Solving Control Problems using Reinforcement Learning

Liam Bugeja Douglas
liam.bugeja-douglas.20@um.edu.mt
University of Malta, Msida, Malta

1 INTRODUCTION

Reinforcement learning (RL) is a type of machine learning technique in which an agent learns to make decisions through interaction with the environment and receives feedback in the form of positive or negative rewards[1]. The agent's purpose is to discover a policy that maximizes the cumulative reward over time. In RL, an agent interacts with the environment in a series of steps also know as epochs or episodes. The agent monitors the current state of the environment and chooses an action to do at each time step. The agent receives a numerical reward signal after completing the action, and the environment changes state.

While RL and machine learning (ML) are closely related fields, there are significant variations between them. In ML, the agent is trained with labeled data and learns a function to make high quality predictions, but in RL, the agent learns by interacting with the environment and learning from feedback in the form of rewards and penalties. Furthermore, the agent in ML receives immediate feedback on its predictions, whereas the agent in RL often receives delayed incentives in order to optimize the cumulative reward overtime. Finally, the agent in ML is provided a predetermined dataset, whereas the agent in RL must explore the environment and balance exploitation and exploration to find the best suited policy [2].

There are three main approaches in RL for learning a policy for an agent to interact with its environment: value-based, policy-based, and actor-critic. Each strategy has advantages and disadvantages, and the best approach to utilize depends on the given problem.

- Value-based approaches: Value-based methods estimate the predicted long-term return or value of a state or a state-action pair. The policy of the agent is then determined from the value function by selecting the action that maximizes the expected value at each stage. Some examples of well known value-based approaches include Q-learning and SARSA.
- Policy-based methods: Policy-based approaches learn the policy directly, in which these policies map states to actions. The policy of the agent is represented by a function, which is changed throughout training to improve policy. The agent's goal is to maximize the policy's predicted cumulative reward. Some examples of well known policy based methods are REINFORCE and Policy Gradient.
- Actor-Critic Methods : Actor-Critic methods are a hybrid of value-based and policy-based methods. They are divided into two networks: the actor network and the critic network. The policy is represented by the actor network, which is adjusted to increase the projected cumulative reward. The value function is represented by the critic network, which is updated to predict the expected cumulative reward under the current policy. The gradient of the critic network is used to update the actor network. Examples of well known actor-critic methods are Advantage Actor-Critic (A2C) and Proximal Policy Optimization(PPO).

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND INFORMATION

2.1 Value Based Methods

As previously mentioned value-based methods are based on the idea of maximising the long-term return or value of a state or state-action pair. The key equation used in such methods is the Bellman equation which expresses a state's value in terms of the values of the states that can be reached from it:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} \sum_{r \in R} p(s', r|a, s)(r + \gamma v_\pi(s'))$$

[2]

Where:

- $V(s)$ is the value of state s under policy π.
- π(a|s) is the probability under policy π of taking action a in state s.
- $P(s', r|s, a)$ is the state transition probability, which is the probability of transitioning to state s' and receiving the reward r after taking action a in state s.
- S set of all possible states.
- A set of all possible actions.
- R set of all possible rewards.
- γ is the discount factor, which is a number between 0 and 1 and gives determines the importance of future rewards.

Value-based approaches seek to discover an ideal value function that approximates the true value of a state or a state-action pair.Q-learning is a prominent and commonly used value-based method that learns the action-value function Q(s,a) by updating it using the Bellman equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t)]$$

[2]

Where:

- $Q(S_t, A_t)$ is the estimated value of taking action A_t in state S_t.
- α is the learning rate which determines the the importance of the new value compared to previous values.
- R_{t+1} is the reward value at time step t+1.
- γ is the discount factor, which is a number between 0 and 1 and gives determines the importance of future rewards.
- S_t and S_{t+1} are the states at time t and $t+1$ respectively.
- A_t is the action at time step t.
- $\max_{a \in A} Q(S_{t+1}, a)$ is the maximum reward that can be obtained over all possible next actions a'.

Using the observed reward r and the predicted best probable future value Q(s',a') given the current policy, this equation is used to update the current estimate of Q(s,a). The objective is to find the best action-value function Q*(s,a) that satisfies the Bellman equation. If the learning rate is adjusted correctly, this update rule assures that Q(s,a) does not decrease over time and that Q(s,a) converges to the optimal action-value function Q*(s,a) as the number of repetitions approaches infinity.

Q-learning is an off-policy algorithm, which means it can learn the optimal Q-function while adhering to any policy, even if it is sub-optimal. This is due to the Q-function being updated based on the greatest predicted value of the next action rather than the action taken by the agent [3]. The following code snippet describes how the Q-values are updated in the Q-Learning algorithm.

```
Q[s][a] = Q[s][a]
        + learning_rate *
        (reward + discount_factor
        * max(Q[new_state])
        - Q[s][a])
```

Another popular value-based method is SARSA (state action reward state action) and is also referred as Modified Q-Learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

[2]

SARSA algorithm updates Q-value estimates based on the action that the agent would do in the next state rather than the action with the highest estimated value, as in Q-learning. This will result in learning a different Q-function and converge to a different policy, which may be sub-optimal [4]. The following code snippet describes how the Q-values are updated in the SARSA algorithm:

```
Q[s][a] = Q[s][a]
        + learning_rate * (reward + discount_factor
        * Q[new_state][new_action]
        - Q[s][a])
```

2.2 Deep Q-Networks algorithm

Deep Q-Networks (DQN) is an algorithm that combines Q-learning with a neural network to alleviate some of the constraints of classic Q-learning in high-dimensional and continuous state spaces. The core idea behind DQN is to employ a neural network known as the Q-network to approximate the Q-function, which represents the expected return for each action performed in a given state. The four important concepts of DQNs are:

- Q-learning: DQNs learn the Q-function using Q-learning, a type of temporal difference learning, which was discussed in the previous section.
- Neural Networks: DQNs employ a neural network to approximate the Q-function. This enables the algorithm to deal with multidimensional states and continuous actions. Neural Networks are powerful function approximators that can generalise well and capture non-linearities in the underlying problem by using numerous hidden layers. In DQNs, the neural network receives a state as input and returns a Q-value for each possible action. The network is trained to minimize the difference between predicted and target Q-values as calculated by the Bellman equation.
- Experience Replay: The technique of experience replay retains a buffer of past experiences that can be sampled at random to update the Q-values. This allows the agent to decouple its experiences from their temporal correlation. Temporal correlation describes how the agent's experiences are connected with past experiences it has had in a the environment, which can slow down the learning process. The agent can sample from a range of experiences via experience replay, which can speed up the learning process and increase the algorithm's stability.
- Target Network: The target network is a duplicate of the core neural network that is utilized during training to generate the target Q-values. This allows the agent to stabilize the training process and avoid Q-value fluctuations. The target network is updated less frequently than the primary network, and its parameters are fixed for some time before being updated with the parameters of the primary network. Using the target network helps the primary network to converge on a constant and stable target rather than allowing the target to shift rapidly. Furthermore, using the target network helps to reduce the correlation between actions and target values, which can assist reduce variation in updates and lead to more stable learning.

To solve some of the original Deep Q-Network (DQN) algorithm's shortcomings and improve its performance, various upgrades and extensions have been proposed in recent years. Among the most notable additions are:

- N-Step Bootstrapping: N-step bootstrapping is an addition to the Q-learning technique that estimates Q-values by looking at numerous steps in the future rather than just the next step. This can help the agent learn more effectively by considering the long-term effects of its actions. N-Step Q-Learning was introduced by Watkins and Dayan in [5].

- Double DQN (DDQN): Double DQN is a modification of the DQN algorithm that solves the problem of Q-value overestimation. DDQN employs two networks: one to select the action to be taken and another to estimate the Q-value of that action. This helps to decrease overestimation and increase the learning process's stability. This technique was introduced in 2016 by [6].

- Dueling DQN: Dueling DQN is a DQN algorithm extension that separates the estimation of a state's value from the estimation of the advantage of each action. This enables the network to learn more efficiently by more effectively generalizing learning across actions. This technique was first introduced in [7].

- Noisy DQN: Noisy DQN is a DQN method that incorporates noise into the network's parameters to increase exploration. This is accomplished by introducing a noise source into the network's weights, which can be controlled by the agent and can be effective in overcoming over-fitting and improving exploration. The Noisy DQN method was first introduced by [8].

These are some of the most well-known DQN algorithm advancements, but there are many more that have been proposed and developed.
The following code illustrates how the DQN algorithm updates the Q-values at each time step:

```
target_qvalues = dqn_target(b_next_states)
max_target_qvalues = torch.max(target_qvalues)
expected_qvalues = b_rewards + GAMMA *
                  (1 - b_dones.type(torch.int64))
                  * max_target_qvalues
```

2.3 Policy Based Methods

Policy-based techniques, as previously discussed, are based on the concept of directly learning the policy, thus it maps states to actions. This is done by updating the weights or the parameters of the policy which is denoted by θ of the current policy. Policy based methods are mostly used in problems having continuous action spaces or high-dimensional data. The following equation is an update rule for the policy parameters in the policy gradient algorithms:

$$\theta_{t+1} = \theta_t + \alpha \nabla \pi_{\theta_t}(a^*|s)$$

Where:

- θ_t and $\theta_t + 1$ is the policy at time step
- α is the learning rate which determines the the importance of the new value compared to previous values.
- $\nabla_\theta \pi(\theta_t, a^*|s)$ represents the policy's gradient π with respect to the parameters θ_t using the optimal action a^* taken at state s.

However, the equation assumes that the best action $a*$ is already known in the environment, however in many real word problems the best action is unknown. Therefore, the equation needs to compensate for the fact that more likely actions will be taken more frequently.

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta \log \pi_{\theta_t}(a|s)$$

In which G_t represents the cumulative reward of the current episode.

REINFORCE is a popular policy-based algorithm which was introduced in [9]. The gradient is determined using Monte Carlo sampling by determining the average of the product of the gradient of the log policy and the discounted cumulative reward for a set of completed episodes.

$$\theta = \theta + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \theta)$$

The following code is the update function for the REINFORCE algorithm in python

```
1    while not done:
2        #Append observations
3        observations.append(observation)
4        #Choose action based on policy
5        action, prob = policy(observation)
6        #Make next step based on policy
7     observation, reward, done, info = env.step(action)
8        #Append values
9        totalreward += reward
10       rewards.append(reward)
11       actions.append(action)
12       probs.append(prob)
```

The algorithm loops until the current episode ends, whilst appending the values observed by the agent. The action is decided based upon the policy given in line 5.

One important distinction between policy-based and value-based methods is that policy-based methods directly optimize the policy, whereas value-based methods first estimate the value function and then extract the best policy from it. Furthermore, because they are often represented as a probability distribution over the action space, policy-based methods may handle continuous action spaces more easily than value-based methods. Another point to consider is that in value-based approaches, the agent makes decisions based on the value of a state or a state-action combination, but in policy-based methods, the agent makes decisions based on the probability of choosing an action in a state, which is the output of the policy function. These distinctions were delved deeper in [10].

2.4 Actor-Critic Methods

Actor-Critic methods are a subclass of RL methods that incorporate the advantages of both value-based and policy-based approaches. These methods concurrently learn a value function and a policy, and then use the value function to enhance the policy. An Actor-Critic algorithm is made up of two neural networks: the actor network and the critic network [2]. The actor network maps states to actions and therefore it determines the optimal policy by defining the best actions for each state. The network outputs a probability distribution over all the possible actions in the current state and, in turn this will maximise the expected cumulative reward. On the other hand, the critic network provides a value function by evaluating state-action pairs, this is done by taking the current state and action taken by the actor network and outputs an expected reward prediction. Therefore, the critic network is trained to provide an accurate prediction reward to provide feedback to the actor network to take the best action[2].

A well known actor-critic algorithm called Actor-Critic with Advantage (A2C) makes use of the mentioned networks and are trained in parallel environments to maximise the expected cumulative reward. Furthermore, it makes use of a variable called the advantage value which is based on the outputs of both the actor and critic network. The advantage value calculates the expected return from each state-action pair, which allows the policy to focus on states which improve the cumulative reward [11]. It is calculated as follows:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

Where:

- st is the state at time step t.
- at is the action at time step t.
- $Q(st, at)$ is the expected reward in state st taking action at.
- $V(st)$ is the value function which is the expected return from the state st.

However as stated in [11] by using both V and Q it would require two separate neural networks which is very inefficient. The author proposes to use the bellman optimality equation due to their relationship to make use of one neural network instead. Therefore the advantage value can be written as:

$$A(s_t, a_t) = r_t + 1 + \lambda V(s_t + 1) - V(s_t)$$

Finally we can input the advantage value in the policy gradient equation in which we defined in the previous section as follows:

$$\theta_{t+1} = \theta + \alpha A(s_t, a_t) * \nabla log \pi_\theta(a|s)$$

The advantage value is calculated as follows in python:

```
target = T(reward).unsqueeze(1) +
         GAMMA * next_state_value
advantage = (target - state_value).detach()
```

Another well known actor-critic algorithm is the Proximal Policy Optimisation (PPO), which is an on-policy algorithm which improves the training stability by limiting the changes that can be made to the policy per episode. This is done since smaller policy updates per episode tends to correlate with better convergence to the optimal solution. Furthermore, large policy changes tend to result in large fall-offs in score and would take many episodes or even no possibility in recovery. The PPO algorithm was introduced in [12] were the above ideas were discussed.

PPO improves the standard actor-critic methods by using the Clipped Surrogate Objective. The objective of this is to improve the training stability as mentioned before. The equation to update the parameters is as follows:

$$\mathcal{L}^{\mathcal{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A_t}, clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A_t} \right) \right]$$

In which $r_t\theta$ is the ratio function which is responsible for giving the ratio probability of taking action a in state s given the current policy versus the previous policy. This is defined as follows in [12]:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

If the ratio is outside the range which is dictated by the ϵ , the clip function returns the corresponding boundary value, effectively limiting the magnitude of the policy update. The clipping parameter ϵ determines the range of values around 1 within which the ratio is allowed to vary.

The following is the python code for the PPO training algorithm:

```
def train(mem, gamma, batchsize, epoch_repeat, epsilon, lmbda):
  for epochrep in range(epoch_repeat):
      for batch in mem.batches(batchsize=batchsize):
        obs, actions, logprobs, rewards, dones, obs_,
            values, values_, gae = batch
      gae = (gae - torch.mean(gae)) / (torch.std(gae) + 1e-6)
            target = gae + values
            state_values = critic_net(obs)
            critic_loss =
      F.smooth_l1_loss(state_values, target).mean()

             new_loc, new_scale = actor_net(obs)
       dist = Normal(loc=new_loc, scale=new_scale+1e-6)
       new_logprobs = torch.sum(dist.log_prob(actions)
             , dim=-1, keepdim=True)
             rho = torch.exp(new_logprobs - logprobs)
             surrgt1 = rho * gae
       surrgt2 = rho.clamp(1-epsilon, 1+epsilon) * gae
       policy_loss = -torch.minimum(surrgt1, surrgt2).mean()

             loss = policy_loss + 0.5*critic_loss
             actor_optimizer.zero_grad()
             critic_optimizer.zero_grad()
             loss.backward()
             actor_optimizer.step()
             critic_optimizer.step()
```

In which the ratio is calculated in line 13 as the variable rho containing the new and previous policy, which is then used in the advantage values for the clipped and unclipped ratios in lines 14 and 15. Finally, the best policy is chosen in 16 where it the lowest value is saved.

3 METHODOLOGY

3.1 LunarLander Problem

The lunar lander issue is a classic reinforcement learning problem in which an agent must learn to manage and successfully land a lunar lander spacecraft on the moon's surface created by Open AI [13]. To reach its landing place, the lunar lander must navigate a dangerous environment of craters and mountains. The problem is frequently implemented as a simulation in a 2D environment, such as Box2D, which gives a physics-based simulation of the movement of the lunar lander. The environment replicates the lander's physical interactions with its surroundings, which include the effects of gravity and air resistance. The lunar lander's action space normally consists of two primary actions: firing thrusters to travel upward or to the left or right. The thrusters power and duration can be adjusted to vary the lander's velocity and direction. The action space can either be discrete or continuous. In the discrete action space the agent must choose one of a limited number of actions at each time step. On the other hand, the continuous action space requires the agent to choose a continuous value for its action. Therefore, it is easier for the agent to learn in the discrete action space due to the environment having a limited number of possible actions. The rewards in the environment are distributed as follows:

- Moving from the top of the screen to the landing pad and coming to rest: 100-140 points.
- Moving away from the landing pad: Lose reward.
- Crashing: -100 points.
- Coming to rest: +100 points.
- Each leg with ground contact: +10 points.
- Firing the main engine: -0.3 points per frame.
- Firing the side engine: -0.03 points per frame.
- Solved: 200 points.

The purpose of this task is to successfully land on the landing pad without crashing the lander and achieve an average score of 195 for the last 50 episodes.

3.2 Experiments and Results

We hope to tackle the lunar lander problem utilizing multiple reinforcement learning methods in this experiment. To address the discrete action space, we will use the Vanilla DQN method and the Policy Gradient Neural Net with parallel environments. For the continuous action space, we will use the Advantage Actor Critic (A2C) and Proximal Policy Optimization (PPO) algorithms. In each experiment the Pytorch library was used to build the model utilizing the different methods including neural network functions and optimisers. Additionally, the numpy library was also used to handle mathematical functions such as means, standard deviation and much more. Finally, the Matplotlib library was used to display the results in a graphical form.

3.2.1 Vanilla Deep Q-Network. For the DQN network we made use of the Vanilla DQN, additionally, the network consisted of the input and output layer having a ReLu activation function in between and uses the Adam optimiser. We first experimented with the hidden inputs of the network and tested using 16 , 32, 64 and 128. With a hidden size of 16 the model took around 1750 episodes to converge to an average score of 195. By increasing the hidden size however this was decreased, and with the largest size of 128 hidden units the model took only 450 episodes to converge. However, we noticed that this was also computationally more expensive as each episode took longer to complete. The below figure showcases the model with 128 hidden units.

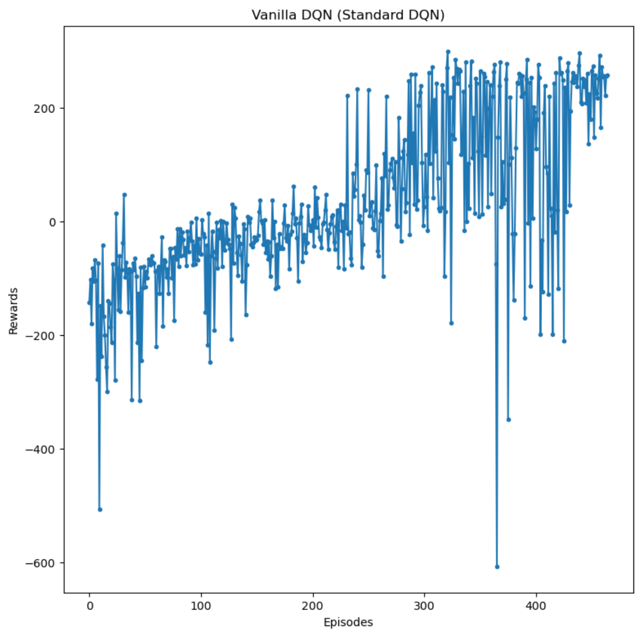

Figure 1: DQN 128 Hidden Units.

After testing the hidden layers we also experimented using different learning rate values using 0.01, 0.001 and 0.0001. It was noted that when increasing the learning rate the model would not converge and would be stuck in a local minima. Furthermore, increasing the learning would also make the model more unstable as the average score would increase and decrease in large values. The figure below shows the model with a learning rate of 0.0001 as it was found the best.

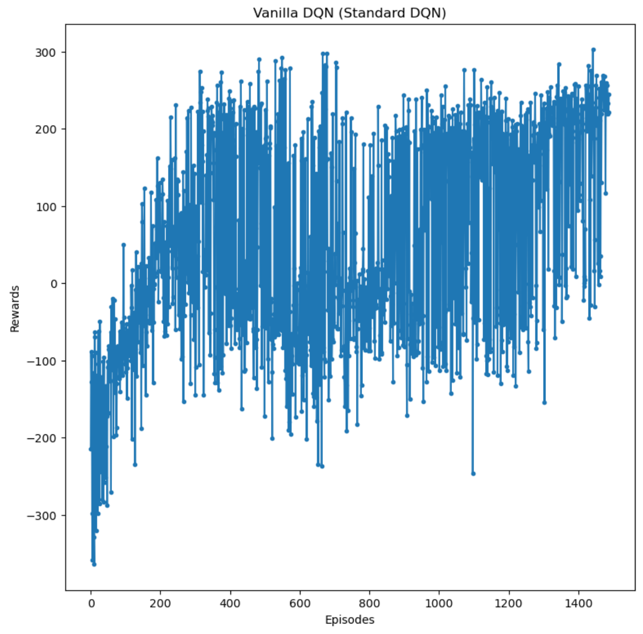

Figure 2: DQN 0.0001 Learning Rate.

Therefore, the best model parameters in the DQN algorithm from the results would be using 128 hidden units with a small learning rate of 0.0001.

3.2.2 Policy Gradient Neural Network. For the policy based algorithm we made use of a Policy Gradient Neural Network, which consisted of one hidden layer with a ReLu activation function and a Softmax activation function on the output layer. The softmax function is used to output the probability of each action which in turn the network predicts the best action to take. We first noted that the algorithm was not able to converge with different learning rates, hidden sizes ,and hidden layers. However after further research such algorithms can make use of parallel environments in which it has been proven to improve the convergence of neural networks such as in [14]. We then initialised 5 parallel environments, in which the model converged to the average results of 195 after sometime. Next we increased the number of parallel environments, we noted that the model was taking longer to train and was very inefficient and decided to stay with only 5 parallel environments.

We experimented with the size of the hidden units of the network and tested using 16, 32, 64 and 128. The best results came from the smaller hidden units (16, 32) as this took the least effort to converge, both models took almost 450 episodes to have an average results of 195. On the other hand, the models having the larger hidden units (64, 128) did not converge properly even after 1000 episodes.The figure below shows the model containing 32 hidden units as it was found as the best.

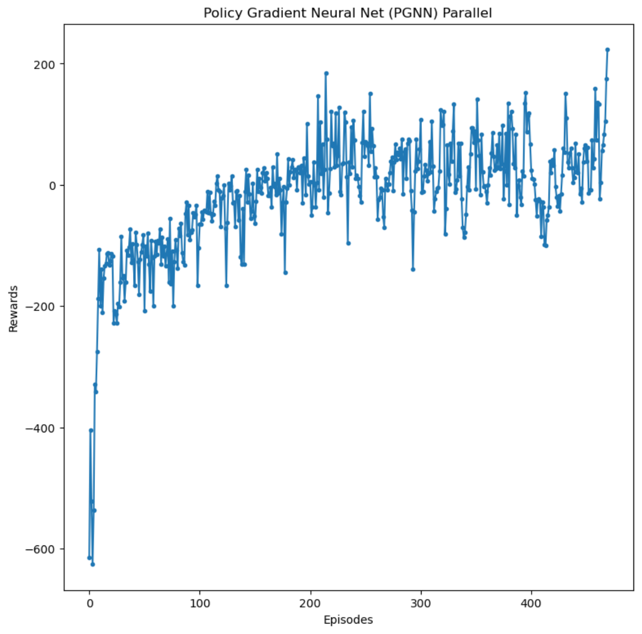

Figure 3: PGNN 32 hidden units.

Next we experimented using different learning rates for the model, we tested the learning rates with the model having the size of the hidden units equal to 32. We tested using the following values 0.03, 0.003 and 0.0003. Using smaller values lead to the model converging earlier and the higher learning rates tend to get stuck in a local minima.The figure below shows the model containing 32 hidden units and a learning rate of 0.0003.

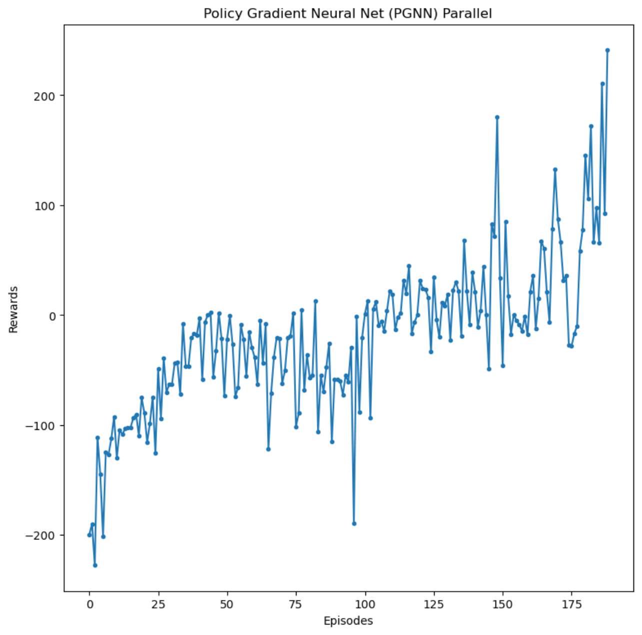

Figure 4: PGNN 0.0003 Learning Rate.

Therefore, the best model parameters in the PGNN algorithm from the results achieved would be using 32 hidden units with a learning rate of 0.0003.

3.2.3 *Advantage Actor Critic.*

The first actor-critic method in which we experimented with was the Advantage Actor Critic (A2C) algorithm, the model contains 1 hidden layer having two ReLu activation functions between the input and output layer. Furthermore, the algorithm made use of parallel environments to converge at earlier episodes. Whilst testing the A2C algorithm we noted that it was less sensitive to different parameters and would converge given enough time with the different parameters values. We firts experimented with the hidden units of the algorithm with the following values: 16,32 ,64 ,128. We noted that by increasing the size of the hidden units, the model would converge in early episodes. The best model was found to be the one containing 128 hidden units, which can be seen below

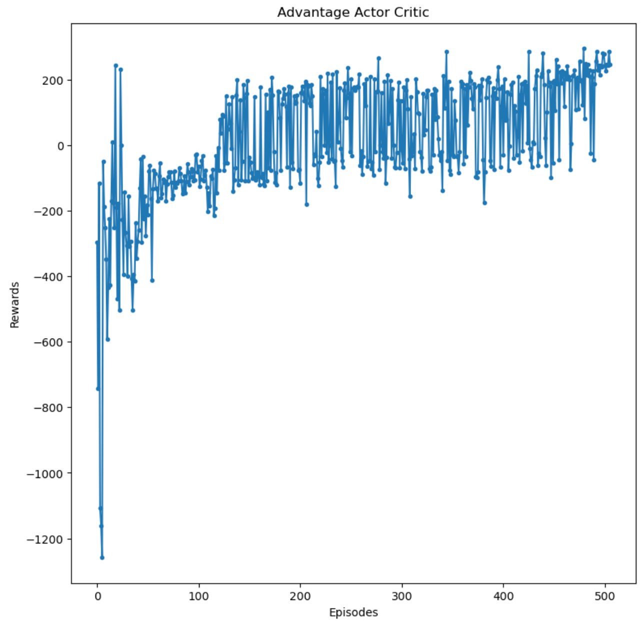

Figure 5: A2C 128 Hidden Units.

Furthermore, when increasing the size the model would also be more stable and fluctuate less. The following learning rate values were tested 0.01 , 0.001 and 0.0001. The smallest learning value (0.0001) achieved the best result converging at an average episode of around 500.

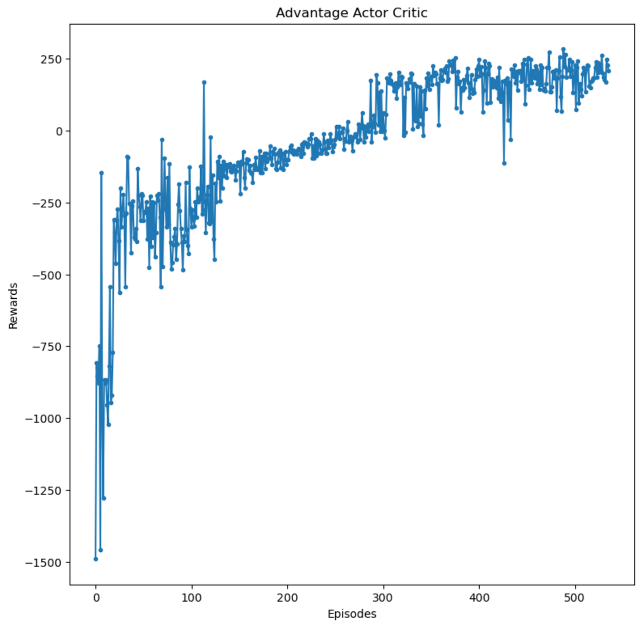

Figure 6: A2C 0.0001 Learning Rate.

Therefore, the best model parameters in the A2C algorithm from the results achieved would be using 128 hidden units with a learning rate of 0.0001.

3.2.4 *Proximal Policy Optimization.*

The second actor-critic method and final algorithm in which we experimented with was the Proximal Policy Optimization (PPO) algorithm. The algorithm consisted of two network one for the critic and another for the action in which both had similar structure, having one dense layer containing a TanH activation function between the input and output layer. The model also made use of parallel environments to converge at an early rate as well. We first experimented with different hidden units with the following values: 16, 32, 64, 128. We noticed that the at certain times the PPO would flatline for a number of episodes and continue normally afterwards. However, each different parameter still converged at an average of 195 with the best parameter being of 16 hidden units achieving convergence at episode 250.

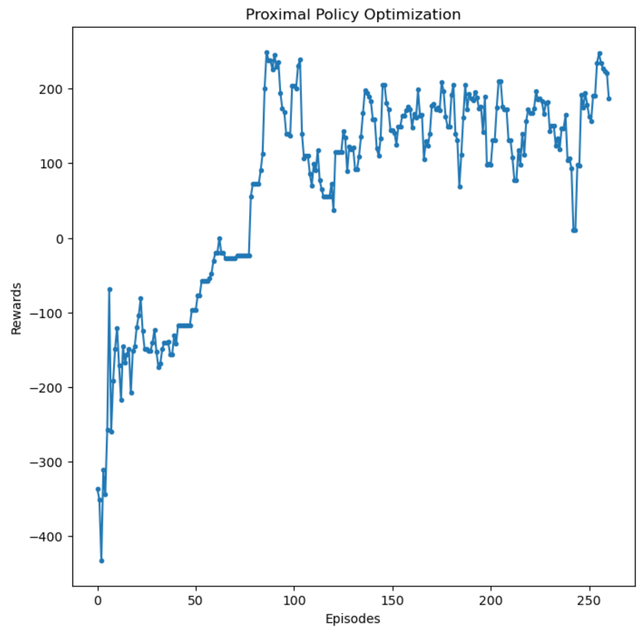

Figure 7: PPO 16 Hidden Units.

When testing for the different learning rates, the model only reached convergence with one learning rate being the smallest one. The following learning rate values were tested:, 0.001, 0.0001. When testing for learning rate 0.01 value errors always occurred and therefore the parameter could not be tested .The following figure shows the best learning rate achieving convergence in episode

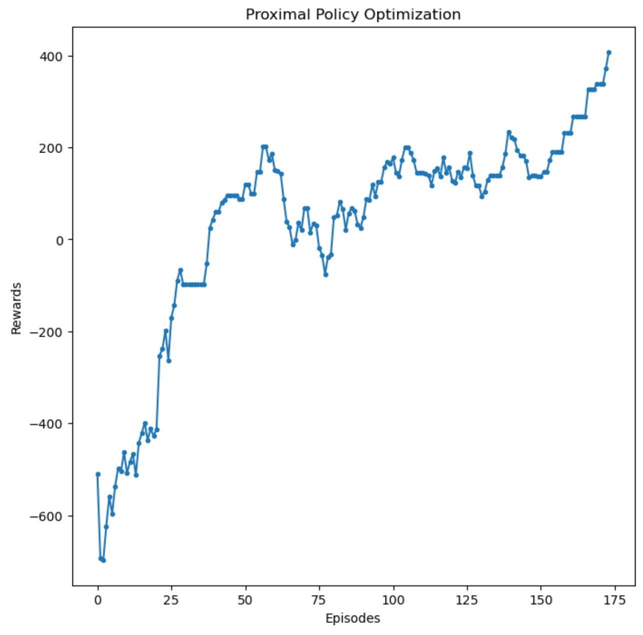

Figure 8: PPO 0.0001 Learning Rate.

Therefore the bet parameters from this experiment for the PPO algorithm would be using a hidden size of 16 with a learning rate value of 0.0001.

3.2.5 Graphs. Due to space constraints each different parameter which was tested cannot be displayed, however, the graphs are available in a separate folder called Graphs.

4 CONCLUSION

Overall, this assignment demonstrated the practical application of advanced Reinforcement Learning concepts , and their ability to solve the LunarLander Problem. Furthermore, in the most recent 50 episodes, each algorithm achieved an average result of 195, indicating that each algorithm has solved the lunar lander problem.

REFERENCES

[1] Shweta Bhatt. Reinforcement learning 101, Apr 2019.
[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
[3] Chathurangi Shyalika. A beginners guide to q-learning, Jul 2021.
[4] Adesh Gautam. Introduction to reinforcement learning (coding sarsa) - part 4, Mar 2019.
[5] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
[6] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
[7] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *International Conference on Learning Representations*, 2016.
[8] Meire Fortunato and Mohammad Gheshlaghi Azar. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.
[9] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.
[10] Sarthak Bhagat, Hritwick Banerjee, and Hongliang Ren. Deep reinforcement learning for soft robotic applications: Brief overview with impending challenges. 2018.
[11] Chris Yoon. Understanding actor critic methods, Jul 2019.
[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
[13] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.
[14] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.