Lunar Landing with Reinforcement Learning

Sergio Robledo

1. Definition

Machine learning is integrated in today's society even if people are not aware of it. For example, anyone that watches Netflix has experienced a machine learning algorithm whenever Netflix recommends movies or TV shows the person might be interested in. This is a form of unsupervised learning where the algorithm does not have someone to tell it what the right answer is; instead, it tries to deduce patterns and clusters movies together, and next time you watch a movie the algorithm sees what movies it has classified to be similar to the one you just watched and recommends it. One the other hand, there is supervised learning where the algorithm is fed a data with the correct answers and tries to find connections between the input data and the correct answer. There are also many supervised learning algorithms constantly working around us; for example, face recognition used in many mobile devices today, has to be trained on your face the first time you use it. Once it has learnt a sufficiently good representation of your face, next time you try to use it to open your phone, it will be able to compare if the face trying to open the phone matches characteristics of the face it was trained on. If it is sufficiently close to the trained face, it will unlock and will use that unlocking to better learn your face. Therefore, every time it unlocks, it has new data with a correct face dataset.

The state-of-the-art machine learning algorithms are being used today in a wide variety of tasks from self-driving cars to skin cancer detection. Many of these state-of-the-art algorithms implement a form of supervised learning called deep learning which involves the use of neural networks with multiple layers and many learning nodes, or neurons, containing the values, or weights, that transform the input to an output. In skin cancer detection an image is used as input to the deep neural network and the neural network will output the type of skin lesion it thinks it is with a certain level of probability; this is the end of a training/testing session. However, not everything can be modeled in a quick one step process. For example, self-driving cars do not stop controlling the car after one second of driving; instead, they are required to process large amounts of data for every fraction of a second that passes by.

This is where reinforcement learning comes in; reinforcement learning algorithms allow the neural networks to account for time as one the parameters of the learning process. Reinforcement learning algorithms use a reward that can be obtained at every time step of the task it is trying to perform; for example, in a quadcopter takeoff, the reward of the quadcopter could increase at every time step if it gets closer to the target position desired by the user. Therefore, the reward acts as the correct answer, since the better the reward, the better the agent is performing in the environment.

1.1. Project Overview

One of the variations of reinforcement learning, known as an actor-critic method, utilizes an actor that decides what action to take given the state of the environment, and the critic judges how good of an action it was given the state by mapping the state-action pair to an action value, or Q value. Further, when the actors and critics are deep neural networks, it becomes a deep reinforcement learning algorithm. After spending countless hours developing a deep deterministic policy gradient, or DDPG, algorithm which is a type of deep reinforcement algorithm, I believe a variation of that network can be applied to more complicated domains. That's when I got the idea of training the DDPG to land a lunar spacecraft at a certain position

on the moon. Attempting to solve problems in a simulated environment without any real-world consequences, is an essential step towards the implementation of the real-world thing that will have real negative consequences if the agent selects a wrong action. In order to simulate the lunar landing environment, I will be using OpenAI's "LunarLanderContinuous-v2" environment; this version of the lunar lander has both continuous state and action spaces.

The only data my reinforcement learning agent will be using is the output of the lunar lander environment as the agent interacts with it; the lunar lander outputs include positions, velocities and other variables that describe the current state of the lander at different time periods. One of the most important outputs of the lunar lander environment is the reward given to the learning agent for making a certain action in a certain environment state. Once a state is given, the agent will predict the best action for the spacecraft to make, whether it be turning on the right, left, main engines or no engine. This loop will be repeated until the environment ends, either through the lunar landing crashing or running out of time steps. The goal of the agent is to maximize its reward by learning what actions output the highest reward in a given state.

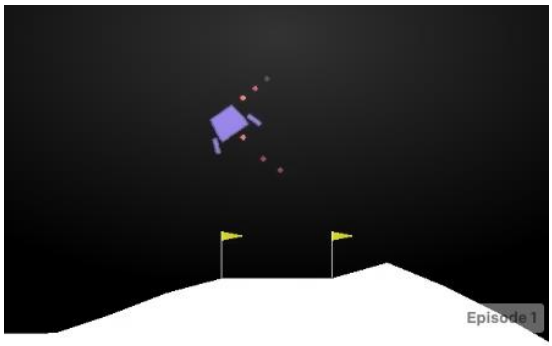

Figure 1: Lunar Lander Continuous Environment

1.2. Problem Statement

The task begins, once the lunar lander environment is reset, and the initial state is given. The agent will make a prediction of what the best action is given the current state of the environment; whether it be turning on the right, left, main engines or no engine. After the action is implemented on the environment, the environment will return its updated state and a reward value corresponding to the action taken at the state that was acted upon. These outputs will be used to update the networks. This loop will be repeated until the environment ends, either through the lunar lander crashing or running out of time steps; this constitutes one episode in training. The goal of the agent is to maximize its reward by learning what actions output the highest reward in a given state. The lunar lander environment is considered solved, once the agent can repeatedly land the spacecraft with a total episode reward of 200^{+}.

1.3. Metrics

The main performance metric will be an episode's overall reward. The episode's overall reward is obtained by summing the rewards obtained at every time step in that episode. As it can be seen in OpenAI's environment description, the reward for landing on the moon landing pad with zero speed is in between +100-140 reward points. If the lander gets further from the landing pad after an action, the reward will be negative; negative reward can also be called a penalty. If the lander reaches the moon's surface, it will receive a penalty of -100 for touching down at a high speed, considered a crash, or a reward of +100 if it landed softly. If a leg of the lunar lander touches the moon's surface, there will be a reward of +10. Finally, a penalty of -0.3 will be given for every time step that the agent turns on the main engine or side engines; if both engines are turned on, the penalty is -0.6. A secondary performance metric will be how many episodes it takes the agent to achieve an average episode reward of 200 points over the last 100 episodes; the less episodes it takes, the faster the agent learns.

2. Analysis

2.1. Data Exploration

The lunar lander environment takes in two continuous actions; one action for the left and right engines and one action for the main engine. The action given to the left and right engines ranges from -1.0 to 1.0; a value less than -0.5 will power the left engine, and a value greater than 0.5 will power the right engine. The left and right engine sharing a single action ensures the left and right

engines' throttle do not cancel each other out, since they act as the stabilizers of the lunar lander. The main engine also takes action values between -1.0 and 1.0; a value from -1.0 to 0.0 keeps the main engine off, and a value greater than 0.0 throttles the main engine which can only work when the throttle is above 0.5.

```
pos = self.lander.position
vel = self.lander.linearVelocity
state = [
    (pos.x - VIEWPORT_W/SCALE/2) / (VIEWPORT_W/SCALE/2),
    (pos.y - (self.helipad_y+LEG_DOWN/SCALE)) / (VIEWPORT_H/SCALE/2),
    vel.x*(VIEWPORT_W/SCALE/2)/FPS,
    vel.y*(VIEWPORT_H/SCALE/2)/FPS,
    self.lander.angle,
    20.0*self.lander.angularVelocity/FPS,
    1.0 if self.legs[0].ground_contact else 0.0,
    1.0 if self.legs[1].ground_contact else 0.0
    ]
assert len(state)==8
```

Figure 2: OpenAI's Lunar Lander Environment GitHub

An excerpt of OpenAI's Lunar Lander Environment source code can be seen above. The code shows the values that make up the state of the lunar lander after an action has been taken. The state includes the x-axis and y-axis position of the lunar lander. It is critical for the reinforcement learning to know its position, because it is the only way for it to know what position is associated with the high reward if it ever lands on the moon; since the agent's learning is done through independent batch updates, there is no other way for the agent to deduce its position from the velocities or any of the other states given. Velocities along the x-axis and y-axis are also given; x-y velocities are needed by the agent to get a sense of what direction it is going with respect to its current position and adjust engine throttles. For example, if the lunar lander's y-axis position is low, and the its y-velocity is very negative, it will know it has to turn on its main engine in order to prevent a penalty of -100 for crash landing. The lander angle and its velocity are crucial for the flying stability of the lunar lander, because it advices when to turn on a side engine and turn off the other one; for example, if the lunar lander's angle is getting high, but the angle velocity is correcting the dangerous high angle; the learning agent will know to keep throttling the side engine its throttling, since its correcting the current large angle. Finally, the last two states that play a role in the reward received by the agent is whether or not the legs of the lunar lander have touched the moon's surface, because the agent will immediately associate a lunar lander's leg touching the moon with an increase in reward; it will then seek landing on the moon.

2.2. Exploratory Visualization

In order to better understand the agent's need for relevant lunar lander state, such as positions, velocities, and angles, among others, a plot of the information outputted by the environment at every time step is needed. Below, lunar lander environment state features of the final, test episode are plotted and analyzed.

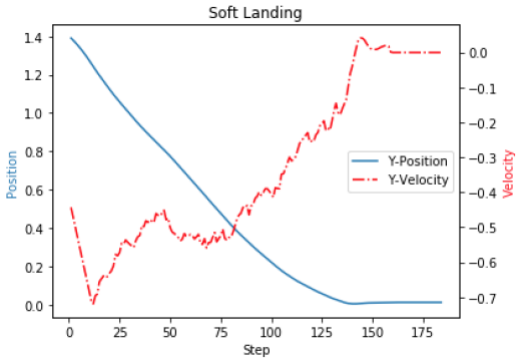

Figure 3: Y-Position & Y-Velocity Relationship

The plot above illustrates the relationship between the lunar lander's y-axis position and velocity at every time step of the testing episode. After being trained for 1,000 episodes, the agent learned that landing at a high velocity would get penalized by -100 reward points, and, as it can be seen in **Error! Reference source not found.**, the closer the lunar lander got to the ground, or a y-axis position of 0.0, the agent made sure to reduce its falling velocity by turning on its main engine. The learning agent also learned the connection between the lunar lander's angle and the side engines which can be seen below.

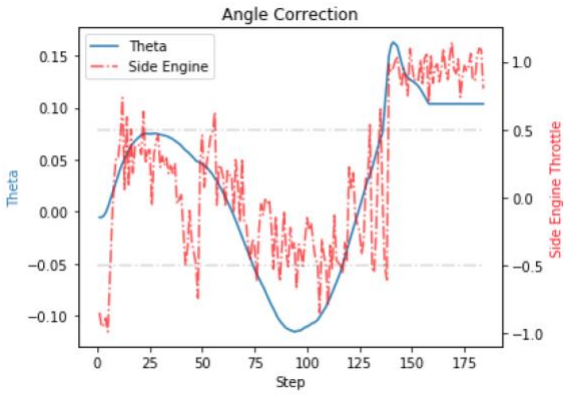

Figure 4: Theta & Side Engine Throttle Relationship

The gray lines represent the values at which the side engines begin working in either direction; if the side engine's throttle value is less than -0.5, the left engine is throttled with a continuous value which can reach a max throttle value of -1, and if the side engine's throttle value is greater than 0.5, the left engine is throttled with a continuous value which can reach a max throttle value of 1. As it can be seen in Figure 4, at a time step of 0, the left engine throttled a max value of -1 which caused the lunar lander to tilt the opposite way and reached a value of 0.10; however, the agent has learned that a high tilt value needs to be urgently stopped as it can cause the lander to spin out. Therefore, the left engine immediately got turned off and the right engine's throttle was activated enough at step 5 to counteract the step 0 left engine's max throttle. At around step 140, the lander saw a large increase, again, in the theta angle, and it was able to reach an angle of 0.15^{+}; therefore, the agent throttled the right engine to the max value of 1 for the remaining steps starting around step 150. From this plot, it can be seen how the agent does learn the relationship between the lunar lander's angle and various action values; the agent chooses actions that maximize its reward, and the agent flipping over, losing control and crashing will bring it a high penalty of -100. The agent effectively learned the relationship between the lunar lander's angle and its side engines' throttle speeds.

2.3. Algorithms and Techniques

The foundation of the learning agent will be based on Deepmind's academic paper, *"Continuous Control With Deep Reinforcement Learning"*. Deepmind presented a reinforcement learning algorithm known as a Deep Deterministic Policy Gradient, or DDPG, algorithm. The DDPG agent is composed of two actor networks and two critic networks; both the actor and critic networks have an identical network known as the target network; the target networks are initialized to have the same weights as the local networks at the beginning of training. The goal of the target networks is to reduce the high variance seen during training which is caused by the constant updating of the local networks' weights at every time step where they might be changed too drastically from one update to the other; without the target networks, actor-critic methods often diverge during training. Target networks' weight are updated at every time step, but they are updated by discounting the updated weights with a small value, or tau, but keeping mostly the previous weights they had. In actor-critic methods, the actors predict what the best action is given the current environment state, and after, the critic maps that action-state pair to the reward received after taking the actor's recommended action on the state.

The DDPG's critic network weights are updated by minimizing the mean squared error between the local critic's predicted value, or Q value, and the target Q value; the target Q value is calculated using the Bellman equation which takes the received reward summed with the gamma-discounted Q value predicted by the target critic. The target network uses the predicted next action of the target actor network which is fed the next state of the experience being trained on. After the critic is updated, the local critic predicted Q values' gradients w.r.t. the actions are taken and passed back to the local actor. The actor uses these gradients and its own actions to get the final weight gradient it will use to update its weights; the actor attempts to maximize its returns by performing gradient descent on the weight gradient.

In order for the agent to learn from different times throughout an episode's states, DDPG uses a replay buffer that stores the agent's previous experiences and is sampled in mini-batches at every learning step; this allows for each sample to be independent of one another which helps the agent generalize learning across a wider range of episodes, and remember previous experiences that might hold important information, such as an experience that received a high reward. Also, to avoid a poor policy

due to overfitting of a local optimum and increase generalization of similar state-action pairs, DDPG adds noise through the Ornstein-Uhlenbeck process which generates time and action correlated noise that reverts to an average, mu, value.

Even though the DDPG algorithm presented by Deepmind posed a brilliant solution to continuous action spaces, it is greatly hindered by a constant Q value overestimation by the critics as is unavoidable when using a Bellman equation which accumulates time differential errors of the past Q value estimates. This leads to a high level of variance in rewards received by the DDPG agent as the policy networks are updated with biased critic estimates of the Q values, since the actor performs gradient descent using the critic's Q gradient, and further propagates the TD error back to the critic network's bias when the actor's action and reward is seen in a critic network update. Therefore, for my final algorithm I will equip the DDPG algorithm with a prioritized experience replay buffer and other modifications found in the TD3 algorithm.

The TD3, or Time Delayed 3, algorithm was presented in the academic paper *"Addressing Function Approximation Error in Actor-Critic Methods"*. The three main modifications to the DDPG algorithm presented by the TD3 algorithm is the use of a third local critic and target critic network, a step delayed update of the actor networks and added noise, not only to the local actor's actions, but also to the target policy's predicted next actions.

Adding a third critic to the algorithm reduces the Q value overestimation of the original critic, by using the minimum predicted next Q value when setting the Q target for the local critic networks; choosing the minimum Q value of the two target networks brings the DDPG's problem of overestimating Q values to a minimum. This, in turn, reduces the reward variance created by an actor following an incorrectly informed policy, since the effect of the critic's Q value overestimation bias on the actor's policy gradient will be reduced. Second, the TD3 paper recommends delaying the actor's update by only updating the actor if the episode's step is divisible by a value; for example, for an environment with a maximum of 2,000 time steps, updating the actor only if the step is divisible by 2 would update the actor at every other time step. This reduces the variance in the actor networks' predicted actions and allows the critic networks to settle and better estimate the Q values for a given set of state-action pairs; therefore, when the actor networks finally see an update, the update will include a better representation of an action's real value given the environment's state. In this implementation of the DDPG algorithm, the time delayed updates will not be included. The last modification the TD3 algorithm applies to the DDPG algorithm is the addition of Gaussian noise to the target actor's predicted action used to predict the target critic's next Q values. Adding a small amount of normally distributed noise to the predicted target actor's actions increases the generalization of action values with similar state-action pairs and, in turn, smooths the policy gradient used for the actor's gradient descent; smoothing of the policy gradients limits the critics' accumulation of biased error, since the policy will not follow as strictly the biased gradient, that favors certain actions with high TD error accumulation. Even though adding more noise to the already variant actions might seem counterintuitive, but the fact that the noise is normally distributed, in turn, normalizes the variance of the actions; variance in actions is inevitable, because the actor networks are constantly being updated to different policies. In order to have a single method of noise production, for both the local and target actors, we will replace DDPG's Ornstein-Uhlenbeck noise with simple Gaussian noise.

Finally, in order to make the sampling of experiences more efficient, a prioritized experienced replay buffer, or PER, will be used. Much of the knowledge of how a PER buffer works was gained by reading the paper, *"Prioritized Experience Replay"*. The PER will assign a priority level equal to the absolute TD error of an experience. In order to get the probability of an experience being samples, the priority of the experience will be brought to a power, a, between 0 and 1 and dividing by the sum of all priorities powered to a; therefore, the bigger the priority of an experience, the higher the probability of the experience being sampled. The closer the a is to 0, the more the replay buffer will resemble a uniform distribution, since all values priority values will have a value close to 1, and the closer a gets to 1,

the greedier PER samples. Below is the equation used to assign the probability value of an experience. *P(i)* is the probability of experience *i* getting sampled, p_i is the experience's priority and p_k is the priority of an experience in a *k* size buffer.

$$P(i) = \left(\frac{p_i^a}{\sum_k p_k^a}\right)$$

Figure 5: Probability given Priorities

Assigning a higher priority to experiences with a high Q error forces the critic networks to learn from those experiences until their Q value prediction better represents the real Q value; however, greedy sampling of experiences with a high TD error will never allow the agent to converge on an optimal policy. Once the agent's learning begins to stabilize due to an accurate estimation of Q values, PER will begin ignoring the high reward experiences with accurately predicted Q values, because the TD error will no longer be big enough for the experience to the high level of priority it had earlier in training; the PER buffer will now select actions with the highest TD error and begin updating its weights based on those experiences, until the previous, high reward experiences return higher TD errors. If left unchecked, this cycle will repeat indefinitely and cause infinite variance in the agent's learning. Therefore, importance sampling weights will be implemented; importance sample weights are inversely related to the batch size, probability of an experience, and the max weight that could be sampled from the buffer. The batch size and probability of the experience is brought to a power value, *b*, in between 0 and 1 and is multiplied by the max weight corresponding to the experience with the smallest probability inside the buffer and the inverse of the product equals the experience's importance sample weight; therefore, the higher the probability of an experience is, the smaller the experience's weight is when training the network. The importance sample weight equation can be seen below; *N* is the batch size, P_i is the experience's probability, and W_{max} is the max weight corresponding to the experience with the lowest probability in the buffer.

$$I.S.Weight = \frac{1}{N^b \cdot P_i^b \cdot W_{max}}$$

Figure 6: Importance Sample Weight Equation

The closer *b* is to 0, the bigger the importance sample weight is equal to the smallest weight in the batch, and all experiences in the mini batch are treated equally by the critic; however, the closer the value is to 1, the more the experience is penalized for having a high sampling probability. During training, *b* will start at a value closer to 0.5, and will be linearly increased to 1 as training continues. Therefore, the closer the agent is to ending a training the smaller the weights will be; the sampled experiences won't have as big an impact as they did during the early stages of training.

2.4. Benchmark

There is an OpenAI leaderboard GitHub repository that showcases implementations that have solved the Lunar Lander environment (getting an average reward of 200^+ on the last 100 episodes). First place solved the environment through the use of a PPO algorithm in 100 episodes, however second place solved the environment in 1,500 episodes using the TD3 algorithm. Third and fourth place solved the environment in 5,000 and 5,300 episodes, respectively. Using the DDPG algorithm, with the addition of a third critic and added noise to the target actor's actions and PER buffer, the algorithm will be considered successful if it can solve OpenAI Gym's "LunarLanderContinuous-v2" environment in less than the 1,500 episodes it took the TD3 algorithm; the DDPG algorithm I will use implements parts of the TD3 algorithm, and therefore, the original TD3's performance can serve as a good benchmark model. The only differences between the Time Delayed 3 algorithm and the algorithm I will be implementing is the absence of a time step delay and the implementation of a PER buffer. Below is a screenshot of the Continuous Lunar Lander's leaderboard.

User	Episodes before solve	Write-up	Video
BS Haney	100	Write-up	YouTube
Nikhil Barhate	1500	Write-up	GIF
Tom	5000	Write-up	YouTube
Sigve Rokenes	5300	Write-up	GIF

Figure 7: OpenAI's "LunarLanderContinuous-v2" Leaderboard

The main metric that will be used to compare the performance of the algorithm implemented in this

paper and the TD3 algorithm in second place is the number of episodes it took to solve the environment, because that is what an algorithms' leaderboard rank is based on in the leaderboard.

3. Methodology

3.1. Data Preprocessing

No preprocessing is needed for OpenAI's Lunar Lander Continuous environment. Usually, if state spaces or action spaces in a given environment range across a large number of values, standardization or normalization of the states and/or action values is desirable to improve the network's learning by limiting spikes in the inputs' values that could cause a drastic change in the networks' weights. However, the lunar lander environment has provided us with a decently scaled environment states where the starting of the quadcopter is 1.4 and is bounded below by 0, and the x-axis range -1 to 1. The height is not bounded above, but the agent is penalized for every time step it activates its engines; therefore, it will not desire continuously turning on the main engine to reach a very high y-position. The reward starts at zero but has a shaping value that returns a value based on the state of the lunar lander, and first sight, it appears to be a big negative penalty; however, as it can be seen in the following lines of code, the shaping function is not the reward.

```
reward = 0
shaping = \
    - 100*np.sqrt(state[0]*state[0] + state[1]*state[1])
    - 100*np.sqrt(state[2]*state[2] + state[3]*state[3])
    - 100*abs(state[4]) + 10*state[6] + 10*state[7]   #
                                                      #
if self.prev_shaping is not None:
    reward = shaping - self.prev_shaping
self.prev_shaping = shaping
```

Figure 8: OpenAI's Lunar Lander Environment GitHub

Instead, the shaping is subtracted by its previous value which will be very similar and will lead to a small scalar value that will become the reward; values will be similar due the nature of time steps and the inability of the lunar lander to teleport to a drastically different state in between steps. Even when the shaping current value is very negative, as long as it is greater than the previous shaping value, the reward will be positive, because subtracting a negative by a bigger negative will result in a positive value due to the change in sign when subtracting by a negative number. For most of an episode, the lunar lander's reward will be a small value which does not need to be standardized or normalized. The biggest reward change happens when the episode is over, and the lunar lander can output +100 reward if it landed softly or -100 if it crashed or went out of bounds. Those values can be left as is, because those rewards/penalties are only handed out once in an episode but are one of the greatest differentiators when deciding if the landing was a success or a failure. Also, the fact that they happen only once per episode means they would be sampled less in a random buffer selection process or given a very low importance sample weight in a PER buffer. All in all, the outputs given by the lunar lander environment are very nicely standardized and seem to have already been preprocessed by OpenAI when they created it; therefore, there is no need to preprocess them further.

3.2. Implementation

The parameters used in the training of the algorithm are as follows; a detailed explanation as to how they were arrived upon will be given in the next section of the paper, **3.3. Refinement**. However, this section will give a quick overview of how the agent was implemented, in case the reader desires to recreate the algorithm. The first layer of the actor networks is an input layer that accepts an array with eight columns, one column for each part of the state returned by the environment. Two hidden layers were used for the actor networks, and each hidden layer had 400 and 300 nodes, respectively, and used the 'relu' activation function. The output layer returned two action values ranging from -1 to 1, one for the side engine throttling and the other for the main engine throttling; in order to get those output ranges, the output layer applied the 'tanh' activation to the action values. The actor networks used the Adam optimizer for gradient descent and was set to the default learning rate of 0.001. In order to train the actor network, a custom function using Keras backend's function and the direct use of the Tensorflow library was needed; below is an image of the actor's custom training function and Tensorflow's implementation.

```
# actor training function
action_grads = layers.Input(shape=(self.action_size,))
actor_weights = self.model.trainable_weights
actor_grads = tf.gradients(actions, actor_weights, -action_grads)
grads = zip(actor_grads, actor_weights)
updates_op = tf.train.AdamOptimizer(self.lr).apply_gradients(grads)

self.train_fn = K.function(
    inputs=[self.model.input, action_grads, K.learning_phase()],
    outputs=[],
    updates=[updates_op])
```

Figure 9: Actor's Custom Training Function

The "train_fn" function can be seen at the bottom of the code, and it passes the states of the mini batch sampled to train the actor to the actor network and the actor network calculates the predicted local actions that will be used to compute the policy gradient; the action gradients are calculated by deriving the critic's Q values w.r.t. the local actor predicted actions and are passed as the second input. The K.learning.phase() input tells the actor model whether it is being trained or not. If the value passed to Keras learning phase is 1, the actor knows it is being backpropagated and its weights will be updated; on the other hand, if its weights are not going to be updated, but an output is needed, a value of 0 will be passed. Since the purpose of the custom function is to train the actor network, the learning phase function will be passed a value of 1. The actor does not need to output anything, but an update operation will occur; the update operation requires the policy gradients computed to the applied to the actor network's weights, but Keras's Adam optimizer does not provide a function that will apply the policy gradient computed directly to the model's weights. Therefore, the update operation will be Tensorflow.train.AdamOptimizer(lr).apply_gradients(policy grads) which returns the update operation defined in the custom training function. In order to compute the policy gradients, Tensorflow will also be used, because Keras's backend gradient function only accepts the parameters and the loss to derive. However, the action gradients are provided by the critic network, and Tensorflow's gradient function does accept already computed gradients as one of its parameters to be applied in conjunction with the parameters. The loss is the local actor's actions in conjunction with the negative action gradient, for gradient descent, and the parameters that derive the actor's actions and their action gradient are the trainable actor policy weights.

Once the policy gradients are computed and the actor's weights are updated, the training process of the actor is done. However, since the action gradients cannot be calculated by the actor's model, another custom Keras function will be needed; the code for getting the action gradient can be seen below. Finally

```
# Compute action gradients (derivative of Q values w.r.t. to actions)
gradients = K.gradients(Q_values, actions)
self.get_gradients = K.function(
    inputs=[*self.model.input, K.learning_phase()],
    outputs=gradients)
```

Figure 10: Custom Action Gradient Function

The custom function "get_gradients" takes the mini batch states and the local actor's predicted actions, and inputs them into the critic's network who will output the predicted Q values given the states and predicted actions. In this function, a value of 0 is passed to the K.learning.phase() function, because the critic network is just being asked to return its current action gradients which are derived from the predicted Q values w.r.t. to the actor's predicted actions. For the calculation of the action values, Keras's gradient function can be used, because action gradients only need to be computed by deriving the Q values, or loss, w.r.t. the actor's predicted actions, the parameter. Once the action gradients have been computed the "get_gradients" outputs them back to the agent.

As to the critic networks' architecture, a similar architecture to the actor networks' architecture is used. However, the critic's inputs include the eight state values along with the two actor's actions; I combined both inputs by using Keras's concatenate layer. The critic networks also used two hidden layers, with 400 and 300 nodes, respectively, and both hidden layers use the 'relu' activation function. The final layer is an output layer that condenses its inputs to a single Q value which it tries to accurately predict by minimizing the mean squared error loss between the Q value predicted by the local critic and the discounted, minimum Q value predicted by the critic target networks plus the experience's rewards; optimization is done by Keras's Adam optimizer with the default learning rate of 0.001.

The PER buffer can hold up to 10,000 samples and uses a binary heap to store an experience's priority value. When training, a minibatch of 32 experiences is sampled. When calculating the probability of the experiences, the *a* power was set to 0.6; the *b* power used to calculate the importance sample weight was set to 0.4 and linearly increased until it reached a

value of 1 at the end of training. For example, when training for 500 episodes with 2,000-time steps each, the in order to linearly increase 0.4 to 1, 0.6 is divided by the product of 500 and 2,000; this outputs an incremental value of 0.0000006 needs to be applied to *b* at every time step. Once the buffer reaches 10,000 experiences, the oldest experience is replaced by any new experience. Before a learning step can happen, the PER needs to be initialized by filling the buffer with experiences created by the actor network initial weights; this avoids PER to sample empty experiences from the buffer during learning. Example code of how this was implemented can be seen below.

```
time_steps = 2000 # set max steps in an episode
steps = np.arange(1, time_steps+1)

while True:
    state = agent.reset_episode() # start a new episode & init. state
    for step in steps:
        action = agent.act(state) # select actor action given state
        next_state, reward, done = LL.step(action)   # take action on environment
        state = next_state                  # update environment's state in loop
        agent.step(action, reward, next_state, done) # add experience to buffer
        num_exp = agent.num_exp
        if done or step==time_steps or num_exp==agent.buffer:
            break
    if agent.num_exp==agent.buffer: # break while loop, since PER buffer is full
        break
print('PER Buffer Initialization Completed')
```

```
PER Buffer Initialization Completed
```

Figure 11: PER Buffer Initialization

After the prioritized replay buffer has been initialized, the learning process can begin. The first step is to reset the environment and get the initial state. Second, an action will be selected by the actor network, Gaussian noise with a mean of 0 and a scale, or standard deviation, of 0.1 will be added and this action will be implemented in the lunar lander's environment The environment will return the next state of the learning process, along with the reward associated with that action a Boolean of 0, or False, if the episode has not been terminated by the environment and a Boolean of 1, or True, if it has. The reason why an environment would end the episode could be accounted to the lunar lander going out of the x-axis bounds, the lunar crashing onto the ground or the lunar lander simply landing. The environment's outputs will be added to the replay buffer and a minibatch of experiences will be sampled from the PER buffer. Each of the samples in the mini batch will contain the states, actions, reward, next states and Booleans of the sampled experiences. Using the next states, the target actor will predict the next actions, and Gaussian noise will be added to those actions; this time a scale of 0.2 will be used with a mean of 0. The target critics will each predict the batch's next action values, or next Q values, given the next states and the next actions with the added noise. The minimum of the next Q values predicted by the target critic networks will move on to the next step. In order to get the Q target values that will be used to train the local critic networks, the minimum Q value will be multiplied by a gamma discount factor of 0.99 and summed with the reward; also, if the Boolean value is 1, which means there is no next state, the Q target will just be the rewards obtained before the episode was ended by the environment, and the calculated next Q value will be discarded. In order to get the absolute time differential error that will be used to update the mini batch experiences' priorities, the local critic networks will each predict the Q values of the states and actions before they get updated by the calculated Q target value. Once the local critics make predict the Q value, each Q value will be subtracted from the Q target value, and the absolute value will be taken; only the scalar absolute value that indicates how big the Q error was is important when updating the PER, and since the PER only takes one priority value per experience, the average of the two absolute Q errors will be taken. Once the mean, absolute Q error is passed to the PER buffer, the local critic model can be updated.

This will be done by using Keras model's train_on_batch method; the state and actions will be the input data and the Q targets will serve as the labeled, or correct, data, and the loss seen by the local critic network will be adjusted using the importance sample weights of the corresponding experiences. Once the local critic networks have been updated, the local actor network can be updated. Its training will begin by feeding the batch experiences' states to the local actor network which will predict the actions it would select given the states. These predicted actions along with the states passed to the local actor network to make such predictions will be passed to the Keras backend custom function, "get_gradients"; this function will return the Q value gradient w.r.t. the actions predicted by the local actor network. The training of the actor network is initiated by the Keras custom function, "train_fn"; the states the local actor used to make the action prediction along with the critic's returned action gradients will be passed and the actor's weights are going to be updated following policy gradient descent. Once both the local critic networks and the local actor network, the target

networks can be updated; I will not implement the time delay by the TD3 algorithm. The target networks' will be soft updated by multiplying their weights by (1 – tau) where tau is a value of 0.005, and the local networks' weights used to update the target networks will be multiplied by tau. Therefore, the target networks will keep 0.995 of its weights and will be updated by 0.005 of the local networks' weights. This will conclude a learning step of the agent, and the loop will begin again, by setting the state to the next state returned by the environment in the beginning of the learning step.

3.3. Refinement

Two hidden layers were used for the actor networks, even though one hidden layer is sufficient for most problems, because the biggest challenge of the lunar lander is the continuous action space; two layers ensure the network is able to map complex reward functions where the Q value cannot be easily mapped with a single hidden layer. The layers' width of 400 and 300 nodes allow for each layer to account for the infinite continuous state and action space values it can produce, because each node can learn a certain characteristic of the state, whether it be focusing on the x-axis positional value or the lunar lander's angle.

The 'relu' layer is generally regarded to be able to learn complex, nonlinear relationships between input values and the output values, especially when they are combined consecutive layers also using the 'relu' activation function. Using 'tanh' or 'sigmoid' activation functions limits the values to the range between -1 and 1 and are really bad at estimating non-linear relationships as the x-axis inputs they take are linearly related to the y-axis outputs. A graph of the 'tanh' function will be shown below to better illustrate this linear relationship.

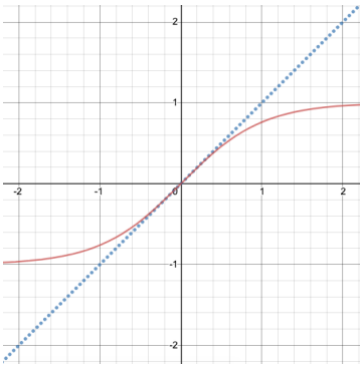

Figure 12: tanh(x) & (x) functions

The solid red line is the plot of $y = \tanh(x)$, and the dotted blue line plots the linear $y = x$ function. It can be seen that the greatest portion of useful tanh(x) values which have a range of -1 to 1 are linearly correlated with the input, x; this leaves a very low amount of room for the 'tanh' function to learn non-linear relationships between its inputs and its outputs. Another downside of hidden layers using activation functions that can limit their output values to a small range like -1 to 1 is their vulnerability to oversaturation on either extreme of the function, because a very large input value of 100 will be treated similar to a value of 10; this will lead to vanishing gradients which will cause divergence to occur. However, having them as output layer activation functions is useful when the output action lies between the -1 and 1 range; or a 'sigmoid' activation function for when the output cannot be negative but is still contained to a range of 0 to 1. Therefore, since the action values the lunar lander environment accepts are between -1 and 1, the 'tanh' activation function is used for my output layer.

Both the actor and critic networks used the Adam optimizer, which has become the default optimizer when training deep neural networks due to its ability to deal with sparse gradients where there is a low amount of data to create the gradient from; since we are sampling from mini-batches with 32 samples, we definitely need an optimizer that is able to handle sparse parameters. Because Adam combines the momentum of the previous gradients with the current gradients, it is able to take a step in a direction that might be better but limits the step from going too far in that direction by applying the averaging that new gradient direction with the previous gradient direction. This sort of update is

useful in the lunar lander environment, since an update using only the current batch's gradient might make the agent overcorrect to the other end; for example, let's say the lunar lander began having a high angle to the left due to the previous batch gradient directing the agent to throttle the right engine. Adam will take the momentum of the previous batch gradient that directed the agent throttle the right engine and average it with the current batch that told the agent to throttle the left engine to balance the lander's angle, and the outcome is just the right amount of left engine throttle to balance both gradients and, in turn, the lander's angle; if we only used an optimizer that used the current batch's gradient, there is a possibility for the agent to overfit to the current batch and throttle the left engine too much, and the agent would take longer to understand that the best throttle value lies between the throttling extremes of either side engine. Adam's default learning rate of 0.001 was left as is, since the last 100 average episodes continuously increased. The critic neural networks hidden layers are identical to the actor networks' hidden layers; however, for the input layer, the critic intakes the lunar lander environment's states and actor's actions to predict the Q values. The output layer is of size 1 which is the Q value, and it uses the default linear activation function in Keras's Dense layers; it needs a linear activation function in order to capture a larger number of Q values, which correspond to the reward given by the environment, because the environment's rewards can range from -100 values to +100 rewards.

Since the PER forces the critic networks to learn from high losses rather than trying to avoid them by creating a gradient that wants to push the actor to actions that are well known but are highly possible to be poor local optima, the PER does not have to store as many experiences, because it constantly samples experiences with a high TD error rather than randomly getting them through the use of a uniform sampler; therefore, a smaller than normal buffer of 10,000 experiences is used. When training, a minibatch of 32 experiences is sampled, because the bigger the batch samples, the more chance of overfitting to experiences that are currently causing the high TD error. The PER *a* power value was initialized to 0.6, because an *a* value much bigger than that ignores experiences with a low TD error, even if they might yield high rewards. On the other hand, since the *b* value will be increased as training goes on, it is better to start at a value closer to 0 than 1, because it dramatically speeds up the early training stages where the agent is first learning to make connections about its environment but begins slowing down the variance in rewards as training reaches late stages where it converges.

Gaussian noise added to the local actor's actions use a scale of 0.1, because they are the actions that actually affect the environment. However, when adding noise to the target actor's actions, a scale of 0.2 is used, because the target actor's actions are not affecting the environment directly and have more room for a higher level of noise; more noise added to the target actor's actions also discounts any bias that the critic networks might have, by going off policy where the policy was created using the critic's gradient. However, so target actions are not drastically changed by the higher noise level, the noise will be clipped to -0.5 and 0.5; a value of 0.5 prevents changing an action by more than 50% of the total action range. When discounting the future Q targets a discount gamma value of 0.99 allows the agent to remember old experiences as much as possible without accumulating to a point where any new experience barely makes a difference in the policy; for example, if a value of 1.0 is used, the agent will never forget old experiences which will prevent the agent from learning new behaviors based on newer experiences. Soft updating the target networks with a value of 0.005 allows the target networks to adjust fast enough to keep up with the local networks learning but is not too big to the point where they resemble the local networks, since that would greatly increase the variance of the learning process.

4. Results

4.1. Model Evaluation and Validation

The final model results were obtained using the algorithm described in **3.2. Implementation**, and it exceeded expectations. When defining the benchmark model to improve upon, a TD3 algorithm that took 1,500 episodes to solve the

environment was chosen. It was proposed that the implementation of a prioritized experience replay buffer, or PER, and no time step delay in between actor network updates would improve upon the TD3 algorithm; however, it could not be predicted by how much, and even a small reduction in episodes until the environment was done, would be considered a success. After running the training with different random seeds, it can be noted that the DDPG algorithm, with a third critic, target action noise and PER is able to constantly solve the environment around the 500 episode mark. The first time the algorithm was ran, it was ran for 1,000 episodes, and was able to solve the Continuous Lunar Lander environment in about 500 episodes. Therefore, for the next runs, the training episodes were reduced to 500 and training was repeatedly producing similar results. Since, every training session used random seeds for the network weights and Gaussian noise, it can be noted that the algorithm responds well to any random initialization. However, in order to produce a result that could be replicated as close as possible, NumPy and Tensorflow seed values of 14 were used. The training session after seeding the networks to 14 can be seen below.

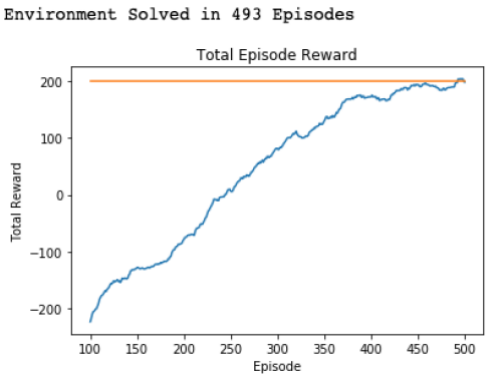

Figure 13: Moving Average of Last 100 Episodes

The total episode rewards' shows a steady increase in the smoothed reward plot where the last 100-episode scores were used to get every data point. The first time the algorithm reached an average reward of 200 over the last 100 episodes occurred at episode 493. The results of this algorithm can be trusted, and if desired, can be reproduced by downloading the contents found at my GitHub.

4.2. Justification

The results of the modified DDPG algorithm found in this paper surpass the results obtained by the benchmark TD3 algorithm which is also a variation of the DDPG algorithm; by adding a PER buffer and not including the time delayed step in the benchmark model, the model in this paper was able to reduce the number of episodes needed to solve the environment by a factor of 3! The sections above cover every detail as to how the algorithm was developed and where the inspiration for modifications came from, and the code used to implement and train the algorithm is available in the GitHub linked in the **4.1. Model Evaluation and Validation** section. The final solution can be interpreted to have achieved the goal of solving OpenAI's "LunarLanderContinuous-v2" environment in less episodes than the benchmark model.

5. Conclusion

5.1. Free-Form Visualization

An interesting plot not shown in this report is the plot of an episode which is considered to have solved the lunar lander environment by getting a reward of 200^{+} and can be seen below.

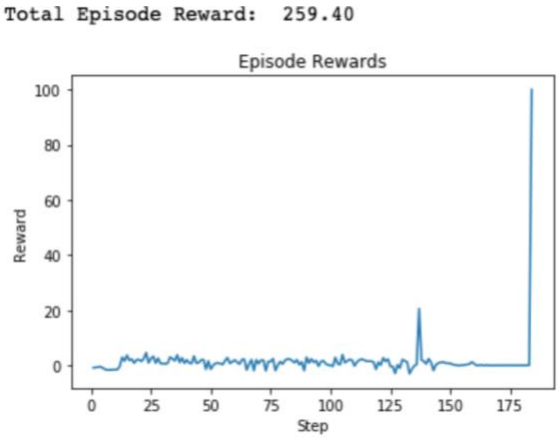

Figure 14: Successful Episode Rewards

As it was stated in section **3.1. Data Preprocessing**, the rewards of the episode did not need to be preprocessed due to OpenAI keeping the values for

most of the time steps at a standardized level. There are two peaks that can be seen in the reward plot. The first one happens around step 137 where the agent received a reward of +20; that peak in reward correlates to both of the legs touching down on the moon's surface. The +20-reward peak shows a good example of a time step with a sudden high change in reward which could've led to a high TD error; that type of experience would be given a high priority by the PER buffer and sampled frequently due to its high sampling probability. However, if this peak were to have happened in a late training stage where the *b* power of the importance sample weight would've gotten really close to 1, the experience would be needed to be sampled many more times than a similar experience in an early training stage before a change in the policy was seen, because the importance sample weight passed to the critic network's update gets smaller as *b* is increased. The only other spike occurred at the end of the episode where a reward of +100 was received by the agent for landing softly; the soft landing can be seen in Figure 3, where the lunar lander slows down the closer it gets to the moon's surface. Peaks like the ones seen in this episode are what makes an algorithm equipped with PER learn faster than an algorithm without PER.

5.3. Reflection

Before the idea of teaching a lunar lander to land on the moon, I was working on teaching a quadcopter to start from the ground, reach a target height and hover at that height until the time steps ended. Obsession to make the quadcopter succeed made me delve into machine learning academic papers, looking for a solution. Therefore, once I concluded that project and began working on the lunar lander environment, I understood the fundamentals of how actor-critic methods worked; especially the DDPG algorithm. I began the lunar lander environment wondering if the DDPG algorithm would work on the environment, and after reading about ways to improve the DDPG algorithm, I learned about the TD3 algorithm. The reason I was looking for ways of improving the DDPG algorithm is that it did not work as well as I had hoped, even when the action space included only a single rotor speed value. Therefore, I concluded that I would try to improve upon the DDPG algorithm I used to train the quadcopter and apply the improved DDPG algorithm to the lunar lander environment.

I began by writing the code for a basic actor and critic network. The code would be split into a separate python file for each class that contained a crucial component of the modified DDPG algorithm. The python files created were the actor, critic, Gaussian noise, PER buffer, task and agent; the Gaussian noise and the task script files were the easiest to write for the lunar lander environment, since they only needed very few lines of code to get them up and running. The Gaussian noise python file only really needed to be passed an action size in order to set the size parameter of NumPy's random normal. To sample the noise, however, the batch size and scale to set the Gaussian noise to had to passed.

For the task, OpenAI Gym made it really easy to implement its "LunarLanderContinuous-v2" with my agent. The states as well as the rewards were already at a scaled size that did not need any preprocessing. One of the hardest parts of a reinforcement learning project is the creation of the reward function, because some functions that might seem like they incentivize good behavior by the agent, may inadvertently cause undesired behavior. For example, if you only give a reward when the lunar lander touches the target ground position, the lunar lander agent will try to reach that target as fast as possible and crash the lander. Despite that not being what is desired from the agent, if it is never rewarded or penalized for its velocity, it will never know that landing at a high velocity is undesirable. On top of the reward function, if the agent is not given relevant state information from which it can derive the factors affecting its reward, it will not be able to learn what behavior to fix in order to improve its score. For example, if the velocity affects the reward function but it is not given to the agent, it will never know that the velocity is what needs to be adjusted; however, the velocity does not have to explicitly stated. Giving the agent two consecutive positions given an action, it can derive the relationship between its action and the change in position, or velocity. OpenAI's lunar lander environment handled both of those crucial aspects of a successful reinforcement learning algorithm and eliminated the need for preprocessing the environment.

The critic and actor networks were defined next, where both had hidden layers with 'relu' activation functions, because two consecutive hidden layers with 'relu' activation functions are able to model a nonlinear, complex relationships very well. The actor networks needed a 'tanh' output activation function in order to keep the action values within the -1,1 range accepted by the lunar lander's environment. One of the biggest stumbling blocks when creating the actor network's update function was the inability to apply gradients directly to the Adam optimizer. In order to circumvent that, I integrated Tensorflow's Adam optimizer's apply gradients and tf.gradients function into the actors' Keras models for gradient descent; once I figured out what Tensorflow functions were useful, integrating them into Keras was not difficult, because of Keras using Tensorflow as backend.

The PER buffer python file became one of the hardest to code, since it required that I understand how a binary heap worked; understanding the data structure being used in crucial for troubleshooting functions and classes that might be receiving errors. For example, if the PER buffer did not become initialized before the minibatch sampling began, dividing by zero warnings and invalid values would be returned. Initializing the replay buffer was explained in section **3.2. Implementation**.

Finally, the agent python file compiled all the other python files and allowed the configuration of all aspects of the agent to be configured in one script. Once all of those files were created, I created the jupyter notebook that called the agent and task scripts. The training process was done here as well, and I recorded any metrics that I deemed important; for example, episode rewards were stored in arrays, and were later plotted to see how well the learning agent did. The performance of the algorithm surpassed my expectations, and I believe that it could be applied to many other environments, as long as a good reward function is created for the environments. Also, hyper-parameter tuning might also be needed when generalizing this algorithm to other environments, such as the learning rate of the optimizer or tau discount rate of the local networks.

5.4. Improvement

There are definitely ways to improve upon the algorithm used in this project. For example, maybe a time delay combined with a change in learning rate of the actor networks would lead to a better result than staying with the actor's learning rate of 0.001 used in this paper. Also, the Ornstein-Uhlenbeck process could be used to generate the noise added to the target actor's actions. Different optimizers, such as the use of the SGD optimizer with momentum could be a good alternative to the Adam optimizer's use of moving average gradients. Also, there is a version of the Lunar Lander Continuous environment that is supposed to be harder and can be activated by changing the environment's seed; I will test my algorithm on that hardened version of the environment.

The first-place algorithm was able to solve the Lunar Lander Continuous environment in 100 episodes which means that the environment could have been solved in less than 100 episode, but the fact that the OpenAI leaderboards require an average of the last 100 episodes might have made that the minimum number of episodes that the environment could be solved in. That solution of the lunar environment implemented the PPO algorithm which I have not invested time into learning how it works, but it will be the next algorithm I set my sights on understanding. Maybe parts of the PPO algorithm can be used to improve the algorithm used in this project, or they might turn out to not be too different. Using this project as my new benchmark, I definitely know that there is an algorithm that solves the environment in even less episodes, since first place in the "LunarLanderContinuous-v2" environment is proof of it.