issue is that the input to the neural network is going to be sequential. That leads to a bias (due to correlation) and doesn't help the neural network find a stable optimal policy. With slight changes in the Q value, the policy changes quickly. Naïve Q-Leaning gradients are large. Deep leaning resolves these issues. In DQN, we use a replay memory buffer to store the experiences and randomly sample experiences out of this buffer for the final learning. This resolves the correlation issue. Along with random sampling, we use two neural networks, one for the policy and another one for determining the target. If we were to use one network for both the policy and target determination, the optimization is will be unstable and will never be able to converge on the optimal policy. Finally, the rewards are normalized and thus help the neural network converge towards to the global optimum.

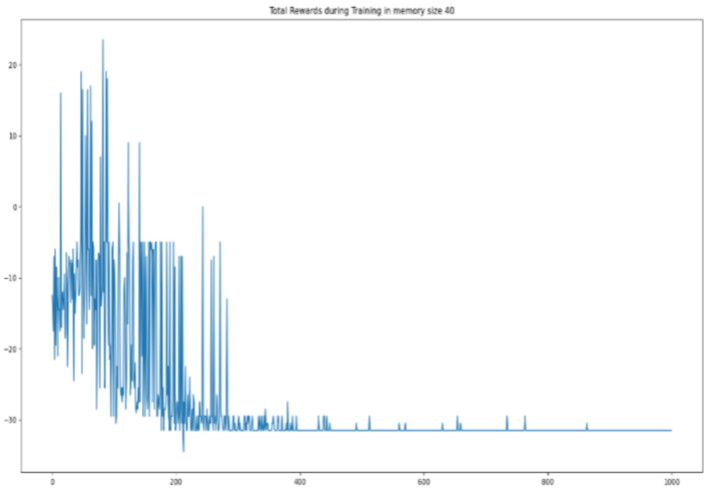
1. **Experience Replay**
   In the naïve Q-Network the input to the network is sequential. This causes the inputs to be correlated. Because of the correlation the updates of the weight occur on linear data and it can lead to problem of overfitting. To overcome this issue in DQN a experience replay is used. In this, after every timestep the experience is stored in the memory. Then random experiences of a certain size are picked up from the memory and the loss is calculated which is then back propagated in the network to update the weights of the network. Because of this randomness of the experiences, overfitting doesn't happen.
2. **Target Network**
   The problem with using just one network is that the network is updated after every loss is back propagated, this causes the target to be updated as well. In the Q-Networks we try to minimize the loss. The loss is given by
   $L(w) = E[(r+ \Upsilon \max Q(s`, a`, w) – Q(s, a, w))]$. If both Q max of next state and Q of current state are selected using the same network it will change the weight of the target as well this can be interpreted as the network is trying to improve the weight to reach the target while changing the target as well. Hence an additional network is used to calculate the target. So, during the training the target is calculated using the Target network and weights are updated of the neural policy network. After certain timesteps the weights of the policy network are copied to target network.
3. **Q function as q(s, w)**
   Since we are using neural network for the implementation of the Q learning. The action taken is dependent on the weights of the neural network. The weights are the deciding factor, based on weights the Q values of each action in that current state are determined. The Q learning gradient is the derivative of the loss function with respect to w. The weights are factors which decide to learn the optimal policy and hence the Q function is defined as function of w.

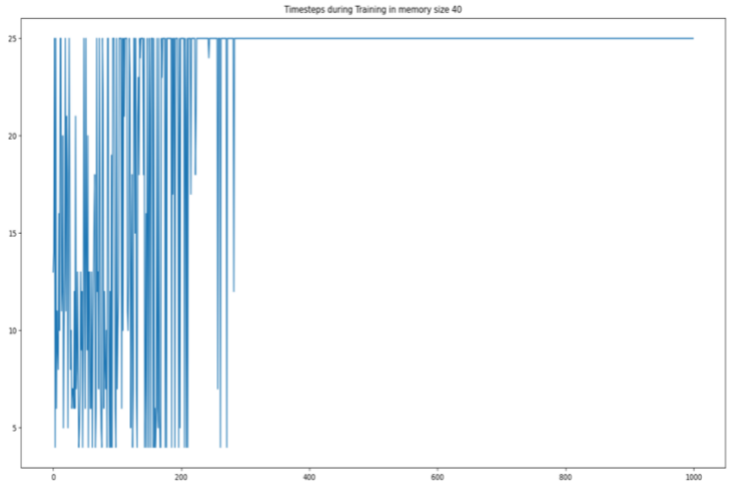
Implementation of DQN on the Robot Grid World

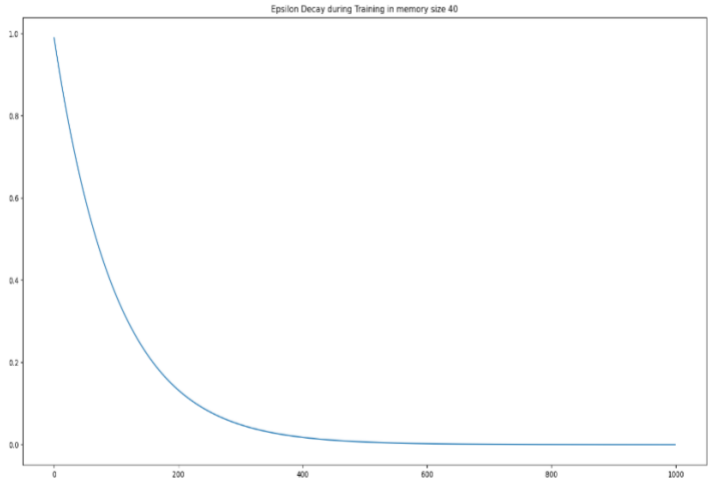
The Robot Grid World was trained using DQN. The agent was trained over 10000 episodes. Two networks were used a policy network and a target network. Both were initialized with random weights in the beginning. Batch size of 40 was used for training of the policy network, and after every 5 timesteps the weights were transferred to the target network. The epsilon decay was set to 0.01 with discount factor of 0.9. Initially the epsilon was set to 1 and after every episode the epsilon value was reduced. During the implementation of the DQN, different sizes of the replay memory were used. The results of the different replay memory size are given below. The graphs include the total rewards, timesteps and epsilon decay during the training episodes and the total rewards and timesteps during testing episodes when the agent was tested using the same network.

Replay Memory Size 40

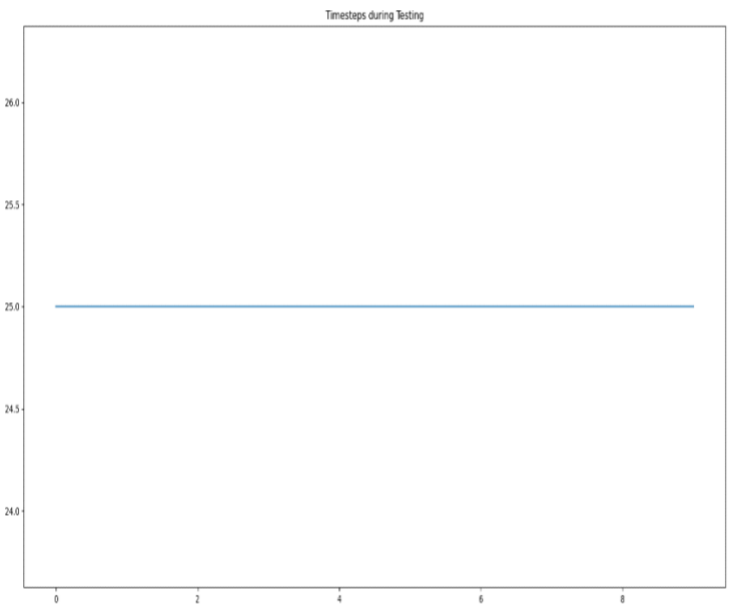
Training

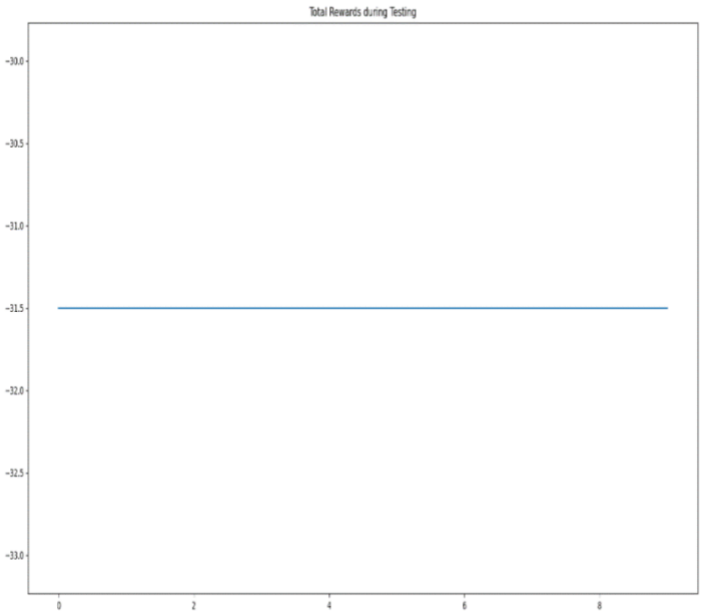






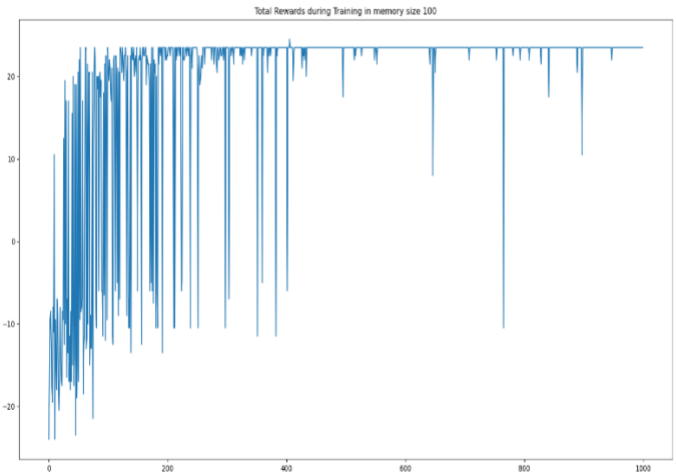
Testing

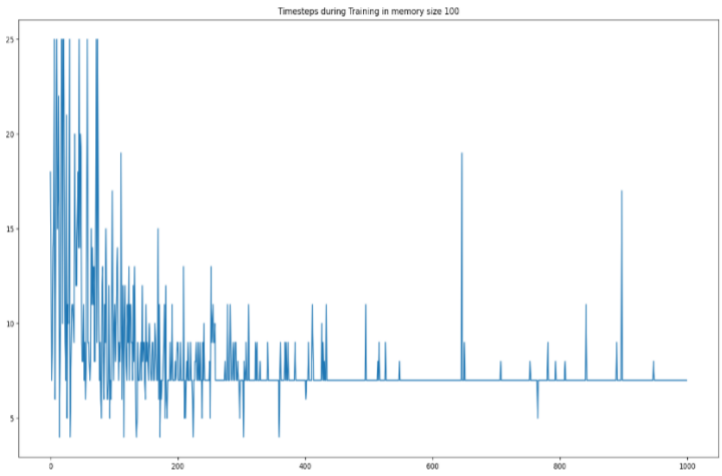


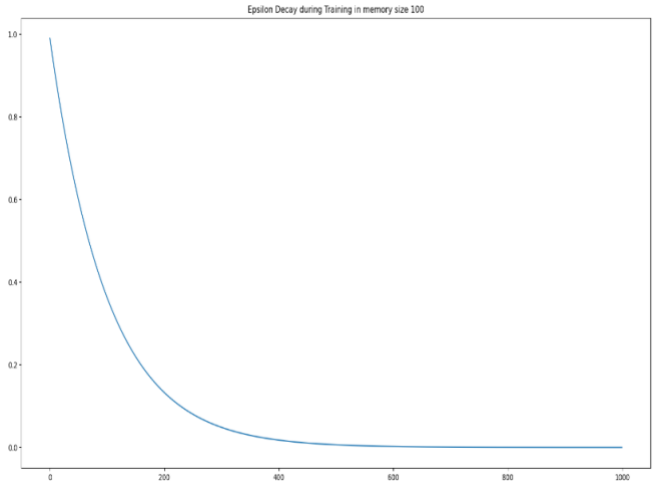


Replay Memory Size 100

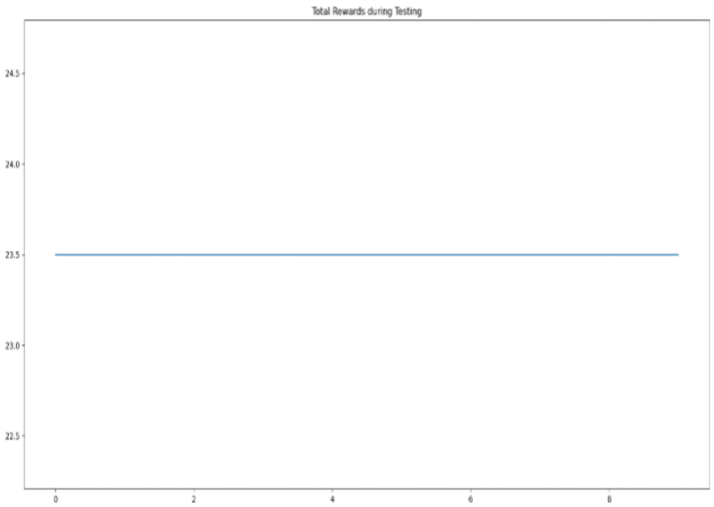
Training

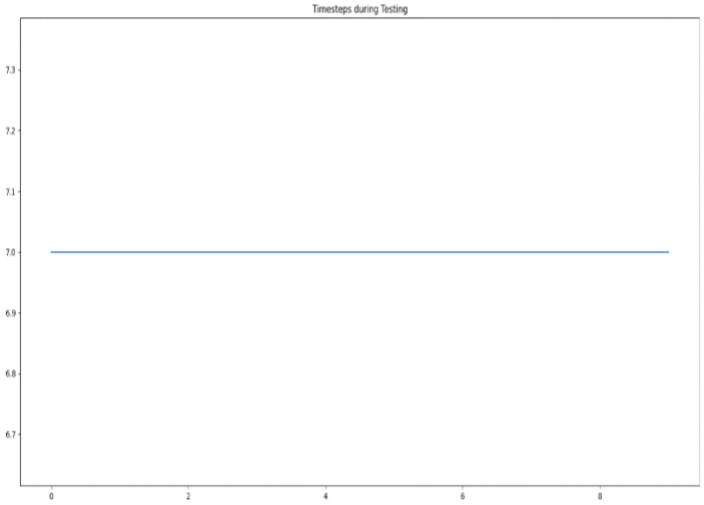






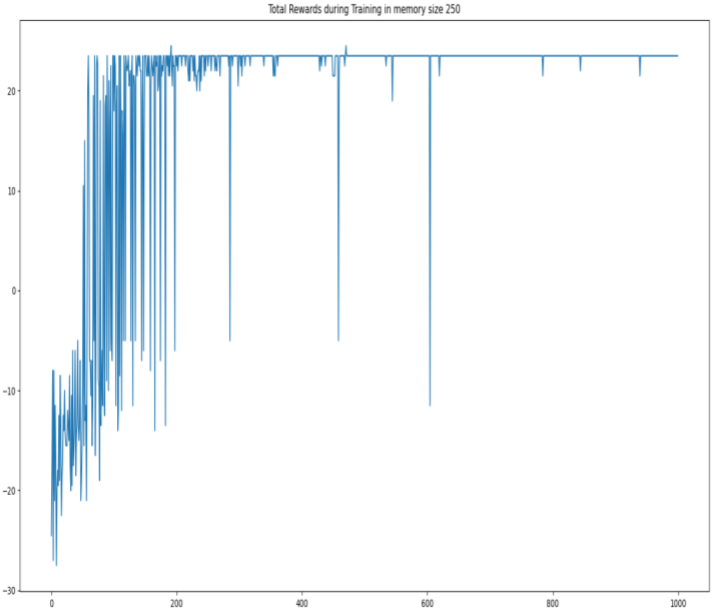
Testing

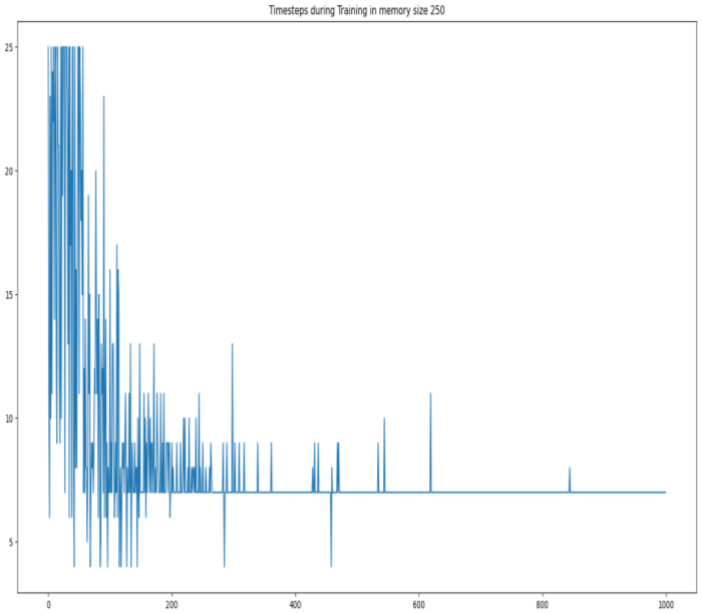


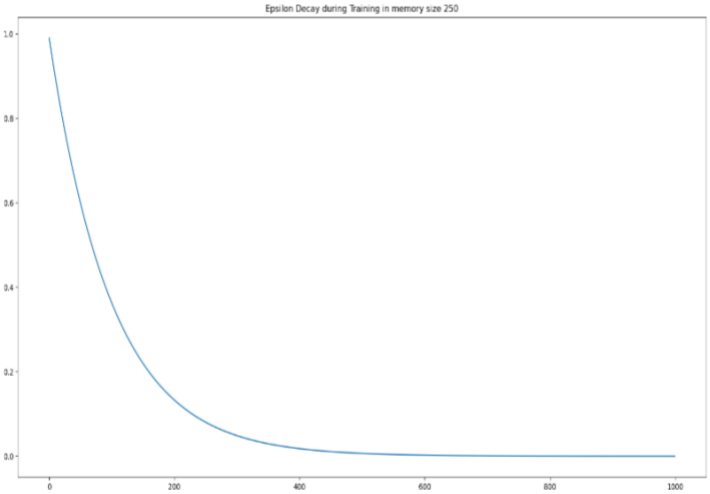


Replay Memory Size 250

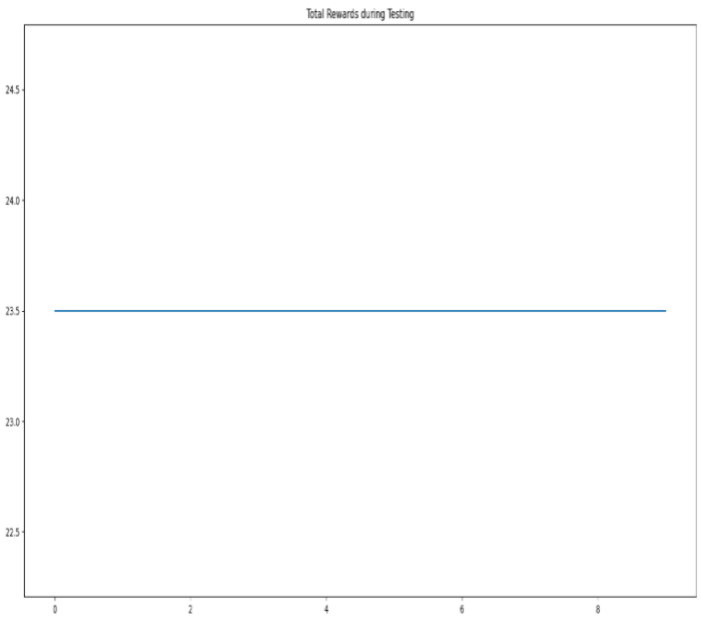
Training

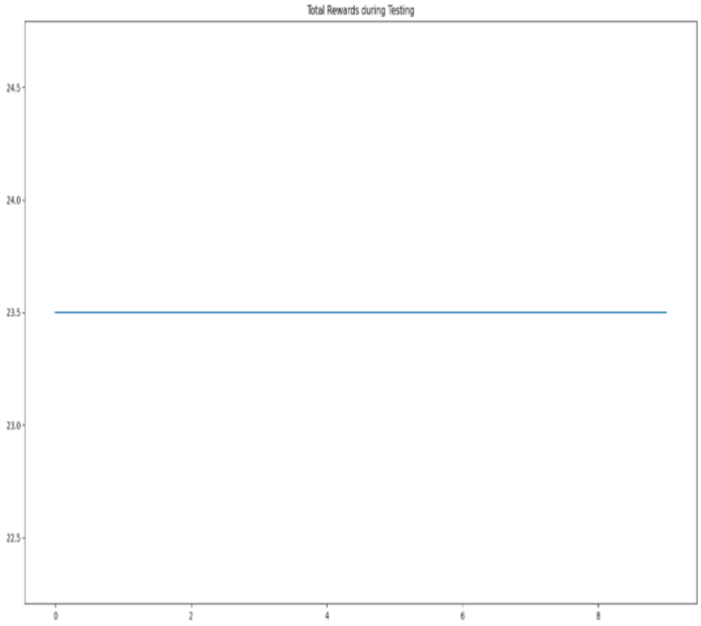






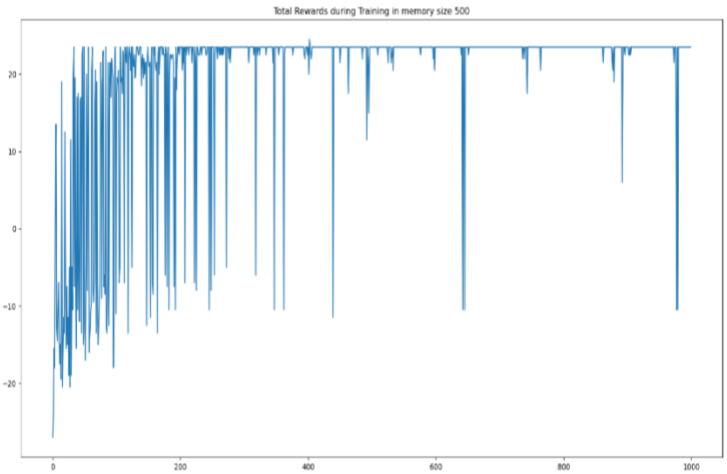
Testing

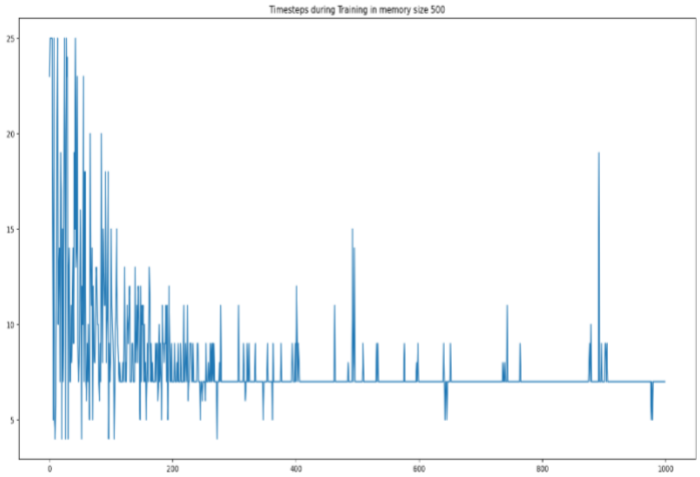


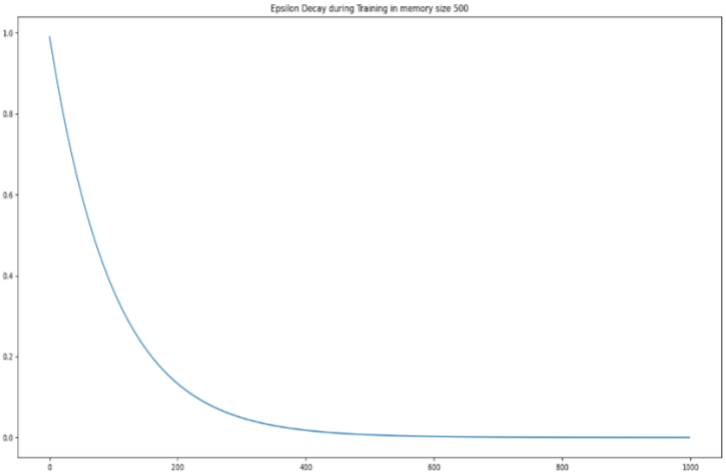


Replay Memory Size 500

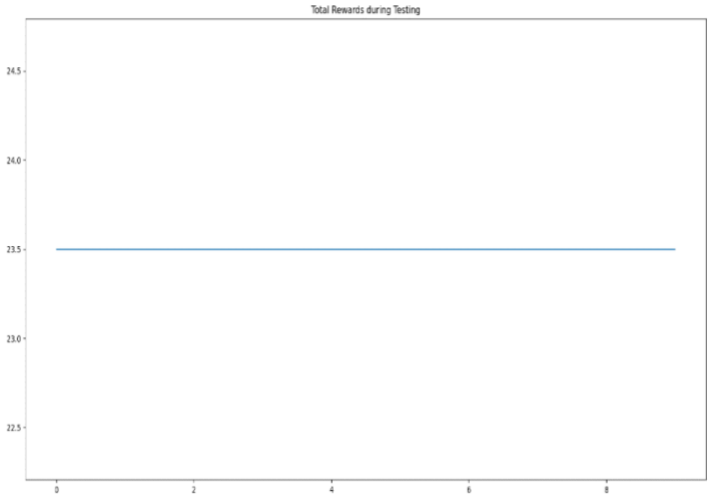
Training

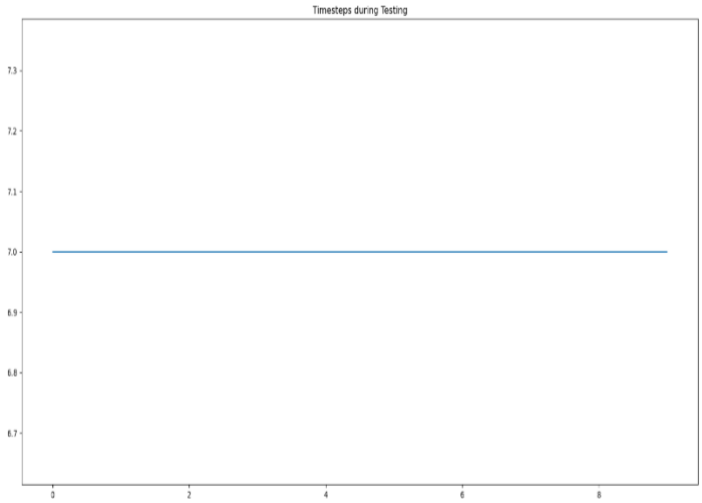






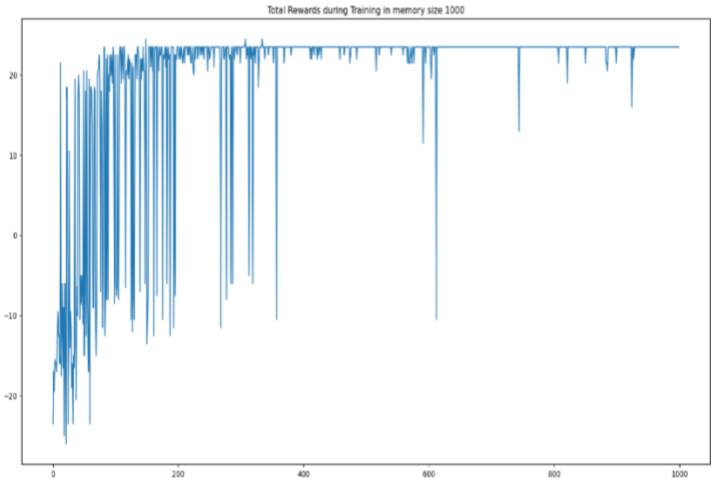
Testing

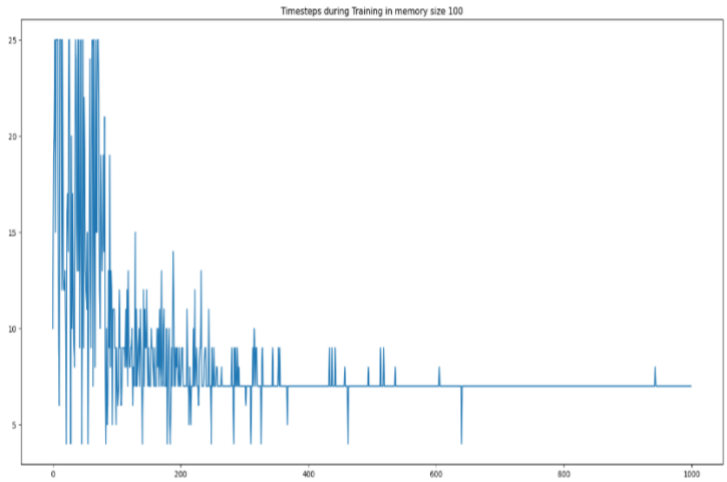


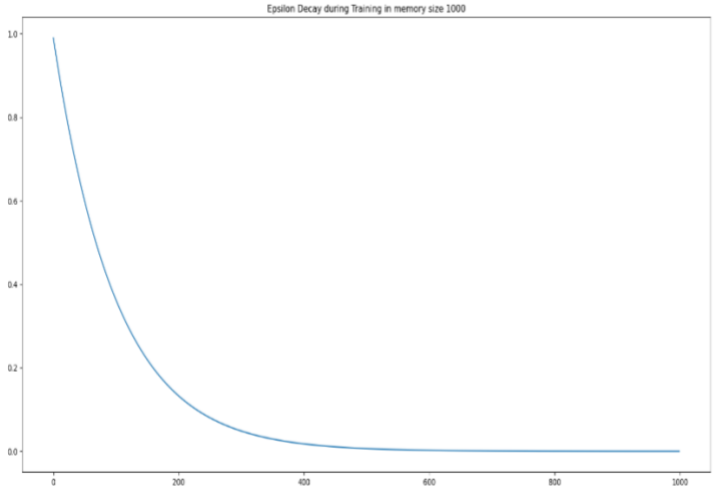


Replay Memory Size 1000

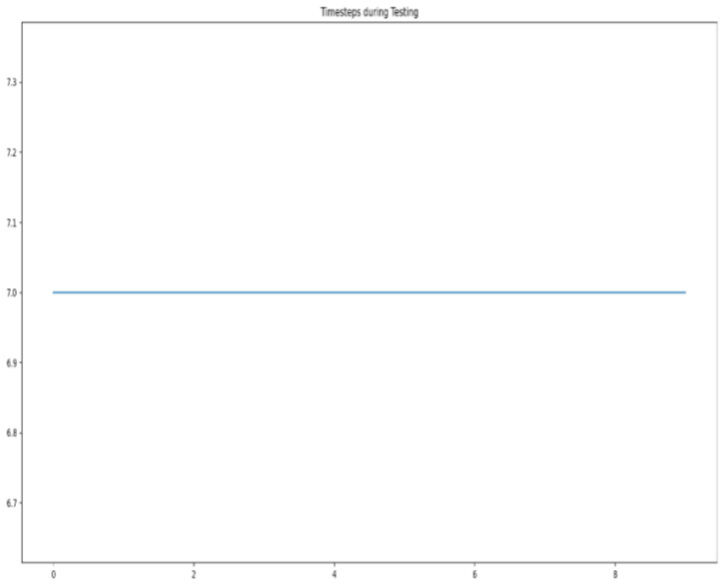
Training

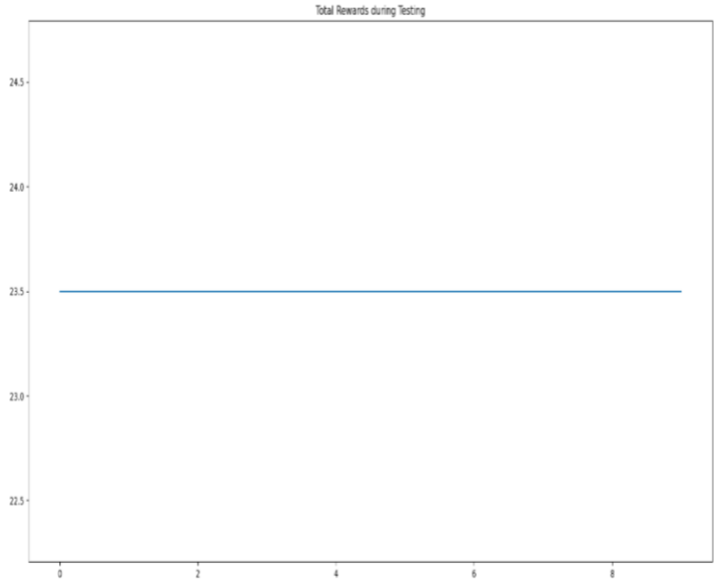






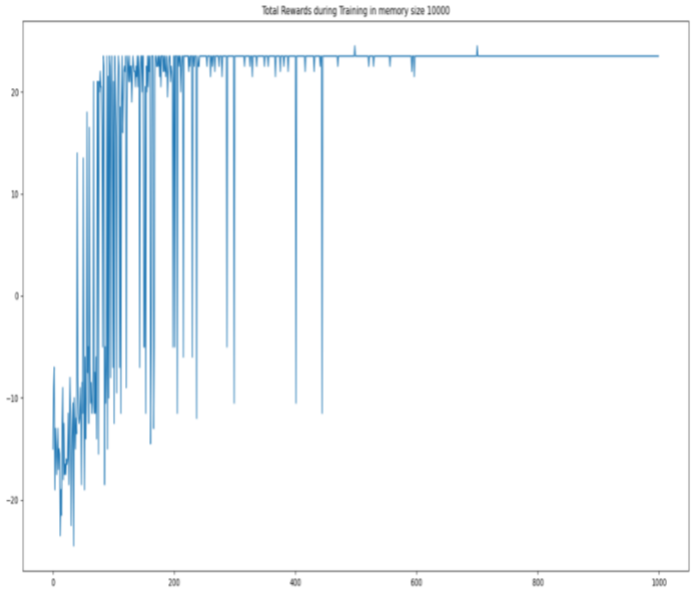
Testing

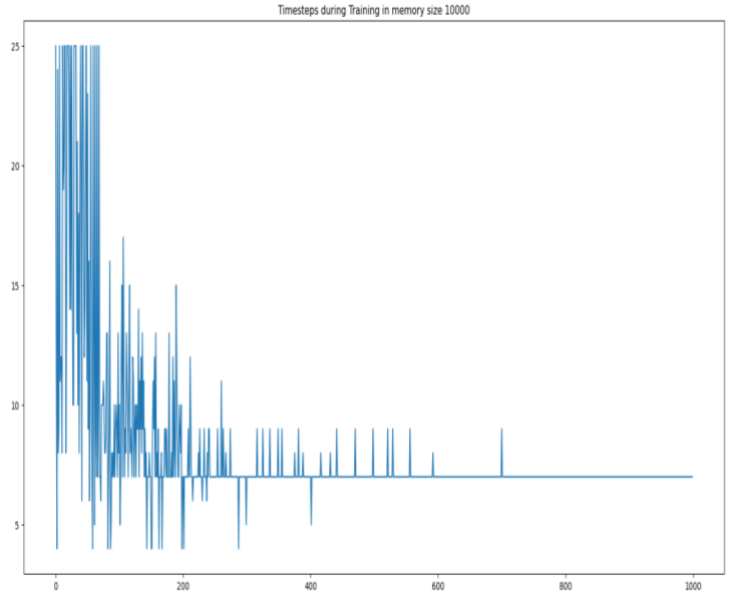


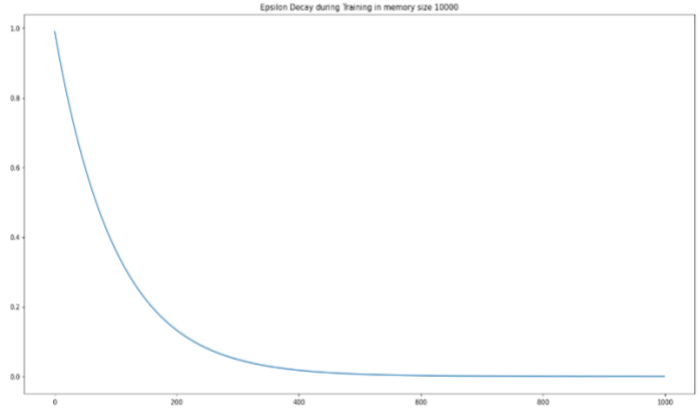


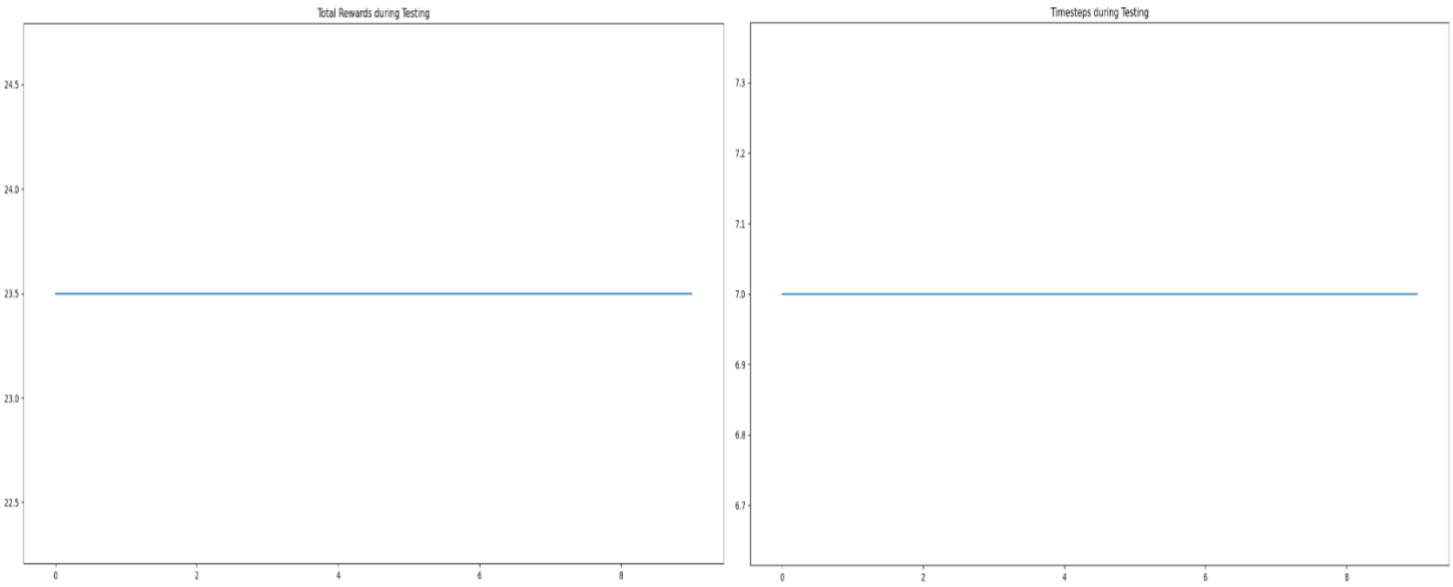
Replay Memory Size 10000

Training









As we can see if the replay memory is too small in the first case it was equal to batch size then the agent doesn't learn anything, and it just keeps exploring and runs out of timestep and terminates. As the memory replay size increases, more and more experiences of the timesteps are stored inside the buffer. There are more experiences from which the random experiences are selected for training of the policy network. This helps in updating the weights of the neural network properly. As it can be seen if the size of the replay memory is increased then the agent learns and converges faster.

One point that we can observe from the graphs is that there are "dips" in the total reward graphs. There can be multiple reasons for this. First is that the agent hasn't explored the entire state space. This leads to the agent reaching the pit, ending the episode prematurely and giving a reward that is proportional to the reward given by reaching the pit. Another reason for the dip maybe just the fact that the agent hasn't encountered a particular state before and is hence exploring the environment. Following from the above assumptions, we can see that when we increase the size of the replay memory buffer, the number of "dips" in the total rewards per episodes decreases and becomes stable as the number of episodes increases.

Part 3: Improving DQN & Solving OpenAI Gym Environments

The vanilla DQN faces the problem of Maximization Bias. In Q-Learning the target value is computed using the formula $\Upsilon \max_{a'} Q^*(s', a')$. The max Q value is sometimes overestimated. This overestimation of the Q value leads to introduction of Maximization bias in the learning process as the loss computed using these values is compounded. Since the agent learns from these Q value estimates, this can lead to problems in the learning process. To solve this problem of overestimation Double Q Learning is introduced. In double Q learning the target is calculated using $\Upsilon Q(s, \text{argmax}_a Q^*(s', a'))$.

In double Q learning instead of just using one estimator which can lead to overestimation, two estimators are used. So even if one estimator overestimates the Q value the other estimator controls this when calculating the maximum of Q. The target network of the vanilla DQN is used as a second estimator to avoid overestimation by the policy neural network. The Q value of double DQN is calculated using

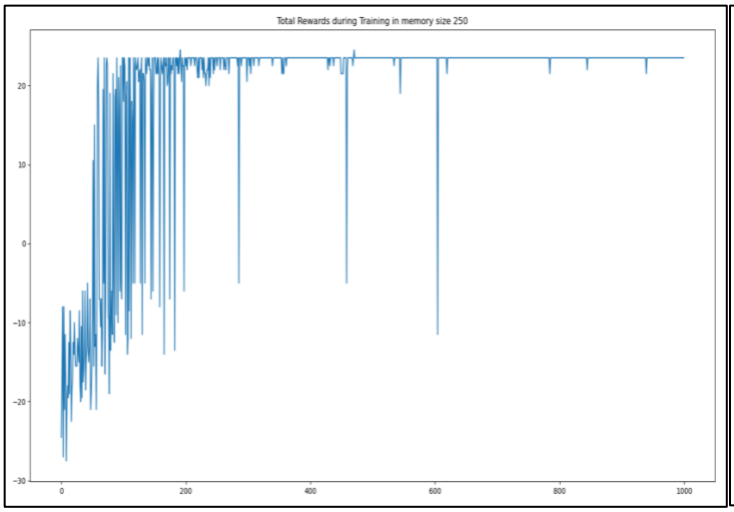
$$Q^*(s_t, a_t) = r_t + \Upsilon Q_\theta(s_{t+1}, \text{argmax}_{a'} Q_{\theta'}(s_{t+1}, a'))$$

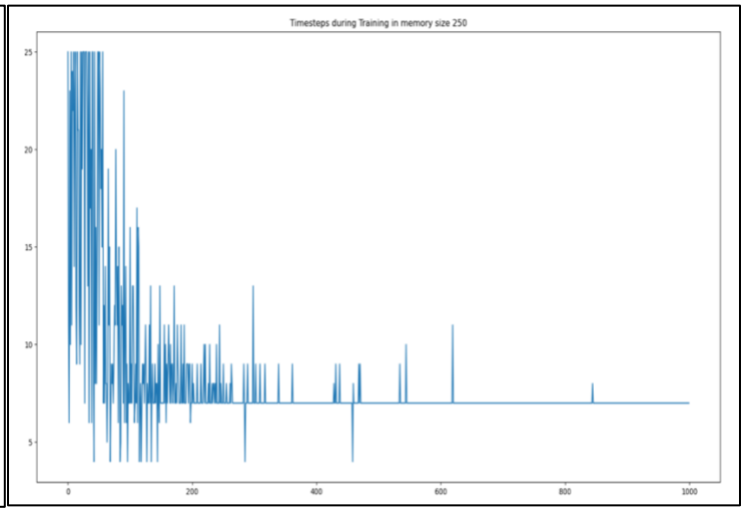
To improve the vanilla version of the DQN we have used the Double DQN

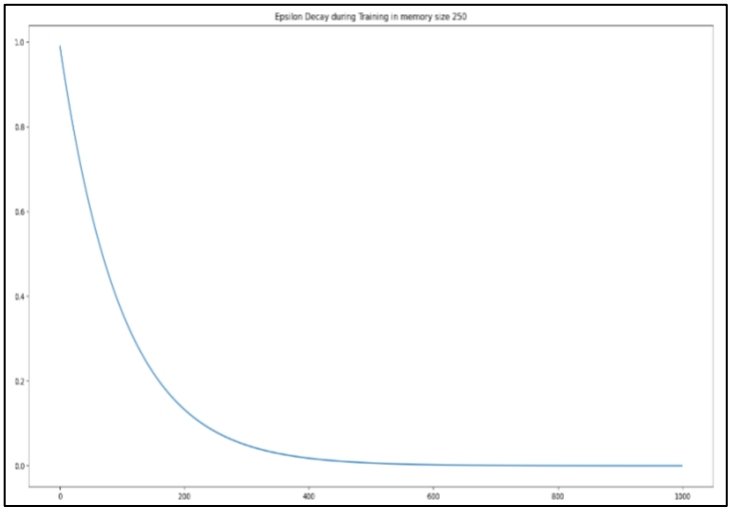
Results of Double DQN and DQN of different environments are given below:

1. Robot Grid World

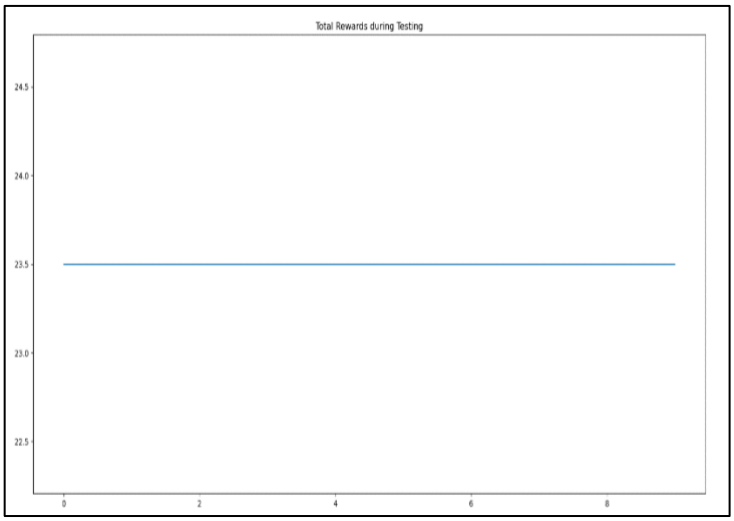
DQN Training

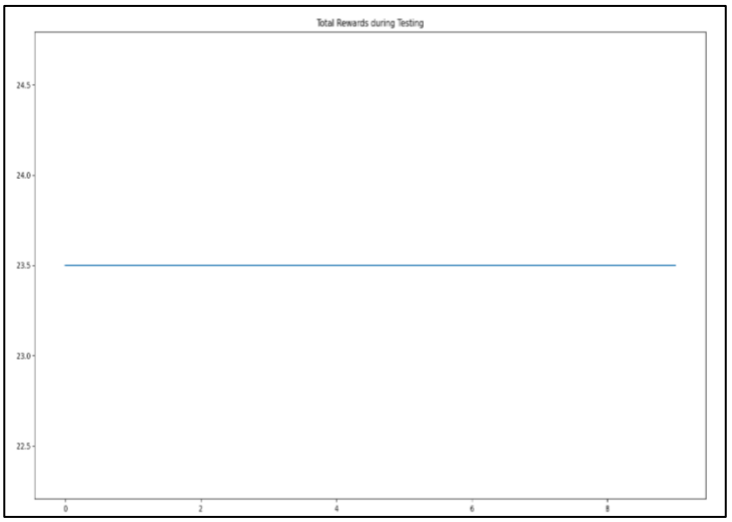






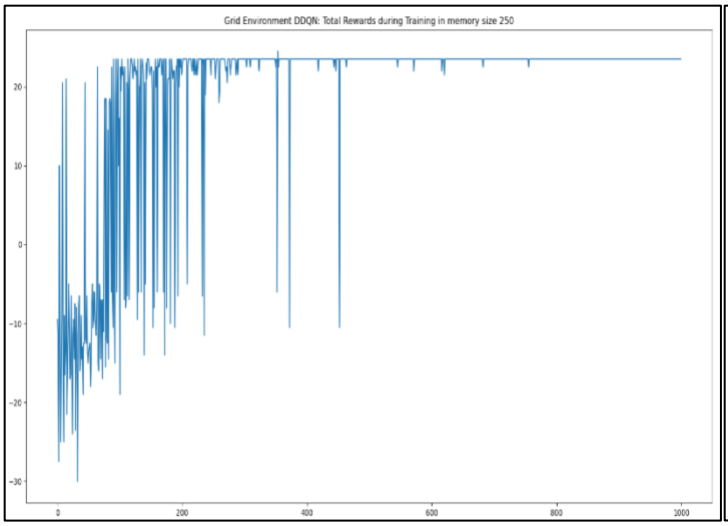
DQN Testing

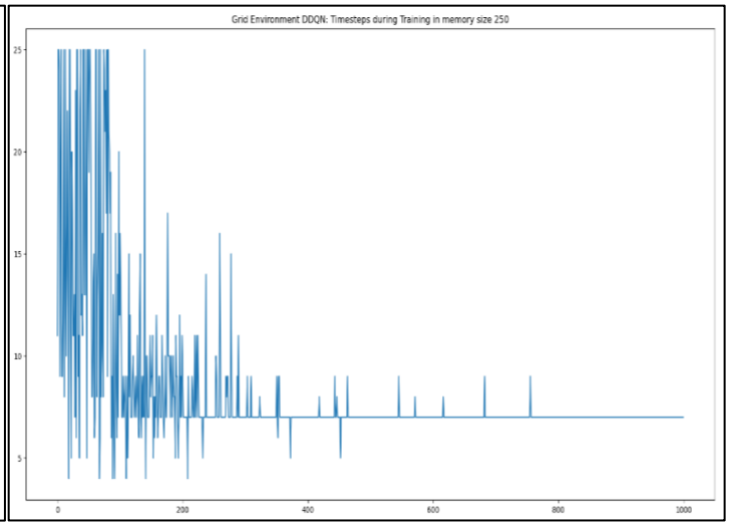


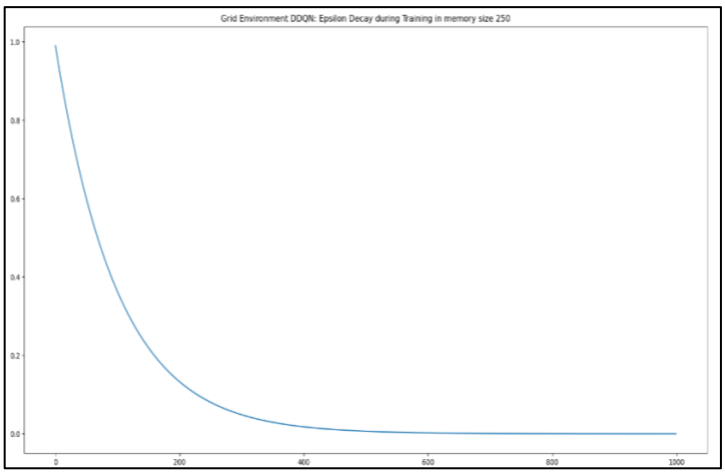


From the DQN graphs, we can clearly see that the algorithm starts converging from the 100th episode and fully converges by the 200th episode. We can also see that there are many dips in the optimal rewards when the algorithm has fully converged. It can be assumed that the dips occur since the agent overestimated on certain states of the environment.

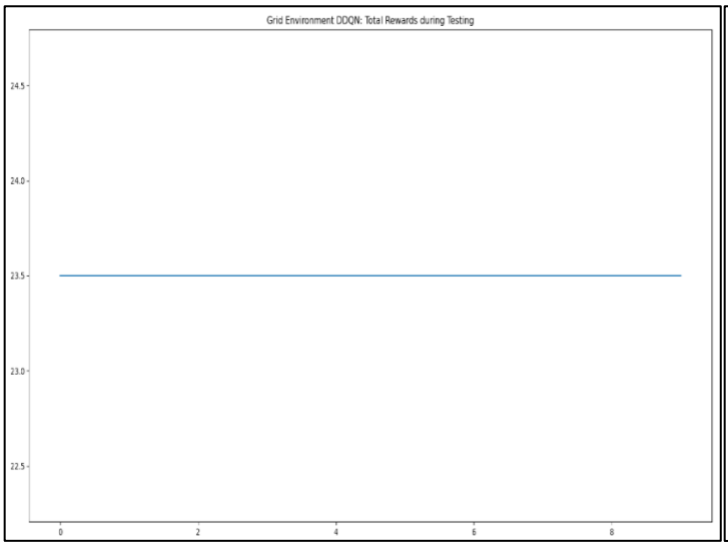
Double DQN Training

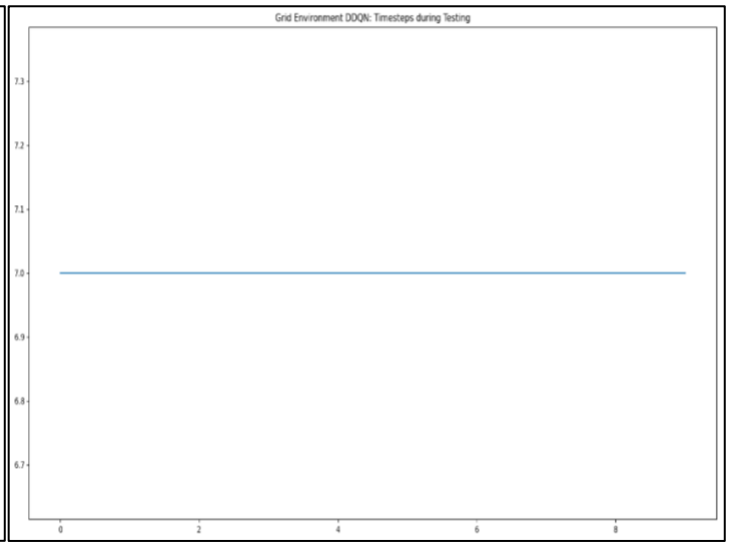






Double DQN Testing

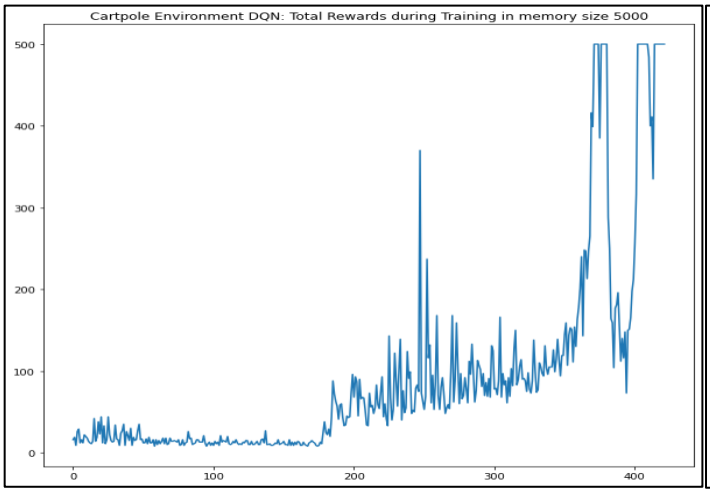


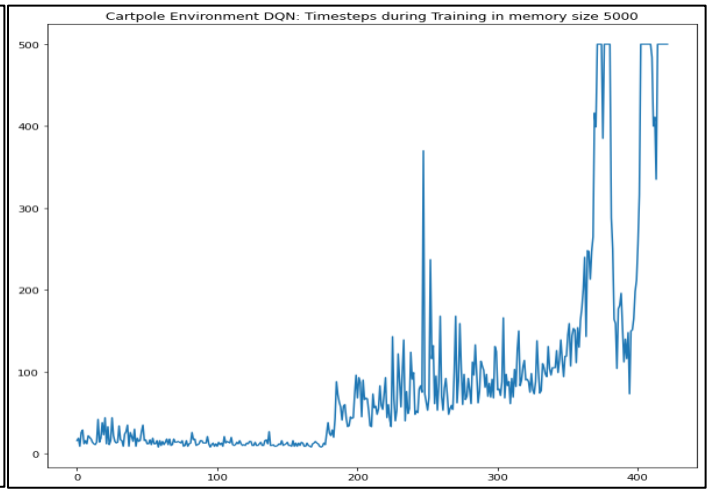


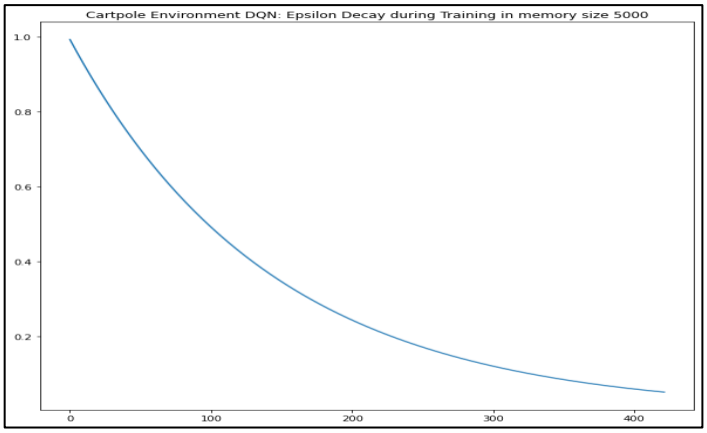
For the grid world, the perform of DDQN and DQN are very similar for the time taken to converge to the optimal reward structure. The main difference is post convergence, where DQN has more significant dips after converging to the optimal policy and continues to be a bit unstable in the long run. Overall, DDQN converges after approximately 200 episodes.

2. CartPole-v1

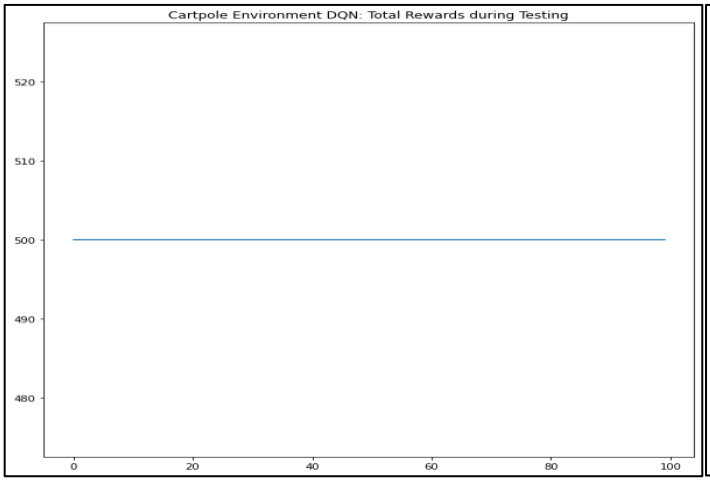
DQN Training

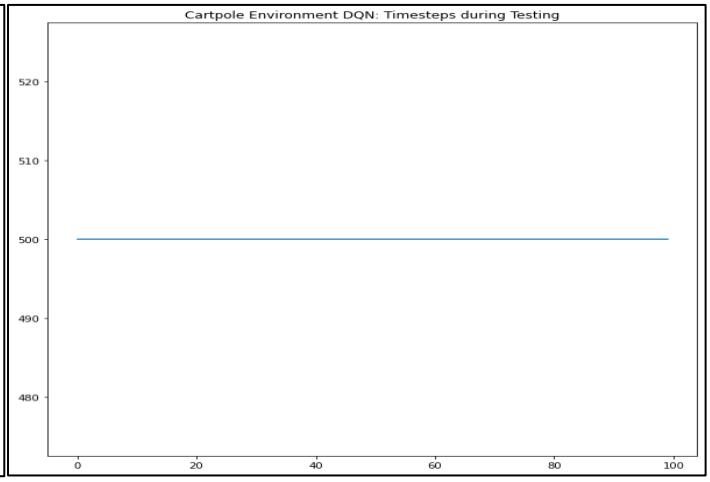






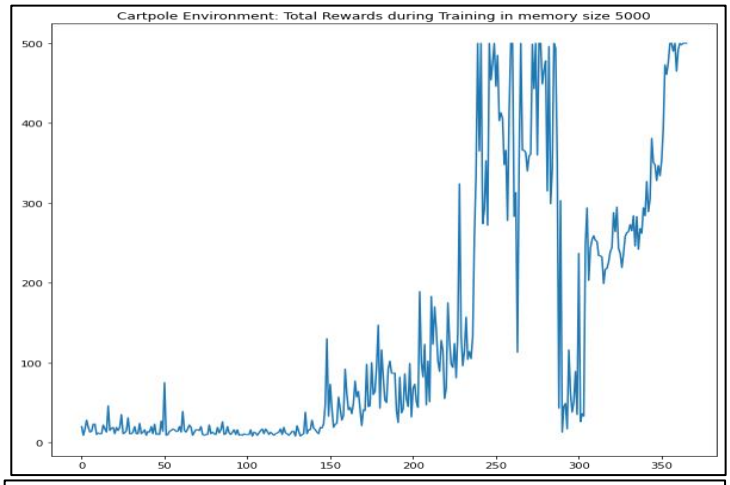
DQN Testing

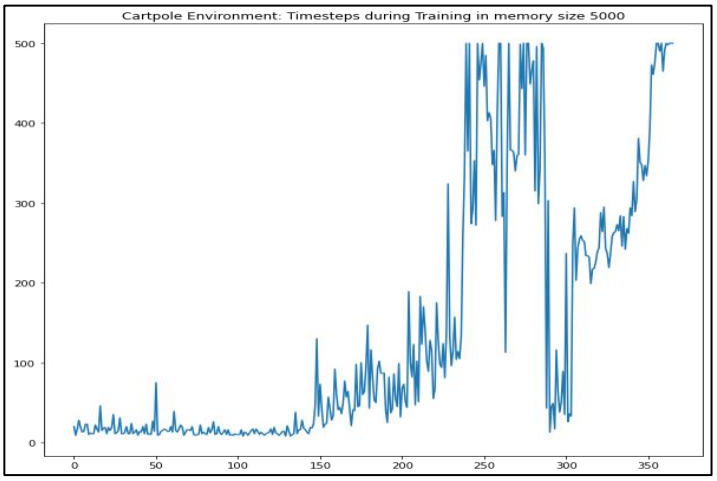


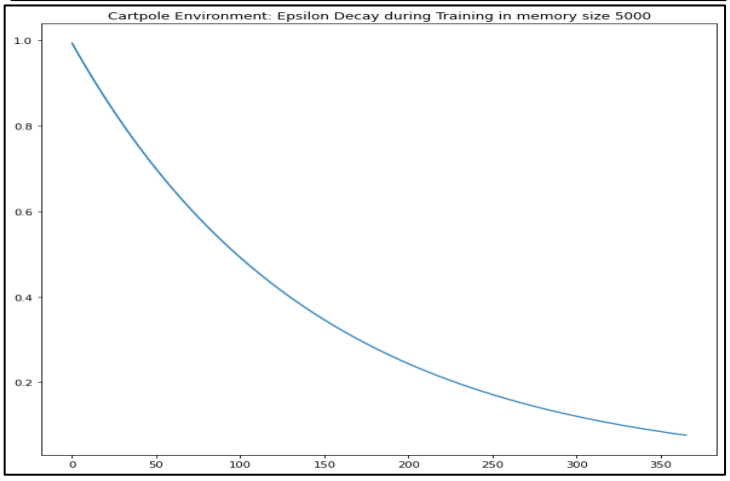


For the Cartpole environment, we incorporated multiple changes in the training phase, namely early stopping (stop training if optimal reward is achieved for 10/15 episodes) and training after 5/10 timesteps per episode. From the graphs, it can be clearly seen that the algorithm converges after 400 episodes and gives consistent rewards during the testing phase. The dips observed in the graphs can be attributed to the fact that the agent has not observed all the states, however it quickly recovers to find the optimal reward since it has already approximated the best Q values.

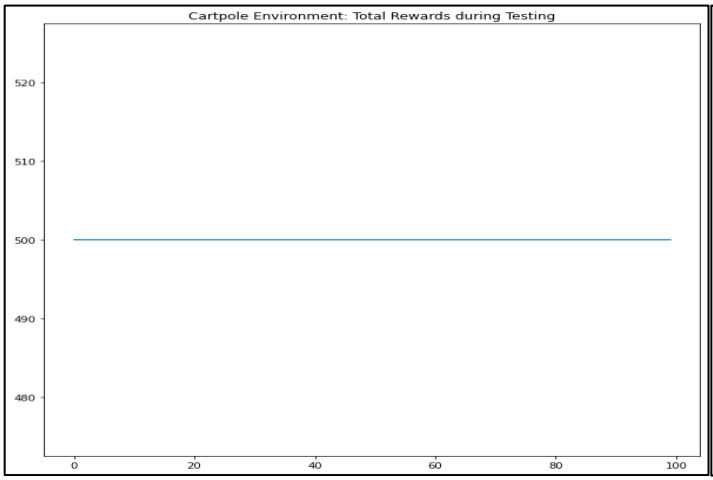
Double DQN Training

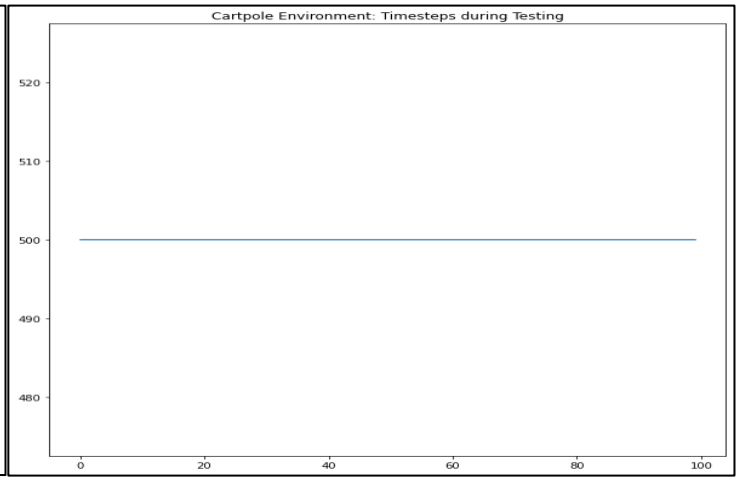






DDQN Testing:

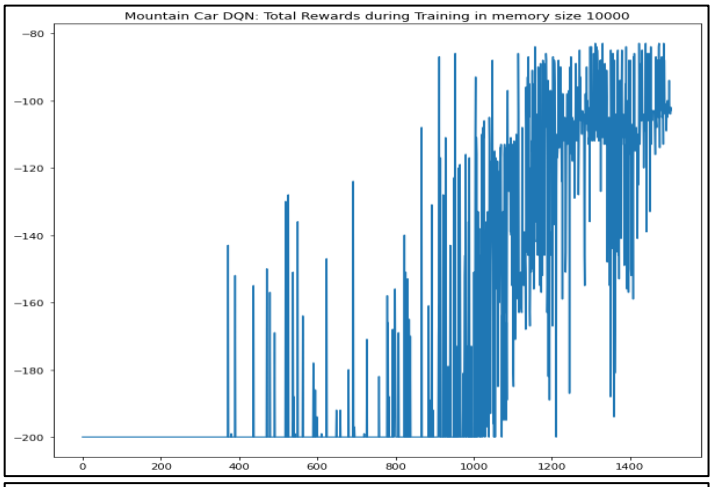


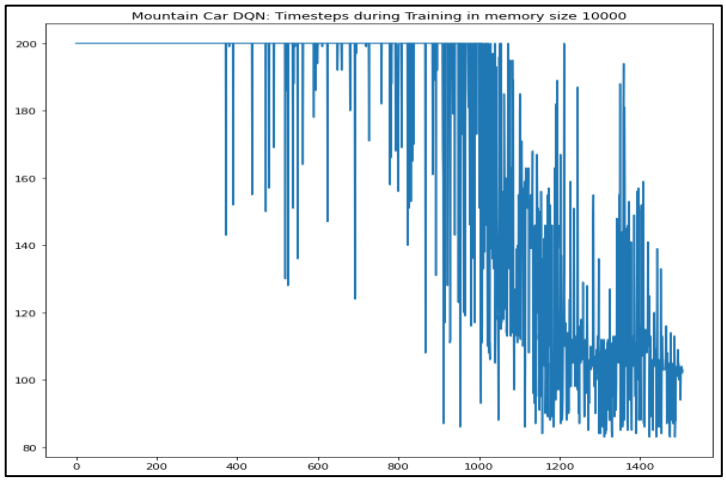


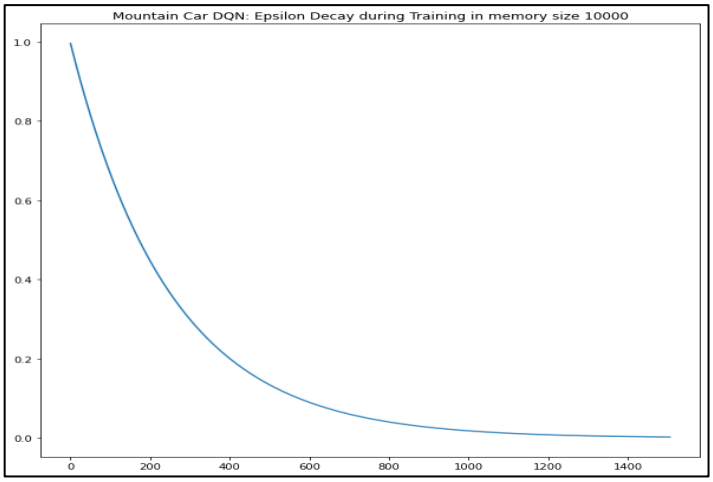
Compared to the DQN graphs, you can see that DDQN algorithm is converging similarly. However over multiple trainings, we observed that DDQN converged slower as compared to DQN. This is primarily because we have overcome the overestimation bias in DDQN. The DDQN algorithm converges after approximately 350 episodes to 800 episodes (350 for the above graphs). Similar to the DQN graph, we can dips in the reward graph, which can be attributed to the fact that the agent has not explored all the states in the environment.

3. MountainCar-v0

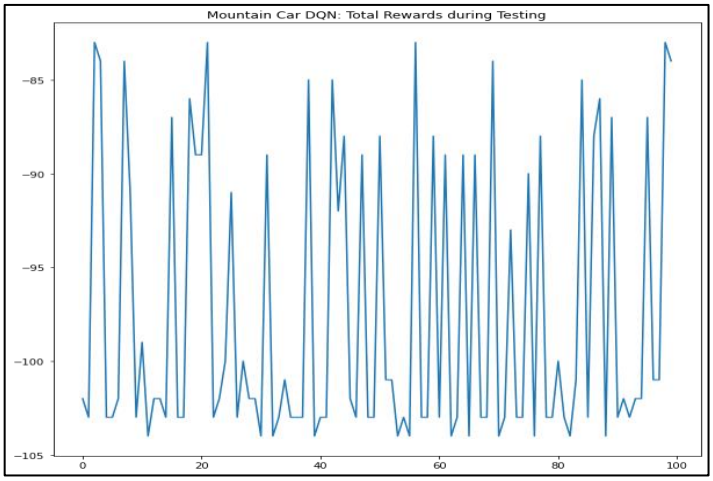
DQN Training

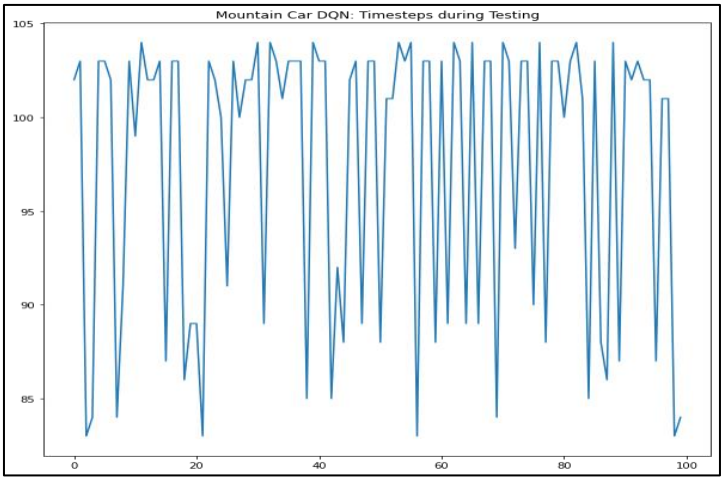






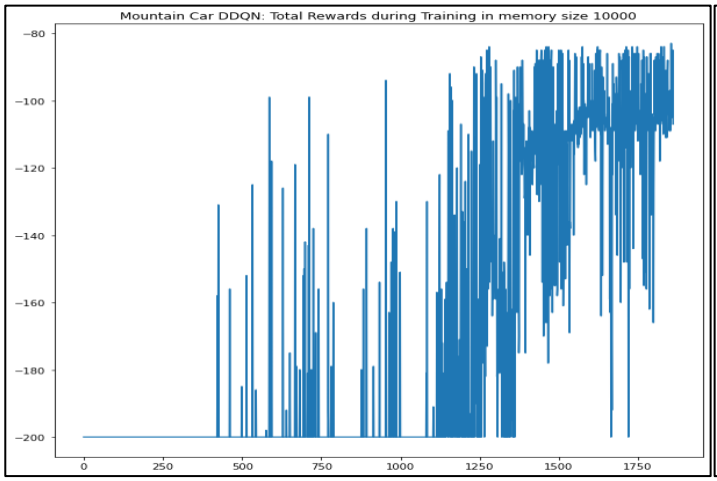
DQN Testing

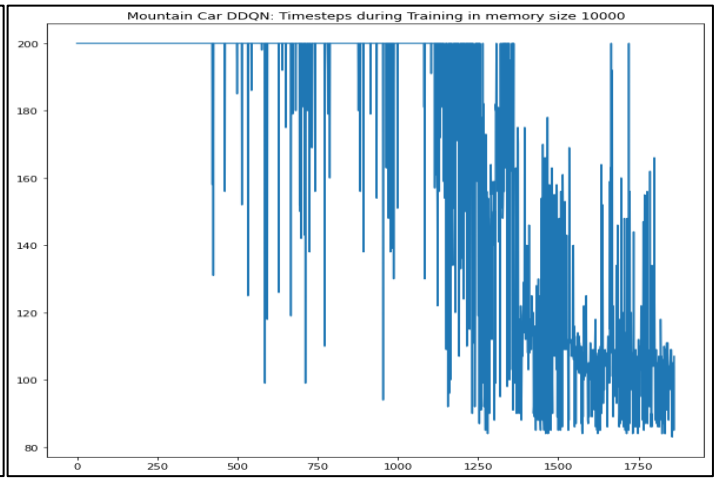


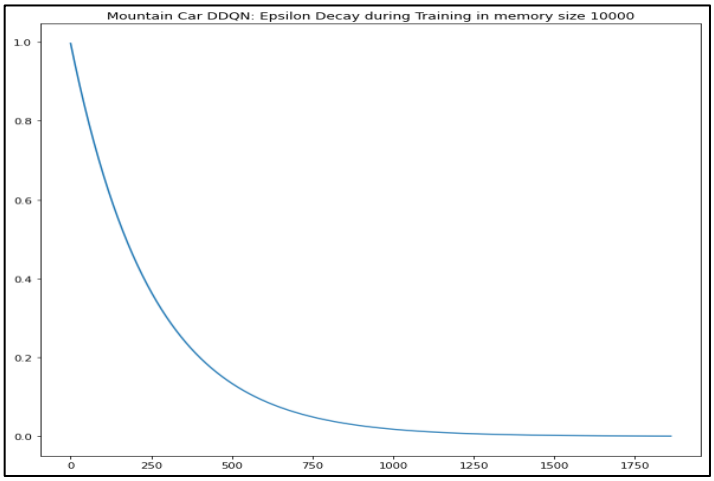


For MountainCar, we went with the same changes for the training loop for the cartpole environment. From the graphs, it can be clearly seen that the algorithm converges after 1400 episodes and achieves the objective of reaching the flag in less than 110 timesteps. Again, there are a few “dips” after the agent has formulated the optimal policy, this can again be attributed to the fact that the agent has not seen all the states of the environment and thus it incorporates those new states to form the optimal policy.

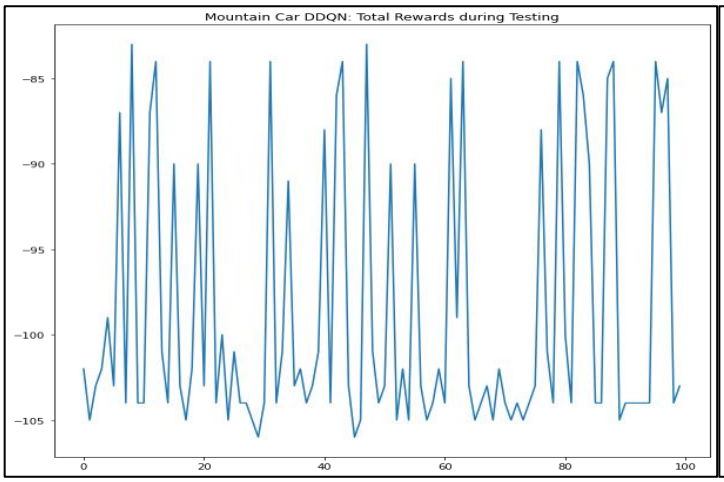
Double DQN Training

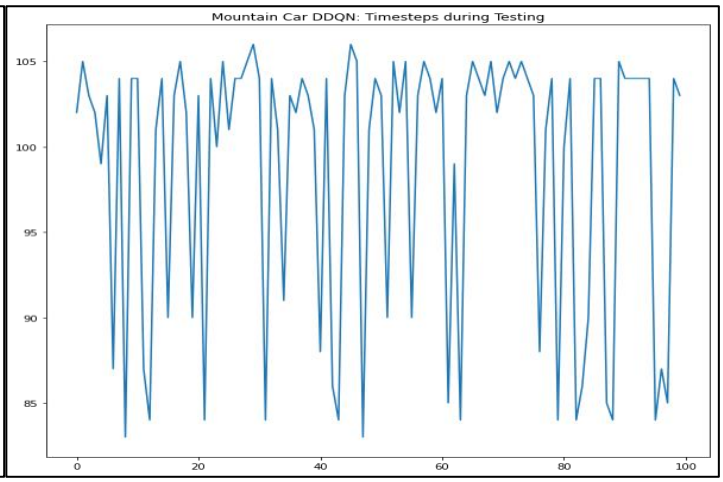






DDQN Testing:





Compared to the DQN graphs, you can see that DDQN algorithm is slower to converge to the optimal rewards. This is primarily because we have overcome the overestimation bias in DDQN. The DDQN algorithm converges after approximately 1700 episodes to 2000 episodes (1750 for the above graphs). We can also see that the algorithm has achieved the objective of reaching the flag in less than 110 timesteps.

Contribution Summary

Team Member	Assignment Part	Contribution (%)
Anup Atul Thakkar	Part 1, Part 2 and Part 3	50%
Pushkaraj Joshi	Part 1, Part 2 and Part 3	50%

Project Management Tool:

Trello Invitation Link- https://trello.com/invite/b/kgLtoKkY/71643945ef3fa820a11d5bba3fc786a0/rl-assignment-2-team-12 (Prof Alina and Nitin have already been added in the beginning of the project)