CSC498 Final Report

Abdulwasay Mehar

April 2021

1 Introduction

In this project we will aim to investigate the some of the Deep learning based algorithms such as the Deep-Q-Network Family on simple Atari games available in the Open-AI Gym library. The main goal of each investigation will be to determine how the hyper-parameters can be fine tuned to get better performance, as well as apply some of the improvements that have been developed to resolve issue in the vanilla algorithm. The main motivation behind choosing the DQN family as the main algorithm is how we can approximate a policy using neural networks. Since there is wide range of neural networks, learning how neural networks are used and fine-tuned for best performance becomes an interesting puzzle and a unique method to learn complex policies.

2 Algorithms

The idea of using neural networks to approximate policy and Q-values stems from the idea of computing Q-Values for large states and actions. Under a real world scenario, it is often the case that the state and the action space are either continuous or very larger, and computing large/continuous Q-Table can very inefficient. In Deep Reinforcement Learning, we are essentially representing different components of the agent such as the values or policies using neural networks, where the parameters of the neural network are optimized through gradient descent over transitions from the environment.

2.1 Deep Q Network

Deep-Q-Network (DQN) is a form of Deep-Reinforcement-Learning, where neural networks are used to approximate Q-values for the state-action pairs. The algorithm makes uses of 2 neural networks during the training process, namely acting-network and target-network. Where the acting-network is used to interact with the environment, while the target-network is used as the validator for the acting-network when computing loss. After a certain number of steps, the weights of the acting-network and the target-network are synchronized.

To define the loss which the algorithm will aim to minimize is simply the squared difference between the values produced by the target-network and the predicted values produced by the acting network.

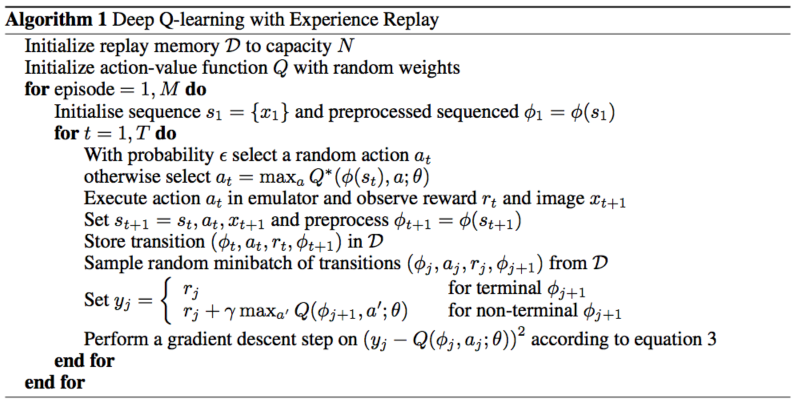
$$Loss = (r + \gamma * max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

$$r = \sum_t r_t$$

Where r is the sum of rewards from time-step t till the end of the episode T, (s', a') is the next state action pair, (s,a) is the current state action pair, (θ') represents the set of weights for the acting-network, while (θ) represents the set of weights for the target-network and γ represents the discount factor.

2.2 Deep Q Network with Experience Replay

One of the issues that the vanilla DQN algorithm faces exists in how it generates transitions for training. In the vanilla DQN, the algorithm essentially switches between training and collecting new transitions, where collecting new transitions is done through the agent interacting with the environment. The method essentially produces batches of sequential experiences which causes the algorithm to be very unstable. In other word, the models creates a bias towards the sequential sets of actions it as seen, instead of generalizing the use of the available actions. To battle this problem, the idea of experience replay explained in [2], will allow us to break the sequential order. Essentially, the idea is to store the generate raw transitions from the environment in to a large table(replay-buffer) of transitions, which will then be used to randomly sample a batch so size n for training. By randomly sampling from a large set of experiences, it will give the model a set of totally unrelated set of transitions for training, and hence will allow the model to generalize the usage of available actions



```
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
```

Figure 1: Pseudo-code of DQN with experience replay

2.3 Double Deep Q Network

$$Loss = (r + \gamma * max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$
$$Q_{target} = r + \gamma * max_{a'} Q(s', a'; \theta')$$

Another issue that DQN experiences is with overestimating the Q-Values of state-action pair. If we look at the loss equation for DQN, we can see that the Q_{target} is being calculated with the weights of the target network. This is problematic especially at the beginning of training since we don't have enough information to say that the best action is for the next state is the one with the highest Q-Value. Which means, if this happens at the beginning of training, the algorithm is basically head towards taking non-optimal actions later in training. Getting the model to take optimal actions later on will be complicated as the neurons have adjusted to produce high Q-Values for non-optimal actions.

Double Deep-Q-Network (DDQN) resolves this issue by changing the way it calculates the Q_{target} value. As specified in [3], the best action is chosen by the acting-network, while the Q-Value is computed using the weights for the target network. This will allow the algorithm to tone down the wrong Q-Values produced by the target network early in training, and hence will lead the model to a more optimal policy. While the overall flow of the DQN algorithm remains the same, the loss computation has changed as it can be seen in the equation below.

$$Loss = (r + \gamma * Q(s', max_{a'} Q(s', a'; \theta); \theta') - Q(s, a; \theta))^2$$

3 Environments

For the investigation, I will be using two environments to see how each variation of the algorithm effects the result of the model in completing the goal task. The two chosen environments exists in the Classic-Control section in the Open-AI[4] library, namely: CartPole-V1 and MountainCar-V0.

3.1 CartPole-V1

As specified in [5], the environment contains a pole attached by an un-actuated joint to a cart, which moves along a frictional surface. The main goal is to prevent the pole from falling by applying force on the cart on its left or right. The list below contains specific details on allowed actions, reward, and resolve state:

- Actions: [1, 2] 1=Apply force left 2=Apply force right
- Rewards: +1 for every time-step the pole remains upright.
- End of Episode: when the pole is more than 15 degrees from vertical or the cart moves 2.4 units from the center
- Solved Requirement: Average score of 195 over 100 consecutive trials

3.2 MountainCar-V0

As specified in [6], the environment contains a car on a one-dimensional track, positioned between two mountains. The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. The list below contains specific details on allowed actions, rewards, and resolved state:

- Actions: [0, 1, 2] 0=Do Nothing 1=Move Left 2=Move Right

- Rewards: -1 for every time-step the car is below the horizontal distance of 0.5
- End of Episode: when the car reaches a score of -110 or 200 time-ssteps have passed
- Solved Requirement: Average score of -110 under 200 time-steps over 100 consecutive trials

4 Investigation

The main focus of the investigation was to determine which variation gave the best overall performance during training. To do this, there were several different configurations for each variation that were used to obtain a good overall performance. Each of the configurations were applied to both of the environments, and their final results were compared to the optimal-results specified for each environment in the pervious section.

4.1 Vanilla DQN

To see the effect of each of the variations of DQN, the vanilla DQN was first applied to see what the current performance of the algorithm. The configuration used for this experiment can be seen in the table below:

Network Layers	2
Total Episodes	200
Steps Before Updating Target	700
Gamma	0.99
Learning Rate	1e-4

Table 1: Configuration 1 for Vanilla DQN

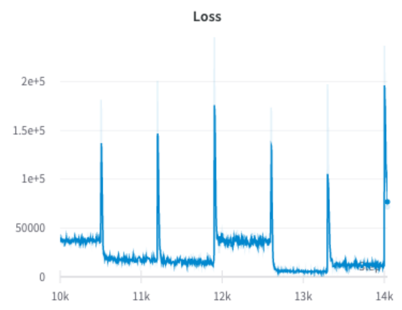


Figure 2: Vanilla DQN Loss Graph for CartPole

4.1.1 CartPole

The first experiment that was conducted was on learning to solve the CartPole problem using the configuration in Table-1, where the algorithm is updating the target model steps within each episode. Now, if we look a the loss graph of this experiment in Figure-2, we can see that the training was clearly unstable, and the algorithm was not able to learn anything useful. If we take the average reward that the model achieves over 100 consecutive runs, we get an average reward of 1.1. This kind of performance was clearly expected from the beginning since CartPole requires more situational based reaction rather than a sequential one.

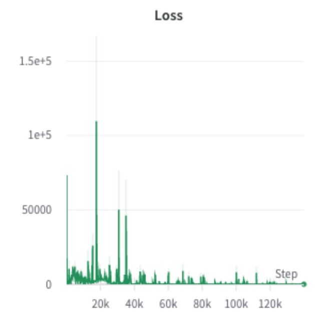


Figure 3: Vanilla DQN Loss Graph for MountainCar

4.1.2 MountainCar

If we apply the same configuration to the learn the MountainCar problem, we see a different but a similar result. If we look at Figure-3 we can see a there are many instances of the loss jumping from a loss of zero to a very large loss; however, one thing to notice here is that the spikes decreased in size over the course of training, which is a good indication that the model has learned something useful. So if we take the average score over 100 consecutive runs, we end up getting a score of -200, where as the optimal score is -110 under 200 timesteps.

The result for this training is somewhat surprising as it doesn't match with what the loss function is indicating. So in order to see the issue, I rendered each of the 100 runs to see what was happening, and I found out that the network was constantly trying to go up right hill without building a momentum. One thing that I suspected could've cause this behaviour could be the reward function. It is possible that model is not able to find a somewhat optimal path due to the negative rewards, and hence is not able to find some method to create momentum. To battle this, I essentially modified the reward function such that if the model is trying to accelerate to the left while going left, or is trying to accelerate to the right while going right, it gets a positive reward of 1.

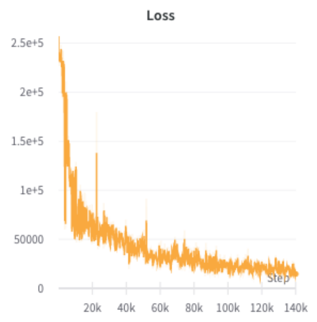


Figure 4: Vanilla DQN Loss Graph for MountainCar with modified reward function

After training on the same configuration, but with the new reward logic, I was able to see a better convergence of the loss than the one in Figure-3. If we look at Figure-4 we can see that although the loss has not reached anywhere close to zero, it has a nice smooth motion of convergence, which is a very good sign of improvement and performance. If we run the model on 100 consecutive trials, we get an average score of -127. A smashing improvement!

This result made sense as the new modified reward function gave the algorithm some positive feedback during its training. Another issue that might've been avoided is the unstability of DQN. Since vanilla-DQN trains on sequential transitions, it seems like it helped the algorithm learn how to create momentum, since creating a momentum requires sequential set of actions.

4.2 DQN with Experience Replay

In order to see if non-sequential transition batches help in obtaining a better performance than the one achieved from the vanilla-DQN algorithm. Experience Replay will now be used to extract out transition batches for training.

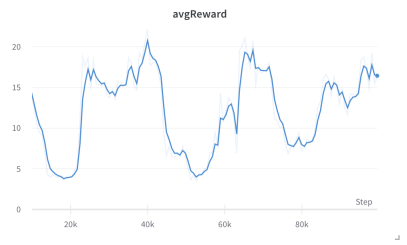
Network Layers	2
Total Episodes	200
Length of each Episode	1000
Environment Steps Before training	100
Steps Before Updating Target	200
Buffer Size	10000
Sample Size	2500
Gamma	0.99
Learning Rate	1e-4

Table 2: Configuration 2 for Vanilla-DQN with Replay Buffer

The flow of the algorithm is a bit different than the one of used for with the vanilla-DQN. To begin with, the first 10000 interactions were used to fill up the replay-buffer before any training is done. Once the buffer has been filled, the following flow of training is used for each epoch:

```
for i in range(10000):

    if i % ENV_STEPS_BEFORE_TRAIN != 0:

        action = select some action
        obs, rew, done = perform the action in environment
        transition = (obs, rew, done, action, lastObs)
        replayBuffer.insert(transition)

    else:

        samples = replayBuffer.sample(BATCH_SIZE)
        loss = compute Network Loss

        perform Back propagation

    if i % ENV_STEPS_BEFORE_TARGET_UPDATE == 0:

        ** Sync weights of acting and target network
```

4.2.1 CartPole



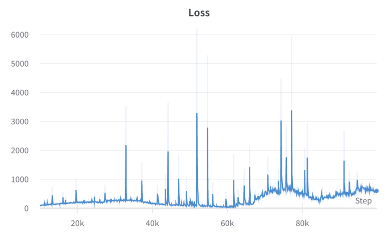


Figure 5: DQN with Experience replay, left=AverageReward, right=Loss

After training the model for 200 episodes, the results were more promising than the one achieved using the vanilla-DQN. If we look at the right image in Figure-5, we can see that the model did indeed learn something useful during the middle of the training, then started too head towards a diverging path. This is also evident from the left image in Figure-5, where the average reward seemed to form an upwards trend till the very end. So if take the average score over 100 consecutive trials, we get an average score of 9.46. Which is not bad compared to 1.1, but still far away from the optimal score of 195.

One alternative that can be used to improve this would be incorporating an epsilon exploration/exploitation approach. The idea, is to essentially use epsilon to determine how we will be selecting actions when interacting the environment.As training progresses, epsilon decays to 0, in which case we are selecting actions with the highest Q-values (i.e. exploitation). In addition to the configurations in Table-2 two additional parameter will be added to the configuration as specified in Table-3.

EPSILON	1.0
EPSILON DECAY	0.99998

Table 3: Configuration 2 with Epsilon Decay

```
EPSILON = EPSILON * EPSILON_DECAY
randomFloat = random()
if randomFloat < EPSILON:
    *Choose action randomly
else:
    *Pick the action with the highest Q Value
```

The pseudocode above represents how actions will be selected whenever the algorithm needs to add new transitions into the replay buffer.

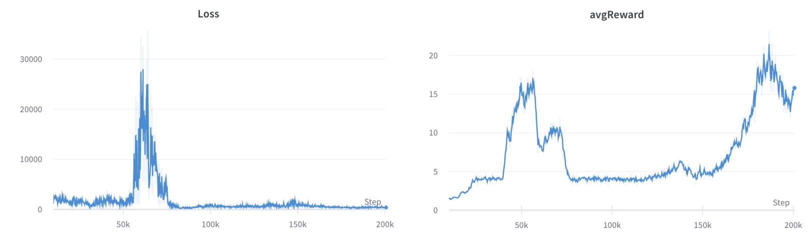


Figure 6: DQN with Experience replay + Epsilon Decay, (left=Loss, right=Average Reward)

If we look at Figure-6, we can see that the overall loss is decreasing and is getting close to zero. The spike at the beginning is simply displaying signs of early exploration due to epsilon. Clearly this method paid off, as it can be seen in the reward graph in Figure-6 that the average reward reaches a new all time all time high especially towards the end of training. Which makes sense since at this point if we look at Figure-7 epsilon is very small, and is making the algorithm pick the action with the highest Q-Value (i.e. conducting exploitation). So if we take the average score over 100 runs, we get a score of 41.24 . Again a really good improvement, but still not close to the optimal.

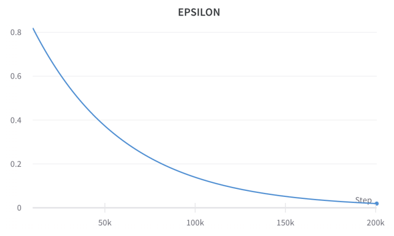


Figure 7: DQN with Experience replay + Epsilon Decay

4.2.2 Mountain Car

In order to get the best performance from the start, I trained on the DQN algorithm containing both the Replay-Buffer and Epsilon. Along with the algorithm, the same configurations as the one specified in Table-2 and the modified reward function was used for this experiment.

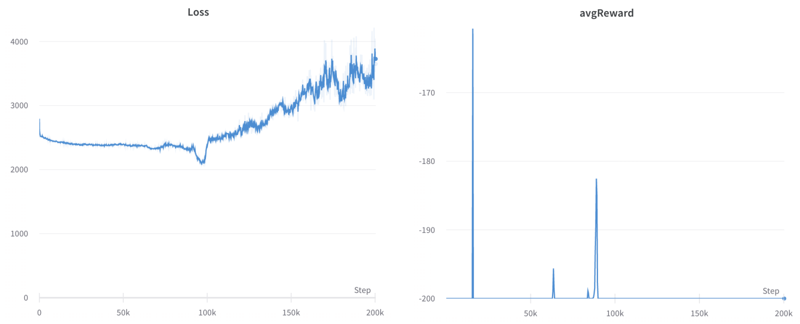


Figure 8: DQN with Experience replay + Epsilon Decay, (left=Loss, right=Average Reward)

As it can be seen in Figure-8, after the mid point of training, the loss actually starts going up. The decrease in performance can also be seen in right image of Figure-8, where at the beginning we see some instances of the model getting better average score, however after the midpoint, the average reward stays at a constant score of -200 (i.e, the model was not able to solve the problem under 200 time-steps). If we take the average score over 100 consecutive iterations, we get a score of -200 as expected. One possible reason why this might've happened could be because of the random sampling from the replay-buffer. The random samples could cause the model to look at beneficial sequence of transitions that could be used to build momentum efficiently as un-rewarding, which caused it to essentially diverge.

4.3 Double Deep Q Network

Finally, the last method is to see how the network performs after training with the Double-Deep-Q-Network algorithm. In order to keep things simple, we will incorporate replay buffer and the additional improvement of epsilon decay into the algorithm from the very start.

4.3.1 CartPole

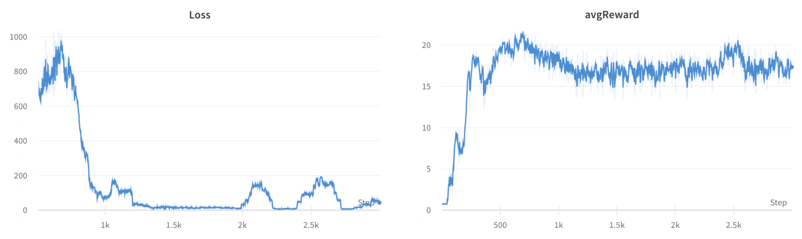


Figure 9: DDQN with Experience replay, (left=Loss, right=Average Reward)

As we know from our previous experiments with the DQN algorithm, it can be said that learning to solve the CartPole problem through sequential transitions is very inefficient and unstable. Which is why, using DDQN with Experience Replay becomes even more beneficial since the way we compute Q_{target} reduces variance, while experience replay will help in breaking the sequential order of transitions for training. For this experiment we will use the same configuration as the one specified in the Table-2 to see the main effect of DDQN.

If we look at the left image of Figure-8, we can see that the algorithm is able to decrease its loss early in training, and continues to do so at a lower rate through out the episodes. Signs of better performance can also be seen in the right image of Figure-8, where we see a similar trend as the one in the loss graph, except that the trend is towards increasing the average reward. If we take the average score over 100 consecutive runs, we get a score of 58.57 , which is a good improvement but still not enough. One method that may improve the performance, is if we train the model for an additional number of episodes. This was not done due to time constraint and computational resources.

4.3.2 MountainCar

To see any improvements, there were essentially two variations of DDQN that were used in the experiment. The first one being the standard DDQN with Replay-Buffer, and the second one being the standard DDQN without the Replay-Buffer. As we have seen previously, that the relay buffer lead the model to a diverging path.

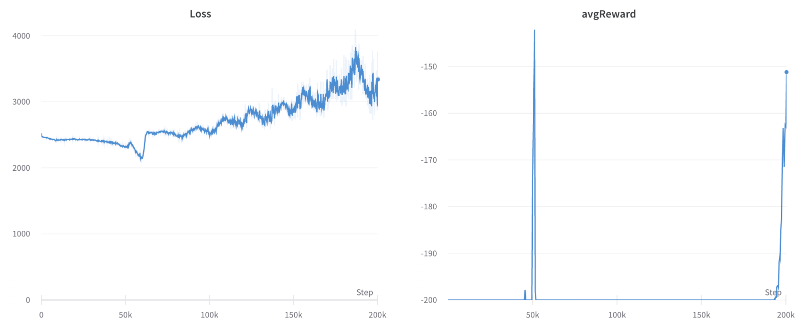


Figure 10: DDQN with Experience replay, (left=Loss, right=Average Reward) for MountainCar

If we look at the left image of Figure-10, the pattern in the loss graph seems almost identical to the loss graph in Figure-8. This is not surprising, especially in the case of MountainCar, where learning from a sequential set of transitions remains to be more beneficial. However, one things to notice in this experiment is that the model showed signs of improving towards the end, which is surprising considering its past trend. If we take the average score over 100 consecutive runs, we get an average score of -154. In terms of using DDQN with experience replay, this is certainly a good improvement compared to the average score of -200 from using DQN with experience Replay.

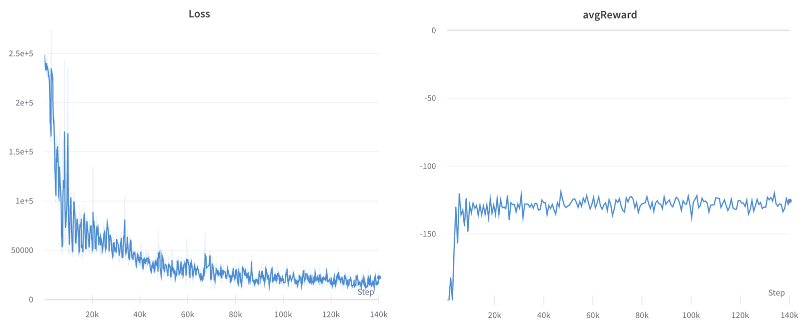


Figure 11: DDQN without Experience replay, (left=Loss, right=Average Reward) for MountainCar

In the second experiment, we are training a model using the standard DDQN algorithm (one without experience replay). In this experiment we will use the same configuration specified in table-1 in order to compare the results between vanilla DQN and vanilla DDQN. Now, if we look at Figure-11 and compare its loss graph with the one in Figure-4, it can be said that the overall trend of the loss graph remains about the same. One thing to notice here is that the decrease in loss is a lot less noisy than the one in Figure-4. This makes sense, since the only thing that changed in the algorithm is the way we compute the target Q value. The modification in computing the target value serves to reduce the over estimations of Q-values, which is clearly evident from the smooth decrease in loss. Additionally, if we look at the reward graph in Figure-11, we can see that there is barely any improvement after reaching a certain average reward. One possible reason why this might be happening, could be because of the model getting stuck at a saddle point, and is not able to find a way to get out that point to find a more optimal minimum. If we take out the average score over 100 consecutive runs, we get a score of -115. A good improvement from -127, and a lot closer to the optimal score.

5 Conclusion

From the above experiments, one can certainly conclude that each variation of DQN has its own pros and cons for any given task at hand. We saw that the incorporating experience-replay into DQN and DDQN to solve the Mountain-Car problem backfired and gave bad performance. While it helped in increasing the performance for the CartPole Problem. Similarly, we can also conclude that use the standard DDQN algorithm, always gave a better performance than the one produced by DQN. Finally we also saw that the usage of epsilon to conduct exploration and exploitation through out the training also helped in improving the performance.

For future work, different variations of configurations can be experimented with to see how each parameter affects the learning process for each of the variations of DQN, and whether there exists some optimal value to obtain optimal performance. In addition to this, a more recent variation of DQN, namely Rainbow DQN, which incorporates changes from several different variants of DQN can be implemented to see how it fairs with the variations tested in the project.

Overall, I would like to say that going through this project was fun and challenging. I would've explored more if more time and resources were available.

6 Running Project

Project Structure

- CartPoleExperiment
  - DDQN
    - DDQN.py
    - targetModels
  - DQN with Replay-Buffer
    - DDQN.py
    - targetModels
  - Standard DQN
    - DDQN.py
    - targetModels
- MountainCarExperiment
  - DDQN
    - DDQN.py
    - Standard-DDQN.py
    - targetModels
  - DQN with Replay-Buffer
    - DDQN.py
    - targetModels
  - Standard DQN
    - DDQN.py
    - targetModels

To train any of the available models for any of the two environments, simply run the corresponding file within the algorithm folder. Before running, please do install the following libraries:

Required Libraries

- torch
- gym
- numpy
- tdqm
- wandb (may have to initialize with personal github account)

7 References

[1]: https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf
[2]: https://www.endtoend.ai/assets/blog/paper-unraveled/cer/original.pdf
[3]: https://arxiv.org/pdf/1509.06461.pdf
[4]: https://gym.openai.com/envs/classic$_{c}ontrol$
[5] $: https : //gym.openai.com/envs/CartPole - v1/$
[6] $: https : //gym.openai.com/envs/MountainCar - v0/$