Project 2 - Lunar Lander

*Hash for latest commit to the GitHub repository - 2cc8dd5ebceaa44bdcaaa7aa9596c52c500e9fd3

Prithi Bhaskar

pbhaskar8@gatech.edu

***Abstract*—This report aims to describe the agent developed and trained for solving the "Lunar Lander- V2" problem found in OpenAI gym. It also discusses the effects of tuning various hyperparameters on the performance of the agent. In addition, it also discusses the various approaches and models considered and evaluated during the process of building the agent.**

I. Lunar Lander - Goals and Constraints

The objective of the problem is to successfully land the lunar lander on the landing pad, which is always at coordinates (0,0). The state space is represented as an 8-dimensional vector - (x, y, x', y', θ , θ', leg_L, leg_R). The first two numbers represent the horizontal and vertical coordinates of the lander, the third and fourth represent the horizontal and vertical speed and θ and θ' are the angle and angular speed of the lander. leg_L and leg_R are binary values to indicate whether the left leg or right leg of the lunar lander is touching the ground. The action space is discrete and the actions available are: do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine.

Reward for moving from the top of the screen to landing pad and zero speed is about 100 to 140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points.

II. Policy Gradient Method

After considering various methods/algorithms available for function approximation, policy gradient method seemed an efficient algorithm as it has several advantages over the other function approximators. One of the main advantages is that parameterizing policies according to a soft-max distribution in action preferences enables the selection of actions with arbitrary probabilities. In problems with significant function approximation, the best approximate policy may be stochastic instead of a deterministic one. Hence REINFORCE: Monte Carlo Policy Gradient(a variant of Policy gradient) was implemented and used to solve the lunar landing problem.

Unlike Q-Learning, where the optimal action values are learnt first and then the optimal policy is derived from the values, Policy Gradient methods search directly for an optimal policy. The aim here is to learn a parameterized policy that can select actions without consulting a value function. Policy gradient algorithms achieve this by an update rule involving a scalar performance measure J(θ). The update is based on the gradient of the performance measure with respect to the policy parameter. The update rule is given by:

$$\theta_{t+1} = \theta + \alpha \nabla J(\theta_t) \tag{1}$$

where $\nabla J(\theta_t)$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument θ_t.

A. Policy Gradient Theorem

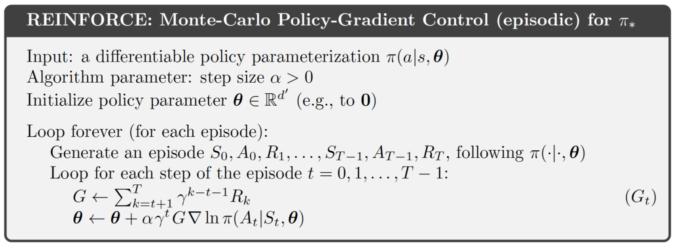
If the performance measure is defined as the value of the start state of an episode, the gradient of the performance measure can be expressed as:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{2}$$

where the gradients are column vectors of partial derivatives with respect to the components of θ, and π denotes the policy corresponding to parameter vector θ. μ represents the distribution of the state space.

B. REINFORCE: Monte Carlo Policy Gradient

REINFORCE (Monte-Carlo Policy Gradient) is a member of the class of Policy Gradient Algorithms. It uses the update rule shown in (1) along with a simple approximation - update at time t involves just $_t$, the one action actually taken at time t instead of involving all the actions as shown in (2).



```
REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for $\pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$
```

Figure 1. Monte Carlo Policy Gradient Algorithm

C. Experiments and Results

Using Neural network weights in place of θ in the algorithm shown in Figure (1), a neural network involving multiple hidden layers was trained according to the algorithm. The product of the discounted episodic return G and the log probability of the action chosen at time T was used as the loss function to be minimized by the network. Since neural networks get trained by minimising the loss function and here the expected reward is used as the loss function, the product was multiplied by a factor of -1 in order to maximize the rewards.

A network with two hidden layers, each consisting of 128 nodes was used to learn lunar landing. Adam optimizer was chosen with a learning rate of 0.001. Rewards were discounted by a factor of 0.99. The network was trained using 3000 episodes generated by interacting with the Lunar Lander environment and the total rewards earned in an episode was plotted for each of the 3000 episodes(shown in Figure 2).

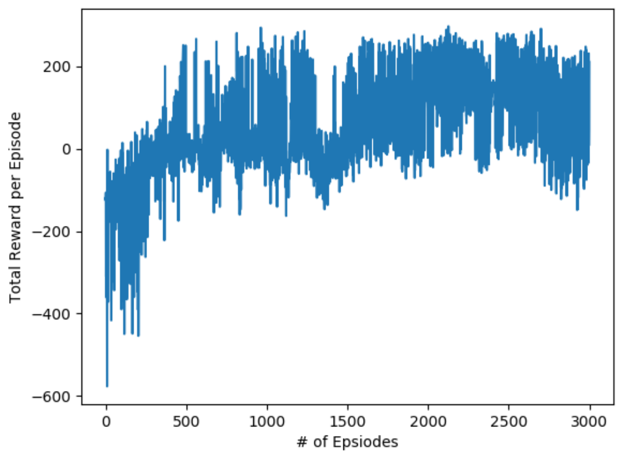


Figure 2. Total Rewards Per Episode during training

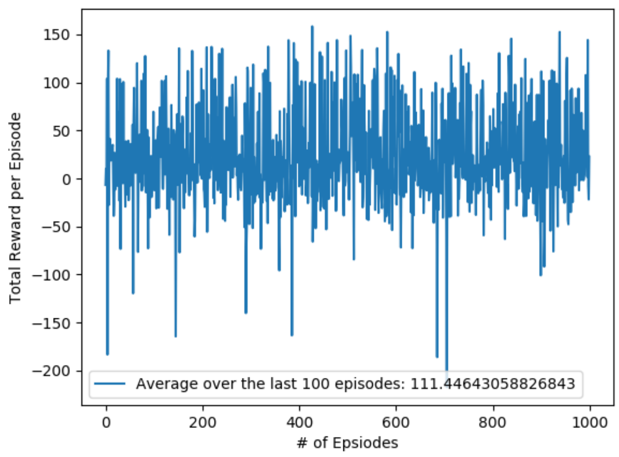


Figure 3. Total Rewards Per Episode using the trained network

The network learns the optimal set of actions needed to be chosen to earn rewards more than 200 in just around 500 episodes. But as can be observed, as more and more episodes are fed, there is a huge variance in the rewards earned even until the last episode(3000). There is instability in learning as the network tries to adjust weights trying to minimize loss as the episodes are fed. As a next step, the trained network was used to navigate the lander and the experiment was repeated until the agent achieved an average score of 200 or above over 100 consecutive episodes or until the agent completed 1000 episodes. The results are shown in Figure 3.

The agent was not able to solve the environment even after 1000 episodes. As can be observed, there is not even a single episode with a reward of 200 or more. Rewards of all the episodes range from -200 to 150. This shows that even though the agent has learnt to hover the lander for a certain time period without crash landing, it is not able to land properly with the proper set of actions(Switching off the engine and zero speed). This is because of the instability in the learning as shown by the high variance in Figure 1.

The high variance can be explained by the fact that each time a gradient update is performed(after each episode), an estimate of the gradient generated by a series of rewards accumulated through out the episode is used. This causes the gradient to go up despite taking a few bad actions, thus making the estimation noisy and ultimately affecting the learning. This has been highlighted as one of the major disadvantages of policy gradient methods. However, the variance can be reduced by using a baseline function that can be subtracted from the reward function so that the gradients are pushed up only for good actions and pushed down for bad actions.

D. *Tuning of Hyper Parameters*

Number of layers in Neural Networks and number of units in each layer play a major role in learning. For example, in networks with layers consisting of a small number of units, the problem of weight sharing is prevalent as changes in weights in one part of the network might affect weights in other parts causing the training to diverge and not converge. So layers with large number of units(two layers with 256 units each) were tried for learning. But a very large number of units also lead to overfitting thus causing poor generalisation. Hence various ranges of values from 30 to 300 were tried as number of units per layer.

In addition to tuning network layers, the experiment was repeated with various values for the discount factor, gamma and learning rate of the neural network. The number of episodes used for training was also increased from 2000 to 3500. But the agent was not able to gain a reward of 200 or more for 100 consecutive episodes. Results obtained from tuning network parameters are shown below.

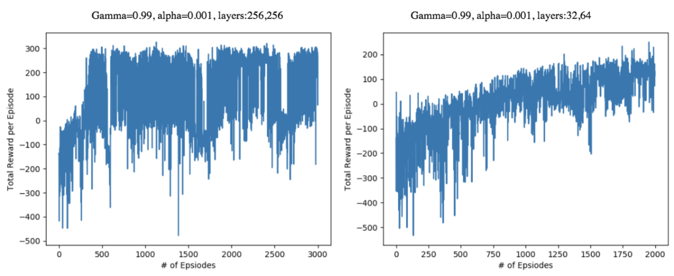


Figure 4. Effect of tuning network layers in Neural Network

As shown in Figure 4, using a small number of units does not help in learning as the agent very rarely receives a reward of 200. This is due to the problem of weight sharing explained earlier. On the other hand, network with layer consisting of 256 nodes performs fairly better but still the problem of variance persists. The agent gains a reward of 300 as early as 500 episodes but as more and more data is fed, the rewards between episodes vary by a very high margin(for reasons cited earlier).

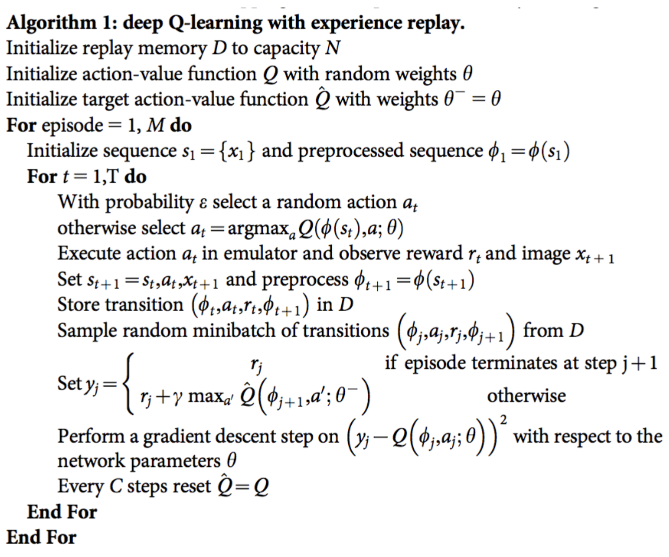
Further tuning of hyper parameters or implementing a baseline function might have helped in solving. But due to time constraints, an algorithm that approximated action values rather than directly approximating the underlying policy seemed more straight forward to implement and easier to make the network converge to near-approximate values. Hence DQN, an algorithm that approximates the action values instead of searching directly for policies was implemented.

III. Deep Q Network

Volodymyr et al.(2015) proposed a novel agent(DQN) that combines reinforcement learning with a class of artificial neural networks known as deep neural networks. The main advantage of DQN is that it uses experience replay that randomizes over the data that helps to smooth the changes in data distribution. This helps in avoiding correlations between successive training episodes. Also, it uses an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated(every C steps), thereby reducing correlations with the target. It uses the following loss function to train the network:

$$L_i(\theta_i) = [(r + \gamma \max_{a}^{\prime} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2] \quad (3)$$

where γ is the discount factor determining the agent's horizon, θ_i are the parameters of the Q-network at iteration i and θ_i^- are the network parameters used to compute the target at iteration i. The entire algorithm as mentioned in Volodymyr et al.(2015) is shown below.



```
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 5. Deep Q Learning algorithm

A. Implementation

Two neural networks were used as part of the DQN algorithm implementation, one with weights θ_i (called the learner) updated at every iteration i and the other with θ_i^- (called the target) used as the target and updated every c steps. The mean square error between the learner and target was used as the loss function. Experiments involved tuning the various hyper parameters - Layers and number of nodes in the neural network, learning rate of the network, discount factor for future rewards, the length of the window after which target network parameters are updated with the learner parameters, etc.

Volodymyr et al. used a mix of convolutional and fully connected layers for learning the Atari 2600 platform since the input involved raw images representing Atari Frames. For the problem of Lunar landing however, the input is an 8-d vector and hence complex convolutional layers were not needed. Two hidden fully connected layers were found sufficient to solve the problem. The number of non-linear units in each layer and the number of layers were experimented with and two fully connected layers, the first layer consisting of 32 units and the second layer consisting of 64 units were found efficient for solving the problem. Relu activation function was used for each of the units and Adam was used as the optimisation function. A value of 1 was used as epsilon, the rate of random actions at the beginning of the training and it was decayed by a factor of 0.99 until it reached a value of 0.01. After that the value of 0.01 was retained until the end of training.

B. Results

After analysing the effect of tuning various hyper parameters(details shown under Experiments Section), the optimum values for the parameters were fixed and the network was trained and later tested to see if the agent was able to solve the problem of lunar landing - i.e. receive an average of 200 or more for 100 consecutive episodes. The plots showing the rewards received per episode during training and testing are shown below.

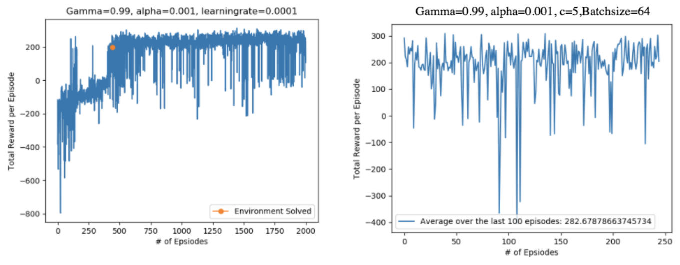


Figure 6. Rewards per episode during training(Left) and testing(Right)

The network is able to learn the problem of lunar landing at around 500 episodes. After that, even though there are occasional dips in the reward gained, the agent is able to maintain a reward of 200 or more for most of the episodes. This shows that the training has converged to the near-approximate values and learning is consistent and has not diverged. This has further been proved by testing the already trained network against the lunar landing environment. From Figure 5(right), it can be seen that the agent solves the problem in 250 episodes.

C. Experiments

1) Effect of tuning Batch size : Batch size plays an important role in training Deep networks. A small batch size might not have a smooth data distribution. While large batches have a smooth distribution and allow larger learning rates, it might lead to poor generalization. So random values ranging from 50 to 100 were tried as batch size and the results were plotted (Shown in Figure 6) to find the optimum batch size. As expected, lower batch sizes caused a lot of instability in training and higher batch sizes were not able to solve the problem due to poor generalisation. Optimal batch size was found to be around 60 and the results for various batch sizes are shown below.

From Figure 6, it can be observed that only the value of 64 for batch size makes the network consistently gain a reward of 200 or more. The network with batch size 60 is also able to solve the problem but with much higher variance. With a batchsize of 65, the agent was not able to solve the problem.

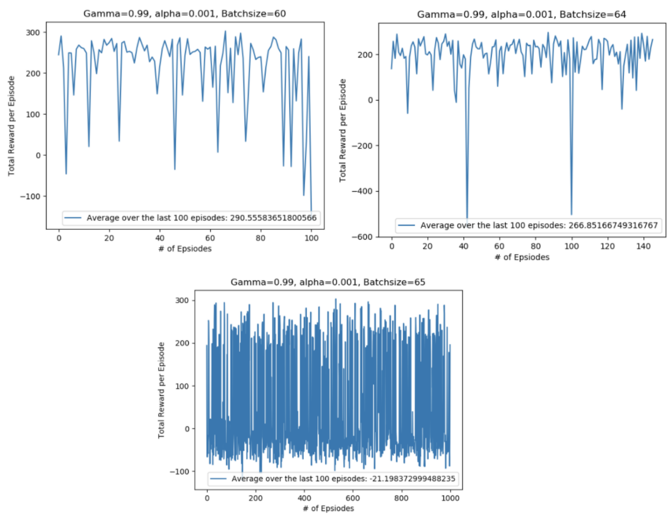


Figure 7. Effect of Tuning Batch size

2) Effect of tuning the Update Window of Target Network: As mentioned earlier, updating the target network only once in every C steps reduces correlations between the action values(Q) and the target. This is because the procedure adds a delay between the time an update to Q is made and the time the update affects the targets, making instability much more unlikely to occur. If the window is too small, we lose the advantage of reducing correlations. However, if the window is too large, the losses are magnified, thereby making large updates to the weights. This further causes the network to diverge rather than converge to the correct values. A range of values(4-10) were tried and the results were plotted. The results are shown in Figure 7(Rewards per episode during training) and Figure 8(Rewards per episode during testing)

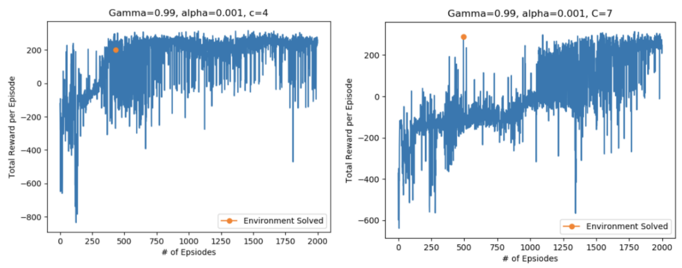


Figure 8. Effect of Tuning Target Network Update Window during Training

Training curve for C=5 is shown in Figure 5. For both C=4 and C=7, the agent receives a reward of 200 as early as 500 episodes. But after that, the values constantly keep oscillating between 200 and -600. This shows that for C=4, the training data is not comprehensive enough to make the network converge and for C=7, the learning has diverged as it reaches a reward of 200 and then constantly drops afterwards.

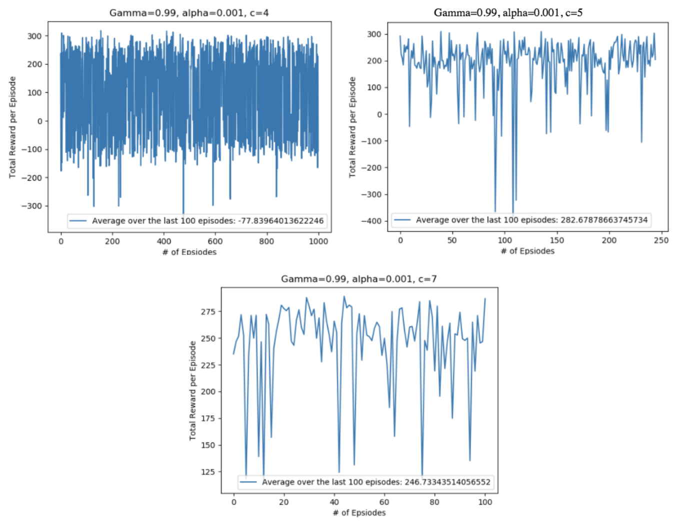


Figure 9. Effect of Tuning Target Network Update Window during Testing

As shown in Figure 7, value of 4 for C results in high variance in reward gained and thus not able to solve the problem. The optimum value of C was found to be 5 since the agent is able to gain rewards more than 200 for more number of episodes than both lower and higher values of C. Value of 7 for C also exhibits high variance in reward gained when compared to C=5.

3) Effect of tuning the discount factor(γ): Discount factor sets the horizon for the agent which determines how deep in to the future the agent should look in order to determine the action values(Q). Low discount factors make the agent short sighted thus making the learning inefficient. Even though the task considered here is episodic, solving the problem still needs insight into the future states since there are states that may not be very rewarding but may lead to high value states when certain actions are taken. A very high discount factor looks too much in to the future and weighs future rewards heavily thus not helping much with learning. A range of values from 0.9 to 1 were tried as discount factors and results were plotted. For some of the values, the agent never learnt how to solve the problem(gain a reward of 200 or more) . It occasionally gained rewards of 200 probably from a random action but the rewards were never consistently high even after 2000 episodes.

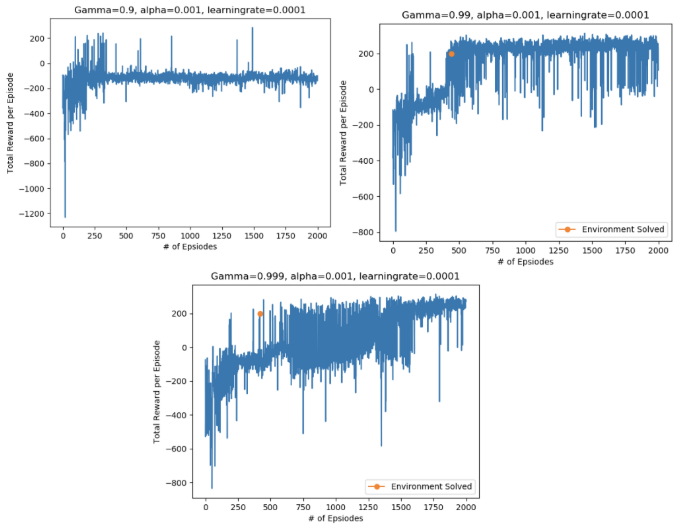


Figure 10. Effect of Tuning gamma on the rewards gained per episode during training

With a discount factor of 0.9, the agent never learnt to successfully land the lander. With a discount factor of 0.999, the agent is successful after 500 episodes but there is a lot of variance in the rewards received even until the last episode. This shows that the agent is not able to converge to the right values as the network keeps adjusting the weights trying to minimize loss. The discount factor of 0.99 makes the network converge to the right values and hence the agent learns to land successfully and consistently. When the trained agents were tested, only the agent trained with a discount factor of 0.99 was able to achieve a reward of 200 or more for 100 consecutive episodes. The rest of the agents failed to solve the problem.

IV. Problems Encountered

A. Choice of Framework

One of the major challenges faced during the implementation of algorithms to solve lunar landing was the learning curve needed for understanding and implementing the neural networks. Given that there are a vast number of choices available for implementing neural networks, having no prior knowledge of any of the libraries demanded reading the documentations of many of them(at least the most popular ones like Pytorch, Keras, Tensorflow) and weighing them in on various factors like ease-of-use, etc. Finally, the network was implemented using pytorch as the documentation was detailed and even had an example framework for solving reinforcement learning problems using neural networks.

B. Tuning Hyperparameters

Tuning hyperparameters was another major challenge as there are a lot of hyperparameters involved and each run took a lot of time. Even though several hyperparameters have a common value that they take in most of the experiments, like 0.9 for gamma, 0.001 for learning rate, etc, some of the hyperparameters had to be tried with a vast range of values in order to narrow down to the most optimal value. For example, C, the update window of target network in DQN, epsilon, the rate of random action and epsilon decay, the rate of decay of epsilon. The algorithm did not converge for a lot of values except for the ones mentioned in the Implementation section under DQN.

C. Choice of Algorithms

Another challenge faced was the availability of a vast number of algorithms under different categories like policy approximation, function approximation, on-policy methods, off-policy methods, etc. Understanding the algorithms and finding one that was efficient for the problem of lunar landing was paramount to solving the problem. The learning process demanded a lot of time in order to understand each of them and weigh them in on relevant factors.

V. Further Experiments

Due to time constraints, several methods/experiments were not done as doing them would mean taking the longer route. For example, further tuning of Policy gradient methods or implementation of a baseline function could have resulted in convergence. But a more straightforward method was chosen in order to solve the problem faster. Actor-critic methods, that learn approximations to both policy and value functions are another area of interest and a promising method that could have resulted in successfully solving the problem.

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] Sutton, R. S., Barto, A. G. (2018). 13. Policy Gradient Methods *Reinforcement learning: An introduction.*MIT press.

[3] Sutton, R. S., Barto, A. G. (2018). 9.On-Policy Prediction with Approximation *Reinforcement learning: An introduction.*MIT press.

[4] Isbell, C., Littman, M.L. *Reinforcement Learning : Generalization* Retrieved from https://classroom.udacity.com/courses/ud600