Reinforcement Learning for Quadcopter Control

Ali Muhammad Tariq
2310569

A thesis submitted for the degree of

Master of Science in Artificial Intelligence

Supervisor: Dr. Michael Fairbank

School of Computer Science and Electronic Engineering

University of Essex
Date: August 21, 2024

Abstract

In this research, I implemented reinforcement learning-based approaches to various control problems. Policy gradient algorithms such as REINFORCE, PPO, and A2C were used to test their limitations on the readily available OpenAI gym environments, Lunar Lander, and Cart Pole. A physics engine was developed for the quadcopter using the Pygame library. Using the PPO algorithm, the quadcopter was successfully made to hover at 300 altitude. Various hypotheses were tested to make the algorithm converge. Different reward functions have different convergence rates. The choice of the episode length and designing the intelligent reward function played a critical role in convergence. Scaling the reward functions improved the convergence rate. The work has significant applications in the areas of delivery services, search and rescue operations, traffic monitoring, security and surveillance, and Mapping.

Acknowledgements

I want to thank my parents for their support; this would not have been possible without them.

I thank Dr. Michael Fairbank for being my supervisor for this project, and for teaching me the Game Artificial Intelligence course, which introduced me to the concepts of artificial intelligence specifically Neural Networks, Deep learning, Reinforcement learning, and Genetic Algorithms. His guidance on the master's thesis was very helpful for the practical implementation of the Reinforcement learning concept to real-world problems.

Contents

1 Introduction

1.1 Problem statement

Reinforcement learning is the machine learning branch where an agent interacts with the environment receives a reward based on the decision it makes and then learns to maximize the reward through the hit-and-trial method using exploration and exploitation. Reinforcement learning algorithms have difficulty converging to the optimal solution. The policy gradient methods which optimize the reward directly rather than the value function are useful in environments with large state space. Can a Policy gradient method such as PPO be used to successfully control a Quadcopter for hovering?

1.2 Objectives

The primary objective of this thesis is to explore and optimize reinforcement learning (RL) techniques for controlling a quadcopter. Specifically, the research aims to develop effective reward functions that guide the learning agent (the quadcopter) towards achieving stable flight and maintaining a desired altitude. To this end, the thesis focuses on the following goals:

1. **Test and Compare RL Algorithms:** Utilize different RL algorithms, such as REINFORCE, Proximal Policy Optimization (PPO), and Advantage Actor-Critic (A2C), to train the OpenAI GYM environments (Lunar lander, Cart Pole) and compare their performance.

2. **Develop custom Physics Engine:** Design a custom physics environment .

3. **Develop and Implement Reward Functions:** Design and implement various reward functions to evaluate their impact on the quadcopter's learning process and performance.

4. **Analyze Performance and Convergence:** Investigate the performance and convergence of the quadcopter under different reward functions, including how these functions affect its ability to hover at a target height and maintain stability.

5. **Optimize and Identify Best Practices:** Identify the most effective reward functions and strategies for achieving optimal control, contributing to the broader field of RL in robotics.

1.3 Structure of Work

The thesis is structured as follows:

1. **Introduction (Chapter 1):** Provides an overview of the problem statement and sets the context for the research. It outlines the objectives, significance, and scope of the study.

2. **Literature Review (Chapter 2):** Reviews existing literature on reinforcement learning, with a focus on its application in robotics and control systems. It discusses various RL algorithms and their relevance to quadcopter control, highlighting the need for effective reward function design.

3. **Implementation of Reinforcement Learning Environments and Algorithms (Chapter 3):** Details the setup and implementation of RL environments using OpenAI Gym, along with the application of algorithms like PPO, A2C, and REINFORCE. The chapter explains the development of custom environments tailored for quadcopter simulations.

4. **Building the Quadcopter Physics Model (Chapter 4):** Describes the creation of the quadcopter's physics model, including its mathematical formulation and implementation in Python. This chapter also covers the development of a custom Gym environment specifically for simulating quadcopter behavior.

5. **Results (Chapter 5):** Presents the results of the experiments, focusing on the quadcopter's performance under various reward functions. This chapter provides a detailed analysis of how each reward function influenced the learning process, including testing and comparison with standardized reward functions.

6. **Conclusion (Chapter 6):** Summarizes the findings, discusses the implications of the research, and suggests potential future directions for improving RL-based quadcopter control.

7. **Appendix (Chapter 7):** Includes additional data, graphs, and code snippets relevant to the research, providing supplementary information that supports the main text.

2 Literature Review

2.1 Why Reinforcement Learning

Reinforcement Learning made a breakthrough when the Google DeepMind team defeated the Go Champion Lee Sedol. Subsequently, it defeated the strongest chess engine stockfish. It was a marvelous achievement because the style of play was very creative. Lee Sedol (9 Dan) is considered a Go genius who took a great beating on ego when alphaGo defeated Lee Sedol (4-1). On 19 November 2019, Lee announced his retirement from professional play, stating that he could never be the top overall player of Go due to the increasing dominance of AI. Lee referred to them as being an entity that cannot be defeated".Yonhap News agency Interview. After the game, Lee Sedol stated **"I thought AlphaGo was based on probability calculations and it was merely a machine but when I saw this move (move 37 in Game 2) Surely Alpha Go is creative. This move was really creative and beautiful. This move made me think about Go in a new light. What does creativity mean in Go? It was a really meaningful move. Losing to alpha zero (5-0) will really hurt my pride. I also feel so bad for the people who have supported me. What surprised me most was that AlphaGo showed us that moves that humans may have thought are creative, were actually conventional. I think this will bring a new paradigm to Go. I have grown through this experience. I will make something out of it with the lessons I have learned. I feel thankful and feel like I have found the reason I play Go. I realize it was really a good choice learning to play Go. It's been an unforgettable experience."** Lee eventually won game 4 and scored (5-1) in the tournament. Lee Sedol: **"I heard the people were shouting in joy when it was clear that AlphaGo has lost the game. I think its clear why. People felt helpless and fearful. It seems we humans are so weak and fragile, and this victory meant that we can still hold our own. As time goes on it'll probably be very difficult to beat AI but winning this one time it felt that it was enough, one time was enough. I have never been congratulated so much for winning one game. I wouldn't give this for anything in the world"** AlphaGo - award-winning documentary

2.2 What is Intelligence?

"Expressions of intelligence in animal and human behavior are so abundant and varied that there is an ontology of associated abilities to name and study them, such as social intelligence, language, perception, knowledge representation, planning, imagination, memory, and motor control"[14]. Many facets of such intelligence, including language, perception, memory, and motor control, are driven by the unconscious mind [21]. One approach to mimicking human intelligence would be to understand how the unconscious mind works and then translate that understanding into computer language. However, one of the biggest challenges to this solution is that the unconscious mind is a black box, and we have no direct access to its functions.

Most aspects of intelligence are driven by the unconscious mind, such as language, perception, imagination, memory, and motor control [21]. In activities like language and perception, humans do not consciously perform the actions. For example, when speaking, complex sentences and ideas are formed without conscious involvement. The unconscious mind processes an idea and converts it into words and language—similarly, perception occurs unconsciously. The retina of the eye is bombarded with photons, sending over 10 million signals to the brain each second. Scientists have tried to determine how many of these signals can be processed consciously. The most liberal estimate is that people can process 40 pieces of information consciously. To handle such a large amount of information, the unconscious mind steps in. One might then question why we don't simply discover how the unconscious mind works and use that understanding to design a machine that mimics it. However, the biggest impediment is that we don't have access to the unconscious mind, and even if we apply metacognition to understand it, we are unable to accurately decipher its functions. The unconscious mind is a black box. Many psychologists, such as Sigmund Freud and Carl Jung, studied this area but were unable to fully decode this black box. One main hurdle is that as soon as you begin to consciously observe unconscious phenomena, they begin to fail. The whole idea of the unconscious is that it operates without conscious intervention. For example, if you try to observe how the brain forms language while speaking, the process of speaking is disrupted because it has become a conscious activity.

This raises the question: If the unconscious is beyond our understanding, then what is intelligence? After all, intelligence is a conscious activity deliberately performed by humans. The other type of brain function is conscious processing, controlled by the prefrontal cortex and based on deliberate control. For example, the command from the brain to pick up a glass of water is fully conscious and controlled. Conscious processing can also display a wide range of intelligence, such as:

- Understanding the deeper meaning of things and how they work.
- Finding solutions to problems.
- Making (wise) decisions.
- Recognizing flaws and subsequently improving decision-making

If we engage in metacognition to explore what is actually happening in the brain during such conscious and intelligent actions, we might find that the brain performs the following actions:

Processing of the Conscious Brain - Cognitive Reasoning (Prefrontal Cortex)

(a) Exposure to a new task.

(b) Perform an action.

(c) Fail/Succeed.

(d) Analyze the situation, ask the right questions, find answers, and develop solutions (which may be right or wrong).

(e) Perform a new action based on analysis.

(f) Fail/Succeed.

(g) If failed, repeat from action (d). If successful, store the lesson in memory for future decisions and explore new solutions.

2.3 Reinforcement Learning

In reinforcement learning, exploration, exploitation, and the reward function translate human cognitive processes into the language of a computer. The exploration and exploitation phases mimic the hit-and-trial approach (steps b, c, e, f), while the reward function encapsulates the logical reasoning (step d) of the human brain to reach the right conclusions. Actions are then performed, and course corrections are applied to decision-making to achieve optimal control. The human brain applies course correction through logical reasoning, but once an optimal solution is found, it is handed over to the implicit functions to be performed automatically.

The difference between reinforcement learning and human intelligence is that reinforcement learning requires a predefined reward function to optimize. Humans have the ability to identify their own reward functions during step (d). If we could somehow decode the process of human reasoning, we might be able to achieve artificial general intelligence (AGI).

2.4 Policy Gradient Methods

Policy Gradient methods are a class of algorithms in Reinforcement Learning (RL) that directly optimize the policy—the function that maps states to actions—by computing the gradient of a performance measure with respect to the policy parameters. These methods are particularly powerful in environments where the action space is large or continuous, making them well-suited for complex control problems like robotics, game playing, and other decision-making tasks[17].

1. **Policy (π)**
   A policy is a function that defines the behavior of an agent by mapping states (or observations) to a probability distribution over actions. The policy can be stochastic (outputting a probability distribution) or deterministic (outputting a specific action)[17].

2. **Parameterization of Policies**
   Policies are usually parameterized by a set of parameters θ, denoted as $\pi_\theta(a|s)$, where θ could be the weights of a neural network (function approximator). The goal of policy gradient methods is to optimize θ to maximize cumulative reward[12].

3. **Objective Function**
   The objective in policy gradient methods is to maximize the expected cumulative reward:
   $$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t)\right]$$
   where $R(s_t, a_t)$ is the reward at time step t, and γ is the discount factor [19][18].

4. **Gradient Estimation**
   The core idea is to compute the gradient of the objective function with respect to the policy parameters, $\nabla_\theta J(\theta)$, and then update the policy parameters in the direction of this gradient. The policy gradient theorem provides a way to compute this gradient:
   $$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t\right]$$

where G_t is the return, often calculated as the discounted sum of future rewards from time step t[20].

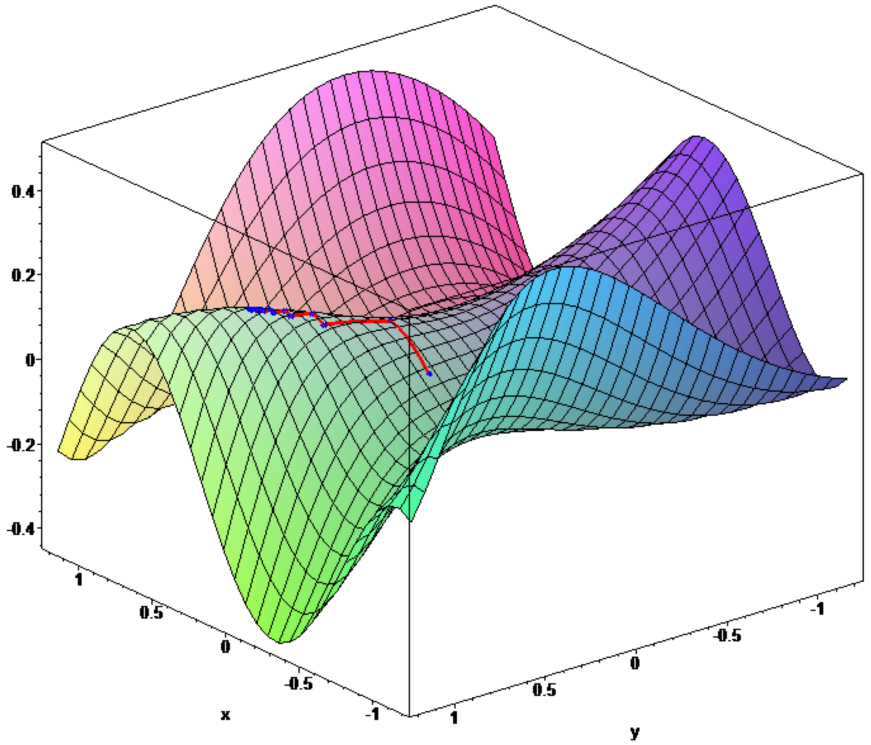


Figure 1: Policy Gradient Methods: Gradient ascent on reward

5. **Algorithm Steps**

    - **Sampling**: Generate trajectories by interacting with the environment using the current policy.
    - **Gradient Estimation**: Estimate the gradient of the policy's performance using the sampled trajectories.
    - **Update**: Adjust the policy parameters using the gradient estimate to improve the policy.

6. **Common Policy Gradient Algorithms**

    - **REINFORCE**: A basic policy gradient algorithm that uses Monte Carlo methods to estimate the gradient of the expected return.
    - **Actor-Critic**: Combines policy gradient methods (actor) with value function approximation (critic) to reduce variance in gradient estimates, leading to more stable learning.
    - **Proximal Policy Optimization (PPO)**: A popular variant that uses a clipped objective to ensure stable updates, making it easier to tune and more robust in practice.

2.4.1 Advantages of Policy Gradient Methods

- **Continuous Action Spaces**: Unlike methods like Q-learning that struggle with continuous actions, policy gradient methods naturally handle continuous action spaces.

- **Stochastic Policies**: These methods can learn stochastic policies, which are beneficial in environments requiring exploration or where actions need to be randomized.

2.4.2 Challenges and Considerations

- **Sample Efficiency**: Policy gradient methods can be sample-inefficient because they often require many interactions with the environment to compute accurate gradients.

- **High Variance**: The gradient estimates can have high variance, leading to unstable training. Techniques like baselines (e.g., subtracting a value function estimate from the return) or variance reduction methods are often used to mitigate this.

- **Sensitivity to Hyperparameters**: These methods can be sensitive to the choice of learning rate, discount factor, and other hyperparameters.

2.5 REINFORCE

REINFORCE is a fundamental algorithm in reinforcement learning, specifically in the category of policy gradient methods. It is used to optimize the policy, which is a function mapping states to actions, in a way that maximizes the expected cumulative reward.

The key idea behind REINFORCE is to adjust the policy parameters to increase the probability of actions that lead to higher rewards[17][20].

1. **Policy and Parameterization**: In REINFORCE, the policy is parameterized by a set of parameters θ, and denoted as $\pi_\theta(a|s)$, where a is the action and s is the state. The goal is to find the optimal set of parameters θ that maximizes the expected cumulative reward.

2. **Objective Function**: The objective of REINFORCE is to maximize the expected cumulative reward, which is the sum of rewards obtained by following the policy over time.

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t)\right]$$

where $R(s_t, a_t)$ is the reward at time step t, and γ is the discount factor that balances immediate and future rewards[19].

3. **Gradient Estimation**: To maximize $J(\theta)$, REINFORCE computes the gradient of the objective function with respect to the policy parameters, $\nabla_\theta J(\theta)$. This gradient indicates how the parameters should be adjusted to increase the expected reward. According to the policy gradient theorem, the gradient can be computed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t\right]$$

where G_t is the cumulative discounted reward from time step t onwards[19][17].

4. **Parameter Update**: The policy parameters θ are updated by moving them in the direction of the gradient, using a step size determined by a learning rate α:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

This update increases the probability of actions that lead to higher rewards, effectively improving the policy over time.

REINFORCE is considered a straightforward yet powerful algorithm for learning policies, especially when combined with function approximators like neural networks. However, it has high variance, meaning the gradients can be noisy, which can slow down learning[17].

Algorithm 1 REINFORCE[17]: Monte-Carlo Policy-Gradient Control (episodic) for π_*

```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^d$ (e.g., to 0)
loopforever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot \mid \cdot, \boldsymbol{\theta})$
    for each step of the episode $t = 0, 1, \ldots, T-1$ do
        $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta})$
    end for
end loop
```

2.5.1 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is particularly favored for its balance between simplicity and performance. PPO is part of the family of policy gradient methods, where the goal is to directly optimize the policy to maximize cumulative reward.PPO builds on the strengths of earlier methods, particularly the Trust Region Policy Optimization (TRPO) algorithm while addressing some of its complexity and computational challenges.

The policy is parameterized by a set of parameters θ, represented as a neural network. The objective is to maximize the expected cumulative reward by adjusting θ. The basic idea is to compute the gradient of the expected reward with respect to the policy parameters, $\nabla_\theta J(\theta)$, and then update the policy parameters in the direction that increases this reward.

Objective Function in PPO: PPO aims to improve upon standard policy gradient methods by stabilizing training. Instead of directly applying the gradient, PPO introduces a new objective function that prevents the policy from making excessively large updates. The PPO objective function ensures that policy updates are efficient and safe. It uses a *clipped surrogate objective*, which prevents the policy from deviating too much from the current policy during training[11][13]. The objective function can be expressed as:

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

- $r_t(\theta)$ is the probability ratio between the new policy and the old policy.
- $\hat{A}_t$ is the advantage estimate at time step t, which measures how much better (or worse) taking an action was compared to the expected action.

- ϵ is a small hyperparameter that controls the range of allowable updates.

The probability ratio $r_t(\theta)$ is defined as:

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

where:

- $\pi_\theta(a_t \mid s_t)$ is the probability of taking action a_t in state s_t under the new policy with parameters θ.
- $\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ is the probability of taking action a_t in state s_t under the old policy with parameters θ_{old}.

The ratio $r_t(\theta)$ provides a relative measure of how much more or less likely the new policy is to take the same action compared to the old policy:

- $r_t(\theta) > 1$: The new policy assigns a higher probability to the action a_t than the old policy.
- $r_t(\theta) < 1$: The new policy assigns a lower probability to the action a_t than the old policy.
- $r_t(\theta) = 1$: The new policy and the old policy assign the same probability to the action a_t.

Clipping Mechanism: The clipping mechanism is a core innovation in PPO. It ensures that the probability ratio $r_t(\theta)$ does not deviate too much from 1, effectively preventing overly large policy updates that could destabilize training. If the policy update is too large (i.e., $r_t(\theta)$ goes outside the range $[1-\epsilon, 1+\epsilon]$), the objective function penalizes it by using the clipped value. This prevents drastic changes to the policy, ensuring more stable learning[11][13].

Advantages of PPO:

- **Sample Efficiency:** PPO strikes a good balance between making effective policy updates and reusing data, leading to better sample efficiency.
- **Stability:** The clipping mechanism in PPO ensures stable training by preventing large policy updates, reducing the likelihood of training instability.

Training Process: During training, the PPO algorithm collects a batch of experiences (states, actions, rewards) by interacting with the environment. It then calculates the advantage function and the probability ratios for these experiences. The policy parameters are updated by optimizing the clipped objective function using gradient ascent[11].

Algorithm 2 PPO-Clip

```
Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
for $k = 0, 1, 2, \ldots$ do
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
    Update the policy by maximizing the PPO-Clip objective:
        $$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \hat{A}_t, g(\epsilon, \hat{A}_t) \right),$$
    typically via stochastic gradient ascent with Adam.
    Fit value function by regression on mean-squared error:
        $$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$
    typically via some gradient descent algorithm.
end for
```

2.6 Reinforcement Learning in Quadcopter Control

Reinforcement Learning (RL) has emerged as a promising alternative for controlling complex systems like quadcopters. RL allows an agent to learn optimal control policies through interaction with the environment, making it well-suited for tasks where precise modeling is difficult. Several studies have explored the use of RL in quadcopter control, focusing on tasks like trajectory tracking, obstacle avoidance, and hovering. RL methods such as Q-learning[8], Deep Q-Networks (DQN)[22], and Policy Gradient methods[10][9]have been applied with varying degrees of success[2][1].

In recent research, autonomous navigation of unmanned aerial vehicles (UAVs) in urban environments has been identified as a challenging task, primarily due to the complexity of the environment and the inherent battery limitations of the UAV. The approach employs the Proximal Policy Optimization (PPO) algorithm for path planning while incorporating collision avoidance mechanisms. Notably, the model also accounts for the UAV's energy constraints to mitigate the risk of crashes caused by low battery levels. A virtual environment simulating a random three-dimensional space with obstacles was created for training and evaluation purposes. The framework demonstrated high effectiveness, with 90% of the simulated missions successfully completed[4].

3 Implementation of Reinforcement Learning Environments and Algorithms

I took a modular approach, as advised by my supervisor, Dr. Michael Fairbank, to address the problem of controlling a quadcopter using reinforcement learning. In reinforcement learning, the agent interacts with the environment and receives rewards—either positive or negative—based on its actions. This concept is intuitive to humans; we interact with the world and receive feedback in the form of rewards. However, translating this process to a computer requires complex skills. It involves developing an environment that can simulate the real world, designing a reward function that can evaluate actions as good or bad, and creating a control policy that can learn from interactions with the environment and rewards.

To tackle this complex task, I began by using available tools to understand the fundamentals of reinforcement learning. I used custom environments provided by OpenAI's Gym, an open-source library available on GitHub. Gym offers multiple environments, such as Lunar Lander, CartPole, and others, each with its own reward functions. We can implement our own control policies to interact with these environments. I experimented with three control policies to understand their limitations and implementation. The first policy was the REINFORCE algorithm, written from scratch and provided by Dr. Michael Fairbank. The second was PPO (Proximal Policy Optimization), and the third was A2C (Advantage Actor-Critic).

3.1 OpenAI Gym

OpenAI Gym is a widely recognized toolkit for developing and comparing reinforcement learning algorithms. Introduced by Brockman [3], Gym provides a diverse collection of environments that serve as benchmarks for testing and evaluating reinforcement learning methods. The framework supports a range of tasks from simple classical control problems to complex robotic simulations, facilitating the development of generalizable and robust algorithms. One of its key features is the uniform API, which simplifies the integration of various algorithms and environments, thereby streamlining the process of experimentation and evaluation. Two of the well-known environments included in OpenAI Gym are the **Lunar Lander** and **CartPole environments**. Additionally, Gym also allows the creation of **custom environments** for specific research needs.

3.1.1 Lunar Lander Environment

The Lunar Lander environment is a classic control problem that simulates a lunar lander attempting to land on a target on the moon's surface. The lander must manage its fuel consumption while navigating the terrain and avoiding obstacles.

Objective: The goal is to land the spacecraft on a designated landing pad. The agent must control the lander's thrusters to slow its descent and align it with the pad.

State Space: The state is represented by an 8-dimensional vector that includes the lander's position (x and y coordinates), velocity (in x and y directions), angle, angular velocity, and indicators for whether the left or right leg is in contact with the ground.

Action Space: The action space is discrete with four possible actions:

1. Do nothing.

2. Fire left engine.
3. Fire main engine.
4. Fire right engine.

Rewards: After every step, a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode.

For each step, the reward:

- is increased/decreased the closer/further the lander is to the landing pad.
- is increased/decreased the slower/faster the lander is moving.
- is decreased the more the lander is tilted (angle not horizontal).
- is increased by 10 points for each leg that is in contact with the ground.
- is decreased by 0.03 points each frame a side engine is firing.
- is decreased by 0.3 points each frame the main engine is firing.

The episode receives an additional reward of -100 or +100 points for crashing or landing safely, respectively.
An episode is considered a solution if it scores at least 200 points[3].

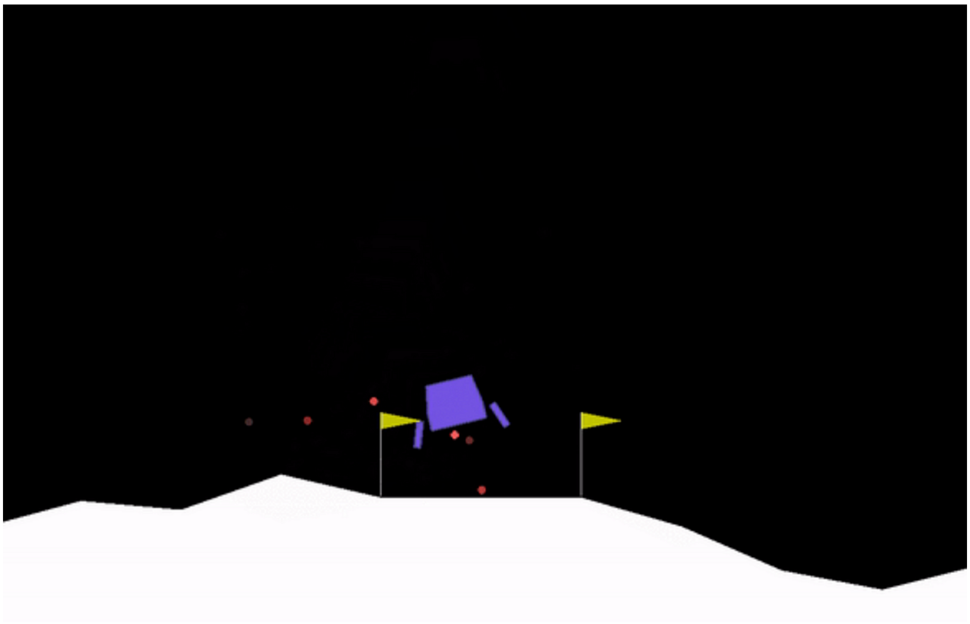


Figure 2: Lunar Lander Environment

3.1.2 CartPole Environment

The CartPole environment is a foundational RL task where a pole is attached to a cart, and the goal is to balance the pole by moving the cart left or right along a track.

Objective: The objective is to keep the pole upright for as long as possible while keeping the cart within the bounds of the track.

State Space: The state is represented by a 4-dimensional vector that includes the cart's position, velocity, pole's angle, and angular velocity.

Action Space: The action space is discrete with two possible actions:

1. Move the cart left.

2. Move the cart right.

Rewards: The agent receives a reward of +1 for every time step that the pole remains upright. The episode ends when the pole falls over or the cart moves too far to either side.

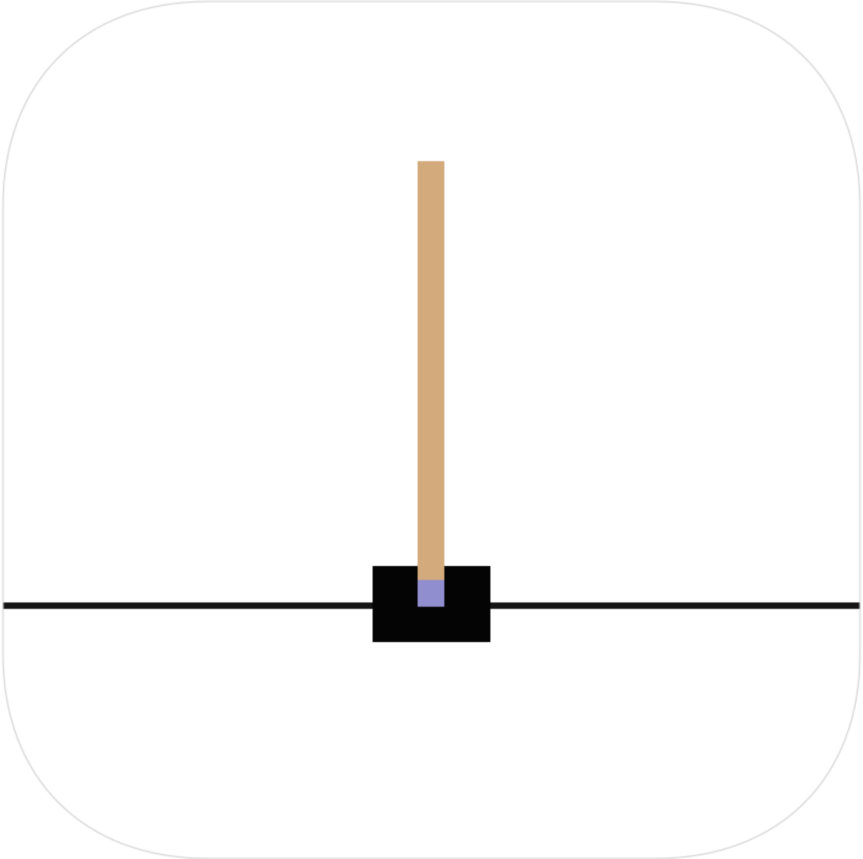


Figure 3: Cart Pole Environment

3.1.3 Custom Environments

OpenAI Gym allows users to create custom environments to suit specific research or application needs. This is particularly useful when existing environments do not fully capture the complexity or particularities of the task being studied.

Steps to Create a Custom Environment:

1. **Define the Environment Class**: Inherit from the `gym.Env` class and implement the necessary methods (`__init__`, `step`, `reset`, and optionally `render` and `close`).

2. **State Space and Action Space**: Define the state and action spaces using Gym's built-in space types like `Box`, `Discrete`, etc.

3. **Reward Function**: Implement a reward function that reflects the objectives of the environment.

4. **Register the Environment**: Use Gym's `register` function to make the custom environment accessible through the Gym API.

3.2 Stable Baselines

Stable Baselines is a widely used library that provides reliable implementations of several popular reinforcement learning (RL) algorithms. It builds upon OpenAI Baselines. The library is designed to support the development and testing of RL algorithms in various environments, including those provided by OpenAI Gym, as well as custom environments[7].

Stable Baselines can implement several RL algorithms, ranging from value-based methods to policy gradient approaches. Among these, Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A2C) are particularly notable for their effectiveness in a wide range of tasks, from simple control problems to more complex, high-dimensional environments.

3.2.1 Proximal Policy Optimization (PPO) Implementation in Stable Baselines3

The `stable_baselines3` library provides a robust implementation of PPO that is configurable through a variety of parameters, allowing for fine-tuned control over the learning process.

3.2.1.1 Key Parameters [16][7]

- **policy**: Specifies the type of policy model to be used, such as `MlpPolicy` (multilayer perceptron) for environments with flat observations or `CnnPolicy` for environments with image-based observations.

- **env**: The environment in which the agent will be trained. This can be any environment compatible with OpenAI Gym, including custom environments.

- **learning_rate** (default: 0.0003): The step size used for gradient descent. This parameter controls how quickly the model learns; lower values slow down learning but provide more stable updates.

- **n_steps** (default: 2048): The number of steps to run in the environment per update. This parameter affects the amount of data collected before updating the model, influencing the trade-off between computation time and learning stability.

- **batch_size** (default: 64): The number of samples used in each mini-batch for gradient descent. Smaller batch sizes lead to more frequent updates, which can speed up learning but may introduce more noise into the process.

- **n_epochs** (default: 10): The number of passes through the entire batch of training data for each update. Higher values can improve learning by allowing more opportunities to refine the policy using the same data.

- **gamma** (default: 0.99): The discount factor for future rewards. This parameter determines how much the algorithm values long-term rewards versus immediate rewards.

- **gae_lambda** (default: 0.95): The lambda parameter used in Generalized Advantage Estimation (GAE), helps reduce the variance of the advantage function, leading to more stable training.

- **clip_range** (default: 0.2): The clipping range for the policy objective function. This parameter prevents the policy from updating too drastically in a single step, which helps maintain stability.

- **clip_range_vf** (default: None): The clipping range for the value function. This is typically left at its default setting but can be adjusted to prevent large updates to the value function in a similar manner to the policy clipping.

- **normalize_advantage** (default: True): If set to `True`, the advantages are normalized before updating the policy, which can lead to improved performance by stabilizing the learning process.

- **ent_coef** (default: 0.0): The coefficient for the entropy term in the loss function. Adding entropy to the loss **encourages exploration** by preventing the policy from becoming too deterministic.

- **vf_coef** (default: 0.5): The coefficient for the value function loss. This parameter controls the balance between optimizing the policy and improving the accuracy of the value function estimate.

- **max_grad_norm** (default: 0.5): The maximum norm for the gradients. This helps in preventing exploding gradients, which can destabilize training.

- **use_sde** (default: False): If set to `True`, State-Dependent Exploration (SDE) is used, which can improve exploration in environments with continuous action spaces.

- **target_kl** (default: None): A target for the KL divergence between the old and new policy, used to stop the optimization early if the updates become too large.

- **tensorboard_log** (default: None): Specifies the directory where TensorBoard logs are saved. This allows for monitoring of the training process using TensorBoard.

- **policy_kwargs**: A dictionary that allows users to pass additional arguments to the policy, such as the number of layers or units in the neural network.

- **device** (default: 'auto'): The device (CPU or GPU) on which the model will be trained. The default setting automatically selects the best available device.

3.2.1.2 Implementation Details The algorithm uses a mix of on-policy updates and a clipping mechanism to ensure stability during training, making it particularly suitable for environments with high-dimensional states or action spaces. Generalized Advantage Estimation (GAE) and the ability to normalize advantages further enhance the convergence of the learning process.

This implementation of PPO is compatible with custom environments. The ability to fine-tune parameters such as learning rate, batch size, and clipping range allows for careful control over the balance between exploration and exploitation, leading to more efficient learning.

3.2.2 Advantage Actor-Critic (A2C) implementation in Stable Baselines3

[15][7]

Advantage Actor-Critic (A2C) is a popular reinforcement learning algorithm that extends the Actor-Critic framework by incorporating advantages to reduce variance in the gradient updates, leading to more stable learning. A2C is an on-policy algorithm, meaning it updates its policy based on actions taken according to the current policy, which makes it well-suited for environments where the policy needs to adapt dynamically to changing situations. A2C operates by maintaining two separate networks: the actor, which decides the best action to take in a given state, and the critic, which evaluates how good the action taken is based on the expected future rewards. The algorithm computes the advantage function to determine how much better or worse an action is compared to the average action, which helps in reducing the variance of policy gradient estimates[7].

3.2.2.1 Key Parameters

- **policy**: Defines the type of policy network used (e.g., `MlpPolicy` for flat observations or `CnnPolicy` for image-based inputs). The policy determines how the agent selects actions.

- **env**: Specifies the environment in which the agent will be trained. This could be any environment compatible with OpenAI Gym, or a custom environment.

- **learning_rate** (default: 0.0007): The step size for gradient updates. A higher learning rate can speed up training but may lead to less stable updates.

- **n_steps** (default: 5): The number of steps the agent takes in the environment before updating the policy. This parameter affects how frequently the model is updated, influencing the trade-off between training stability and computation time.

- **gamma** (default: 0.99): The discount factor used to calculate the present value of future rewards. It determines the importance of long-term rewards compared to immediate rewards.

- **gae_lambda** (default: 1.0): The lambda parameter used in Generalized Advantage Estimation (GAE), helps in reducing variance in the advantage estimates. A value of 1 corresponds to standard advantage estimation without variance reduction.

- **vf_coef** (default: 0.25): The coefficient for the value function loss in the total loss function. This parameter controls how much weight is given to improving the critic's accuracy versus optimizing the policy.

- **ent_coef** (default: 0.01): The coefficient for the entropy regularization term in the loss function. Adding entropy encourages exploration by ensuring that the policy does not become too deterministic too quickly.

- **max_grad_norm** (default: 0.5): The maximum norm for gradient clipping. This helps in preventing the problem of exploding gradients, which can destabilize training.

- **use_rmsprop** (default: True): If set to `True`, the RMSProp optimizer is used instead of the default Adam optimizer. RMSProp can be beneficial in some environments due to its ability to adapt the learning rate based on the variance of the gradients.

- **normalize_advantage** (default: True): Normalizes the advantage function to have zero mean and unit variance, which can help improve the stability of the updates.

- **tensorboard_log** (default: None): Specifies the directory where TensorBoard logs will be saved. This allows for detailed monitoring and visualization of the training process.

- **policy_kwargs**: A dictionary of additional arguments passed to the policy, such as the architecture of the neural network, allowing for customization of the model complexity.

- **device** (default: 'auto'): Specifies the device (CPU or GPU) used for training. The default setting automatically selects the most appropriate device based on availability.

3.2.2.2 Implementation Details A2C is particularly effective in environments where the trade-off between exploration and exploitation is crucial, such as those involving sparse rewards or complex navigation tasks. It provides a good balance between performance and computational efficiency.

3.3 REINFORCE Algorithm Implementation

The REINFORCE algorithm is provided by Dr. Michael Fairbank and is written from scratch [6]. It has the following components.

1. **Neural Network Policy** A neural network is used as the policy model, which maps observations from the environment to a probability distribution over possible actions. The network is constructed using TensorFlow's Keras API, consisting of input layers corresponding to the environment's observation space, 2 hidden layers with 6 nodes each with shortcut connections, and an output layer that uses a softmax activation function to produce action probabilities.

2. **Gradient Calculation**: After an episode, the algorithm calculates the policy gradient. This gradient is used to update the policy network in a direction that maximizes expected rewards. The gradient calculation considers the discounted rewards from the episode.

3. **Gradient Ascent**: The calculated gradients are applied using an optimizer (Adam) to update the network weights.

4. **Discounted Reward Calculation**: The rewards obtained during an episode are discounted using a discount factor (0.98) to emphasize earlier rewards.

5. **Running the Policy**: At each time step, the policy network outputs action probabilities given the current observation. An action is then sampled from this distribution.

6. **Training Process** The training loop runs for a 1000 number of episodes. During each episode, the agent interacts with the environment, collects observations, actions, and rewards, and uses these to update the policy network. The training includes periodic evaluation and visualization of performance to monitor progress.

7. **Model Saving and Playback** After training, the model is saved for future use. A demo mode allows the trained agent to be observed in action.

Algorithm 3 REINFORCE Algorithm for Lunar Lander[6]

```
Initialize: Policy network with random weights, learning rate $\alpha$, discount factor $\gamma$
for each episode do
    Reset the environment and initialize an empty list to store episode transitions
    while episode is not done do
        Observe the current state from the environment
        Choose an action based on the policy network's probabilities
        Execute the chosen action in the environment
        Record the reward and the next state
    end while
    Compute the accumulated discounted rewards for the episode
    Calculate the policy gradient using the episode's states, actions, and rewards
    Update the policy network weights in the direction of the policy gradient
end for
Output: Trained policy network
```

3.4 Training the REINFORCE, PPO and A2C on Lunar Lander and Cartpole

Lunar lander: The REINFORCE, PPO, and A2C algorithms were trained in the lunar lander environment for 1000 episodes. REINFORCE runs for 400 steps per episode, turning off the engine, action[0] 3.1.1 after 400 steps to ensure convergence while still collecting rewards during free fall. PPO and A2C were trained with their default parameters 3.2.1, 3.2.2 for 1000 steps per episode.

3.5 Testing the REINFORCE, PPO and A2C on Lunar Lander and Cartpole

The trained policy networks for REINFORCE, PPO, and A2C are tested by loading their weights and performing actions based on the environment's state. The cumulative rewards for each episode are recorded and plotted to compare the performance of each policy.

4 Building the quadcopter physics model

The physics engine simulates the motion of a quadcopter in a vertical plane, governed by basic principles of Newtonian mechanics. The development process focused on the dynamics influenced by two forces: vertical thrust and weight. The key objective was to create a simple yet accurate model that captures the quadcopter's vertical movement in response to thrust inputs, which can be visualized in real-time using a graphical interface.

4.1 Mathematical Formulation

The quadcopter's vertical motion can be modeled by considering Newton's Second Law, which states that the net force acting on an object is equal to its mass multiplied by its acceleration ($F = ma$). For the vertical dynamics, the forces acting on the quadcopter are:

1. **Thrust (a)**: The upward force generated by the quadcopter's rotors.
2. **Weight (w)**: The downward force due to gravity, calculated as $w = mg$, where g is the acceleration due to gravity.

Given these forces, the net force F_{net} acting on the quadcopter in the vertical direction is:

$$F_{\text{net}} = a_{\text{input}} - w$$

Since $F_{\text{net}} = ma$, where a_{net} is the net acceleration, we can express the net acceleration as:

$$a_{\text{net}} = \frac{F_{\text{net}}}{m} = \frac{ma_{\text{input}} - mg}{m} = a_{\text{input}} - g$$

The state of the quadcopter is described by two variables:

- y: The vertical position (height) of the quadcopter.
- $\dot{y}$ (or v): The vertical velocity of the quadcopter.

The time evolution of the velocity and position can be expressed as:

$$\dot{y} = v$$

$$\dot{v} = a_{\text{input}} - g$$

These equations can be discretized using the Euler method, which provides a straightforward numerical approach to integrate these differential equations over small time steps Δt. The updates for the next state $(y_{\text{new}}, v_{\text{new}})$ are given by:

$$v_{\text{new}} = v_{\text{current}} + (a_{\text{input}} - g) \cdot \Delta t$$

$$y_{\text{new}} = y_{\text{current}} + v_{\text{new}} \cdot \Delta t$$

4.2 Implementation in Python

The following Python function implements the above discretized equations to compute the next state of the quadcopter given its current state and the applied thrust:

```
def calculate_next_state(current_state, action, delta_t=0.1):
    start_y, start_v = current_state
    # Compute the new velocity using the discretized acceleration equation
    new_v = start_v + (action - 9.8) * delta_t
    # Compute the new position using the discretized position equation
    new_y = start_y + new_v * delta_t

    # Boundary conditions to keep the quadcopter within the screen limits
    if new_y > 600 or new_y < 90:
        return 90, 0
    else:
        return new_y, new_v
```

The function `calculate_next_state` takes the current state of the quadcopter (vertical position and velocity), the thrust action, and the time step Δt as inputs. It computes the new velocity and position based on the thrust applied and returns the updated state.

4.2.1 Simulation and Visualization

To visually simulate the quadcopter's motion, a graphical interface was developed using the Pygame library. The quadcopter's movement is animated based on the calculated states, providing a real-time visual representation of the quadcopter's response to user inputs.

The simulation initiates with the quadcopter at a height = 300 units and zero velocity. The user can control the thrust by pressing the UP and DOWN keys, which correspond to increasing and decreasing the vertical thrust, respectively. The simulation loop continuously updates the quadcopter's state using the `calculate_next_state` function, and the graphical interface renders the quadcopter's current position on the screen. The real-time visualization helps in understanding the quadcopter's dynamics and the impact of thrust adjustments on its vertical motion.

The physics engine developed for this project offers a simplified yet effective simulation of a quadcopter's vertical dynamics, governed by basic principles of physics. By integrating the mathematical model with real-time visualization, the engine provides an interactive tool for understanding the behavior of a quadcopter under different thrust conditions. The approach can be further extended to include additional dynamics, such as rotational motion and horizontal movement, for a more comprehensive simulation.

Algorithm 4 Quadcopter Simulation Algorithm

```
Initialize the simulation environment and graphics window.
Load and resize the quadcopter image.
Initialize the quadcopter's initial state ($y_{current}, v_{current}$).
Set the time step $\Delta t$ and frame duration.
while simulation is running do
    Capture user input (e.g., UP and DOWN keys) to determine the action.
    Calculate the next state ($y_{new}, v_{new}$) using the calculate_next_state function:
        $v_{new} = v_{current} + (action - g) \cdot \Delta t$
        $y_{new} = y_{current} + v_{new} \cdot \Delta t$
    if $y_{new}$ is out of bounds then
        Reset the position and velocity to within bounds.
    end if
    Update the frame index for animation.
    Clear the screen.
    Render the quadcopter at the new position ($x_{fixed}, y_{new}$).
    Update the display.
    Control the frame rate by waiting for the frame duration.
end while
Exit the simulation and close the graphics window.
```

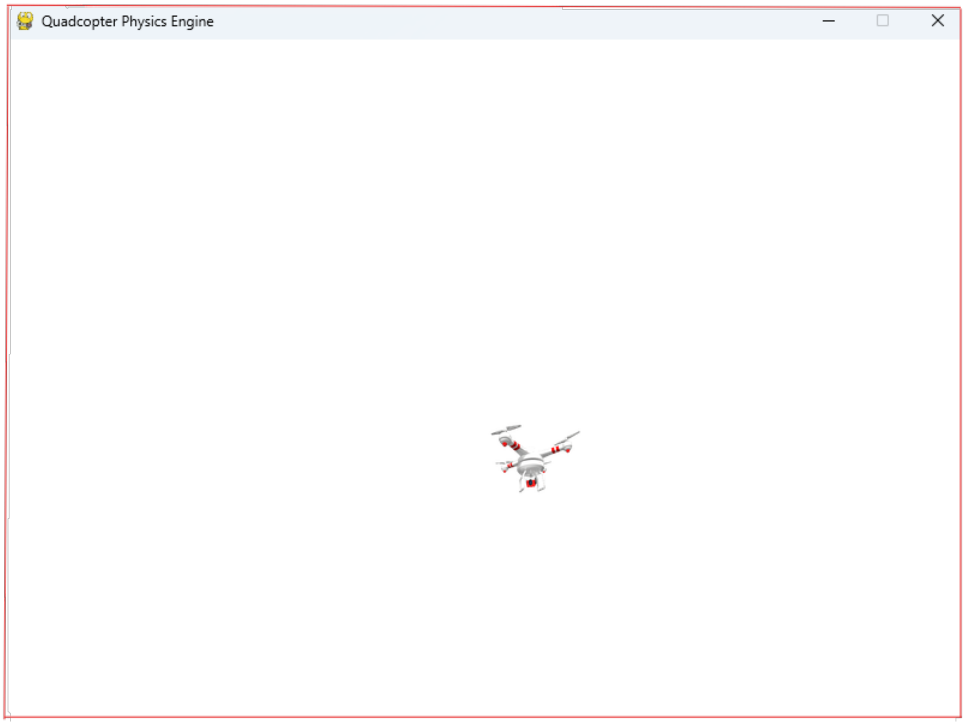


Figure 4: Quadcopter Physics Engine

4.3 Development of a Custom Gym Environment for Quadcopter Simulation

I adopted the physics engine to a custom GYM environment. The environment is constructed using the OpenAI Gym framework and integrates the physics engine, providing a structured setting for training and testing reinforcement learning algorithms.

4.3.1 Environment Setup

The custom environment, `QuadcopterEnv`, is designed as a subclass of the `gym.Env` class. The environment models the quadcopter's vertical dynamics, which are governed by its height and vertical velocity. The primary goal of the environment is to maintain the quadcopter at a target height of 300 units.

4.3.2 Action and Observation Spaces

The environment's action space and observation space are defined as follows:

- **Action Space:** A discrete action space with two possible actions:
  - `0`: No thrust applied.
  - `1`: Increase thrust [30 units].
  - `2`: decrease thrust [-10 units]

  These actions are mapped to thrust values, with `0` corresponding to no thrust and `1` corresponding to a thrust value of 30 units.

- **Observation Space:** A continuous observation space represented by a two-dimensional vector:
  - The first dimension corresponds to the quadcopter's height, bounded between 0 and 600 units.
  - The second dimension represents the vertical velocity, which can take any real value.

4.3.3 Initialization and Rendering

Upon initialization, the environment sets up the Pygame interface to render the quadcopter's motion visually. The quadcopter's images are loaded from a GIF file, resized, and prepared for rendering in the Pygame window. The environment's initial state is set with the quadcopter positioned at a height of 300 units and a vertical velocity of 0.

4.3.4 Dynamics and State Transition

The state transition in the environment is governed by a physics-based model that calculates the next state based on the current state and the applied action. The equations of motion are derived from Newton's second law:

$$v_{\text{new}} = v_{\text{current}} + (a_{\text{thrust}} - g) \cdot \Delta t$$

$$y_{\text{new}} = y_{\text{current}} + v_{\text{new}} \cdot \Delta t$$

where a_{thrust} is the input acceleration $0 =$ No thrust applied, $1 =$ Increase thrust 30 units, $2 =$ decrease thrust -10 units, g is the acceleration due to gravity, and Δt is the time step.

4.3.5 Reward Function

The reward function is designed to encourage behaviors that achieve the desired objective and discourage those that do not. In this task, the goal is to hover the quadcopter. The reward function is a critical component in shaping the agent's policy, helping it learn which actions are most beneficial in different states of the environment. Multiple reward functions were tested and described in 4.4.

4.3.6 Episode Termination

The environment defines the conditions under which an episode terminates:

- The episode ends after 500 steps, simulating a fixed time duration.
- The episode also terminates if the quadcopter's height exceeds 590 units or falls below 100 units, representing a crash or going out of operational bounds.

4.3.7 Integration with PPO

To train the quadcopter to maintain its position at the target height, the environment was integrated with the Proximal Policy Optimization (PPO) algorithm from the `stable_baselines3` library. The PPO model is initialized with the 1000 episodes default parameters of PPO 3.2.1, with the trained model saved periodically.

The custom environment is registered with Gym using the `gym.envs.registration.register` method, enabling it to be used as a standard Gym environment for reinforcement learning experiments.

4.4 Reward Engineering

In reinforcement learning (RL), reward engineering involves creating a reward system that effectively signals to the agent when it is performing correctly. For tasks like attitude control, the reward function needs to reflect how well the agent meets the specified attitude goals. As these goals grow more intricate and challenging it's essential to identify the most relevant performance metrics. This ensures that the intelligent control systems trained with RL can achieve optimal performance[9].

"As reinforcement-learning-based AI systems become more general and autonomous, the design of the reward mechanisms that elicit desired behaviors become both more important and more difficult" [5].

Different reward functions were tested for optimal control and discussed in (Section 5). Each of these reward functions offers a different approach to guiding the quadcopter's behavior and helps in understanding how different reward structures impact learning and performance in reinforcement learning tasks.

4.5 Standardization for testing to compare reward functions

The test reward function is designed to encourage the quadcopter to stay close to the target height of 300 units. The reward is calculated as the negative absolute difference between the current height and the target height, normalized by a factor of 100:

$$\text{reward} = -\frac{|y_{\text{new}} - 300|}{100} \tag{1}$$

This reward structure ensures that when each trained policy is tested, the results are collected based on a standardized test reward function. This works because, regardless of the reward function used during training, each policy receives the same reward for a given action during testing. By using this fixed reward function, we can fairly compare how well different reward functions perform.

5 Results

5.1 Quadcopter Control

This section explores the hypothesis testing and process of reward function evolution to reach the optimal control

5.1.1 Reward Function Design

In reinforcement learning, designing an effective reward function is crucial for guiding the agent towards achieving its objectives. Several reward functions have been proposed to optimize the quadcopter's performance in maintaining its target height. Two constraints were designed to help make the algorithm converge.

Constraint 1: Steps = 500 Terminate the episode

Constraint 2: Quadcopter exceeds the limits $y > 590$ or $y < 100$ Terminate the episode

5.1.2 Reward Function 1

```
if 300 < new_y < 350:
reward = 100
else:
reward = 0

if self.step_counter > 500:
done = True
```

The reward function assigned a reward of 100 while the quadcopter hovered between altitudes of 300-350 units. The episode was terminated after 500 steps, but no other constraints existed. The training graphs show that the agent accumulated a maximum reward of approximately 5500 units. However, when the model was loaded, the agent initially attempted to maximize its reward during the first 55 steps. Afterward, it began ascending because no reward was available for the remaining steps. The agent determined that it was better to avoid any penalty by simply ascending and ending the episode quickly, rather than adjusting inputs to maintain a 100-point reward. Thus, it concluded that sending out one input and finishing the episode was the optimal strategy.

(Figure 5) shows the mean episode length during the training of the model and (Figure 6) shows the mean episode rewards during the training of the model (Figure 7, 8, 9) shows the testing of the best reward function against the standardized test reward function (eq 1).

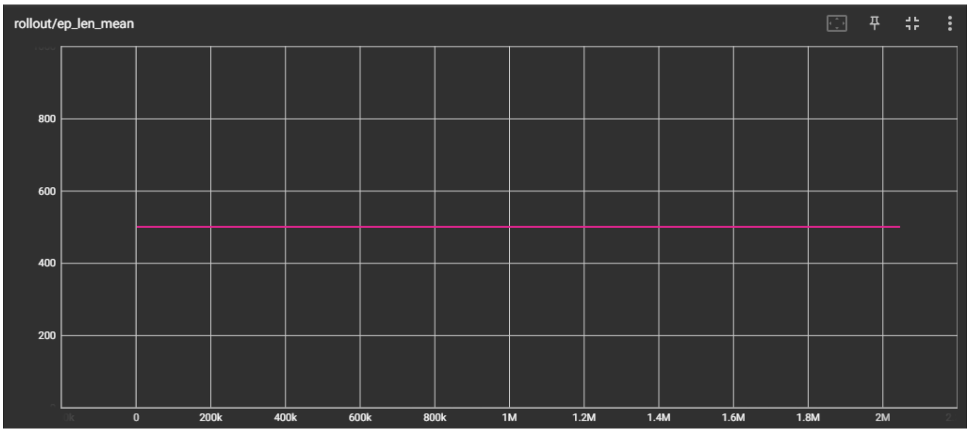


Figure 5: Reward 1 (mean ep length training)

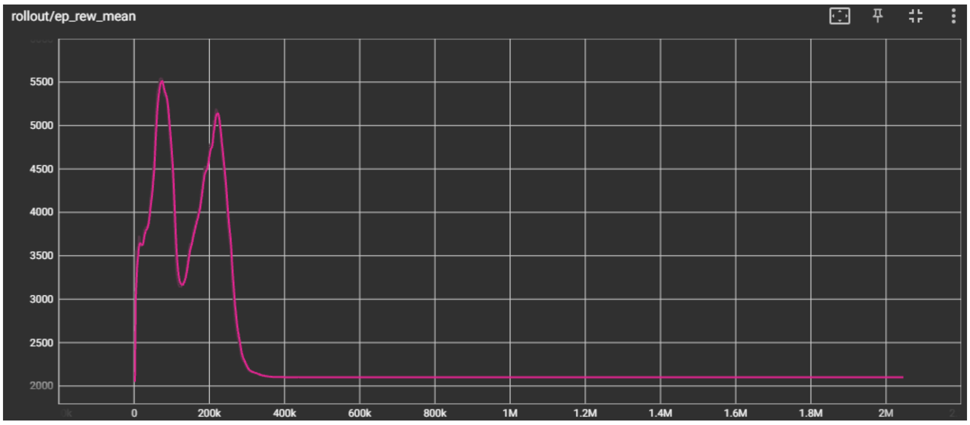


Figure 6: Reward 1 (mean ep reward training)

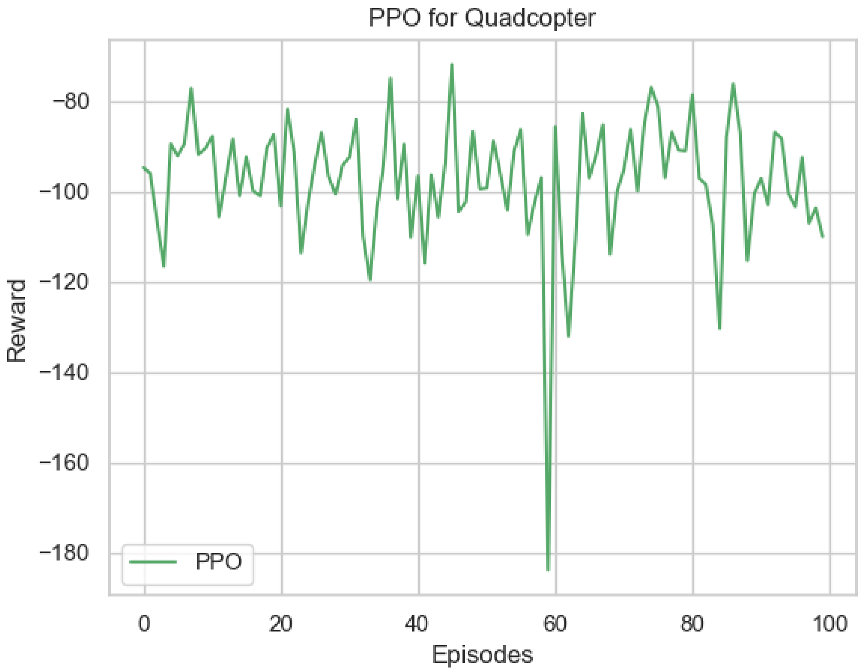


Figure 7: Testing Reward 1

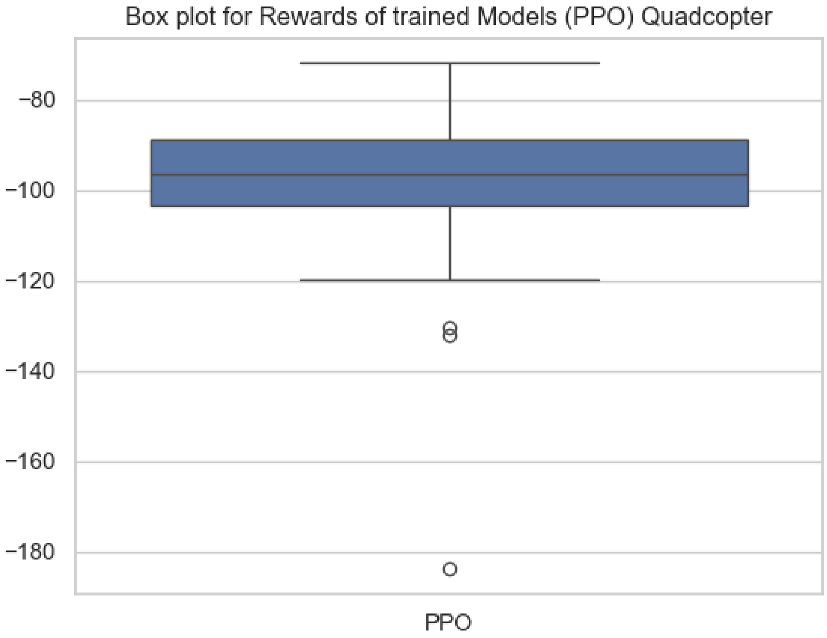


Figure 8: Box Plot of PPO Quadcopter, Reward 1

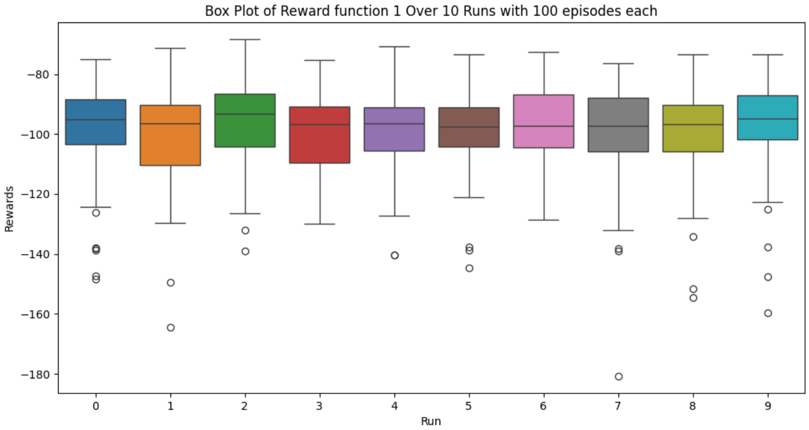


Figure 9: Box Plot of Reward function 1 Over 10 Runs with 100 episodes each

5.1.3 Reward Function 2

This reward function provides a gradient of rewards based on how close the quadcopter is to the target height. It is defined as follows:

```
if new_y > 400:
reward = -100
elif 350 < new_y < 400:
reward = -50
elif 300 < new_y < 350:
reward = 100
elif 150 < new_y < 300:
reward = -50
elif new_y < 150:
reward = -100
else:
reward = 0
```

In this function, rewards are assigned based on the height ranges: large negative rewards are given for heights far from the target (either too high or too low), while a reward of 100 is given for heights close to the target range (300 to 350 units). This design encourages the quadcopter to maintain its height within a specific range and penalizes deviations.

Since Reward Function 1 does not perform well in enabling the quadcopter to hover, a new gradient-based reward function was developed. The number of steps is limited to 500 units without any other constraint. This gradient reward function performs better than Reward Function 1; however, during the training phase, as shown in (Figure 11), the agent is still unable to accumulate higher rewards.

(Figure 10) shows the mean episode length during the training of the model and (Figure 11) shows the mean episode rewards during the training of the model (Figure 12, 13, 14) shows the testing of the best reward function against the standardized test reward function (eq 1).

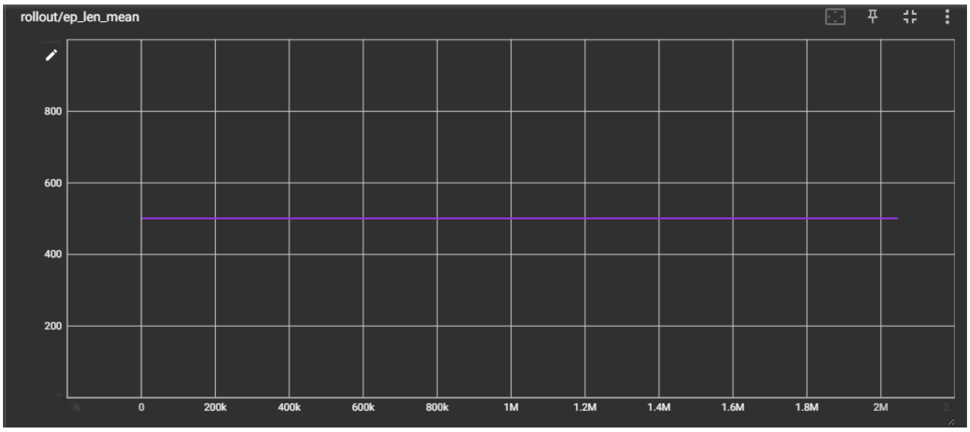


Figure 10: Reward 2 (mean ep length training)

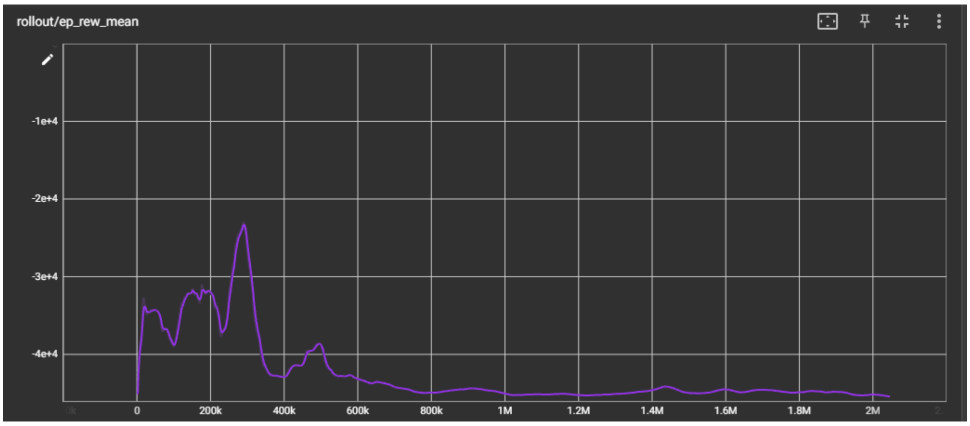


Figure 11: Reward 2 (mean ep reward training)

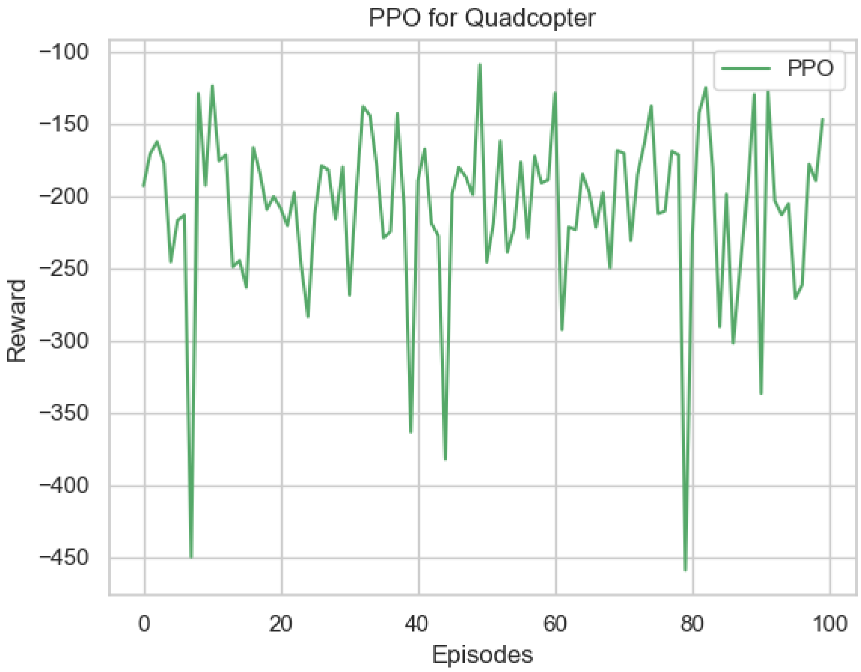


Figure 12: Testing Reward 2

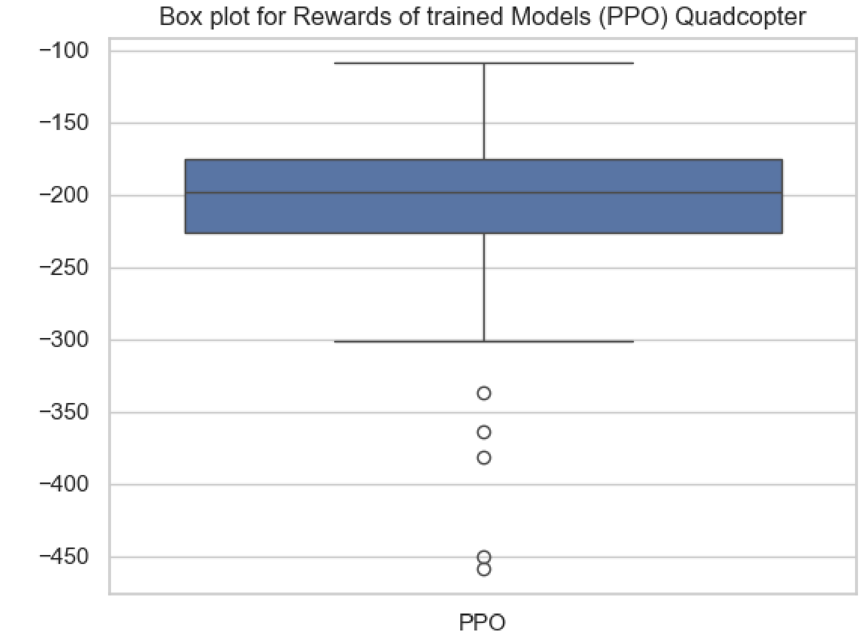


Figure 13: Box Plot of PPO Quadcopter, Reward 2

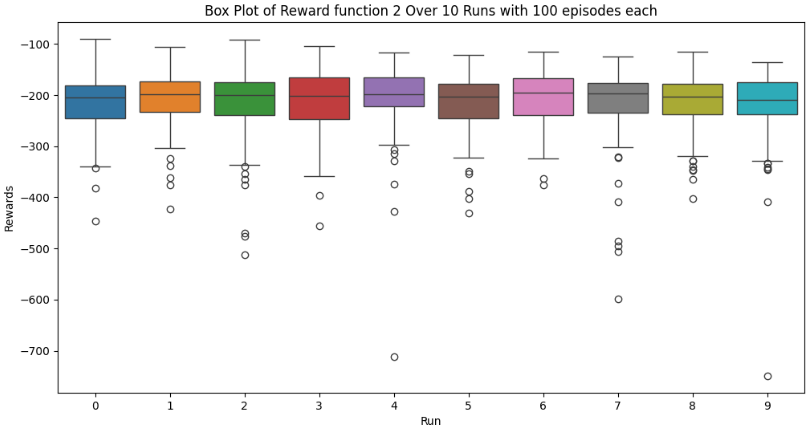


Figure 14: Box Plot of Reward function 2 Over 10 Runs with 100 episodes each

5.1.4 Reward Function 3

```
if new_y > 400:
reward = -100
elif 350 < new_y < 400:
reward = -50
elif 300 < new_y < 350:
reward = 100
elif 150 < new_y < 300:
reward = -50
elif new_y < 150:
reward = -100
else:
reward = 0

if self.step_counter > 500:
if self.step_counter > 500 or new_y > 590 or new_y < 100:
done = True
```

Reward Function 3 is similar to reward function 2 but with boundary constraints. Since Reward Function 2 performed poorly in accumulating higher rewards, a new constraint was introduced. It was hypothesized that because the state space is infinite, making it bounded by boundary constraints would help PPO converge. A new constraint was therefore introduced: if the quadcopter exceeds the limits $y > 590$ or $y < 100$, the episode is terminated. The reward accumulation during training improved; however, the quadcopter discovered that the optimal strategy was to terminate itself immediately, thereby accumulating less negative reward. As a result, the intended purpose of the constraint was ultimately defeated.

(Figure 15) shows the mean episode length during the training of the model and (Figure 16) shows the mean episode rewards during the training of the model (Figure 17, 18, 19) shows the testing of the best reward function against the standardized test reward function (eq 1).

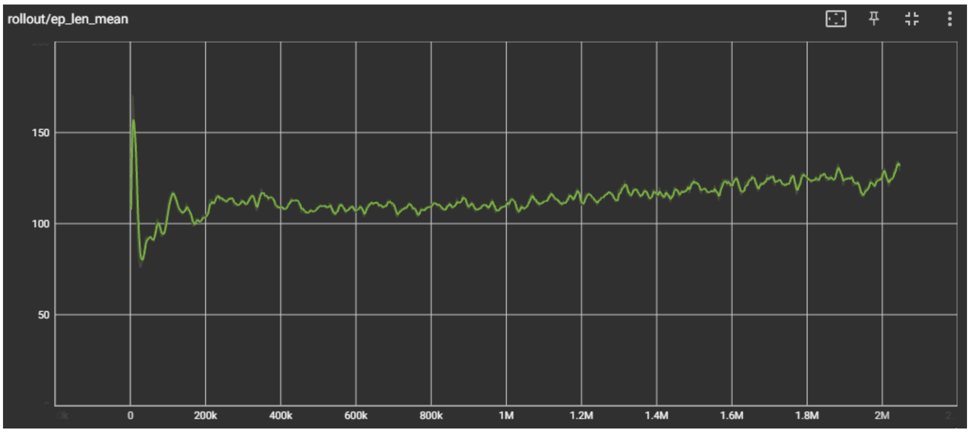


Figure 15: Reward 3 (mean ep length training)

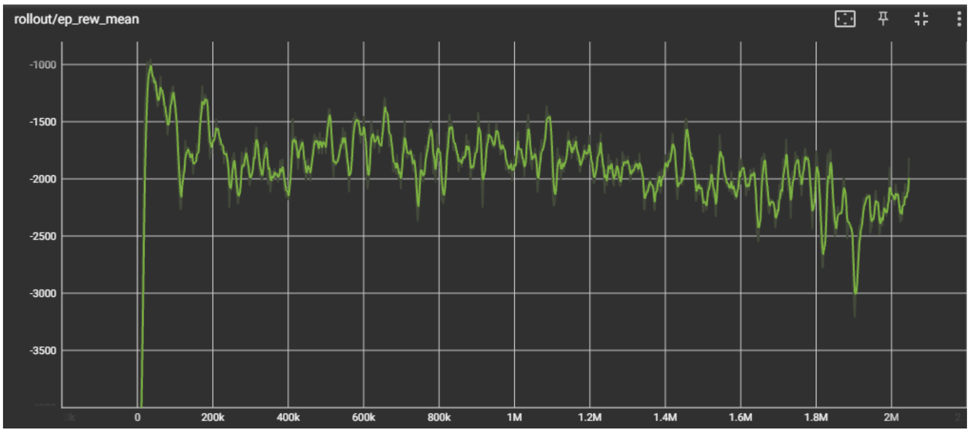


Figure 16: Reward 3 (mean ep reward training)

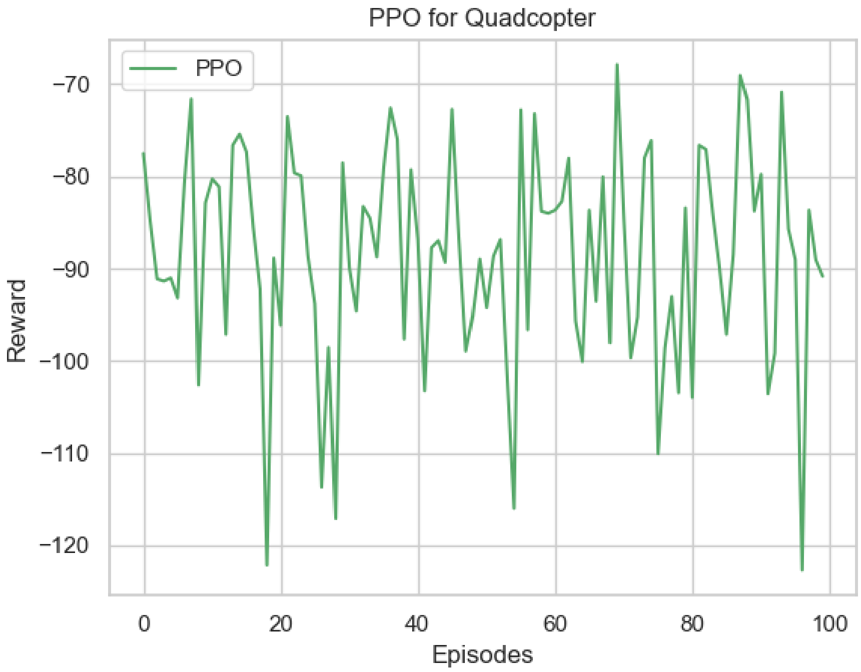


Figure 17: Testing Reward 3

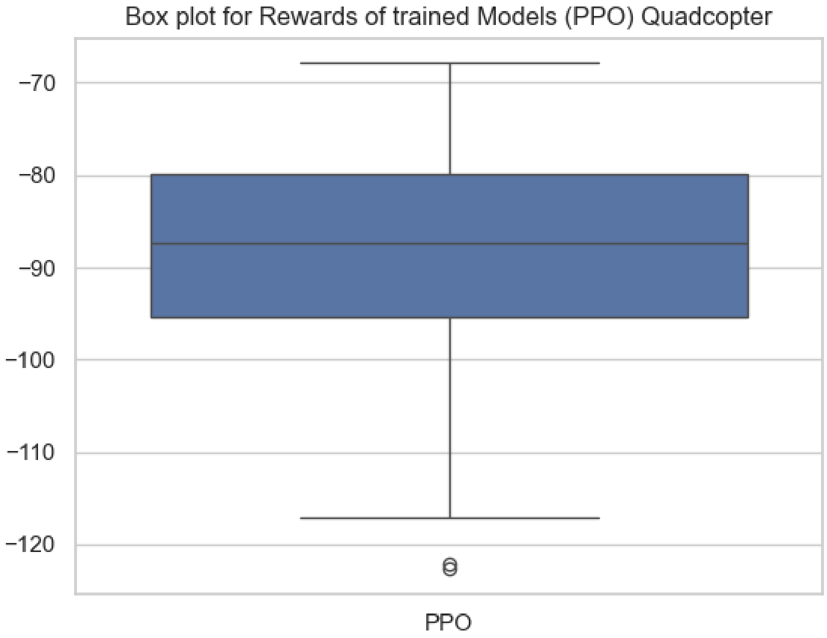


Figure 18: Box Plot of PPO Quadcopter, Reward 3

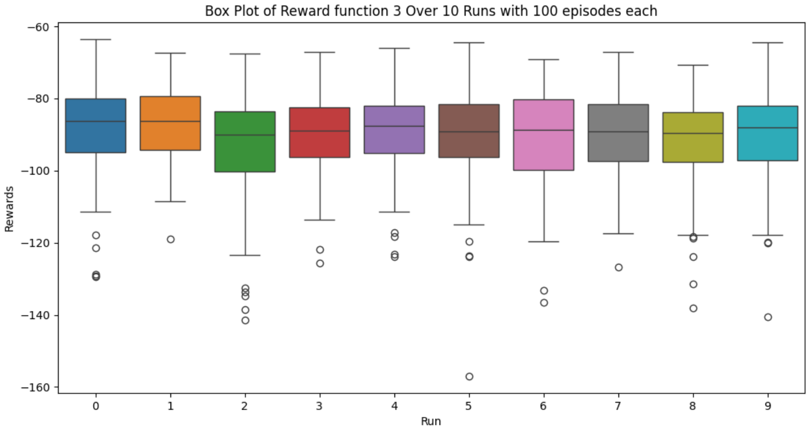


Figure 19: Box Plot of Reward function 3 Over 10 Runs with 100 episodes each

5.1.5 Reward Function 4

This function provides a more gradual reward structure:

```
if new_y > 400:
reward = -1
elif 350 < new_y < 400:
reward = -0.5
elif 300 < new_y < 350:
reward = 0
elif 150 < new_y < 300:
reward = -0.5
elif new_y < 150:
reward = -1
else:
reward = 0
```

This reward function assigns smaller penalties for heights that are farther from the target, creating a smoother gradient of rewards. A reward of 0 is given when the height is close to the target, while negative rewards are given for deviations.

Larger reward values could perturb the algorithm and result in high variance, so I tested the algorithm with smaller reward values. There were no boundary constraint, only step constraint. This reward function performed much better and was more successful compared to the other reward functions. The quadcopter was able to hover but eventually failed and fell due to gravity. However, this reward function provided the insight that scaling the reward to smaller values can help PPO converge.

(Figure 20) shows the mean episode length during the training of the model and (Figure 21) shows the mean episode rewards during the training of the model (Figure 22, 23, 24) shows the testing of the best reward function against the standardized test reward function (eq 1).

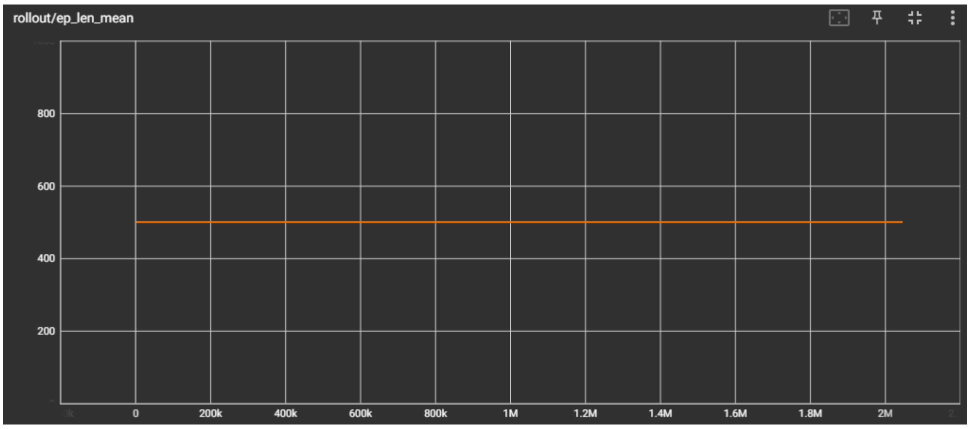


Figure 20: Reward 4 (mean ep length training)

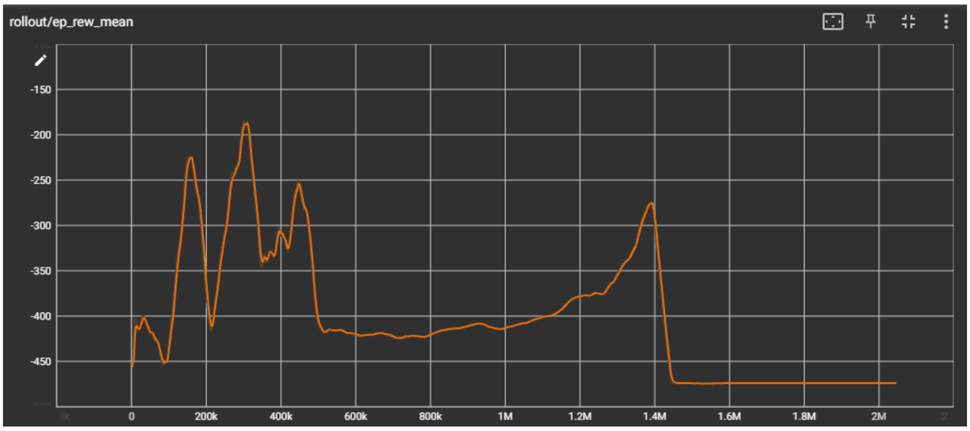


Figure 21: Reward 4 (mean ep reward training)

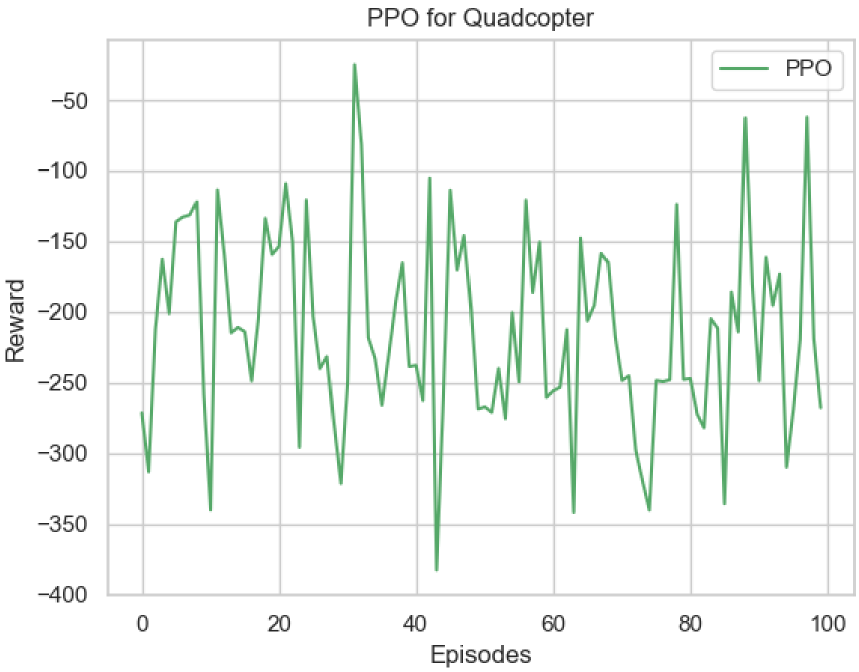


Figure 22: Testing Reward 4

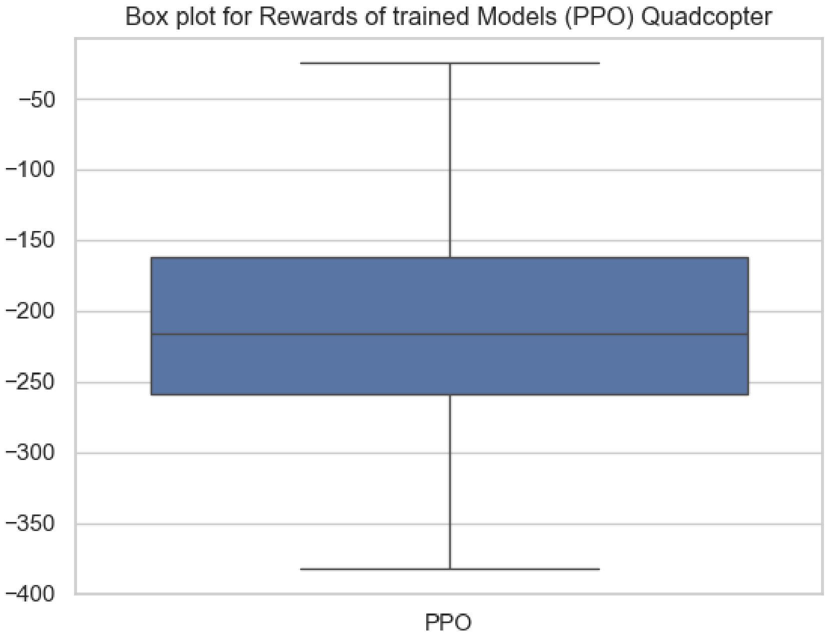


Figure 23: Box Plot of PPO Quadcopter, Reward 4

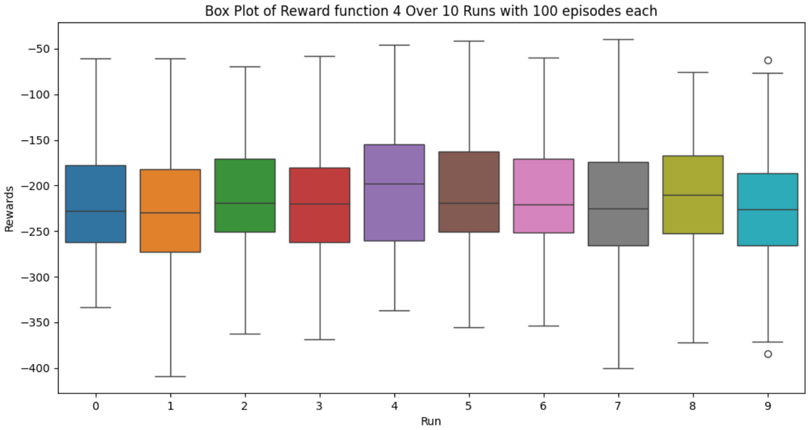


Figure 24: Box Plot of Reward function 4 Over 10 Runs with 100 episodes each

5.1.6 Reward Function 5

```
if new_y > 400:
reward = -1
elif 350 < new_y < 400:
reward = -0.5
elif 300 < new_y < 350:
reward = 0
elif 150 < new_y < 300:
reward = -0.5
elif new_y < 150:
reward = -1
else:
reward = 0

if self.step_counter > 500:
if self.step_counter > 500 or new_y > 590 or new_y < 100:
done = True
```

Reward 5 is just an extension of 4 with the boundary constraint. However, here again, the quadcopter discovered that the optimal strategy was to terminate itself immediately.

(Figure 25) shows the mean episode length during the training of the model and (Figure 26) shows the mean episode rewards during the training of the model (Figure 27, 28, 29) shows the testing of the best reward function against the standardized test reward function (eq 1).

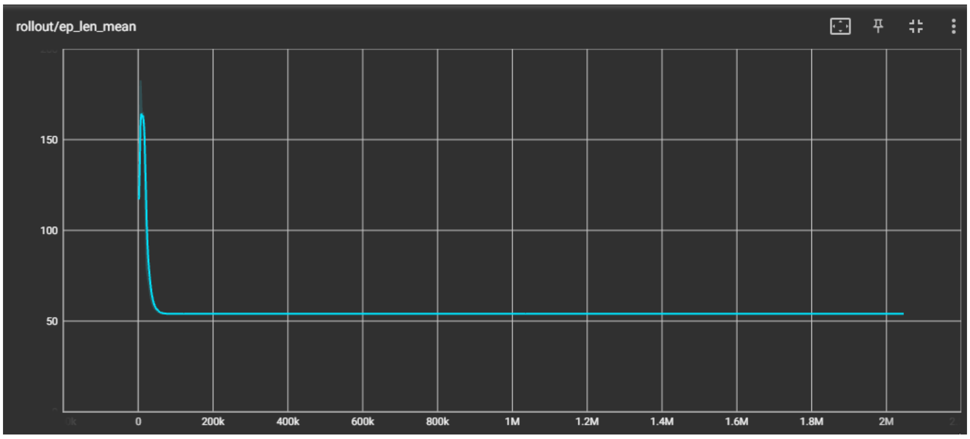


Figure 25: Reward 5 (mean ep length training)

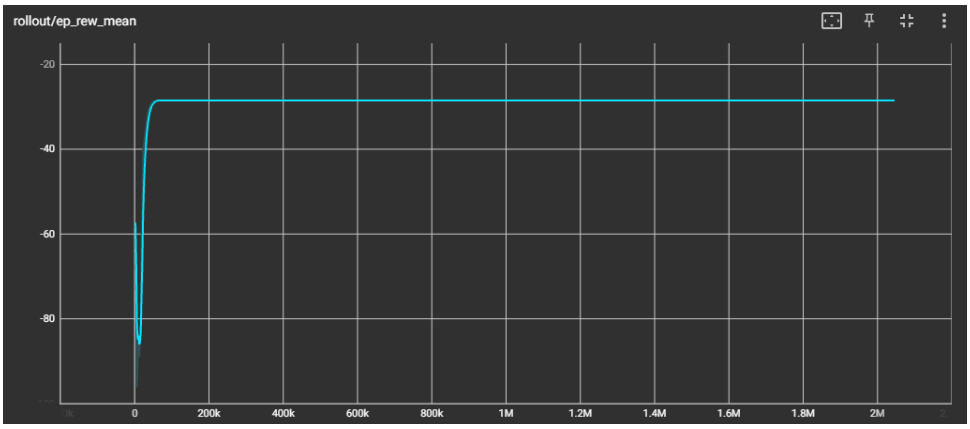


Figure 26: Reward 5 (mean ep reward training)

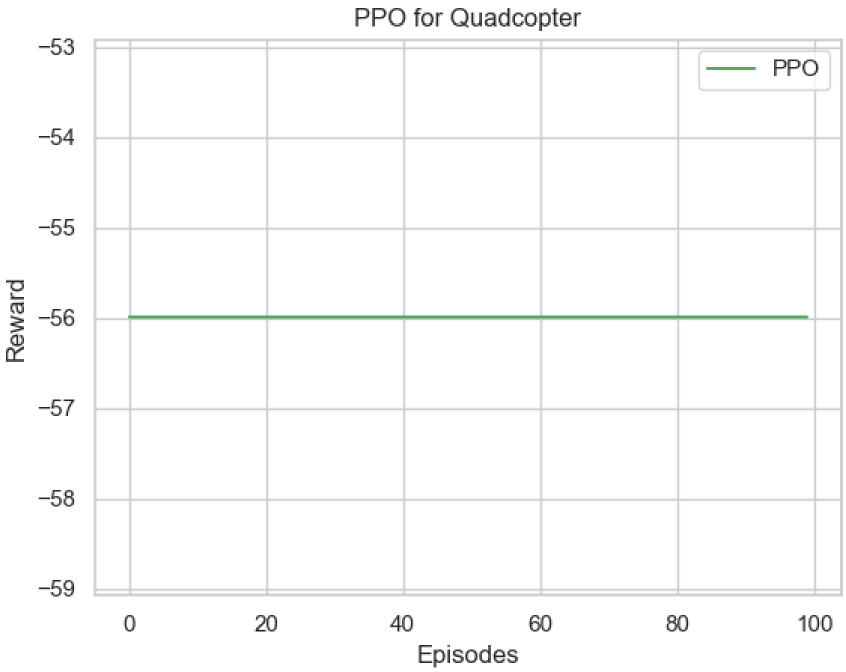


Figure 27: Testing Reward 5

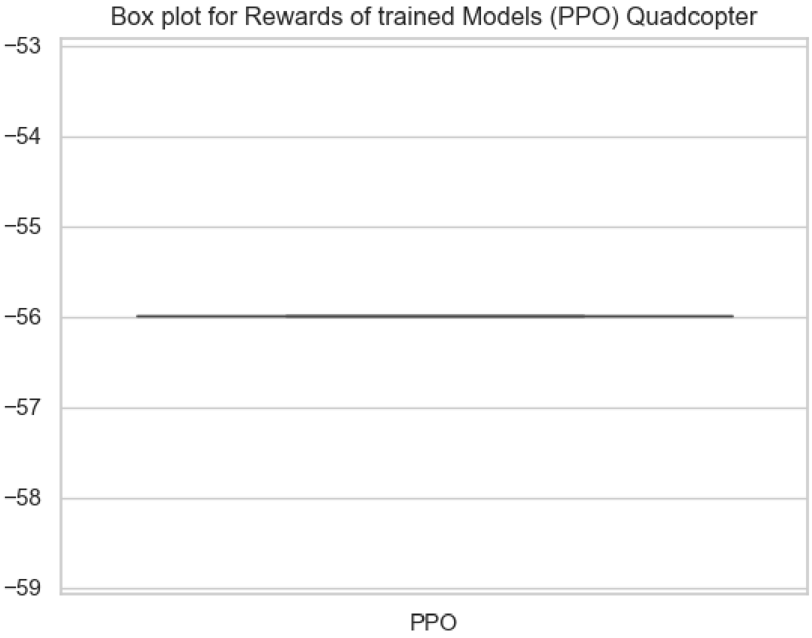


Figure 28: Box Plot of PPO Quadcopter, Reward 5

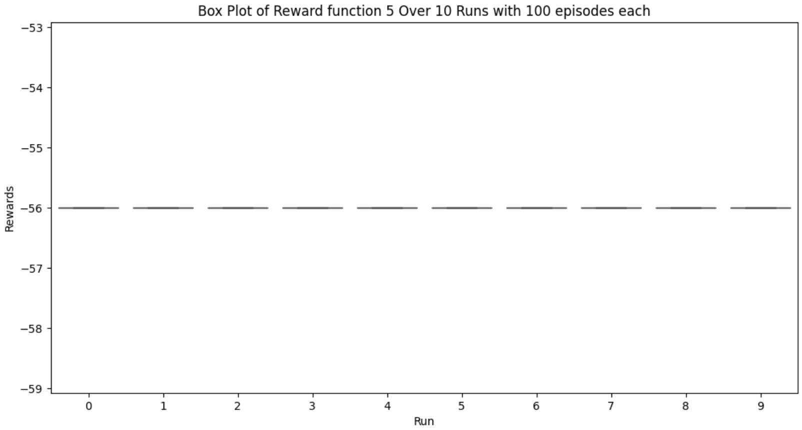


Figure 29: Box Plot of Reward function 5 Over 10 Runs with 100 episodes each

5.1.7 Reward Function 6

This function uses a continuous penalty based on the absolute deviation from the target height:

```
reward = -abs(new_y - 300) / 10
if self.step_counter > 500 or new_y > 590 or new_y < 100:
done = True
```

The reward is calculated as the negative absolute difference between the current height and 300 units, divided by 10. The maximum and minimum of the reward interval are within the range [10-30] This function penalizes the agent more as the deviation from the target height increases, encouraging precise height control.

This reward function, which imposes a continuous penalty based on the absolute deviation from the target height, provides the agent with a continuous gradient reward, unlike the previous piecewise-defined reward functions. The quadcopter manages to hover, and the box plot (Figure 33) shows that the mean reward is -25. This indicates that while the reward is zero when the quadcopter hovers at 300 units, we are still slightly off from our target.

(Figure 30) shows the mean episode length during the training of the model and (Figure 31) shows the mean episode rewards during the training of the model (Figure 32, 33, 34) shows the testing of the best reward function against the standardized test reward function (eq 1).

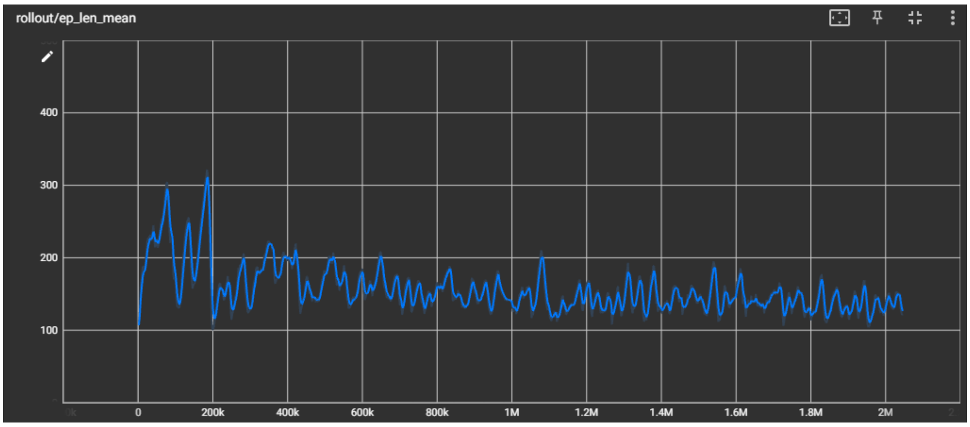


Figure 30: Reward 6 (mean ep length training)

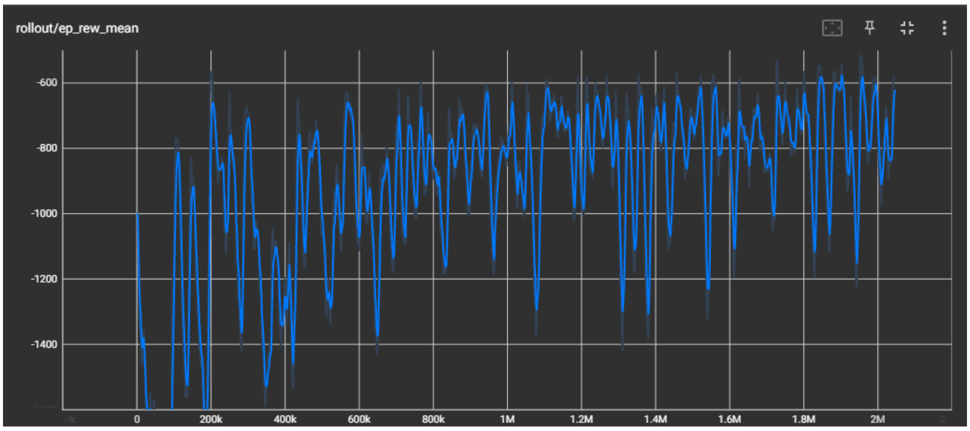


Figure 31: Reward 6 (mean ep reward training)

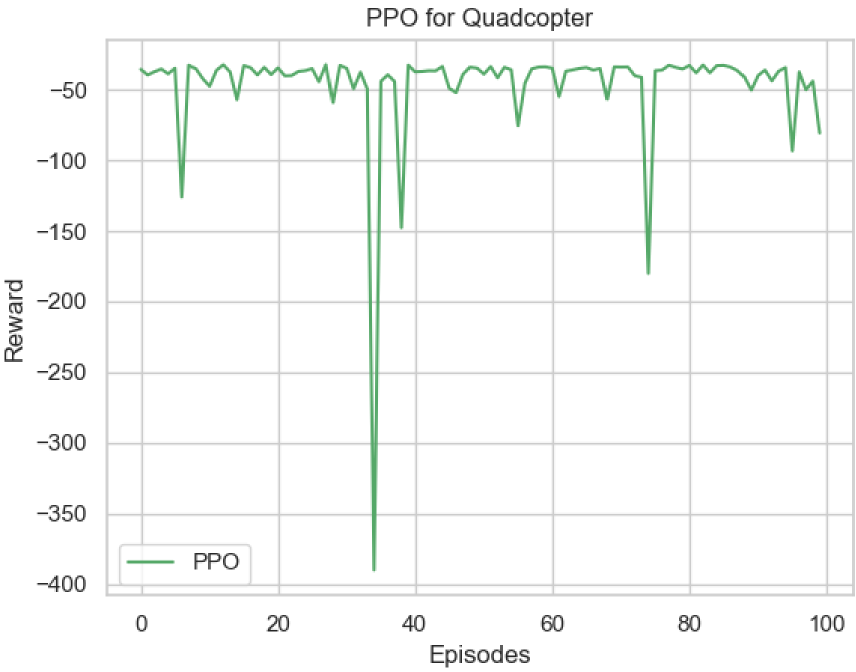


Figure 32: Testing Reward 6

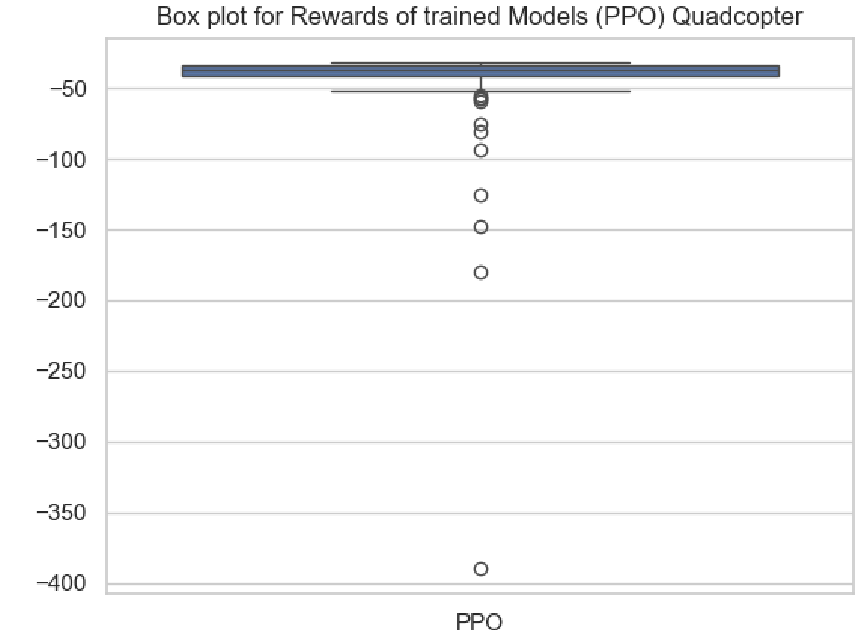


Figure 33: Box Plot of PPO Quadcopter, Reward 6

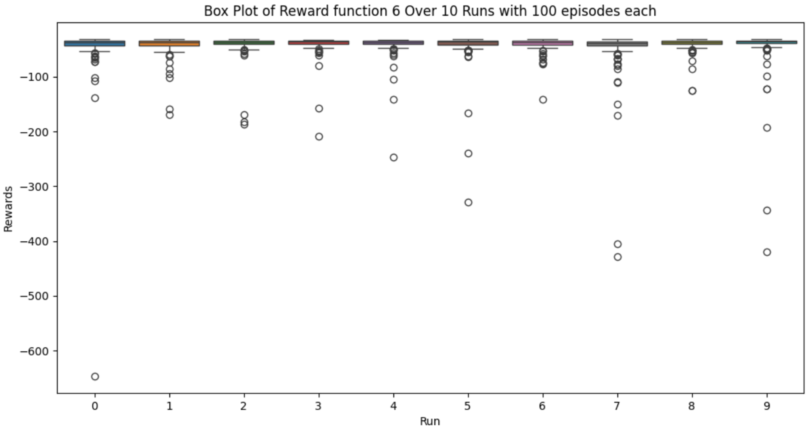


Figure 34: Box Plot of Reward function 6 Over 10 Runs with 100 episodes each

5.1.8 Reward Function 6.1

```
reward = -abs(new_y - 300) / 10
if self.step_counter > 500:
done = True
```

This reward function is similar to reward 6 however there is only one constraint 500 steps termination. The mean reward drops (figure 38) one reason for this could be that the gradient rewards could give a reward signal at each step and the huge state space could result in being stuck in state space which could be very far away from the target however as can be seen in (figure 36.), as training proceeded, the model improved. The key takeaway was that for continuous gradient the small episode/state space could make the PPO converge very quickly.

(Figure 35) shows the mean episode length during the training of the model and (Figure 36) shows the mean episode rewards during the training of the model (Figure 37, 38, 39) shows the testing of the best reward function against the standardized test reward function (eq 1).

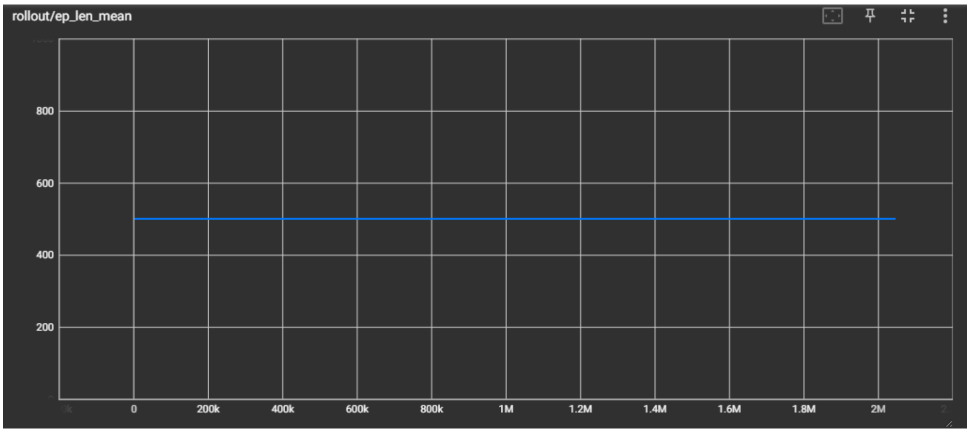


Figure 35: Reward 6.1 (mean ep length training)

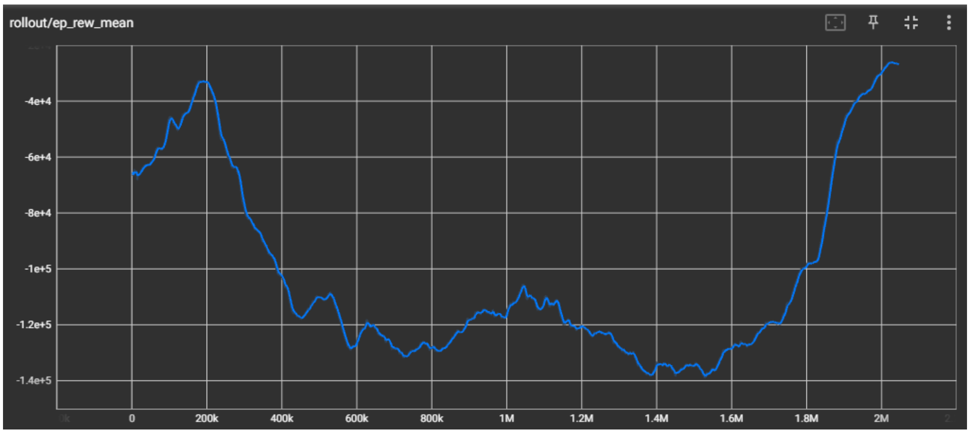


Figure 36: Reward 6.1 (mean ep reward training)

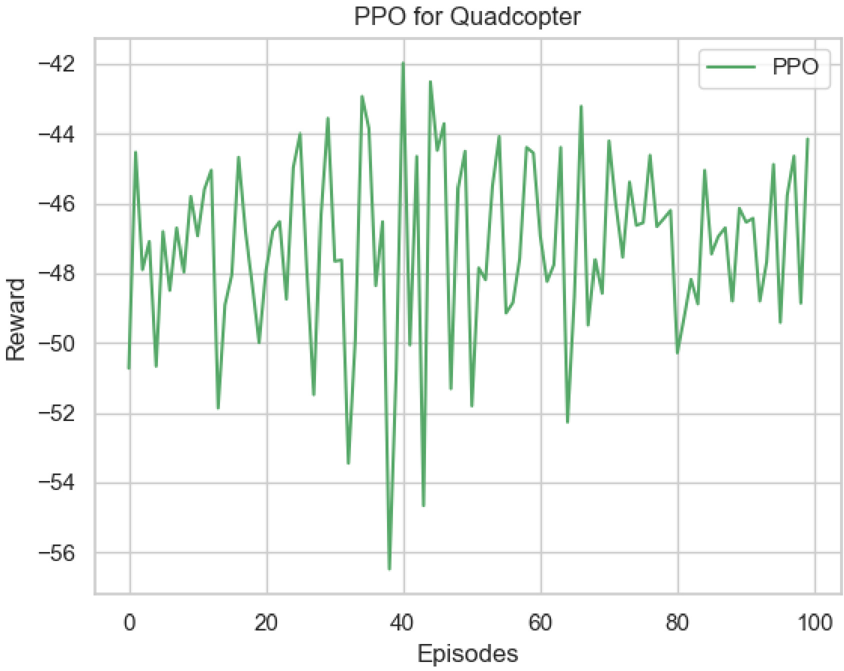


Figure 37: Testing Reward 6.1

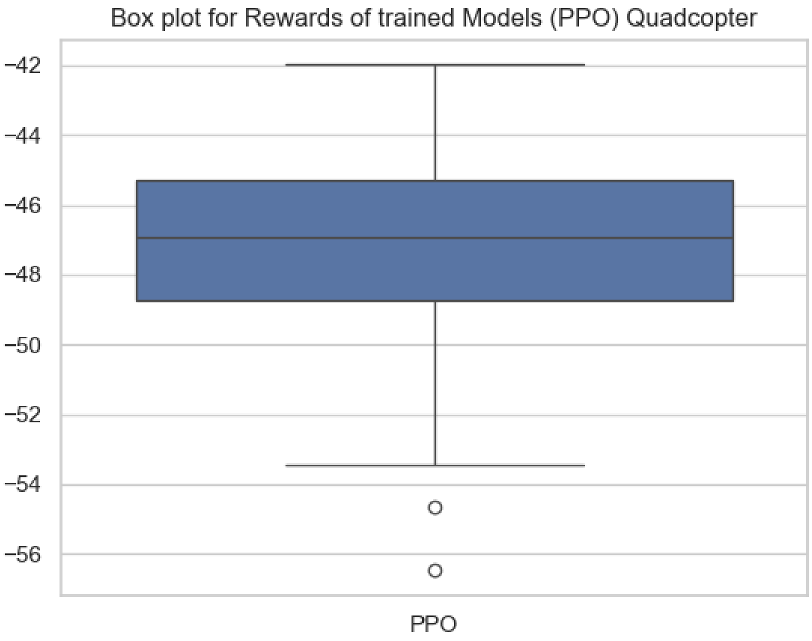


Figure 38: Box Plot of PPO Quadcopter, Reward 6.1

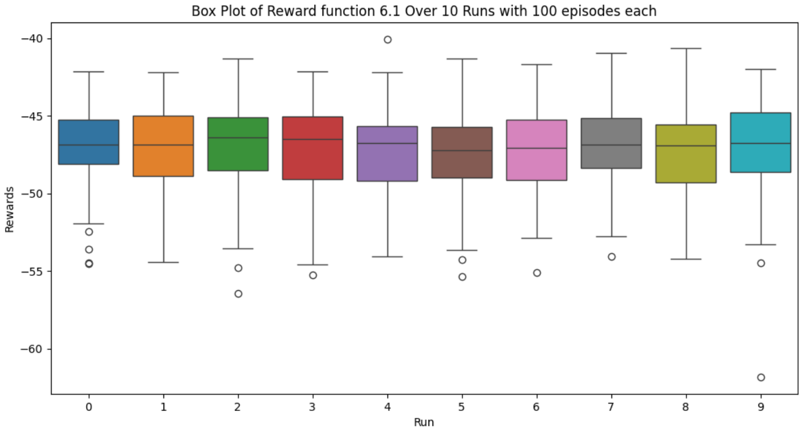


Figure 39: Box Plot of Reward function 6.1 Over 10 Runs with 100 episodes each

5.1.9 Reward Function 7

```
reward = -abs(new_y - 300) / 100
if self.step_counter > 500 or new_y > 590 or new_y < 100:
done = True
```

This function provides a normalized penalty based on the deviation from the target height. The reward is calculated as the negative absolute difference between the current height and the target height, divided by 100. This means the maximum and minimum of the reward interval are within the range [1-3].

In designing this reward function, I aimed to incorporate and synthesize all the observations and insights that I gathered from extensively testing and analyzing the previous reward functions. By doing so, I sought to create a more effective and responsive reward system that would address the shortcomings identified in earlier models and better guide the quadcopter toward achieving its target behavior.

The two key observations were:

- **Scaling the reward**
- **Continuous Gradient reward**

This reward function represents a significant leap forward in terms of improvement. The quadcopter successfully began hovering, and the box plot of the best model (Figure 43) illustrates that the mean reward has started to converge towards zero. This indicates that the physics model now allows the quadcopter to hover stably without prematurely terminating the episode.

(Figure 40) shows the mean episode length during the training of the model and (Figure 41) shows the mean episode rewards during the training of the model (Figure 42, 43, 44) shows the testing of the best reward function against the standardized test reward function (eq 1).

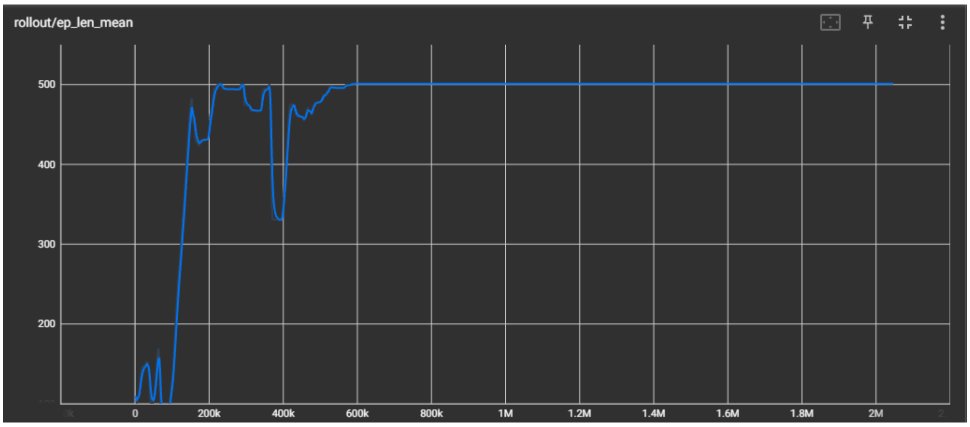


Figure 40: Reward 7 (mean ep length training)

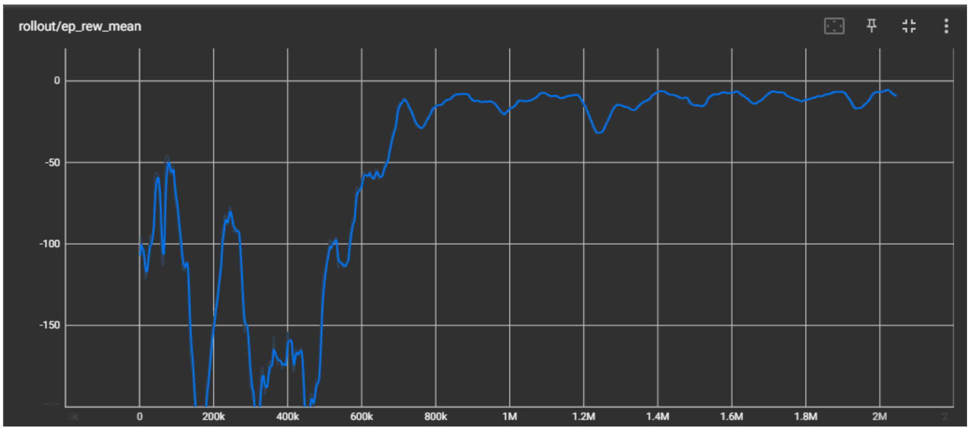


Figure 41: Reward 7 (mean ep reward training)

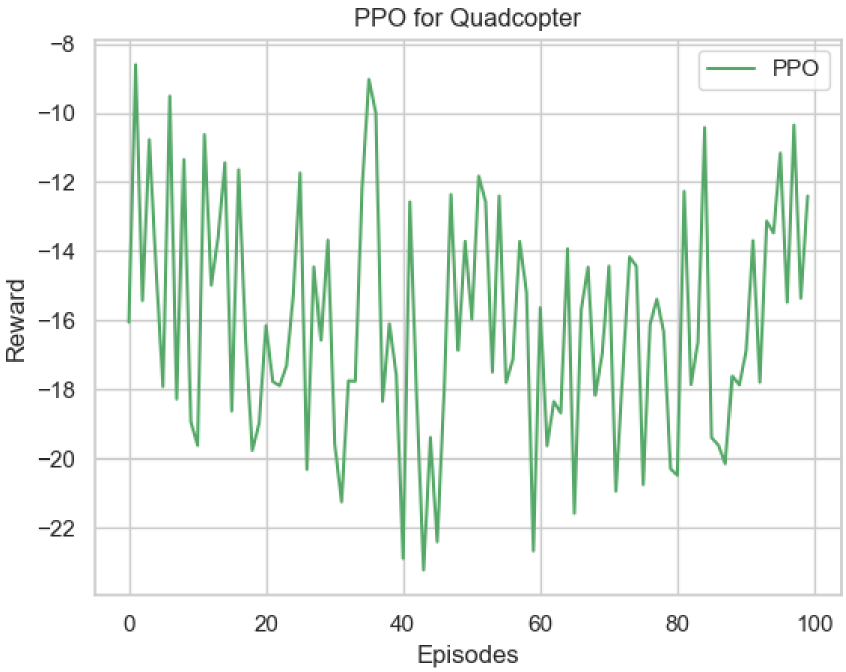


Figure 42: Testing Reward 7

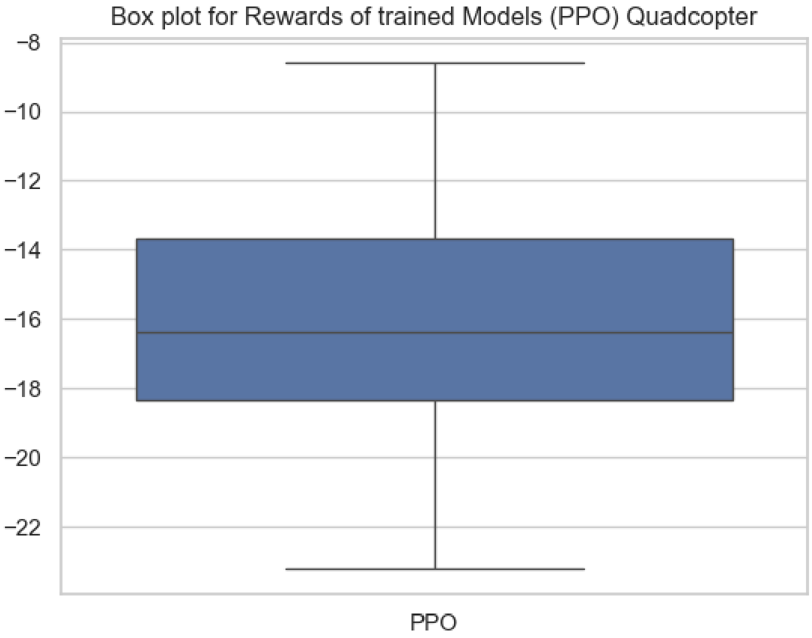


Figure 43: Box Plot of PPO Quadcopter, Reward 7

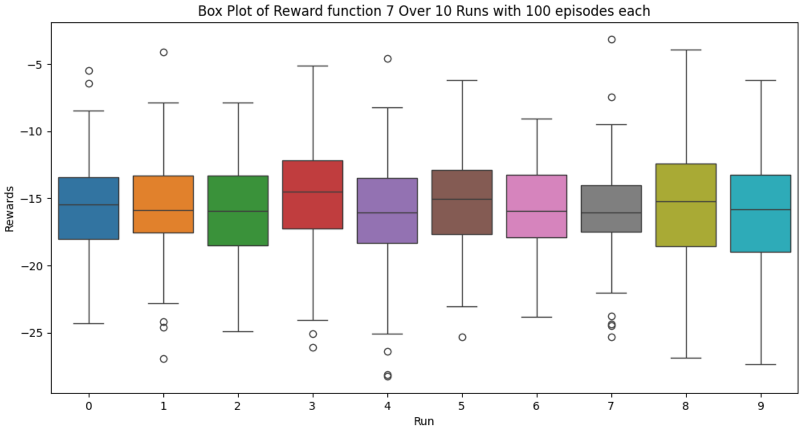


Figure 44: Box Plot of Reward function 7 Over 10 Runs with 100 episodes each

5.1.10 Reward Function 7.1

```
reward = -abs(new_y - 300) / 100
if self.step_counter > 500:
done = True
```

This Reward function is similar to reward 7 however without the boundary constraints. The key reason to test this was to see if the scaling works for unbounded states however when the best model was loaded it underperforms and does only lift maneuver.

(Figure 45) shows the mean episode length during the training of the model and (Figure 46) shows the mean episode rewards during the training of the model (Figure 47, 48 49) shows the testing of the best reward function against the standardized test reward function (eq 1).

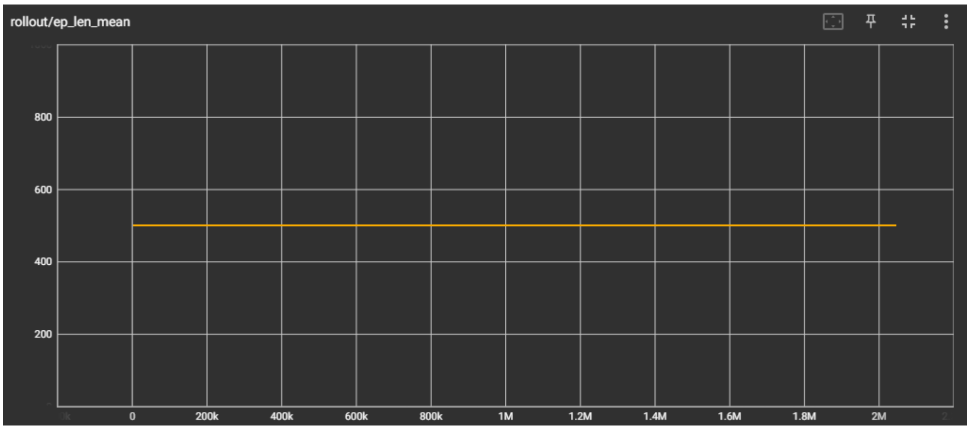


Figure 45: Reward 7.1 (mean ep length training)

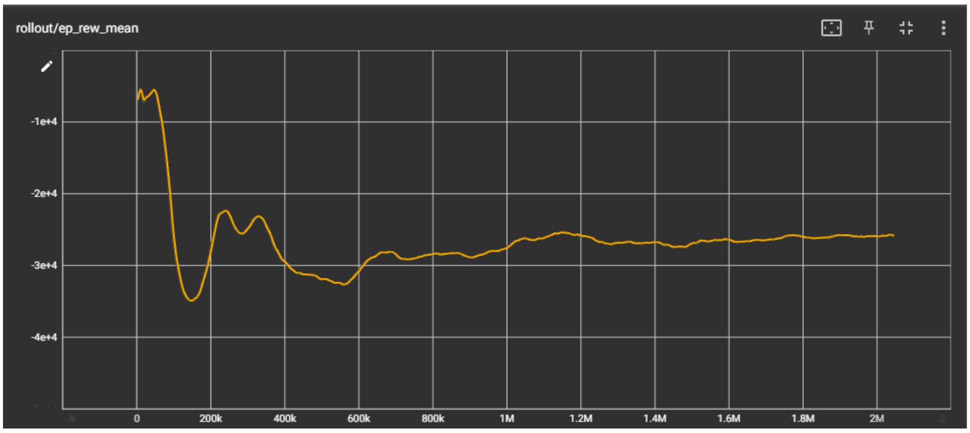


Figure 46: Reward 7.1 (mean ep reward training)

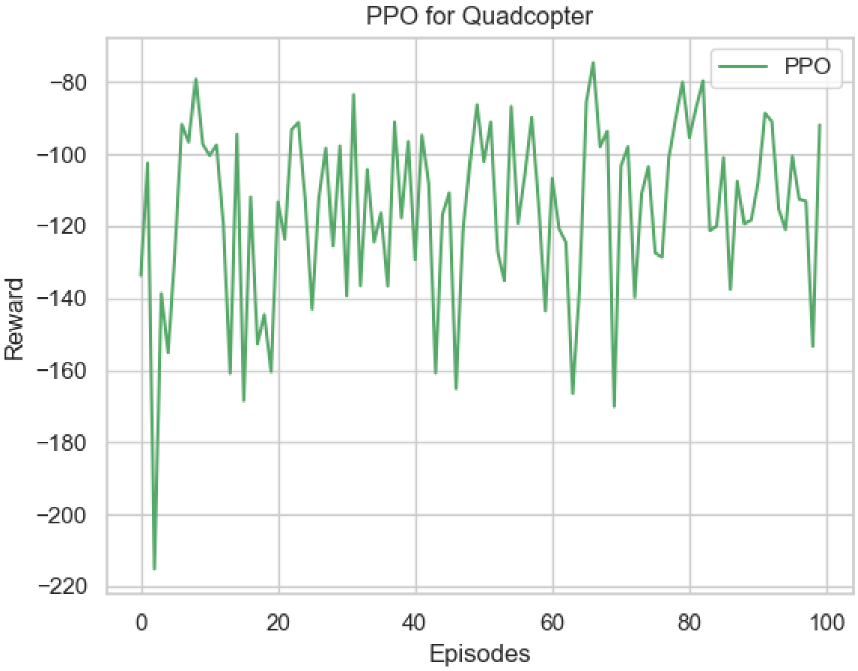


Figure 47: Testing Reward 7.1

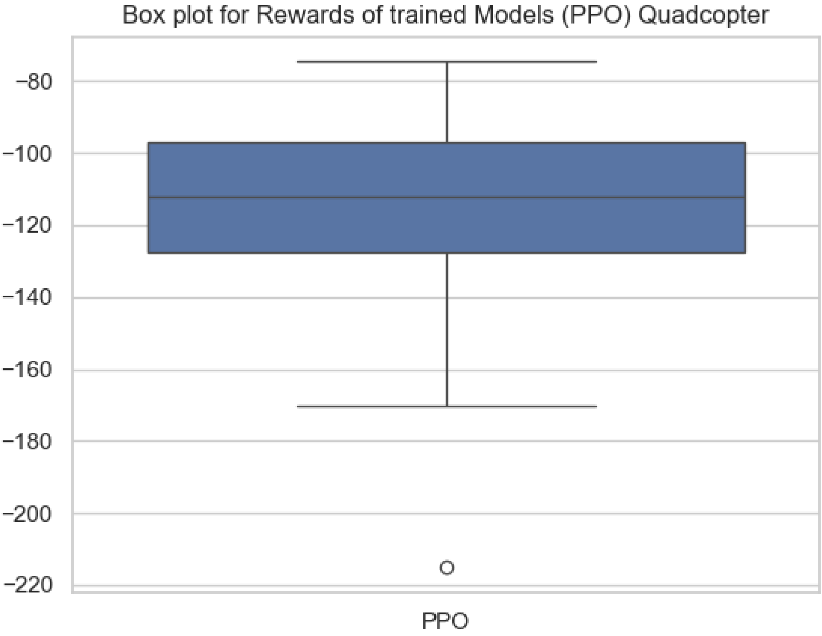


Figure 48: Box Plot of PPO Quadcopter, Reward 7.1

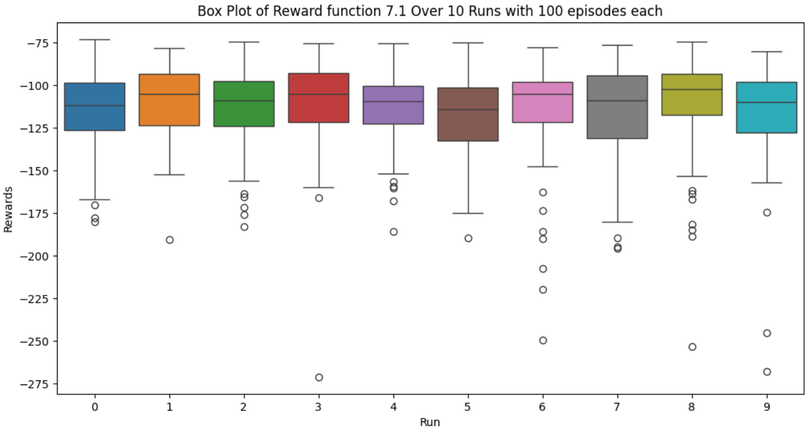


Figure 49: Box Plot of Reward function 7.1 Over 10 Runs with 100 episodes each

5.1.11 Reward Function 8

```
reward = -abs(new_y - 300) / 1000
if self.step_counter > 500 or new_y > 590 or new_y < 100:
done = True
```

This reward function is similar to reward 7 however this tries to experiment with how much we can scale the reward function. It divides that abs reward function by 1000. The model still performs well however when you load the best model and plot the box plot we can see that the mean reward is -19 compared to reward 7 which is around -16. The difference is not much but sufficient to conclude that further scaling results in underperformance compared to reward function 7.

(Figure 50) shows the mean episode length during the training of the model and (Figure 51) shows the mean episode rewards during the training of the model (Figure 52, 53, 54) shows the testing of the best reward function against the standardized test reward function (eq 1).

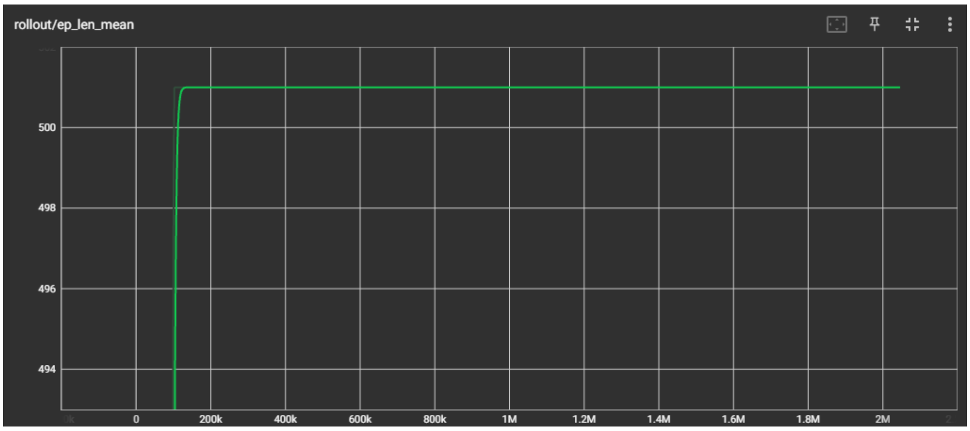


Figure 50: Reward 8 (mean ep length training)

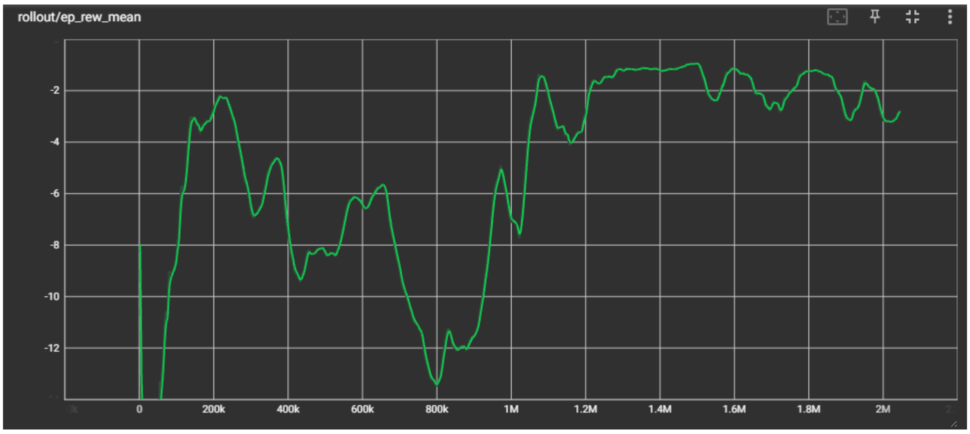


Figure 51: Reward 8 (mean ep reward training)

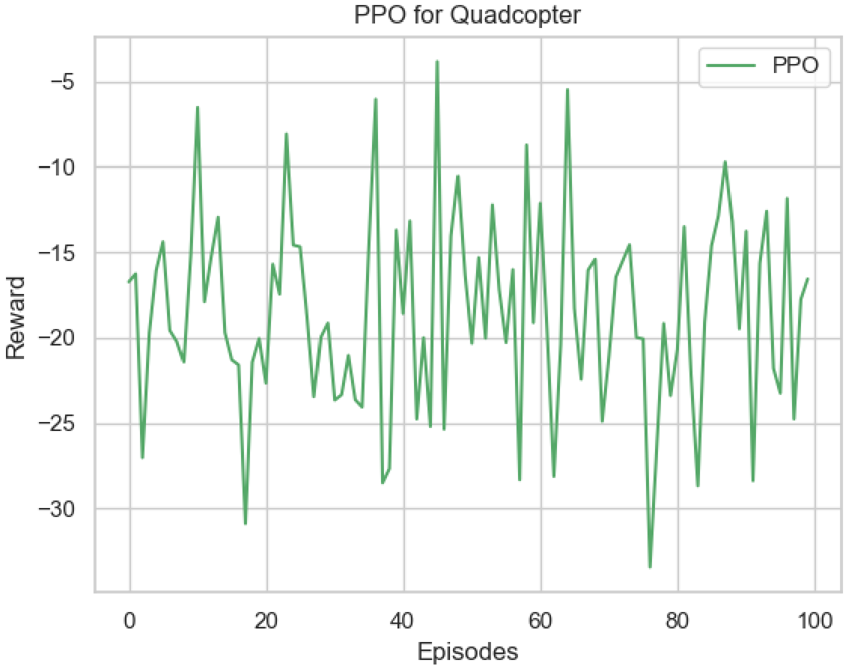


Figure 52: Testing Reward 8

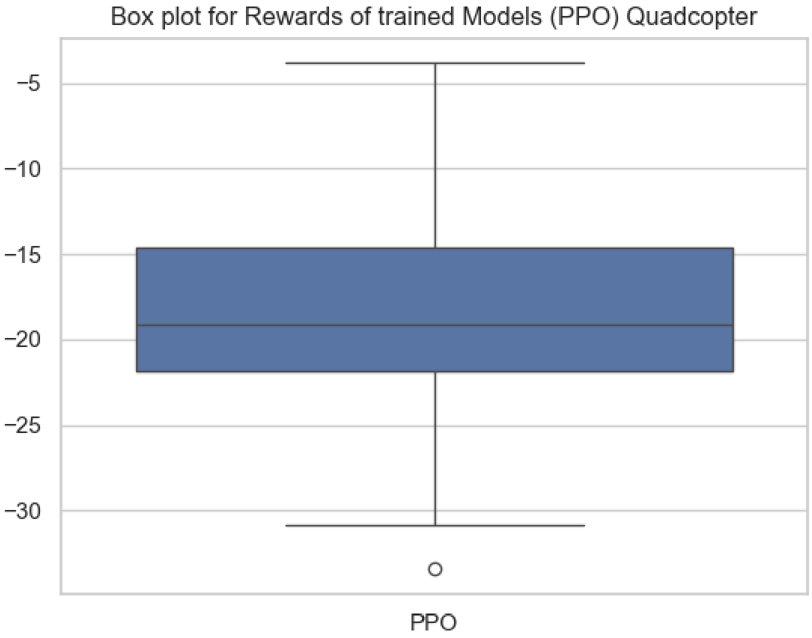


Figure 53: Box Plot of PPO Quadcopter, Reward 8

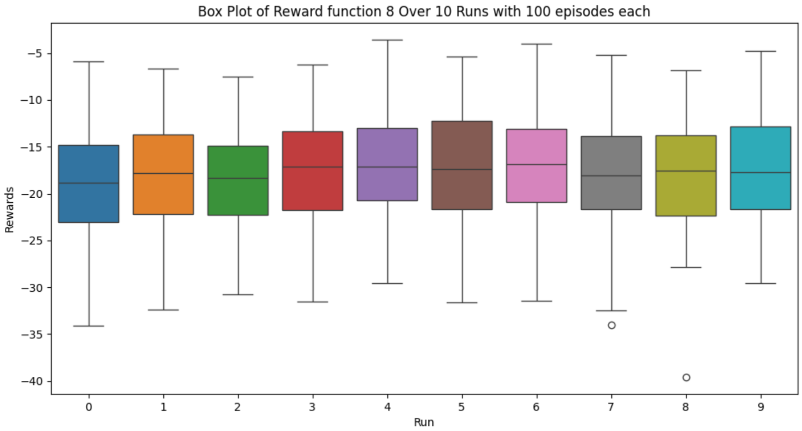


Figure 54: Box Plot of Reward function 8 Over 10 Runs with 100 episodes each

5.1.12 Reward Function 9

Reward 9 is similar to reward 8 however the action space is reduced to two actions (Up and No action) to check if the performance improves. The box plot shows that the mean reward for the best model for reward 9 is -11 compared to reward 8 which is -19.

At this point we have some solid conclusions and we can use them to design the refined reward function that can produce a tighter control.

(Figure 55) shows the mean episode length during the training of the model and (Figure 56) shows the mean episode rewards during the training of the model (Figure 57, 58, 59) shows the testing of the best reward function against the standardized test reward function (eq 1).

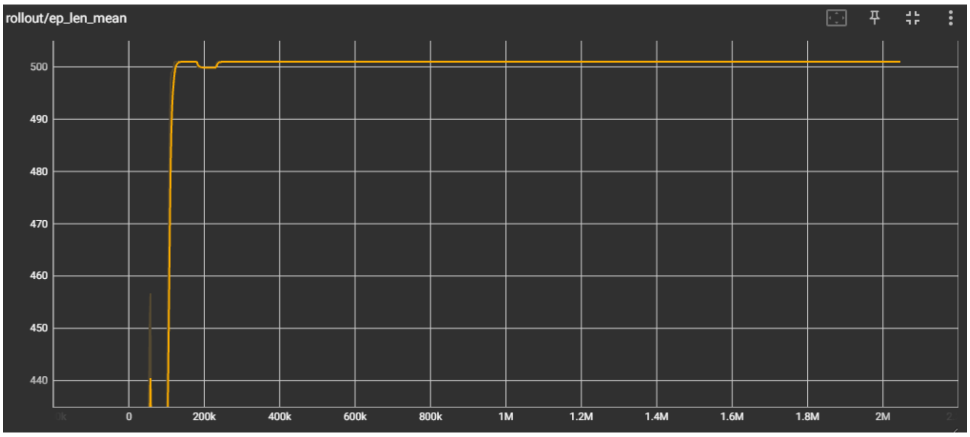


Figure 55: Reward 9 (mean ep length training)

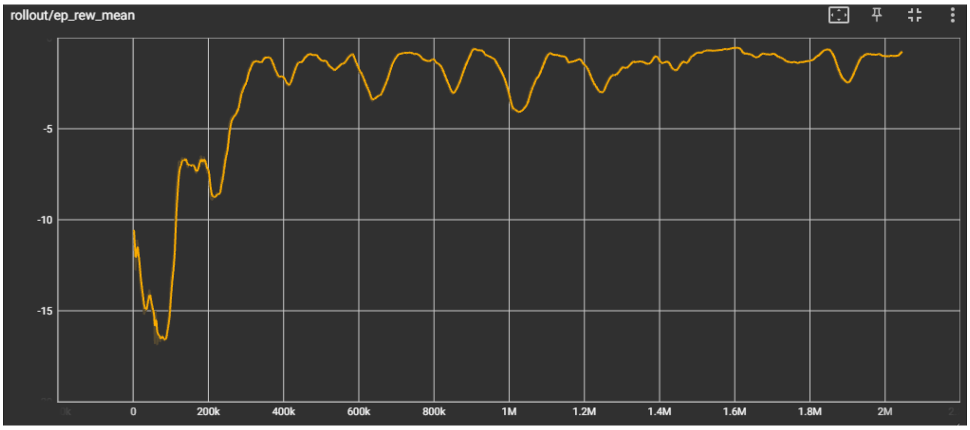


Figure 56: Reward 9 (mean ep reward training)

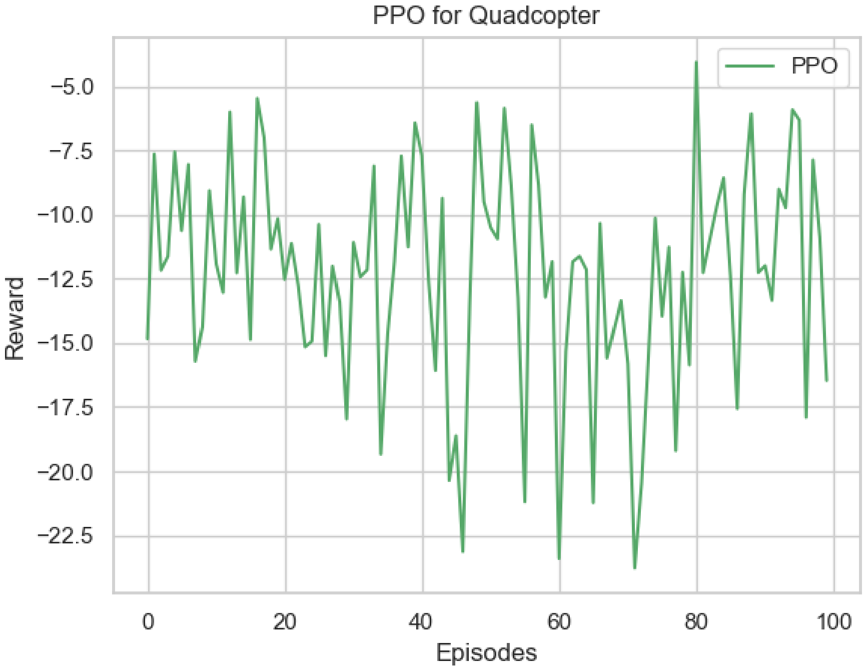


Figure 57: Testing Reward 9

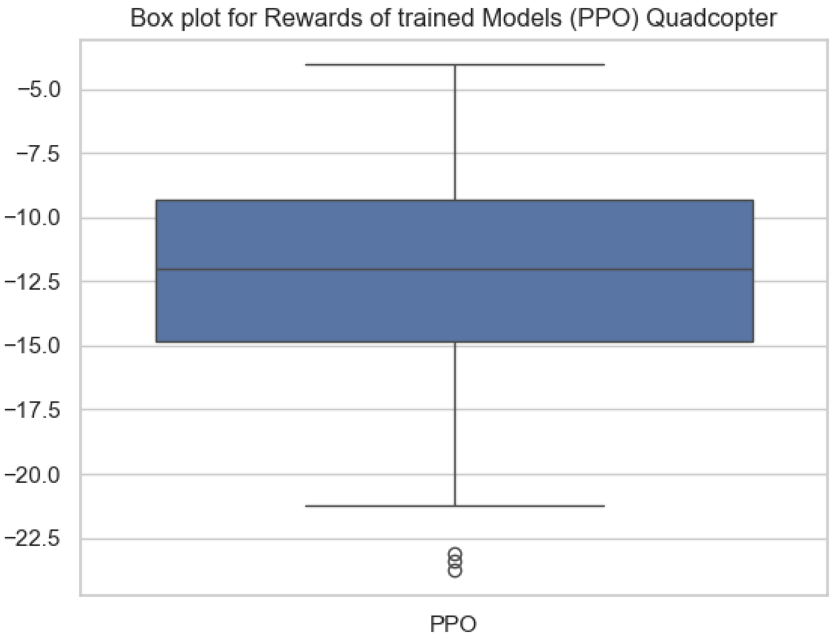


Figure 58: Box Plot of PPO Quadcopter, Reward 9

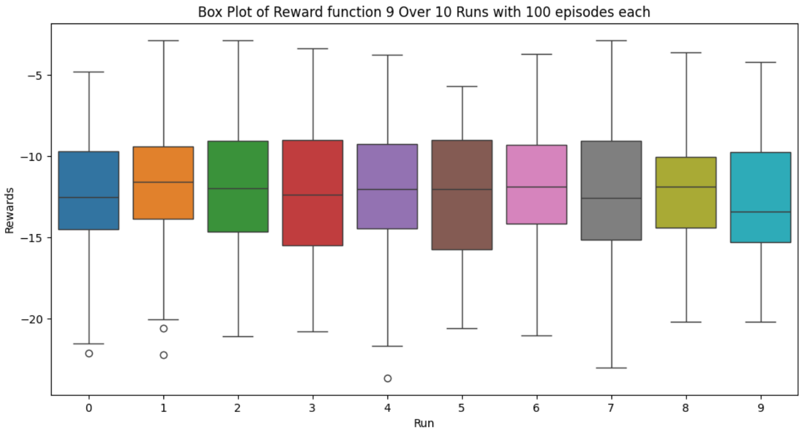


Figure 59: Box Plot of Reward function 9 Over 10 Runs with 100 episodes each

5.1.13 Best Reward function: Reward Function 10

Reward function 10 is the best reward function where we did the accurate scaling, used the continuous reward function with the bounds, and used the two actions up and No action.

we have a very tight control the training curve approaches zero signaling the agent is hovering around mid and testing the best model we get a box plot where the mean reward is around -3 and the quadcopter holds tight around 299.1-300 units.

Since this was the best reward function I trained it on GPU on Google Colab and plotted the training curves against each other. we can the model trained on Colab outperforms the model trained on PC. The Blue line is Google Colab and green is PC.

(Figure 60) shows the mean episode length during the training of the model and (Figure 61) shows the mean episode rewards during the training of the model (Figure 62, 63, 64) shows the testing of the best reward function against the standardized test reward function (eq 1).

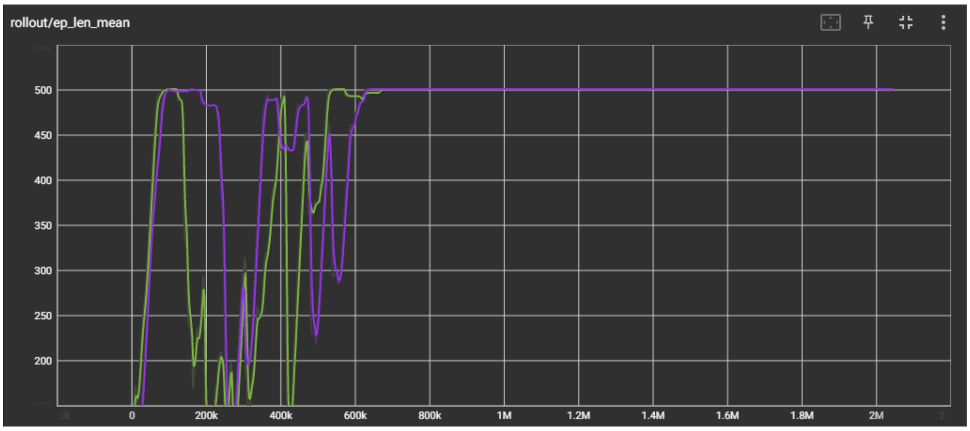


Figure 60: Reward 10 (mean ep length training)

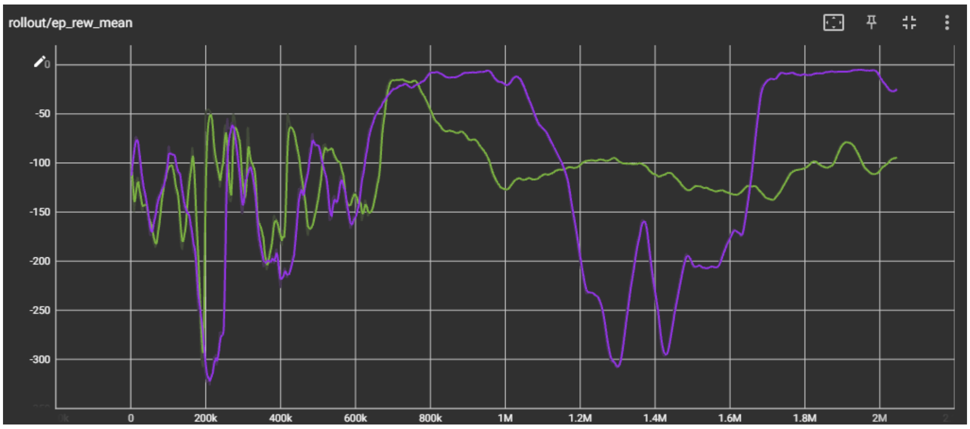
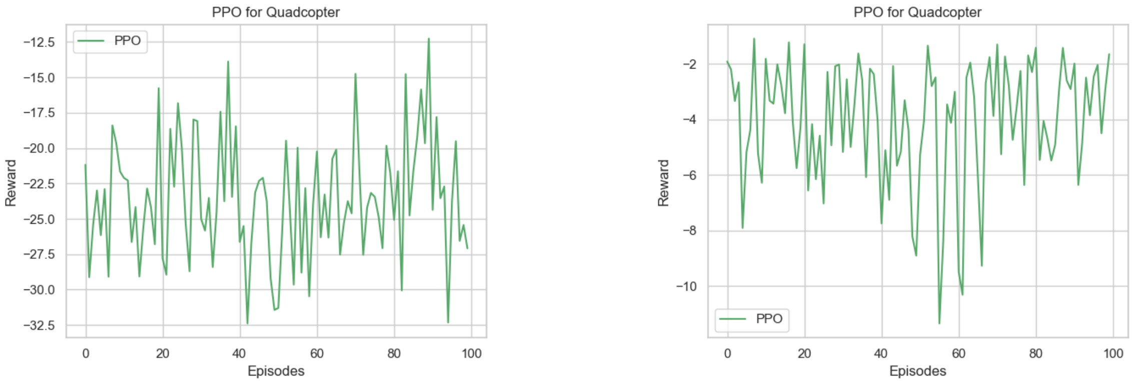


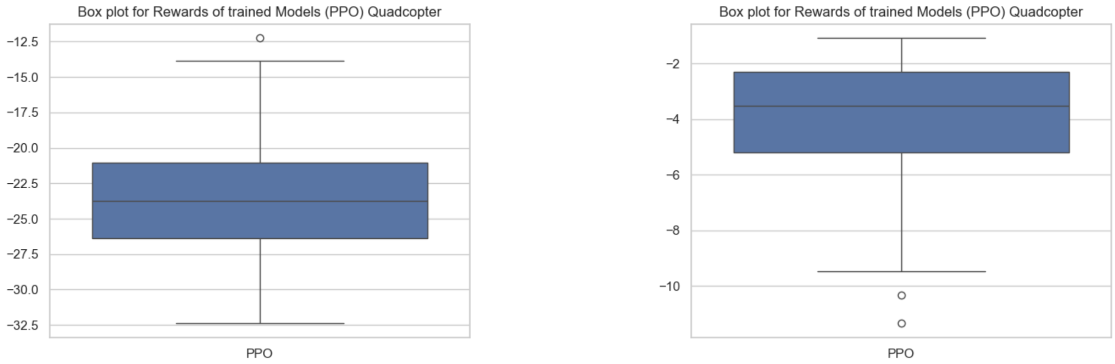
Figure 61: Reward 10 (mean ep reward training)



(a) Testing Reward 10

(b) Testing Reward 10 Google Colab

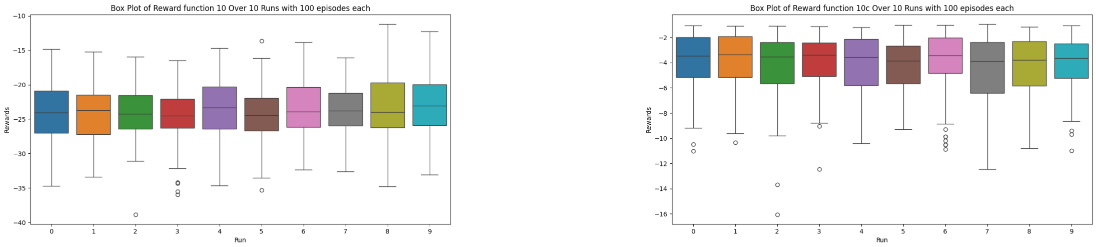
Figure 62: Comparison of reward 10 PC vs Google colab



(a) Box Plot of PPO Quadcopter, Reward 10

(b) Box Plot of PPO Quadcopter, Reward 10 Google Colab

Figure 63: Comparison of Box Plots for Reward 10 PC vs Google Colab



(a) Box Plot of PPO Quadcopter, Reward 10 (10 Runs, 100 Episodes each)

(b) Box Plot of PPO Quadcopter, Reward 10 Google Colab (10 Runs, 100 Episodes each)

Figure 64: Comparison of Box Plots for Reward 10 PC vs Google Colab (10 Runs, 100 Episodes each)

5.2 Evolution of Reward Function

To visually observe the changes in the reward function, I plotted the reward function for values of the vertical axis y ranging from 0 to 600, corresponding to the screen limits of the physics engine. This visualization helps to better understand the effects of the modifications made to the reward function and how these changes influenced the quadcopter's behavior.

As shown, Reward Function 1 is discontinuous and does not provide a gradient reward. Reward Function 2 is also discontinuous but more spread out, while Reward Function 3 remains discontinuous but is scaled down, which aided in learning the desired behavior. Reward Functions 6, 7, and 8 are continuous gradients, and the effects of scaling can be observed.

The final (Figure 70) presents the test runs of all reward function models across 10 batches of 100 episodes each, plotted against each other. We can observe an increasing trend in the mean reward as we apply the lessons learned from previous reward functions, refining the new ones to better achieve the desired objective.

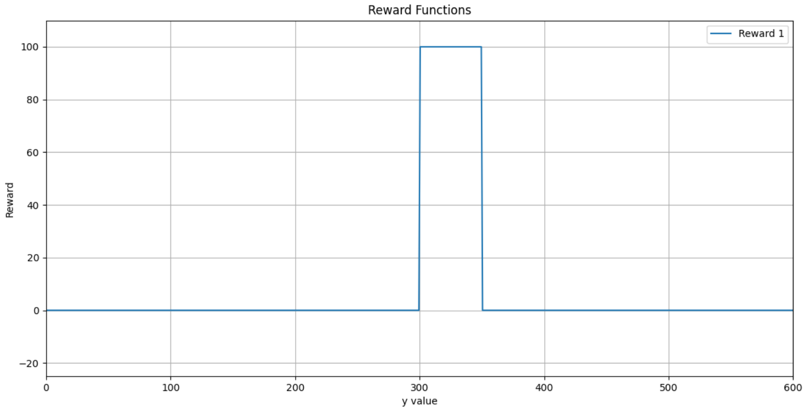


Figure 65: Plot of Reward Function 1

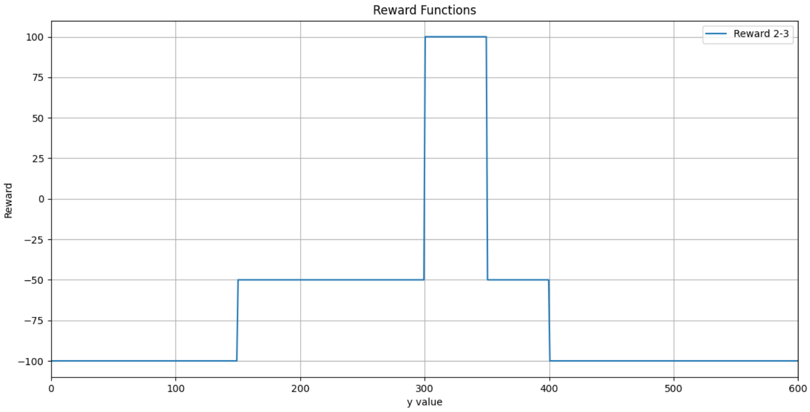


Figure 66: Plot of Reward Functions (2-3)

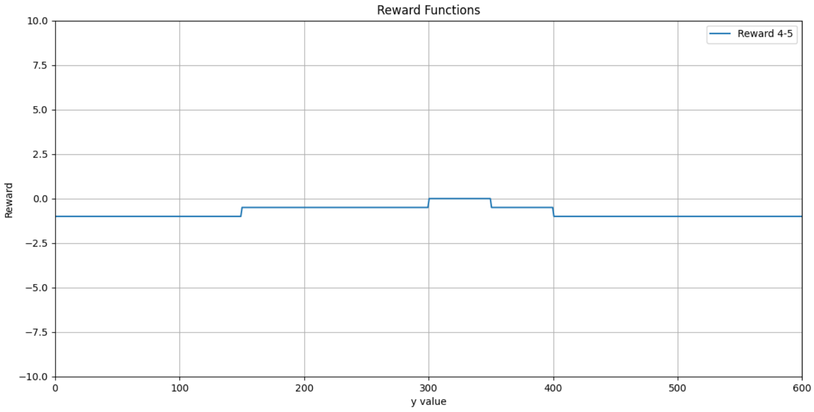


Figure 67: Plot of Reward Functions (4-5)

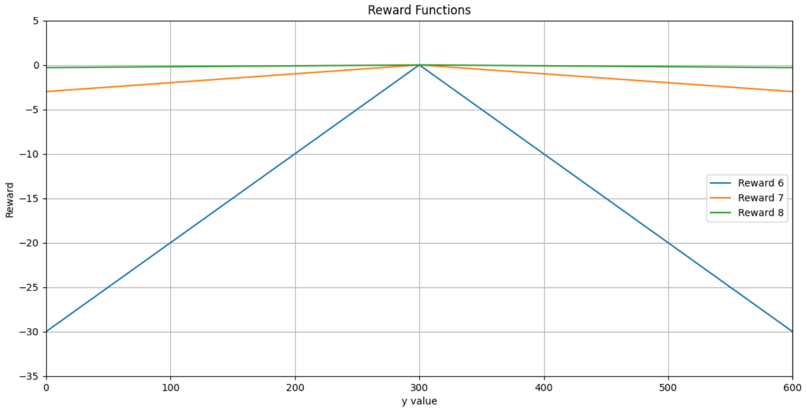


Figure 68: Plot of Reward Functions (6,7,8)

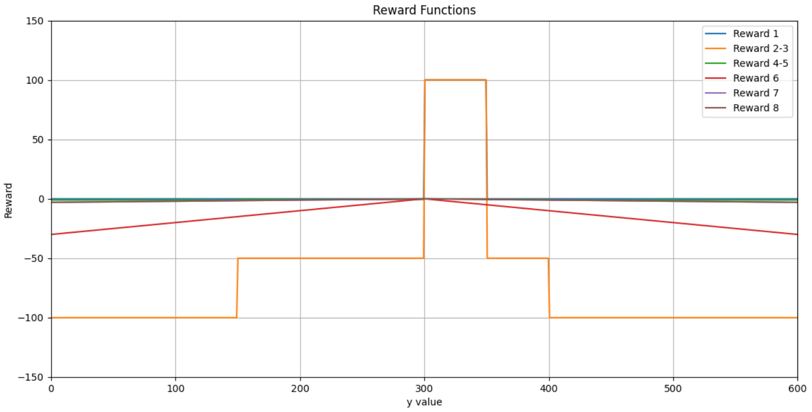


Figure 69: Plot of all Reward Functions

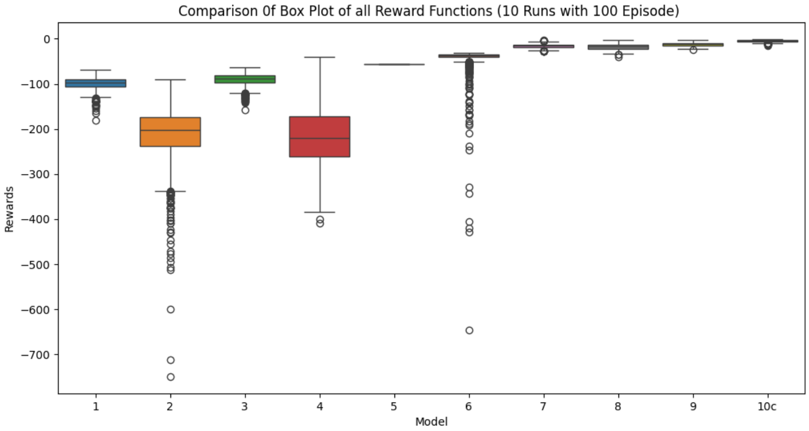


Figure 70: Box Plot of all Reward Functions (10 runs with 100 episodes each)

6 Conclusion

This thesis explored the development and refinement of reward functions to optimize the performance of a quadcopter in a reinforcement learning framework. Throughout the research, various reward functions were tested and analyzed to achieve stable hovering behavior at a target altitude.

Initially, simple reward functions led the agent to suboptimal strategies, such as terminating the episode prematurely to avoid penalties. As the reward functions evolved, incorporating elements like gradient rewards, boundary constraints, and scaled rewards, significant improvements were observed. Notably, Reward Function 7 introduced a more sophisticated approach by synthesizing insights from previous models, leading to more consistent hovering behavior.

However, further refinements, including Reward Functions 8 and 9, demonstrated the limitations of scaling rewards and reducing the action space. These adjustments provided valuable insights but also highlighted the delicate balance required in reward engineering to avoid underperformance.

The final reward function, Reward 10, successfully combined accurate scaling, continuous rewards, and constrained actions, resulting in tight control and stable hovering at the target altitude. The results show that the agent was able to maintain its position close to 300 units with minimal deviation, indicating the effectiveness of the designed reward system.

This research underscores the critical role of reward engineering in reinforcement learning, particularly in tasks requiring precise control, such as quadcopter stabilization. By systematically evaluating and refining reward functions, this thesis contributes to the broader understanding of how to design effective reward systems that guide agents towards optimal performance in complex, real-world environments. The findings have implications for future work in autonomous systems and advanced control strategies using reinforcement learning.

References

[1] Fadi AlMahamid and Katarina Grolinger. Autonomous unmanned aerial vehicle navigation using reinforcement learning: A systematic review. *Engineering Applications of Artificial Intelligence*, 115:105321, 2022.

[2] Ahmad Taher Azar, Anis Koubaa, Nada Ali Mohamed, Habiba A Ibrahim, Zahra Fathy Ibrahim, Muhammad Kazim, Adel Ammar, Bilel Benjdira, Alaa M Khamis, Ibrahim A Hameed, et al. Drone deep reinforcement learning: A review. *Electronics*, 10(9):999, 2021.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[4] Khalil Chikhaoui, Hakim Ghazzai, and Yehia Massoud. Ppo-based reinforcement learning for uav navigation in urban environments. In *2022 IEEE 65th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, 2022.

[5] Daniel Dewey. Reinforcement learning and the reward engineering principle. In *2014 AAAI Spring Symposium Series*, 2014.

[6] Michael Fairbank. Ce811: Game artificial intelligence, 2024. Course Material, University of Essex.

[7] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Timothee Traore. Stable baselines. https://github.com/DLR-RM/stable-baselines3, 2018.

[8] PB Karthik, Keshav Kumar, Vikrant Fernandes, and Kavi Arya. Reinforcement learning for altitude hold and path planning in a quadcopter. In *2020 6th International Conference on Control, Automation and Robotics (ICCAR)*, pages 463–467. IEEE, 2020.

[9] William Koch, Renato Mancuso, Richard West, and Azer Bestavros. Reinforcement learning for uav attitude control. *ACM Transactions on Cyber-Physical Systems*, 3(2):1–21, 2019.

[10] Alex X Lee, Anusha Nagabandi, Pieter Abbeel, and Sergey Levine. Stochastic latent actor-critic: Deep reinforcement learning with a latent variable model. *Advances in Neural Information Processing Systems*, 33:741–752, 2020.

[11] OpenAI. Proximal policy optimization. Accessed: 2024-08-13.

[12] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[14] David Silver, Satinder Singh, Doina Precup, and Richard S Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, 2021.

[15] Stable Baselines3. A2c — stable baselines3 documentation, 2024. Accessed: 2024-08-20.

[16] Stable Baselines3. Ppo — stable baselines3 documentation, 2024. Accessed: 2024-08-20.

[17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[18] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[19] Philip S Thomas and Emma Brunskill. Policy gradient methods for reinforcement learning with function approximation and action-dependent baselines. *arXiv preprint arXiv:1706.06643*, 2017.

[20] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[21] Timothy D Wilson. *Strangers to ourselves: Discovering the adaptive unconscious.* Harvard University Press, 2004.

[22] Shanglin Zhou, Bingbing Li, Caiwu Ding, Lu Lu, and Caiwen Ding. An efficient deep reinforcement learning framework for uavs. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 323–328, 2020.

7 Appendix

7.1 Lunar Lander

The lunar lander environment is trained using the REINFORCE (figure 71), PPO, and A2C (figure 72). The best model for each one of them is used to plot the comparison. PPO outperforms. The box plot in (figure 74) shows the mean and Standard deviation. PPO has the smallest standard deviation and higher mean which means that PPO performs much better than the rest of them.

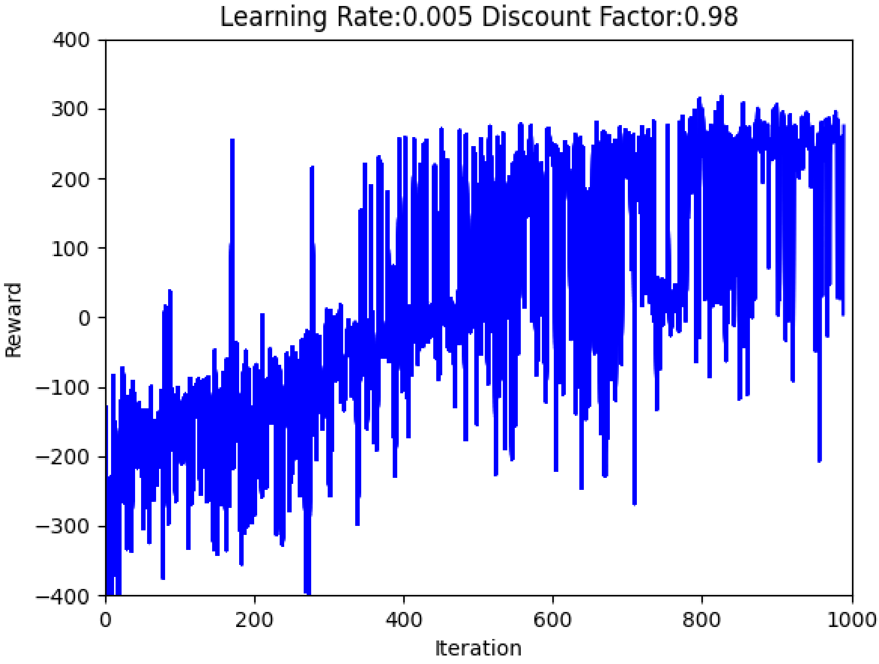


Figure 71: REINFORCE training for Lunar lander

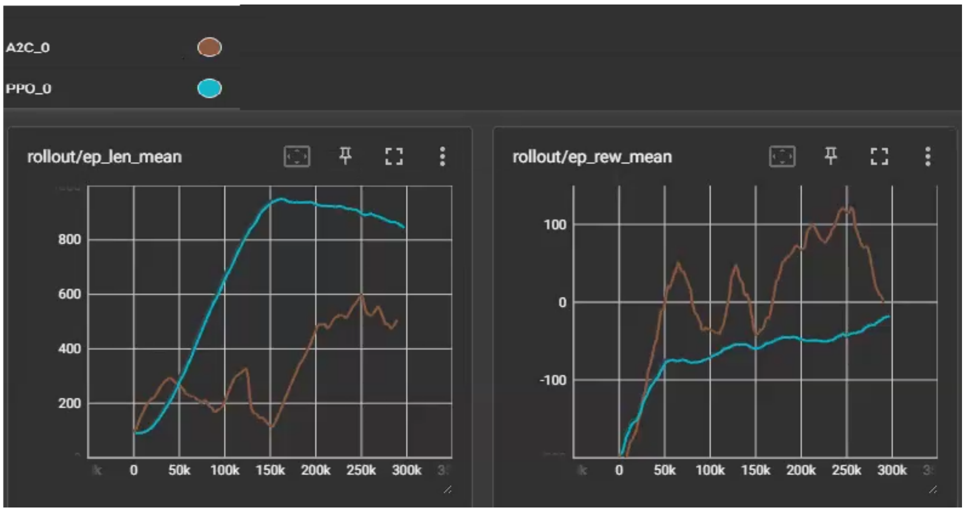


Figure 72: PPO and A2C training for Lunar lander

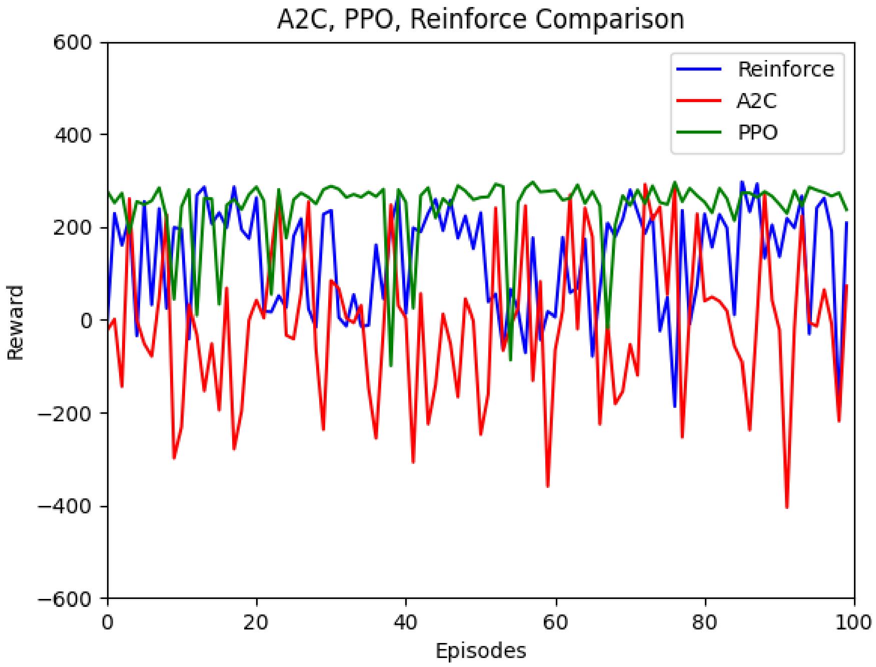


Figure 73: REINFORCE, PPO and A2C testing for Lunar lander

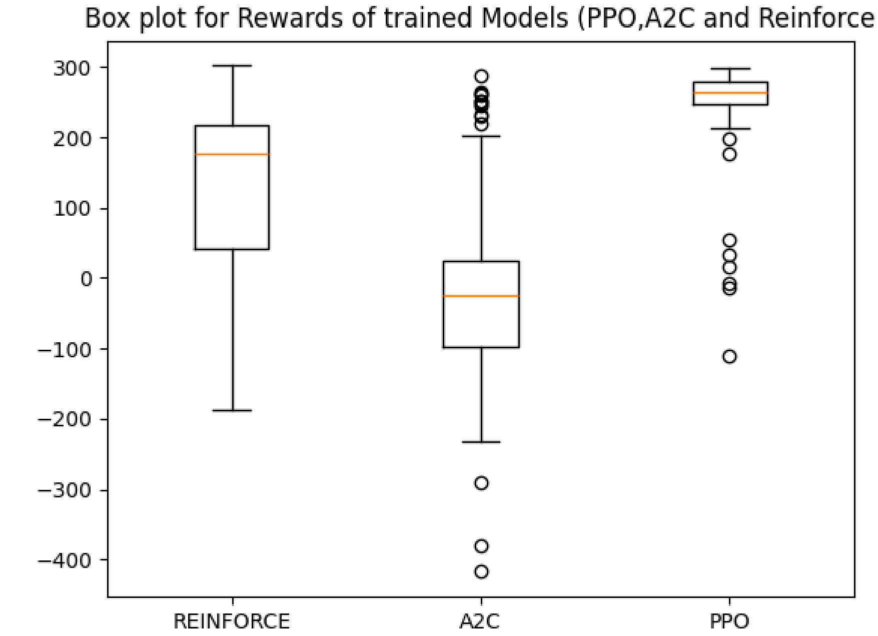


Figure 74: Boxplot: REINFORCE, PPO and A2C testing for Lunar lander

7.2 Cart Pole

REINFORCE algorithm was adopted for the cart pole environment. The REINFORCE was trained for 1000 iterations (figure 75). PPO and A2C were also trained to test against each other (figure 76) method of testing as explained in the methodology section. The best model for each one of them is used to plot the comparison. PPO and REINFORCE outperform. The box plot in (figure 78) shows the mean and Standard deviation. All of them have almost the same mean and standard deviation however A2C has more outliers as can be seen in the (figure 77.)

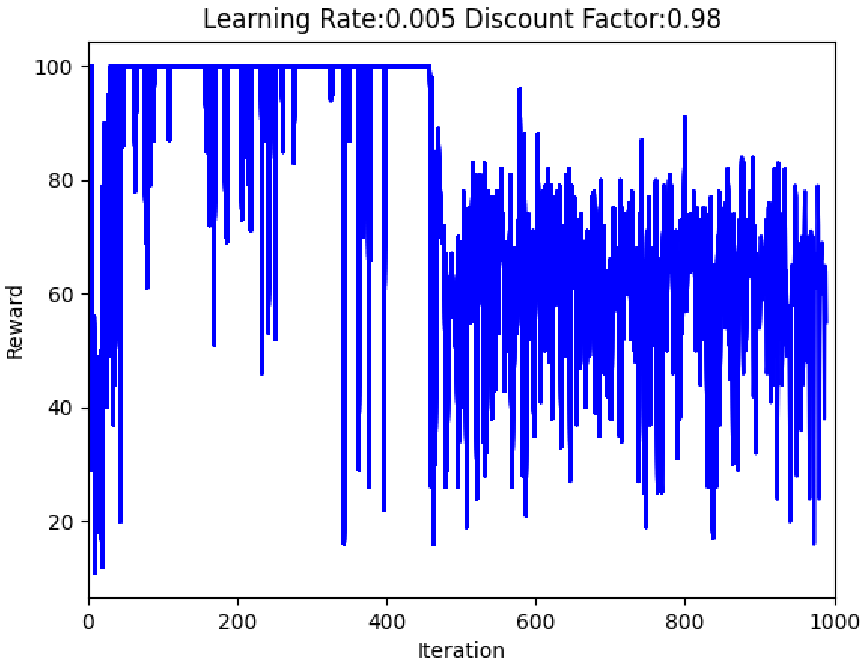


Figure 75: REINFORCE training for cartpole

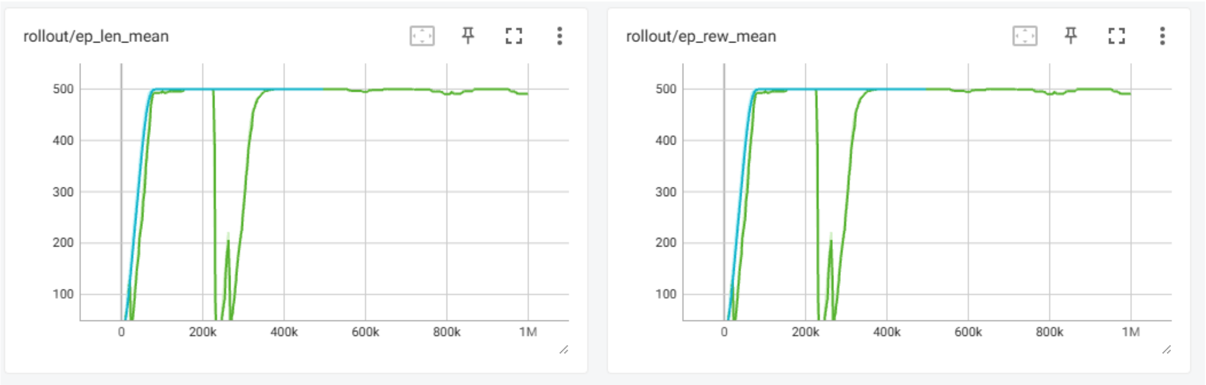


Figure 76: PPO and A2C training for CartPole. The green line represents A2C, and the blue line represents PPO.

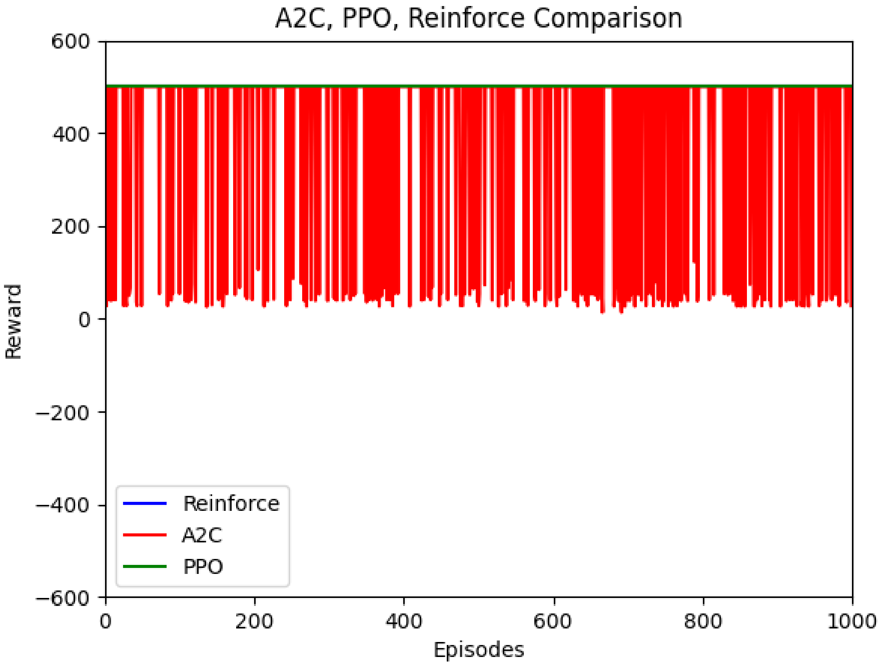


Figure 77: A2C, PPO, Reinforce Comparison for Cart Pole

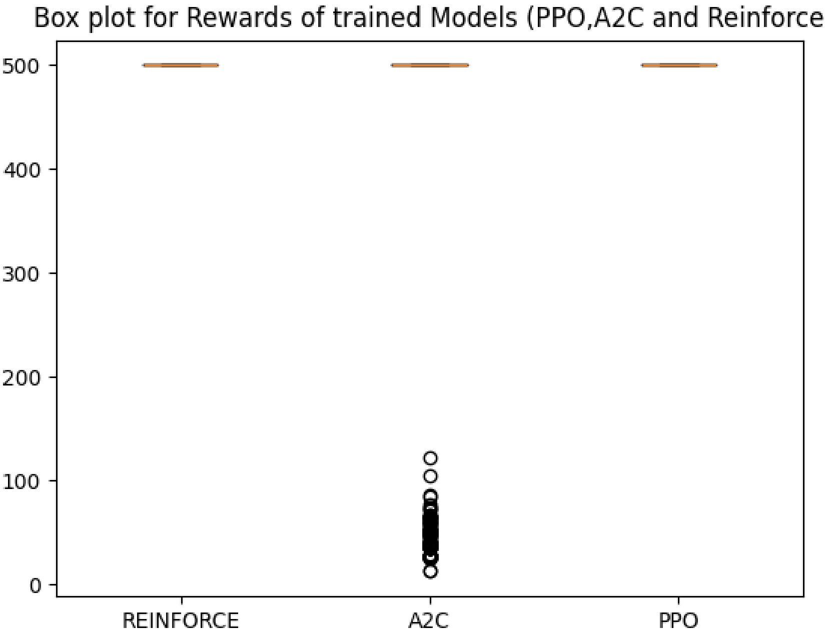


Figure 78: Boxplot: A2C, PPO, Reinforce Comparison for Cart Pole