Applying DQN, DDQN on Different Environments

Nandini Chinta
SUNY Buffalo, NY
nandinic@buffalo.edu

Sri Amarnath Mutyala
SUNY Buffalo, NY
sriamarn@buffalo.edu

Abstract

This project deals with applying the Deep Q-Network, Double Deep Q-Network on a 5*5 custom grid environment, OpenAI gym environment Cartpole-v1 and LunarLander-v2. We successfully coded and applied this algorithm to learn these environments. We compare the performance of vanilla DQN vs the improved Double DQN.

1. Deep Q-Network

In the deep Q-Network we try to approximate the Q-values in the traditional Q-Learning by using a fully connected neural network. Deep Q-Networks are deep learning neural networks which are used to make the agent learn to directly from the high-dimensional sensory inputs.

a. Experience Replay

Experience replay is memorizing the state action transitions and training the model using mini batches of these transitions to predict the Q value. It is storing the past experiences and using them to train the model. In other words, here in experience replay, we use multiple transitions to update the Q-table instead of using only one transition.

Uses:

- Since the data is independent and identically distributed and the sequence of actions can be highly correlated which were used in Q-Learning, Experience Replay in DQN helps us break that correlation and learn mostly from independent tuples of experiences.
- Since we train the model with mini batches, it helps to speed up the training process.
- The size of the experience replay, buffer size is also related to the performance of the model as the small buffer size correlates highly. But also using large buffer size may decline the performance.

b. Target Network

Target Network is using two Q networks, where Q values are to be considered from one Q network and the updating of Q table is involved with the other Q network. First Q network will be updated according to the second one after given epochs.

Uses:

- Since we use two Q networks, target parameters are fixed.
- Loss between Q networks and the targets will be optimized.
- It stabilizes learning for the agent.

Q(s,w) helps us add in a non-linear approximator to approximate the expected value of a state s. It prevents the need to compute Q values explicitly for each state, action pair which is not possible in large models. With Q(s,w) it is possible to model problems with large state space and also helps us achieve human like performance on unconventional input like images.

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-)$$

2. Double Deep Q-Network

Deepmind team noticed that while DQN performs well in solving a task it tends to overestimate the Q-value. To counter this it was proposed to use the target network for choosing the action and the online Q-network to handle the Q-estimation. This way the max operator which causes the overestimation bias is removed from the Q estimation.

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

2.1. Algorithms

In this project, we have used DQN and Double DQN algorithms.

DQN:

DQN is similar to Q -learning except that in Q-Learning we use Q-table to choose the greedy action where as in DQN we use a Neural Network to do the job. NN implemented here estimates the q-value for each state provided with the training of previous state, action, reward, next_state pairs to the network. NN takes a state or an observation as input and the number of actions as an output.

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a \right]$$

We use below algorithm for DQN:

```
**Algorithm 1** Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  **for** episode $= 1, M$ **do**
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      **for** $t = 1, T$ **do**
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
      **end for**
  **end for**
```

Double DQN:

Double DQN is an improvised version of DQN where we use two networks, one to choose the action and the other to evaluate the q-values. We use online network to choose the action and the target network to estimate the Q-value.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\ D}\left(r + \gamma Q(s', \arg\max_{a'} Q(s',a';\theta);\theta_i^-) - Q(s,a;\theta_i)\right)^2$$

As an improvement, we have implemented Experience Replay to both DQN and Double DQN. Experience Replay is we store the transitions (state, action, reward, next_state, done) in a memory, provided the size and use this memory to train the neural network, We periodically fetch a sample from the memory and feed it to the network so that the model learns all possibilities in the environment.

```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
     With probability $\varepsilon$ select a random action $a_t$
     otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
     Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
     Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
     Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
     Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
     Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
     Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
     network parameters $\theta$
     Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

2. Environments

a) Cartpole-v1

Cart-pole system with continuous actions. We have an unactuated joint attached to a cart which can go linearly on a 1-D frictionless track. Our goal is to keep the pole upright with a degree of less than 12 from the vertical and the cart should not move beyond 2.4.

Source:

This environment of the cart-pole problem was first described by Barto, Anderson, and Sutton.

Observation:

The state of the environment consists of the cart's x-position, cart's velocity, pole's angle in radians, and the angular velocity in rads/sec.

No.	Observation	Minimum	Maximum
1	Cart's Position	-4.8	4.8
2	Cart's Velocity	-Infinity	Infinity

3	Pole Angle	-0.418 rad(-24 deg)	0.418 rad(24 deg)
4	Pole Angular Velocity	-Infinity	Infinity

Actions:
{0,1}. 0 action takes the cart to the left and 1 takes it to the right.

Reward:

The reward is -1 for every step taken, and 0 if the cart-pole system is balanced. We want to motivate the agent to quickly reach the balancing.

Starting State:

At the start, the cart is upright, and the pole lies at 20 degrees and has no velocity.

Episode Termination:

Ends in Failure:

pole angle > 12 degrees cart position > 2.4 (out of bounds) episdoe length > 200

Solved:

average return > 195 over 100 consecutive trials.

b) LunarLander-v2

This environment explores the simulation of landing a space craft on the lunar surface. There are three engines on the bottom, both the sides of the spacecraft. The environment has discrete actions: engine on or off. There are two environment versions: on or off.
The landing pad is at the origin (0,0). The state vector is the co-ordinate position of the spacecraft in that frame. Fuel is infinite but each firing is penalized with a goal to help learn quick landings.

Observation space
There are in total 8 states:

- `x-y`: cartesian coordinates of the lander
- `Vx-Vy`: Linear Velocities of the lander
- `theta`: Orientation angle of the lander
- 'Atheta': Angular velocities
- 'Cl,Cr': Booleans that say if lander's left and right legs are in contact with the ground.

Action Space
There are four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine

Rewards
The reward function is defined as:

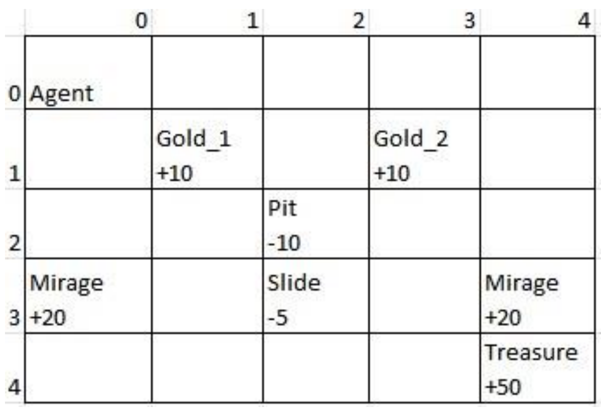
- Top of page to landing: 100-140 points
- If moving away from landing pad, it loses reward
- In case of a crash a reward of -100 is given.
- Each leg in contact with the ground is given +10 reward.
- Firing the main engine is -0.3 points for each frame and the side engines a penalty of -0.03 points.
- Correctly landing on the landing pad would be given a reward of +200 points.

Starting State
The lander starts at the vertical top in the centre.

`Episode Termination`
The episode finishes if:
1) When the lander crashes into the surface.
2) the lander's x co-ordinate is > 1. (out of bounds)
3) the lander is not awake. (i.e. it doesn't move or do anything)

c) GridWorld

Here in this project, the environment is defined as follows: a 5*5 grid world



	0	1	2	3	4
0	Agent				
1		Gold_1 +10		Gold_2 +10	
2			Pit -10		
3	Mirage +20		Slide -5		Mirage +20
4					Treasure +50

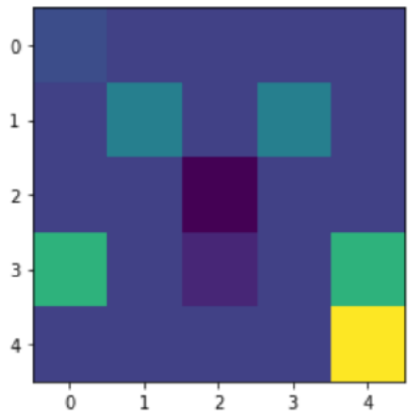
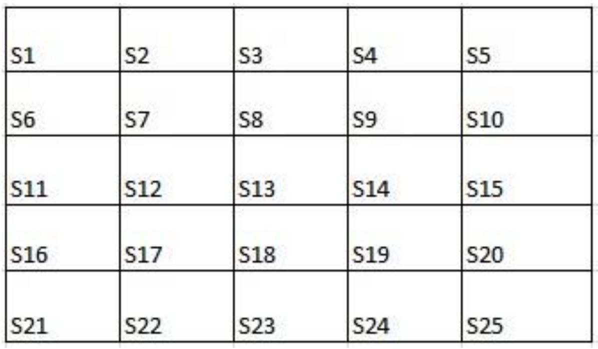


Figure 2.1: Custom Grid Environment designed

Actions: The environment has a set of 12 actions which are: {Right, Left, Up, Down, right up diagonal, right down diagonal, left up diagonal, left down diagonal, jump right, jump left, jump up, jump down}

States: The environment consists of the set of states: {S1, S2, …. S23, S24, S25}



S1	S2	S3	S4	S5
S6	S7	S8	S9	S10
S11	S12	S13	S14	S15
S16	S17	S18	S19	S20
S21	S22	S23	S24	S25

Figure 2.2: States of the custom environment

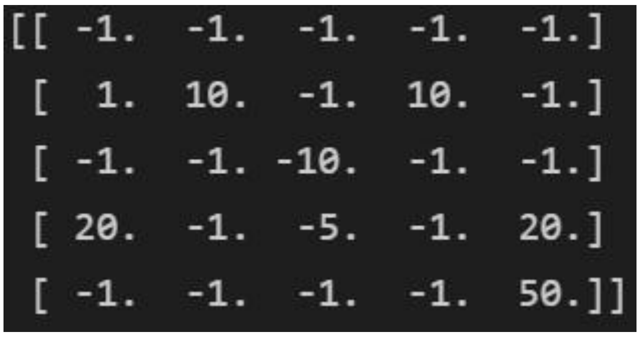
Rewards: There are five rewards in this environment.

Treasure: The final goal point located at [S25] where the agent must reach containing 50 reward points.

Gold 1&2: These are intermediate reward points located at [S7] and [S9] having reward points of 10 each.

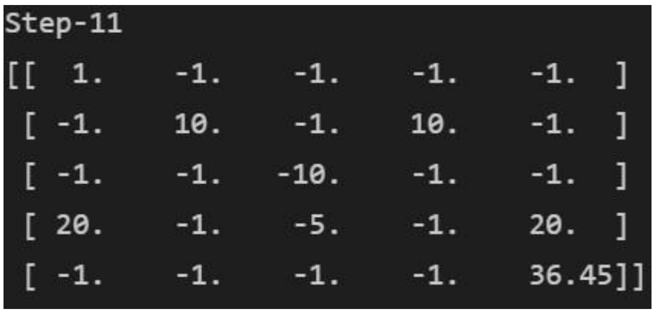
Mirage 1& 2: A special case of intermediary reward points located at [S16] and [S20]. It has a special property that when the agent comes into this location, the rewards agent has will be vanished and will be rewarded with 20 at each mirage location.

If agent comes into this location, say [S20] right above the cell [S25] where the final treasure is located, the treasure reward points will be reduced to 10% of it.



```
[[ -1.  -1.  -1.  -1.  -1.]
 [  1.  10.  -1.  10.  -1.]
 [ -1.  -1. -10.  -1.  -1.]
 [ 20.  -1.  -5.  -1.  20.]
 [ -1.  -1.  -1.  -1.  50.]]
```

Figure 2.3: Reward table of the custom environment



```
Step-11
[[  1.    -1.    -1.    -1.    -1.  ]
 [ -1.    10.    -1.    10.    -1.  ]
 [ -1.    -1.   -10.    -1.    -1.  ]
 [ 20.    -1.    -5.    -1.    20.  ]
 [ -1.    -1.    -1.    -1.    36.45]]
```

Figure 2.4: Reward table after few penalties

Penalties: To balance, the environment was created with two cells where, when agent enters those cells, the agent will be penalized.

Pit: This cell is located at [S13] position. So, if the agent enters this cell, the agent will be penalized with -10 points and the agent will be moved to a random position between the cells located at first 2*2 subgrid except the cell which has pit in it.

For example, if agent falls into the pit located at S13, it may end up in the states {S1, S2, S3, S6, S7, S8, S11, S12, S13}.

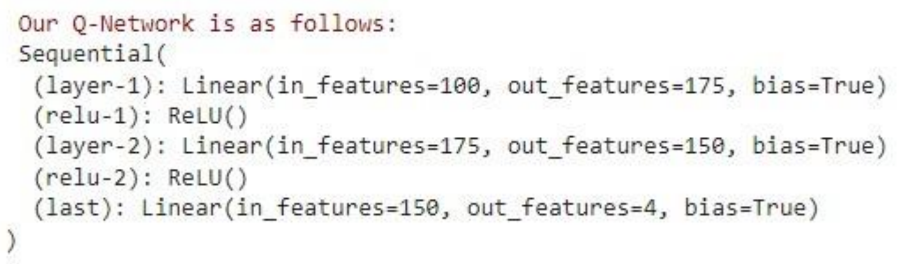
Slide: This is located at S18. So, if agent enters this cell, the agent has 0.1 probability that he might end up in S19 with a penalty of -5.

Main Objective: The main objective of this grid environment is, the agent is positioned at S1 and the treasure is located at S25. The agent must find a way to obtain the treasure by going through various obstacles. This project runs a random agent for 10 timesteps by taking various random actions and returns the observation, reward and other information about the environment.

3. DQN Implementation

a) For GridWorld

In this project, we have used ReLu based feed forward networks with three layers and output dimensions equal to the action space. Below is the structure of our Neural networks used in this project.



```
Our Q-Network is as follows:
Sequential(
  (layer-1): Linear(in_features=100, out_features=175, bias=True)
  (relu-1): ReLU()
  (layer-2): Linear(in_features=175, out_features=150, bias=True)
  (relu-2): ReLU()
  (last): Linear(in_features=150, out_features=4, bias=True)
)
```

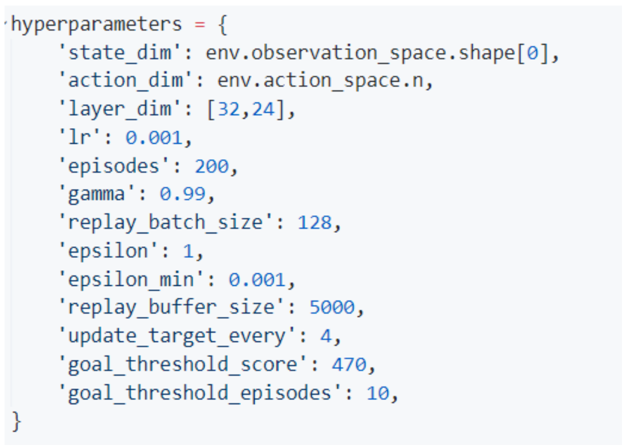
Fig 5.1: Feed froward Neural Networks

After hyper parameter tuning, we found the agent converge after 1000 epochs with following hyoer parameter values.

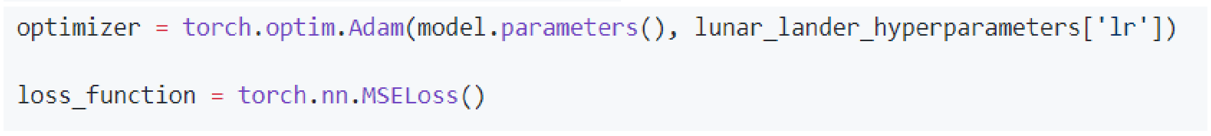
Learning rate = 2*e^-4
Discount factor = 0.89
Epsilon Decay – Linear Epsilon decay initiating at 0.9 and decaying till 0.1
Maximum steps = 20

b) Cartpole-v1

Like the grid environment, we define a FC neural network with 2 hidden layers of 32, 24 neurons respectively. The other hyperparameters are defined as below.



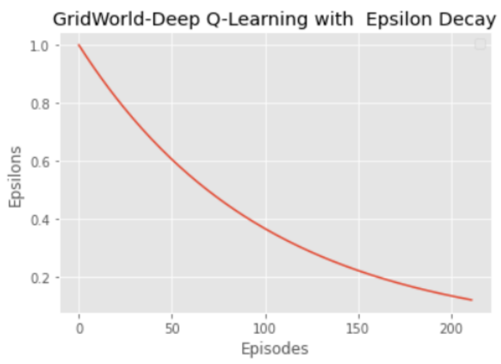
```
hyperparameters = {
    'state_dim': env.observation_space.shape[0],
    'action_dim': env.action_space.n,
    'layer_dim': [32,24],
    'lr': 0.001,
    'episodes': 200,
    'gamma': 0.99,
    'replay_batch_size': 128,
    'epsilon': 1,
    'epsilon_min': 0.001,
    'replay_buffer_size': 5000,
    'update_target_every': 4,
    'goal_threshold_score': 470,
    'goal_threshold_episodes': 10,
}
```



```
optimizer = torch.optim.Adam(model.parameters(), lunar_lander_hyperparameters['lr'])

loss_function = torch.nn.MSELoss()
```

We use an exponential decay function to handle the exploration-exploitation policy.



c) LunarLander-v2

Like the cartpole case we use a fully connected network with two hidden layers of size 150 & 120 and Mean Squared Error loss with Adam optimizer to do the gradient descent.

```
lunar_lander_hyperparameters  = {
    'state_dim': env.observation_space.shape[0],
    'action_dim': env.action_space.n,
    'layer_dim': [150, 120],
    'lr': 0.001,
    'episodes': 400,
    'gamma': 0.99,
    'replay_batch_size': 64,
    'epsilon': 1,
    'epsilon_min': 0.01,
    'replay_buffer_size': 100000,
    'update_target_every': 4,
    'goal_threshold_score': env.spec.reward_threshold,
    'goal_threshold_episodes': 100,
}
```

```
optimizer = torch.optim.Adam(model.parameters(), lunar_lander_hyperparameters['lr'])

loss_function = torch.nn.MSELoss()
```

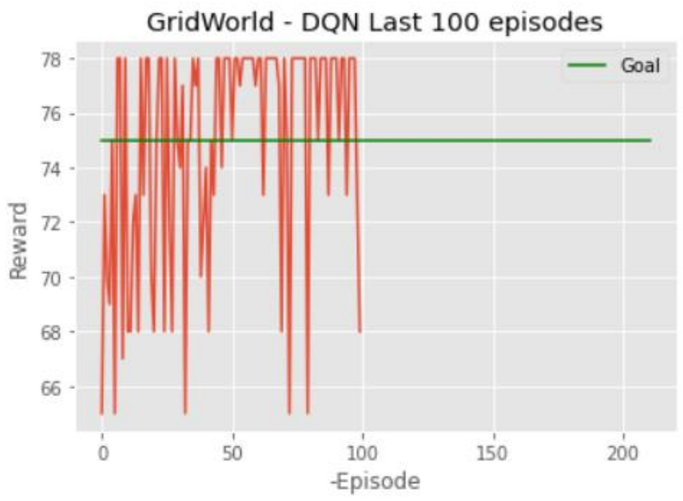
Epsilon was initiated at 0.9 and decayed till 0.1 using the formula

*decay factor = (Final epsilon value / Initial epsilon value) **(1/number of episodes)*

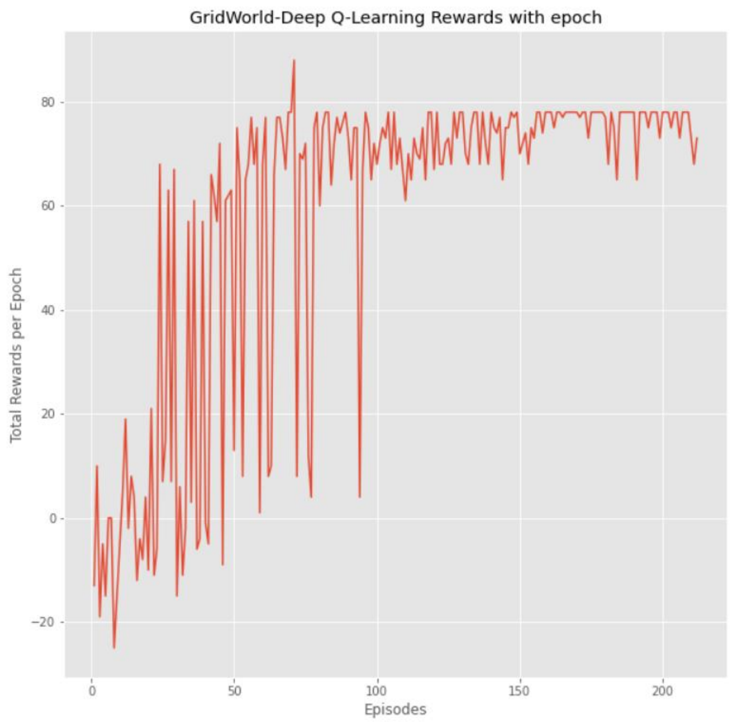
4. Evaluation

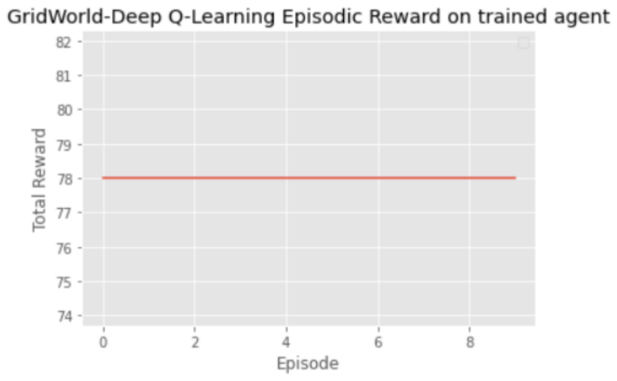
After training the model we have achieved optimal policy in 8 timesteps. After iterating the test model for 5 epochs, we have got following results.

a) GridWorld
DQN:

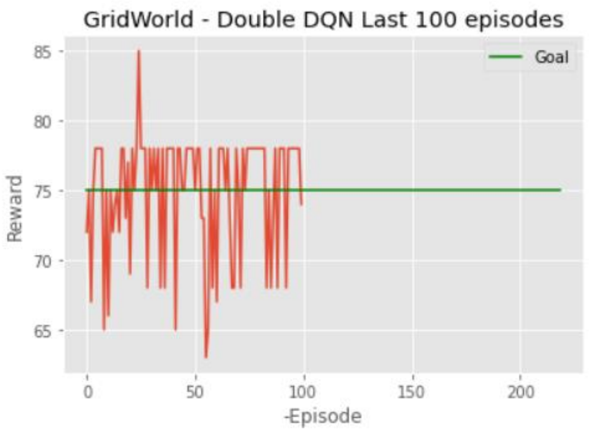


We reach the goal of an average of 75 reward for the agent in the grid world at episode 211.



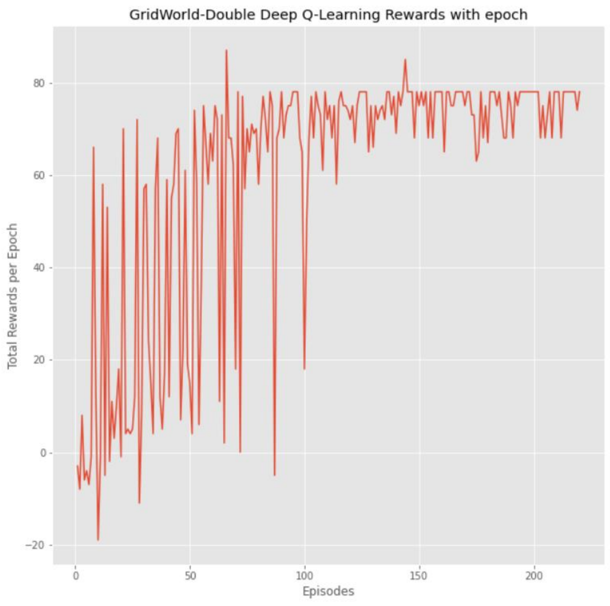


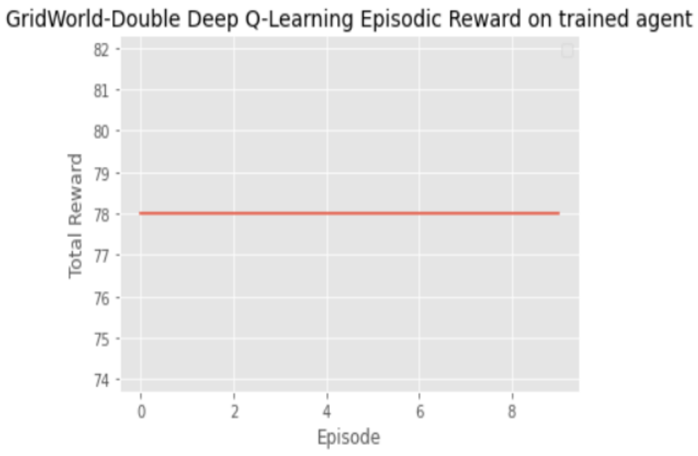
DDQN:

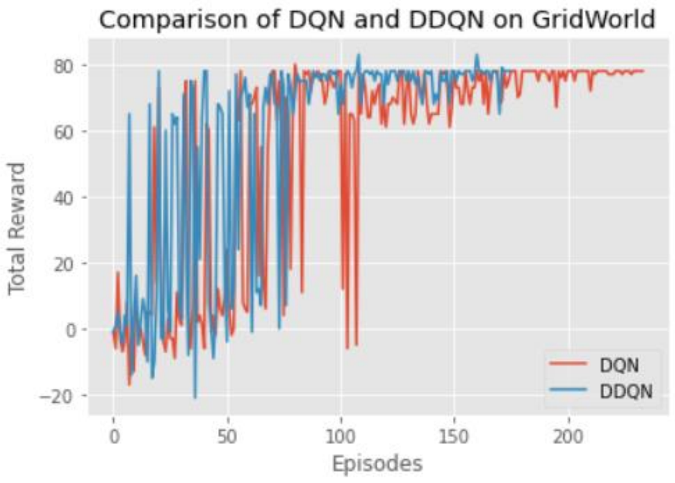


We reach the goal of an average of 75 reward for the DDQN algorithm at episode 219.

We can see that DDQN converges a bit faster, and it has less overestimation after convergence and during exploitation.



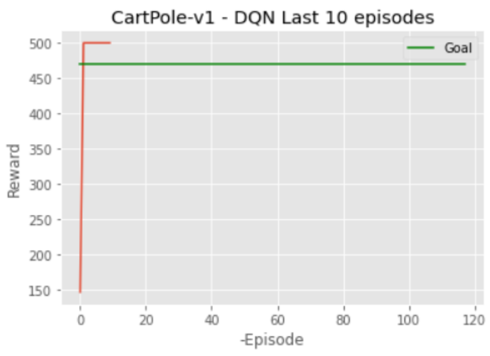




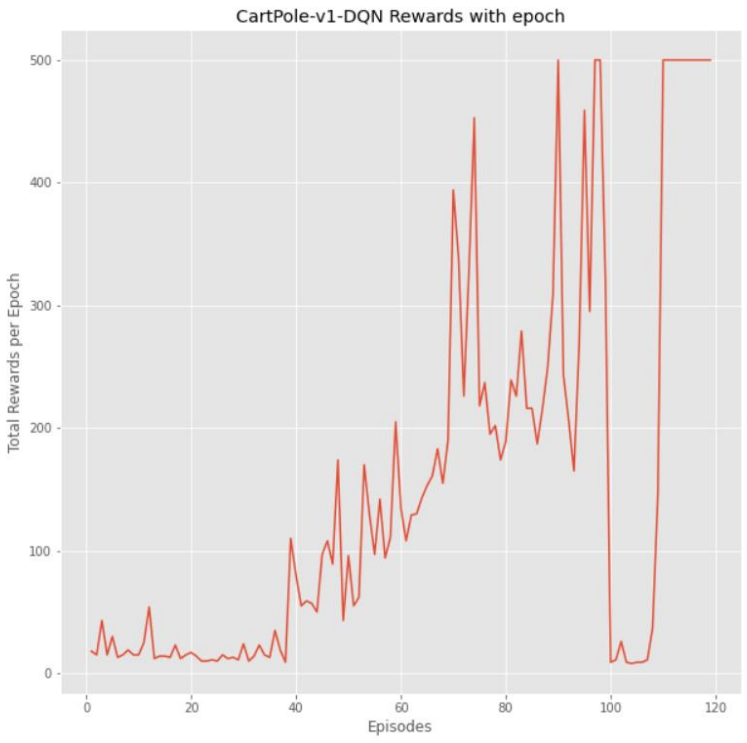
We can see that DDQN converges a bit faster, and it has less overestimation after convergence and during exploitation.

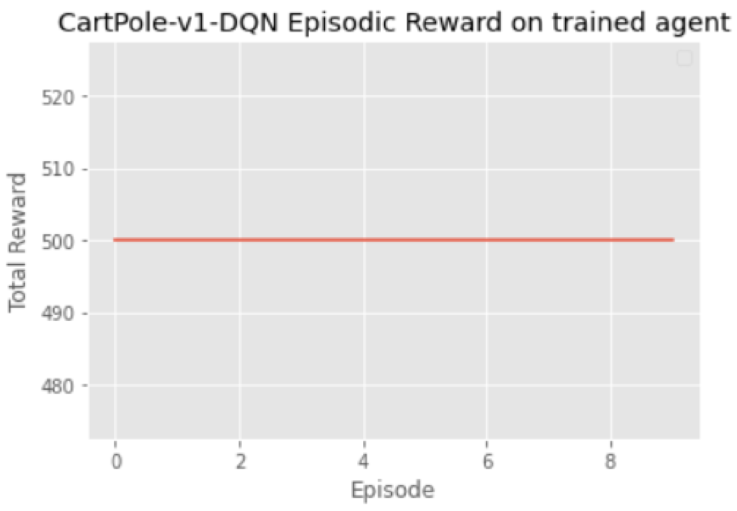
b) Cartpole-v1

DQN:

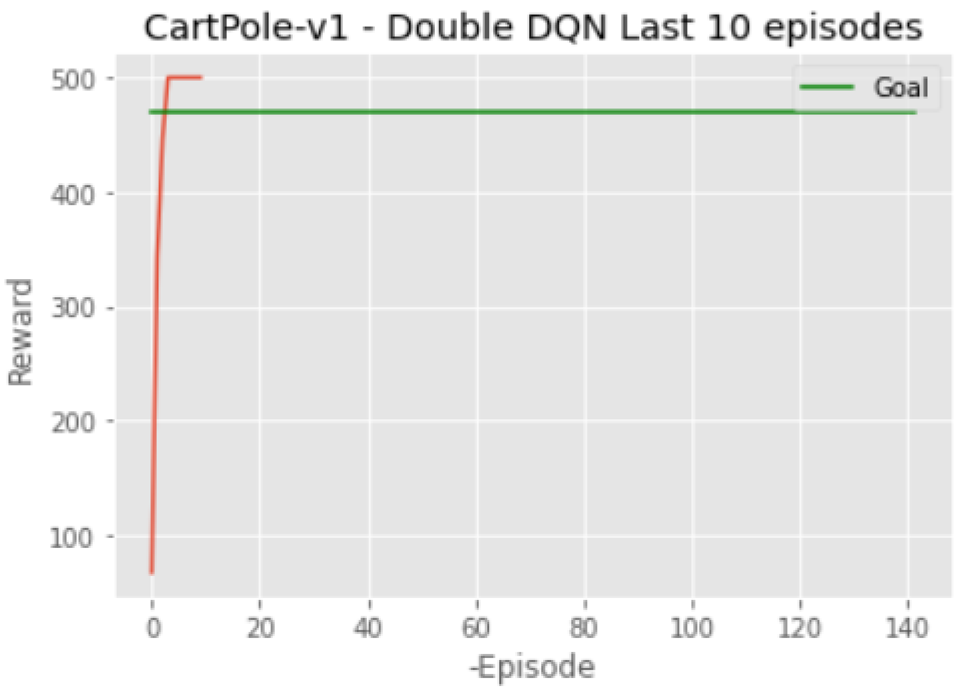


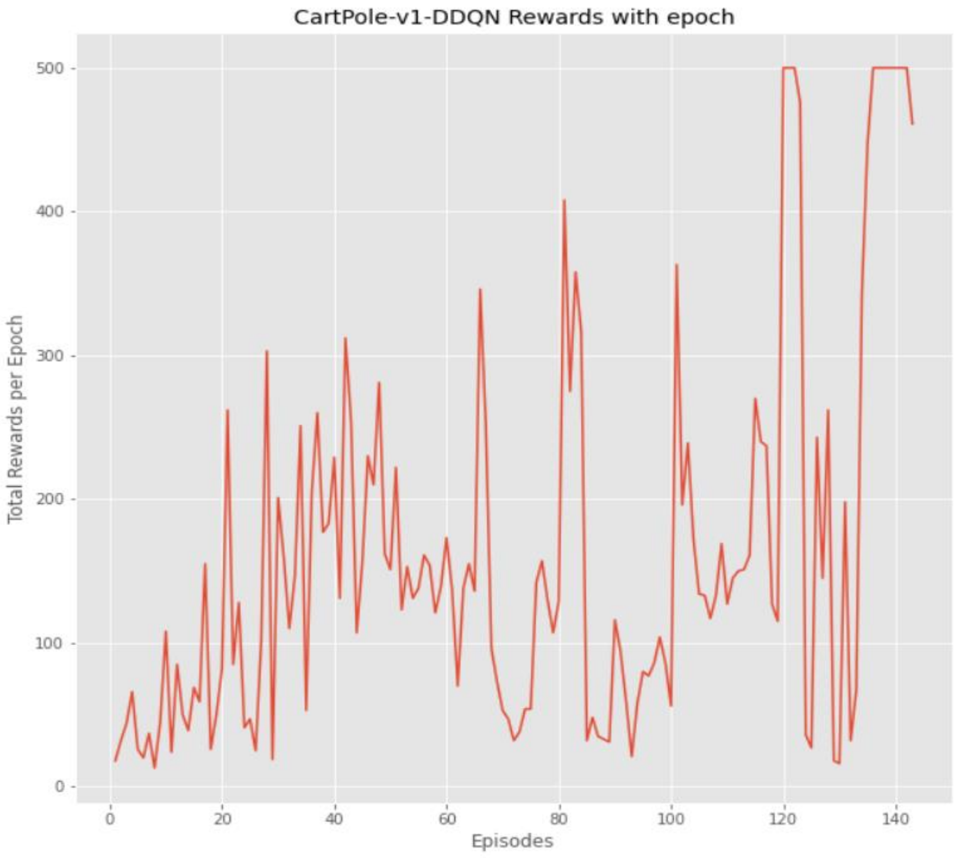
We see convergence for the Cartpole at episode 118, we reach an average reward of 470 for the last 10 episodes.

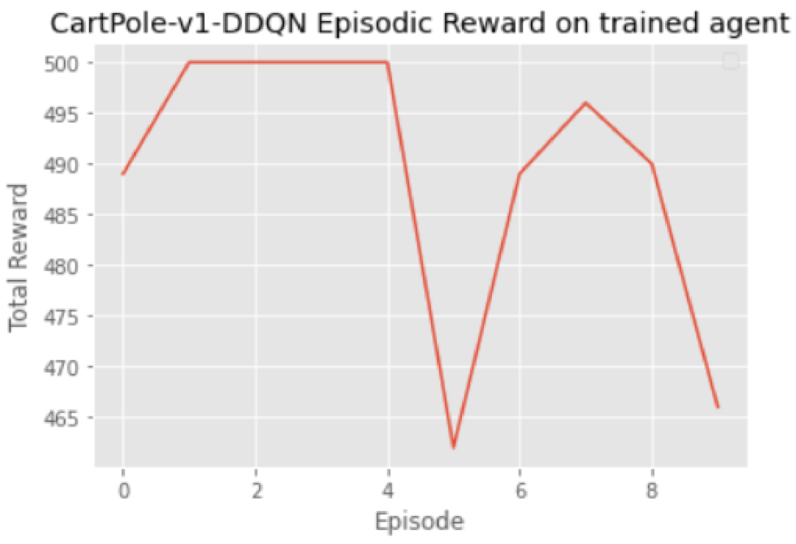


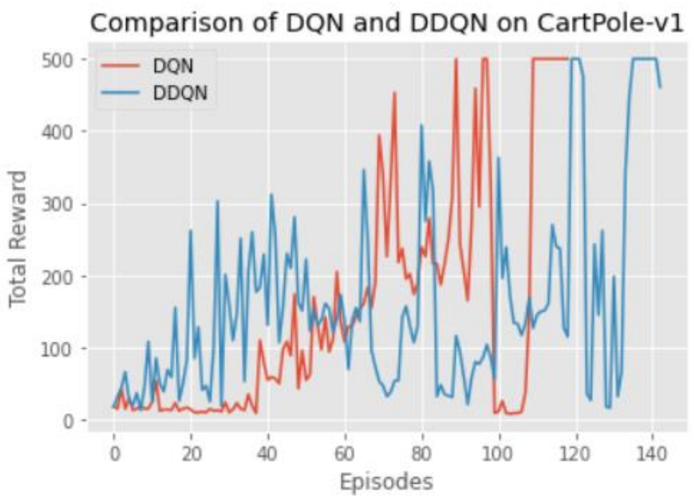


DDQN:


CartPole-v1 - Double DQN Last 10 episodes
Reward
Episode
Goal
500
400
300
200
100
0
20
40
60
80
100
120
140

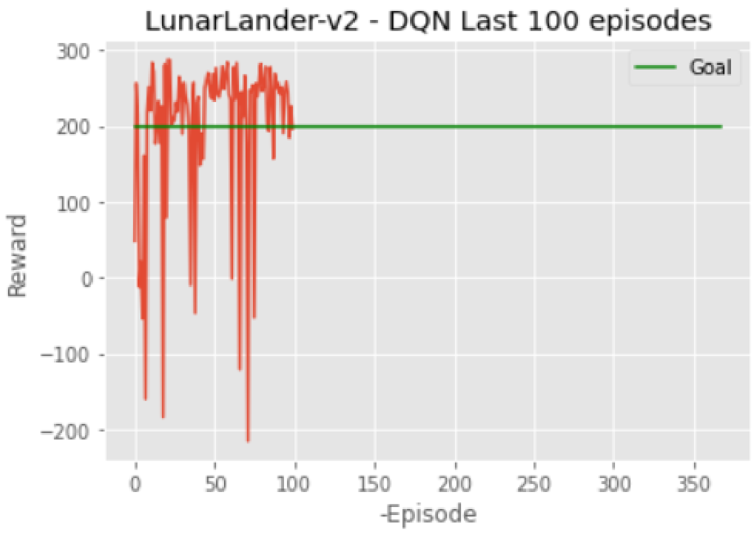

CartPole-v1-DDQN Rewards with epoch
Total Rewards per Epoch
Episodes
500
400
300
200
100
0
20
40
60
80
100
120
140



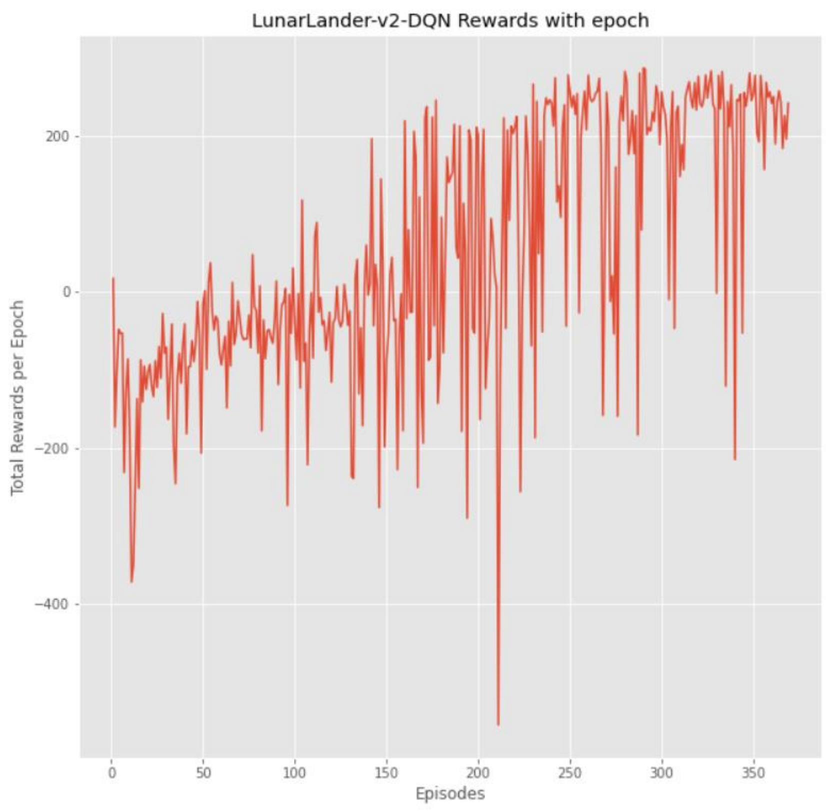


c) LunarLander-v2

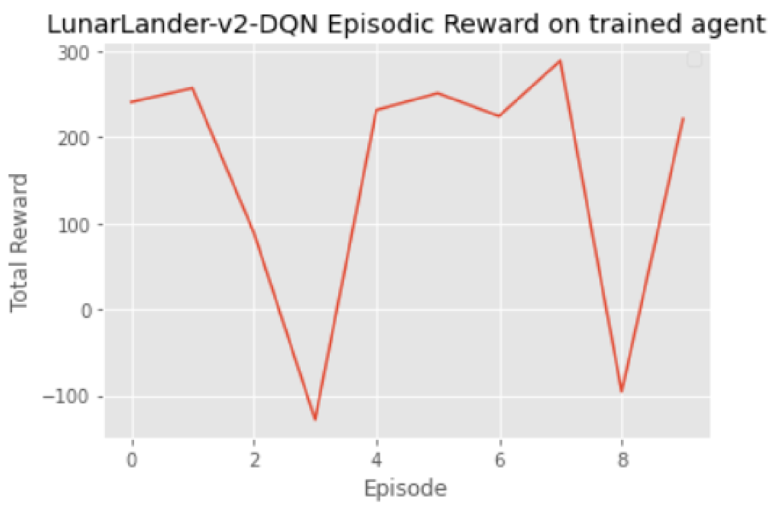
DQN:



The goal is 200 average rewards for last 100 episodes is reached at episode 368.

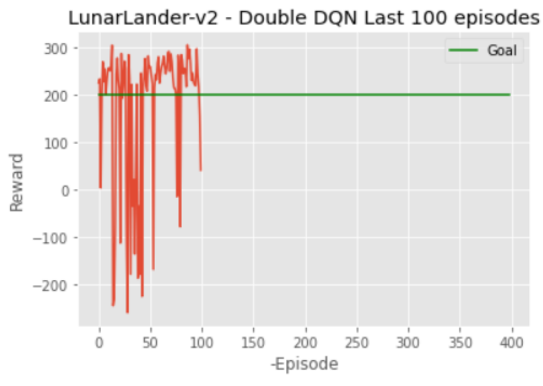


LunarLander-v2-DQN Rewards with epoch
Total Rewards per Epoch
Episodes
200
0
−200
−400
0
50
100
150
200
250
300
350

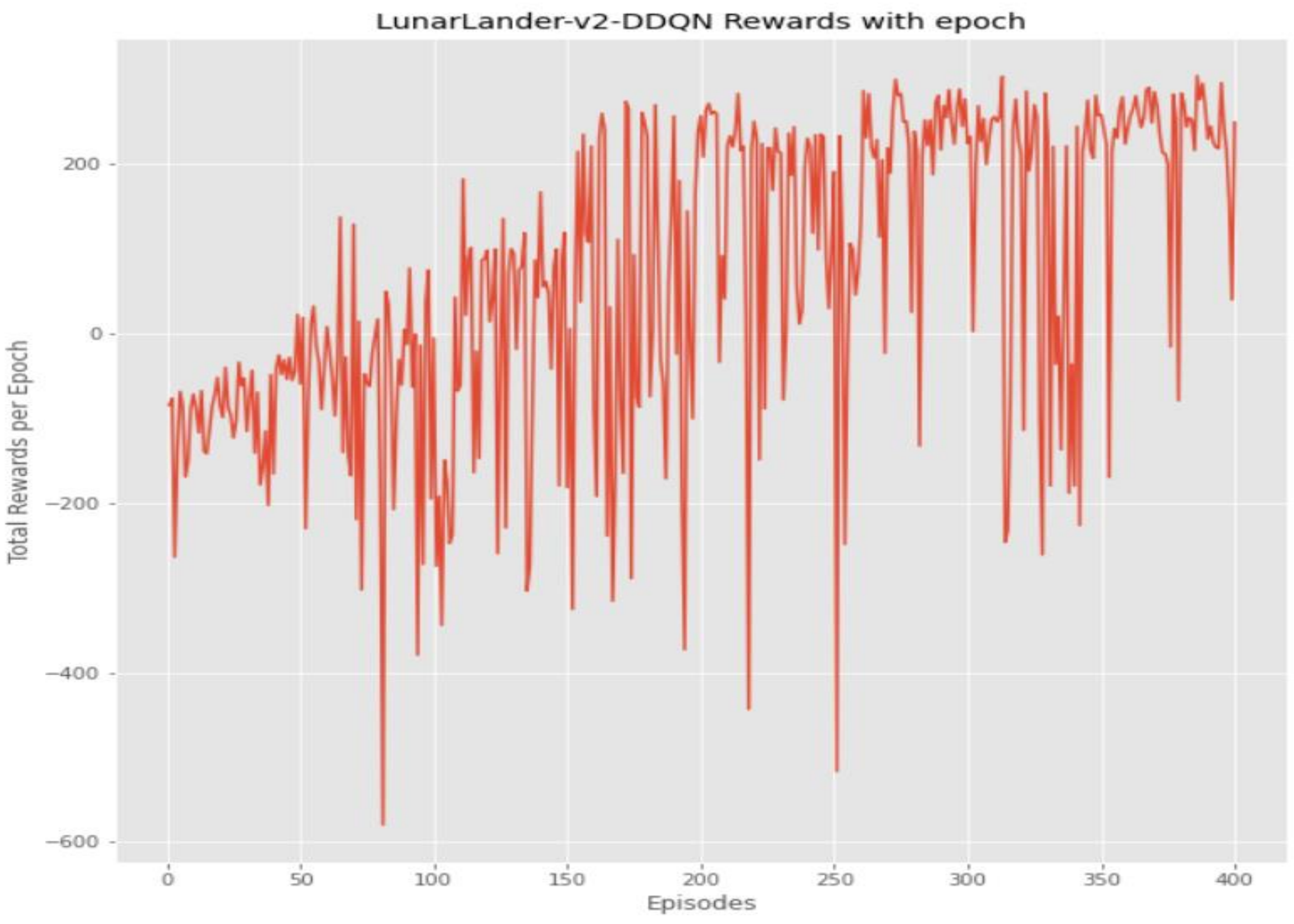


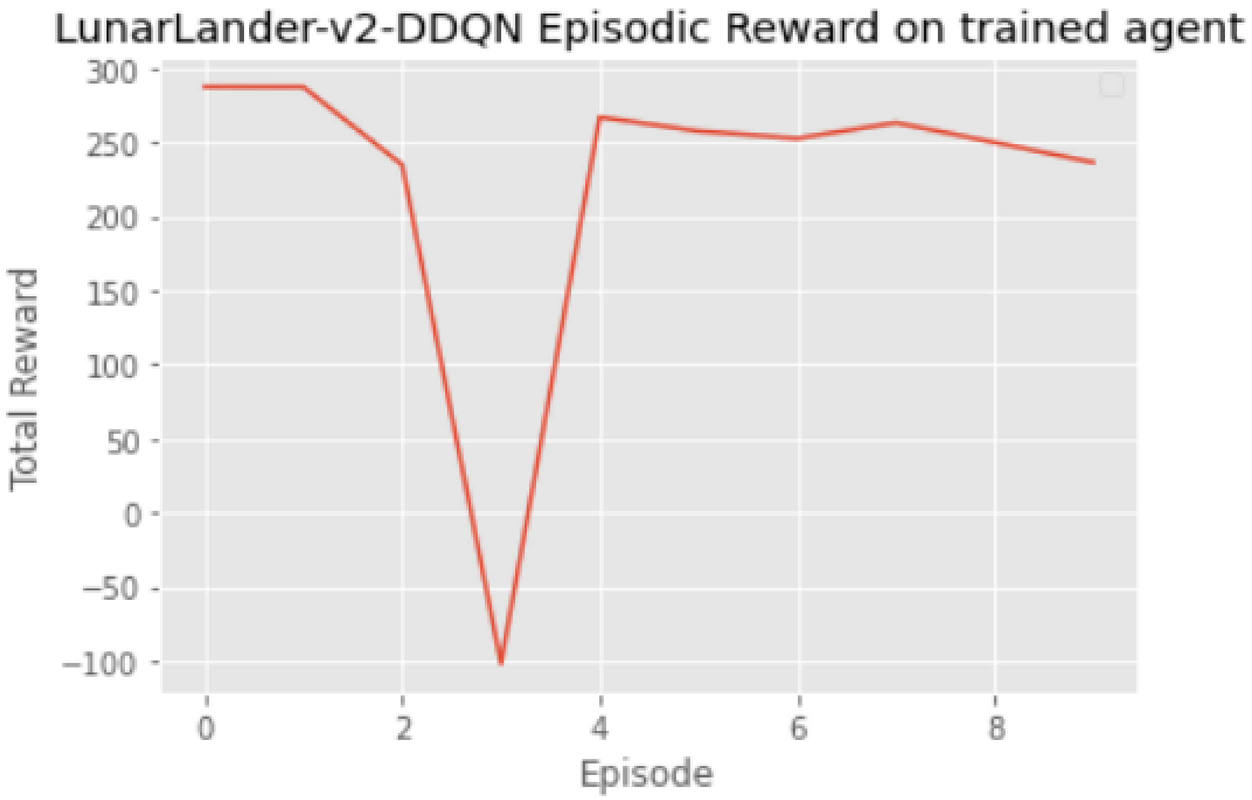
LunarLander-v2-DQN Episodic Reward on trained agent
Total Reward
Episode
300
200
100
0
−100
0
2
4
6
8

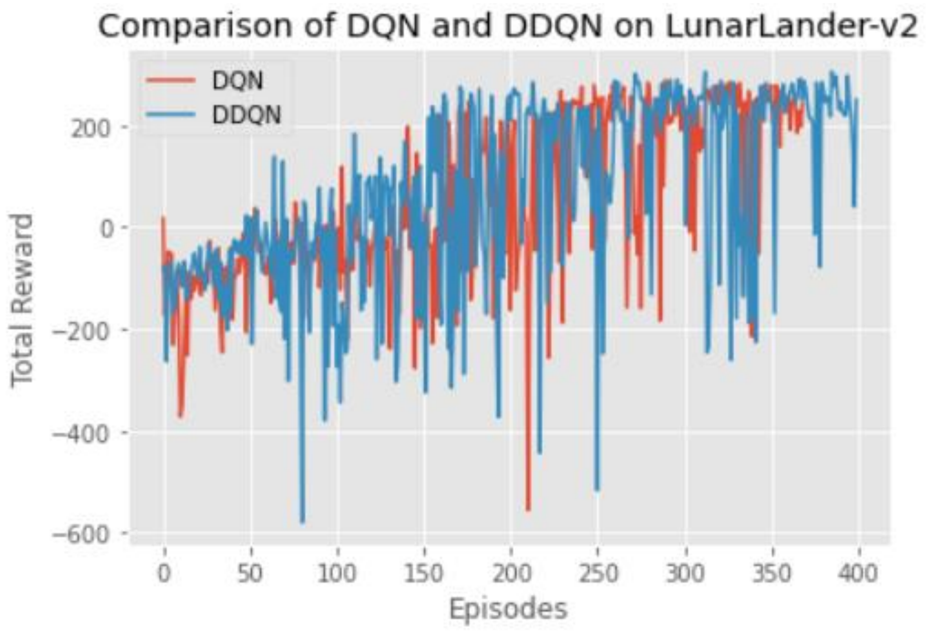
DDQN:



LunarLander-v2 - Double DQN Last 100 episodes
Goal
Reward
-Episode
300
200
100
0
−100
−200
0
50
100
150
200
250
300
350
400


LunarLander-v2-DDQN Rewards with epoch
Total Rewards per Epoch
Episodes
200
0
−200
−400
−600
0
50
100
150
200
250
300
350
400


LunarLander-v2-DDQN Episodic Reward on trained agent
Total Reward
Episode
300
250
200
150
100
50
0
−50
−100
0
2
4
6
8



For the lunar lander, we can see that the performance of both DQN and DDQN is very similar.

5. Conclusion

To conclude, this project has implemented DQN, DQNN with feed forward neural networks as the Q-Networks and achieved optimal policy on a custom Grid World, OpenAI Cartpole-v1, OpenAI LunarLander-v2.

We 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.

References

[1] https://gym.openai.com/docs/
[2] https://en.wikipedia.org/wiki/Reinforcement_learning
[3] Lecture Slides
[4] DQN Explained | Papers With Code and Original papers referenced here.