Lunar Lander DQN RL Agent

Git Hash: 321c7e3f1cd3b6e58cd20b2721a678794e8e23e4

Devanathan Ramachandran Seetharaman
CS 7642 - Reinforcement Learning
Georgia Institute of Technology
Atlanta,GA
rsdevanathan@gatech.edu

Abstract—**This project document details the implementation and analysis of the Deep Q-Network Reinforcement Learning agent to solve the Lunar Lander problem in OpenAI Gym.**

I. Introduction

The objective of the project is to build a **Deep Q-Network(DQN)** Reinforcement Learning Agent to solve the Lunar Lander problem from OpenAI Gym. This project particularly focuses on the implementation based on the paper [1], where the authors use the DQN RL agent to solve an atari game. In this project, I have used the "experience replay" and "target network" techniques proposed by authors in [1], to efficiently solve the reinforcement learning problems. The discussion focuses on the DQN algorithm, architecture of the DQN model, various hyperparameters used, and the convergence of the problem. The technical solution is implemented in python and uses Keras framework with Tensorflow backend to build and train the DQN agent.

II. Lunar Lander Problem

The Lunar Lander problem involves landing the spaceship on the lunar surface between the given flag poles. The spaceship is controlled by three throttles, left, right, and downward. The environment has an eight-dimensional state space $\left(x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, \text{leg}_L, \text{leg}_R\right)$ representing, horizontal and vertical position, horizontal and vertical speed, angle and angular speed and left right leg-ground contact variables respectively. And the environment has four possible discrete actions, "do nothing", "fire the left orientation engine", "fire the main engine" and "fire the right orientation engine". Rewards and Penalty is based on, the distance away/towards the pad, crashes/comes to rest, leg-ground contact, and the firing of engines. The final score of 200 or over for 100 consecutive runs is considered solved for the problem.

III. DQN RL Algorithm

This section summarizes the DQN method explained by authors in [1].

In [1], authors propose the Deep Q-Network method which combines reinforcement learning with the Artificial Neural network to solve the atari games. The method is suitable for problems that have high dimensional input spaces, which is the case with the most real-world problems. Lunar lander problem with the eight-dimensional input space will be a good use case to experiment with the DQN method. The action selection of the algorithm uses the ϵ-greedy policy. Since the DQN method is a non-linear approximation function, the authors have proposed multiple approaches to handle the instability associated with the non-linear approximation functions. Two key approaches that I have used from [1] are "experience replay" and "target network".

The experience replay technique stores the experience of the agent in each step into a memory. The state, action, reward, done flag, and next state are stored for each step. To avoid the memory overflow the total experience saved in the memory is capped at some value N. When the length of the memory exceeds the value N, the earliest experience is removed from the memory. In each step of the episode, experiences are randomly drawn from the memory with some batch size to train the DQN agent. Authors of [1] observe that this method would be better and more stable than online learning where each experience is individually considered. Also drawing random samples is more efficient than using the consequent samples which will be strongly correlated to each other. The model is trained using the drawn samples only after the length of the recorded memory reaches a certain threshold for the same reason to avoid the correlated samples in the smaller memory size.

The second technique used in the implementation is "target network". The implementation uses a separate network for generating the targets.. The architecture of the target network will be the same as the evaluation network. In specific intervals, the weights are copied from the actual network to the target network. Authors, in [1], notes that using a target network that has an older set of parameters will make the agent stable avoiding oscillations and divergence.

Also in the ϵ-greedy policy, instead of using fixed ϵ value, epsilon decay is used for the exploration-exploitation strategy. ϵ is started with the value of 1 to start with random actions, in the beginning, then gradually decreasing it to a very low value once the agent learns. This technique is used to ensure the exploration-exploitation balance during the training progress.

The total rewards are recorded for each episode as a sum of the rewards for each step of the episode. Along with the total reward, the average reward for the last 100 episodes is recorded to plot against the episodes to obtain a smoother curve.

Below is the detailed algorithm used for Deep Q-Network Reinforcement learning implemented for the Lunar Lander problem.

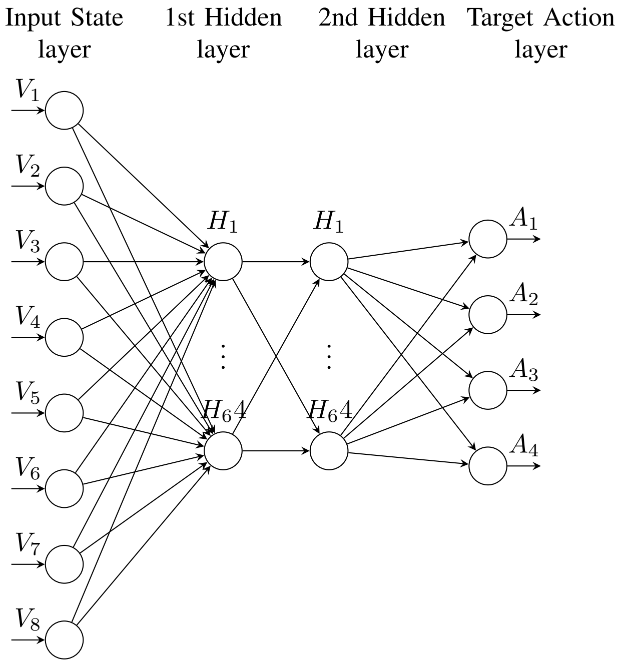
Algorithm 1 DQN Method

```
- Initialize Memory
- Create Evaluation Network
- Create Target Network
for each Episode do
    - Initialize State
    repeat
        - Take action based on $\epsilon$ - greedy
        - Observe reward, nextstate based on action taken
        - Record state,action,reward,nextstate and done flag in
          Memory
        if length of Memory is greater than N then
            - Draw samples from Memory with size of batch size
            - Set $y = r$ for terminal state
            - Set $y = r + \gamma \max(TargetNet(S_{t+1}))$ otherwise
            -Fit Evaluation Network with the drawn states and $y$
        end if
        -Update nextstate as state and continue
    until done
    - Record total reward and average reward for each episode
end for
-Plot and Evaluate the training rewards
```

.

IV. DQN Architecture

Below is architecture of Neural Network model used for both the evaluation network and the target network



The Neural Network model for DQN is created with two hidden layers with 64 neurons each. The input layer has eight neurons, one for each state-space variable. The output layer has four neurons for four possible actions.

The hidden layers use the "relu" activation function and the output layer uses the "linear" activation function to get the continuous values for output. "mean squared error is used as a loss function. ADAM optimizer is used for the training and the learning rate of $0.0001$is used. The smaller learning rate is chosen to ensure the convergence although it might also increase the execution time. The number of hidden layers, number of neurons, optimizer, and learning rate were chosen based on the multiple iterations of the experiment.

V. DQN Training Analysis

The DQN Reinforcement algorithm and neural network model explained in the previous two sections are implemented in python using the Keras framework. The parameters of the algorithm can be grouped into three high-level categories. Neural Network-related parameters, Memory-related parameters, and Algorithm-related parameters. My implementation and this document focus on hyper parameter tuning of Algorithm-related parameters. The parameters related to the other two categories are kept fixed and values for them are chosen mostly in line with [1].

Some of the key fixed parameters are initial epsilon($=$ 1), memory size($=$ 10000), minimum memory size($= 2 * batchsize$), target network update interval($=$ 1000), hidden layer count($=$ 2), hidden layer neurons($= [64, 64]$),learning rate(=0.0001) and optimizer(=$Adam$).

The implementation particularly focuses on three major Algorithm-related parameters, Batch Size, Gamma, and Epsilon Decay parameter. The range of values is tested for each of these variables and the best parameter is chosen. The hyperparameter tuning for each of these parameters is explained in detail in section VII, VIII, IX. This section and graphs focus on the final training once the hyperparameters are chosen, .i.e. run with best hyperparameters.

The best hyperparameters chosen are $BatchSize = 32$,$Gamma = 0.99$ and $EpsilonDecay = 0.998$.The final training is run for maximum of 2000 episodes and the convergence is chosen as consecutive rewards above 200 for the episodes.

The below graph shows the reward for each episode until covergence.

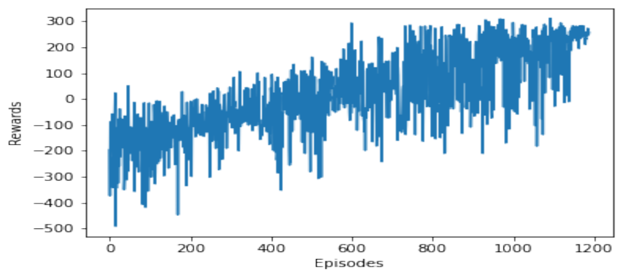


Fig. 1. Final Training - Reward

The above figure shows that the agent starts to converge to reward of 200 around 800 episodes. However, there is a strong oscillation initially. The rewards begin to stabilize around 1200 episodes. Still, there are some occasional oscillations in the training.

The below figure shows the smoothed curve with the Average reward, be calculating of mean rewards for the last 100 episodes. This plot clearly shows the convergence as the episodes reaches 1200.

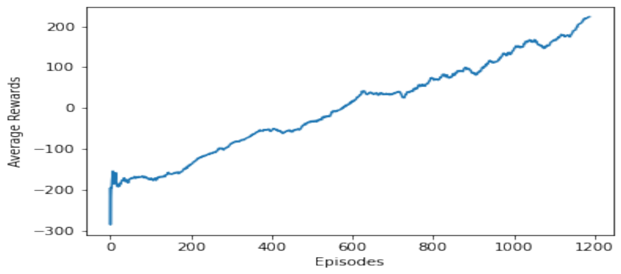


Fig. 2. Final Training - Mean Reward(Last 100 Episodes)

At the end of the training, the trained agent is saved to evaluate the trained agent.

VI. DQN Trained Agent

The trained agent from the previous section is run on 200 episodes to observe the performance. The below plot shows the performance of the agent. The agent gets the reward of more than 2000 for most of the episodes, however, there are few episodes where the rewards are low,sometimes even reaching 0. However these are very minimal and the reward is always in positive territory.

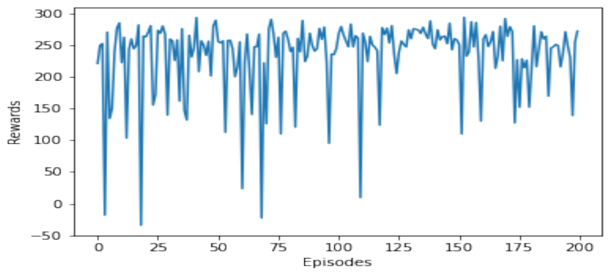


Fig. 3. Trained Agent - Reward

The below figure shows the smoothed curve of rewards for the trained agent and it can be observed that the average rewards are consistent for all the episodes. s.

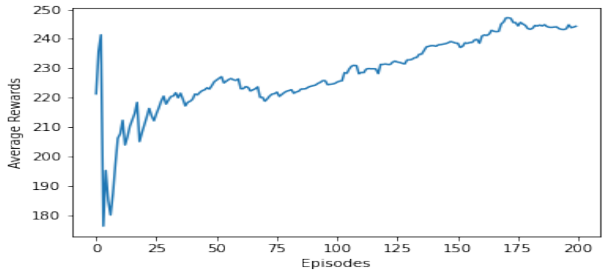


Fig. 4. Trained Agent - Mean Reward(Last 100 Episodes)

The average remains above 200 for most episodes and stabilizes around the reward of 240.

VII. Hyper Parameter - Batch Size

Batch Size is one of the key parameters used in the algorithm. It represents the number of samples used from the memory to train the Deep Q-Network in each step. Having a very low batch size caused an unstable and divergent agent whereas using a very high batch size increased the training time drastically. In this implementation, I experimented with three batch sizes, 4, 8, and 16. Each batch size is analyzed below.

The below figure shows the average reward plot for training with a batch size of 4. We could see that the training performs very poorly with a batch size of 4 and the average reward never crosses 200 and it is completely unstable.

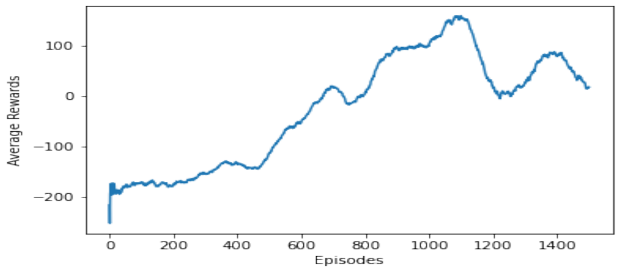


Fig. 5. Batch Size 4 - Mean rewards

The below figure shows the training reward plot for a batch size of 8. The average reward plot is much better than the batch size of 4, the training is still unstable and oscillating a lot in the middle of the training.

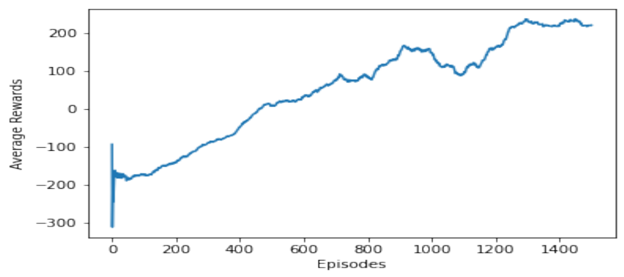


Fig. 6. Batch Size 8 - Mean rewards

The final experiment in the Batch size hyperparameter tuning is with the batch size of 16. The mean rewards look much better than the other 2 values and there is no major divergence and converges to reward above 200 in around 100 episodes.

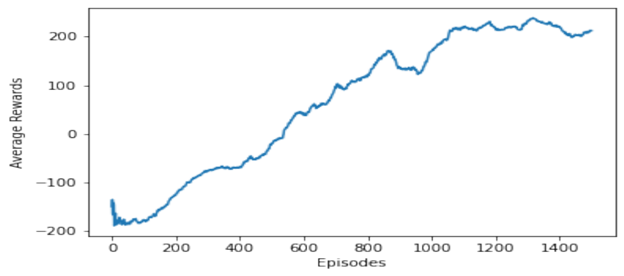


Fig. 7. Batch Size 16 - Mean rewards

Batch sizes higher than 16 were not evaluated due to the very slow run speed. Hence the batch size of 16 is chosen for the final training.

VIII. Hyper Parameter - Gamma

The second hyperparameter chosen for experimentation is Gamma. Gamma represents the discount factor in the Reinforcement Learning algorithm by which the future rewards are discounted. Three values for Gamma, 0.99, 0.9, and 0.75 are evaluated in training the agent.

The below plot shows the rewards for the Gamma value of 0.99 which is very close to 1(no discount). The plot converges finally for the reward above 200. However, there are a couple of major oscillations at the end. The hyperparameter tuning was run for the limited iterations due to the time taken and hence couldn't observe the stable episodes of the agent in the below plot.

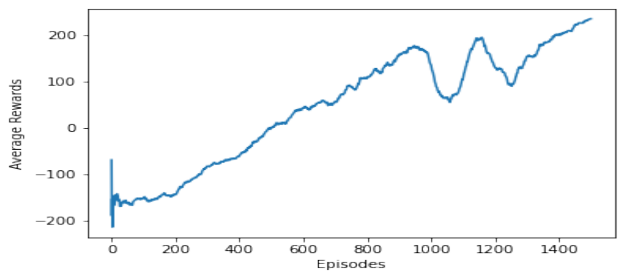


Fig. 8. Gamma 0.99 - Mean rewards

Below is the reward plot for the Gamma value of 0.90. The training rewards look very bad and never converges. The mean reward was never in the positive territory.

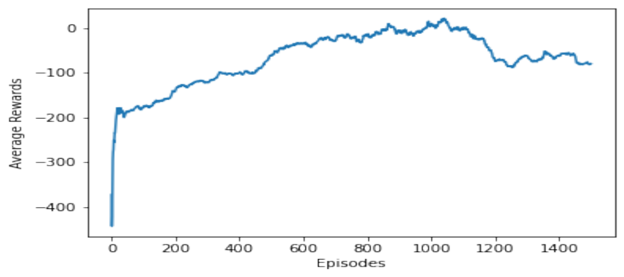


Fig. 9. Gamma 0.90 - Mean rewards

Below is the training plot for the Gamma value of 0.75. The plot again is poor and the average reward never crosses 50.

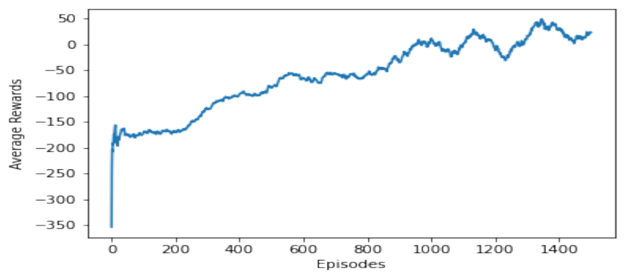


Fig. 10. Gamma 0.75 - Mean rewards

From the hyperparameter tuning experiments for Gamma values, the Gamma value of 0.99 seems to perform much better than the other 2 values. Hence Gamma value is chosen as 0.99 for the final training.

IX. Hyper Parameter - Epsilon Decay

The final hyperparameter tuned for DQN training is Epsilon Decay. Epsilon decay is the rate at which the epsilon value of ϵ - greedy policy is reduced. This variable controls the exploration-exploitation strategy as the training progresses.

The first decay value tested is 0.9998, which is very low decay. This value even after 1000 episodes would decay the epsilon value from 1 to just 0.8. Hence ϵ - greedy would use random action most of the time and hence the performance is very poor and never in positive territory.

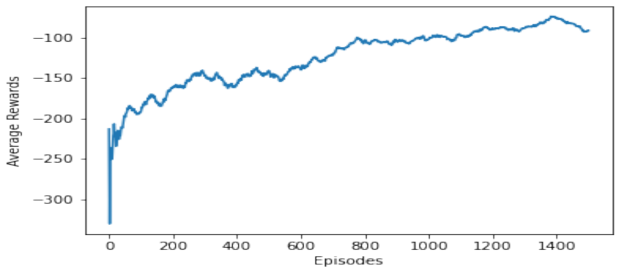


Fig. 11. Epsilon Decay 0.9998 - Mean Reward

The second decay value tested is 0.998 and below is the training plot for average rewards. The training converges and reaches 200 around 1000 episodes. There are some oscillations in the episodes but the model is relatively stable.

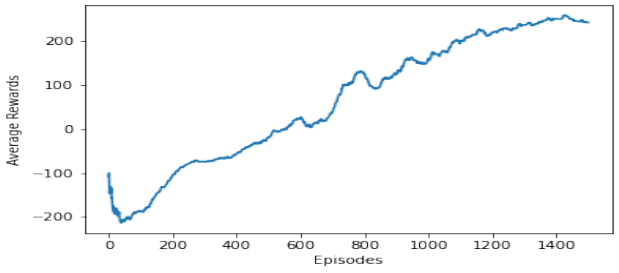


Fig. 12. Epsilon Decay 0.998 - Mean Reward

The final decay value considered is 0.98. The decay is rapid and reaches minimum epsilon early in training. This reduces the exploration in the training and the model is relatively unstable compared to the previous value of 0.998.

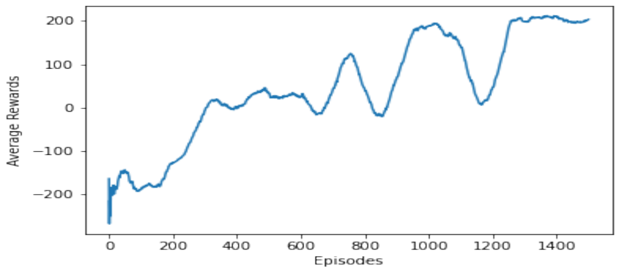


Fig. 13. Epsilon Decay 0.98 - Mean Reward

Out of the decay values tested, 0.998 performs better and hence chosen for the Final training. The Final training with best hyperparameters and the performance of Trained agent is already explained in detail in Section V and VI.

X. Observations and Challenges

One of the main challenges faced in the implementation is the performance and run time of the training. The complete training process took around 2 hours in the CPU machine. Hence I could not explore a wide range of values for hyper-parameters and limited to 3 values for each.

Another observation is, irrespective of experience replay and target net techniques, there were consistent oscillations and divergence even for the best hyperparameter and could not replicate the success of [1]. This forced me to change the convergence criteria as a reward of more than 200 for 25 episodes instead of 100 episodes as required.

XI. Conclusion

In this project, I implemented the Deep Q-Network reinforcement learning as proposed in [1] to solve the lunar lander problem from OpenAI gym. The implementation was able to achieve the convergence of consistent rewards above 200 although there were some oscillations observed.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518, 529–533.
[2] http://incompleteideas.net/book/RLbook2020.pdf