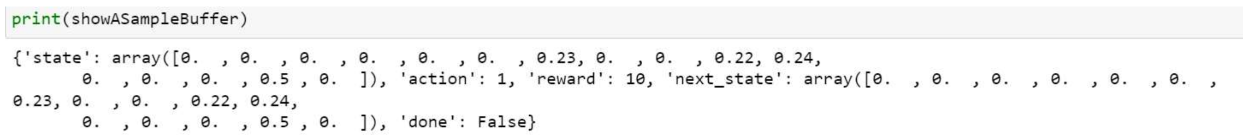
The Double DQN solves the overestimation problems but can be more expensive to train since it uses two different networks, which requires more processing and memory than DQN. It took 10-mins of Google's standard TPU time to just train the network. While DQN is simpler to train and can quickly converge in simpler environments like grid or cart pole environments. The overestimation problem cause the agent to take no optimal actions. The DQN is very sensitive to hyperparameters.

- Using experience replay in DQN and how its size can influence the results

In Deep Q-networks (DQNs), Experience Replay is used to increase the effectiveness and stability of learning. This involves randomly selecting a batch of experiences from the replay buffer to train the agent's neural network after storing the agent's experiences (i.e., observations, actions, rewards, and next states) in the replay buffer. Randomly sampling a batch of experiences from the replay buffer, the agent's neural network can break the correlation between consecutive experiences, which can improve learning efficiency and stability.

I implemented a DQN using torch library on the ‘CartPole-v1’, second complex environment, and grid-world environment. DQN uses a replay buffer to push the agent’s experiences (shown below) from which a batch is sampled out and used to train the agent’s neural network. The results looked much better when using a buffer and experience replay to train the DQN.



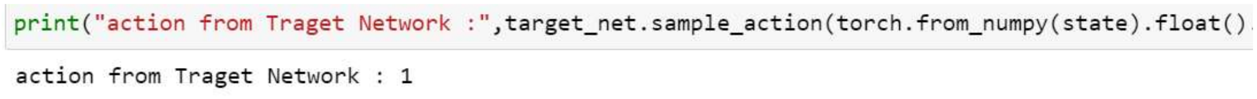
```
print(showASampleBuffer)
```

```
{'state': array([0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.23, 0.  , 0.  , 0.22, 0.24,
       0.  , 0.  , 0.  , 0.5 , 0.  ]), 'action': 1, 'reward': 10, 'next_state': array([0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
0.23, 0.  , 0.  , 0.22, 0.24,
       0.  , 0.  , 0.  , 0.5 , 0.  ]), 'done': False}
```

- Introducing the target network

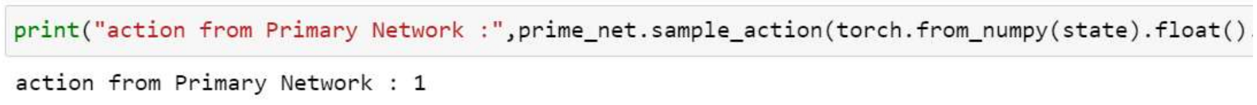
In deep Q-learning, introducing a target network is a technique used to improve the stability and efficiency of learning. The target network is a separate neural network that is used to estimate the Q-values of the next state in the Q-learning update, instead of using the same network that is being updated during training.

In Implemented Models, I used a target network and policy network, Two separate instances of the DQN. The primary network is used to select actions that maximize the expected future reward, while the target network is used to estimate the Q-values of the next state.



```
print("action from Traget Network :",target_net.sample_action(torch.from_numpy(state).float()
```

```
action from Traget Network : 1
```



```
print("action from Primary Network :",prime_net.sample_action(torch.from_numpy(state).float()
```

```
action from Primary Network : 1
```

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

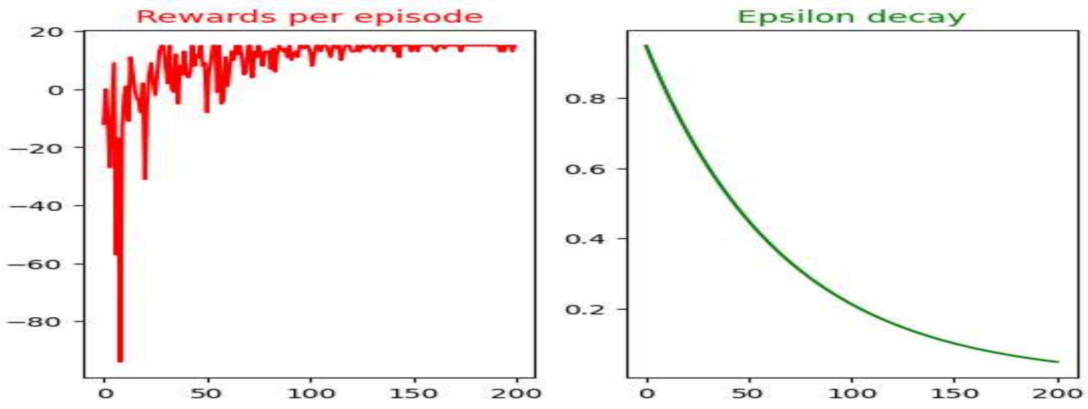
The Q-function (or state-action value function) is a function that maps a state-action pair to the expected cumulative reward. Representing the Q-function in high-dimensional state and action spaces in complex environments with complex state and action spaces. This makes it difficult to represent the Q-function explicitly using a lookup table or other conventional methods. I used a neural network for function approximation.

Q-function is represented as a parameterized function qˆ(s, w), where s is the state, w is the weight vector of the function, and qˆ(s, w) is the estimated value of the state-action pair. We used this parameterized Q-function qˆ(s, w) as an instance of the DQN class in the implemented environments.

To Briefly describe ‘CartPole-v1’, the second complex environment, and the grid-world environment that I used (e.g. possible actions, states, agents, goals, rewards, etc).

Grid Environment:

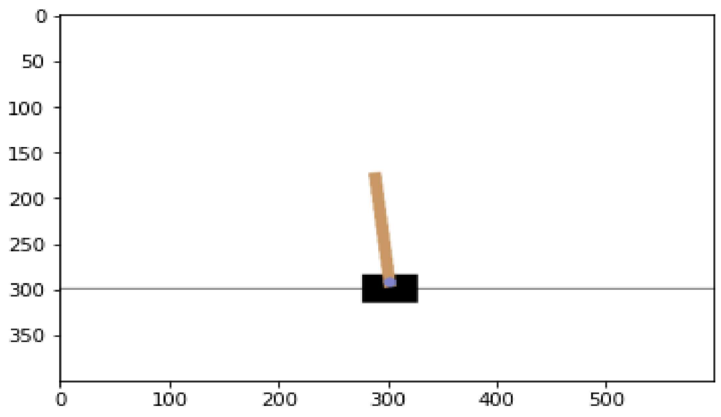
Gridworld is a simple environment that can be used for testing different algorithms. The environment consists of a two-dimensional grid with cells, where each cell can be either empty, an reward/monster, and a goal state or a terminal state. The agent is located in a cell (0,0). can take 4 possible actions at a particular state, move up, down, left, or right to neighboring cells. The goal is to reach the goal cell from the starting cell while collecting the highest reward possible.



The state of the environment is represented by the position of the agent in the grid, and the actions are the four possible directions the agent can move. The reward for moving into a cell is -1, except for when the agent reaches the goal, in which case the reward is {+Reward}. If the agent tries to move into one of these reward states like [1,2], [2,1], and [2,2], The agent bags a reward. Everywhere else there is death and destruction (a monster with a {-3 negative / or worse, Death}.

The Gridworld environment can be configured to have different sizes, shapes, and numbers of rewards/monsters and goals, making it a good testing environment for the implemented DQN Algorithm using torch libraries. I used the DQN on the grid world and plotted the results

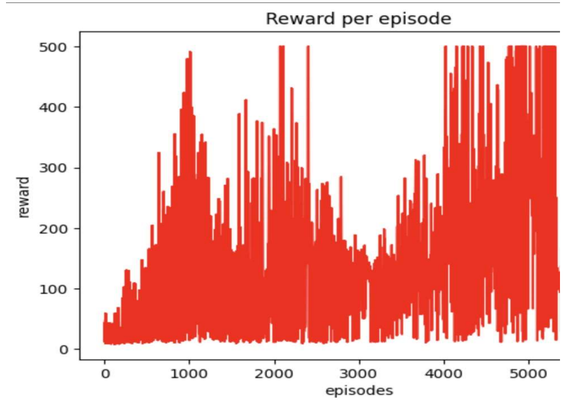
CartPole-v1

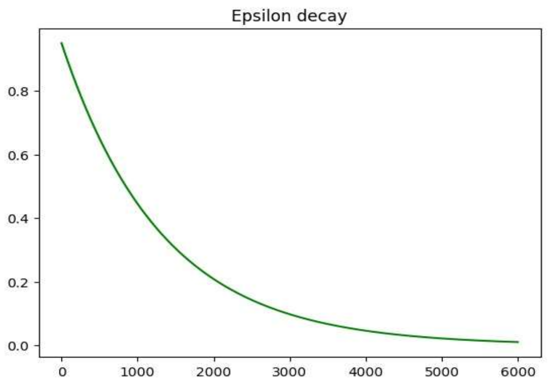


The main goal of the environment is to balance a pole on top of a cart by moving the cart left or right. A solved state is when the pole remains upright. The environment is considered “Done” if the pole remains upright for a specified period of time or the pole has moved to an angle out of reach.

[The position and velocity of the cart, The angle and angular velocity of the pole] make the states of the Cart Pole Environment, and actions are to move the cart either to the left or the right (binary). For each time step that the pole remains upright, a reward of +1 is added to the list to plot the graphs. If the Pole drops below an angle, done (Jai Ballayya).

We have used the Implemented DQN on the Cartpole-v1 environment and the DQN solved the environment as defined by the gym as the average reward over the number of episodes. The results are plotted against the Rewards per Episode and Epislon decay as shown below.





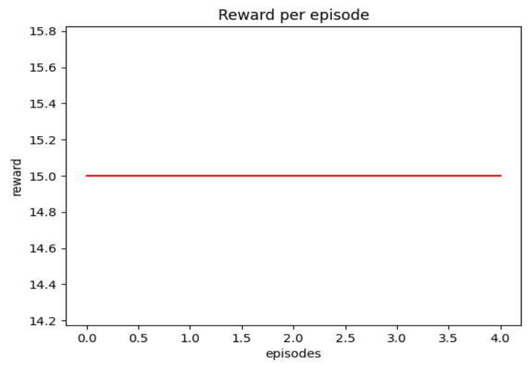
Lunar Lander

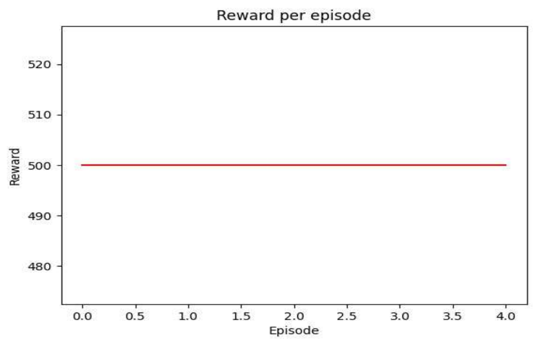
The Moon Landing Environment is a somewhat complex OpenAI gym environment. An agent (a spacecraft) must learn how to safely land on the moon. The environment allows the agent to manage the spacecraft's main engine, side engines, and thrusters as well as receive sensor readings including altitude, speed, and direction.

The agent, in our case- A Lunar Lander, has actions as a 2-dimensional vector representing the amount of thrust to apply to the main engine and side engines. The goal of the environment is to get 200 points. We have the trained DQN Agent on Lunar Landing Environment and plotted Rewards per Episode and Epsilon decay.

Agent on the environments for 5 episodes, the agent chooses only greedy* actions from the learned policy.

1) GridWorld environment ____________ Cart-Pole





.

++ Update CartPole and Lunar Lander environment using the Double DQN Algorithm. ***Algorithm Implemented.***

The Double DQN algorithm implemented uses an Experience Replay, which is used to increase the effectiveness and stability of learning. This involves randomly selecting a batch of experiences from the replay buffer to train the agent's neural network after storing the agent's experiences (i.e., observations, actions, rewards, and next states) in the replay buffer. Randomly sampling a batch of experiences from the replay buffer, the agent's neural network can break the correlation between consecutive experiences, which can improve learning efficiency and stability.

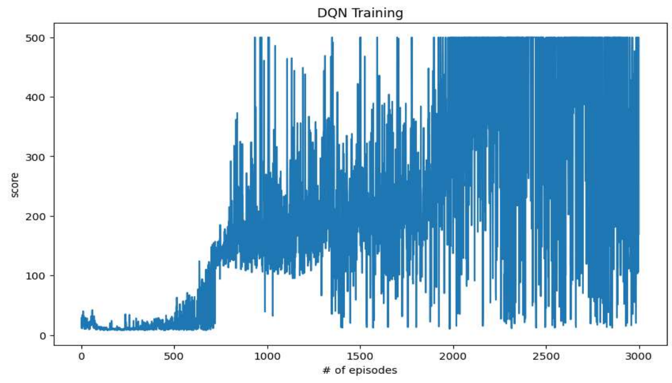
We implemented Double DQN using torch library on the 'CartPole-v1', The Lunar Lander V-2 environment, and the Grid-world environment. The target network is used for calculating the Q-value of the action selected by the primary network in the next state. This will solve the overestimation problem in calculating Q-values that can occur in standard DQN implementation in RL envs.

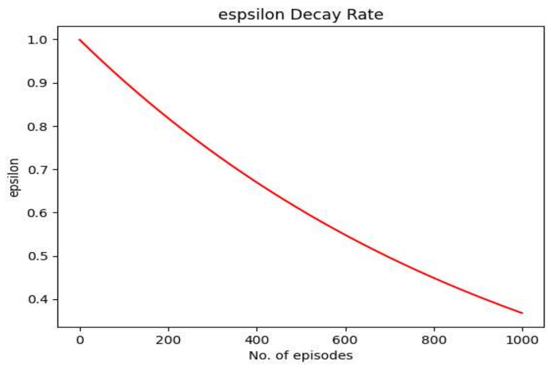
An improvement over the vanilla DQN

Double DQN addresses the issue of overestimation of Q Values which results in more precise value estimates and better performance. When compared to DQN, double DQN is more trainable and less sensitive to the selection of hyperparameters with a cost of more computational requirements. The Double DQN uses two separate networks to select and evaluate the action and update the Q network's weights. We used the Neural Network that we built using the Pytorch libraries to implement the improved version of vanilla DQN.

Double DQN

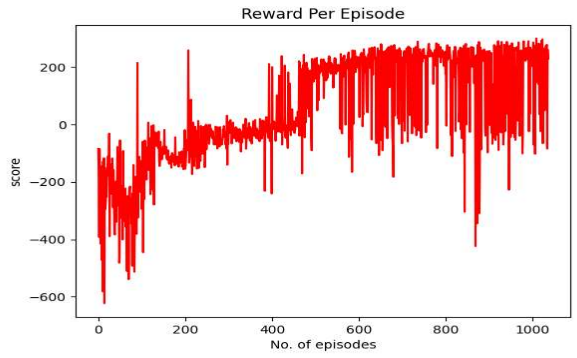
CartPole-v1:

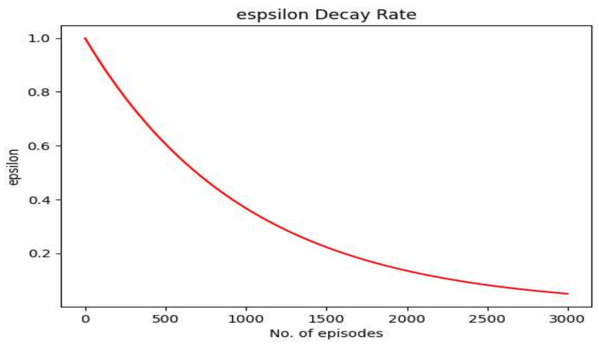




Lunar Lander -V2:

Double Deep Q-Learning: The lunar lander is a box-2d environment, Where the agent, The lander must land on the moon. It's a complex environment to solve with higher state and action space. The **Double-DQN** was able to solve the environment and we can observe the lander achieving 200+ rewards and being able to maintain the moving average. The results are plotted below as reward per episode and epsilon decay.

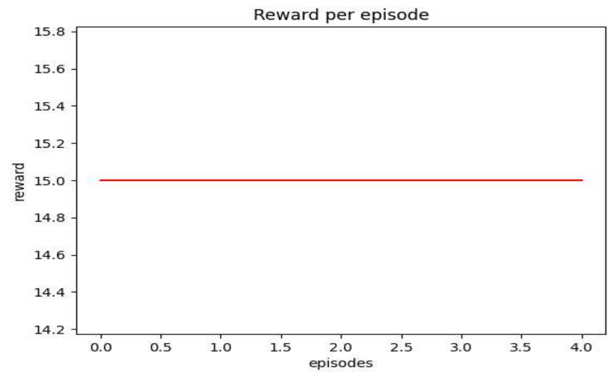




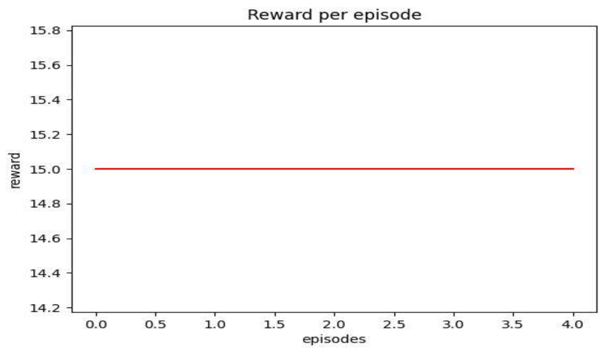
Environment run for 5 episodes, *where the agent chooses only greedy actions from the learned policy.*

GridWorld

Double DQN

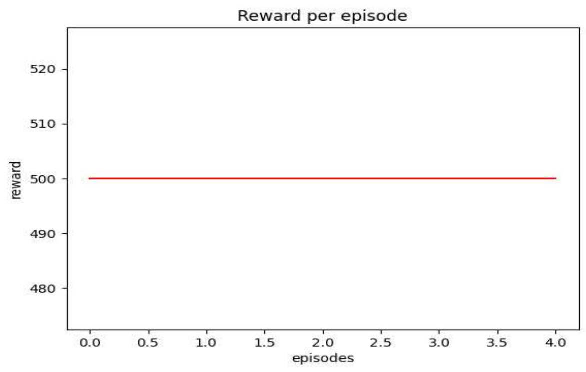


DQN

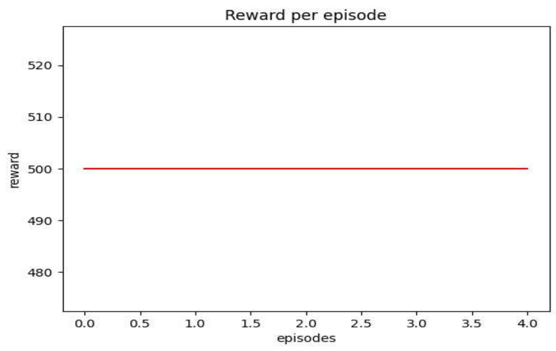


CartPole-v1

Double DQN

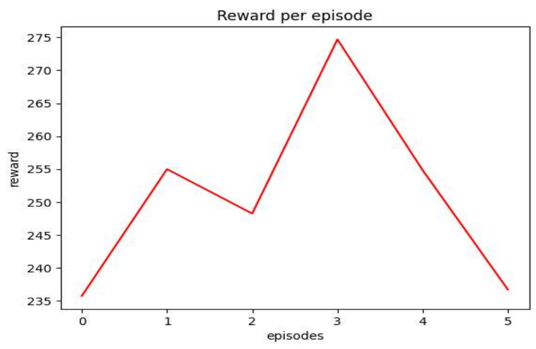


DQN

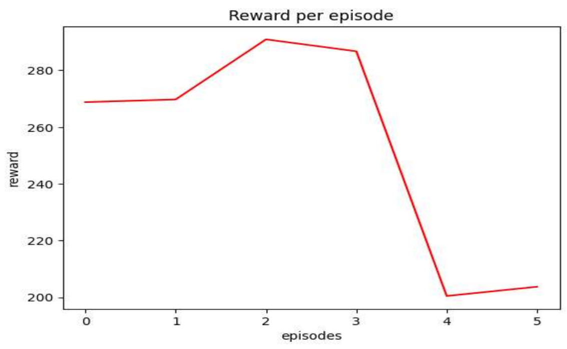


LunarLander-v2

Double DQN



DQN

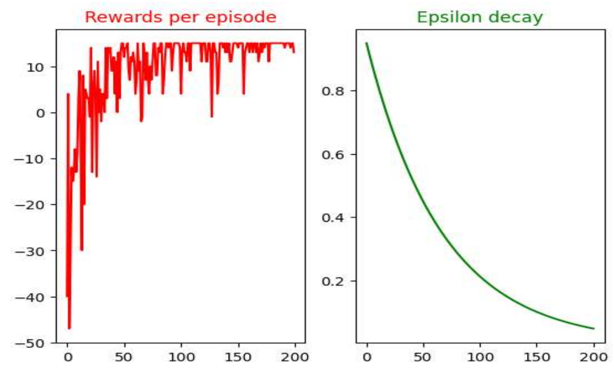


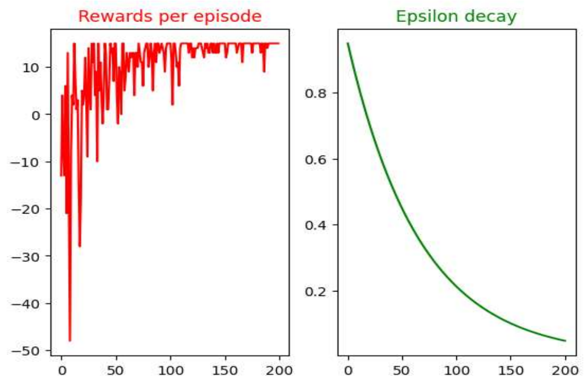
Performance on the same environments with rewards plots & results from algorithms applied:

- ***Grid-world environment***
- ***'CartPole-v1'***
- ***'LunarLander-v2'***

Achieved optimal rewards in the defined grid world environment. The Double DQN took the path of exploring more and gradually started to exploit the q-network. DQN followed a similar path in this environment.

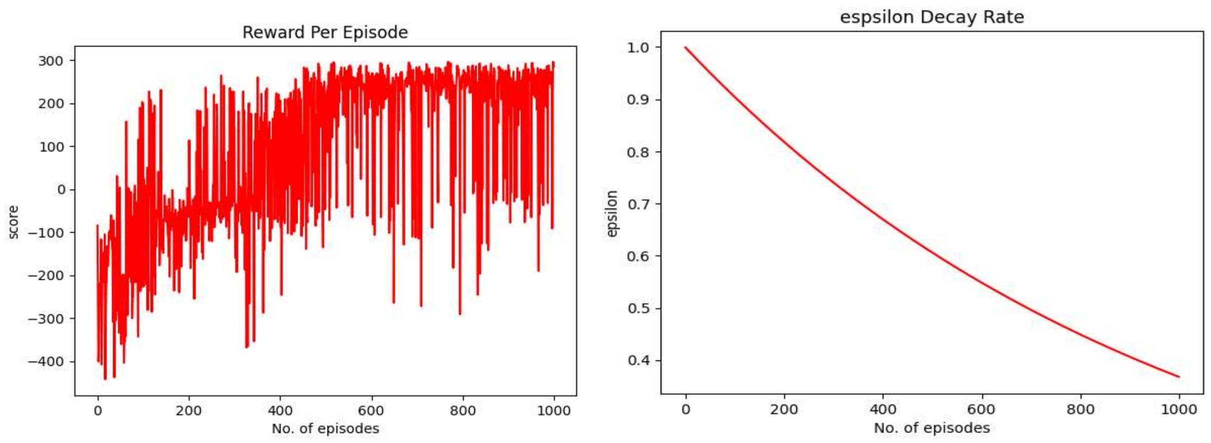
Double DQN **DQN**



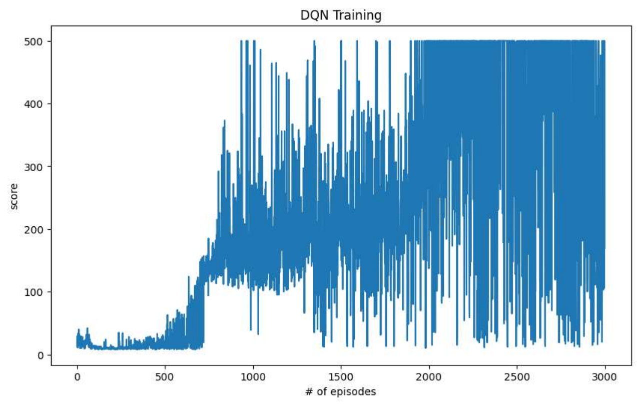


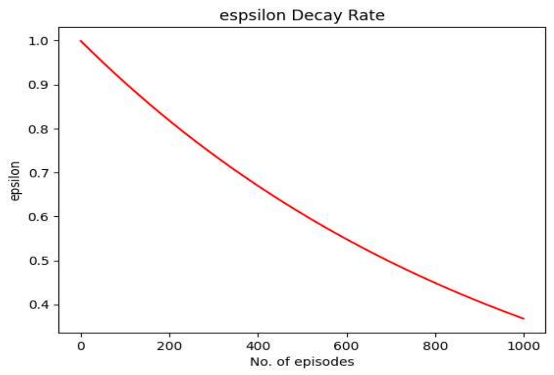
CartPole-v1:
The DQN started with more negative regard and slowed learning the optimal policy from the network while the Double DQn used the exploration more for the first many episodes and slowly used the q network to achieve the convergence and solve the environment. Overall Bothe algorithms were able to solve.

DQN



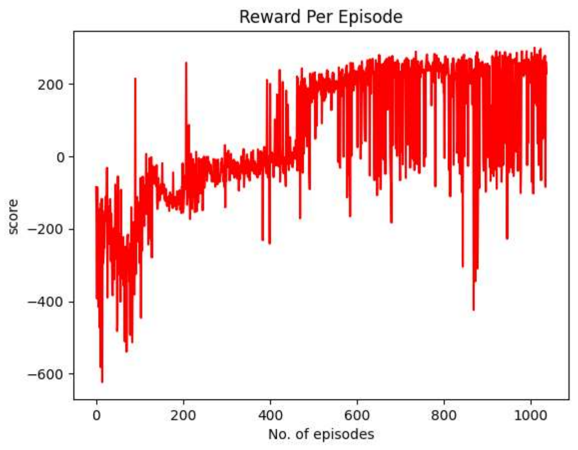
Double DQN:

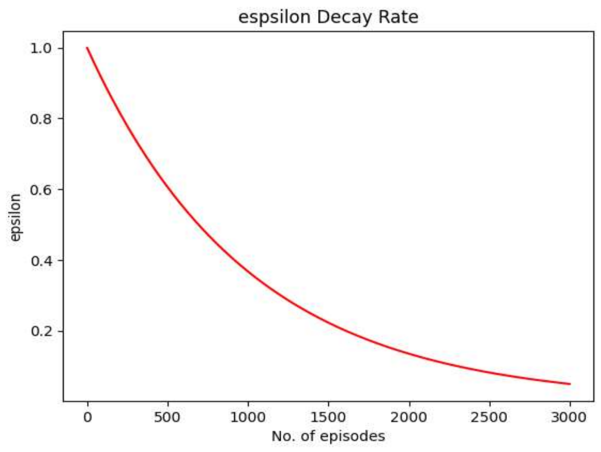




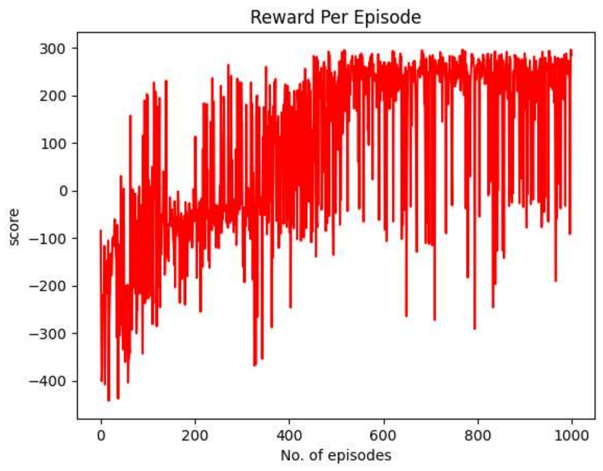
Lunar Lander -V2:
We have solved the Lunar Lander environment, using Deep Q-Network (DQN). However, a Double Q-Network (DQN) that solves the overestimation issue in Q-learning can be used to further boost its performance. The Double DQN was able to solve the environment and was able to learn the optical path.

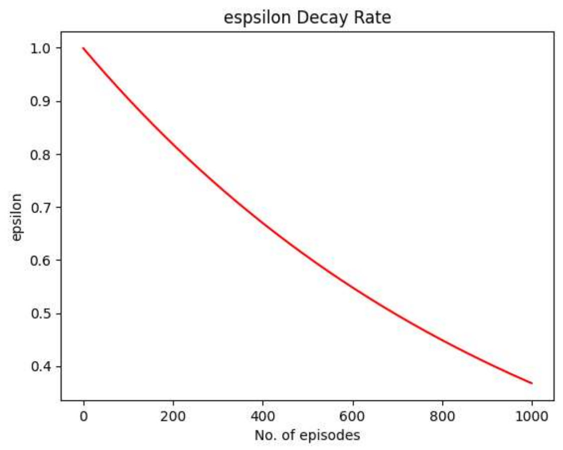
Double-DQN





DQN


Reward Per Episode
score
No. of episodes


espsilon Decay Rate
epsilon
No. of episodes