CS 7624 Project 2 – Lunar Lander DQN

Arjun Chintapalli[1]
1 Department of Computer Science, Georgia Institute of Technology, arjun.ch@gatech.edu
Github Repo: https://github.gatech.edu/achintapalli3/CS7642_Project2

ABSTRACT

This paper documents the results of implementing a Deep Q Network [DQN] to solve the popular Atari Lunar Lander game provided by the OpenAI Gym package (specifically the 'LunarLander-v2' environment). The DQN for this project was implemented utilizing Mnih et al.'s groundbreaking paper "Playing Atari with Deep Reinforcement Learning" (2013) [1] with the added inclusion of a soft update procedure as described in "Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning" (2015)[2]. The implemented DQN utilizes a neural network that takes in the observable states and outputs the Q values of the possible actions in order to finally determine the optimal action for an agent to take at a particular state. Various experiments were then conducted on this DQN to try to improve performance such as the varying the discount factor, tau, the number of hidden layers, and implementing Combined Experience Replay.
Key Words: CS 7642, Project 2, Lunar Lander, Deep Q Network

1. INTRODUCTION TO LUNAR LANDER

The Lunar Lander game involves attempting to land a rocket ship on to the landing pad always at coordinates (0,0). This environment necessitates an eight-dimensional continuous state space represented by {x-coordinate of lander, y-coordinate of lander, velocity in x direction, velocity in y direction, angle of the lander, angular velocity, if left leg on the ground, if right leg on the ground} and a four dimensional discrete action space represented by {do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine}. At each timestep, if a lander moves away from the landing pad then it receives negative reward, and if the main engine is fired then -.3 points per timestep. At the terminal timestep, the reward for a successful landing can range from 100 – 140 points and for crashing is -100 points. The game terminates either crashing or landing. A DQN was used to solve the problem by achieving a score of 200 points or higher on average of over 100 consecutive runs.

2. INTRODUCTION TO DQN

DQN's are a class of reinforcement learning algorithms (specifically model-free Q Learners), which learn the Q(states,actions) function utilizing a neural network.

Q Learning involves constructing a table of dimensions *states* x *actions* such that the value at each index represents the quality, a proxy for total expected future rewards, of taking that action at that state. The Q function is initialized, and then updated during each step *t* of training till the terminal state as follows:

$$Q^{new}(s_t, a_t) = (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a)\right) \quad (1)$$

Where α is the learning rate such that $0 < \alpha \leq 1$ and determines the degree of replacement of prior information at this (state, action) with information from the current timestep. Where γ is the learning rate such that $0 \leq \gamma \leq 1$ and determines the extent that rewards at the future timestep is discounted relative to the current timestep.

Although the base Q Learning algorithm is guaranteed to have convergence of Q values to the optimal Q values, but only if all state action pairs are visited infinitely often. This proves to be a challenge with the Lunar Lander environment because of the 8-dimensional continuous state space of the problem. Thus, for a continuous state space environment a function approximation algorithm such as DQN is necessary.

The DQN method utilizes a neural network to approximate the Q function, where the network's input nodes map to the continuous state space and the output nodes map to the discrete action space.

$$\hat{Q}(s, a) = [F(\theta)](s, a) \tag{2}$$

Here, *F* represents the neural network mapping to the approximated Q function, and θ represents the weights of the neural network. Thus, at a particular state this model can be used to retrieve the Q values for each discrete action and use a policy from these values to determine the optimal action to take.

Each of the weights of the DQN are updated using a Huber Loss as recommended in Piazza. The DQN is updated at each timestep utilizing experience replay and soft updates because otherwise DQN's can become unstable or divergent. Instead of updating the Q Table at each time step as mentioned before, at each timestep the network is fit with a random batch of prior timesteps thereby implementing experience replay. This was done in order to ensure stability of the neural network as otherwise the network will overwhelmingly focus on recent episodes. To further ensure stability of the network, a soft update of the target weights was implemented such that two networks, a source network $F(\theta)$ and a target network $F(\theta')$, are maintained. The target network is used to calculate the target values for the loss function. Only the source network's weights get updated with each batch and the target network's weights are then updated proportional to τ:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \tag{3}$$

To determine what action to take at each time step, the epsilon-greedy policy was taken such that if a randomly generated number is less than epsilon than a random action is taken, otherwise the DQN is used to predict the optimal action to take, and the epsilon is decayed at the end of each episode by a decay rate past an initial number of episodes. This approach is necessary to balance the exploration vs exploitation dilemma.

A base case implementation was derived with the parameters: a 2 hidden layer network with 40 neurons each and relu activation, $\tau = .01$, replay memory of size of 50,000, $\gamma = .99$, initial epsilon of 1 with an epsilon decay rate of .975 after 50 episodes, and an $\alpha = .002$ with a decay rate of .00001 using the Adam optimizer. The below results were achieved for this case with a mean test reward of 208.7 with the Testing Rewards graph further indicating that the lander has managed to land the Lunar Lander in every test. Since this problem is considered solved with a score of 200 points, this implementation can be considered to have effectively solved the environment!

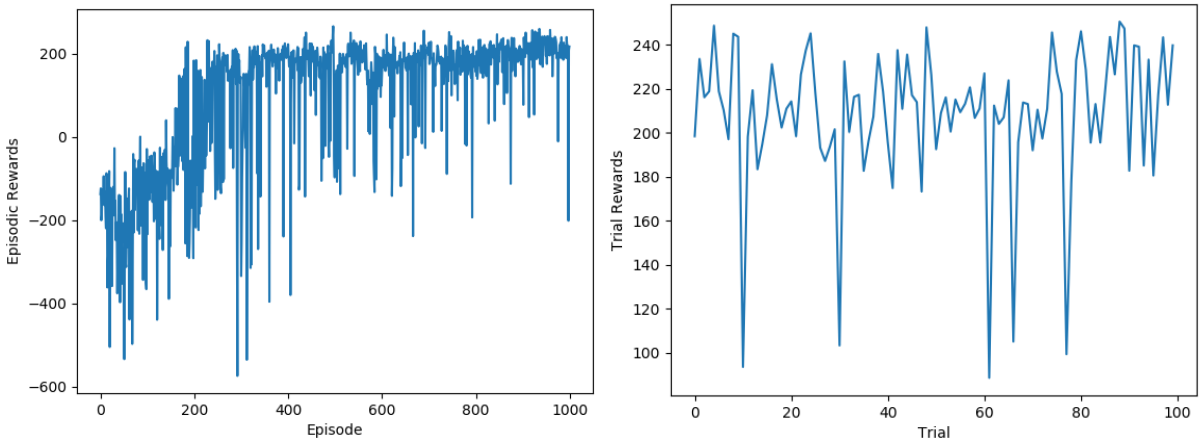

Figure 1: Training and Testing Rewards per Episode for Base Case [Training on Left and Testing on Right]

From this base implementation other parameters were varied to derive their relationships to solving this problem as well as to improve the base case implementation. For all the experiments and base

case, both the random number generators and the AI Gym environments were seeded with the same value to ensure reproducibility of experiments.

3. EXPERIMENT 1: VARYING GAMMA (DISCOUNT FACTOR)

Gamma was varied from .9, .99 and .999 with the .99 experiment being the base case described above.

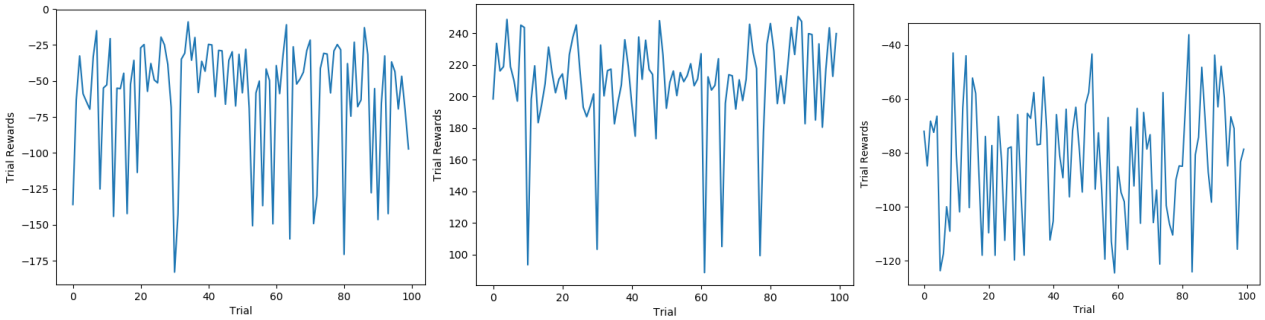

Figure 2: Test Rewards for Gamma = .9,.99,.999 with mean rewards [-60.1, 208.7, -82.8] respectively

The value of γ determines the sensitivity of the DQN to later rewards as opposed to earlier rewards. In this particular Lunar Lander game, the only positive reward comes at the end of an episode when the Lunar Lander successfully lands. Thus, the DQN needs to have a high as possible discount factor to maximize sensitivity to this final reward and make the DQN far-sighted. But counterbalancing this, discount factors just shy of 1 exacerbate the instability and error propagation especially when using neural networks to approximate the Q function. The results of this experiment validate these conclusions because when $\gamma = .999$ and $\gamma = .9$ the DQN performs poorly, with high variability and never get positive rewards indicating that these implementations have not learned to land the lander under any scenario. Presumably in the $\gamma = .9$ case, even when the DQN landed the lander in a training episode, the positive rewards achieved then were discounted such that the approximated Q function barely budged and the DQN achieved very little learning and data efficiency. At the other end of the spectrum, with the $\gamma = .999$ case the DQN became unstable as rewards at all timesteps were effectively the same leading to drastic changes in the Q function with each weight update.

4. EXPERIMENT 2: TAU

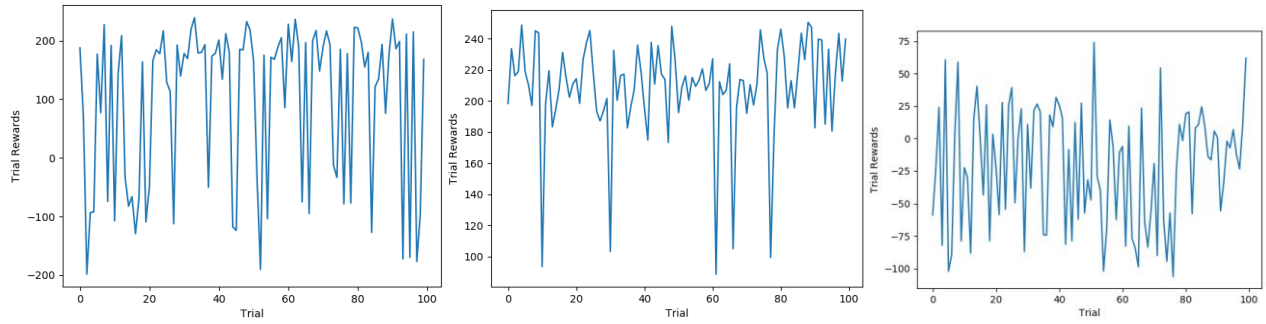

Figure 3: Test Rewards for Tau = .001,.01,.1 with mean rewards [96.7, 208.7, -21.4] respectively

With the soft update implementation, τ determines how fast the target networks, used to calculate the target values for Loss Estimation, track the weights of the learning network. Low τ values would increase the stability of the algorithm but reduce the learning rate by delaying the propagation of value estimations. High τ values on the other hand can lead to drastic changes in the target network and values, which then compound to lead to drastic changes in the Loss Estimation and weight updates of the learning network.

Thus, to determine the exact nature of this relationship the hyper parameter τ was also varied. A review of the literature determined that this hyperparameter is critical to ensuring the stability and convergence of the DQN with a commonly referenced value of .01 (the base case implementation). These results demonstrate that although the two experimental DQN's using other τ values did not diverge or

become unstable, as indicated by their ability to land the rover in a significant portion of the tests, they did perform dramatically worse than the base implementation using $\tau = .01$.

5. EXPERIMENT 3: 1 HIDDEN LAYER VS 2 HIDDEN LAYERS

As recommended by the TA's on Piazza, a neural network with only one hidden layer with less than 100 neurons (60 in this experimented) was tested to see if performance would improve.

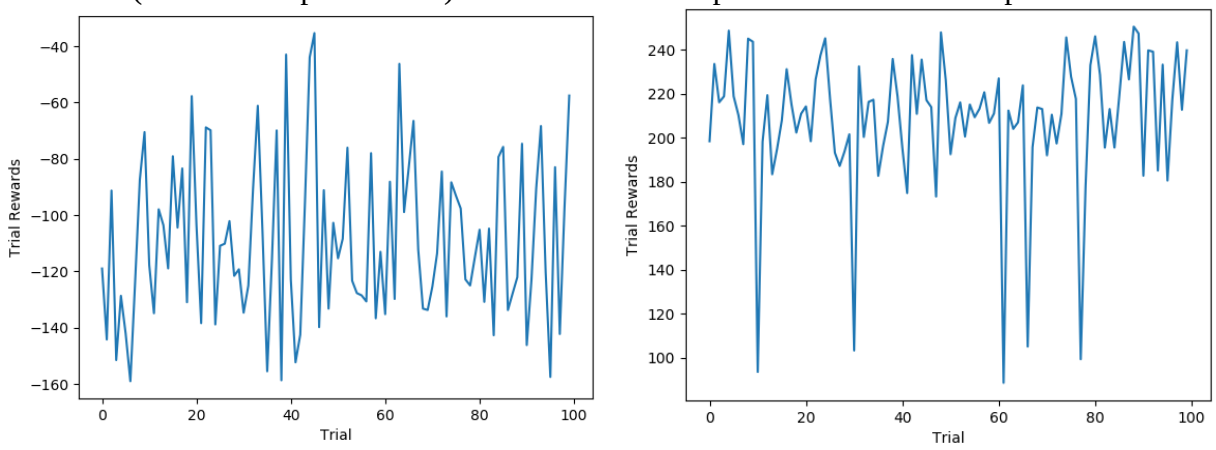

Figure 3: Test Rewards for 1 Hidden Layer (60 neurons) and base case with mean rewards [-108.9, 208.7]

Since the 1 layer hidden neural networks was tested with the same parameters as the base case with 2 layers, the poor results are expected. Although this 1 hidden layer DQN would have also probably worked, the hyper parameters would need to be retuned for this architecture.

6. COMBINED EXPERIENCE REPLAY

Combined Experience Replay was implemented such that the output of the latest timestep is added to the batch to be updated as described in "A Deeper Look at Experience Replay" by Zhang and Sutton [3]. This procedure is used to improve performance and mitigate the effects of choosing a sub-optimal memory buffer size with little additional performance cost.

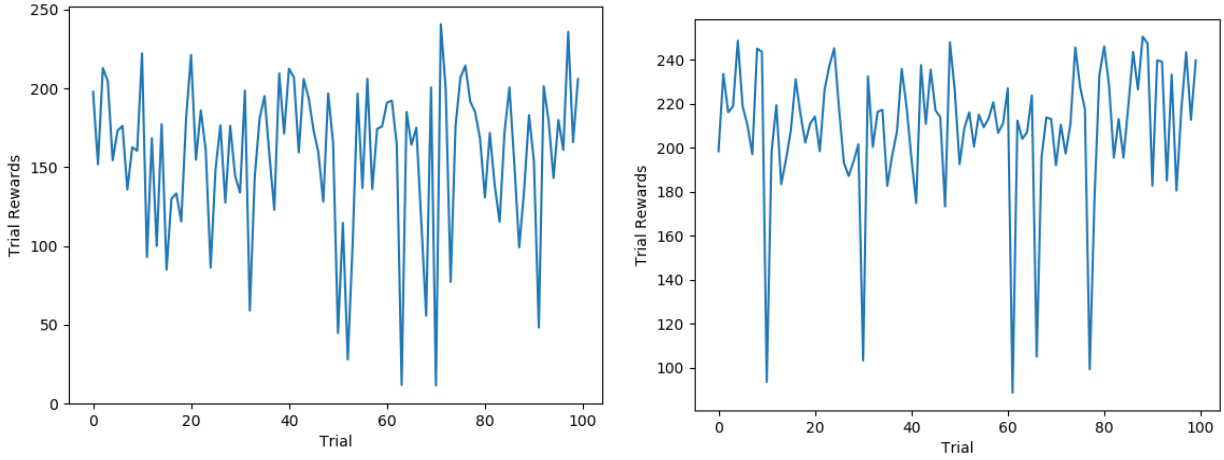

Figure 3: Test Rewards for CER and base case with mean rewards [157.1, 208.7]

The results demonstrate that although implementing CER leads to a working DQN implementation with the DQN managing to land the lander every time, the performance as measured by average rewards in the testing set is still worse than the base case. Zhang and Sutton's implementation of CER performed better than the conventional DQN for a similar buffer size as implemented here (10000) for the Lunar Lander environment. Thus these results look at first contradictory but all their implantations always had negative total returns indicating DQN's that have learned poorly. More experiments similar to Zhang and Sutton with varying the buffer size of both the conventional and CER DQN's are required to achieve conclusive results on if CER can improve performance.

7. CONCLUSION

The experiments made it very clear how important the fine tuning of the hyper parameters of a DQN is. Thus, even with a logically correct algorithm a programmer might not be able to solve an environment without repeated trial and error fine tuning the parameters.

From this set of experiments, the base case implementation seems to have been the best implementation tested. But as this implementation itself was the result of countless undocumented experiments, this result was to be expected. Although changing the discount factor more effected the testing outcome than changing τ, this could be a result of the range of parameters to be tested. Despite the low resolution in testing the hyper parameters, the trends emergent in the experiments confirm what has been described in the DQN literature.

If I had more time, I would have tested the hyperparameters with more values and higher resolution, but each experiment took a significant amount of time to compute. Additionally, getting a working base case implementation itself took a significant amount of time because I was unsure if my algorithm was wrong or if I just had the wrong hyper parameters.

REFERENCES

[1] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).
[2] Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." arXiv preprint arXiv:1509.02971 (2015).
[3] Zhang, Shangtong, and Richard S. Sutton. "A Deeper Look at Experience Replay." *arXiv preprint arXiv:1712.01275* (2017).