Project Report - Lunar Lander

Srinivas Rahul Sapireddy
ssdx5@umsystem.edu

Abstract

This report briefly introduces the Lunar Lander Environment from OpenAI Gym. Solving this Lunar Lander using traditional Tabular Methods is practically challenging and expensive due to its large state space and complex reward structure. So, we turn to Function Approximations - An idea central to solving complex Reinforcement Leaning problems such as Lunar Lander through generalization of state space into a lower dimensional feature space. We specifically discuss Non-Linear Function Approximation in detail. Then we discuss the role of Neural Networks as a Non-Linear Function Approximator and formulate a solution for Lunar Lander though a Deep Neural Network. Specifically, we will discuss the DQN algorithm introduced in (Mnih and et al 2015) paper and its application to solving Lunar Lander Environment using Deep Q-Network. DQN algorithm is know to be optimistic and it overestimates the action value function, to address this shortcoming we will briefly review the Double DQN Algorithm from (Hasselt, Guez, and Silver 2015) paper. Finally, a soft update optimisation is used for updating target network weights instead of a hard update proposed in (Mnih and et al 2015) paper for improving the learning performance on the Lunar Lander task and the results of this experiments are shared in detail including the hyper parameters used for training and the role of these hyper parameters in the learning process.

1. Introduction

OpenAI Gym is a toolkit for building and comparing Reinforcement Learning algorithms in a simulated environment. Lunar Lander is one such environment where the goal is to land a spaceship on the landing pad. Lunar Lander is a Continuous State Space Markov Decision Process(MDP) with states, actions and rewards described as below.

States Lunar Lander has 8-dimensional state space vector, with six continuous states and two discrete states as below

$$(x; y; \dot{x}; \dot{y}; \vartheta; \dot{\vartheta}; leg_L; leg_R)$$

Where state variables x and y are the current horizontal and vertical position of the lander, $\dot{x}$ and $\dot{y}$ are the horizontal and vertical speeds, ϑ and $\dot{\vartheta}$ are the angle and angular speed of the lander and leg_L and leg_R are the discrete binary values to indicate whether the left leg and right leg of the lunar lander are touching the landing pad.

Actions It has four possible discrete actions - do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine to control the lander.

Rewards For moving from top of the screen to the landing pad the lander has following rewards structure. If the lander moves away from the landing pad it is penalized the amount of reward that would be gained by moving towards the pad. An episode is finished if the lander crashes (or) comes to rest, receiving -100 or +100 points, respectively. Each leg-ground contact is worth +10 points. Firing the main engine incurs a cost of -0.3 points, firing the orientation engines incur a cost of -0.03 point for each occurrence.

As we can infer from the above description of the Lunar Lander, it is a complex MDP with large state space and complex reward structures. For this reasons, Employing a Tabular Method to solve this MDP is practically hard by even discretizing the continuous state space into bins, but its solvable with clever optimisations leading a complex solution. Instead, we will be exploring a much simpler solution to solving the Lunar Lander Environment using Function Approximation techniques. Specifically we will be discussing Deep Neural Nets as function approximators to solving this MDP.

2. Generalization

Since, Complex MDPs suffer from the curse of dimensionality and using standard tabular methods to solve them is expensive in terms of compute resource to build huge tables for computing the Expected Return Value Functions the next best thing that we could do is to safely approximate the Value Function. This process of approximating the Value Function is known as Generalization. Different approaches were explored in the literature on generalization, a few common approaches are listed below.

- Linear Function Approximations
- Non-Linear Function Approximations
- Hybrid of Dynamic Programming and Function Approximation (Boyan and Moore 1995)

In this paper, we will be focusing on Non-Linear Function Approximations to solve the Lunar Lander Environment. Specifically we will be using Deep Neural Networks as the means to solve it.

3. Non-Linear Function Approximations

The main drawback of Linear Function Approximation and other methods compared to Non-Linear Function approximation, such as the Deep Neural Networks, is the need for hand picked features to approximate the High Dimensional State Space to lower dimensional Feature Space, which requires domain expertise, making the job to build Reinforcement Learning Agents harder.

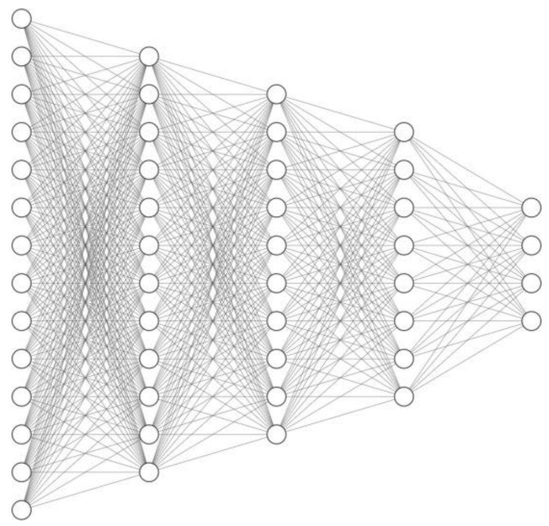


Figure 1: Fully Connected - 5 Layer Deep Neural Network

For the fore-mentioned reasons, we will be exploring the Deep Neural Networks as a Non-Linear Function Approximator for solving the Lunar Lander Environment. In this case, we have replaced the Q values table with a Fully Connected 5-Layer Deep Neural Network to approximate the Value Function for the Lunar Lander. A schematic of such Fully Connected Deep Neural Network is shown in Fig.1 which we refer to as Deep Q-Network (DQN). For solving the Lunar Lander, We create an agent using this network and train it to approximate the Q-Value Function. This agent interacts with the simulated gym environment and collects the samples for learning the Q-Value Function.

4. Deep Q-Networks

4.1 DQN Reinforcement Learning is known to be unstable when a Non-Linear Function Approximator such as a Deep Neural Network is used to represent the Action Value Function (or) Q-Function. This instability comes from causes such as - Correlations in the observation sequences, Small updates to Q-Function in feature space may significantly change the policy and therefore change the data distribution, and the correlations between the Action Values (Q) and the Target Values $r + \gamma max_{a'} \; Q(s', a')$.

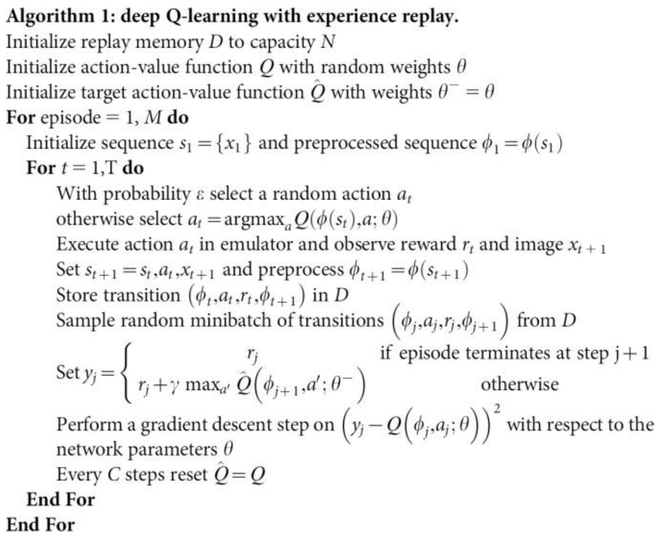
To address these instabilities (Mnih and et al 2015) paper introduces a Q-Network and a variant of Q-learning, which uses two key ideas. First, a mechanism called experience replay is used to randomize over the data to remove correlations in the observation sequence. Second, an iterative update to adjust the Action Values (Q) towards Target Values periodically is used to reduces the correlations with the target data distribution.

DQN Algorithm introduced by (Mnih and et al 2015) parameterizes an approximate value function Q(s,a;ϑ_i) using the Deep Neural Network, in which ϑ_i are the weights of the Q-network at iteration i. To perform experience replay the agent stores the experiences at each time step t in a fixed size buffer called dataset D. For Q-learning updates, minibatches of these experiences are sampled uniformly at random from the previously stored experience. The Q-learning update at iteration i uses the loss function in Fig 2 below for the learning process.

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

Figure 2: DQN Loss Function

Where γ is the discount factor, ϑ_i are the weights of the Q-network at iteration i, ϑ_i^- are the weights of the target network which are updated periodically to ϑ_i. The algorithm presented by (Mnih and et al 2015) for training the DQN Agent is show in Fig 3.



```
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 $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
     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
```

Figure 3: DQN Algorithm

4.2 Double DQN DQN Algorithm discussed in the previous section is proven to overestimate the error substantially by (Hasselt, Guez, and Silver 2015) paper. The fundamental problem with DQN (Mnih and et al 2015) is its systematic overestimation of Q values initially investigated in (Thrun and Schwartz 1993) paper. This over estimation is due to the max operation taken over all Action Values. Lets look at a simple example of such a problem - Let's say the expected value of a dice roll is 3.5, but if we throw the dice 100 times and take the max over all throws, we're very likely to get a value > 3.5. Its still okay if all values are equally overestimated, since only the difference between the Q values matters. But *max* is a non-linear operation making this over estimations non-uniform and leading to slowness in the learning process.

This overestimation is solved with Double Q-Learning by decomposing the *max* operation in the target into action selection and action evaluation phases. Here the target network in the DQN architecture is used as a second value function and either network is selected with 0.5 probability for Q updates. With this approach we effectively end up with two different function approximators that are trained on different samples and one is used for selecting the best action and other for evaluating the value of this action. since these two approximators have seen different samples, it is less that they overestimate the same action. Hence the name Double Q-Learning.

4.3 DQN with Soft Updates The (Mnih and et al 2015) paper proposes copying over the weights of the Q-Network to the Target network every C steps. This update is referred to as polyak/hard update in the literature. To mitigate the overestimation problem discussed in previous section an alternative soft update approach is explored and successfully implemented in solving Lunar Lander. This approach reduces the overestimation error by using slow weights propagation from the Q-Network to Target Network by using the averaging mechanics. The rate of flow of weights from Q-Network to Target Network is controlled using the parameter τ as given in the following equation

$$Q_{Target} = Q * \tau + (1 - \tau) * Q_{Target}$$

6. Experiments & Results

All the experiments have used DQN in solving the Lunar Lander with variations in different aspects of the DQN such as the number of layers in the Q-Network, The Size of the replay buffer, Training Batch Size, ϵ decay rates for trade off between exploration and exploitation, τ for weight propagation control, early stopping of training. To improve the performance of the agents training, i have pruned the low scoring episodes by capping the number of steps in any given episode at 1000 steps. For the experimental purpose the Q-Table is replaced with a 5 layer deep Fully Connected Neural Network with 256 nodes in the input layer, 224 nodes in 3 hidden layers and 8 node in the output layer.

6.1 Training Rewards For Each Step First of all, lets look at the rewards obtained at each training episode with and with out early stopping of training. From Fig 4 and Fig 5 it is evident that the Lunar Lander is solved initially in the range of 300+ episodes and we are consistently averaging above 200+ score after about 500+ episodes. Key conclusion to note here are as follows. First, The network has converged and it has not run into any problems such as catastrophic forgetting well beyond the 500+ episode range which we can observe from Fig 5 where training was extended to 1000 episodes. Second, We are observing few random drops in score post 500+ episode mark and this observations are seen because the agent is exploring a completely new random state action pair that it hasn't seen before with bad consequences. We start to see this random drop reduce with more training towards the end of the graph in the range to 800 to 1000 episodes which is expected from the agent as it learn from its mistakes.

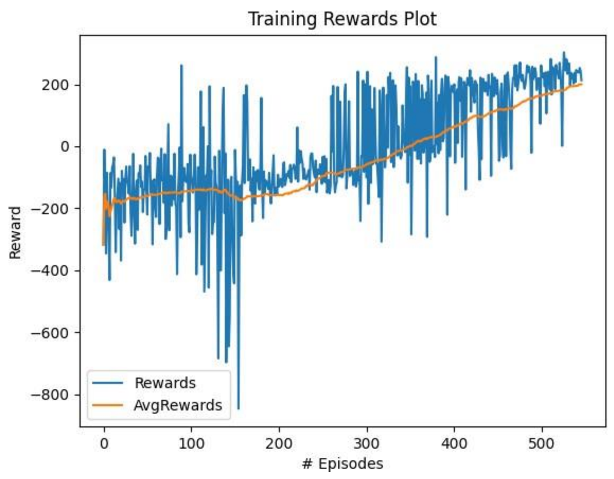


Figure 4: Rewards obtained at each training episode with early stopping

6.2 Consecutive Rewards For 100 Episodes Now, lets look at the rewards per episode for 100 consecutive episodes of the agent with and with out early stopping. From Fig 6 and 7 we can observe that the DQN Agent has generalised and learnt to land the Lander on the pad in both the cases. But in case of early stopping we see relatively more bad episodes scoring lower than 200 compared to the agent trained for longer duration of episodes. The reason for this behaviour is that with more episodes of training the non-early stopped agent had a chance to look at more bad samples and learn from its mistakes, leading to better performance.

6.3 Effect of Hyper Parameters In this section we will look at the effect of ϵ, τ, Replay Memory Size (D) hyper parameters on the training performance.

From Fig 8 it is evident that ϵ decay has significant impact on the learning process. Higher the epsilon decay mul-

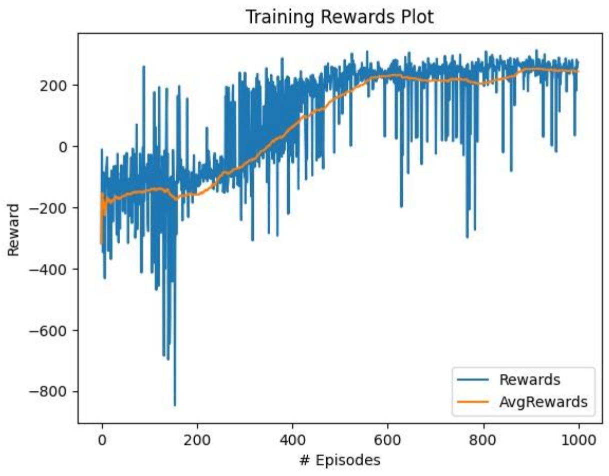


Figure 5: Rewards obtained at each training episode without early stopping

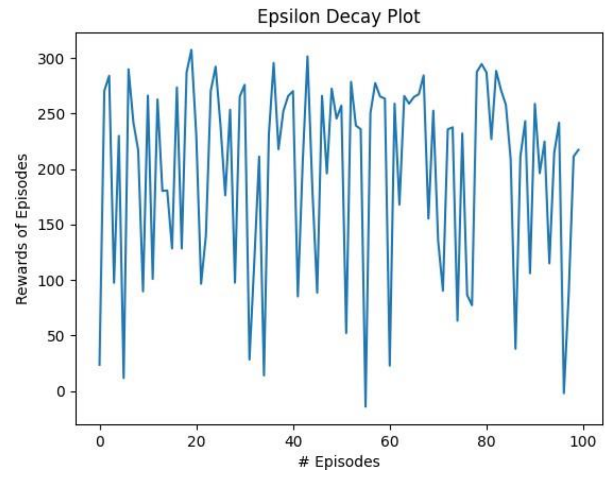


Figure 6: Reward per episode for 100 consecutive episodes with early stopping

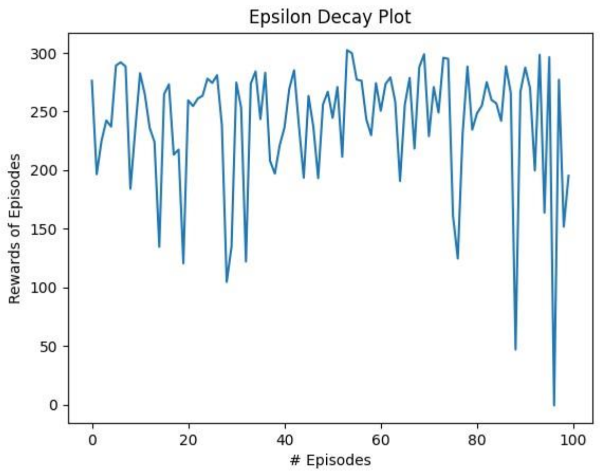


Figure 7: Reward per episode for 100 consecutive episodes without early stopping

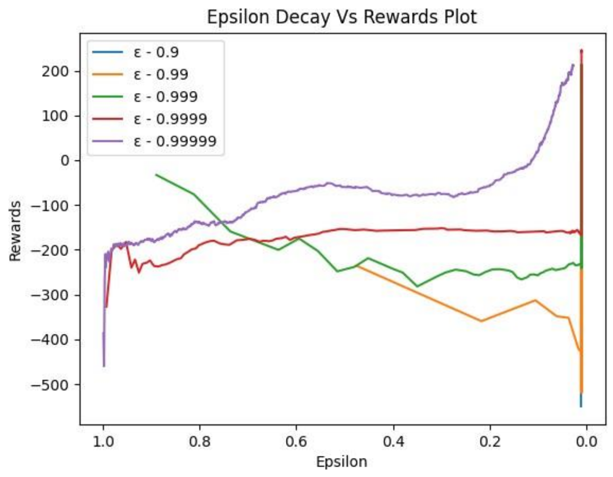


Figure 8: Reward Performance for different values of ϵ

tiplier, better the learning performance. Intuitively, when the epsilon decay multiplier is higher it allows for the agent to explore instead of exploiting the known best values. Therefore it allows the agent to visit better state and learn better values.

Weight propagation factor τ, results are presented in Fig 9. A value of 1 for weight propagation factor means a hard update of weights every C steps similar to DQN. When 10 is used as weight propagation factor we are amplifying the weights significantly leading to heavy overestimation error and making the agent learn poorly. We can observe that the values of 0.1, 0.01 has good learning performance compared to the value 1. In case of 0.001, 0.0001 we are observing the performance degrade again as the updates are done very slowly. Overall the weight propagation factor is optimal between 0.1 to 0.01.

Finally, lets look at the Replay Memory Size (D) hyper parameter results. From Fig 10 there is no clear winner in terms of learning performance, all the replay buffer sizes from 10 to 1 million seems to have similar learning performance with minor differences. Ideally, one would expect this parameter has an impact on the learning performance since this controls the sampling probabilities for data set and how old data samples are used for sampling.

6.4 Pitfalls Searching for right set of hyper parameters is a tedious task. An unguided tuning of parameters is not recommend while tuning the algorithm for best results. Instead of tweaking the hyper parameters in small amounts its better to take about 4-5 values for each hyper parameters which

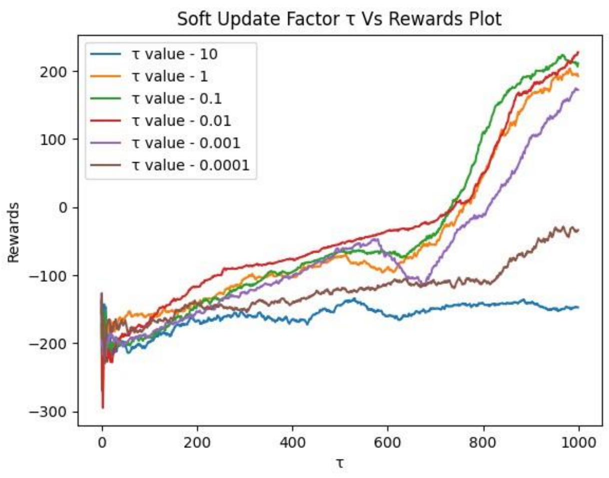


Figure 9: Reward Performance for different values of τ

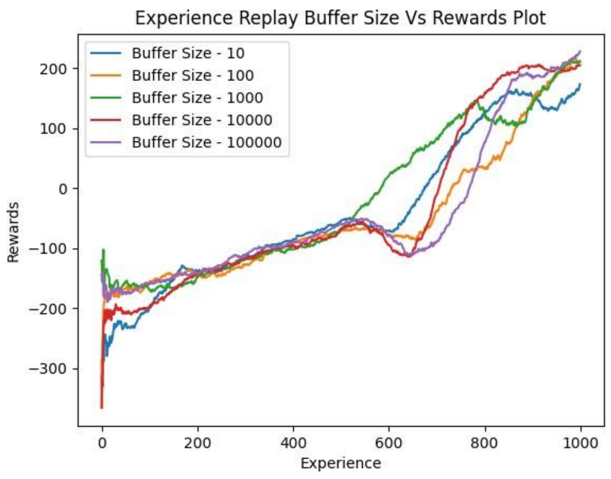


Figure 10: Reward Performance for different values of D

are an order of magnitude apart and run the experiments with those values. This approach will help to get the direction of improvements for the hyper parameters.

6.5 Worked Best & Didn't work DQN algorithm with Soft Updates worked best with τ value in between 0.1 to 0.01 and ϵ value of 0.99999. Replay Memory Tuning didn't work as per expectation.

6.6 Next Steps Looking at the result of the Replay Memory Size (D) hyper parameter, one key idea to explore is, if replay buffer size and the training batch size simultaneously has any correlation in terms of learning performance. Next, it would be interesting to compare and quantify the learning performance gains of Double DQN over DQN. Finally, it would be interesting to explore other hyper parameters such as C, Learning Rate, Training Batch Size.

7. Conclusion

The experiments conducted as part of this project solved the Lunar Lander Environment using DQN Algorithm with Soft Updates. It is observed that the weight propagation parameter τ and weight decay parameter ϵ greedy has significant impact on the learning performance.

References

[Boyan and Moore 1995] Boyan, J. A., and Moore, A. W. 1995. Generalization in reinforcement learning: Safely approximating the value function.

[Hasselt, Guez, and Silver 2015] Hasselt, H.; Guez, A.; and Silver, D. 2015. Deep reinforcement learning with double q-learning.

[Mnih and et al 2015] Mnih, and et al. 2015. Human level control through deep reinforcement learning.

[Thrun and Schwartz 1993] Thrun, S., and Schwartz, A. 1993. Issues in using function approximation for reinforcement learning.