**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],

}

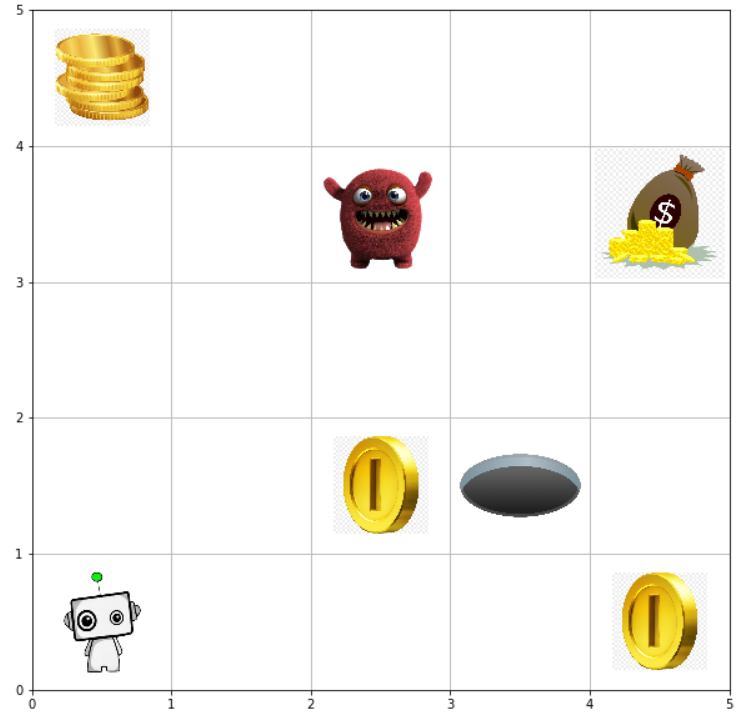
**Actions**

There are four possible actions that the robot can take.

**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).

**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.

1. **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+ ϒ 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.

1. **Q function as q(s, w)**

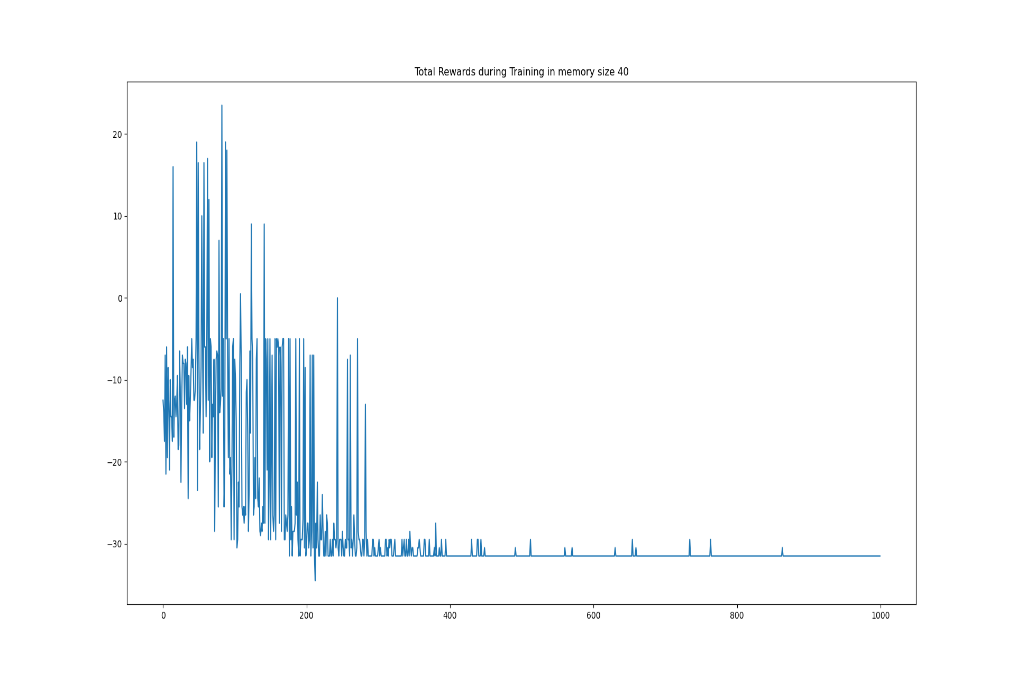
Since we are using neural network for the implementation of the Q learning. The action taken is dependent on the weights of the neural network. The weights are the deciding factor, based on weights the Q values of each action in that current state are determined. The Q learning gradient is the derivative 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**

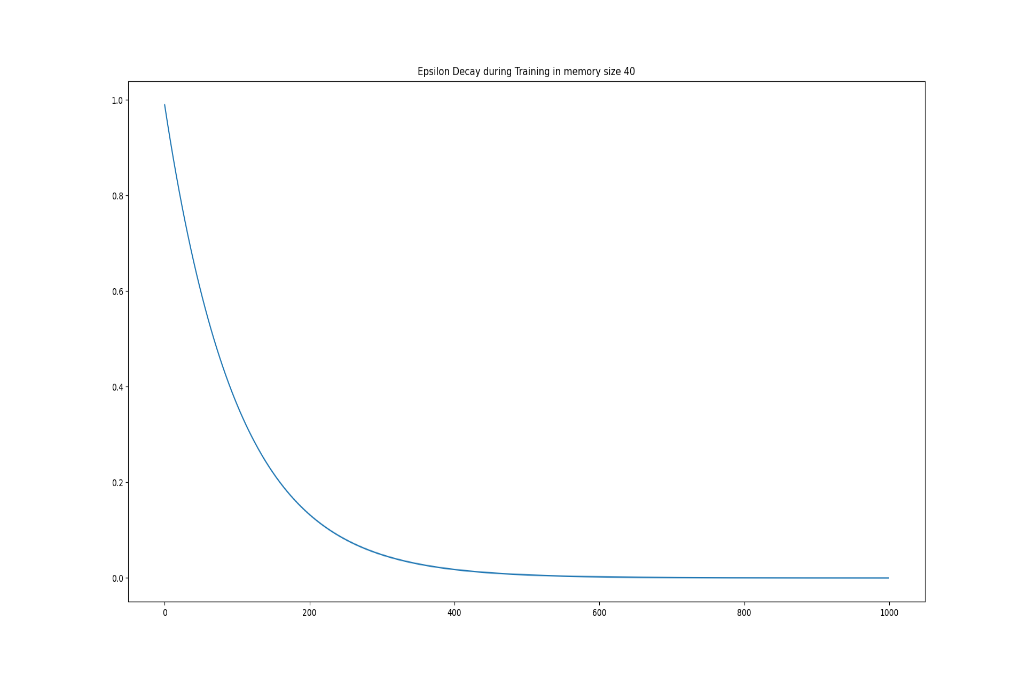
The Robot Grid World was trained using DQN. The agent was trained over 10000 episodes. Two networks were used a policy network and a target network. Both were initialized with random weights in the beginning. Batch size of 40 was used for training of the policy network, and after every 5 timesteps the 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**

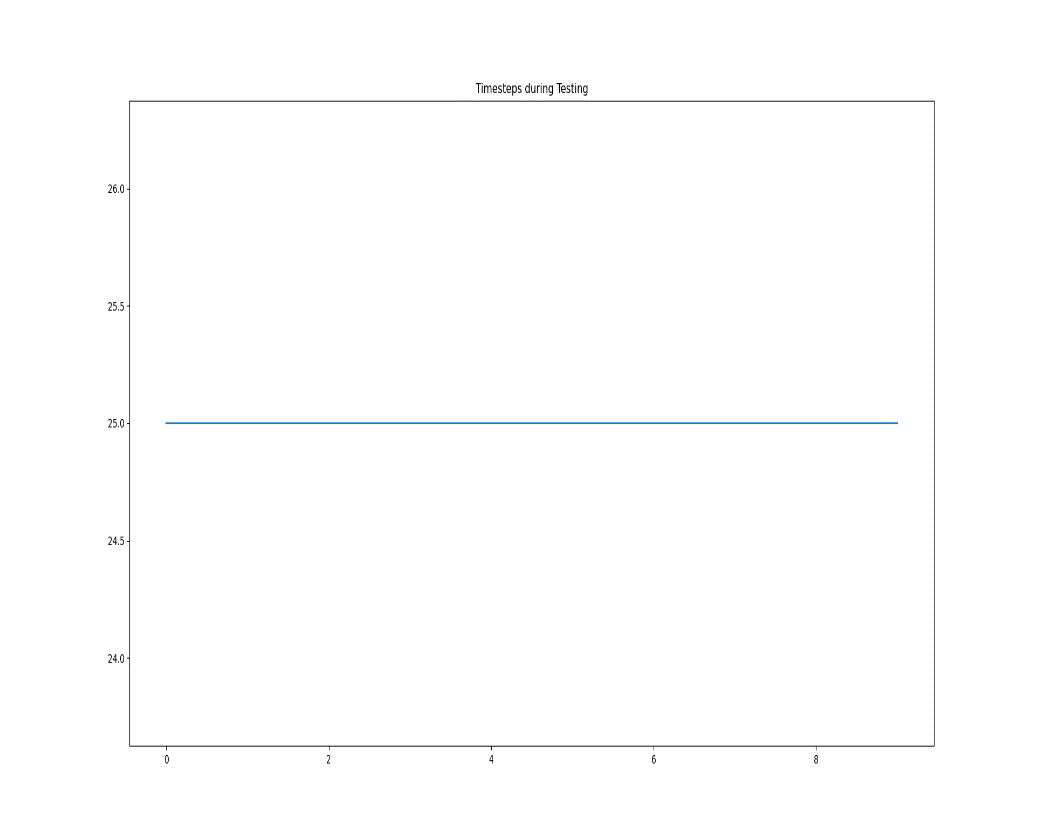
**Training**

** Chart, bar chart, histogram

Description automatically generated**



**Testing**

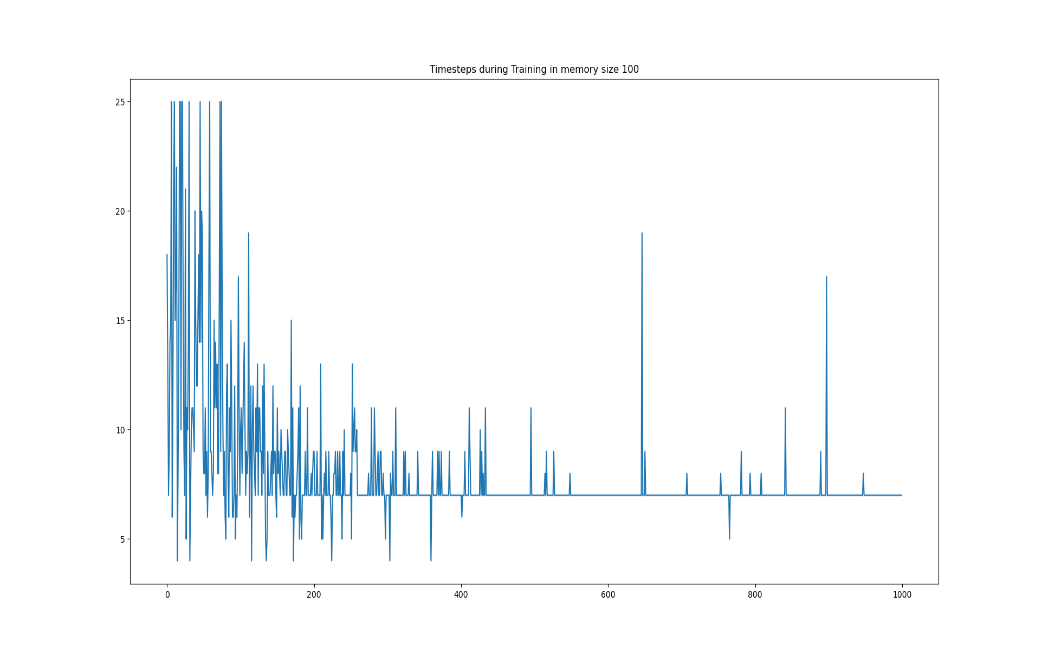
** Chart

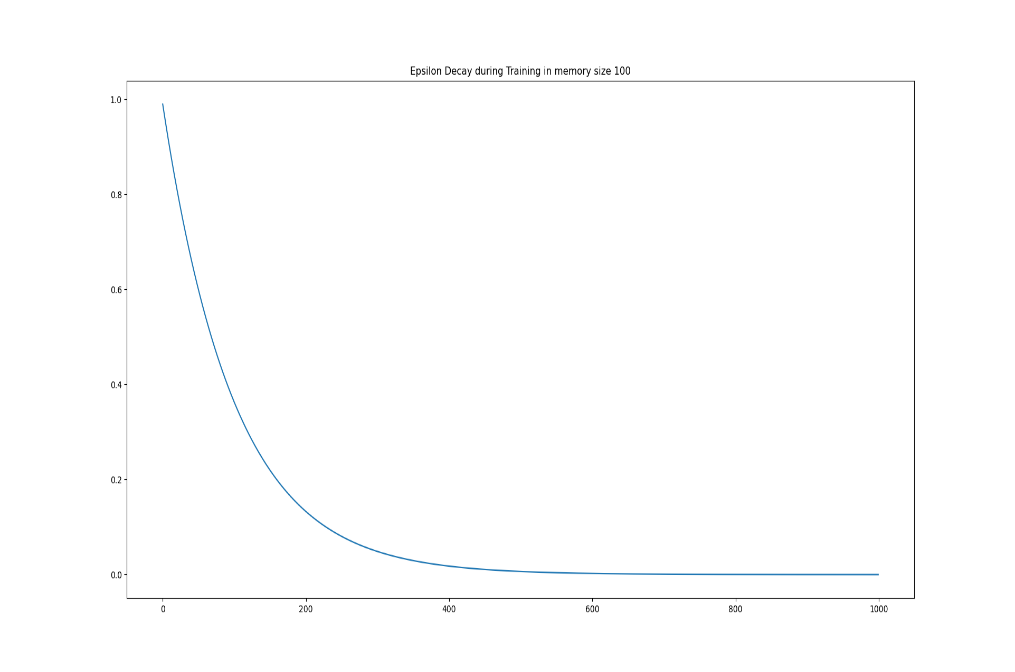
Description automatically generated**

**Replay Memory Size 100**

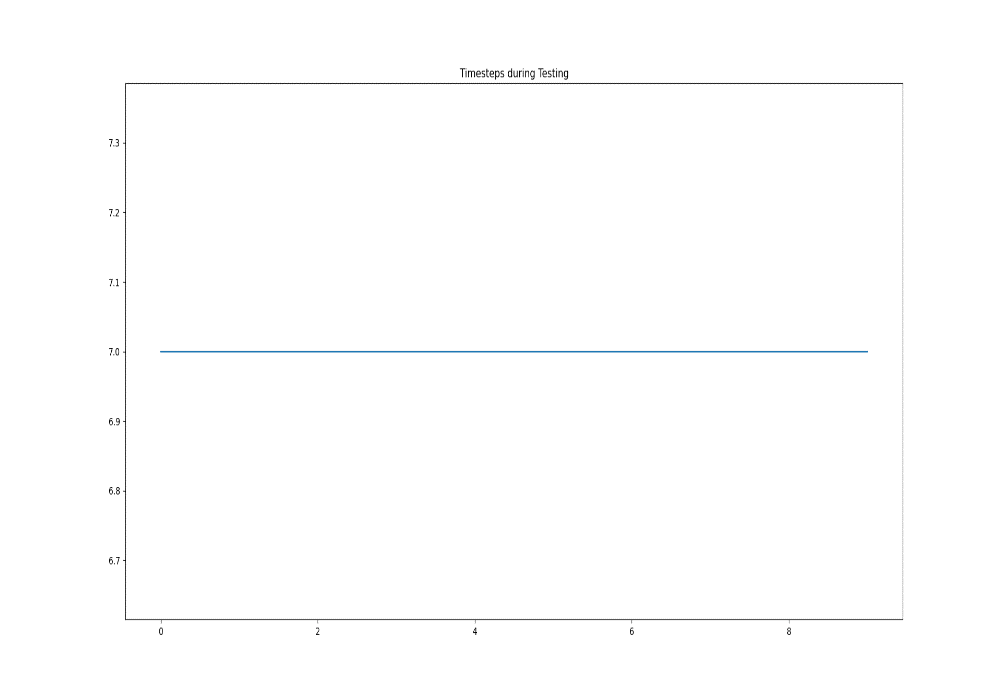
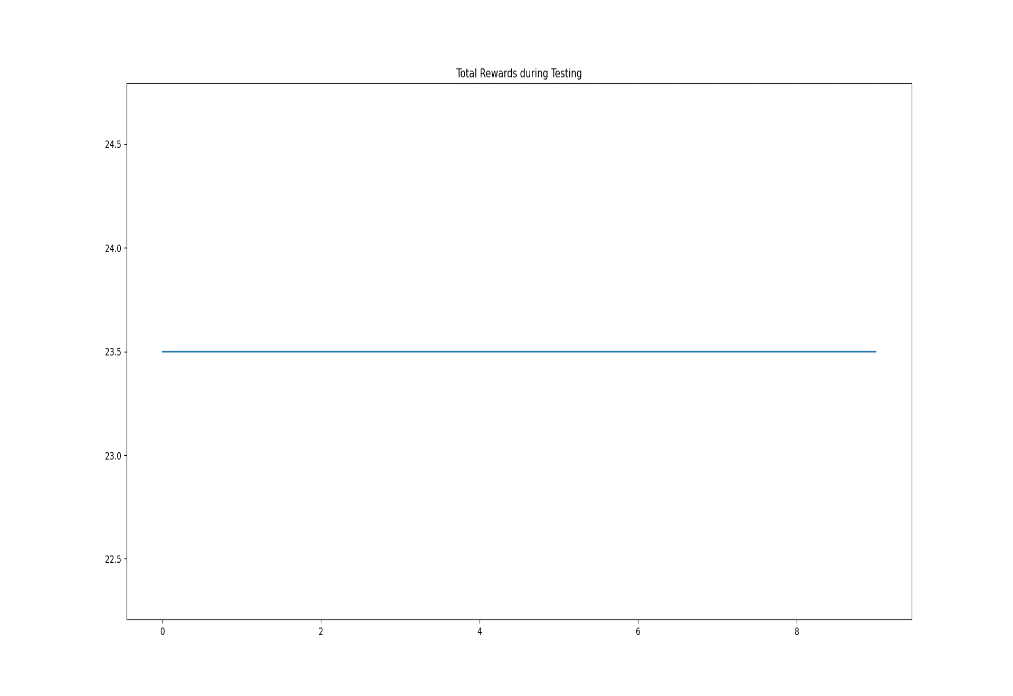
**Training**

Chart

Description automatically generated

****

**Testing**



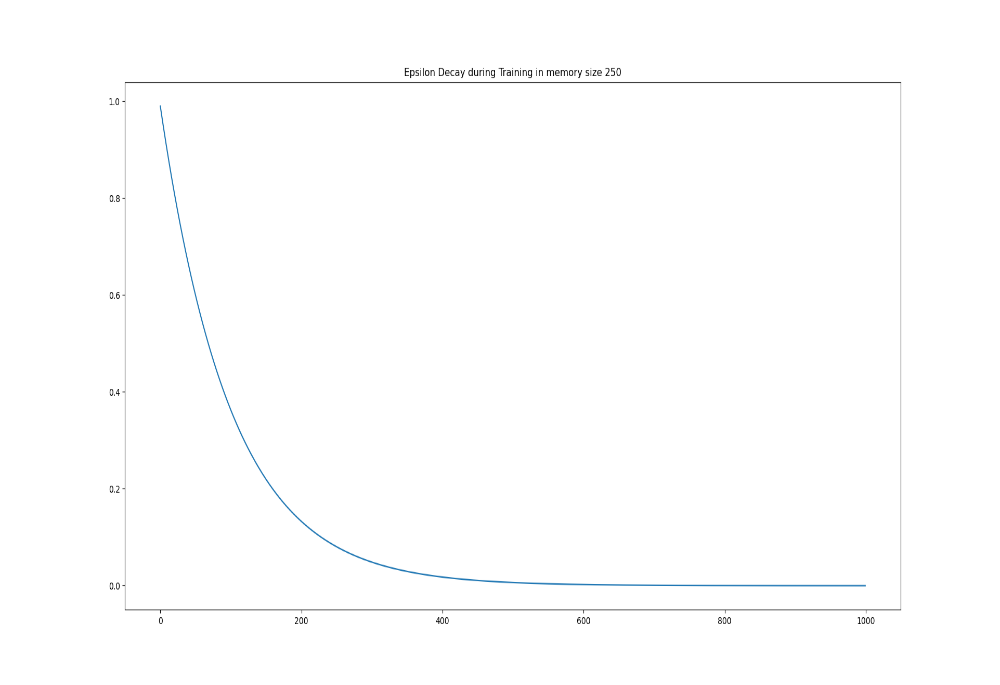
**Replay Memory Size 250**

**Training**

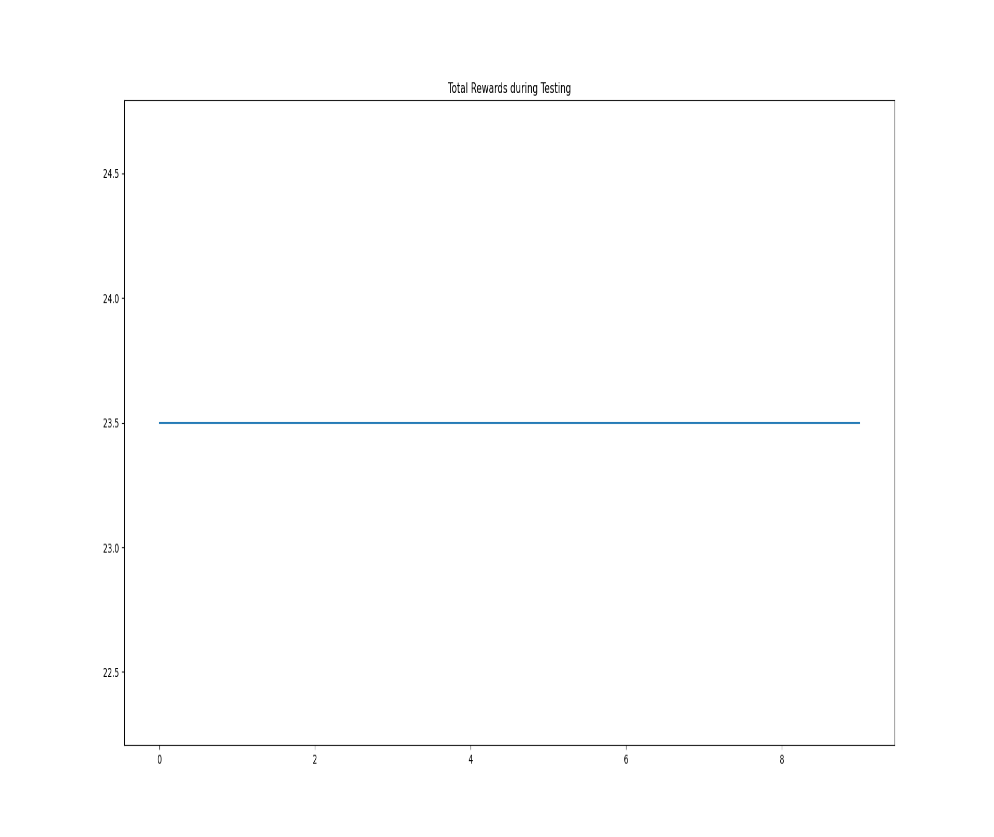
Chart

Description automatically generatedChart, histogram

Description automatically generated



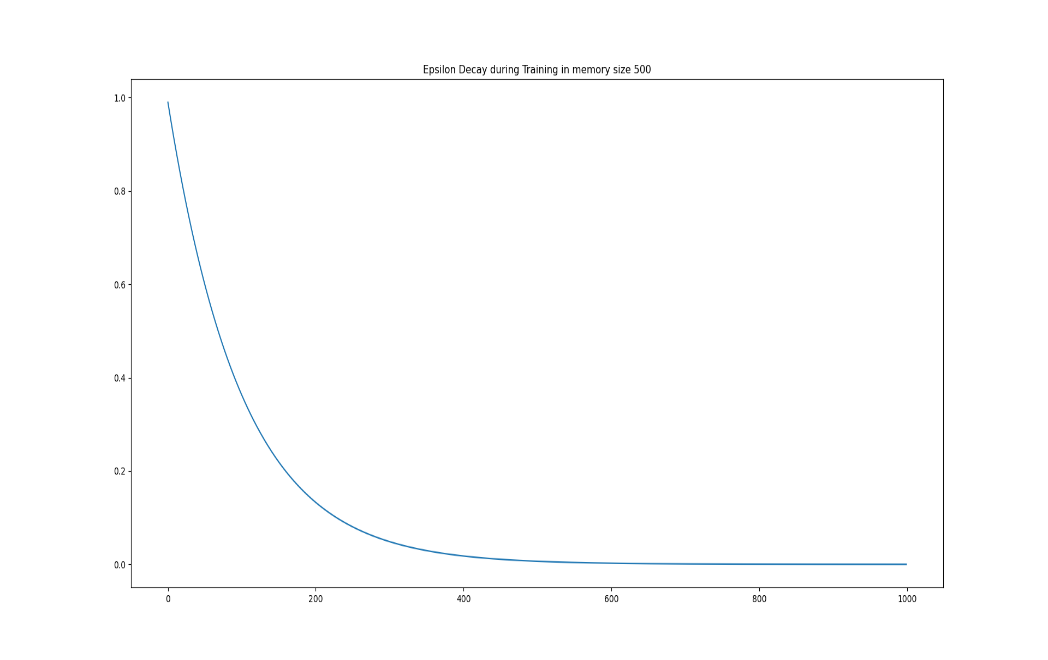
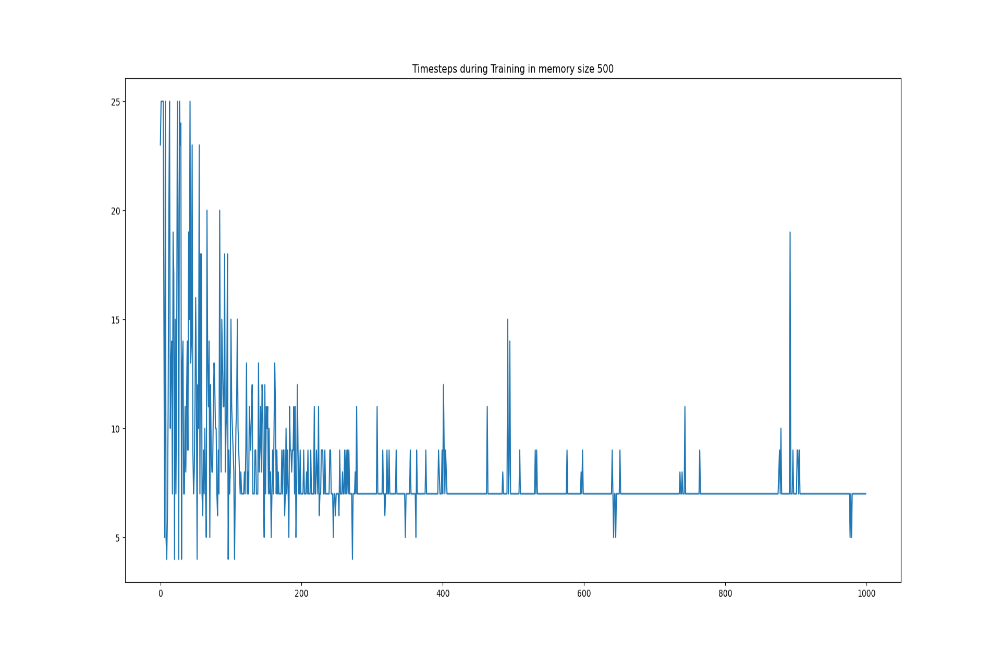
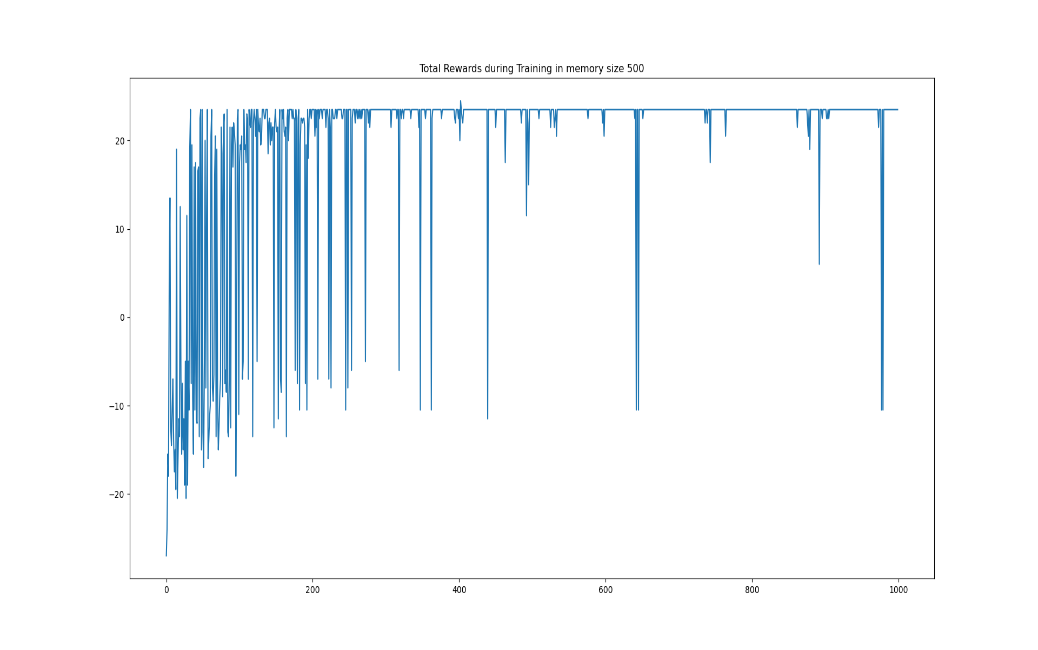
Testing

 Chart

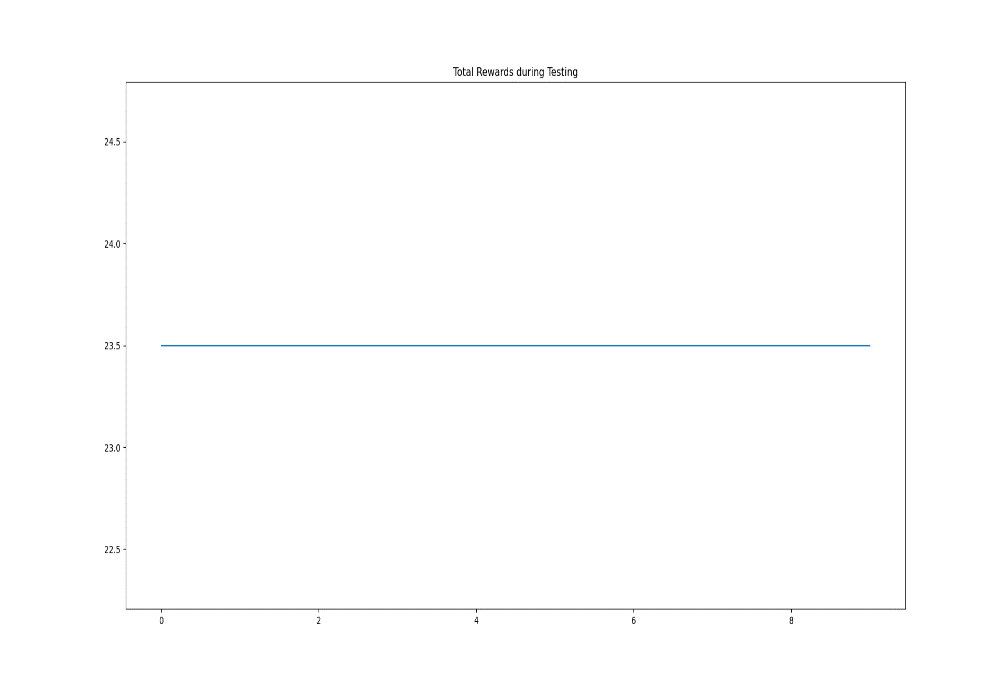
Description automatically generated

**Replay Memory Size 500**

**Training**

****

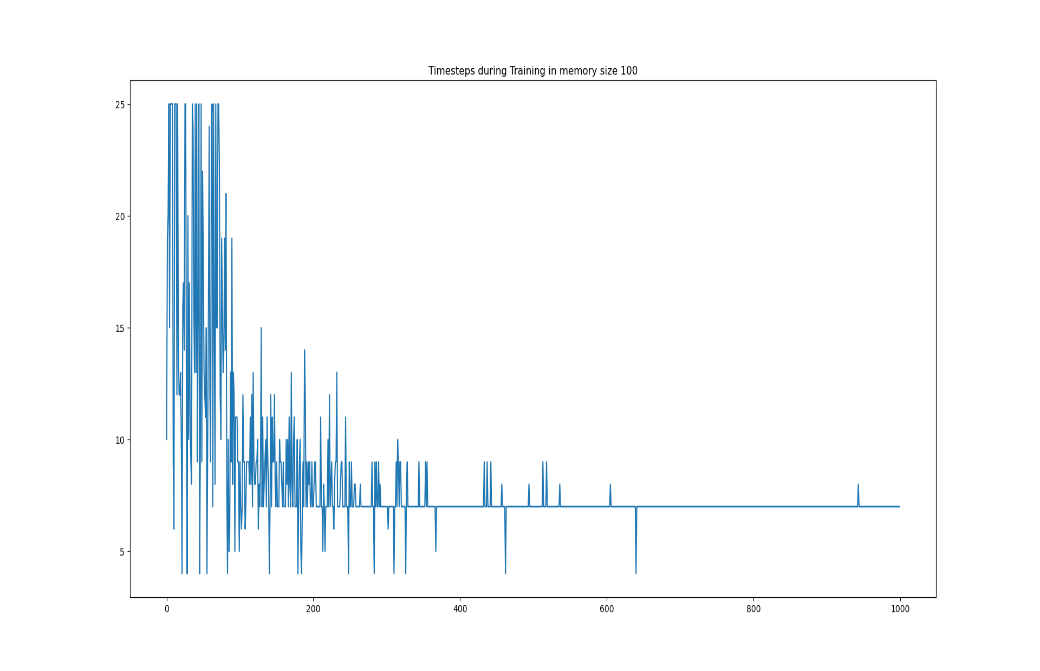
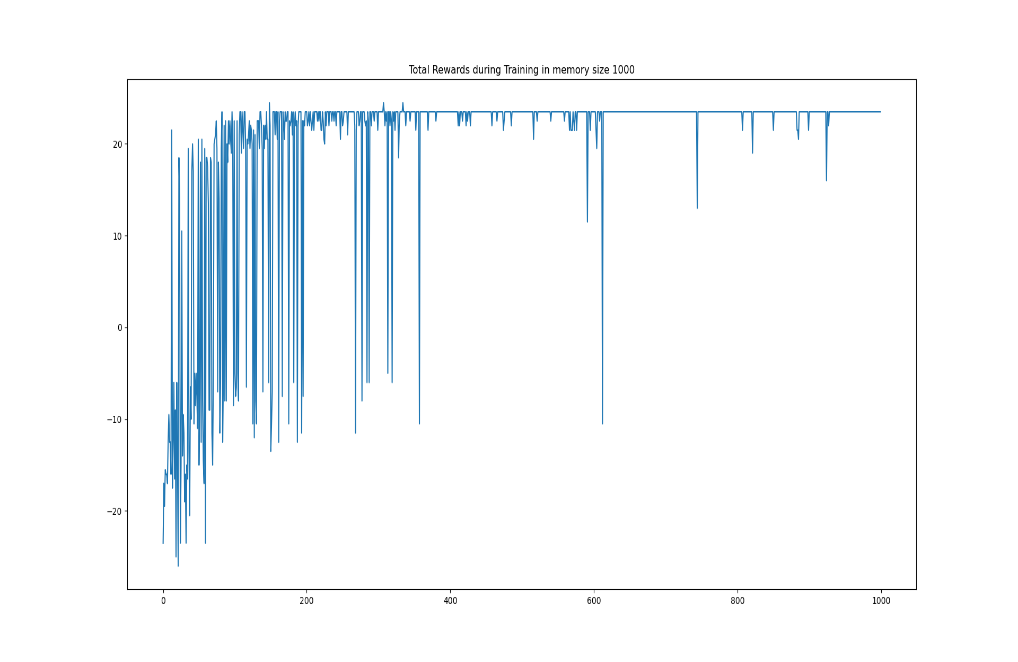
**Testing**

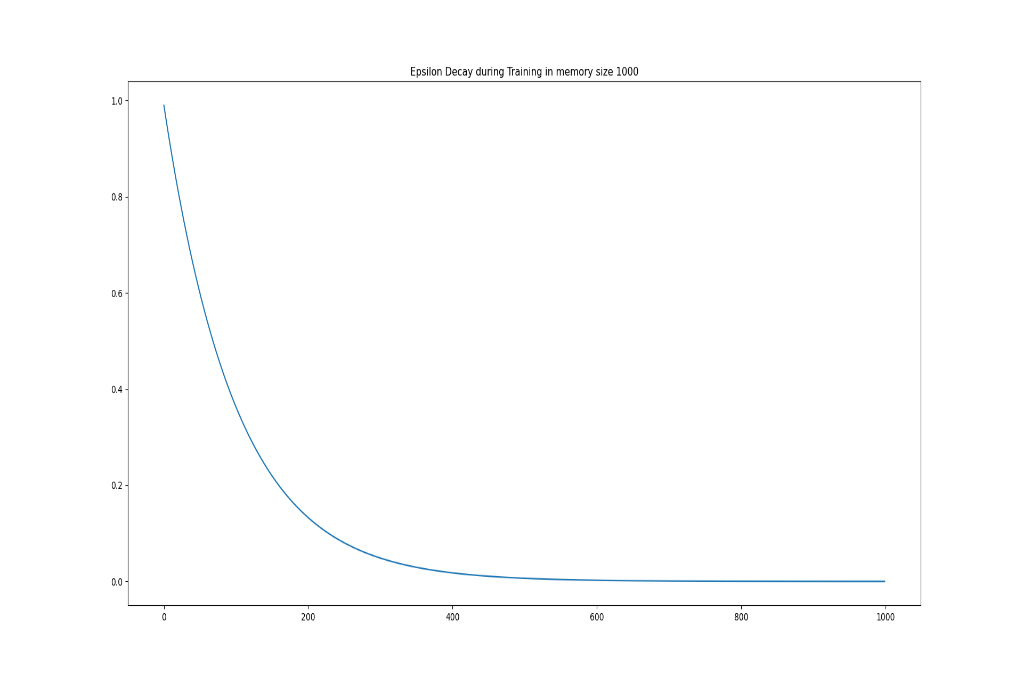
 A picture containing chart

Description automatically generated

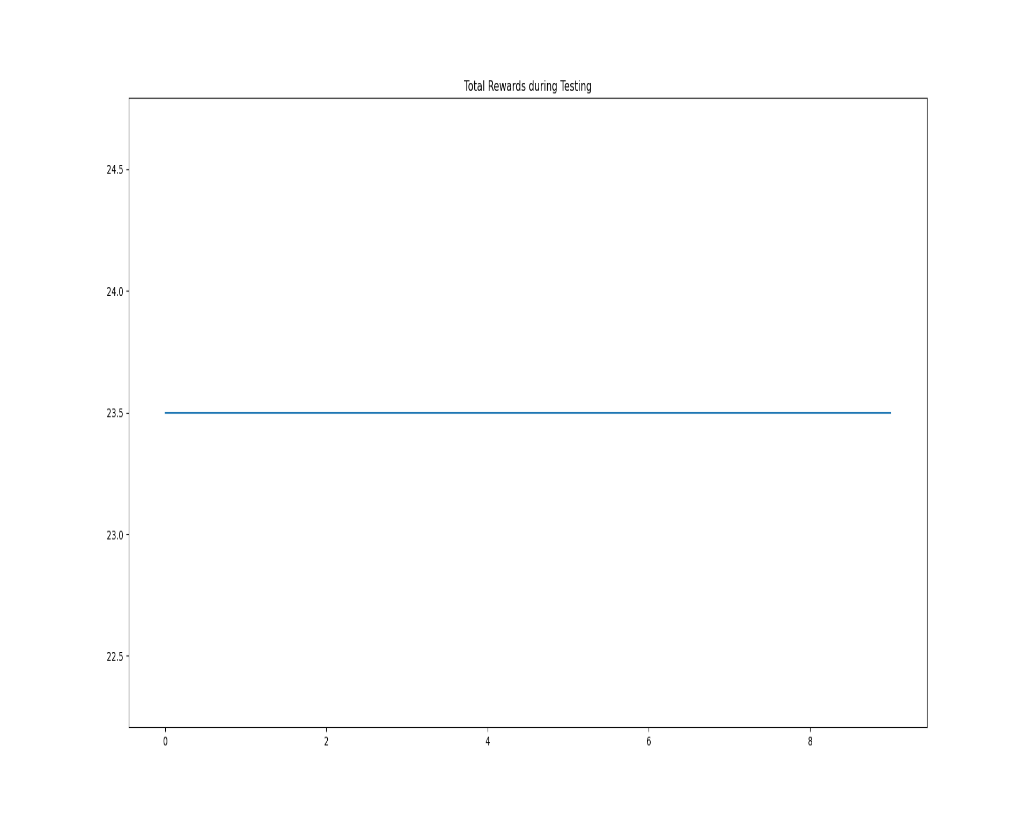
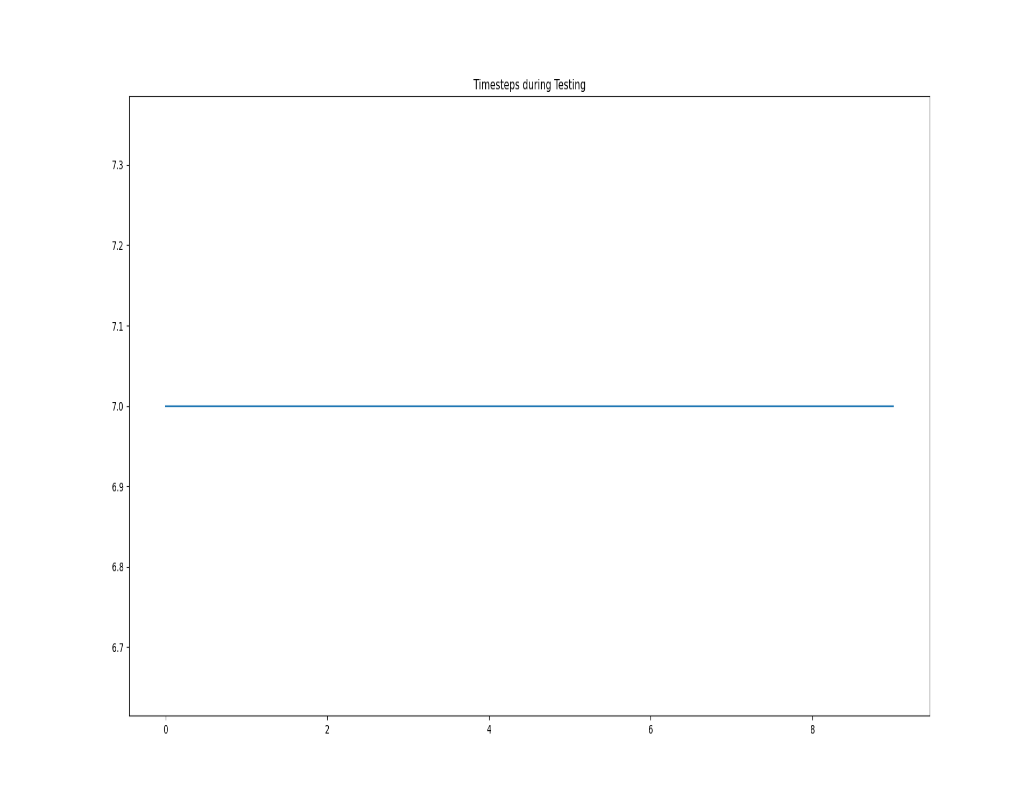
**Replay Memory Size 1000**

**Training**





**Testing**



**Replay Memory Size 10000**

**Training**

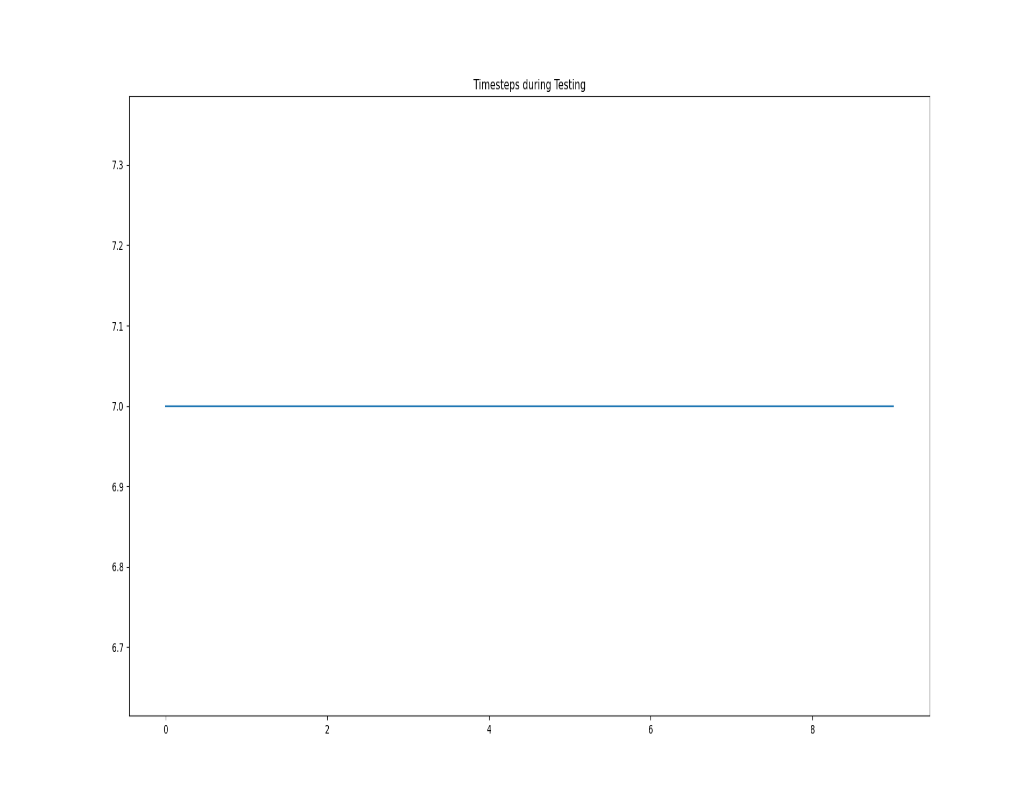
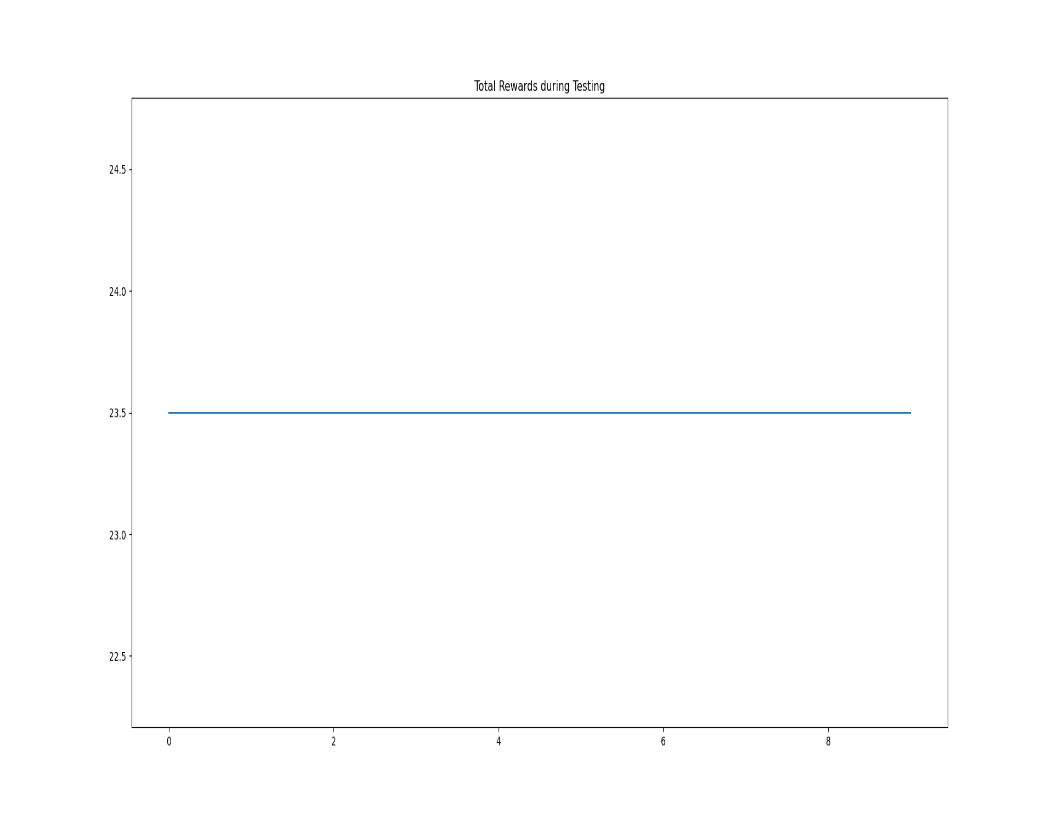
Chart

Description automatically generated Chart, histogram

Description automatically generated

A picture containing graphical user interface

Description automatically generated



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.