Reinforcement Learning for Optimising Autonomous Kite-Powered Vessel Control

A Novel Approach

By

JOSHUA CAREY

Department of Computer Science
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of MASTERS OF COMPUTER SCIENCE in the Faculty of Engineering.

SEPTEMBER 2023

Abstract

The following paper explores the application of reinforcement learning (RL) techniques in autonomously controlling a kite-powered vessel. This novel approach is grounded in the growing need for sustainable maritime transportation technologies that can mitigate the environmental impact associated with traditional fuel-powered propulsion systems. The research focuses on developing a sophisticated simulation environment for training an RL agent. Utilising the MLAgents toolkit, Unity Game Engine, Python, and Proximal Policy Optimisation (PPO) algorithm, the project constructs a virtual marine environment alongside realistic models of a boat and a kite. The simulation is then used to train an RL agent to autonomously control the boat and kite with the aim of autonomously navigating a predefined course. The project successfully demonstrates the ability of an RL agent to learn the task of autonomously controlling a kite-powered vessel, however limitations in the simulation environment and the RL implementation prevent the agent from learning complex autonomous navigation. Future work should focus on improving the complexity of the simulation and the configuration of the RL, while continuing to think about the real world implementation and deployment of this system.

Dedication and Acknowledgements

This journey of exploration and learning would not have been possible without the support and inspiration from some remarkable individuals to whom I owe my deepest gratitude.

First and foremost, I extend my heartfelt thanks to Jack Lippold, whose enthusiasm for coding ignited my passion in this field.

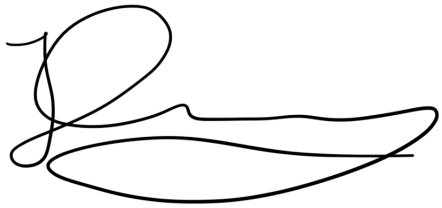
I would also like to express my sincere appreciation to James, the coffee man from Rolling Italy, who produces by far the best coffee on campus and can be found outside Senate House. James, your coffee has been the fuel that powered me through the countless hours of study and work. Thank you for being an integral part of my academic journey.

To my family, especially my parents, brother and girlfriend, your unwavering support, understanding, patience and encouragement have been the driving force behind my efforts.

Lastly, I'd like to thank my supervisor, Andrew Calway, for his guidance and support throughout this project.

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.



SIGNED: JOSHUA CAREY

DATE: 06/12/2023

TABLE OF CONTENTS

List of Figures

Figure **Page**

List of Tables

Table **Page**

CHAPTER 1

INTRODUCTION AND MOTIVATION

Maritime travel has been a cornerstone of human civilisation, facilitating the exchange of goods, ideas, and cultures around the globe. The annals of history are full with instances of seafaring civilisation harnessing the power of wind to propel their vessels across the oceans. It is posited that ancient Neanderthals embarked on maritime voyages in the southern Ionian Islands between 110 to 35ka BP [1]. The quintessence of maritime travel has predominantly been wind-powered sails, which remained unchallenged until the industrial revolution ushered in the era of fuel-powered engines.

The art and science of sailing have evolved significantly over millennia, from rudimentary rafts and canoes to sophisticated sailing ships with complex rigging systems. Ancient civilisation, including the Egyptians, Phoenicians, and Polynesians, made remarkable advancements in sailing technology, enabling them to explore and trade over larger swathes of the ocean [2]. The medieval period saw the advent of the compass and the astrolabe, which further facilitated maritime navigation and exploration. The Age of Discovery, epitomised by the voyages of Columbus, Vasco da Gama, and Magellan, was propelled by advancements in sailing technology, which enabled transoceanic voyages and the establishment of maritime empires.

The industrial revolution in the 18th and 19th centuries marked a significant turning point in maritime propulsion. The invention of the steam engine heralded the decline of wind-powered sailing and the rise of fuel-powered propulsion systems. Steam-powered ships and later, diesel-powered ships, offered greater reliability, speed, and capacity compared to their wind-powered predecessors, thus becoming the preferred mode of maritime transportation [3]. The transition to fuel-powered engines also mirrored the broader industrial and technological advancements of the era, which prioritised speed, efficiency and profit over traditional methods.

1.1 A Renewed Interest in Wind Propulsion

The environmental costs of fuel-powered maritime transportation have become increasingly apparent in the modern era. The shipping industry is a notable contributor to global carbon emissions, and the negative effects of pollution on marine ecosystems around the world are well-documented [4]. These challenges have rekindled interest in wind propulsion as a sustainable alternative, prompting a re-examination of the principles that guided ancient and medieval sailors. The modern iteration of wind propulsion seeks to amalgamate the age-old wisdom of harnessing wind power with contemporary technological advancements to create eco-friendly and efficient maritime transportation systems.

Contemporary wind propulsion technologies like Flettner rotors [5], wing sails, and kite systems are being revisited to mitigate the environmental impact of maritime travel. Among these, kite-powered vessel technology stands out due to its potential for higher efficiency and ease of retrofitting. Kites offer two main advantages over traditional sails: they can move relative to the vessel, generating their own apparent wind and can be flown at higher altitudes, accessing different wind systems, where the wind is stronger and more consistent. The relative movement of a kite generates apparent wind, allowing for it to generate near its maximum potential force even when the vessel is stationary. This enhanced apparent wind results in a larger force compared to a sail of equivalent area.

However, the effective operation of kite-powered vessels requires precise control, which is skill-intensive. To leverage the full benefits of kites as a scalable propulsion method, implementing autonomous control is crucial.

Reinforcement Learning (RL), a subset of machine learning, presents a compelling avenue for optimising the autonomous control of kite-powered vessels. RL, which is based on learning through interaction with an environment, offers a potential for developing advanced control systems and strategies that could greatly improve the effectiveness of kite-powered vessels.

1.2 Project Definition

1.2.1 Research Question

How can reinforcement learning (RL) techniques be effectively applied to autonomously control a kite-powered vessel?

1.2.2 Objective and Scope

This project focuses on developing a simulation environment to train an RL agent for autonomous control of a kite-powered vessel. Given the impracticality of directly training in the real world due to safety concerns, cost, and time constraints, the simulation environment offers a controlled and cost-effective alternative. However, this approach raises critical questions about realism

and integration with real-world conditions: What level of realism is sufficient? Which real-world features are crucial to simulate? How can we ensure the agent's real-world performance?

1.2.3 Vessel Configuration

The kite-powered vessel that will be used in this project is based on the concept of taking an existing recreational boat and attaching a mass produced Leading Edge Inflatable (LEI) kite to it. Figure 1.1 shows a Silent 60 [6] electric catamaran with a Core [7] LEI kite attached to the front with the Wingit Kite Control system, which is discussed in section 2.4. Figure 1.2 shows the kiteboat model that was created for this project, this model is based on a monohull boat with a LEI kite attached to the front. The configuration and mechanics of this setup are elaborated in section 3.2.

Figure 1.1: Silent 60 with Wingit Kite Control

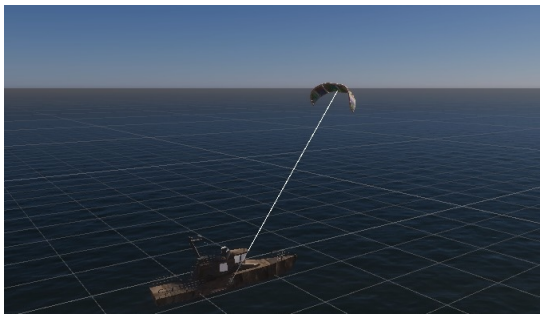


Figure 1.2: Simulation Kiteboat

1.2.4 Autonomous Control Definition

In this context, 'autonomous control' refers to enabling the vessel to operate independently without human intervention. The primary training objective for the RL agent is to navigate the kiteboat towards varying waypoints, increasing in difficulty as the agent's performance improves. This escalating challenge aims to approximate full autonomous navigation and control. 'Control' in this scenario encompasses steering the boat's rudder and managing the kite's flight. A single agent is tasked with managing both aspects, presenting a complex but novel challenge in achieving comprehensive vessel control.

1.2.5 Simulation Environment

The choice of the simulation environment is crucial, given the complex physical dynamics involved in machine learning. Unity game engine, with its MLAgents toolkit, was selected for its exceptional support for machine learning applications, realistic physics engine, and robust visual capabilities. Unity serves primarily as the visual interface and the 'gym' for machine learning simulations.

1.2.6 Simulation Components

Central to this simulation is the representation of water, a key element in the boat's navigation. Unity HDRP Water System 16.0.3 (Unity 2023.2.0b9) offers a realistic depiction of water dynamics, essential for the project's fidelity. While a complete particle fluid simulation was considered, it was deemed excessively computationally intensive and time-consuming for the project's primary focus—training the RL algorithm for kiteboat control. Therefore, Unity's water system was chosen for its balance of realism and efficiency.

1.3 Aims and Objectives

The overarching aim of this research is to develop a system for controlling kite-powered vessels using Reinforcement Learning (RL) techniques. This objective stems from the need to advance sustainable maritime travel technologies and reduce the environmental impact of current propulsion systems.

To achieve this primary aim, the objectives have been structured as follows:

Objective 1: Simulation Environment Development

- To design and implement a virtual marine environment that accurately emulates real-world maritime conditions.
- To construct a realistic model of a boat that exhibits appropriate physical movements in response to environmental forces such as wind and water currents.
  Outcome Goal: To have a physics-based boat able to be controlled and driven around a scene by a human player.

Objective 2: Kite Propulsion Modeling

- To create a physics-based model of a kite within the simulation that reflects authentic aerodynamic behaviours and integrate it onto the boat model.
- To integrate kite control mechanics into an agent's available action space.
  Outcome Goal: To have a physics-based kiteboat able to be controlled and driven around a scene by a human player, using an agent's heuristic controls.

Objective 3: Reinforcement Learning Framework Establishment

- To formulate a set of observations, actions, and rewards that encapsulate the dynamics of autonomous kite-boat control and navigation.

- To deploy the Proximal Policy Optimisation (PPO) algorithm, leveraging its actor-critic method for effective policy learning.
  Outcome Goal: To have an agent begin training using PPO to learn to control the kiteboat.

Objective 4: Autonomous Agent Development

- To develop an RL agent capable of learning basic control and manoeuvres, starting with simple navigating towards a target and maintaining a constant course.
- To refine the agent's capability to adaptively control the kite's position and angle to optimise propulsion for speed while navigating towards a target.
  Outcome Goal 1: To have an agent that can navigate towards a target in a straight line.
  Outcome Goal 2: To have an agent that can navigate towards a target in any direction, including using manoeuvres to take the optimal path.

Objective 5: Efficacy and Optimisation Testing

- To utilise High-Performance Computing (HPC) [8] resources for scaling up simulations and optimising the training process.
- To rigorously evaluate the trained agent's performance in simulating autonomous navigation in various environmental scenarios.
  Outcome Goal 1: To train an agent using the HPC resources.
  Outcome Goal 2: To have an agent that can navigate towards a target in a straight line under various environmental conditions, including wind and waves.

Objective 6: Real-World Applicability Assessment

- To extrapolate simulation findings to assess real-world applicability and propose a practical deployment of RL in kite-powered vessels.
- To provide recommendations for further research and development based on empirical results obtained from the simulation studies.

These objectives pave the path towards achieving the central goal, ensuring each phase of development builds upon the last. By concluding this research it is anticipated that the contributions made could impact sustainable maritime transportation solutions.

CHAPTER 2

BACKGROUND

Maritime innovation has consistently driven technology forward, transitioning from simple oars to advanced sails. With the emergence of highly efficient and mass produced recreational kites, kite-powered vessels are at the forefront of maritime innovation, aiming to outperform traditional sails in efficiency, manoeuvrability and reliability.

However, the introduction of kites as a propulsion mechanism brings forth a new set of challenges. The dynamic nature of kites, combined with the unpredictable marine environment, requires advanced control systems capable of real-time adaptation and decision-making, in order to be a viable replacement for sails. This is where the application of machine learning, and more specifically Reinforcement Learning (RL), becomes paramount. An RL agent learns to make decisions that maximise a certain objective, often framed as a cumulative reward. The potential of RL in maritime propulsion is evident: it offers a framework for developing control systems that can adapt to changing conditions, environments and learn from experience.

This chapter aims to provide an in depth background into the technologies that will be leveraged in this thesis. This includes a detailed examination of the core concepts of RL, the mechanisms behind the PPO algorithm, and the Unity game engine. We will also explore the current state of kite-powered vessel technologies, identify gaps in existing research, and highlight the novelty and potential contributions of the proposed work.

2.1 Reinforcement Learning (RL)

Reinforcement Learning (RL) is a prime example of machine learning and has been a hot topic in artificial intelligence (AI) over the last few years, with applications ranging from autonomous driving to sophisticated game-playing algorithms. At its core, RL is about learning by interaction: an agent takes actions in an environment to maximise some notion of cumulative reward. The

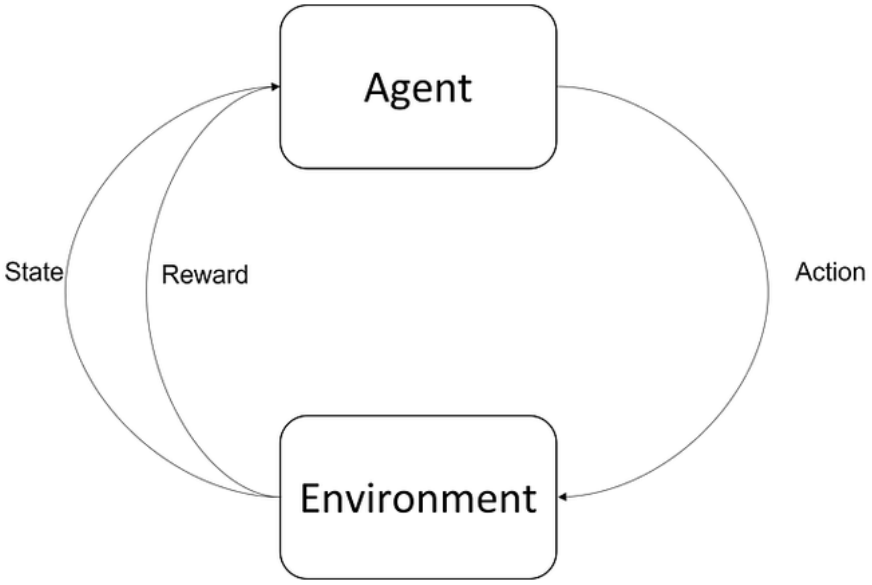


Figure 2.1: A diagram of the RL Loop

agent learns from the consequences of its actions, rather than from being explicitly taught, making it a powerful tool for tasks where the optimal strategy is unknown or hard to define [9].

Imagine teaching a child to ride a bicycle. You don't provide a step-by-step manual; instead, the child learns by trying different actions (like pedaling or balancing) and receiving feedback (falling down or moving forward). This trial-and-error approach is the essence of how RL works. The agent (in this case, the child) interacts with its environment (the bicycle and the ground) and learns a policy that dictates the best action to take in any given situation based on the rewards (or penalties) it receives [10]. Figure 2.1 illustrates this loop.

Historically, RL has its roots in the fields of operations research and behavioral psychology. The idea of learning optimal strategies through interaction has been explored in various contexts, from game playing to industrial optimisation [11] to fine-tuning large language models as recently shown by OpenAI. However, it's the modern advancements in computational power and algorithms, of the last few years, that have propelled RL to the forefront of AI research. Games like Go, which were once considered too complex for computers to master, have now been conquered by RL agents, showcasing the potential of this approach[12].

Boats, with their intricate dynamics and the unpredictable nature of water, present a challenging environment for control systems. Traditional control methods often rely on predefined rules and heuristics, which might not always be optimal or apply to unique situations, especially in changing conditions. These autonomous controls vary from something as simple as a piece of bungee to Proportional–integral–derivative (PID) controllers to more complex systems like Model Predictive Control (MPC) [13]. Enter RL. With its ability to learn from experience, an RL-based control system should be able to adapt to varying conditions, ensuring smooth sailing even in

turbulent waters, and gusty winds.

But why stop at boats? The concept of using kites to harness wind power for propulsion is not new. Historically, kites have been used in various cultures for fishing, transportation, and even warfare [14]. In the modern context, kites offer an exciting alternative to traditional sails, providing more power and manoeuvrability. However, controlling a kite, especially in varying wind conditions, is a complex task. Kite control has only been explored in recent years, primarily in the field of renewable energy, where large ram-air kites are used to harness wind power for electricity generation [15]. However these problems are static and do not handle situations where the base of the kite is moving. This is where RL shines. By continuously interacting with the environment and adjusting the kite's position and angle, an RL agent can learn the optimal control strategy to harness the maximum wind power, propelling the boat efficiently.

The potential applications of RL in marine navigation are vast. From optimising routes for cargo ships to ensuring safe navigation in crowded ports, the possibilities are as vast as the sea. Moreover, as environmental concerns become more pressing, the need for efficient and sustainable maritime solutions becomes paramount. RL, with its ability to optimise and adapt, can play a pivotal role in addressing these challenges [16].

Reinforcement Learning is not just another tool in the AI toolkit; it's a paradigm shift in how we approach problem-solving. Its potential in the maritime world is just beginning to be tapped. As we venture into the future, with boats steered by intelligent agents and sails replaced by kites controlled with precision, it's clear that RL will be involved [17].

2.2 Proximal Policy Optimisation (PPO)

Proximal Policy Optimisation (PPO) has emerged as a landmark algorithm in the domain of Reinforcement Learning (RL), distinguishing itself through its blend of efficiency, simplicity, and effectiveness [18]. PPO is part of the policy gradient family of RL algorithms and is designed to resolve some of the challenges encountered in earlier RL methods.

Traditional policy gradient methods typically update the policy based on each data sample, a process that can be inefficient and unstable. PPO innovates on this by introducing a 'surrogate' objective function. This function is key to PPO's operation: it not only guides the improvement of the policy but also ensures the new policy does not deviate too far from the previous one. This balance is achieved through multiple epochs of minibatch updates, allowing for a more gradual and stable policy evolution [19].

PPO's development was motivated by the need for an algorithm that could offer scalability, data efficiency, and robustness in a more accessible form than its predecessors. Deep Q-learning and standard policy gradient methods often fell short in robustness and efficiency. Trust Region Policy Optimisation (TRPO), although effective in ensuring safe policy updates, was complex and less adaptable to different architectures [20].

The cornerstone of PPO lies in its objective function, which uses clipped probability ratios. This clipping mechanism acts as a safety net, ensuring that updates do not diverge too greatly from the current policy, providing a lower bound on policy performance. This mechanism allows PPO to iteratively sample data from the environment and optimise the policy, maintaining a balance between exploration and exploitation.

Empirical studies have demonstrated PPO's superiority in various applications, particularly in environments with continuous control tasks and complex decision-making scenarios. In comparative analyses, PPO has consistently outperformed other algorithms, achieving better sample complexity and simpler architecture while maintaining competitive performance [21].

PPO employs an 'actor-critic' approach, utilising two neural networks: the actor, which determines the policy, and the critic, which evaluates the actions based on the value function. This dual-network system enables PPO to learn optimal policies effectively by balancing the actor's exploration of new strategies with the critic's evaluation of past experiences.

The actor-critic method can be broken down into the following steps:

1. **Actor**: The actor network proposes an action given in the current state. The action is drawn from a probability distribution (the policy π) parameterised by the networks weights.

2. **Critic**: The critic network estimates the value function $V(s)$, which predicts the expected return (sum of future rewards) from state s under the current policy.

3. **Advantage Estimation**: The advantage function $A(s,a)$ quantifies how much better taking a particular action a is, compared to the average action in state s, and is computed as

   $$A(s,a) = Q(s,a) - V(s) \tag{2.1}$$

   where $Q(s,a)$ is the action-value function, which is the expected return after taking action a in state s.

4. **Objective Function**: PPO optimises a clipped surrogate objective function to prevent large policy updates, which could lead to performance collapse. The objective function L^{CLIP} is defined as

   $$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.2}$$

   where $r_t(\theta)$ is the probability ratio between the new and old policy, $\hat{A}_t$ is the advantage at time step t, and ϵ is a hyperparameter that controls the size of the policy update.

5. **Policy Update**: PPO uses this objective to update the actor network's weights, maximising the expected return while avoiding too large policy updates.

6. **Value Function Loss**: The critic network is trained to minimise the value function loss, which is typically the Mean Squared Error between the estimated value function $V(s)$ and the observed return R.

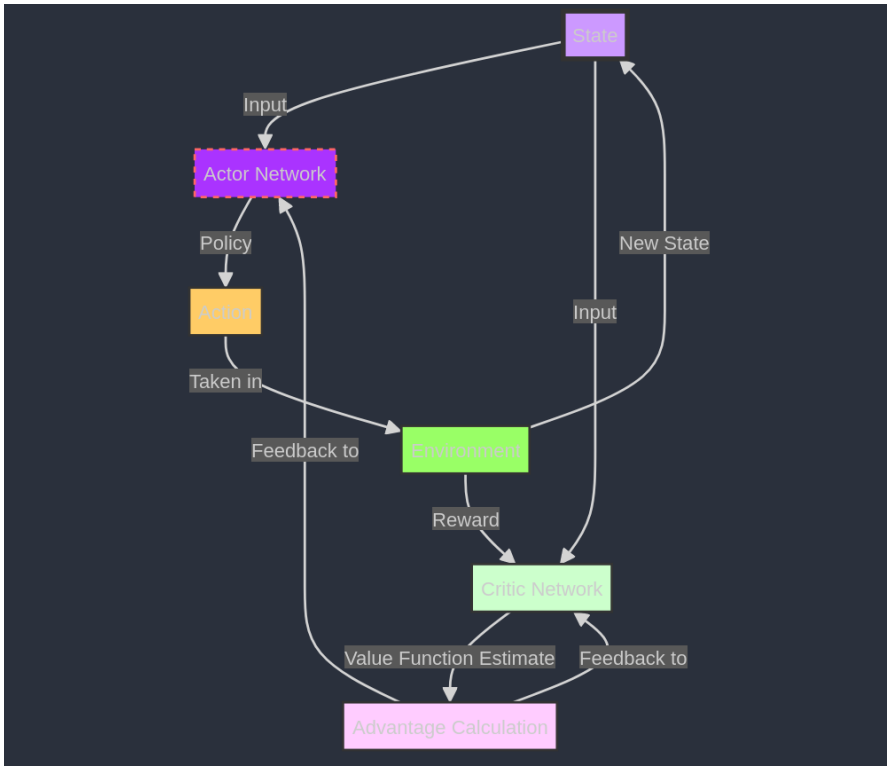


Figure 2.2: Actor-Critic Method

7. **Entropy Bonus**: To encourage exploration, PPO adds an entropy bonus to the objective function, which incourages the policy to produce a wider distribution of actions. This is done by adding the entropy $H(\pi)$ of the policy to the objective function, weighted by a hyperparameter β.

PPO iterates between sampling data through interaction with the environment and optimising the clipped objective function using stochastic gradient ascent. This optimisation is typically done using minibatch updates for efficiency.

By employing PPO, the agent learns to balance exploration (trying new actions) with exploitation (taking known rewarding actions), which is particularly effective for complex tasks like sailing a kiteboat where the agent must adapt to dynamic conditions and long-term consequences of actions. Figure 2.2 illustrates the actor-critic method visually.

But the story doesn't end with PPO alone. Unity's ML-Agents toolkit, which is introduced in section 2.3, seamlessly integrates with PPO. ML-Agents provides a platform for training intelligent agents within the Unity environment, and when combined with the power of PPO, it paves the way for robust and efficient training regimes. This synergy between PPO and ML-Agents is particularly promising for complex simulations, such as kiteboat training, where agents can iteratively learn and refine their strategies for optimal performance [22].

The Proximal Policy Optimisation algorithm is a testament to the continuous evolution and

innovation in the field of Reinforcement Learning. Its simplicity, efficiency, and robustness make it a prime choice for a huge range of applications.

2.3 Unity Game Engine

Unity, a name that resonates with game developers was born in the city of Copenhagen, Denmark, in 2005, Unity has since evolved into a powerhouse, responsible for games like 'Among Us' and 'Pokemon Go' [23].

At its heart, Unity is a cross-platform game engine designed to craft both 2D and 3D experiences. It offers a harmonious blend of a powerful graphical editor and the flexibility of `C#` coding, allowing developers to translate their visions into virtual realities [24], while the engine's core is written in C++.

Diving into the basics of Unity game development, one is greeted with a plethora of tools and components that simulate real-world interactions. Unity's lighting, physics, rigidbody, and colliders work in tandem to create immersive and impressively realistic environments. Whether it's the glint of sunlight reflecting off a surface or the bounce of a ball [25]. Developers can further enhance objects with custom `C#` scripts, giving rise to unique and custom gameplay experiences.

Unity's rigid body component adds a whole new dimension to game development, allowing them to be influenced by gravity. Combine this with the material component, and one can create mesmerising visual effects [26]. Unity has two types of gameplay updates: Update and FixedUpdate. While the former is called every frame during gameplay, ensuring fluid animations and interactions, the latter syncs with the physics engine's frame rate, making it ideal for moving objects around and applying forced in realtime [27].

Unity's ML-Agents toolkit is a game-changer for those looking to infuse artificial intelligence into their games. ML-Agents provides a platform to train intelligent agents within the Unity environment using Reinforcement Learning, making it an ideal choice for complex simulations like kiteboat training. Unity is not just a game engine; it's a platform for innovation, and a testament to the limitless possibilities of virtual worlds.

2.4 Existing Kiteboat Technologies

As kite technologies improve and kites become more mainstream and accessible, there are several companies that have started to explore the potential of kite-powered vessels, both for a commercial and recreational purpose. Wingit [28], is a German company that have specialised in creating autonomous kite control systems that can be integrated onto pleasure vessels, seen earlier in figure 1.1. This system can use any LEI (leading edge inflatable) kite and has its own custom kite-control-unit, as well as a remote control and autopilot. It is the autopilot that is particularly interesting, as this is what will be investigated throughout this thesis. The Wingit autopilot has 2 modes:

Figure 2.3: Beyond The Sea's 'SeaKite'

- Lying Eight - The kite follows a horizontal figure of eight pattern, this is widely regarded as the best, most stable, and most consistent way to generate the maximum power.
- Zenith - The kite remains at 12 o'clock, directly above the boat.

The crucial thing about these modes is that they are pre-programmed. A complex control system, using line sensors to work out the position of the kite, has been created to allow the autopilot to fly the kite in these preconfigured patters and positions. This is at its core a PID controller, and while it is a good approach to controlling the kite, it is not very adaptable.

Beyond The Sea [29] is another company with kites as their primary focus. They are developing full solutions including kite and control systems for leisure and commercial applications. Their SeaKite seen in figure 2.3, is their latest and most curring edge innovation, which claims to utilise some 'AI' in its control mechanism.

Autonomous control systems for kites are new of the last 10 years, but have primarily focused on methods utilised by Wingit, a pre-programmed control system. Due to the technological boom in artificial intelligence and computing in the last 5 years, forward thinking companies like Beyond The Sea are just starting to explore machine learning as an alternative control method. This project aims to do just that, explore the creation of an autonomous kite and boat control mechanism using machine learning.

CHAPTER 3

METHODOLOGY

This chapter outlines the methodology used to develop an autonomous control system for a kite-powered vessel using Reinforcement Learning (RL). It begins with an overview of the essential tools, including the MLAgents toolkit and its integration with Python and PyTorch, crucial for the RL algorithm and the simulation environment. The focus then shifts to the development of the simulation environment, detailing the boat and kite models and the assumptions underlying their design.

The chapter delves into the RL implementation, discussing the agent script and the dynamics of observations, actions, and rewards that guide the agent's learning within the Unity environment. Finally, the chapter covers the training process, highlighting the use of Curriculum Learning to enhance the agent's learning efficiency and the optimisation strategies employed, including hyperparameter tuning and the utilisation of the Blue Crystal High-Performance Computing (HPC) facility. This section aims to provide an understanding of the computational efforts and processes that were pivotal in the project's development, shown in figure 3.1. All environment scripts discussed in the methodology can be found in the `ai_kite/Assets/Scripts` directory of the project files.

3.1 MLAgents

MLAgents is an open-source project that allows games and simulations to serve as the environment for training intelligent agents. At its core MLAgents utilises RL, although it also supports other methods such as imitation learning.

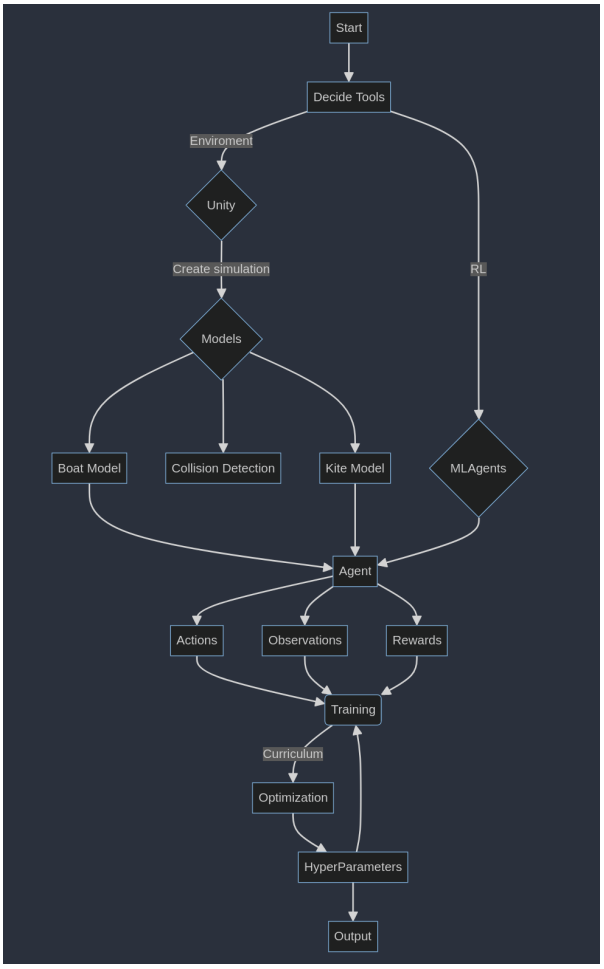


Figure 3.1: Development Process

3.1.1 Python Implementation

MLAgents toolkit is a Python Library that acts as an interface between the environment (gym), in this case Unity, and PyTorch [30]. PyTorch is an open-source machine learning library based on the Torch library, known for its flexibility, ease of use, and native support for GPU acceleration, which is essential for the computation-heavy processes involved in training neural networks. Torch is a Python-based scientific computing package that provides prebuilt components for machine learning and deep learning, as well as a wide range of mathematical functions. MLAgents uses a low level API to communicate directly with the Unity environment (`mlagents_envs`) frame by frame, and an entry point to train (`mlagents-learn`). This stepping process allows for the synchronous collection of observations, executions of actions and retrieval of rewards.

The algorithm used in this project is a Proximal Policy optimisation (PPO) network, and so will utilise the actor-critic method as discussed in section 2.2.

RL, preveously discussed in section 2.1, is an approach to learning where an agent learns

to make decisions by interacting with its environment. The fundamental components of this interaction with the environment are observations, actions and rewards.

- Observations (State): These are the pieces of information that the agent receives from the environment at each step or frame. In Unity, observations are collected through sensors or manually coded to be extracted from the game objects. They are typically fed into the neural network as a vector of floating-point numbers, representing the current state of the environment.
- Actions: Based on the observations, the agent takes actions which are the outputs of the neural network. These actions can be discrete (e.g., turn left, turn right) or continuous (e.g., change angle by a certain degree). The neural network's output layer is designed accordingly to provide the appropriate action space for the agent. (Configured as part of the behavioural parameters in Unity)
- Rewards: After taking an action, the agent receives a reward signal, which is a numerical value indicating how well the action contributed to achieving its goal. This reward is used to adjust the neural network's weights, with the aim of maximising the total accumulated reward.

A full breakdown of the actions, observations, rewards, and how the agent script configures these for this project can be found in section 3.4.

A sequence diagram of the interaction between the Unity environment and the Python neural networks, can be seen in figure 3.2.

The technical instructions to setup MLAgents in Python and Unity are as follows:

- Initialise a python virtual environment
- Install MLAgents with the command: `pip install mlagents`
- Configure Unity project to use MLAgents by importing the MLAgents package, see section 3.4 for more details.

3.2 The Environment

The first step to this process is to create the environment, which the agent will use to train. In this case that will need to look something like a sailing game; it will have some form of water, a boat, a kite and a course.

The boat part of the kiteboat had two main component scripts, the buoyancy and the rudder, allowing the boat to float and be steered. The implementation of Buoyancy and the rudder are discussed in more detail in section 3.2.1. The kite had a single script that defined the physics and its explanation can be found in section 3.2.2. The final training environment can be seen in figure 3.3.

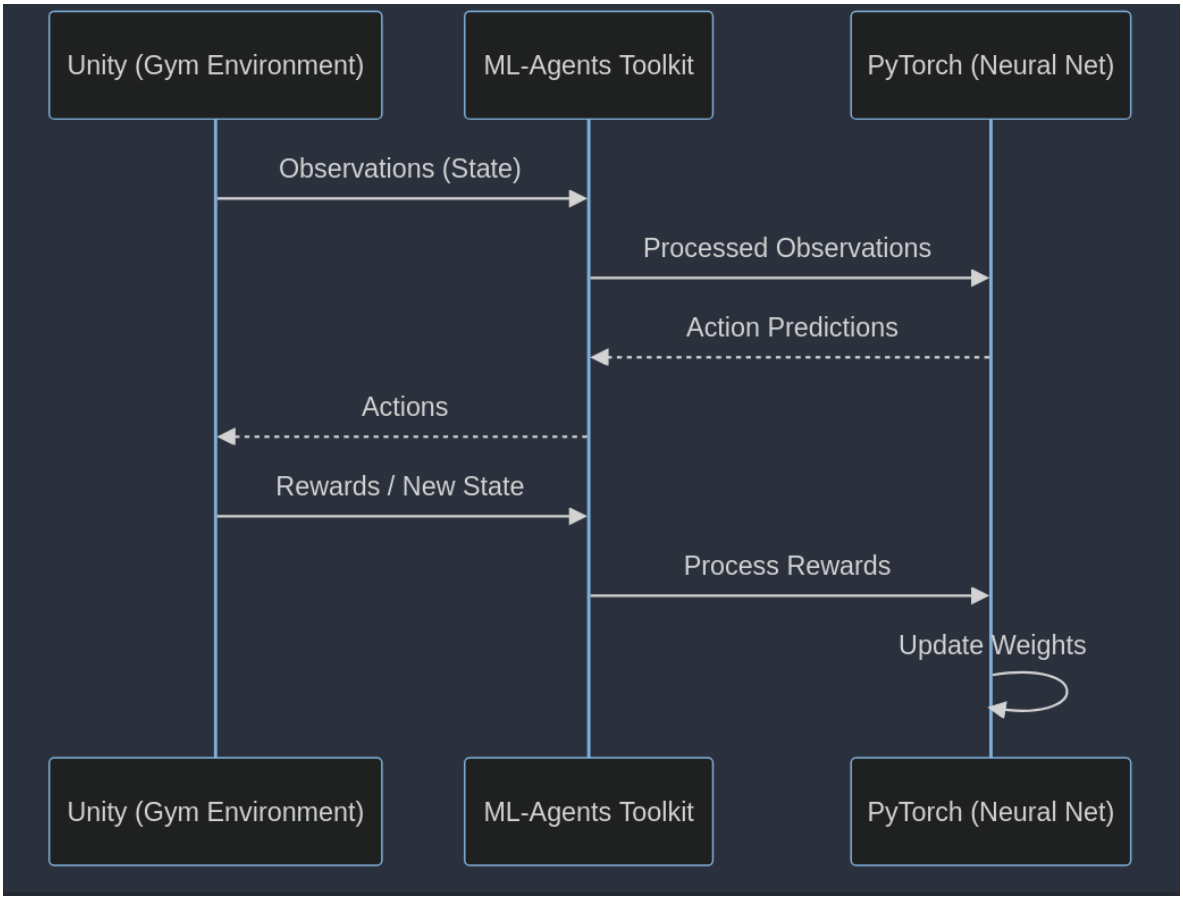


Figure 3.2: MLAgents Sequence Diagram

3.2.1 Boat Model

As with any simulation the first step was to define a series of assumptions that would simplify model.

Boat Assumptions

- The Archimedes force is uniform across all submerged sections of the boat.
- The rudder forces of lift and drag could be aproximated to a torque applied about the rear of the boat.
- Once the boat is moving at a speed greater than 0.25m/s the lift and drag forces of the keel are equal to the downwind component of the kite's resultant force- essentially providing a non'slip condition.

Buoyancy, the force that allows ships to float, was the first physical property to be addressed. Rooted in Archimedes' Principle, it dictates that the buoyant force exerted on a submerged body is equivalent to the weight of the fluid displaced by that body. In our Unity environment, the

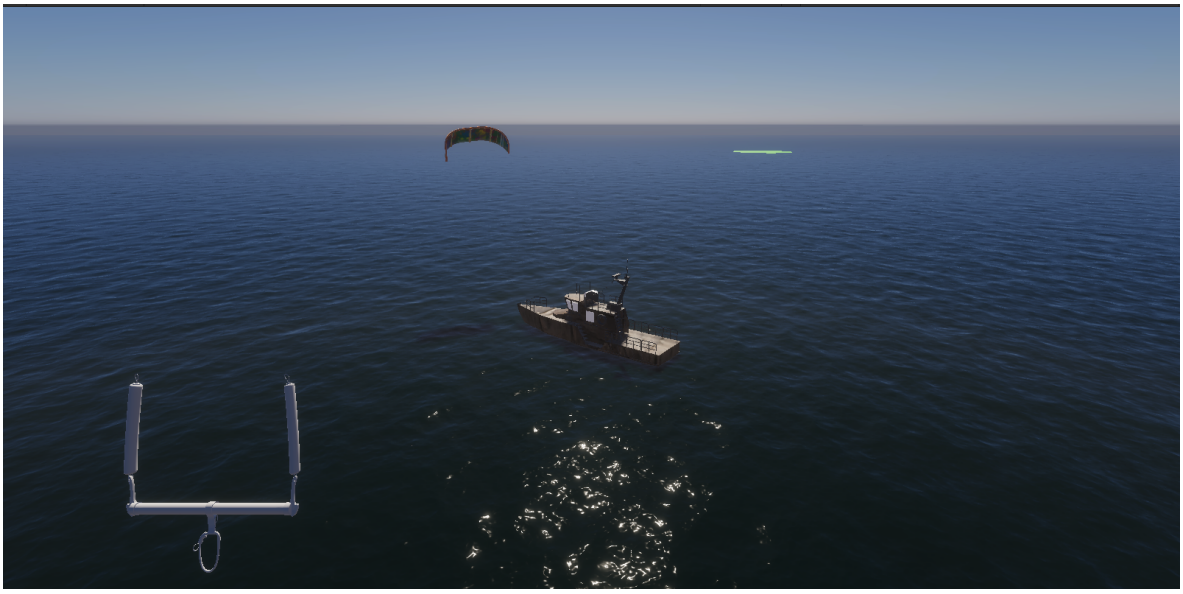


Figure 3.3: Training Environment

boat's hull, represented as a 'mesh', was divided into many small triangles or Voxels. These Voxels became the fundamental units for calculating buoyancy, allowing for a granular and realistic representation of the boat's interaction with water. This was achieved by first calculating the total Archimedes force (AF) of the entire boat using equation 3.1, followed by a local AF at each Voxel. The water level, y component, was then computed at each voxel's (x,z) coordinates to determine if it was above or below the surface. If below the surface the component of the AF was applied vertically at each voxel.

$$F_B = \rho_w g V \tag{3.1}$$

While buoyancy ensures our boat doesn't sink, it's the rudder that grants it direction. The Rudder.cs script handles the implementation of the rudder and the keel. The equation for the torque applied about the rear of the boat is shown in equation 3.2.

$$\tau = \alpha v R \tag{3.2}$$

where τ is the torque, α is the angle of the rudder, v is the speed of the boat and R is the rotation scale.

3.2.2 Kite Model

Capturing the intricate movements of a kite as it fly's through the air involves a complex balance between theoretical aerodynamics and the unpredictability of real-world conditions. In this model

several assumptions were made to streamline the complexity into a more manageable form and are shown below.

Kite Assumptions

- The kite is modelled as a symmetrical aerofoil, with constant lift and drag coefficients.
- Constant wind angle and laminar flow over the entire kite.
- The kite is always in the air, for the initial model the case where the kite crashes and requires relaunching was not considered. This would require adding buoyancy to the kite.

This sections outlines a kite model that, while simplified, serves as an effective tool for designing and testing the RL algorithm. The model is geared towards a realistic representation of the kites behaviour and its response to control inputs. The kite chosen for this project was a Leading Edge Inflatable (LEI) kite, which is the most popular and mass produced recreational style of kites that exists. These kites connect to a control bar via 4 dyneema kite lines. Two center power lines take the load of the kite, while the outside two are responsible for steering, as shown in figure 3.4.

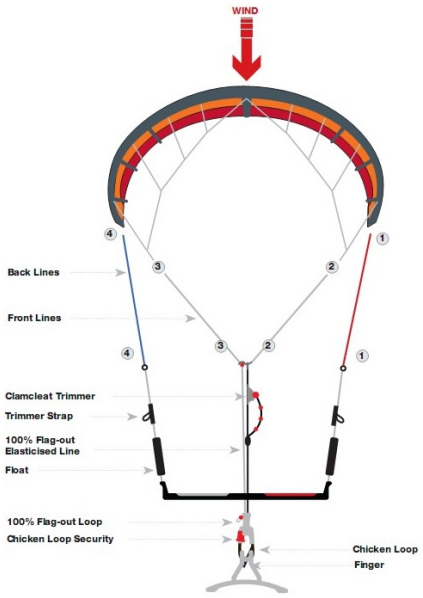


Figure 3.4: LEI Kite Diagram

The kite model was implemented using the kite.cs script. The kite was modelled as a symmetrical aerofoil, and implemented as a rigid body with constant lift and drag coefficients. The lift and drag forces were calculated using the equations shown in equation 3.3 and equation 3.4.

$$F_L = \rho C_L A \frac{v^2}{2} \tag{3.3}$$

$$F_D = \rho v^2 C_D A \frac{v^2}{2} \tag{3.4}$$

where F_L is the lift force, F_D is the drag force, ρ is the density of air, C_L is the lift coefficient, C_D is the drag coefficient, A is the area of the kite and v is the velocity of the kite relative to the wind.

In order to replicate the kite's mechanics and bar configuration, the model uses 4 configurable joints. These joints were fixed at certain lengths from the deck of the boat, designed to permit movement across all rotational axis without any 'bounce' effect. This design enables the kite to descend or 'fall' in conditions of low wind, mirroring real-world behaviour where the kite may lose altitude but can be menoeuvered back into position. To simulate the bar being pulled in, the lift and drag coefficients were increased, this has the effect of increasing the angle of attack of the kite.

3.2.3 Collision Detection

The Gilbert-Johnson-Keerthi (GJK) [31] algorithm is a sophisticated method for collision detection between convex shapes. This algorithm was the approach taken to detect weather the kiteboat had reached the waypoint during training, and later, to work out if it had rounded the marks of the racecourse. The GJK algorithm operates by iterative refinement of a simplex, which is a set of points that can define a line segment, triangle, or tetrahedron. The algorithm progresses by assessing whether the simplex contains the origin, which would imply an overlap between the two shapes. The initial direction d is determined by the normalised vector from center of object 1 to the center of object 2, constrained in the x-z plane by nullifying the y component, meaning the algorithm is only concerned with the horizontal plane.

Within the GJK method, the Support function plays a pivotal role, calculating the Minkowski difference between the two shapes in a specified direction. This is achieved by finding the farthest points along that direction on both shapes and then subtracting them to obtain a single point in the Minkowski space.

The HandleSimplex function is a recursive strategy that adjusts the simplex and direction d based on whether the current simplex is a line or triangle. For a line, the LineCase function is invoked, and for a triangle, the TriangleCase is employed. These functions adjust the simplex and direction of search to move closer to the origin, if it is not already contained within the simplex.

3.3 Controls

In order to ensure the agent would be able to learn to sail the kiteboat a playable game version was created. As discussed above the kite was modelled using 4 configurable joints to replicate the line system. There are several ways of configuring these so that a simple control input will result

in the desired movement of the kite. The controls chosen for this project were 6 discrete actions, shown in table 3.2, allowing the kiteboat to be steered and the kite to be flown.

3.4 RL Implementation

In order to use Unity game engine as the environment for RL it must be setup correctly, the steps are as follows:

1. Enable MLAgents in Unity: Window → Package Manager → MLAgents → Install
2. Create an empty game object in the scene.
3. Create a new `C#` script of with class of type 'agent' and attach it to the game object, include the methods discussed in subsection 3.4.1.
4. Add the Behaviour Parameters component to the game object.
5. Create a new agent config file (`.yaml`) and set the 'Behavioural Name' in the Behaviour Parameters component to the name of the config file.

3.4.1 The Agent Script

The kiteboat agent sets up the scene for learning and in order for Unity to correctly process the script it must have the following 4 functions implemented:

- OnEpisodeBegin()
- CollectObservations(VectorSensor sensor)
- OnActionReceived(ActionBuffers actions)
- Heuristic(in ActionBuffers actionsOut)

These are the minimum requirements for the agent setup, however there are several other functions that can be implemented to further customise the agent, handle errors and provide additional information. The agent starts by taking in the Kite and the Boat rigid bodies, as well as the rudder and kite scripts. A new gjkCollisionDetection is also initialised. The script starts with the OnEpisodeBegin method, which in turn starts by ensuring the training environment has been reset. The reset method sets the velocities and angular velocities of all rigid bodies in the scene to 0 and returns the kiteboat to a starting position. The remaining methods will be discussed in more detail below.

3.4.2 Observations

CollectObservations provides the network will all the information about the State of the environment, and so aims to provide all the required information for the agent to make an informed decision. This means fully describing the movement of the kite and boat, as well as the position of the boat relative to the waypoint, allowing it to gain an idea of direction. It is easy to see how the number of observations could rapidly increase, but this would also increase the complexity of the network and the likelihood of the agent becoming 'confused' by the data its receiving. With this in mind the goal is to provide all the required information about the state in the minimum number of observations possible, this is achieved by combining vectors where appropriate. Another consideration when thinking about the observations was that they should be possible to collect if this system were to be created in the real world. This means that the observations should be possible to collect using sensors, such as GPS, wind speed and direction, and accelerometers. The observations used in this project are shown in table 3.1.

Observation	Vector Size	Description
Distance to the waypoint	1	The distance to the waypoint from the boat
Boat Speed	1	The speed of the boat in the forwards direction
Wind vector	3	The direction of the wind
Relative Boat Angle	1	The angle of the boat relative to the wind
Kite Position	3	polar angles of the kite relative to the wind
Relative velocity of the kite	3	The velocity of the kite relative to the boat
Kite altitude	1	The height of the kite above sea level
Rudder Angle	1	The angle of the rudder

Table 3.1: Observations

The total number of observations passed to the network is 14, this is a relatively small number of observations and so the network should be able to process them quickly. When providing observations to an agent is is essential they have no discontinuity in their values, as this can cause the network to become unstable. Observations that may be discontinuous are normalised to a value between 0 and 1. Normalised observations can increase the learning speed of algorithms, as the network does not have to learn the scale of the observations.

3.4.3 Actions

When the agent starts to train it has no idea what to do, it is essentially a blank slate, so starts by randomly flicking around the actions. The agent was given a discrete action space of 6 actions, these are shown in table 3.2. The actions are passed to the network as a vector of discrete 6 values (i.e float of either 0 or 1). The network then interprets these values and outputs the appropriate action. The actions are then passed to the OnActionReceived method, which in turn calls the methods that control the kite and boat. Discrete actions were chosen so that the rate of change

of the angles that control the rudder and kite are not the choice of the network. Some brief experimentation with continuous actions showed the agent was prone to extremes. By limiting the rate of change of the rotations it gives the network an easier job and means the amount of freedom the network has can be configured. i.e. if the network could pick any value between 0 and 1 for the bar position, it would see far more aggressive controls making it harder for a stable flight to be achieved. On the other had the discrete action space encourages the agent to make a decision and stick with it while the action is being applied, this behaviour was further encouraged with the reward functions.

These actions were tested as part of making a playable game, where the agent receives the same actions as a human controlling the kiteboat simulation. The Heuristic method that is one of the required functions for the agent script, allows the agent to be controlled by a human when no model is present. This is useful for testing the environment and the controls, as well as for playing the game. The Heuristic method takes in the action vector and sets the values of the actions to the values of the keyboard inputs. The keyboard inputs are set in the Unity editor and are shown in table 3.2.

Action	**States**	**Description**	**Keyboard Input**
Kite Bar Position	Left, Off, Right	Turns the bar to the right or left	Arrow left/right
Rudder Angle	Left, Off, Right	The angle of the rudder	Keys 'A' and 'D'

Table 3.2: Actions

3.4.4 Rewards

The rewards provided to the agent each step are the fundamental input that influences the future actions taken by the neural network, and so must be carefully applied when the agent performs positive actions and penalised when it performs negative actions. The reward function is the most important part of the RL algorithm, as it is the only way the agent can learn. The reward function is also the most difficult part of the RL algorithm to get right, as it is very easy to create a reward function that does not encourage the desired behaviour, moreover it is common to create a reward function that helps the agent fall into local maxima while training, especially with complex tasks. Table 3.3 shows the rewards used in this project, These rewards were however not all applied at the same time, they were applied in stages as the agent progressed through the curriculum. This is discussed in more detail in the following section 3.5.

It is crucial to emphasise that rewards are allocated incrementally, as apposed to a single value per step. For instance, the 'Boat Forward Speed' reward is calculated and added every fixed update cycle, whereas the 'Kite Flying' reward is applied after each action taken by the agent. As discussed in section 2.2, the PPO algorithm employed utilises multiple iterations

[1]'previous' is the previous distance to the waypoint at the previous time step, and 'current' is the current distance to the waypoint.

Reward	Condition	Value	Timing	Description/Impact
Boat Forward Speed	>1m/s	+0.005	Every Step	Optimise the boat for forward speed
	>4m/s	+0.01	Every Step	
	<0.1m/s	-0.001	Every Step	
Rudder Angle	> 60°	-0.001 * abs(rudderAngle-60)	Every Step	Penalise the agent for aggressive steering
Distance to Waypoint	current < previous[1]	$+\frac{10}{current}$	Every Step	Encourage the agent to move towards the waypoint
	current > previous	-0.01 * (current - previous)	Every Step	
Waypoint Reached	Waypoint Reached	+10	On End	Encourage the agent to reach the waypoint
Kite Crashes	Kite Hits Water	-10	On End	Penalise the agent for crashing the kite
Kite Flying	Kite Angle > 15°	+0.01	On Action	Encourage the agent to keep the kite in the air
	Kite Angle > 45°	+0.05	On Action	
Keep Alive	Always	-0.001	Every Step	Encourage the agent to explore the action space
Decision Making	Current Actions != Previous Actions	-0.1	On Action	Encourage the agent to be decisive in its actions

Table 3.3: Rewards

of minibatch updates. This approach necessitates the immediate allocation of rewards to the network, ensuring they are included in the next epoch evaluation, thus aligning the reward system with the temporary dynamics of the learning process.

3.5 Training

Before commencing the training of the kiteboat agent, a simpler test agent was created to check the workflow and ensure the environment was setup correctly. This test agent was a simple cube that was trained to move towards a waypoint, following a pacemaker. This was a good test of the environment as it was a simple task that could be easily visualised as shown in figure 3.5 However this test agent proved more tricky than initially anticipated and after additional research some core RL training concepts that improve the quality of training were discovered. The first of these was to make the problem a 'Keep Alive' i.e. do the correct thing or die. In the context of the path follower this meant that should the cube ever be further away from the pacemaker than its original distance of 2m the episode was ended and a large negative reward given. This method encourages the agent to do the correct process if it wants to stay alive. Making the problem a keep alive is a simple way to improve the quality of training and is a common practice in RL. The path follower changed from not making much progress over the course of 500000 steps to being able to complete full laps of the course in around 100000 steps.

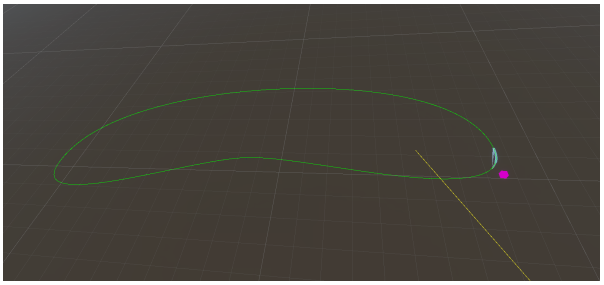


Figure 3.5: Keep Alive

3.5.1 Curriculum Learning

The next RL technique to help improve training is called Curriculum Learning (CL) [32]. CL is an instructional strategy that structures the learning process, much like how a school curriculum guides human learning. It involves organising the learning tasks from simple to complex, facilitating the agent's ability to incrementally acquire, transfer and refine knowledge. In essence it breaks down large complex tasks into more manageable bitesized chunks that the agent can progress through. CL was not utilised in the path follower but was used in the kiteboat agent. The kiteboat agent was trained in several stages shown below.

1. Master the controls- a keep alive problem
2. Sail downwind towards a target
3. Progressively move the waypoints upwind with each completed waypoint
4. Randomly generate waypoints in any direction

Step 1 in the curriculum ensures the agent has a fundamental grasp of the controls and is able to keep the kite in the air. This is a keep alive problem and so the agent is rewarded for keeping the kite in the air and penalised for crashing. The direction the boat sails is not important at this stage, so the reward for distance to the waypoint was not present in this stage. Step 2 is a simple task that the agent can learn quickly, having now gained a grasp of the kite control it must learn to steer straight towards the waypoint. This was also turned into a keep alive problem, move closer to the next waypoint or end the episode. Step 3 is where the agent starts to learn to sail its own path, initially this will still be a straight line until the waypoints are spawning upwind of the kiteboats initial location. At this point the agent must start to experiment with finding the VMG (velocity made good) [33], which is a measure of the speed at which a vessel is moving directly towards its destination, considering both its heading and wind direction. In stage 3 the waypoints will spawn more and more upwind until they are directly upwind, this is in an effort to encourage the agent to find the emergent property of Tacking, a manoeuver by which the nose of the boat transitions through the wind while it turns around.

Figure 3.6 shows the extremes of the difference in the paths at stages 2 and 3 of the curriculum. The path in stage 2 is a mostly straight line towards the waypoint, with the boat navigating downwind. The path in stage 3 is a zigzag upwind from the boats initial location to the waypoint.

3.5.2 Hyperparameters

Hyperparameters are critical configurations that govern the training process of machine learning models. In reinforcement learning, particularly within the scope of PPO, they play a pivotal role in determining how effectively the model learns from its environment. The hyperparameters are defined within the boatAgent.yaml config file. A variety of different configurations were tested

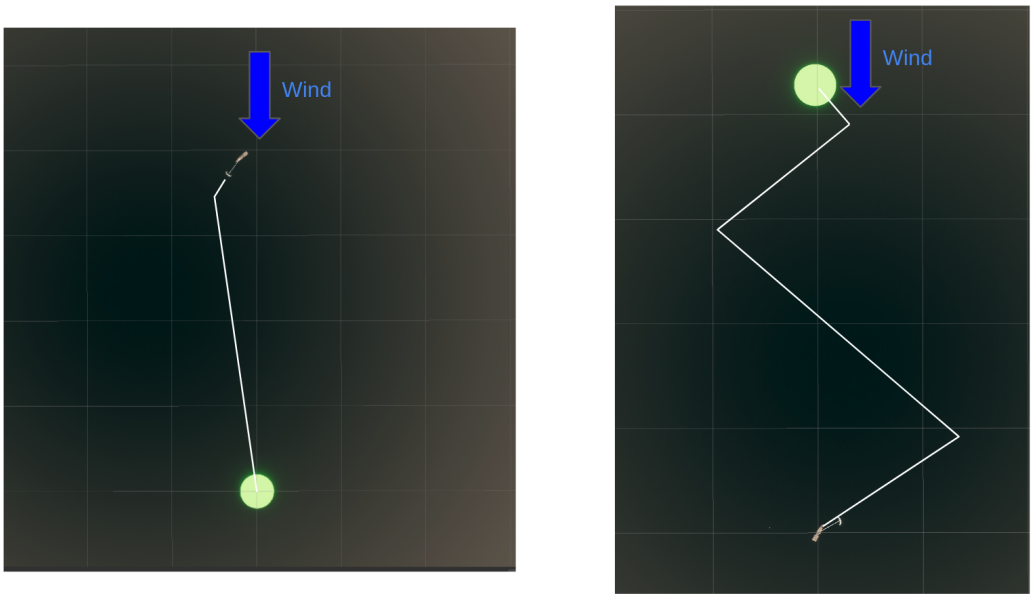


Figure 3.6: Stages 2 and 3 of the curriculum

to find a set of hyperparameters that yielded the best results. It is also possible to schedule the hyperparameters to change over the course of the training, this can potentially lead to better results as the agent can adapt its learning behaviour during different stages of training. To test which hyperparameters were the most effective a grid search was performed. Section A.2.1 of the appendices shows the python script used to generate the 100 config files to be tested. The performance metric of the average reward per episode with a max steps of 1,000,000 was used to determine the best hyperparameters to then be used for the final long training runs. The following hyperparameters were experimented with:

- **Normalised network inputs**- Adjusting normalised inputs ensures that the model receives input data within a consistent scale, which can aid in stabilising and speeding up the training process.
- **Number of layers in the network**- The depth of the network, defined by its number of layers, determines the level of abstraction the model can achieve, impacting its ability to learn complex patterns.
- **Number of nodes in each layer**- The width of each layer, given by the number of nodes, influences the model's capacity to capture nuances in the data, with more nodes allowing for more complex representations.
- **Beta**- The beta hyperparameter balances the exploration-exploitation trade-off by controlling the strength of the entropy term in the objective function, which affects the agent's

policy diversity.

- **Epsilon**- Epsilon values dictate the degree of policy randomness, serving as a threshold for exploration in the agent's decision-making process, promoting the discovery of new strategies and techniques.
- **Learning rate**- The learning rate controls the magnitude of updates to the model's weights, with too high a rate risking overshooting minima and too low delaying convergence.
- **Lamda**- The lambda hyperparameter determines the trade-off between the current and future rewards.

After training the first 100 config files for 1,000,000 steps it could be seen the training was very inconclusive, so further experimentation was conducted. 500 config files were randomly picked from a pool of unique files that varied 10 different parameters by 3 options, comprised of the 6 hyperparameters, 2 network parameters and 2 reward signals. The 500 config files were trained for 10,000,000 to ensure the agent had enough time to learn. The limitation of this approach was that there were a possible 3^{10} combinations of hyperparameters, which is 59,049 unique different configurations, meaning the random 500 selected only represented 0.8% of the possible combinations. However a random selection from the sample it should help identify some of the better hyperparameter configurations. After the best configs from this run were found they were tuned manually and retrained again. The comparison between these different configurations can be viewed in chapter 4, section 4.1.1. The best config can be found in section A.1 of the appendices and is further explained in chapter 4. Given more time it would have been prudent to conduct a larger grid search of the hyperparameters to ensure that the best combination was found for this project.

3.6 Optimisation

3.6.1 Blue Crystal HPC

The Blue Crystal HPC, operated by the University of Bristol, offers significant computational resources tailored for intensive tasks such as machine learning simulations.

To utilise Blue Crystal for MLAgents simulations, the following steps were undertaken:

- **Access and Security:** Gained access to the university's HPC and set up SSH keys for secure communication.
- **File Preparation:** Built the .x86_64 Unity build file and uploaded it, along with the necessary config files and credentials, to Google Drive.
- **Automation with Shell Script:** Developed a shell script to automate the process. This script:

- Retrieves the build files from Google Drive.
- Sets up a virtual environment on Blue Crystal.
- Installs the ML-Agents.
- Manually runs the training for a given number of steps.
- Upon completion, uploads the results back to Google Drive.

The shell script can be found in section A.2.2 of the appendices. Although the process appears simple there were a number of challenges encountered during the usage of the HPC. First and foremost was handling the files required for training, Google Drive proved very useful.By using a link to a public folder, the build files could be easily accessed using the `wget` command. The results dir, which is created during the training and contains the trained `.onx` model was zipped up and uploaded back to google drive using the credentials.json file. As this process had to be entirely autonomous and the HPC node could not perform any mouse clicks, the credentials.json was created by uploading the file to google drive locally and performing the manual authentication process, then uploaded to google drive. Again as no authentication could be completed the drive folder had to be public so anyone with the link could download it, this presented a security risk albeit not major. To mitigate this security risk the OAuth 2.0 Client ID was deleted after each training session. To handle the running of many hundreds of config training runs another shell script was created that configured the setup of the runs. This included downloading the config.zip (that contained the unique config files), and the build.zip that contained the prebuild Unity files. The shell script then unzipped these files and ran the training script for each config file, adding it to the *SLURM* queue. *SLURM* (Simple Linux Utility for Resource Management) is an open-source, scalable cluster management and job scheduling system for Linux clusters, and is the queue management system used on the HPC.

Useful commands for working with Blue Crystal:

- **sbatch:** Submits a job to the queue.
- **sacct:** Checks the status of a job.
- **scancel:** Cancels a job.

Parallelisation and Optimisation

Initially, a single node on Blue Crystal was employed to run the simulation. This node with 28 CPUs was responsible for both hosting the environment and executing the model. However, Blue Crystal's architecture allows for more advanced parallelisation strategies. Distributing the simulation across multiple nodes can enhance efficiency. Additionally, offloading the ML-Agents toolkit to a GPU core can further accelerate the learning process.

However, it's worth noting that the demand for GPUs on Blue Crystal is high. For tasks that don't necessitate the power of GPUs, relying on CPUs, even if they take longer, is a practical choice given the limited GPU availability.

CHAPTER 4

RESULTS AND EVALUATION

This chapter will investigate the results generated from the modeling and reinforcement learning discussed in the previous chapters; it is structured to first present the outcomes of the training steps and hyperparameter tuning, followed by an in depth analysis of the training results and the agents learning process. Subsequent sections provide insights into the agents overall performance, including comparisons to human baselines, and a discussion of the practical deployment of a trained model. The chapter concludes with a critical evaluation of the project, discussing the extent to which the project aims were achieved and the potential for future work.

4.1 Training Results

The training scene used for the final training runs can be found in:
`Assets/Scenes/kiteboat_training`. The final experimental setup was a combination of the best performing elements from the previous experiments.

This section presents the results of the hyperparameter tuning and the final training runs. It includes analyses of the learning curves, performance metrics, and unexpected behaviours observed during training.

4.1.1 Hyperparameter Tuning

This subsection presents the findings from the grid search of hypereparameters, and the effect they had on the training runs. The goal was to identify the best combination of hyperparameters for this training application to then use for the final training runs. The dataset comprised 500 runs, each with a unique configuration file. The hyperparameters include `batch_size`,

buffer_size, learning_rate, beta, epsilon, lambd, num_epoch, hidden_units, num_layers, curiosity_strength, and curiosity_learning_rate. The performance was measured using the 'Best Episode Length'- a metric that represents the duration for which the model successfully flew the kite while navigating towards the waypoint.

A statistical summary was conducted to try and understand the most important tendencies of the data. The mean best episode length was found to be 5785 with a standard deviation of 7030, indicating a huge variability across different runs. A correlation matrix was computed to explore the linear relationship between the hyperparameters and the performance metric, the results of which can be seen in table 4.1. It was observed that num_epochs had the strongest positive correlation (r=0.3007), suggesting a tendency for longer episodes with more epochs. On the other hand batch_size and epsilon showed a negative correlation (r=-0.3007 and r=-0.3007 respectively) to the episode length.

Hyperparameter	**Correlation**
num_epoch	0.3007
buffer_size	0.2640
num_layers	0.1246
Av Loss	0.0973
lambd	0.0964
beta	0.0424
curiosity_strength	-0.0074
hidden_units	-0.0185
curiosity_learning_rate	-0.1168
learning_rate	-0.1168
epsilon	-0.1258
batch_size	-0.2275

Table 4.1: Correlation matrix of hyperparameters

A Random Forrest Regressor [34] was utilised to estimate the importance of each hyperparameter in predicting the best episode length. The model was trained on a subset of the data and validated using a test set, resulting in a Mean Squared Error (MSE) of 53 million. This very large value indicates significant variability and suggests the model's performance is highly sensitive to changing hyperparameters. Figure 4.1 shows the feature importance bar plot, which emphasises the significance of num_epochs and buffer_size in the model's success. The analysis.py script, that performs the above calculations can be found in section A.2.5 of the appendices.

In hindsight, incorporating a dynamic learning rate that varies throughout the training phase would have been beneficial i.e scheduling it to change over the course of the training run. This approach would involve initiating the training with a relatively high learning rate, thereby facilitating rapid learning in the initial stages. As the training progressed, gradually reducing the learning rate would allow the agent to fine tune its policy without unlearning previously learned behaviours, which is a common problem with RL agents. Moreover, employing an non-linear

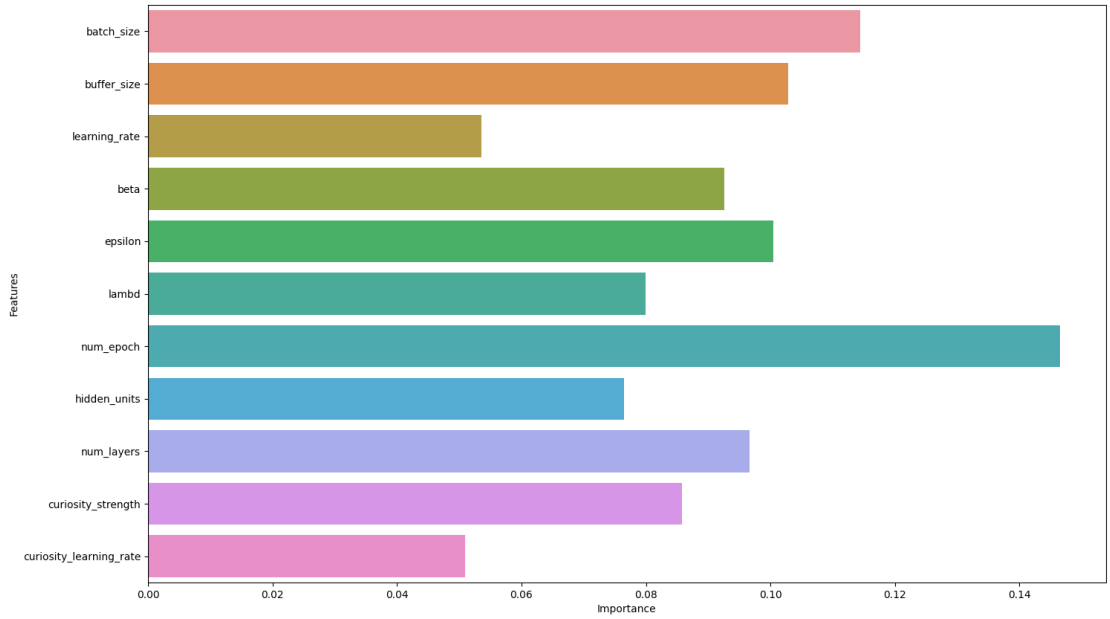


Figure 4.1: Feature importance of hyperparameters

learning rate schedule could have provided a more controlled reduction in pace, enhancing the stability in the latter stages of the training.

4.1.2 Final Training

The final training consisted of 6 different config files, each run for *500,000*, *1,000,000*, *10,000,000* and *30,000,000* steps. Figure 4.2 shows the Environment/Cumulative reward for the training runs conducted for 500k steps. It appears the score converges, but on closer inspection it can be seen that all these runs converge on approximately -9.75, which is very far from the theoretical maximum available. This graph suggests these runs were falling into local maxima after learning and didn't have the number of iterations for the curiosity to force the agent to explore beyond this local maxima.

For the other training runs, it is hard to draw any conclusions about the quality of the training from the model output graphs, and so all the models were run in heuristic mode in an evaluation scene that logged the performance over time. Several of these models were not even able to control the kiteboat and had completely failed to learn anything. Figure 4.3 shows the cumulative reward accumulated over a 90s evaluation run for the 3 models that were unable to control the kiteboat. It can be seen that the cumulative reward never rises as the agents are unable to maintain control of the kiteboat.

The highest performing model was boatAgent_2 trained for *10,000,000* steps, figure 4.4 shows how it performed in the heuristic evaluation given a 300 second time slot. This is the

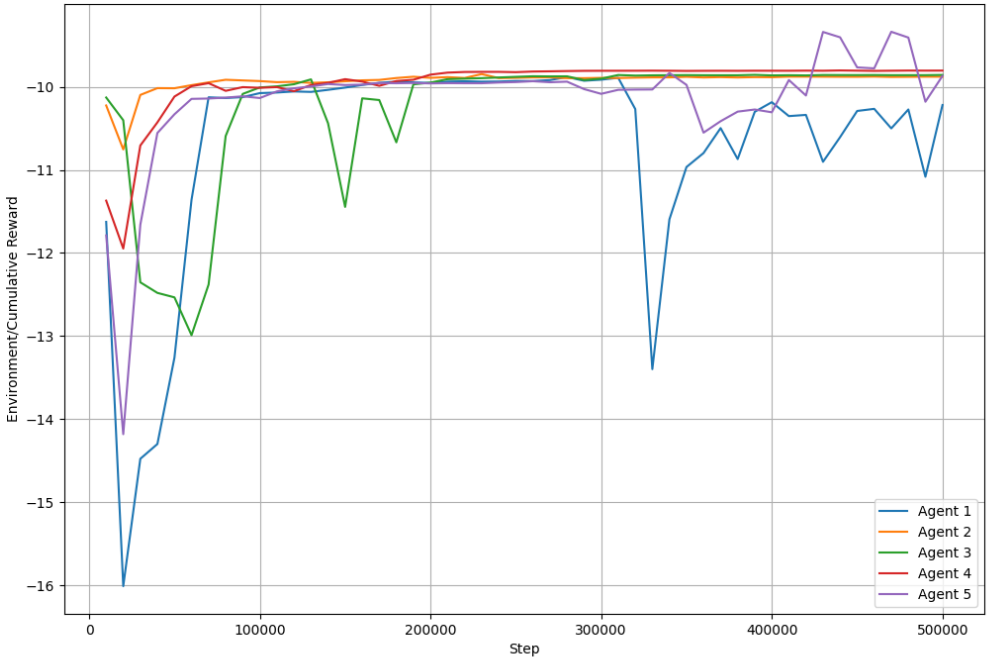


Figure 4.2: Environment/Cumulative reward for 500k steps

best performing heuristic run of 5 conducted. These figures present the evaluation metrics of `Cumulative Reward`, `Completed Waypoints`, `Distance Covered`, `Boat Speed`, `Rudder Angle` and `Kite Height`. After some initial inconsistency the agent navigated through 3 waypoints, with an average boat speed of 5.05 m/s. Over the course of the evaluation it crashed the kite into the water 6 times. These results show the best performing model was able to sail the kiteboat across a number of waypoints, gaining a large reward and maintaining a good speed, however it was still inconsistent unable to fly the kite reliably for the total duration of the evaluation scene.

Human Comparison

Earlier it was mentioned that a heuristic playable game was created to make sure the model felt sensible and could be played by a human player. To create a baseline for the agent's performance a human player played the game for 5 minutes in heuristic mode and the results were logged. The top performing agent was then compared against this baseline to see how it performed. The results can be seen in figure 4.5. This figure shows the average `Cumulative Reward`, `Distance Traveled`, `Boat Speed` and Waypoints Reached per episode, from the 300s evaluation period. It can be seen that the human outperformed the agent in all catagories

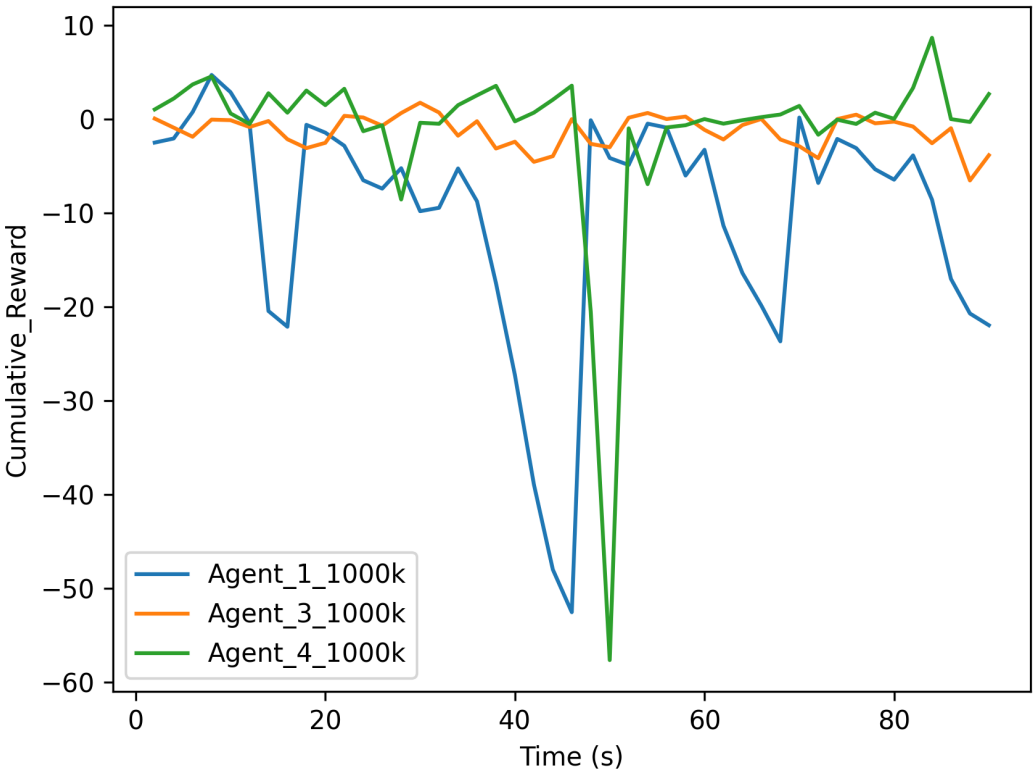


Figure 4.3: Cumulative reward of 3 Agents evaluated for 90s

4.1.3 Hardware

There were several limitations that affected the quality of the training and thus the trained model. First and foremost was compute, as expected when conducting any machine learning training, the more compute available the better. The local machine used for training was a 16-core i9 with 32GB of RAM and a T2000 nvidia graphics, and took approximately 2 hours per million steps completed. This was not a viable option for training multiple agents to a high level of performance, and so the training was attempted to the university HPC. The HPC has 525 Lenovo nx360 m5 nodes each with two 14 core 2.4 GHz CPUs, and 32 GPU nodes with two cards each. Initially it was anticipated the training should be without issue, this was not quite the case. Unity does not support native multi threading, due to the complex nature of its physics engine, so it runs on a single CPU unless manually specified. Manual threading was possible but only for separate tasks that could be called independently of the model, such as the collision detection algorithm. As expected this severely limits the speed of training, and using the hpc did not radically increase these run times. However, the HPC was not without its advantages, the main being that it was possible to run multiple training runs in parallel, and so the hyperparameter

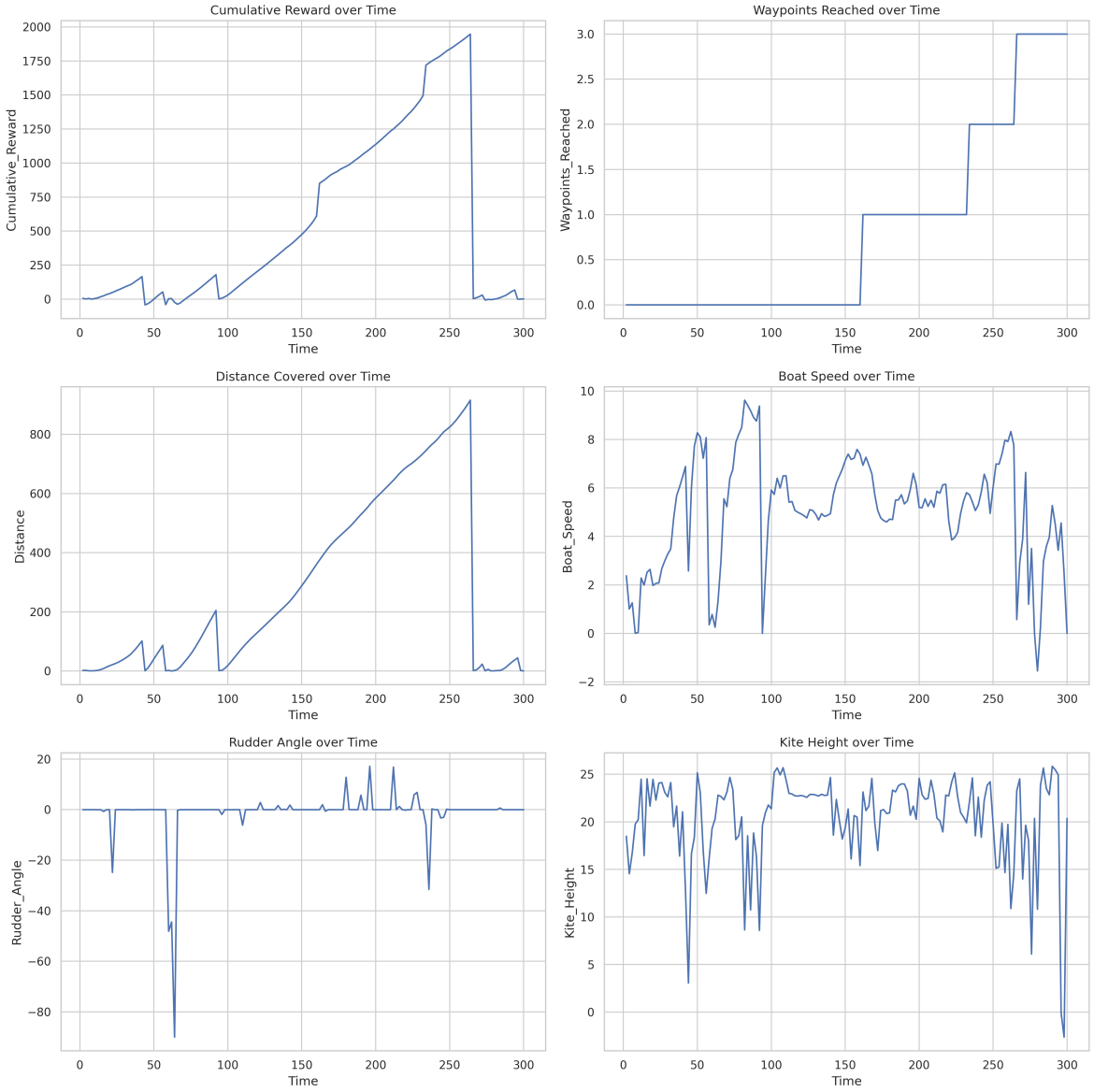


Figure 4.4: Agent stats for 10,000,000 steps model

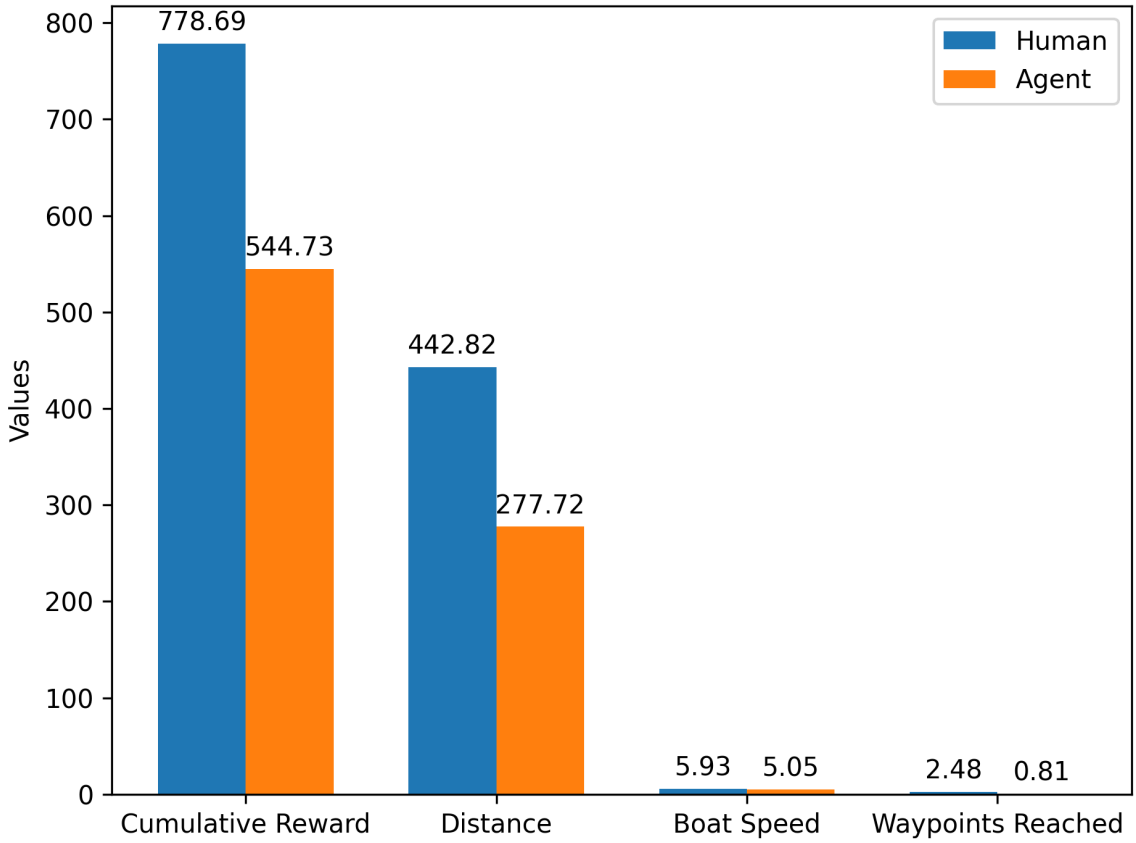


Figure 4.5: Human comparison vs Model Evaluation

tuning was conducted on the HPC. This would not have been possible on the local machine as it would have taken far too long to test all the combinations. The ability to submit a large batch of jobs to be run in parallel and then the results collection automated was a huge time saver. The HPC also had the advantage of being able to run the training for longer periods of time without causing inconvenience, so even though the training speed was limited it was not a problem to let them run for many days.

4.2 Agent Performance

Difficulties Encountered

There were several difficulties encountered when trying to get the agent to learn anything let alone the combination of directionally sailing a kiteboat. One of the most common local maxima that the agent fell into was to steer on 'hard lock' with the rudder at˜$70-90°$, shown in figure 4.6 where the target can be seen as the green area in the distance. This behaviour allowed it to learn to fly the kite very reliably with the boat in a more consistent and stable position. These

episodes provided false positives in the training data because as soon as the agent started to explore the rudder space more it was not able to fly the kite. To try and combat this behaviour a large negative reward was added for aggressive steering as shown in table 3.3. This went some way to discouraging this behaviour but it was still observed in some of the later training runs. After this rudder reward was added it was observed that the agent took almost 5 times as long to learn to fly the kite with some reliability.

Figure 4.6: The agent steering on hard lock

Performance in Curriculum

While the agent showed promise in its basic understanding and control, it was unable to progress sufficiently through the training curriculum. The curriculum, which was designed to gradually increase in complexity, required the agent to reliably control of the kiteboat if it wanted to progress to the next stages, including sailing upwind. The agent's overall inconsistent performance in kite control hindered its progression, preventing it from reaching these more advanced stages.

4.3 Practical Deployment

The results of the model demonstrate unreliability in the agents performance and so it would not be recommended to deploy this particular model into the real world. However, the results do show promise and it is likely possible to create a model that performs reliably enough to warrant building a physical prototype. Integration into a physical system would first require a boat and kite. The boat would need to be large enough to safely contain all the components (including electrics) so a vessel of appropriately 18 - 30ft would be advised. For the kite a standard off the shelf LEI kite would perform fine, the size of the kite could be changed depending on the wind

conditions it would be tested in as it wouldn't noticeably affect the handling. The remainder of the system would involve addressing the following design and manufacture points:

- Kite Control System:
  - Custom reinforced attachment point to the boat. Similar to a mast base attachment point.
  - Electronic winches for kite line management.
  - Load cells for tension measurement in kite lines.
  - Gyroscopic sensors to measure kite orientation.
  - Anemometer for wind speed and direction data.
  - Manual override for emergency control.
- Rudder Control System:
  - Electric linear actuators
  - Rudder position sensor
  - Manual steering mechanism
- Central Processing Unit
  - Microcontroller to run the control system.
  - Data acquisition system (DAQ) to interface with sensors and actuators.
  - GPS module for positioning.
  - Compass module for heading data.
- Power Supply
  - Battery bank with charge controller.
  - Solar panels for charging.

A visual breakdown of how this system would come together is shown in figure 4.7. Addressing these points would lead to an mvp capable of testing. It is important to note that all the observations provided to the agent in the simulation must still be provided to the trained model, so the same observations would have to be collected.

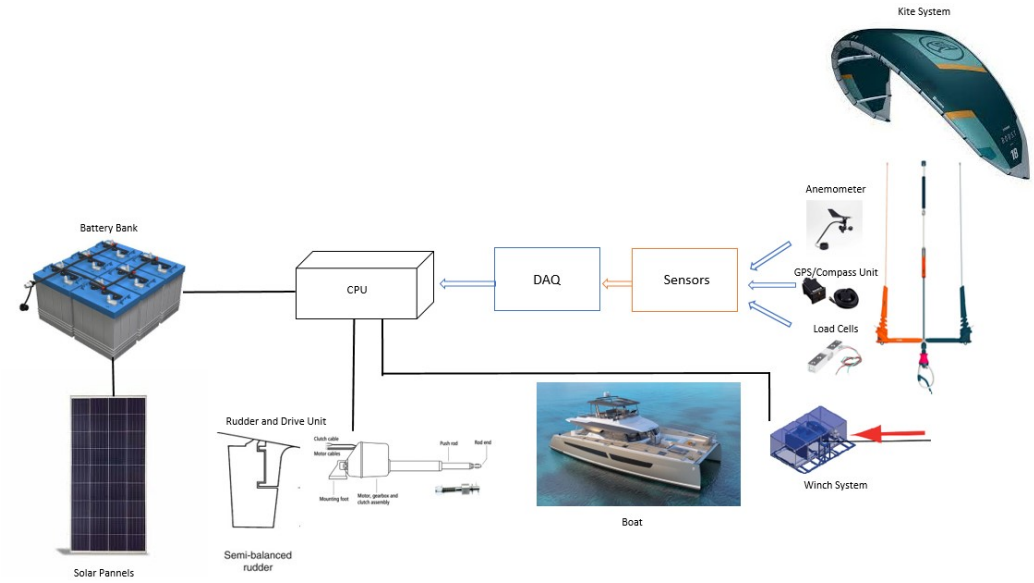
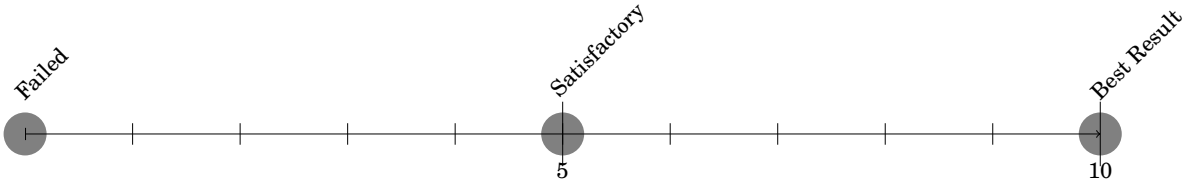


Figure 4.7: Physical kiteboat system

4.4 Critical Evaluation

The overarching aim of this research paper was to develope a system for autonomously controlling a kite-powered vessel using reinforcement learning. Tables 4.2 - explores the extent to which this aim was achieved by investigating the success of each objective and its outcome goal. To evaluate each objective in detail a score was assigned from 0 to 10, with the range shown in figure 4.4. A total score was calculated for each objective and its value discussed.



Objective 1 scored a total of $\frac{27}{30}$, the Unity environment implemented was capable of delivering real world conditions with a variety of wind and waves. The boat model was able to be accurately controlled by a human player and experienced buoyant and drag forces. There was room for improvement within the boat model by implementing a custom particle simulation instead of the Unity water system, which would have provided complete control over the interaction between the water and the boat.

Objective 2 scored a total of $\frac{24}{30}$, shown in table 4.3, the physics of the kite model were satisfactory. The kite generated a resultant force that was capable of pulling the boat and the outcome goal was fully achieved. However there was definitely room for improvement in the kite

Objective 1: Simulation Environment Development	**Score**
To design and implement a virtual marine environment that accurately emulates real-world maritime conditions.	9
To construct a realistic model of a boat that exhibits appropriate physical movements in response to environmental forces such as wind and water currents.	7
Outcome Goal(s)	
To have a physics-based boat able to be controlled and driven around a scene by a human player.	10
Total	$\frac{26}{30}$

Table 4.2: Objective 1 Evaluation

Objective 2: Kite Propulsion Modeling	**Score**
To create a physics-based model of a kite within the simulation that reflects authentic aerodynamic behaviours and integrate it onto the boat model.	5
To integrate kite control mechanics into an agent's available action space.	9
Outcome Goal(s)	
To have a physics-based kiteboat able to be controlled and driven around a scene by a human player, using an agent's heuristic controls.	10
Total	$\frac{25}{30}$

Table 4.3: Objective 2 Evaluation

model, specifically in how the kite rotated. To keep the model simple a torque was applied to the kite about the centre of mass, this was not a realistic representation of how a kite would rotate. A more realistic approach would have been to apply a torque about the point on the wingtip of the kite, switching left/right depending on the direction of turn. Even better than this would be to create a kite model that didn't rely on the assumption that wind was laminar over the entire kite. This would have allowed for non uniform lift over the kite giving it a realistic rotation behaviour. That being said that model employed in this project was sufficient. The kite controls were successfully placed into discrete actions that the agent could control, it would have been interesting to experiment further with continuous actions, but this would have also increased the complexity of the already difficult learning task.

Objective 3 scored a total of $\frac{27}{30}$, shown in table 4.4, the observations, actions, and rewards were all successfully formulated and the agent was able to begin training. The observations and rewards were tuned through lots of trial and error in an attempt to create a stable system for the agent to learn. The PPO algorithm was successfully implemented and the agent was able to train.

Objective 4 scored a total of $\frac{21}{40}$, shown in table 4.5, the agent was capable of learning the

Objective 3: Reinforcement Learning Framework Establishment	**Score**
To formulate a set of observations, actions, and rewards that encapsulate the dynamics of autonomous kite-boat control and navigation.	9
To deploy the Proximal Policy Optimisation algorithm, leveraging its actor-critic method for effective policy learning.	8
Outcome Goal(s)	
To have an agent begin training using PPO to learn to control the kiteboat.	10
Total	$\frac{27}{30}$

Table 4.4: Objective 3 Evaluation

Objective 4: Autonomous Agent Development	**Score**
To develop an RL agent capable of learning basic control and maneuvers, starting with simple navigating towards a target and maintaining a constant course.	8
To refine the agent's capability to adaptively control the kite's position and angle to optimise propulsion for speed while navigating towards a target.	5
Outcome Goal(s)	
To have an agent that can navigate towards a target in a straight line.	8
To have an agent that can navigate towards a target in any direction, including using maneuvers to take the optimal path.	0
Total	$\frac{21}{40}$

Table 4.5: Objective 4 Evaluation

controls of the kiteboat, and able to reliably fly the kite and steer. It was at this point where the complexity of the task started to limit the agents progression through the curriculum. Reliability was a problem when attempting to optimise for speed, the agents performance was inconsistent meaning that on occasion it was able to maintain control and attempt to navigate towards several waypoints as discussed in section 4.1.2, but this was not consistent enough to progress through the curriculum and learn to sail upwind. As a result the model was not able to navigate in any direction. It is likely that this control problem is well within the capabilities of RL and the PPO algorithm, but the complexity of the task necessitated a more realistic model, more complex reward function and better hyperparameter tuning. Objective 4 was not fully achieved, but the agent was able to learn the controls of the kiteboat and navigate towards a target in a straight line.

Objective 5 scored a total of $\frac{26}{40}$, shown in table 4.6. The HPC was used extensively for training a large number of models, but it does not warrant a full score as it was not used with any in depth parallelisation strategies, merely single runs on single nodes. The trained agents were run

Objective 5: Efficacy and optimisation Testing	**Score**
To utilise High-Performance Computing (HPC) resources for scaling up simulations and optimising the training process.	8
To rigorously evaluate the trained agent's performance in simulating autonomous navigation in various environmental scenarios.	8
Outcome Goal(s)	
To train an agent using the HPC resources.	8
To have an agent that can navigate towards a target in a straight line under various environmental conditions, including wind and waves.	2
Total	$\frac{26}{40}$

Table 4.6: Objective 5 Evaluation

in the evaluation scene and compared against human baselines, testing all the learnt behaviours and scoring them appropriately. It would have been exciting to pit multiple agents against one another during the evaluation for a direct comparison, but this remains future work. The agent was able to navigate towards a target in a straight line, and was still able to do so under changing sea states. The wind was not changed during the evaluation as the agent was only trained on laminar wind and never progressed to the stages of changing weather conditions.

Objective 6: Real-World Applicability Assessment	**Score**
To extrapolate simulation findings to assess real-world applicability and propose a practical deployment of RL in kite-powered vessels.	8
To provide recommendations for further research and development based on empirical results obtained from the simulation studies.	8
Total	$\frac{16}{20}$

Table 4.7: Objective 6 Evaluation

Objective 6 scored a total of $\frac{16}{20}$, shown in table 4.7. A practical deployment of the kiteboat was considered in section 4.3, although there is likely to be a significant amount of work to build a physical prototype. Other considerations such as where to test the mvp, and how to ensure safety would need to be addressed. The recommendations for further research and development are discussed in Chapter 5.

After the evaluation of the objectives and their outcome goals a final score of $\frac{141}{180}$ was achieved, which equates to 78.3%. This score is an individual evaluation of the performance of the system against the specified objectives and so is not a direct score of how well the system performs as a whole, merely how close it came to meeting the goals set out at the start of the project.

CHAPTER 5

FUTURE WORK

Simulation Environment

The simulation (training environment), albeit not perfect, was a good representation of the real world, and was a good starting point for the project. In future it would be good to create a more accurate simulation, representing the real world as closely as possible. This would allow for easier integration into a physical system, should the training reach a point where it is ready to be tested in the real world. To create a better simulation, the first aspect to be addressed would be the water model. Remove any water system and create a particle simulation from scratch, this would allow for proper mechanics and realistic water behaviour.

Once the agent has learnt the simpler task of the kiteboat controls, it would be good to start adding in adverse weather conditions. This would be in the form of non laminar wind, i.e. gusty shifty wind, and varying waves. These conditions must be added to the model if the agent is ever going to progress to the point of becoming reliable in the real world, as these are the conditions that the agent will be faced with. Once a reward function has been found that allows the agent to learn the task, it is at this point where the conditions should be added and the agent retrained from scratch; it would be interesting to observe weather this same reward function is still able to learn the task in the new conditions, or if a new reward function is required.

Another part of improving the simulation (the water system), would be to add the ability for water relaunching. LEI kites have the ability to relaunch from the water, and this is a key reason for their popularity in the kiteboarding community. Adding the mechanics for water relaunching into the simulations and training the agent to perform this task would be a good next step, improving the reliability and robustness of the agent. One of the main concerns for the viability autonomously controlled kites is what happens if the kite crashes into the water. It is easy to imagine many dangerous and difficult situations that could arise, and so it is important to try

and mitigate these risks as early as possible to improve the change of success in the real world.

Curriculum

It is easy to imagine the limitless possible combinations of rewards available, and as some of the desired behaviour was observed in the simulation, it is clear that the reward function was on the right lines. However, as explained earlier, due to the complex nature of the challenge at hand, a curriculum was used to split the reward function into smaller, more manageable stages, which proved beneficial but there was room for dissecting the task into even smaller stages. This would have allowed for a more gradual learning process and hopefully given the agent a better chance at retaining what it had learnt in the previous stage. It would also have been good to try multiple different curriculum's to see if the agent could learn the task in a different order.

Agent Architecture

It is worth noting that the pursuit of a single agent that would be capable of full autonomous control may not be the best approach. It may be more beneficial to split the challenge and assign an specific agent to each task, for instance one agent for the kite control working with a hard coded steering algorithm. Another agent could be employed to direct the kite agent into different 'modes', i.e to the left/right, close haul/bare away, based on the heading of the boat. This would massively simplify the task for each agent allowing for a much simpler reward function hopefully producing more reliable and successful results for the specific tasks. If the rudder control is removed from the agent it may be beneficial to use a control algorithm that is more suited to this task, such as a PID controller. Together these agents would be able to control the vessel, however this still has its own challenges, such as how to coordinate the agents and how to train them together for the same application.

5.1 Conclusion

This research successfully demonstrates the feasibility and potential of using reinforcement learning for autonomous control of kite-powered vessels. The project developed a simulation environment that emulated real world conditions and integrated a kite propulsion system with a boat model. Utilising the MLAgents toolkit and Proximal Policy Optimisation (PPO) networks, the RL agent exhibited proficiency in managing the dual aspects of the boat's rudder and the kite's flight mechanics.

However the agent did not manage full autonomous control of the kiteboat and failed to produce the emergent behaviour of maneuvers such as tacking. This shortfall was primarily due to the complexity of the task, the limitations of the simulation environment and the tuning of the RL configuration. Future endeavours should concentrate on improving the simulation environment and refining the model in order to produce more reliable results under more complex

and varying conditions and scenarios. The project successfully demonstrated the ability of an RL agent to learn multiple complex tasks, and provided a solid foundation for future research into the field of autonomous kite-powered vessels. There is a compelling argument to move away from fuel powered vessels and towards more sustainable and environmentally friendly alternatives, and this research has shown that RL could be a viable and promising solution to this problem.

APPENDIX

APPENDIX A

A.1 Final Configuration

```
    behaviors:
  BoatAgent:
    trainer_type: ppo
    hyperparameters:
      batch_size: 256
      buffer_size: 4096
      learning_rate: 3.0e-4
      beta: 5.0e-4
      epsilon: 0.3
      lambd: 0.99
      num_epoch: 4
      learning_rate_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 256
      num_layers: 6
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
      curiosity:
        strength: 0.1
        gamma: 0.99
        learning_rate: 3.0e-4
    max_steps: 30000000
    time_horizon: 64
    summary_freq: 10000
    checkpoint_interval: 50000
    keep_checkpoints: 25
```

A.2 Scripts

A.2.1 Grid Search

```
import itertools
```

```
import os

# Define the ranges for each hyperparameter you want to vary, with fewer options
batch_size_options = [256, 512]
buffer_size_options = [2048, 4096]
learning_rate_options = [1.0e-4, 3.0e-4]
beta_options = [1.0e-4, 5.0e-4]
epsilon_options = [0.2, 0.3]
lambd_options = [0.95, 0.99]
num_epoch_options = [3, 4]
hidden_units_options = [128, 256]
num_layers_options = [4, 5]

# Create a product of all the hyperparameter options
grid_search = list(itertools.product(
    batch_size_options,
    buffer_size_options,
    learning_rate_options,
    beta_options,
    epsilon_options,
    lambd_options,
    num_epoch_options,
    hidden_units_options,
    num_layers_options
))

# If there are more than 100 configurations, randomly sample 100 from them
import random
if len(grid_search) > 100:
    grid_search = random.sample(grid_search, 100)

    # Create a directory for the config files if it doesn't exist
config_directory = "config_files"
os.makedirs(config_directory, exist_ok=True)

# Function to generate the config file content
def generate_config_content(batch_size, buffer_size, learning_rate, beta, epsilon,
lambd, num_epoch, hidden_units, num_layers):
```

```
    return f"""behaviors:
  BoatAgent:
    trainer_type: ppo
    hyperparameters:
    batch_size: {batch_size}
    buffer_size: {buffer_size}
    learning_rate: {learning_rate}
    beta: {beta}
    epsilon: {epsilon}
    lambd: {lambd}
    num_epoch: {num_epoch}
    learning_rate_schedule: constant
    network_settings:
    normalize: false
    hidden_units: {hidden_units}
    num_layers: {num_layers}
    reward_signals:
    extrinsic:
    gamma: 0.99
    strength: 1.0
    curiosity:
    strength: 0.1
    gamma: 0.99
    learning_rate: {learning_rate}
    max_steps: 1000000
    time_horizon: 64
    summary_freq: 10000
    checkpoint_interval: 50000
    keep_checkpoints: 25
    """

    # Generate and save the config files
    for idx, config in enumerate(grid_search):
    batch_size, buffer_size, learning_rate, beta, epsilon, lambd, num_epoch,
    hidden_units, num_layers = config

    config_content = generate_config_content(batch_size, buffer_size,
```

```
    learning_rate, beta, epsilon, lambd, num_epoch, hidden_units, num_layers)

    config_filename = f"config_{idx+1:03d}.yaml"
    config_filepath = os.path.join(config_directory, config_filename)
    with open(config_filepath, 'w') as file:
    file.write(config_content)

    print(f"Generated␣{len(grid_search)}␣configuration␣files␣in␣the␣directory
␣␣␣␣'{config_directory}'.")
```

A.2.2 HPC Shell Script

```
    #!/bin/bash
#SBATCH --account=COSC027924
#SBATCH --job-name=ai_kiteboat
#SBATCH --partition=cpu
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=14
#SBATCH --time=2-10:00:00
#SBATCH --mem=5000M


# 1. Create and activate a Python virtual environment.
python3 -m venv myenv
source myenv/bin/activate
pip install --upgrade pip
pip install pydrive

# 2. Install the ML Agents toolkit.
python -c "
try:
␣␣␣␣import␣mlagents
␣␣␣␣print('ml-agents␣already␣installed')
except␣ImportError:
␣␣␣␣import␣os
␣␣␣␣os.system('pip␣install␣mlagents==0.27.0')
␣␣␣␣print('Installed␣ml-agents')
"
```

```
# 3. Download the config file and Unity executable from Google Drive.

# Direct download links converted from the shared URLs
yaml_link="https://drive.google.com/uc?export=download&id=..."
creds_link="https://drive.google.com/uc?export=download&id=..."
setting_link="https://drive.google.com/uc?export=download&id=..."

# navigate to the directory where the files will be downloaded ~/ML_PPO
mkdir ML_PPO
cd ~/ML_PPO

wget -O trainer_config.yaml $yaml_link
wget -O credentials.json $creds_link
wget -O settings.yaml $setting_link

fileid="..."
filename="build.zip"
query=`curl -c ./cookie.txt -s -L "https://drive.google.com/uc?export=download&id=
${fileid}" |\
     perl -nE'say/confirm=(\w+)/'`
curl -Lb ./cookie.txt "https://drive.google.com/uc?export=download&confirm=${query}&id=
{fileid}" -o ${filename}


# Unzip the Unity build folder
unzip  -o build.zip

# 4. Run the Unity executable and train the ML Agents.
env_name="./HeadlessBuilds/build.x86_64"
trainer_config_file="./trainer_config.yaml"
run_identifier="test_1"

chmod +x $env_name
chmod +x $trainer_config_file
chmod +x HeadlessBuilds

mkdir -p results
```

```
python -c "

from mlagents_envs.environment import UnityEnvironment
env = UnityEnvironment(file_name='$env_name', no_graphics=True)

env.reset()

behavior_name = list(env.behavior_specs)[0]
spec = env.behavior_specs[behavior_name]


max_steps = 1000  # Adjust this value based on your desired max steps
current_steps = 0

while current_steps < max_steps:
    env.reset()
    decision_steps, terminal_steps = env.get_steps(behavior_name)
    tracked_agent = -1 # -1 indicates not yet tracking
    done = False # For the tracked_agent
    episode_rewards = 0 # For the tracked_agent
    while not done:
        # Track the first agent we see if not tracking
        if tracked_agent == -1 and len(decision_steps) >= 1:
            tracked_agent = decision_steps.agent_id[0]

        # Generate an action for all agents
        action = spec.action_spec.random_action(len(decision_steps))

        # Set the actions
        env.set_actions(behavior_name, action)

        # Move the simulation forward
        env.step()
        current_steps += 1

        # Get the new simulation results
        decision_steps, terminal_steps = env.get_steps(behavior_name)
        if tracked_agent in decision_steps:  # The agent requested a decision
```

```
            episode_rewards += decision_steps[tracked_agent].reward
        if tracked_agent in terminal_steps:  # The agent terminated its episode
            episode_rewards += terminal_steps[tracked_agent].reward
            done = True
    print('Total rewards until now: ' + str(episode_rewards))


env.close()
print('Closed environment')
"
# ls -la

# 5. Zip the results file and upload it to Google Drive.
results_folder="results"
zip_file="${results_folder}.zip"
zip -r $zip_file $results_folder


python -c "
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive

# Authentication
gauth = GoogleAuth()
# gauth.LoadCredentialsFile('credentials.json')
gauth.CommandLineAuth()

if gauth.credentials is None:
    print('Missing credentials.json')
    exit(1)

drive = GoogleDrive(gauth)

# Upload the zipped results file
file_drive = drive.CreateFile({'title': '$zip_file'})
file_drive.SetContentFile('$zip_file')
file_drive.Upload()
# print('Uploaded ' + zip_file + ' to Google Drive')
```

"

A.2.3 Extract Results

```
import os
import pandas as pd
from tensorboard.backend.event_processing import event_accumulator

# Directory where the ML-Agents results are stored
results_directory = 'results'

# Function to extract scalar data from a single event file
def extract_scalar_data_from_event_file(event_file_path):
    # Create an accumulator to collect the data from the event file
    ea = event_accumulator.EventAccumulator(event_file_path)
    ea.Reload()
    print(f'Extracting data from {event_file_path}')

    # Extract scalar data
    scalar_data = {}
    for tag in ea.Tags()['scalars']:
        events = ea.Scalars(tag)
        scalar_data[tag] = [(e.step, e.value) for e in events]

    return scalar_data

# Function to convert the extracted data into a pandas DataFrame and format it for rea
def scalar_data_to_dataframe(scalar_data):
    dataframes = {}
    for tag, values in scalar_data.items():
        df = pd.DataFrame(values, columns=['Step', 'Value'])
        df['Tag'] = tag
        # Format the 'Value' column to round to 4 decimal places for readability
        df['Value'] = df['Value'].round(4)
        dataframes[tag] = df

    if dataframes:
        # Concatenate all dataframes and sort by 'Tag' and 'Step' for better readabili
        return pd.concat(dataframes.values(), ignore_index=True).sort_values(by=['Tag'
```

```
        else:
            return pd.DataFrame(columns=['Step', 'Value', 'Tag'])

    # Find all event files in the results directory
    event_files = [os.path.join(dirpath, f)
                   for dirpath, dirnames, files in os.walk(results_directory)
                   for f in files if f.startswith('events.out.tfevents')]

    # Process each event file and save the results to a CSV file
    for event_file in event_files:
        scalar_data = extract_scalar_data_from_event_file(event_file)

        if scalar_data:
            df = scalar_data_to_dataframe(scalar_data)
            # Determine the directory for the current event file
            run_directory = os.path.dirname(os.path.dirname(event_file))
            # Save the DataFrame to a CSV file in the run directory
            csv_file_path = os.path.join(run_directory, 'results.csv')
            df.to_csv(csv_file_path, index=False)
            print(f'Results saved to {csv_file_path}')
        else:
            print(f'No scalar data found in {event_file}')
```

A.2.4 Combine Results

```
  import pandas as pd
import os
import yaml

results_dir = 'results_100x2'
combined_data = []

# Specify the nested keys for the varying parameters you want to extract
varying_params = {
    'batch_size': ['behaviors', 'BoatAgent', 'hyperparameters', 'batch_size'],
    'buffer_size': ['behaviors', 'BoatAgent', 'hyperparameters', 'buffer_size'],
    'learning_rate': ['behaviors', 'BoatAgent', 'hyperparameters', 'learning_rate'],
    'beta': ['behaviors', 'BoatAgent', 'hyperparameters', 'beta'],
    'epsilon': ['behaviors', 'BoatAgent', 'hyperparameters', 'epsilon'],
```

```
    'lambd': ['behaviors', 'BoatAgent', 'hyperparameters', 'lambd'],
    'num_epoch': ['behaviors', 'BoatAgent', 'hyperparameters', 'num_epoch'],
    'hidden_units': ['behaviors', 'BoatAgent', 'network_settings', 'hidden_units'],
    'num_layers': ['behaviors', 'BoatAgent', 'network_settings', 'num_layers'],
    'curiosity_strength': ['behaviors', 'BoatAgent', 'reward_signals', 'curiosity', 'str
    'curiosity_learning_rate': ['behaviors', 'BoatAgent', 'reward_signals', 'curiosity',
}

def get_nested_value(dct, keys):
    for key in keys:
        dct = dct.get(key, {})
    return dct if isinstance(dct, (int, float)) else None

# Loop through each subdirectory in the results directory
for model_dir in os.listdir(results_dir):
    model_path = os.path.join(results_dir, model_dir)
    if os.path.isdir(model_path):
        config_path = os.path.join(model_path, 'configuration.yaml')
        results_path = os.path.join(model_path, 'results.csv')

        with open(config_path, 'r') as yaml_file:
            config = yaml.safe_load(yaml_file)

        # Extract only the varying parameters using the nested keys
        config_values = {param: get_nested_value(config, keys) for param, keys in varying

        # Check if the results.csv exists before trying to read it
        if os.path.exists(results_path):
            df_results = pd.read_csv(results_path)
            df_episode_length = df_results[df_results['Tag'] == 'Environment/Episode Len
            best_episode_length = df_episode_length['Value'].max()
            df_av_loss = df_results[df_results['Tag'] == 'Losses/Value Loss']
            av_loss = df_av_loss['Value'].mean()
            df_av_curiosity = df_results[df_results['Tag'] == 'Policy/Curiosity']
            av_curiosity = df_av_curiosity['Value'].mean()
            df_av_reward = df_results[df_results['Tag'] == 'Policy/Extrinsic Reward']
            av_reward = df_av_reward['Value'].mean()
```

```
        else:
            best_episode_length = float('inf')  # Use infinity to represent missing or u

        combined_data.append({
            'Model': model_dir,
            **config_values,
            'Best Episode Length': best_episode_length,
            'Av Loss': av_loss,
            'Av Curiosity': av_curiosity,
            'Av Reward': av_reward
        })

# Convert the combined data to a DataFrame
df_combined = pd.DataFrame(combined_data)

# Convert 'Best Episode Length' to numeric, setting errors='coerce' to handle non-numeri
df_combined['Best Episode Length'] = pd.to_numeric(df_combined['Best Episode Length'], e

# Drop rows where 'Best Episode Length' could not be converted to a number
df_combined.dropna(subset=['Best Episode Length'], inplace=True)

# Sort by 'Best Episode Length'
df_combined.sort_values('Best Episode Length', ascending=False, inplace=True)

# Save the sorted DataFrame to a CSV file
output_csv_path = os.path.join(results_dir, 'combined_results.csv')
df_combined.to_csv(output_csv_path, index=False)

print(f'Combined CSV file created at: {output_csv_path}')
```

A.2.5 Analysis

```
  import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import pandas as pd
```

```
# Load the dataset
data = pd.read_csv('combined_results.csv')

# Define the features (config parameters) and the target (Best Episode Length)
features = data.drop(columns=['Model', 'Best Episode Length', 'Av Loss', 'Av Curiosity',
target = data['Best Episode Length']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.3, ran

# Initialize the Random Forest Regressor
rf = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the model
rf.fit(X_train, y_train)

# Predict on the test set
y_pred = rf.predict(X_test)

# Calculate the feature importances
feature_importances = rf.feature_importances_

# Calculate the Mean Squared Error on the test set
mse = mean_squared_error(y_test, y_pred)

# Visualize the feature importances
sns.barplot(x=feature_importances, y=features.columns)
plt.xlabel('Importance')
plt.ylabel('Features')
# plt.savefig('feature_importances.png', dpi=300)
plt.show()

# Scatter plot matrix for the most correlated variables
most_correlated = data[['Best Episode Length', 'num_epoch', 'buffer_size', 'batch_size',
sns.pairplot(most_correlated)
plt.suptitle('Scatter plot matrix for the most correlated variables', y=1.02)
# y=1.02 to adjust the title position
plt.savefig('scatter_plot_matrix.png', dpi=300)
```

```
mse, feature_importances
print(mse)

statistical_summary = data.describe()

# Computing the correlation matrix to see the relationship between parameters and perfor

correlation_matrix = data.corr(method='spearman')

(statistical_summary, correlation_matrix['Best Episode Length'].sort_values(ascending=Fa
print(correlation_matrix['Best Episode Length'].sort_values(ascending=False))
```

BIBLIOGRAPHY

[1] G. Ferentinos, M. Gkioni, M. Geraga, and G. Papatheodorou.
Early seafaring activity in the southern ionian islands, mediterranean sea.
Journal of Archaeological Science, 39(7):2167–2176, 2012.
doi: 10.1016/J.JAS.2012.01.032.
URL https://dx.doi.org/10.1016/J.JAS.2012.01.032.

[2] Lionel Casson.
Ships and Seamanship in the Ancient World.
Princeton University Press, 1995.

[3] Basil Greenhill.
The Advent of Steam - The Merchant Steamship before 1900.
Conway Maritime Press Ltd., 1993.

[4] J.J. Corbett et al.
Mortality from ship emissions: A global assessment.
Environmental Science & Technology, 41(24):8512–8518, 2007.

[5] M. Vahs.
Retrofitting of flettner rotors – results from sea trials of the general cargo ship "fehn pollux".
Proceedings of the Royal Institution of Naval Architects - Part A: International Journal of Maritime Engineering, 2020(A4):641, 2019.
doi: https://www.intmaritimeengineering.org/index.php/ijme/article/view/1146.

[6] Silent Yatchs.
Silent yatchs, 2023.
URL https://www.silent-yachts.com/silent60/.
Accessed: 2023-11-20.

[7] Core Kites.
Core kites, 2023.
URL https://ridecore.com/us.
Accessed: 2023-11-20.

[8] University of Bristol.
ACRC - Advanced Computing Research Center. 2023.
Retrieved 30/10/2023 from https://www.bristol.ac.uk/acrc/.

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

[10] Christopher J Watkins and Peter Dayan.
Q-learning.
Machine learning, 8(3-4):279–292, 1992.
URL https://doi.org/10.1007/BF00992698.

[11] Richard Bellman.
Dynamic Programming.
Princeton University Press, 1957.

[12] David Silver et al.
Mastering the game of go with deep neural networks and tree search.
Nature, 529(7587):484–489, 2016.
URL https://www.nature.com/articles/nature16961.

[13] Michael Erhard and Hans Strauch.
Control of towing kites for seagoing vessels.
IEEE Transactions on Control Systems Technology, 21(5):1629–1640, 2013.
URL https://arxiv.org/pdf/1202.3641.pdf.

[14] Richard Hallion.
Taking flight: inventing the aerial age, from antiquity through the First World War.
Oxford University Press, 2003.

[15] Uwe Fechner.
A Methodology for the Design of Kite-Power Control Systems.
PhD thesis, Delft University of Technology, 2016.

[16] Marielle Christiansen, Kjetil Fagerholt, and David Ronen.
Ship routing and scheduling in the new millennium.
European Journal of Operational Research, 228(3):467–483, 2013.

[17] Volodymyr Mnih et al.
Human-level control through deep reinforcement learning.
Nature, 518(7540):529–533, 2015.

[18] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov.
Proximal policy optimization algorithms.
arXiv preprint arXiv:1707.06347, 2017.
URL https://arxiv.org/pdf/1707.06347.pdf?fbclid=IwAR1DwRSkBzhqXRhVh3XQ_Nu_XYvN_sMR4ppYM28h3qAtx9EK03LrlOcF7Dg.

[19] Aleksandr Beznosikov, Eduard Gorbunov, Hugo Berard, and Nicolas Loizou.
Stochastic gradient descent-ascent: Unified theory and new efficient methods, 2023.

[20] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry.
Implementation matters in deep rl: A case study on ppo and trpo.
In *International Conference on Learning Representations*, 2020.
URL https://openreview.net/forum?id=r1etN1rtPB.

[21] TN Larsen, HØ Teigen, T Laache, D Varagnolo, and A Rasheed.
Comparing deep reinforcement learning algorithms' ability to safely navigate challenging waters.
Frontiers in Robotics and AI, 8:738113, 2021.
doi: 10.3389/frobt.2021.738113.
URL https://www.frontiersin.org/articles/10.3389/frobt.2021.738113/full.

[22] miyamotok0105.
unity-ml-agents.
GitHub repository, 2018.
URL https://github.com/miyamotok0105/unity-ml-agents/blob/master/docs/Training-PPO.md.

[23] Merlin.
Unity in 100 seconds, 2023.
URL https://www.youtube.com/watch?v=iqlH4okiQqg.

[24] Unity Technologies.
Unity User Manual, 2021.

[25] Will Goldstone.
Unity 3.x Game Development Essentials.
Packt Publishing Ltd, 2010.

[26] Ryan Henson Creighton.
Unity 3D Game Development by Example Beginner's Guide.
Packt Publishing Ltd, 2010.

[27] Unity Technologies.
Unity scripting api: Update and fixedupdate, 2022.
URL https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html.

[28] Wingit.
Wingit, 2023.
URL https://www.kite-boat.com/en/.
Accessed: 2023-11-20.

[29] Yves Parlier.
Beyond the sea, 2023.
URL https://beyond-the-sea.com/en/seakite/.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al.
Pytorch: An imperative style, high-performance deep learning library.
In *Advances in Neural Information Processing Systems*, 2019.
URL https://ar5iv.org/abs/1912.01703.

[31] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi.
A fast procedure for computing the distance between complex objects in three-dimensional space.
IEEE Journal on Robotics and Automation, 1988.
The seminal work on the GJK collision detection algorithm.

[32] Yoshua Bengio, Jerome Louradour, Ronan Collobert, and Jason Weston.
Curriculum learning, 2009.
URL https://qmro.qmul.ac.uk/xmlui/handle/123456789/15972.
Accessed: 2023-11-20.

[33] VMG.
Vmg.
Dahlberg, 2023.
URL https://www.dahlberg-sa.com/en/sailing-what-is-vmg/.
Accessed: 2023-11-20.

[34] IBM.
What is random forest?
dfssd, 2023.
URL https://www.ibm.com/topics/random-forest.