Mars Obiter Landing using Reinforcement Learning

EE599 Final Project Report

Group #27 – Wenjun Li, Ishan, Tamoghna

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

An important part of Deep Learning is Reinforcement Learning which refers to goal-oriented algorithms, which learn how to attain a complex objective. The Lunar Lander environment is an integral part of Open AI gym and a great example of reinforcement learning. The environment consists of a surface with two flags and a lander which tries to land in between them. Building up on this environment, we created a Lunar Orbiter environment. In this new environment, the position of the flag near which the lander lands has been made random for every episode. The terrain has been made rougher and more mountainous, increasing the slopes to make it more difficult for the agent. The agent itself has been changed by decreasing the angle of legs and making the center of gravity higher, thus making it less stable. Thus, a new reward function and state variables have been devised for the new environment. Three different Reinforcement Learning Algorithm have been applied on the new environment – Deep Q Learning, Policy Gradient and Asynchronous Advantage Actor Critic Algorithm. The outputs of the three algorithms on the new environment have been compared based on the number of episodes it took to train the agent and the number of perfect landings during test. Based on the tests, the Deep Q Learning algorithm took the least time to train but the Policy Gradient had the best test result, with the agent landing most times during test.

CHAPTER 1. INTRODUCTION

1.1 Project Motivation and Objectives

This project is inspired by Space X falcon rocket retrieval. Also, many countries are planning to land a rover on Mars to explore Mars. So, landing a Mars obiter automatically using reinforcement learning pop into our minds.

The objectives of this project is to dive deep into reinforcement learning and explore this field in more detail.

- Understand what's the difference between different RL algorithms.
- Build a new environment based on the baseline environment and make it more challenging.
- Understand more about the state function, reward function and value function through tuning them to fit with the new task.

1.2 Introduction to Reinforcement Learning

Unlike machine learning and deep learning, reinforcement is an unsupervised learning method. There is no supervisor in the model and the model only needs a reward signal. Depending on the reward signal, agent can learn by itself in the environment.

To better understand reward, here the definition of reward hypothesis is as follows: all goals can be described by the maximization of expected accumulative reward. So, the agent's job is to select decision to maximize the total future reward.

Below are some terms and concepts used in RL.

- Markov State: P[St+1|St] = P[St+1| S1, S2, … , St]
- Agent: who takes actions in an environment
- State: the information used to determine what happens next: St = func(Ht)
- Environment State: Ste - not visible to the agent
- Agent State: Sta - information used by RL algorithms: Sta = func(Ht)
- History: Ht = A1, O1, R1, …, At, Ot, Rt
- Observation: what the agent observes from environment
- Markov Decision Process: full observability: agent directly observes environment state: Ot = Sta = Ste

Major Components of Agent

- Policy: agent's behavior function

$$\begin{cases} a = \pi(\mathrm{s}) & Deterministic\ Policy \\ a\ (a|s) = P[A = s \mid S = s] & Stochastic\ Policy \end{cases}$$

- Value Function: to evaluate the goodness/badness of each state and action

$$V_\pi = E_\pi \left[R_{\mathrm{t}} + \gamma\, R_{\mathrm{t+1}} + \gamma^{2} R_{\mathrm{t+1}} + \ \ldots \middle|\ S_{\mathrm{t}} = s \right]$$

- Model: predict what the environment will do

- Transition: P predicts the next state give current state and action
- Reward: R predicts the next reward give current state and action

Functions that used to solve problems:

- Value Function: $V_\theta(s)$
- Action-Value Function: $Q_\theta(s, a)$

Based on different components of agents, there are 3 major RL approaches:

i. Value-based RL
   Estimate the optimal value function Q^* (s, a)

ii. Policy-based RL
   Directly search for the optimal policy π^*

iii. Value & Policy -based RL
   Actor Critic: updates action-value function parameter *w* (actor) and updates policy parameter θ in the direction suggested by critic (critic)

1.3 Project Baseline

This project is based on an already available environment in OpenAI gym, and many people has tries some RL algorithms on this simple environment.

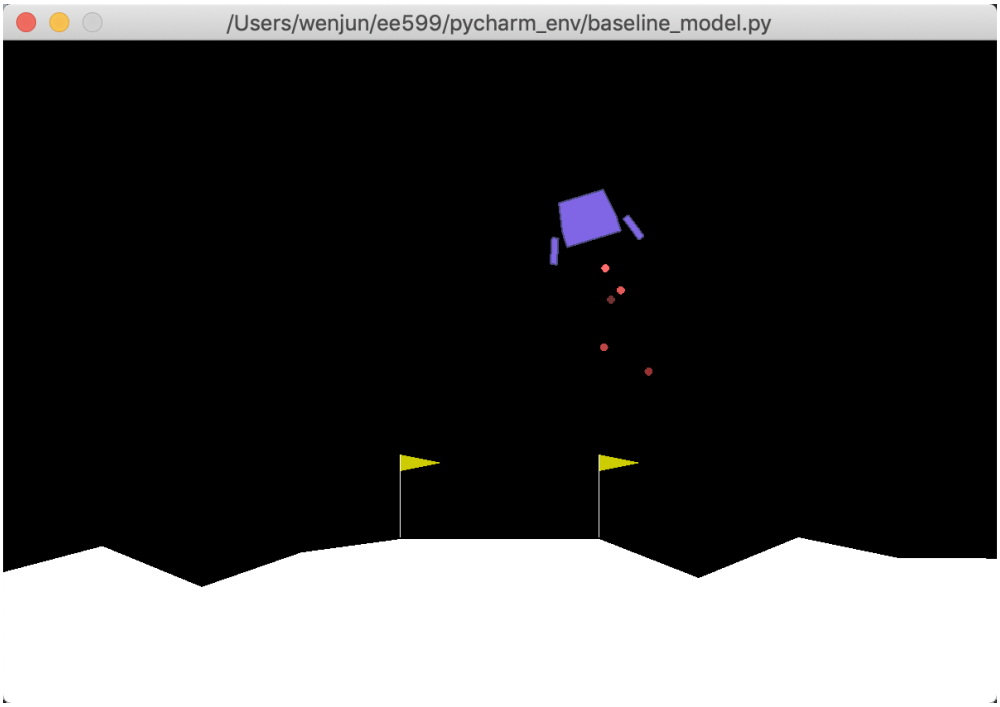

Figure 1

The figure above shows the original environment, which is pretty simple. The target landing area is always in the middle, i.e. between two flags; there is no big slope or high mountain on the terrain; the rocket has a low gravity center and wide-open legs, that can let it be more stable; etc.

Based on this OpenAI gym, we have tried Q-Learning, Policy Gradient and Actor Critic on it. Below are the algorithms' performance on the baseline environment.
The Q-Learning algorithm took about 700 episodes to solve the baseline Environment. It gave pretty good output results and convergence after training.

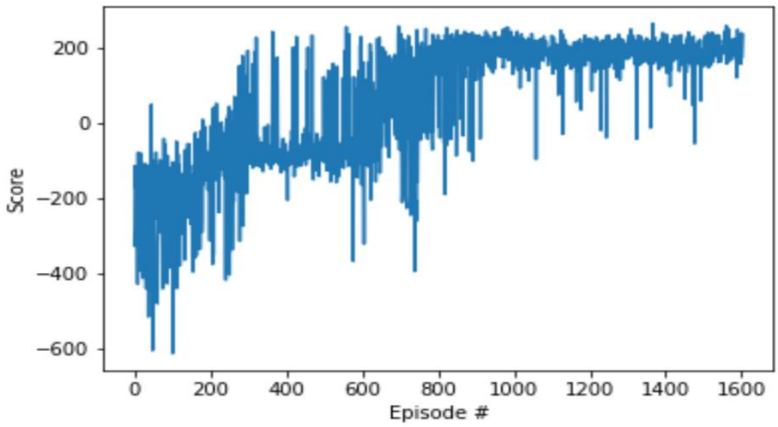

Figure 2

The Policy Gradient algorithm also took about 500 episodes to solve the baseline Environment.

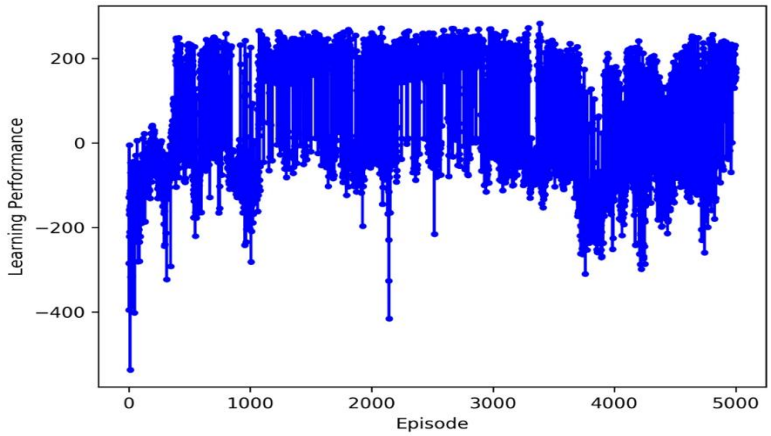

Figure 3

The Actor Critic algorithm took about 3500 episodes to solve the baseline Environment.

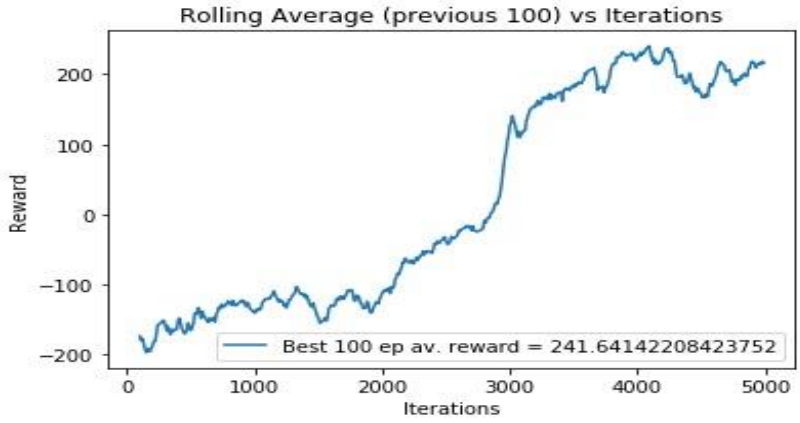

Figure 4

1.4. Introduction to Environment

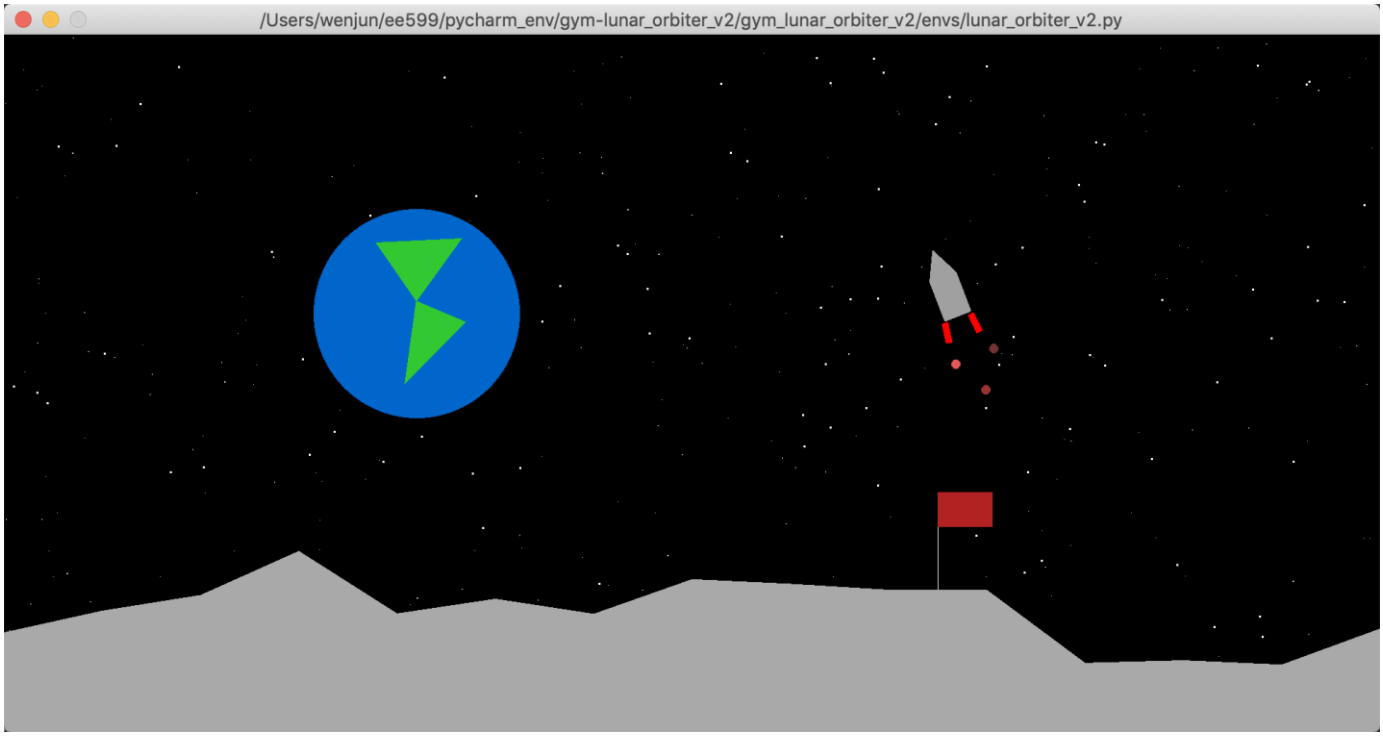

Figure 5

The new environment compared to the old environment is strikingly different visually.

Listing those visual changes:

1. Addition of the planet earth.
2. Addition of stars in the night sky.
3. The Terrain is more rugged and rocky.
4. There is only one Flag and its shape is changed.
5. The Shape of the Orbiter is modified.

Behind the scenes changes:

1. Main and Side Engine power have been increased.
2. Landing spot (placement of Flag) is random.
3. Terrain randomly generated.
4. Orbiter body and properties have been modified to make it more higher-gravity. This is done in-order to make this orbiter more unstable for it to land properly.

The other changes are based on the **State**, **Actions**, **Reward** and **Task**.

Task: In the previous environment, the orbiter had to land exactly between the two flags at (0,0). In the new environment, the orbiter needs to land as close as possible to the single flag mounted on the surface.

State Variables: x-distance and y distance are adjusted accordingly to land near the flag and the previous 6 state variables remain the same as the old environment. The other 6 state variables are linear x-velocity, linear y-velocity, angle of orbiter, angular velocity and leg 1 contact of orbiter and leg 2 contact of orbiter.

Rewards:

1. When the orbiter gets closer to a range of 0.1 of the Flag, reward of +0.3 was given continuously.
2. When range was 0.05 of the Flag, reward of +0.5 was given continuously.
3. The Agent was penalized when it was outside the above-mentioned range. Negative reward of -0.1 was given.
4. If the lander was alive and landed near the flag within the mentioned range of 0.05, it was given a reward of +150.
5. Negative rewards were given for firing main and side engines with a scale of 0.05 and 0.001 times the current power.
6. If the orbiter crashed or goes out of the frame, the episode finished and a negative reward of -200 is awarded.
7. $-150 * (\sqrt{x^2 + y^2})$, negative reward for the distance between the flag and the orbiter.
8. $-100 * \sqrt{vel\ x^2 + vel\ y^2}$, negative reward the linear velocity movement.
9. $-80(\ abs(orbiter\ angle) + 10 * (leg\ 1\ contact) + 10 * (leg\ 2\ contact)\ )$, negative reward is given for the stability of the orbiter and landing pattern.

With these changes we define our new environment which is registered under OpenAI gym. Therefore, anybody can use this environment to train and test their orbiter landing models.

CHAPTER 2. Q-LEARNING

2.1. Introduction

Q-learning was introduced by Watkins in 1989. It is a model-free reinforcement learning algorithm which means that it does not require the environment's transition probability distribution associated with its Markov Decision Process (MDP) to decide the agent's action and next state when present in a given initial state. Q-Learning is utilized in identifying an optimal action-selection policy for a given environment. The **"Q"** in Q-Learning stands for the **"Quality"** of an action in a given state.

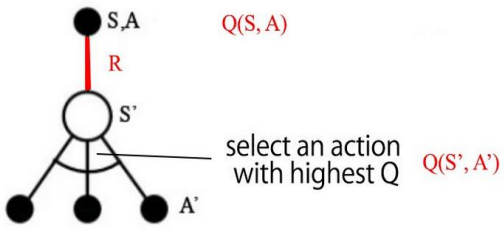

Figure 6

2.2. Q-value Function

Before covering the Q-Learning algorithm, it is imperative to understand it's building blocks.

1. Q-value function is defined as the expected value of the sum of future rewards by following a policy π. The value of this function (Reward r) gives an indication of how good the action-**a** is compared to the other actions in a given state-**s**.

$$Q^{\pi}(s,a) = \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s, a\right]$$

2. Q-value function can be re-written as a Bellman-Equation which shows the relationship between the current Q-value and its successive time-step Q-value.

$$Q^{\pi}(s,a) = \mathbb{E}_{s',a'}\left[r + \gamma Q^{\pi}(s',a') \mid s, a\right]$$

3. The maximum achievable value of the Q-value function becomes its Optimal solution given by:

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a) = Q^{\pi^{*}}(s,a)$$

$$\mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^{*}(s',a') \mid s, a\right]$$

4. The Optimal Q-value function Q* now produces the Optimal Policy π*

$$\pi^*(s) = \underset{a}{\text{argmax}}\ Q^*(s, a)$$

2.3 Q-value Iteration Algorithm

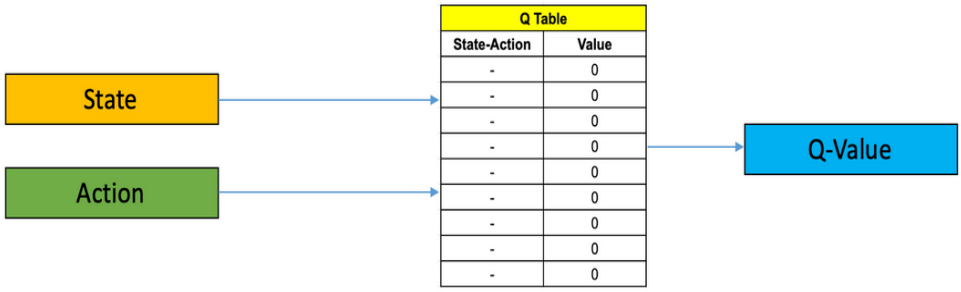

Figure 7

Using the above building blocks, the value-iteration algorithm is performed for achieving the optimal Q-value function Q* to find the Optimal Policy π*.

$$Q_{i+1}(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s', a') | s, a\right]$$

As "i" in the above equation goes to infinity the Qi converges to Q*. The updated Q-values are stored in tables known as Q-Tables that have values corresponding to every state-action pair. An Action is selected based on the Q-table, the reward for that action is measured and then the Q-value is updated.

The Drawback of the value-iteration algorithm in approximating the Q-value function is that it is not scalable for complex real-life environments as it must calculate Q(s,a) for every state-action pair. This becomes infeasible for ginormous environments.

2.4. Naïve Deep Q-Learning Algorithm

The shortcomings of the value-iteration algorithm is tackled with the Naïve Q-Learning algorithm. Now, the Q-value function is represented as a Q-Network with weights **w** using Deep Neural-Networks.

$$Q(s, a, \mathbf{w}) \approx Q^*(s, a)$$

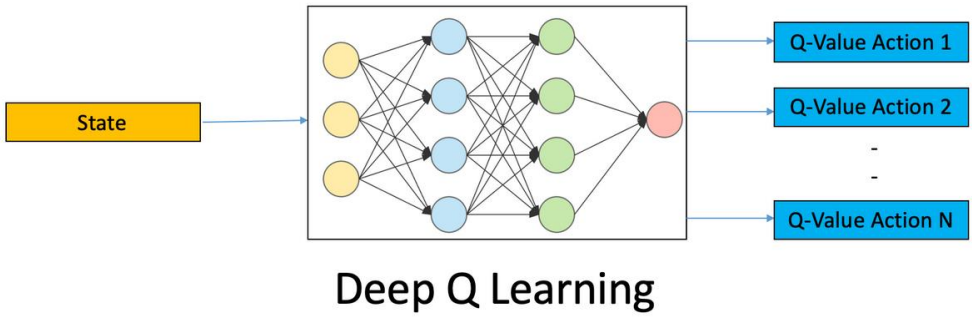

Figure 8

An Objective Function is then defined by mean-squared error in Q-values,

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

The blue-highlighted term is treated as the target term and Q(s,a,w) is treated as the predictor term. This temporal difference between the next-time step Q-value and the current one from the bellman equation defines the loss function.

So , $Loss = E\,(\,Target - Predicted\,)^2$

The Q-Learning Gradient is obtained by the equation below,

$$\frac{\partial \mathcal{L}(w)}{\partial w} = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w)\right) \frac{\partial Q(s, a, w)}{\partial w}\right]$$

The Loss/Objective function is then minimized using the above Q-Learning gradient and the Stochastic Gradient Descent (SGD) update rule.

2.5. Issues with Naïve Deep Q-Learning

Trying to approximate Q-value function using deep neural networks face divergence or oscillation problems. These problems can be due to the following reasons as stated below:

1. The data is sequential implying strong correlation between samples and therefore destroys the idea of independent and identically distributed samples (i.i.d).
2. Non-stationary Q-Targets cause oscillation as slight Q-value changes leads to rapid changes in policy.
3. The Reward range is unknown as large gradients can lead to instabilities during backpropagation.

2.6. Deep Q-Networks (DQN)

The Deep Q-Networks (DQN) address the above issues by the following techniques:

1. **Experience Replay**: This techniques is used to break correlations between samples by building a data buffer D known as the **Replay Memory** which contains the agents previous experience. Sample these experiences instead of taking the most recent action and apply the update instead.

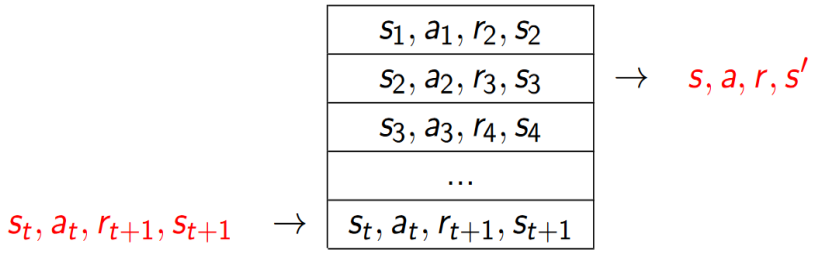

Figure 9

2. **Separate Q-Target Network**: Prevent oscillations by building a second Q-Network and fixing the weight parameter w in the target network and the weights in the predictor network are updated. This maintains stationarity.

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

3. **Reward Normalization**: Clipping rewards to the range of [-1,+1] and using batch-normalization helps define Reward range and maintain stability during backpropagation.

2.7. Implementation of DQN Algorithm with Experience Replay

The DQN algorithm works on the flowchart given and the pseudo code mentioned below.

Algorithmic Flowchart:

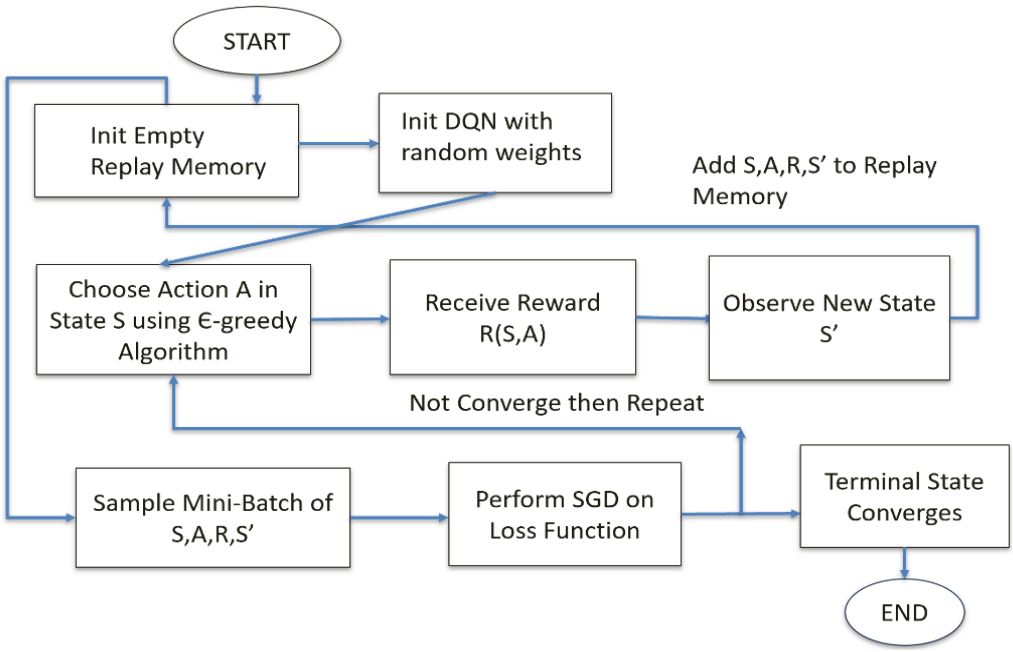

Figure 10

Pseudo Code For DQN:

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 ϵ 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

The **Є-greedy** policy starts from numerical value 1.0 and decays slowly. This parameter ensures that there is some exploration by forcing the agent to take random probabilistic actions when given in a state. It mimics the simulated annealing algorithm in the sense that there is exploration before exploitation.

2.8. Deep Neural Network Architecture in DQN

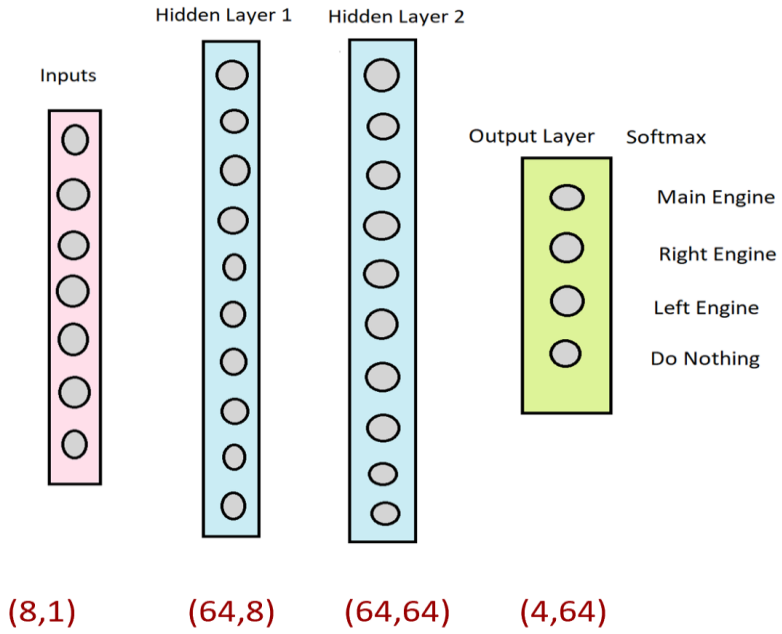

Figure 11

The deep neural network of the architecture is as given above. The network takes in the eight state variables and has 2 hidden layers of (64,8) and (64,64). The output layer has a SoftMax function applied and produces 4 discrete action outputs namely: Main, Right and Left Engine plus remain idle/ Do Nothing. This network has lesser than 5000 neural network parameters.

2.9 Result and Output Plots

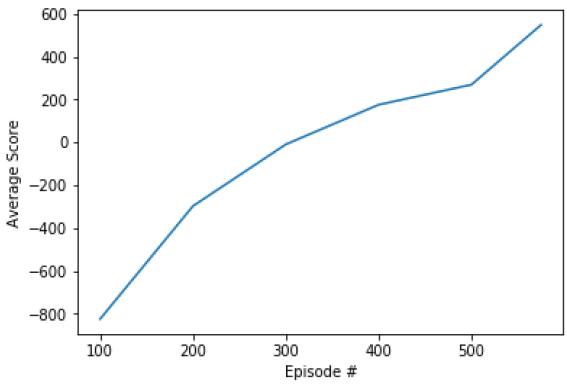

Figure 12

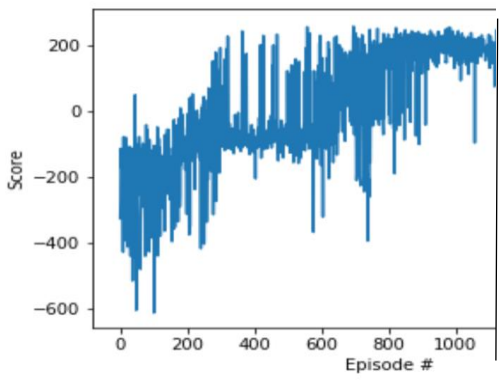

Figure 13

Average reward per 100 episodes and the next plot is of reward after each and every episode. We see quite a smooth curve for the average reward plot which implies fast and good convergence. In the second plot we do observe large variances in the beginning which stabilize quickly after some 300-400 episodes. The Average convergence for the DQN model is around 550 episodes. When run on an average, it is seen that in around 30 out of 50 episodes, the agent lands perfectly and in the other episodes it seems to be blocked out of the rocky-rugged terrain.

The Performance can be seen on the following YouTube Video link:
https://www.youtube.com/watch?v=V3I30vjILuQ

2.10. Shortcomings of DQN

There are some defects in the DQN algorithm, which are as mentioned below:

1. Complex Q-Value Function leads to bad performance as they cannot be learnt easily and approximated.

2. It does not work well for continuous action space environments as the innumerable Q-values from [-1,+1] would blow-up and lead to extremely high computation cost.

CHAPTER 3. POLICY GRADIENT

3.1 Introduction of Policy Gradient

What is the requirement for policy-based algorithms since value-based algorithms can be used to solve problems? That is because policy-based algorithms can offer us several advantages that value-based algorithms do not have.

The No.1 reason of using policy-based algorithm is that it is effective in high-dimensional and continuous action spaces. In such environments, maxima value function (Q) can be prohibitively complicated and tricky, while using policy can be more straight-forward and simpler.

The second reason is policy-based algorithm has better convergence properties. The value function needs to be very precise and accurate if value-base algorithms are used, or the maximization function will oscillate around optima finally instead of converging. However, if policy-based algorithms is used, the agent is guaranteed to reach at least a local optimum. Because the policy is updated better and better for each step and the policy will improve incrementally gradually and reach at least local optima.

The third reason is that policy-based algorithms can learn stochastic policies. So, why do we want to learn stochastic policies? It is because we want our agent to have stochastic actions, so that it can enhance "exploration" in the environment. A deterministic policy hurts exploration badly and the curvature of accumulated rewards can be very steep and bad for training. Therefore, we want our agent to behave stochastically although it will also bring high variance.

3.2 Monte Carlo Policy Gradient

In this project, Monte Carlo policy gradient was used, which is a fundamental version of policy gradient. While in Q-Learning $\varepsilon - greedy$ to maximize Q* is used, in Monte Carlo policy gradient the best θ is found given policy $\pi_\theta(s,a)$ with parameter θ. To do this, here the policy objective function is introduced as follows.

$$J(\theta) = \begin{cases} V^{\pi_\theta}(s_1) = E_{\pi_\theta}(v_1) & episodic\ env \\ \sum_s d^{\pi_\theta}(s)\, V^{\pi_\theta}(s) & continuous\ env \end{cases}$$

Next, we try to find maximum in $J(\theta)$ by ascending the gradient of the policy w.r.t parameter θ. First, compute the derivative of policy $\pi_\theta(s,a)$ w.r.t θ and we have the below equation:

$$\nabla_\theta log\pi_\theta(s,a) = \nabla_\theta\ \pi_\theta(s,a)$$

Then, we plug this result equation into the derivative of $J(\theta)$ w.r.t θ and we get:

$$\nabla_\theta J(\theta) = E[\nabla_\theta log\pi_\theta(s,a) \cdot r]$$

where r is the instantaneous reward. Finally, we replace the instantaneous reward r with long-term reward value $Q^{\pi_\theta}(s,a)$ and we obtain the Monte Carlo policy gradient, which is:

$$\nabla_\theta J(\theta) = E[\nabla_\theta log\pi_\theta(s,a) \cdot Q^{\pi_\theta}(s,a)] = E[score\ function \cdot value\ function]$$

Below is the pseudo code of Monte Carlo policy gradient (also called Reinforce):

function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function

Here, we use return v_t as an unbiased sample of $Q^{\pi_\theta}(s,a)$ because we do not have Q function and we do not need to find Q function in policy gradient.

In next part, we will discuss how to combine Monte Carlo policy gradient with a deep neural network to do optimization.

3.3 Deep Neural Network

All the reinforcement learning algorithms need to derive output (action) from input (state), which means they are a transformation between input and output. Since linear transformation and simple transformation are not powerful enough, we introduce deep neural network to do this job.

The input of the deep neural network takes the state (shape=(8, 1)) of the rocket and the output of the deep neural network gives the next action (shape=(4, 1)) of the rocket. Based on input/output dimension and the complexity of this problem, we build a deep neural network with 2 hidden layers with below structure:

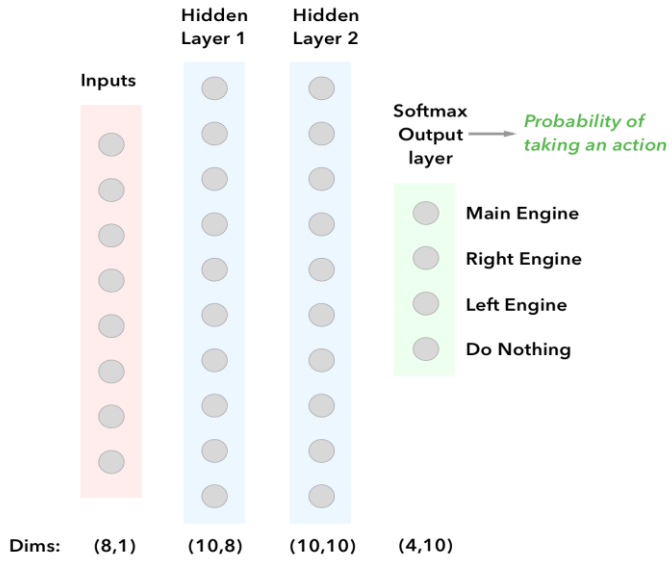

Figure 14

For the loss function, we computed the softmax cross entropy between logits (output

action) and labels (learned action) and then multiply by discounted episode reward. By calculating this, the DNN will know how much future reward it will receive given current logits (action) and automatically improve its behavior.

Since reinforcement learning is un-supervised learning, we still need to collect data for training the DNN. The data here are state, action, reward of current episode and previous episodes.

Combining all the above operations, we derive the below flow chart of Monte Carlo policy gradient algorithm in this problem:

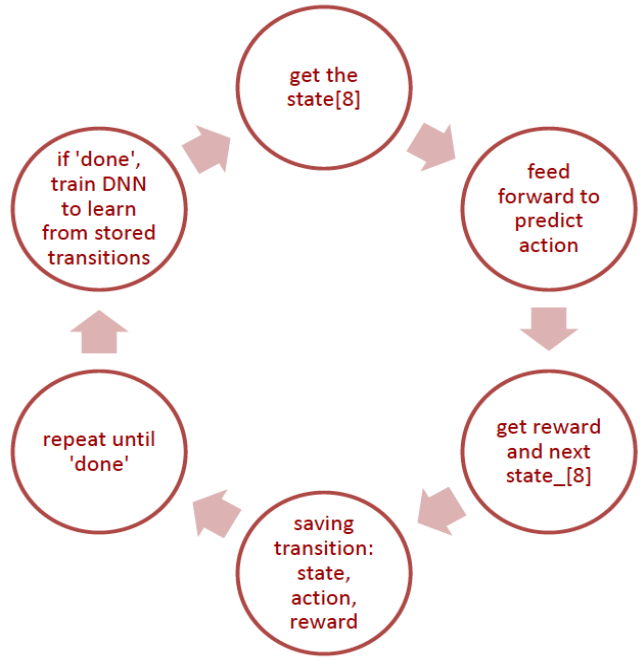

Figure 15

where ‘done’ means crash, go out of the window or land successfully here.

3.4 Ad-Hoc Tuning with the Reward Function

At the beginning of the training, the performance of the algorithm is not that good. Sometimes the rocket will not land close enough to the flag and it will hover around the flag. After analyzing the problem, we found that we needed to tune the reward function. We increased the reward the rocket will receive when it lands close to the flag within a radius and when it flies closer to the flag. Also, we reduced the reward for maintaining upright, because this is the reason it hovers around the flag (hover while maintaining its position upright can get reward) rather than landing as well.

We re-trained my agent after done with above adjustment and found another problem. When the rocket lands far away from flag, it cannot launch again and fly to the flag. So, we reduced the penalty (minus reward) the rocket will receive for firing engine, so that the rocket can launch again when it is far away from the flag.
All in all, We’ve done a lot of tuning with the reward function to achieve a better landing

performance and finally the landing performance are way better than the beginning.

3.5 Result

We will show the best landing performance in this part and give the Youtube video links.

At the beginning of training, the rocket does not know how to fly, not even mention how to land: https://www.youtube.com/watch?v=LBvHGeOcKTA

After about 200 episodes' training, the rocket gradually learns how to fly and land somewhere safely: https://www.youtube.com/watch?v=PIIETocvyXo

About 500 episodes later, the rocket becomes very smart and it can land close to the flag successfully for most of the time: https://www.youtube.com/watch?v=zdgyTt9MjV8

The plot of reward-episode.

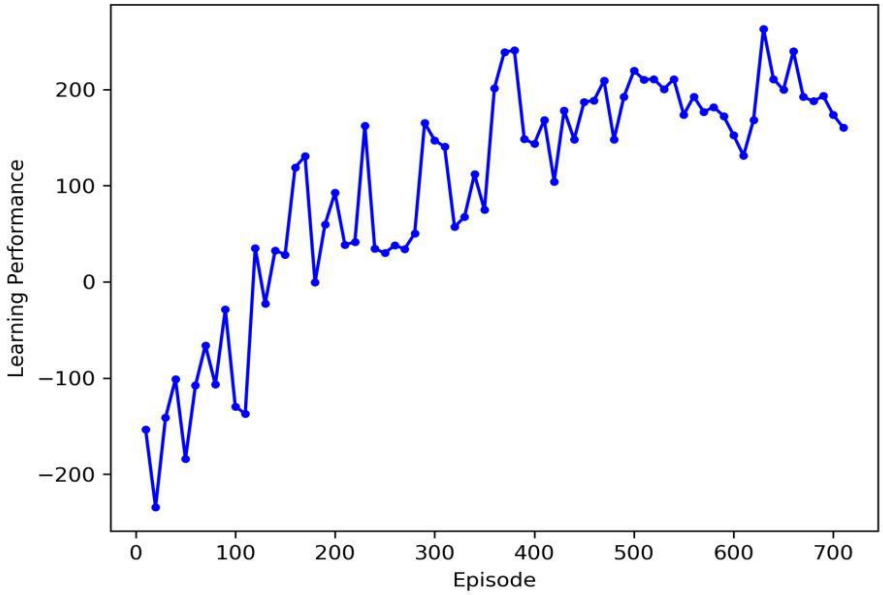

Figure 16

As you can see from the plot, although policy gradient converges after about 400 episodes, it has high variance.

3.6 Analysis

There are some failures: https://www.youtube.com/watch?v=SwkSplrTDlE . There are several reasons that it will fail. The first is that the rocket sometimes will land on a slope and then it will choose to maintain its upright position to get reward rather than fly to the flag to get higher reward (stuck in the local optima). Another reason is that sometimes there will be a high mountain block in the middle of flag and rocket, and the rocket cannot go over this mountain because it does not have observability to this mountain (none of the state variable can tell if there is a mountain).

Therefore, we can conclude that policy gradient (policy-based algorithms) also have 2 inherently shortcomings.

- Evaluating a policy is often inefficient and high-variance
- Sometimes, it converges to a local optimum rather than a global optimum

CHAPTER 4. ASYNCHRONOUS ADVANTAGE ACTOR CRITIC ALGORITHM

4.1 INTRODUCTION

The Asynchronous Advantage Actor Critic Algorithm or A3C as it is called, was developed by Google's DeepMind group. The algorithm on previous built environments, gives better results on less training times. It is faster, simpler and more robust. Also, it can work on both continuous and discrete action spaces. Thus, A3C has quickly become a go-to algorithm for challenging problems with complex states and action spaces.

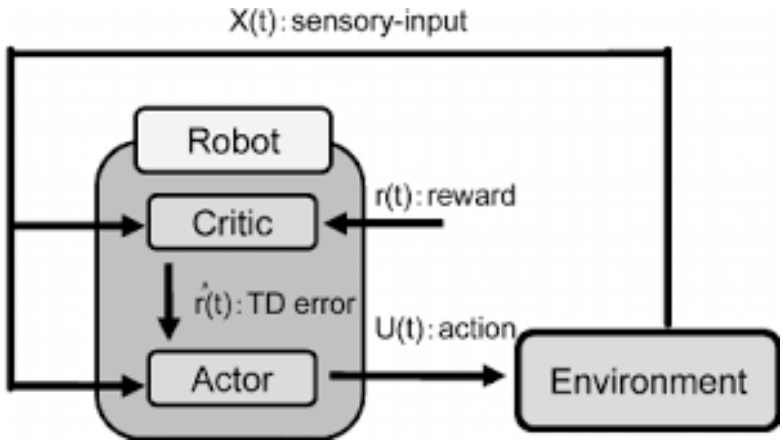

Figure 17

The algorithm consists of three important parts : Actor - Critic, Asynchronous and Advantage.

4.2 THE ACTOR – CRITIC

The A3C model is based on the combination of both value based and policy based approaches. There are two different outputs to the model:

1. A set of outputs which indicate possible actions e.g., $Q(s,a_1)$. It's called Actor or Policy(s).
2. A separate value V(s) which indicates value of the state. This is called critic.

So, basically, there are two sets of neural networks: an actor to determine optimal action and a critic to estimate the potential reward for the action. The networks look as followed:

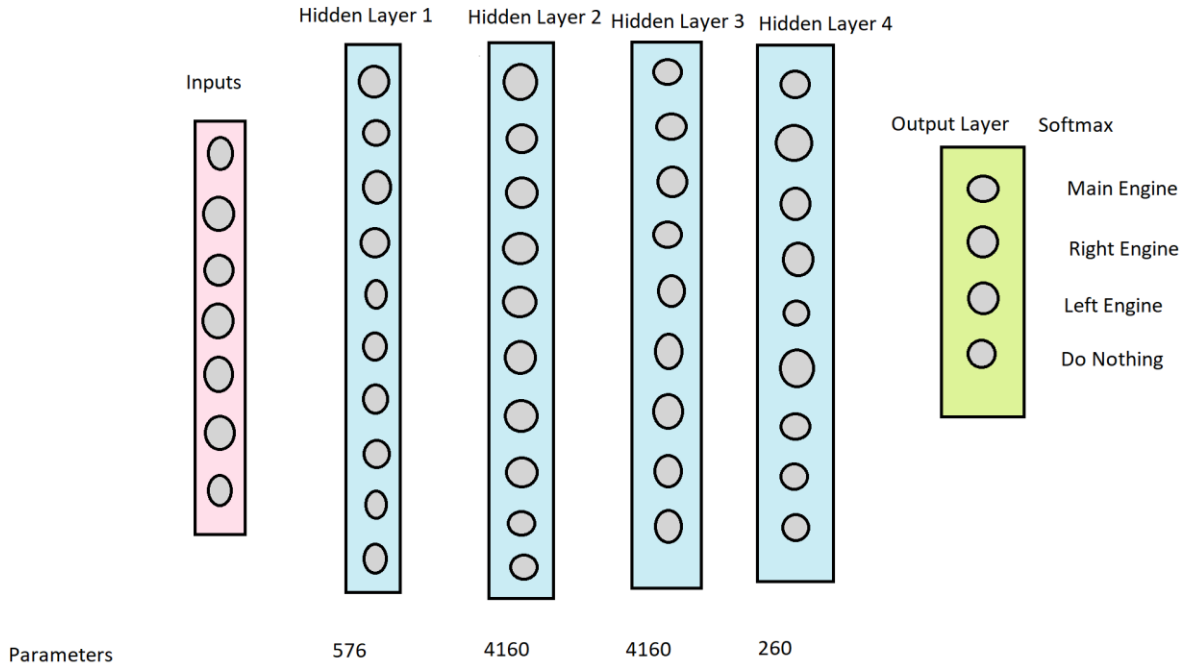

Figure 18 - The Actor Model

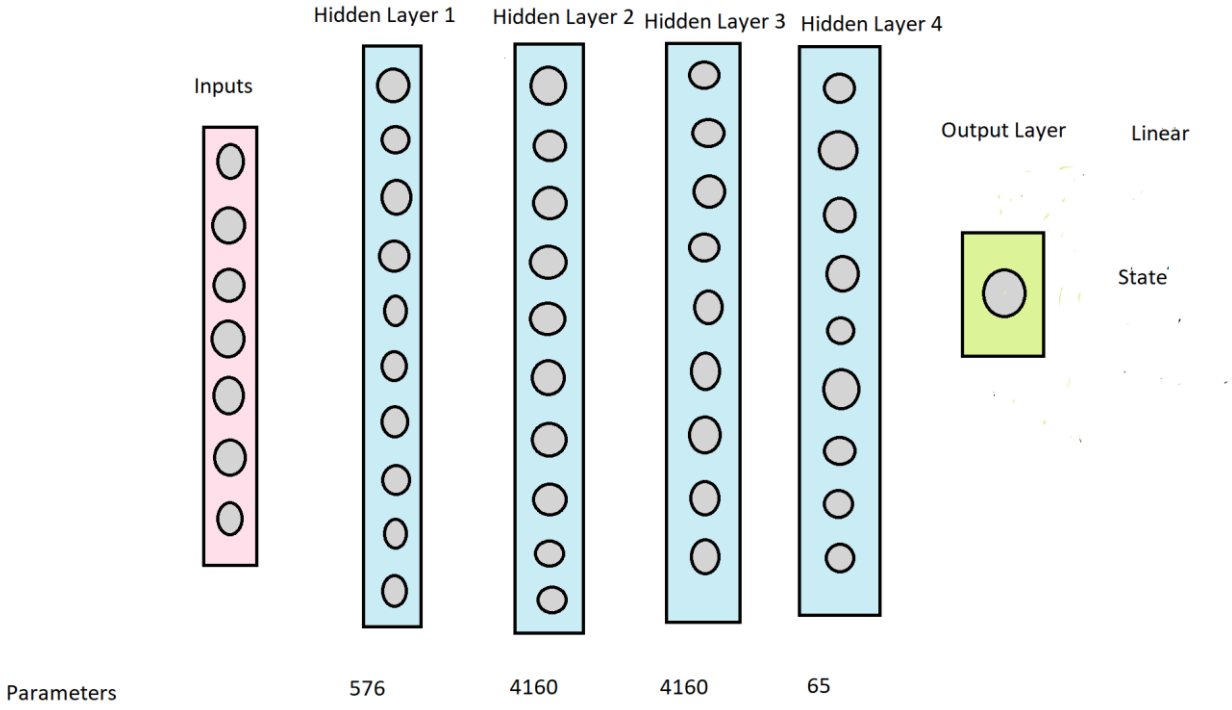

Figure 19 - The Critic Model

4.3 ASYNCHRONOUS

In A3C, multiple agents attack the environment at the same time to learn more efficiently. All the agents are initialized differently and hence called asynchronous. So there is a global network and multiple worker agents which have their own set of network parameters. Each agent works with their own copy of the environment at the same time and as the experience of each agent is independent of other, the overall experience available for training becomes more diverse. This reduces the chance of agents getting stuck in local maxima.

The agents share experience through critic. Thus, V(s) is common for all the agents and calculated through rewards from the environment. Agents adjust their neural network as they go forward to better match the expected value of V(s).

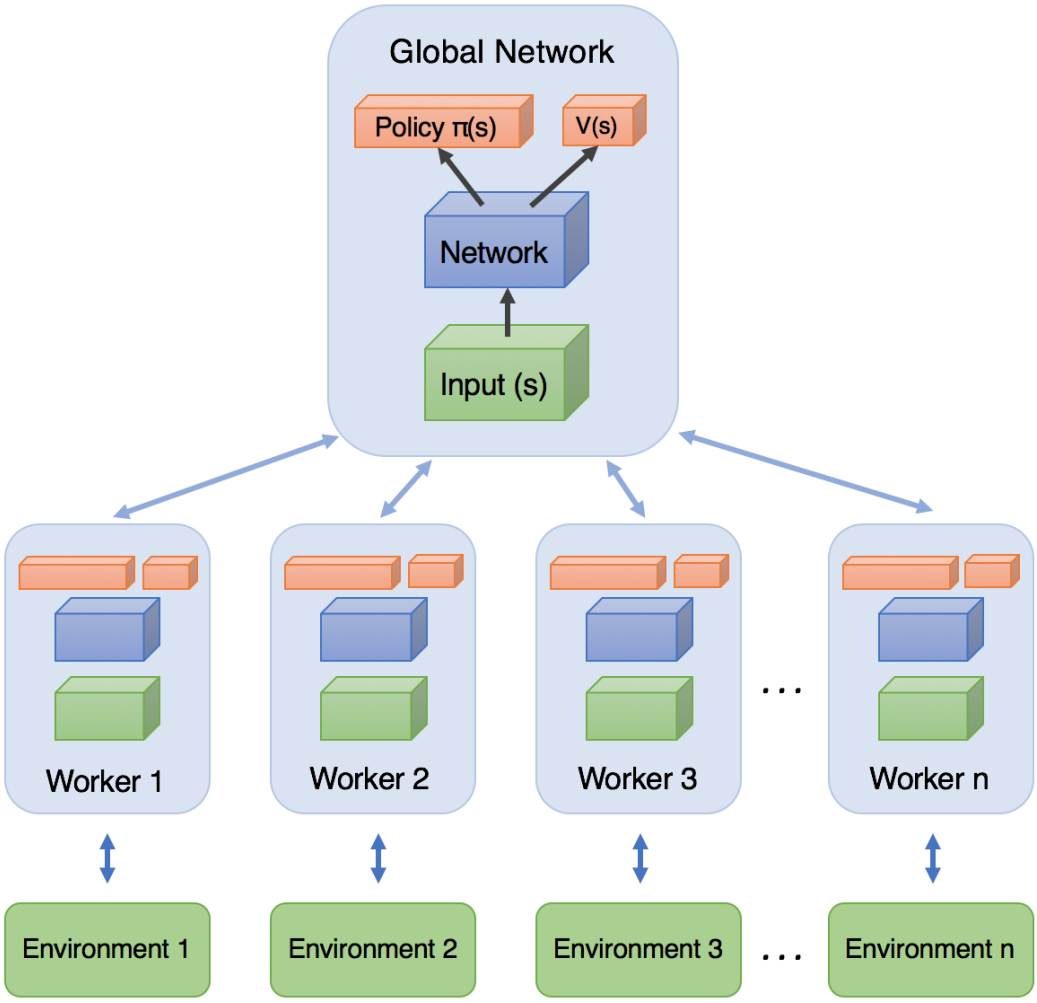

Figure 20

4.4 ADVANTAGE

There are two losses in the algorithm: Value Loss related to V(s) and Policy Loss related to Q(s,a).
For Value Loss, the actual value of V(s) is compared to network produced Value.

$$\text{Value Loss: } L = \Sigma(R - V(s))^2$$

This Value Loss is backpropagated through the network to update the weights.

For Policy Loss,

$$\text{Policy Loss: } L = -\log(\pi(s)) * A(s) - \beta * H(\pi)$$

The Policy Loss is dependent on the value of Advantage (A), which is the difference between the Q value of the action chosen to play to reach the current state and the Value of the state. A is used to calculate the policy loss which is then backpropagated and weights are adjusted in such a way that A is maximized.

4.5 IMPLEMENTATION OF ALGORITHM

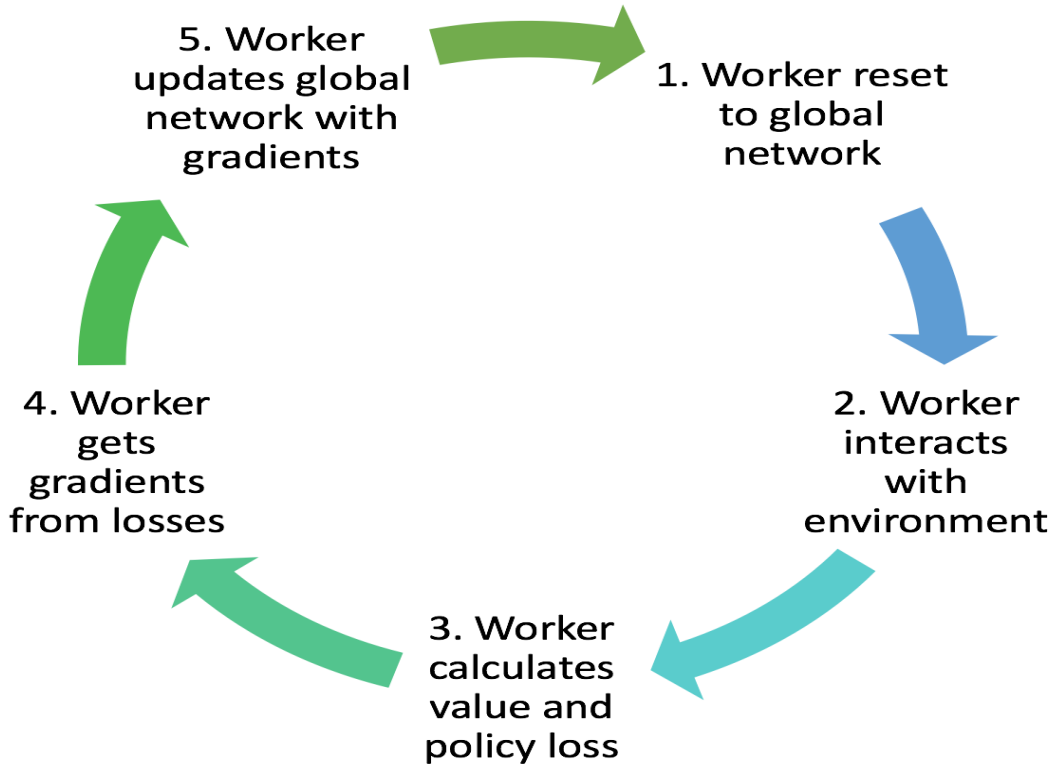

Figure 21

online actor-critic algorithm:

1. take action $\mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{s})$, get $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
2. update $\hat{V}_\phi^\pi$ using target $r + \gamma \hat{V}_\phi^\pi(\mathbf{s}')$
3. evaluate $\hat{A}^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \hat{V}_\phi^\pi(\mathbf{s}') - \hat{V}_\phi^\pi(\mathbf{s})$
4. $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) \hat{A}^\pi(\mathbf{s}, \mathbf{a})$
5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

This is the pseudo code of the algorithm.

4.6 RESULT AND OUTPUT PLOTS

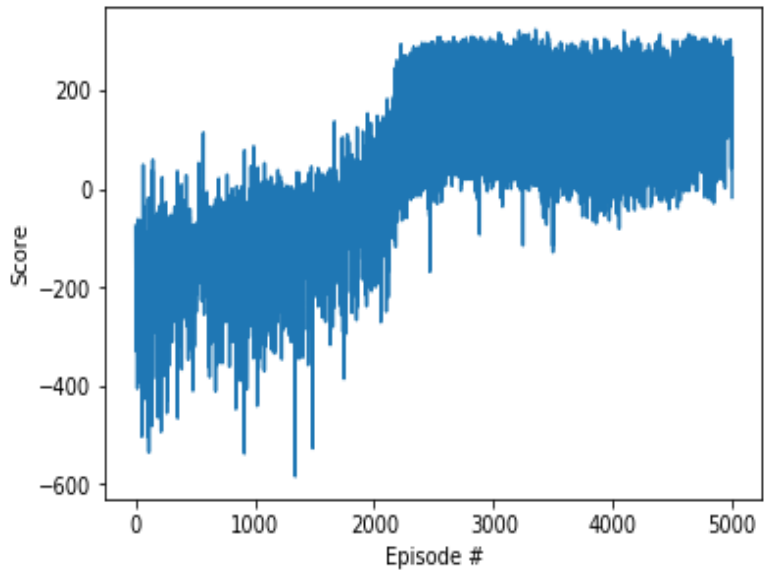

Figure 22

The plot shows the rewards for each episode. The rewards have a lot of variance over the whole training as can be seen. It also took a lot of episodes to train the model. Around 3000 episodes, the model got it's first reward of above 200. The rolling average of previous 100 episodes vs iterations look as follows:

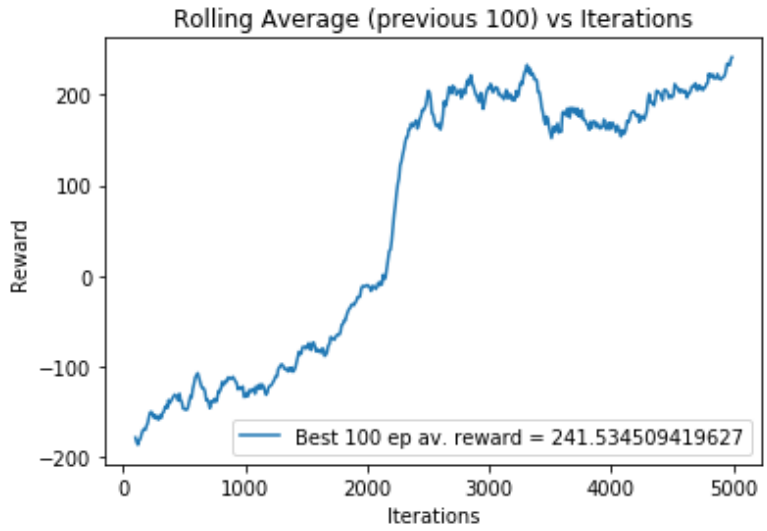

Figure 23

The performance was not as good as the other algorithms for A3C for this environment. After training, it landed around 25 times out of 50 episodes during test.

The Performance can be seen on the following YouTube Video link: https://www.youtube.com/watch?v=H7zdaunhSOg&feature=youtu.be

4.7 CONCLUSION

In A3C each agent talks to the global parameters independently, so it is possible sometimes the thread-specific agents would be playing with policies of different versions and therefore the aggregated update would not be optimal. This is the case seen here, where the algorithm performs worse than the other two algorithms.

This is mainly because the environment keeps changing the position of the flag at each iteration.As a result, the different versions have different policies which make learning difficult.

CHAPTER 5. SUMMARY, CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSION

Thus, after applying the three algorithms on the new Lunar Orbiter environment, we came to the following conclusions:

1. For the environment, Policy Gradient Algorithm worked the best. Though it took more episodes to train the algorithm but it had the best average landing performance.
2. The Q-Learning Algorithm performed pretty well and had the second best performance. It also took the least amount of time to converge and train.
3. Actor Critic was the worst performing algorithm for this particular environment. It took the most time to train and still gave worse performance.

RL Algorithms	Episode Convergence	Average Landing Performance
Q-Learning	~550 episodes	30/50
Policy Gradient	~600 episodes	35/50
Actor Critic	~3000 episodes	25/50

Table 2

In the table, Episode Convergence indicates the number of episodes needed to train the algorithm.
Average Landing Performance indicates the number of times the lander landed during testing out of 50 episodes.

5.2 FUTURE WORK

1. Improvement on the algorithms is always a target for future work, with, more time given to hyperparameter tuning.
2. We can also work on the environment to make it more random by making the flag move during the episode on a platform. This will further increase the difficulty of the environment.
3. Add more variables to the rocket state, such as the location of the highest mountain, whether there are slopes near the flag, etc. These observability and information are reasonable, because landing a rocket in real world even needs more information to make sure a 100% successful rate.

CHAPTER 6. LIST OF REFERENCES

The list of references are as follows:

1. David Silver, UCL Course on Reinforcement Learning, https://www.youtube.com/watch?v=2pWv7GOvuf0
2. Gabriel Garza, Blog Post on Medium, https://medium.com/@gabogarza/deep-reinforcement-learning-policy-gradients-8f6df70404e6
3. Gaudet, Brian, Richard Linares, and Roberto Furfaro. "Deep reinforcement learning for six degree-of-freedom planetary powered descent and landing." *arXiv preprint arXiv:1810.08719* (2018).
4. https://github.com/nikhilbarhate99/TD3-PyTorch-BipedalWalker-v2
5. https://github.com/Jeetu95/Rocket_Lander_Gym
6. https://gym.openai.com/envs/#box2d
7. https://github.com/openai/gym#environments
8. https://github.com/gregd190/AI-Gym---LunarLander-v2-Actor-Critic
9. Thomas Simonini, Blog Post, https://medium.freecodecamp.org/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d
10. Q-Learning Wikipedia, https://en.wikipedia.org/wiki/Q-learning
11. Introduction to Q-Learning, https://medium.freecodecamp.org/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc
12. Deep Reinforcement Learning, https://ipvs.informatik.uni-stuttgart.de/mlr/wp-content/uploads/2018/04/18-deeprl-intro.pdf
13. Deep Q-Network, https://www.slideshare.net/guo_dong/dqn-deep-qnetwork