12-2023

Collaborative Task Completion for Simulated Hexapod Robots Using Reinforcement Learning

Tayler Don Baker
Utah State University, a01634939@usu.edu

COLLABORATIVE TASK COMPLETION FOR SIMULATED HEXAPOD ROBOTS USING REINFORCEMENT LEARNING

by

Tayler Don Baker

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Mario Harper, Ph.D. Major Professor	Steve Petruzza, Ph.D. Committee Member
Greg Droge, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2023

ABSTRACT

Collaborative Task Completion for Simulated Hexapod Robots Using Reinforcement Learning

by

Tayler Don Baker, Master of Science
Utah State University, 2023

Major Professor: Mario Harper, Ph.D.
Department: Computer Science

There is growing interest in developing autonomous systems capable of exhibiting collaborative behaviors. Using methods such as reinforcement learning is another way to train multiple robots for collaborative task completion. This research talks about two reinforcement learning algorithms such as PPO and MA-POCA and uses them to train multiple hexapod robots to push a target to a designated goal collaboratively. This required each robot to learn how find the target and push that target to a goal. It shows the results of the simulations and talks about conclusions and future work. This work suggests that using reinforcement learning for collaborative task completion for hexapod robots may simplify the complexity of the software and improve the decisions that they make.

(56 pages)

PUBLIC ABSTRACT

Collaborative Task Completion for Simulated Hexapod Robots Using Reinforcement Learning

Tayler Don Baker

There is growing interest in developing autonomous systems capable of exhibiting collaborative behaviors. Using methods such as reinforcement learning is another way to train multiple robots for collaborative task completion. This study was able to successfully in simulation train multiple hexapod robots to push a target to a designated goal collaboratively. This required each robot to learn how find the target and push that target to a goal. This work suggests that using reinforcement learning for collaborative task completion for hexapod robots may simplify the complexity of the software and improve the decisions that they make.

ACKNOWLEDGMENTS

I would like to thank all of the people that participated in this research. I'd like to thank my research advisor Dr. Mario Harper for allowing me to do this research under him. I would also like to thank him for all of the knowledge, guidance, and mentorship he provided for me along the way. I would like to thank my committee members Dr. Steve Petruzza and Dr. Greg Droge for their willingness to be part of my research and their added guidance and support. I would like to thank my dad and mom, Donald Baker and Claudia B. Baker, for their support for my continued education and for instilling into me a strong work ethic and desire to seek truth. I also recognize Utah State University and the computer science department and faculty as part of my success. Lastly, I thank everyone else who has supported me in my research and educational pursuits.

Tayler D. Baker

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

RL	Reinforcement Learning
PPO	Proximal Policy Optimization
MA-POCA	Multi-Agent Posthumous Credit Assignment
PG	Policy Gradient
TRPO	Trust Region Policy Optimization
MARL	Multi-Agent Reinforcement Learning
COMA	Counterfactual Multi-Agent
USV	Unmanned Vessel
NN	Neural Network
UAV	Unmanned Aerial Vehicle
API	Application Programming Interface
SDK	Software Development Kit
DOF	Degrees of Freedom

CHAPTER 1
INTRODUCTION

There is growing interest in developing autonomous systems capable of exhibiting collaborative behaviors. These systems have the potential to greatly improve robotics by simplifying intelligence architectures currently used in search and rescue operations, environmental monitoring, and industrial automation. One particular group of robots that can benefit significantly from collaborative intelligence are ant-like hexapedal robots, individually limited but capable of group action. Inspired by the sophisticated coordination and cooperation observed in ant colonies, these robots possess unique locomotion capabilities and can navigate complex terrains.

The successful implementation of collaborative behaviors in hexapedal ant-like robots requires the adoption of appropriate learning algorithms. Among the various approaches available, RL has emerged as a promising technique to achieve the desired objectives. RL enables robots to learn from their interactions with the environment by receiving feedback in the form of rewards or penalties, allowing them to make informed decisions and improve their performance over time. There are several compelling reasons why leveraging RL is advantageous over other methods when it comes to building collaborative behaviors for hexapedal ant-like robots.

Firstly, RL offers a versatile framework that can handle a wide range of complex tasks. Hexapedal ant-like robots often operate in dynamic and uncertain environments, where traditional rule-based approaches may struggle to provide effective solutions. RL algorithms, on the other hand, are capable of adapting and learning from experience, making them well-suited for handling the inherent complexity and uncertainty associated with collaborative behaviors.

Secondly, RL allows for decentralized decision-making and coordination among multiple robots. In the context of collaborative behaviors, it is crucial to enable individual robots to make autonomous decisions while maintaining a collective goal or objective. RL algorithms provide a distributed learning mechanism, allowing each robot to learn its own policy based on local observations and rewards, while also facilitating coordination and cooperation among the robots towards achieving a common goal.

Furthermore, RL has the potential to facilitate the emergence of novel and adaptive strategies. Traditional programming approaches often require extensive manual engineering and may not capture the full complexity of collaborative behaviors. In contrast, RL enables robots to explore and discover new strategies through trial and error, leading to the emergence of innovative and adaptive behaviors that can enhance the overall performance and efficiency of the collaborative system.

Lastly, leveraging RL can enable continuous improvement and lifelong learning in hexapedal ant-like robots. As the robots interact with the environment and learn from their experiences, they can continuously refine their behaviors and adapt to changing conditions. This ability to learn and improve over time is crucial in scenarios where the robots are deployed in dynamic and evolving environments, as it allows them to cope with new challenges and unforeseen situations.

This thesis explores a promising approach for building collaborative behaviors in hexapedal ant-like robots. By enabling adaptability, decentralized decision-making, and the emergence of novel strategies, RL empowers these robots to navigate complex environments, work together towards common goals, and continuously improve their performance. This thesis aims to explore RL techniques specifically tailored for collaborative hexapedal ant-like robots, paving the way for more sophisticated and efficient collaborative robotic systems.

RL follows the idea that an autonomously acting agent obtains its behavior policy through repeated interaction with its environment on a trial-and-error basis [1]. The purpose of RL is there is an agent that is aware of the state of environment at time t, the agent takes an action at t which leads to state $t + 1$, where the agent receives a reward at Rt. As illustrated in Figure 1.1 this cyclical model encourages the agent to take actions to maximize the reward given.

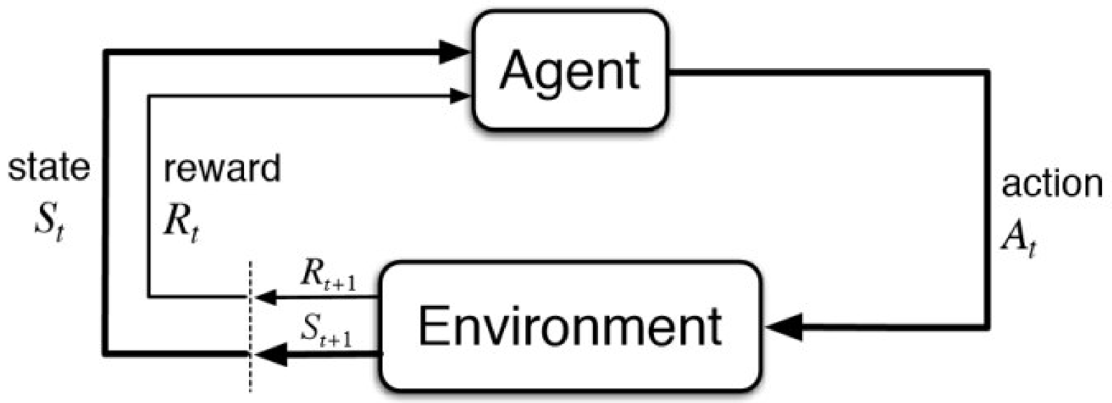


Fig. 1.1: RL Cycle [1]

As seen RL gives the option of instead of designing a software to control multiple hexapod robots it is possible to design a higher-level task description in the form of the reward [6].

The higher level task of this paper is to have two hexapod robots in simulation collaboratively push a target to a goal. The hexapod robots are similar to the A-Pod robot by Lynxmotion as illustrated in Figure 1.2 is a six-legged robot with a claw that can open and close so that it can grasp objects. This was done so that in future work the same RL that was done on the simulated model could be put on a similar physical robot.

This paper goes as follows: We first talk about the different RL algorithms that were used in this research. We then discuss related work of what has already been done in the research of multi agent systems. Next we discuss the simulation tools that we used. I talk about the preliminary work that was done to learn about the different RL algorithms with the results of the training. Next we talk about the methods of this research and how it was set up. After we discuss the results of the research. We then give a conclusion and lessons learned about the research. Finally we talk about possible future work needed for this research.

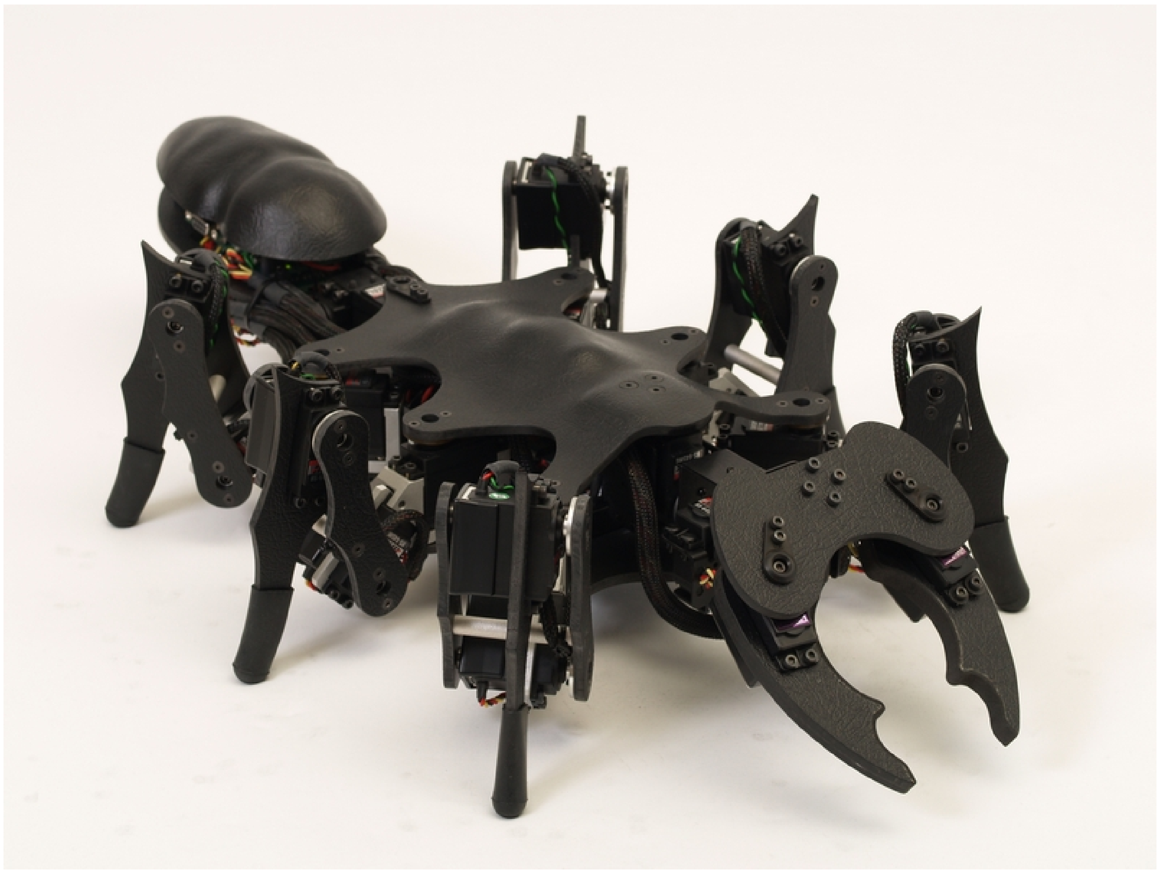


Fig. 1.2: A-Pod Robot by Lynxmotion [2]

CHAPTER 2
BACKGROUND

Many RL frameworks rely on policy gradient methods. PG methods are a class of RL algorithms that directly optimize the policy of an agent in order to maximize its cumulative reward. Unlike value-based methods that estimate the value function, policy gradient methods learn a parameterized policy that maps states to actions. By iteratively updating the policy based on the observed rewards, these methods enable the agent to improve its decision-making over time. Policy gradient methods offer advantages such as the ability to handle continuous action spaces and the potential for end-to-end learning. They have been successfully applied to various domains, including robotics, where they have shown promise in training agents to perform complex tasks and exhibit adaptive and collaborative behaviors.

In RL, algorithms based on policy gradient provide an outstanding paradigm for continuous action space problems. The purpose of all such algorithms is to maximize the cumulative expected rewards [7]. The two RL algorithms were used in this paper: PPO and MA-POCA.

2.0.1 PPO

Many RL frameworks rely on policy gradient methods. Policy gradient methods are a class of RL algorithms that directly optimize the policy of an agent in order to maximize its cumulative reward. Unlike value-based methods that estimate the value function, policy gradient methods learn a parameterized policy that maps states to actions. By iteratively updating the policy based on the observed rewards, these methods enable the agent to improve its decision-making over time. Policy gradient methods offer advantages such as the ability to handle continuous action spaces and the potential for end-to-end learning. They have been successfully applied to various domains, including robotics, where they have shown promise in training agents to perform complex tasks and exhibit adaptive and collaborative behaviors.

The PPO algorithm is a particular form of PG-RL which has emerged as a popular RL technique. PPO addresses some of the limitations of previous policy gradient methods by providing a stable and efficient approach for training complex robotic systems. Its ability to handle high-dimensional action spaces and continuous control tasks makes it particularly relevant for the development of collaborative behaviors in hexapedal ant-like robots.

Four particular benefits of PPO are:

- Sample Efficiency: Traditional policy gradient methods often suffer from poor sample efficiency. Since these methods update the policy based on Monte Carlo estimates of the expected returns, they require a large number of interactions with the environment to achieve good performance. PPO addresses this limitation by utilizing the collected data more effectively. It uses a surrogate objective function that constrains the policy update within a certain range, ensuring more stable and efficient updates. This leads to better sample efficiency and reduces the number of interactions needed to learn an effective policy.

- Policy Divergence: Another limitation of policy gradient methods is the potential for policy divergence during training. As the policy is updated based on the current estimate of expected returns, large updates can cause the policy to deviate significantly, leading to instability and poor performance. PPO mitigates this issue by employing a clipping mechanism in the surrogate objective function. By constraining the policy update to a range around the current policy, PPO prevents drastic policy changes that could lead to divergence. This clipping mechanism ensures more stable updates and helps maintain policy convergence.

- Exploration-Exploitation Trade-off: Balancing exploration and exploitation is crucial for effective learning in RL. Traditional policy gradient methods often struggle with this trade-off, as they can get trapped in suboptimal policies or fail to explore promising regions of the action space. PPO addresses this limitation by leveraging the importance sampling ratio and the clipped surrogate objective function. These techniques encourage exploration by allowing the policy to explore actions with higher probabilities and discouraging excessive exploitation. This way, PPO achieves a balance between exploration and exploitation, facilitating the discovery of optimal policies.

- High-Dimensional Action Spaces: Many real-world robotic systems, including hexapedal ant-like robots, have high-dimensional action spaces. Traditional policy gradient methods can face challenges in effectively exploring and learning in such spaces. PPO tackles this limitation by providing stable and efficient updates even in high-dimensional action spaces. Its surrogate objective function and the use of a trust region ensure that the policy updates remain within a reasonable range, allowing effective learning in complex action spaces.

PPO enhances the performance and stability of policy gradient methods, making it a valuable technique for training robotic systems, including hexapedal ant-like robots, to exhibit sophisticated and collaborative behaviors.

PPO is well-suited for the hexapedal ant-like robots described above for several reasons. Firstly, these robots possess a large action space, as their locomotion involves coordinated movements of multiple limbs. Traditional RL algorithms may struggle to handle such high-dimensional action spaces effectively. PPO, however, uses a surrogate objective function that ensures efficient exploration and stable updates, making it suitable for learning complex locomotion policies for these robots.

Secondly, PPO offers sample efficiency, which is crucial when dealing with resource-constrained robotic systems. Training hexapedal ant-like robots through physical interactions can be time-consuming and expensive. PPO's ability to extract more information from each interaction and effectively utilize collected data leads to faster and more efficient learning, reducing the number of interactions required to achieve desirable collaborative behaviors.

Furthermore, PPO provides a balance between exploration and exploitation, allowing the hexapedal ant-like robots to discover optimal coordination strategies while avoiding premature convergence to suboptimal policies. The ability to strike this balance is critical in developing robust and adaptive collaborative behaviors, as the robots need to explore a wide range of behaviors and learn from their interactions to achieve effective coordination and cooperation.

PPO is a policy gradient algorithm that learns a parameterized policy that attractively updates the parameters of the policy by solving a local optimization problem [8]. PPO is built off of the algorithm TRPO but has unconstrained surrogate objective function and generalized advantage estimation [9].

The objective of the unconstrained surrogate objective function is shown as

$$J^{\text{ppo}}(\theta) = \mathbb{E}_{s,a}\left[\min\left(\rho_\theta A^{\pi_{\theta_{\text{old}}}}, \text{clip}\left(\rho_\theta, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_{\text{old}}}}\right)\right]$$

where $p_\theta = \frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)}$ When $A^{\pi_{\theta old}} > 0$, the term p_θ would tend to be much larger than 1 to make the performance as high as possible, which leads to unstable learning. The objective of PPO cuts this motivation by clipping p with $1 + \epsilon$. Same situation is with $A^{\pi_{\theta old}} < 0$ [9].

The advantage function for the PPO policy gradient can be estimated by

$$\hat{A}^\pi(s,a) = \hat{Q}^\pi(s,a) - V(s,w),$$

where $\hat{Q}^\pi(s,a)$ is the action-value function estimated by samples, $V(s,w)$ is the approximation of the state-value function. TRPO used Monte Carlo methods to construct $\hat{Q}^\pi(s,a)$, i.e., [9]

$$\hat{Q}^\pi(s_t,a_t) = \sum_{l=t}^{\infty} \gamma^{l-t} r_l.$$

It is unbiased but suffers from high variance. Actor-critic methods use one-step bootstrapping to form $\hat{Q}^\pi(s,a)$, i.e., [9]

$$\hat{Q}^{\pi}(s_t, a_t) = r_t + \gamma V(s_{t+1}, w),$$

this is biased but has a low variance. Generalized advantage estimation uses the linear combination on n-step bootstrapping to obtain both low bias and low variance, which is shown as [9]

$$\hat{Q}^{\pi}(s_t, a_t) = \sum_{l=t}^{\infty} (\gamma\lambda)^{l-t} \delta_l + V(s_t, w),$$

where δ_l is the TD error,

$$\delta_l = r_l + \gamma V(s_{l+1}, w) - V(s_l, w).$$

Compared with TRPO, PPO is much simpler and faster to implement because it is a first-order optimization algorithm and has better convergence speed when it is combined with generalized advantage estimation. However, PPO is an on-policy method and inevitably has high sample complexity [9]. PPO has some benefits of TRPO but is much simple to implement as illustrated in Figure 2.1, and has better sample complexity [3]

Algorithm 1 PPO, Actor-Critic Style

```
for iteration=$1, 2, \ldots$ do
    for actor=$1, 2, \ldots, N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Fig. 2.1: PPO Algorithm [3]

2.0.2 MA-POCA

MARL is a subfield of RL that focuses on developing algorithms and techniques to enable multiple agents to learn and interact in a shared environment. Unlike single-agent

RL, where a single agent learns from its own experiences, MARL tackles the challenges of coordinating and learning in the presence of other agents with their own goals and policies. MARL plays a crucial role in domains such as robotics, game theory, and multi-robot systems, where collaborative behaviors, competition, or coordination among multiple agents are essential. By leveraging MARL, agents can learn to adapt, strategize, and cooperate, leading to more sophisticated and intelligent decision-making in complex, dynamic environments.

Posthumous credit assignment is an approach in which rewards are allocated to past actions based on their contribution to the final outcome, even if their impact is not immediately observable. In the context of hexapedal ant-robots building a cooperative task intelligence, posthumous credit assignment can provide several benefits compared to using the PPO algorithm alone.

Firstly, in a cooperative task where multiple ant-like robots work together, the individual actions of each robot may not yield immediate observable benefits or rewards. For example, one robot may clear an obstacle that allows another robot to access a valuable resource. With posthumous credit assignment, the delayed rewards associated with such actions can be appropriately attributed, allowing the robots to understand the long-term consequences of their decisions and learn to collaborate more effectively.

Secondly, hexapedal ant-robots often operate in complex and dynamic environments where the outcome of a cooperative task may be influenced by a combination of actions from multiple robots. Posthumous credit assignment enables the robots to analyze the joint effect of their actions, fostering coordination and collaboration. By assigning rewards retrospectively based on the collective outcome, the robots can learn to prioritize and adapt their behaviors to achieve better cooperative task performance.

Furthermore, posthumous credit assignment facilitates the learning of strategic and coordinated behaviors. By explicitly accounting for the delayed effects of actions, the robots can develop an understanding of how their behaviors interact and influence the overall task execution. This can lead to the emergence of sophisticated cooperative strategies and enhance the overall intelligence of the collaborative system.

While PPO provides a powerful framework for RL, it may face challenges in explicitly capturing the delayed and cumulative effects of actions in cooperative tasks. Posthumous credit assignment complements PPO by providing a mechanism to allocate rewards retrospectively, enabling the hexapedal ant-robots to build a cooperative task intelligence by understanding and optimizing their long-term contributions to the task's success.

One of the challenges of MARL is the early agent terminations. This early termination is a problem because terminated agents generally does not learn from its fellow agents. This is referred to as the Posthumous Credit Assignment problem. Many MARL methods solve these problems by placing terminated agents in an absorbing state until all the agents complete the task or are terminated [10]. This solution is both costly and ineffective since agents do not share rewards which in turn increases training time.

Unity found a solution by developing its own MARL, MA-POCA which naturally handles agents that are created or destroyed within an episode but share a reward function. It can scale to an arbitrary number of agents [10]. According to the ML-Agents documentation, MA-POCA will allow to give rewards to the agents collectively as a group. This will help the agents learn best how to contribute to achieving that award [4]. MA-POCA trains a centralized critic that acts as a trainer for all of the agents, instead of individual rewards to individual agents, it is possible to reward the whole team of agents teaching them to act in the best interest of the team rather than individually [11].

To handle a varying number of agents per timestep, first encode the observations of all active agents $g_1(o_t^i)_{1\leq\ i\leq\ k_t}$ where o_t^i is the observation of agent i at time t. These encodings then pass through an RSA block. The RSA block is similar to the one used in vanilla Transformer architecture [12] but without positional encodings [13]. The centralized state value function is parameterized by ϕ has the form [10]

$$V_\phi(RSA(g_i(o_t^i)_{1\leq\ i\leq\ k_t}))$$

and is trained with TD(λ) [14]

$$J(\phi) = (V_\phi(RSA(g_i(o_t^i)_{1\leq\ i\leq\ k_t})) - y^{(\lambda)})^2$$

where

$$y^{(\lambda)} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)}$$

$$G_t^{(n)} = \sum_{n=1}^{\infty}\lambda^{\iota-1}r_{t+\iota} + \lambda^n V_\phi(RSA(g_i(o_t^i)_{1\leq\ i\leq\ k_{t+n}}))$$

Where k_{t+n} is the number of agents active at the time t + n. k_{t+n} can be greater or less than k_t as agents can spawn and despawn at time step t, this is how expected value from t + n can propagate back to agents that terminated at time t [10].

Through the MA-POCA Counterfactual Baseline agent j learns the value functions based on the observation of agents i. The baseline parameterized by ψ for agent j has the form [10]

$$Q_\psi(RSA(g_j(o_t^j), f_i(o_t^i, a_t^i)1 \leq i_{i\neq j} \leq k_t))$$

The objective for the baseline is

$$J(\psi) = Q_\psi(RSA(g_j(o_t^j), f_i(o_t^i, a_t^i)1 \leq i_{i\neq j} \leq k_t)) - y^{(\lambda)})^2$$

The advantage for agent j to be used in the update is given by

$$Adv_j = y^{(\lambda)} - Q_\psi(RSA(g_j(o_t^j), f_i(o_t^i, a_t^i)1 \leq i_{i\neq j} \leq k_t))$$

MA-POCA was found to perform as well as or slightly better than COMA and PPO for tasks without dying or spawning, and significantly outperform both methods for tasks where agents die and/or spawn [10].

CHAPTER 3
RELATED WORK

A significant amount of work has been done already in the field of multi agent systems using RL to complete tasks. Such methods that have been explored have been where agents coordinate with its neighbors and follow a leader, sever implementation, and actor-critic solutions.

Such an example of research is by researchers [15] where through simulation they had multiple virtual robot formation system which consisted of one USV and three USV followers [15]. These researchers implemented two layers of learning to accomplish their objective. The followers learn the behaviors of the virtual objects in the first layer based on the adaptive NNs. Then another layer realizes the tracking performance between the virtual objects and the leader by introducing the local data-based performance index [15]. Through these two layers each agent could only communicate with its neighboring agents and were required to complete formation tasks by following their leader USV's trajectory. One flaw of this type of method is that in the researchers conclusions they found that due to tracking errors the followers had to be close enough to the leader or else they got confused and off course.

Another example of multi agent learning is by researchers [16] who used batch RL tailored to the use of multilayer perceptrons to approximate value functions over continuous state spaces [16]. They implemented a Markov Decision Process with a closed control loop with discrete time steps on each agent. They also were able to implement a client/server that simulated the playing field, communication, the environment and its dynamics, while the agents, eleven per team, sent their intended actions, once per simulation cycle, to the server [16]. While this research did prove successful it did require the use of a server and a control loop to be implemented on each agent.

Research on an application for multi agent learning was done by researchers [17] who used multi agent learning in a coverage application task by using an UAV as their agent. Each agent acted independently and can observe the preceding actions of its teammates to make a decision accordingly [17]. For this scenario a single reward was given to the group to encourage collaboration amongst the individual agents. To accomplish this the researchers proposed a noel actor-critic multi-agent RL algorithm to address the problem of multi-UAV target assignment and path planning [17].

CHAPTER 4
UNITY

Unity is a cross-platform game engine that allows the implementation of art assets or models, physics, and code for creating game experiences or simulations. Unity is built using the model of a scene, or stage of the experience with game objects in that scene. Game objects are separate entities that each have a position, Cartesian coordinates (x,y,z) with the scale being 1 equals to one meter, in the scene and have the ability to have components added. Such components could be a renderer, collider, rigid body, or C# script.

This reinforcement training was done using Unity ML-Agents which is a Unity plugin that uses Python and PyTorch to set up training environments for RL. As illustrated in Figure 4.1, Unity is where the environment is setup for the training and that environment connects to the PyTorch Python API. This provides the ML-Agents SDK which contains all functionality necessary to define environments within the Unity Editor along with the core C# scripts to build a learning pipeline [5]. This allows for different environments to be set up for training agents with different deep learning algorithms.

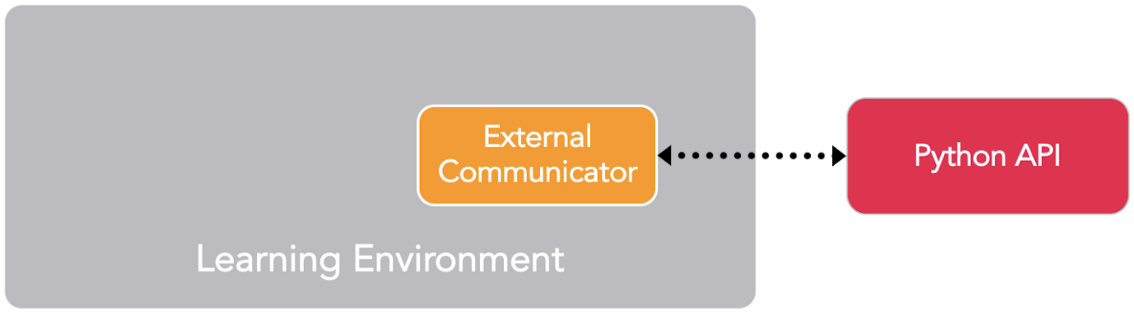


Fig. 4.1: Unity Learning Environment with Python API [4]

The main components of the ML-Agents SDK are sensors, agents, and the academy. Any object that is in the scene and can observe, receive rewards, and take action is an agent or an intelligent entity. An agents collects observation via many means, such as ray casting, images, knowledge of x, y, and z locations of other objects, or Boolean logic such as if agent has a ball. Each agent has a policy labeled with a behaviour name. Multiple agents can have a policy with the same behaviour name [5].

Agents can receive both positive and negative rewards during training by using the Unity scripting system. Examples of such rewards my be agents completing a task, agents failing, or smaller intermediate rewards such as agents getting closer to a target. The reward structure and size is totally dependant on the training scenario designer. An agent's top priority is to maximize their reward output, which should be taken into consideration on how much to reward both positive and negatively because it will affect the training outcome. Rewards can be given to agents individually or in the case of a MARL where collaboration is encouraged algorithms such as MA-POCA the reward can be given to the agents as a whole.

The academy is the overseer of the simulation by keeping track of things such as simulation steps and agents. The academy also contains the ability to define environment parameters, which can be used to change the configuration of the environment at runtime [5]. As seen in Figure 4.2 agents take action and observe the environment while the academy takes care of coordination amongst the agents and environment simulation.

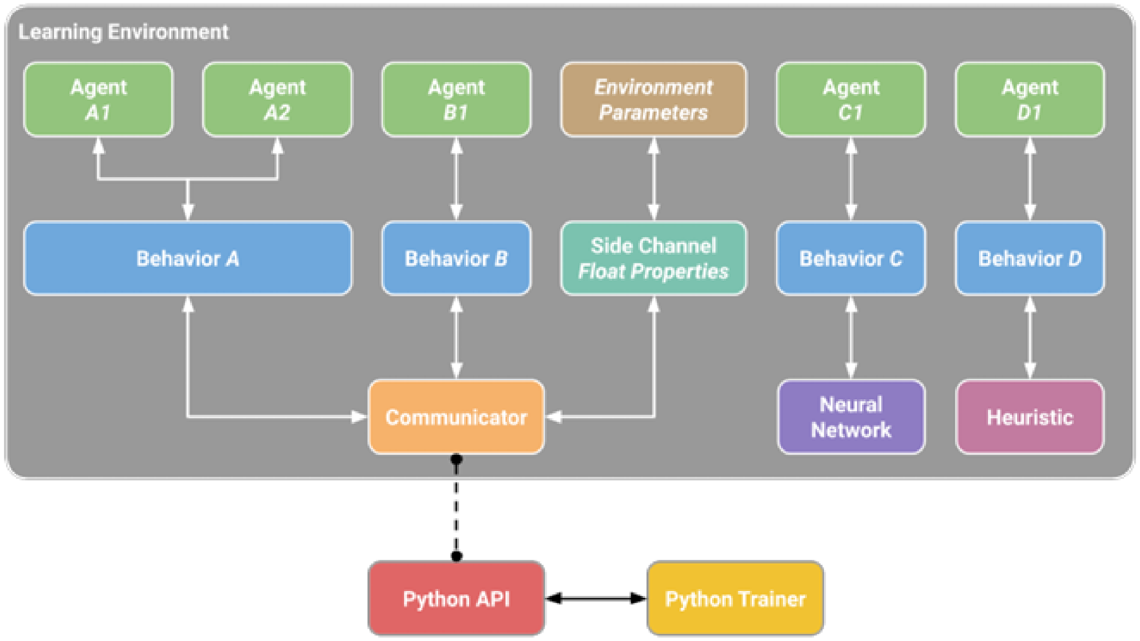


Fig. 4.2: Unity Learning Environment with Academy and Agents [5]

The environments that were created for this paper are set up with an agent(s) in a walled rectangular shaped arena so that the agent cannot leave the arena. To speed up the training each training scenario had multiple environments, each with their own agent(s), target(s), arena, and goal, to speed up training.

CHAPTER 5
PRELIMINARY WORK

To implement multiple simulated hexapod robots completing a task into simulation, preliminary work was done before hand to learn and to test the different RL algorithms. There was a total of four different training scenarios implemented to break down the complexity of creating the different environments. The scenarios are: Bring Ball to Goal, Push Target to Goal - Single Agent, and Push Target to Goal - Multiple Agents. All of these different training scenarios used PPO as the RL algorithm unless otherwise stated.

5.1 Bring Ball to Goal

As illustrated in Figure 5.1 in this scenario an agent was placed in an arena with the objective to bring a target to a goal. On environment initialization the agent, target, and goal are randomly spawned on the playing field. The target is sphere-shaped, however, it does not have the ability to roll around the environment keeping the same initial position unless acted upon by the agent. As soon as the agent collides with the target, the target's position will be set to the agent's position so that the agent can bring the target to the goal. The episode ends when the agent collides with the goal if the target's position is set to the agent's position first.

5.2 Push Target to Goal - Single Agent

For this scenario an agent was placed in an arena with the objective to push a target to a goal. As illustrated in Figure 5.2 on environment initialization the agent and target are randomly spawned in the arena, with the goal randomly spawned either on the left or right side of the arena. The target is cubed-shaped so that the agent and target both have flat surfaces for pushing. The target is able to be moved freely so that if the agent runs into it and has a constant speed, the target will be pushed with the same vector as the agent's

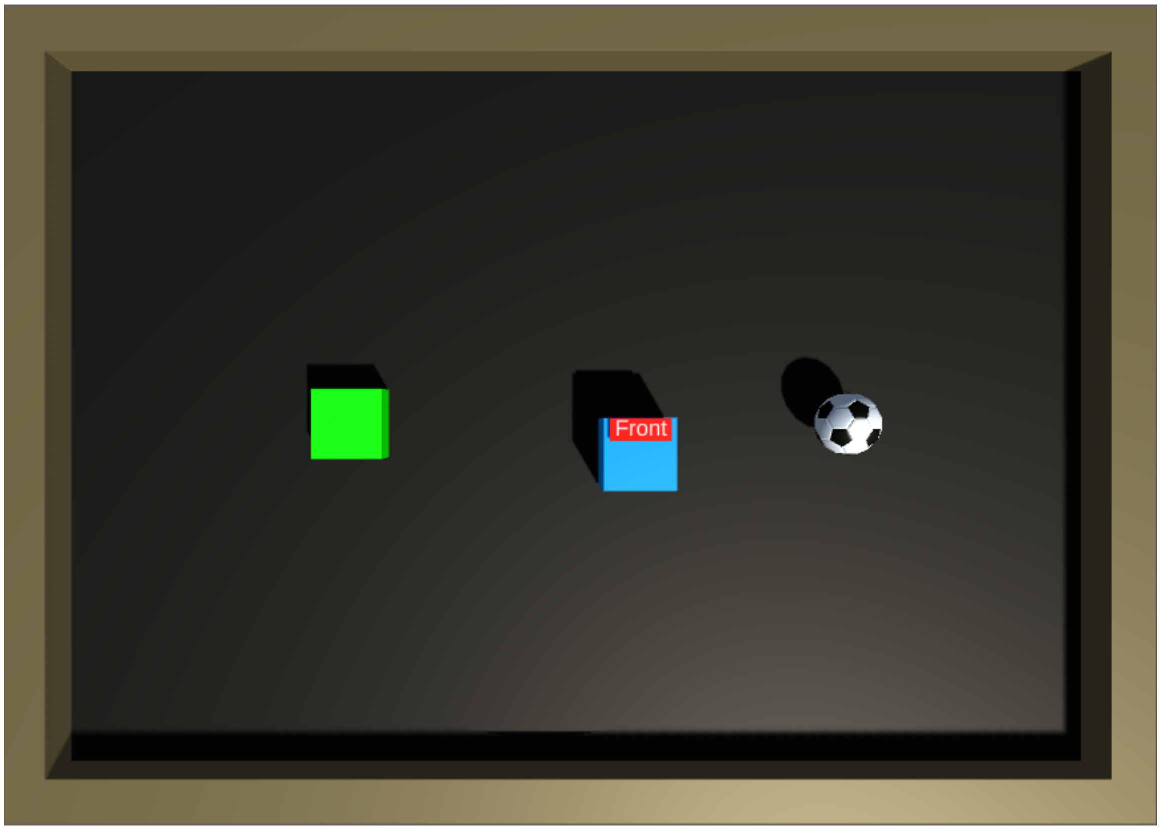


Fig. 5.1: Ball To Goal Environment

vector. There is no momentum in the target so if the agent stops pushing it the target's position will cease to changed. The episode ends when the target collides with the goal.

5.3 Push Target to Goal - Multiple Agents

In this scenario three agents are placed in an arena with the objective to push six targets to a goal. Because of the use of multiple agents the RL algorithm that was used was MA-POCA, which as being described earlier allows all the agents to share a reward. This will allow for shared knowledge amongst the agents and will help them achieve their goal faster.

As illustrated in Figure 5.3 on environment initialization the agents and targets are randomly spawned in the arena, with the goal randomly spawned either on the left or right side of the arena. The targets are cubed-shaped so that the agent and target both have flat

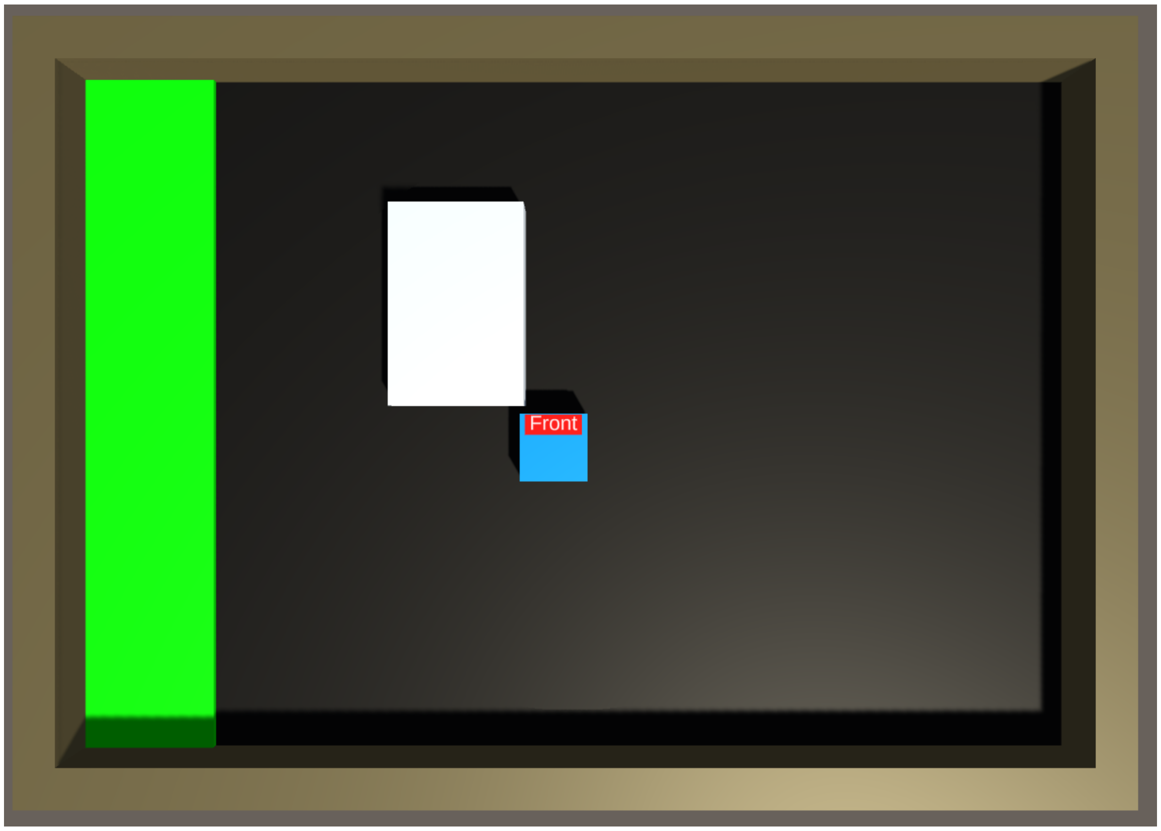


Fig. 5.2: Push Target to Goal - Single Environment

surfaces for pushing. As before the targets are able to be moved freely so that if an agent runs into it and has a constant speed, the target will be pushed with the same vector as the agent's vector. However, there are three different sizes of targets: small, medium, and large. The small target, labeled "1", has a mass of 10 kg requiring at least one agent to push it to be moved, the medium target, labeled "2", has a mass of 150 kg requiring at least two agents to push it simultaneously for it to be moved, and the large target, labeled "3", has a mass of 250 kg requiring all three agents to push it simultaneously for it to be moved. There is no momentum in the targets so if the agents stop pushing a target its position will not be changed. Once a target has collided with the goal it will be removed from the arena. The episode ends when all the targets have collided with the goal.

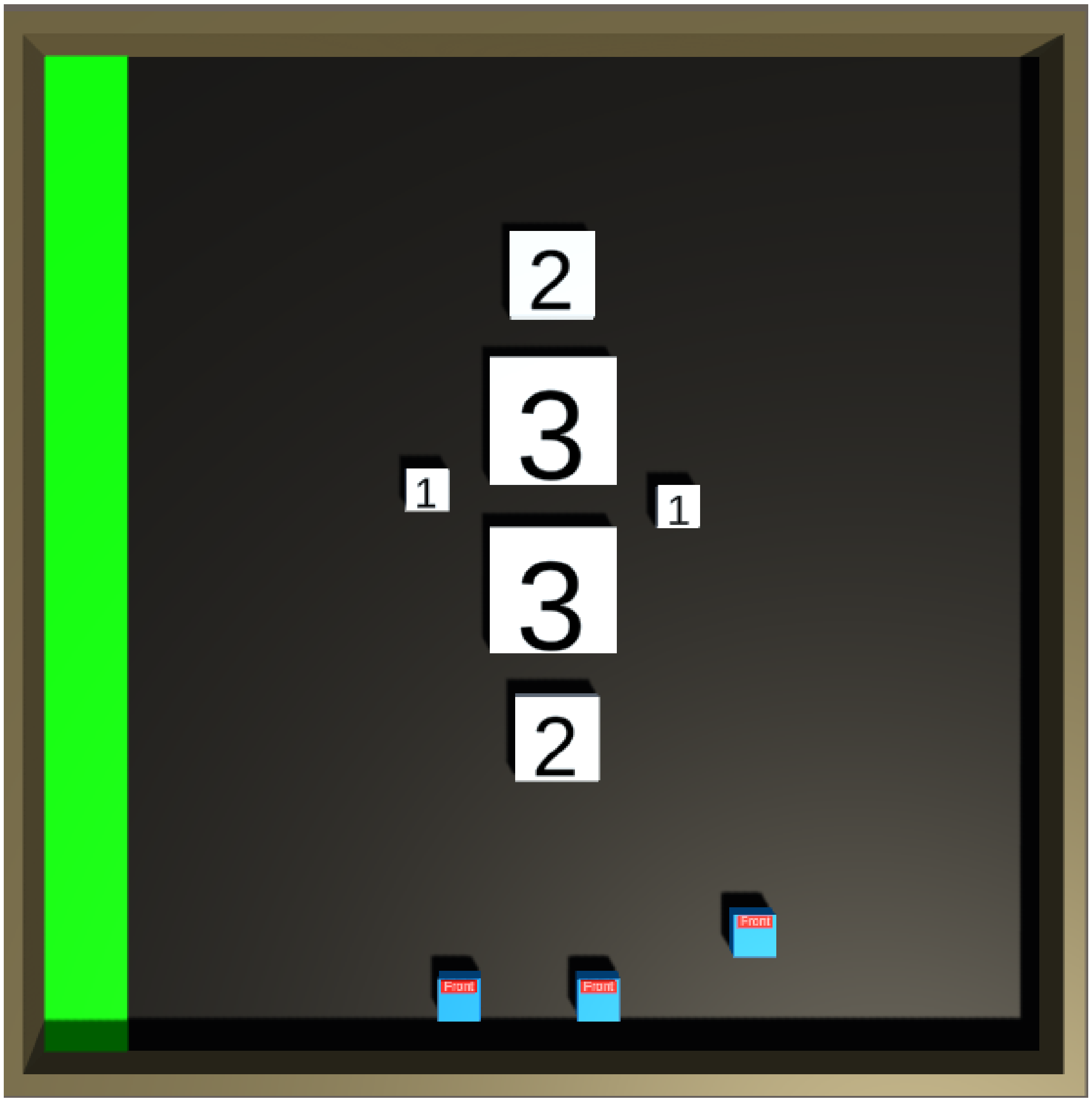


Fig. 5.3: Push Target to Goal - Multiple Environment

5.4 Preliminary Work Results

After the different environments were setup they were each trained separately using PyTorch. The first training scenario, Ball to Goal, was trained by running the agent for two million actions, or steps. As illustrated in Figure 5.4 the training results are shown with the cumulative reward on the y-axis and the number of steps on the x-axis. As seen, as the steps increase the cumulative reward consistently increases showing that the agent in this environment was learning by getting a higher reward. During the training the agent did not obtain the maximum reward possible, however, the trained result was good enough as the trained agent was consistently able to find the ball, grab the ball, and take the ball to the goal.

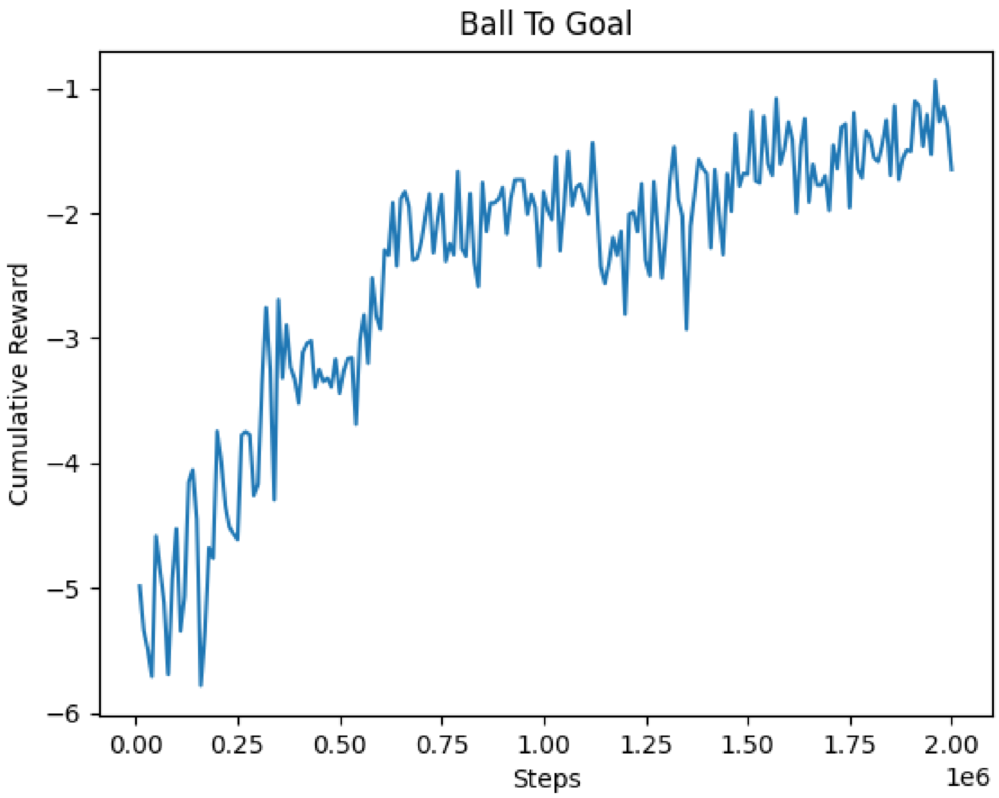


Fig. 5.4: Ball To Goal Results

For the next scenario, Push to Goal - Single Agent, the agent was trained for three million steps. As illustrated in Figure 5.5 the training results are shown with the cumulative reward on the y-axis and the number of steps on the x-axis. As seen at around 1.25 million steps the cumulative reward stopped increasing at around positive two or three. This is because the agent achieved the maximum reward possible and it was not possible to receive a better reward. At the end of the training the agent was able to complete the task of pushing the target to the goal.

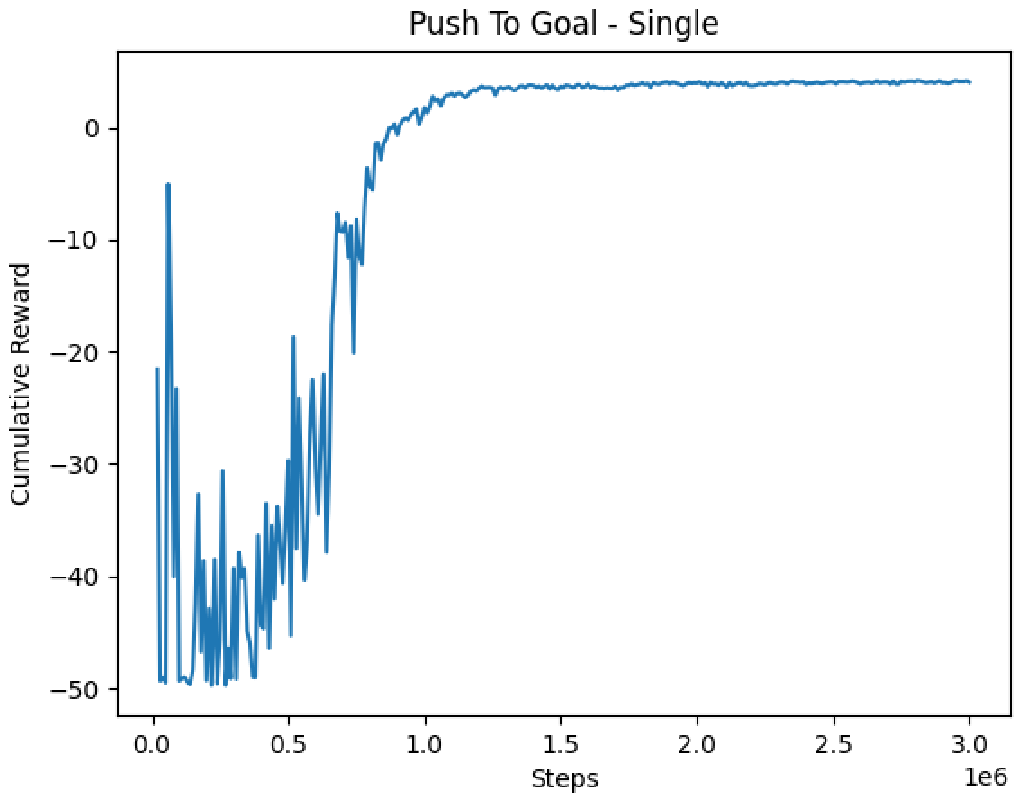


Fig. 5.5: Push To Goal - Single Results

For the last scenario, Push to Goal - Multiple Agents, the agents were trained for twenty five million steps. As illustrated in Figure 5.6 the training results are shown with the cumulative reward on the y-axis and the number of steps on the x-axis. As the steps increase the cumulative reward increases this proves that the agents did learn by increasing their cumulative reward. The reason for the spikes in the graph because at some points in the training the agents were able to push the medium and large targets to the goal. After the training the agents were able to consistently push the small targets to the goal, however, were not able to consistently push the medium and large targets to the goal proving that the agents needed more time to train.

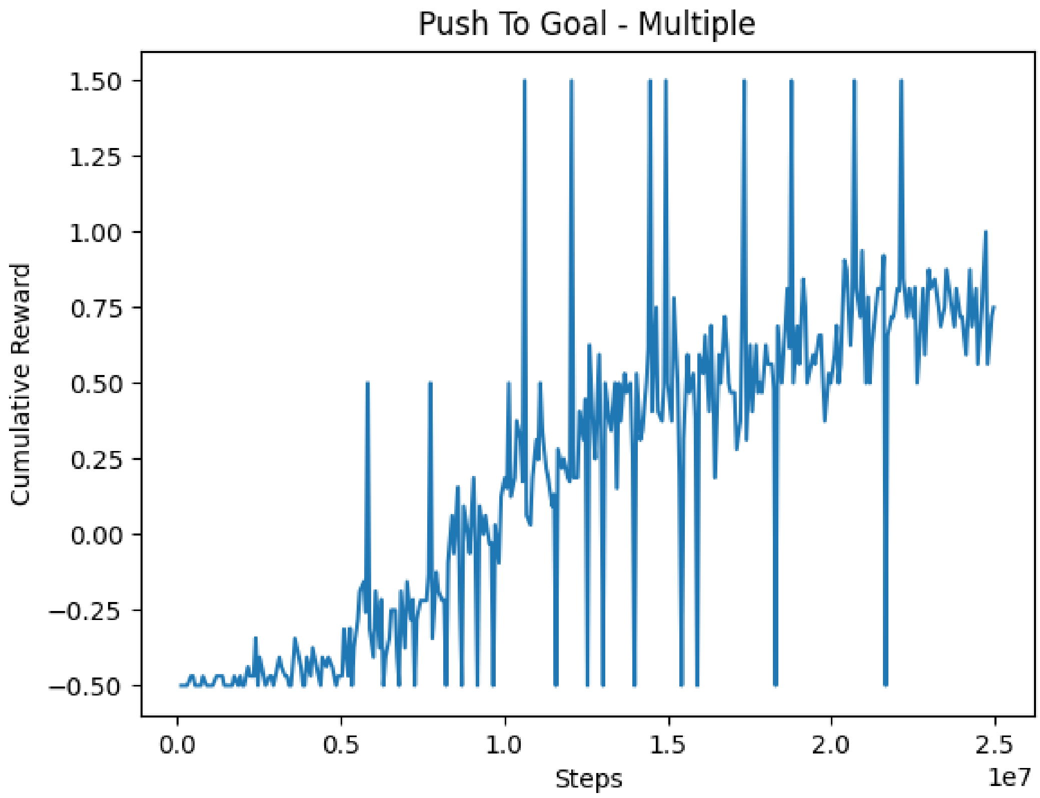


Fig. 5.6: Push To Goal - Multiple Results

CHAPTER 6
METHODS

6.1 Training Environment

The training environment that was setup for the learning for this research was setup using Unity. As seen in Figure 6.1 the environment was composed of a floor, which is a (100x, 1y, 100z) cube, with walls surrounding the area to keep the agents in the environment. Both the floor and walls have box colliders around them so that the goals, targets, or agents cannot fall through them. Each environment contains two blue hexapod agents, a green target, and two white goals on the top and bottom.

The target is a (20x, 1y, 1z) cube. The starting position of the target is randomized randomized position between -35 to 35 in the x, 1 in the y position, and -40 to 40 in the y with the origin being the center of the training environment. If a target collides with a goal it will respawn randomly in the training environment under the same random parameters. The mass was modified for different training scenarios and will be discussed later on in the Results chapter. I has an angular drag of 0.05 which requires both agents to push it effectively. The target also cannot rotate so if an agent is push on one of the target's edges the target will always be parallel to the goal. The goals are (100x, 2y, 5z) cubes where one is at the top and the other one is at the bottom of the environment. To detect collision both the target and goals have box colliders on them.

The goal for the training environment is to have the two agents push the target to the goal collaboratively by first learning how to walk, second know where the target and goal are, and third learn how to push the target to a goal.

6.2 Environment Controller

The environment controller handles what needs to happen in the scene as well as the

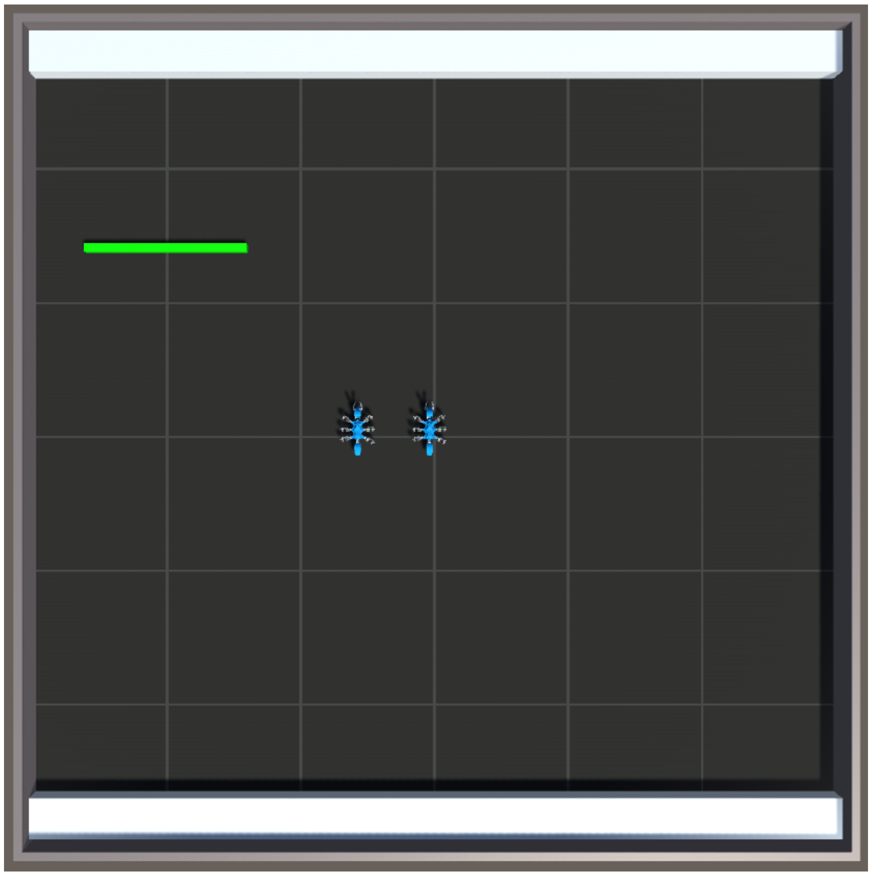


Fig. 6.1: Unity Training Environment with Agents

reinforcement parameters such as max environment steps and the reward function. The max environment steps which are how long the environment will run before it resets. For this training the max environment steps is 25000, each step is called every 0.02 seconds so the environment will reset every 500 seconds. The environment controller handles spawning the target randomly in the scene at scene initialization and when the target hits the goal. At the beginning of each episode it will reset the agents to their starting positions and will randomly spawn the target in the training environment. Once the target hits the goal the environment controller will randomly spawn it to a new location.

The environment controller uses Unity's SimpleMultiAgentGroup class which registers all of the agents under one entity. This allows the ability to give the whole team of agents a reward collectively. It also knows the agent's and target's starting position, starting rotation, and ridged body parameters.

6.3 Agent

As shown in Figure 6.2 each agent was a 3D hexapod ant robot that was modeled by gusmangananda [18]. Each agent has six legs with three different components per leg (femur, tibia, and a tarsus) that can all move as separate linked components.

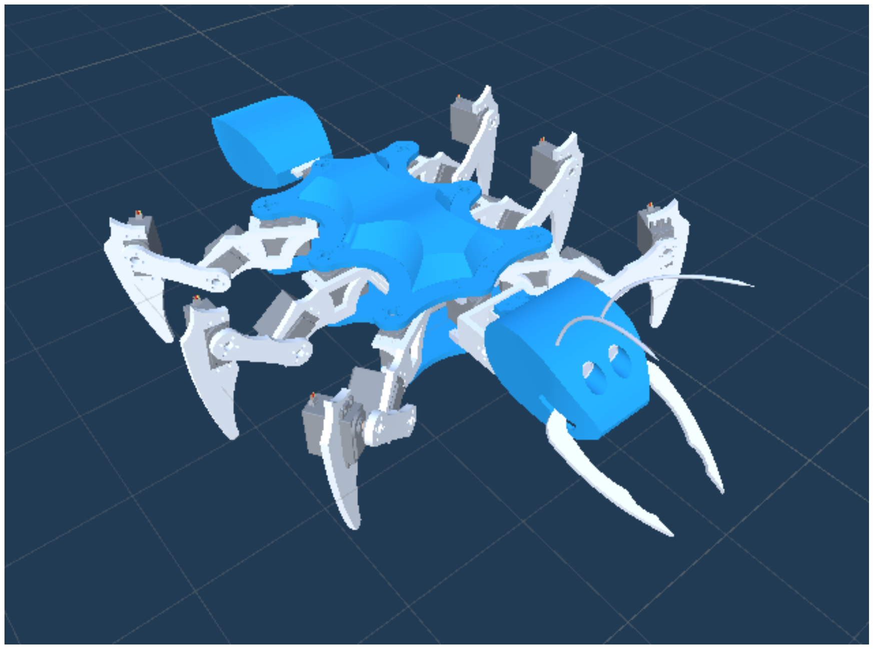


Fig. 6.2: Agent Hexapod Model

For the agent to be able to walk, each leg component needed a set amount of degrees of freedom (DOF) set which specified the amount of DOF each joint could move. The was done by adding a Unity Configurable Joint component, which allows setting a number of DOF in either x, y, or z axis to each each leg joint. This is so that the agent will be able to move realistically. Each DOF was modeled after the A-Pod robot by Lynxmotion. Since each leg component could only move on one axis all axes were set to zero unless otherwise specified. Each leg components were given the same DOF except for the two legs in the middle of the model where the femur only had 60 DOF on the y axis.

The femur DOF was set to 120 on the y axis as shown in Figure 6.3 the DOF on the femur is represented by the greenish-yellow shading area near the joint connecting the body and femur on the model.

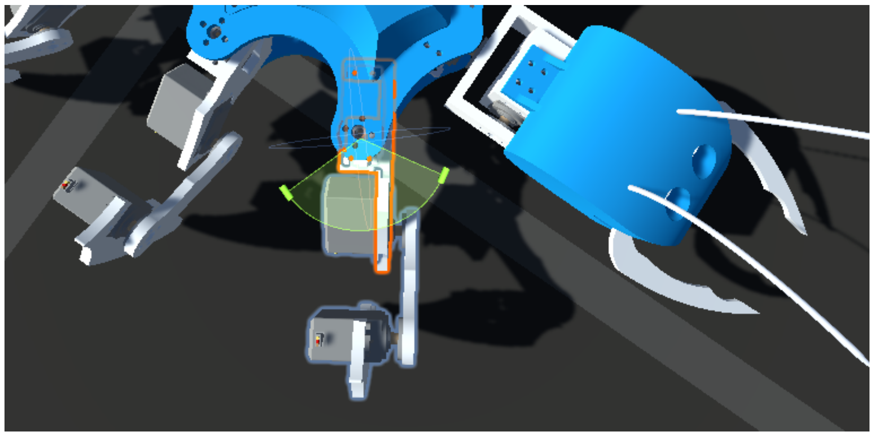


Fig. 6.3: Femur DOF

The tibia DOF was set to 110 on the z axis as shown in Figure 6.4 the DOF on the tibia is represented by the blue shading area near the joint connecting the femur and tibia on the model.

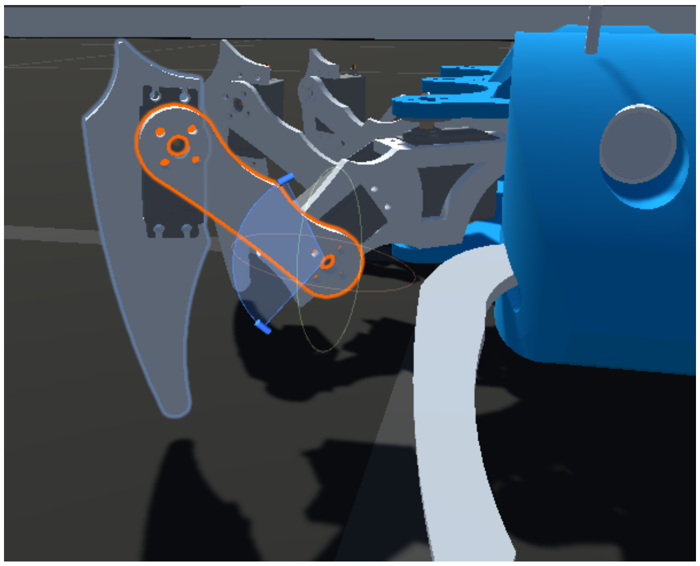


Fig. 6.4: Tibia DOF

The tarsus DOF was set to 180 on the z axis as shown in Figure 6.5 the DOF on the tarsus is represented by the blue shading area near the joint connecting the tibia and tarsus on the model.

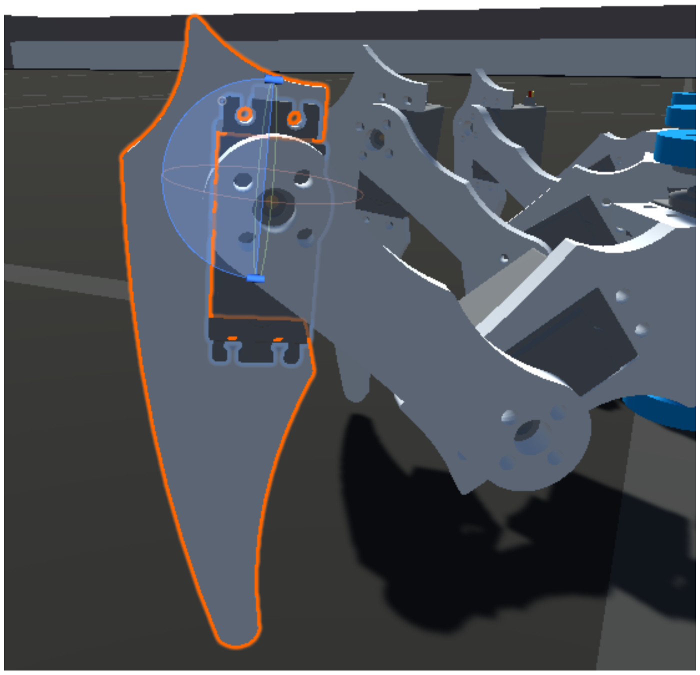


Fig. 6.5: Tarsus DOF

Each agent has a mass of 20. The body, tail, and head has a box collider and the femur, tibia, and tarsus all have mesh colliders, this allows the agent the ability to push the target and so it will not be able to fall through the floor.

6.3.1 Agent Observations

Each agent is given observations of their environment. This is in hope that based off of these observations, the agent can make the best decisions possible. For this training all of the observations have to do with target and goal positions, where the agent is pointed, each of the agent's leg component positions, the joint strength of each leg component, the velocity of the agent, and whether or not each joint of the agent is touching the ground. Table 6.1 shows all of the agent's observations.

Table 6.1: Agent's Observations

Begin of Table		
Observation Name	Data Type	Number of Observations
Orientation Cube Forward	Vector3	3
Velocity Goal	Vector3	3
Average Velocity	Vector3	3
Velocity Goal - Average Velocity	Float	1
Agent Distance from Target	Float	1
Average Agent Body Velocity Relative to Target	Vector3	3
Velocity Goal Relative to Target	Vector3	3
Rotation Delta	Quaternion	4
Position of Target Relative to Orientation Cube	Vector3	3
Goal 1 Position	Vector3	3
Goal 2 Position	Vector3	3
Direction to Goal 1	Vector3	3
Direction to Goal 2	Vector3	3
Body Touching ground	Boolean	1
Leg 1 Femur Touching Ground	Boolean	1
Leg 2 Femur Touching Ground	Boolean	1
Leg 3 Femur Touching Ground	Boolean	1
Leg 4 Femur Touching Ground	Boolean	1
Leg 5 Femur Touching Ground	Boolean	1
Leg 6 Femur Touching Ground	Boolean	1
Leg 1 Tibia Touching Ground	Boolean	1
Leg 2 Tibia Touching Ground	Boolean	1
Leg 3 Tibia Touching Ground	Boolean	1
Leg 4 Tibia Touching Ground	Boolean	1
Leg 5 Tibia Touching Ground	Boolean	1

Observation Name	Data Type	Number of Observations
Leg 6 Tibia Touching Ground	Boolean	1
Leg 1 Tarsus Touching Ground	Boolean	1
Leg 2 Tarsus Touching Ground	Boolean	1
Leg 3 Tarsus Touching Ground	Boolean	1
Leg 4 Tarsus Touching Ground	Boolean	1
Leg 5 Tarsus Touching Ground	Boolean	1
Leg 6 Tarsus Touching Ground	Boolean	1
Leg 1 Femur Joint Strength	float	1
Leg 2 Femur Joint Strength	float	1
Leg 3 Femur Joint Strength	float	1
Leg 4 Femur Joint Strength	float	1
Leg 5 Femur Joint Strength	float	1
Leg 6 Femur Joint Strength	float	1
Leg 1 Tibia Joint Strength	float	1
Leg 2 Tibia Joint Strength	float	1
Leg 3 Tibia Joint Strength	float	1
Leg 4 Tibia Joint Strength	float	1
Leg 5 Tibia Joint Strength	float	1
Leg 6 Tibia Joint Strength	float	1
Leg 1 Tarsus Joint Strength	float	1
Leg 2 Tarsus Joint Strength	float	1
Leg 3 Tarsus Joint Strength	float	1
Leg 4 Tarsus Joint Strength	float	1
Leg 5 Tarsus Joint Strength	float	1
Leg 6 Tarsus Joint Strength	float	1
End of Table		

Each agent has its own orientation cube which will rotate so that it points towards the target at all times. This gets updated every frame in the code. The algorithm as shown in Figure 6.6 takes the difference of vectors of the target's position and the orientation cube's position. After using a quaternion it uses that difference of the vectors to know how to rotate the orientation cube and set the rotation and position. That rotation is the new rotation from the quaternion and the position is the position of the orientation cube.

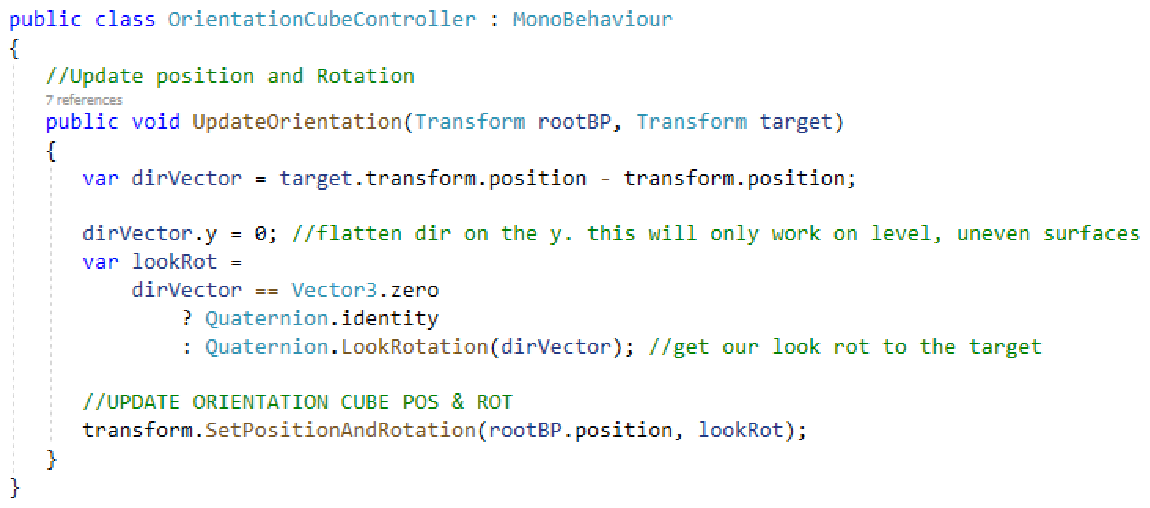

```
public class OrientationCubeController : MonoBehaviour
{
    //Update position and Rotation
    7 references
    public void UpdateOrientation(Transform rootBP, Transform target)
    {
        var dirVector = target.transform.position - transform.position;

        dirVector.y = 0; //flatten dir on the y. this will only work on level, uneven surfaces
        var lookRot =
            dirVector == Vector3.zero
                ? Quaternion.identity
                : Quaternion.LookRotation(dirVector); //get our look rot to the target

        //UPDATE ORIENTATION CUBE POS & ROT
        transform.SetPositionAndRotation(rootBP.position, lookRot);
    }
}
```

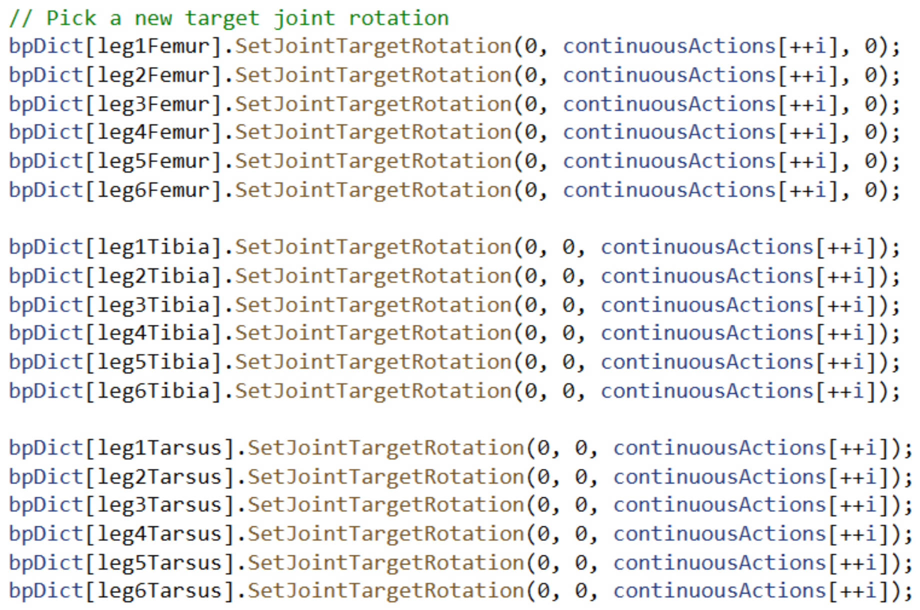
Fig. 6.6: Orientation Cube Controller [4]

6.3.2 Agent Actions

The agents uses continuous actions which are floating point values in an array that range from -1 to +1. These continuous actions are used to set the joint rotations as shown in Figure 6.7 and setting the joint strength as shown in Figure 6.8.

Set Joint Target Rotation

The algorithm for setting the joint target rotation as shown in Figure 6.9 takes in all of the continuous actions as either x, y, or z. Since each leg joint only moves in one direction on the axes as shown in Figure 6.7 each leg joint has only one continuous action, y for the femur and z for the tibia and tarsus, the other axes are set to zero. The Lerp function for Mathf is structured as Lerp(float a, float b, float t) where a is the start value, b is the



```
// Pick a new target joint rotation
bpDict[leg1Femur].SetJointTargetRotation(0, continuousActions[++i], 0);
bpDict[leg2Femur].SetJointTargetRotation(0, continuousActions[++i], 0);
bpDict[leg3Femur].SetJointTargetRotation(0, continuousActions[++i], 0);
bpDict[leg4Femur].SetJointTargetRotation(0, continuousActions[++i], 0);
bpDict[leg5Femur].SetJointTargetRotation(0, continuousActions[++i], 0);
bpDict[leg6Femur].SetJointTargetRotation(0, continuousActions[++i], 0);

bpDict[leg1Tibia].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg2Tibia].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg3Tibia].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg4Tibia].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg5Tibia].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg6Tibia].SetJointTargetRotation(0, 0, continuousActions[++i]);

bpDict[leg1Tarsus].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg2Tarsus].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg3Tarsus].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg4Tarsus].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg5Tarsus].SetJointTargetRotation(0, 0, continuousActions[++i]);
bpDict[leg6Tarsus].SetJointTargetRotation(0, 0, continuousActions[++i]);
```

Fig. 6.7: Setting Joint Rotation

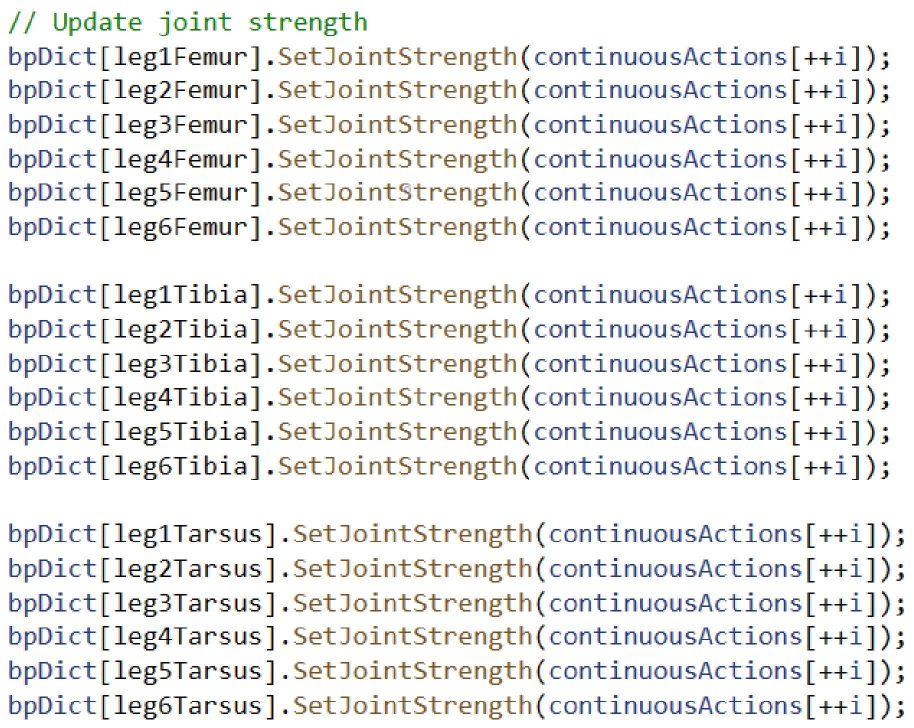
end value, and t is the interpolation value between the start value and end value where $0 \geq t \leq 1$. The purpose of

$$x = (x + 1f) * 0.5f;$$

$$y = (y + 1f) * 0.5f;$$

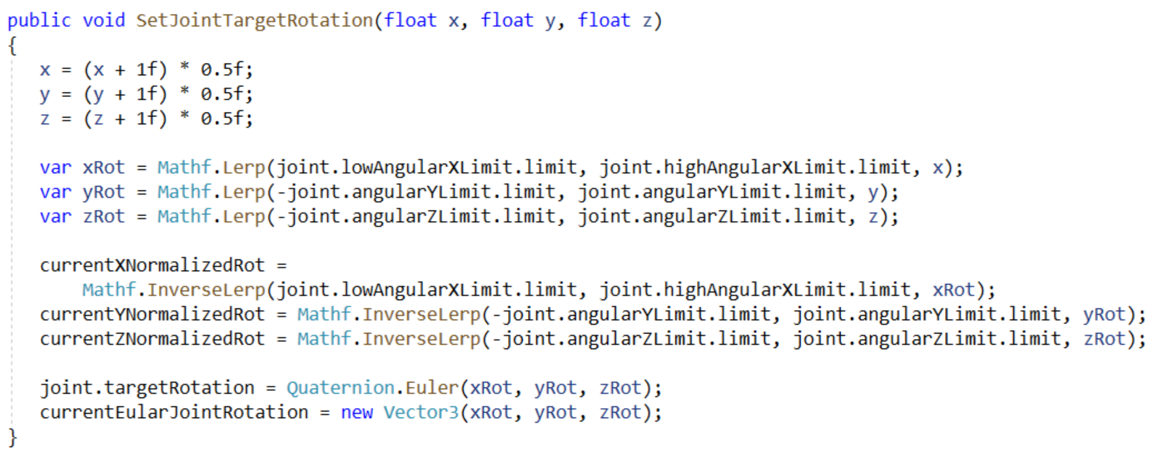
$$z = (z + 1f) * 0.5f;$$

is so that whether the continuous action value be between -1 and 1 we will always have $0 \geq t \leq 1$. With that we get xRot, yRot, or zRot as the interpolated float between both the low angular limit and high angular limit for x or negative angular limit and angular limit for y and z. The angular limit for the femur is 60 degrees for the front and back legs and 30 degrees for the middle legs, with 55 degrees for the tibia, and 90 degrees for the tarsus. The angular limit is essentially half of the DOF for each leg component. The joint target rotation which is a quaternion takes in xRot, yRot, or zRot which are Euler angles so it is necessary to convert them to a quaternion.



```
// Update joint strength
bpDict[leg1Femur].SetJointStrength(continuousActions[++i]);
bpDict[leg2Femur].SetJointStrength(continuousActions[++i]);
bpDict[leg3Femur].SetJointStrength(continuousActions[++i]);
bpDict[leg4Femur].SetJointStrength(continuousActions[++i]);
bpDict[leg5Femur].SetJointStrength(continuousActions[++i]);
bpDict[leg6Femur].SetJointStrength(continuousActions[++i]);

bpDict[leg1Tibia].SetJointStrength(continuousActions[++i]);
bpDict[leg2Tibia].SetJointStrength(continuousActions[++i]);
bpDict[leg3Tibia].SetJointStrength(continuousActions[++i]);
bpDict[leg4Tibia].SetJointStrength(continuousActions[++i]);
bpDict[leg5Tibia].SetJointStrength(continuousActions[++i]);
bpDict[leg6Tibia].SetJointStrength(continuousActions[++i]);

bpDict[leg1Tarsus].SetJointStrength(continuousActions[++i]);
bpDict[leg2Tarsus].SetJointStrength(continuousActions[++i]);
bpDict[leg3Tarsus].SetJointStrength(continuousActions[++i]);
bpDict[leg4Tarsus].SetJointStrength(continuousActions[++i]);
bpDict[leg5Tarsus].SetJointStrength(continuousActions[++i]);
bpDict[leg6Tarsus].SetJointStrength(continuousActions[++i]);
```

Fig. 6.8: Setting Joint Strength

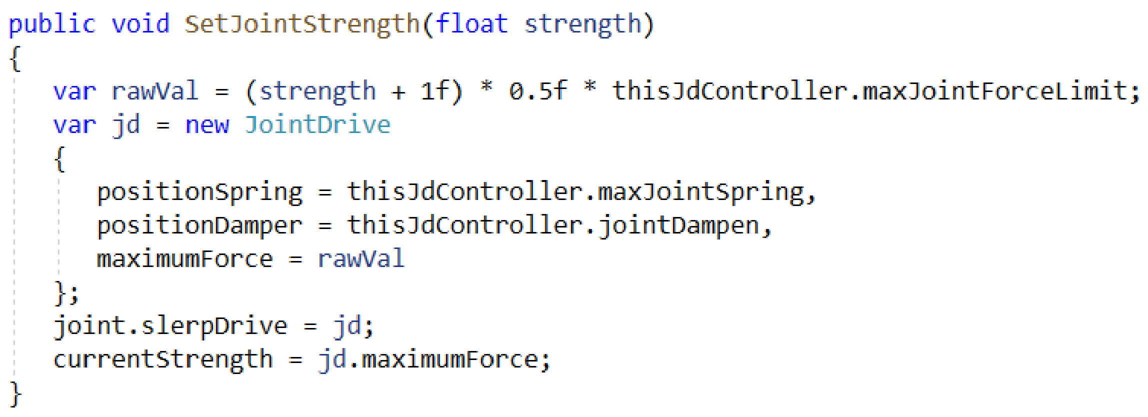


```
public void SetJointTargetRotation(float x, float y, float z)
{
    x = (x + 1f) * 0.5f;
    y = (y + 1f) * 0.5f;
    z = (z + 1f) * 0.5f;

    var xRot = Mathf.Lerp(joint.lowAngularXLimit.limit, joint.highAngularXLimit.limit, x);
    var yRot = Mathf.Lerp(-joint.angularYLimit.limit, joint.angularYLimit.limit, y);
    var zRot = Mathf.Lerp(-joint.angularZLimit.limit, joint.angularZLimit.limit, z);

    currentXNormalizedRot =
        Mathf.InverseLerp(joint.lowAngularXLimit.limit, joint.highAngularXLimit.limit, xRot);
    currentYNormalizedRot = Mathf.InverseLerp(-joint.angularYLimit.limit, joint.angularYLimit.limit, yRot);
    currentZNormalizedRot = Mathf.InverseLerp(-joint.angularZLimit.limit, joint.angularZLimit.limit, zRot);

    joint.targetRotation = Quaternion.Euler(xRot, yRot, zRot);
    currentEularJointRotation = new Vector3(xRot, yRot, zRot);
}
```

Fig. 6.9: Setting Joint Rotation Algorithm [4]

Set Joint Strength

The algorithm for setting the joint strength as shown in Figure 6.8 takes in the continuous action, called strength, and multiplies it with the maxJointForceLimit, which for this training scenario is 20000. It then creates a new JointDrive object which set position spring and position damper based off of maxJointSpring which is 40000 and jointDampen which is 5000. It then sets the slerpDrive, which is how the joints rotation will behave around all local axes, to this new object. The only difference between this new object and prior ones is the maximumForce of the joint which is calculated using the continuous action.



```
public void SetJointStrength(float strength)
{
    var rawVal = (strength + 1f) * 0.5f * thisJdController.maxJointForceLimit;
    var jd = new JointDrive
    {
        positionSpring = thisJdController.maxJointSpring,
        positionDamper = thisJdController.jointDampen,
        maximumForce = rawVal
    };
    joint.slerpDrive = jd;
    currentStrength = jd.maximumForce;
}
```

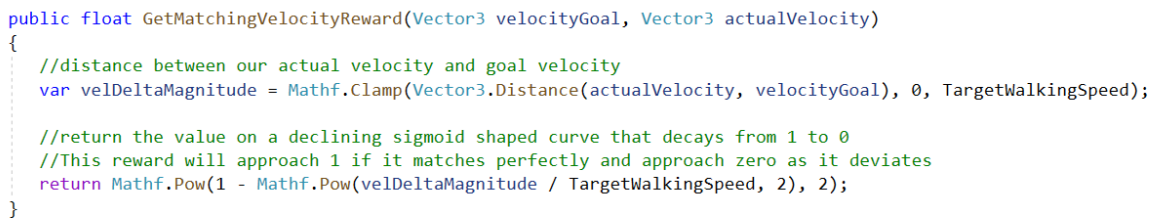
Fig. 6.10: Setting Joint Strength Algorithm [4]

6.4 Reward Function

To help the agents achieve their goal a reward structure was implemented. Since the multiple agent nature of this training MA-POCA was used as the RL algorithm which will also be useful if an agent need to be reset. There are three parts of the reward function that was implemented.

First, if any body parts of the agent touch the ground except for the tarsus the team will receive a reward of -1. This is to keep the agent from falling down. If an agent receives this reward this will cause the agent to reset back to its starting position. Only the agent that was responsible will reset and will not affect other agents in play other than effecting the team reward.

Second, the agent will receive a positive reward for looking at the target and if velocity of the agent is matching the velocity goal. The actual velocity is calculated by taking the average velocity of all of the body parts that have a ridged body component which are the body, femur, tibia, and tarsus of all the legs, 19 in total. It also has a velocity goal which is calculated by multiplying the way the orientation cube is pointing and the target walking speed. We calculate a match speed reward as seen in Figure 6.11 which is how the actual velocity is matched with the velocity goal. Here we get velDeltaMagnitude by getting the distance between our actual velocity vector and velocity goal vector. We use Mathf.Clamp to be sure the distance is between 0 and our target walking speed. By raising our values to the power twice, we return anywhere from zero to one. As our actual velocity and velocity goal are closer together we return a one and a zero as they get farther apart.



```
public float GetMatchingVelocityReward(Vector3 velocityGoal, Vector3 actualVelocity)
{
    //distance between our actual velocity and goal velocity
    var velDeltaMagnitude = Mathf.Clamp(Vector3.Distance(actualVelocity, velocityGoal), 0, TargetWalkingSpeed);

    //return the value on a declining sigmoid shaped curve that decays from 1 to 0
    //This reward will approach 1 if it matches perfectly and approach zero as it deviates
    return Mathf.Pow(1 - Mathf.Pow(velDeltaMagnitude / TargetWalkingSpeed, 2), 2);
}
```

Fig. 6.11: Matching Velocity Reward [4]

We also calculate a look at target reward by taking the dot product of the orientation cube and the agents forward body position. We then add one to that value so to be sure that we have a positive value and then multiply it by 0.5. We then reward the team by multiplying the match speed reward with the look at target reward. This reward will always be between a zero or one based off of what was previously was mentioned.

Lastly, if the target collides with one of the goals the team will receive a +100 which is the highest reward possible signifying to the agents that this is the end goal and most important priority.

CHAPTER 7
RESULTS

For the research, multiple training sessions were run with different RL algorithms and different target weights. Each training session was run for 30 million steps to give the agents sufficient time to learn and to start to converge on a solution.

As shown in Figure 7.1 all training sessions were run with six training environments running simultaneously to speed up training. Each training environment had two hexapod agents with the goal to push the target to a goal, the only difference between the training sessions was the weight of the target and which algorithm was used.

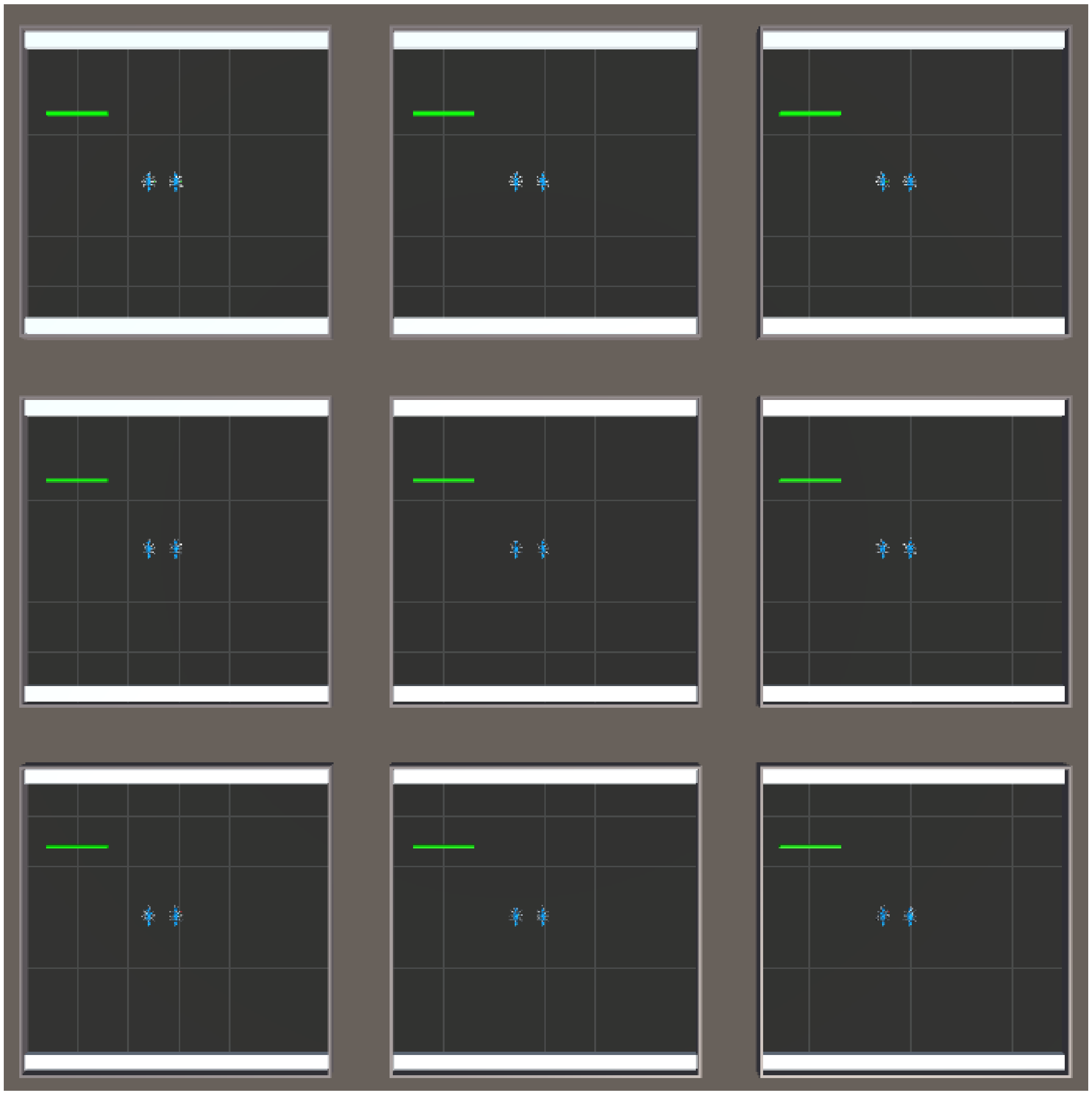


Fig. 7.1: Training Session with Six Training Environments

7.1 MA-POCA and PPO

The training sessions for the MA-POCA RL algorithm the reward structure was kept the same as before mentioned under the Methods chapter. For the training sessions for PPO, the reward function had to be modified. It has the same values for rewards as the MA-POCA, however the rewards are not shared as a team but rather given to an agent individually.

In all of the resulting graphs for the training sessions, the MA-POCA results are shown by the group cumulative reward and the PPO results are shown by the mean reward of all of the agents individually. The rewards for both MA-POCA and PPO are the numbers given to the team of agents or individual agents from the reward function which is shown on the y-axis. The steps, which is how long the agents ran for is on the x-axis.

7.1.1 10 kg Target

For the training session for the 10 kg target it was possible, although difficult, for one agent to push the target by itself. The results for the 10 kg target are shown in Figure 7.2. As seen for this target weight, the PPO algorithm was able to converge faster and achieve an overall higher average reward than MA-POCA.

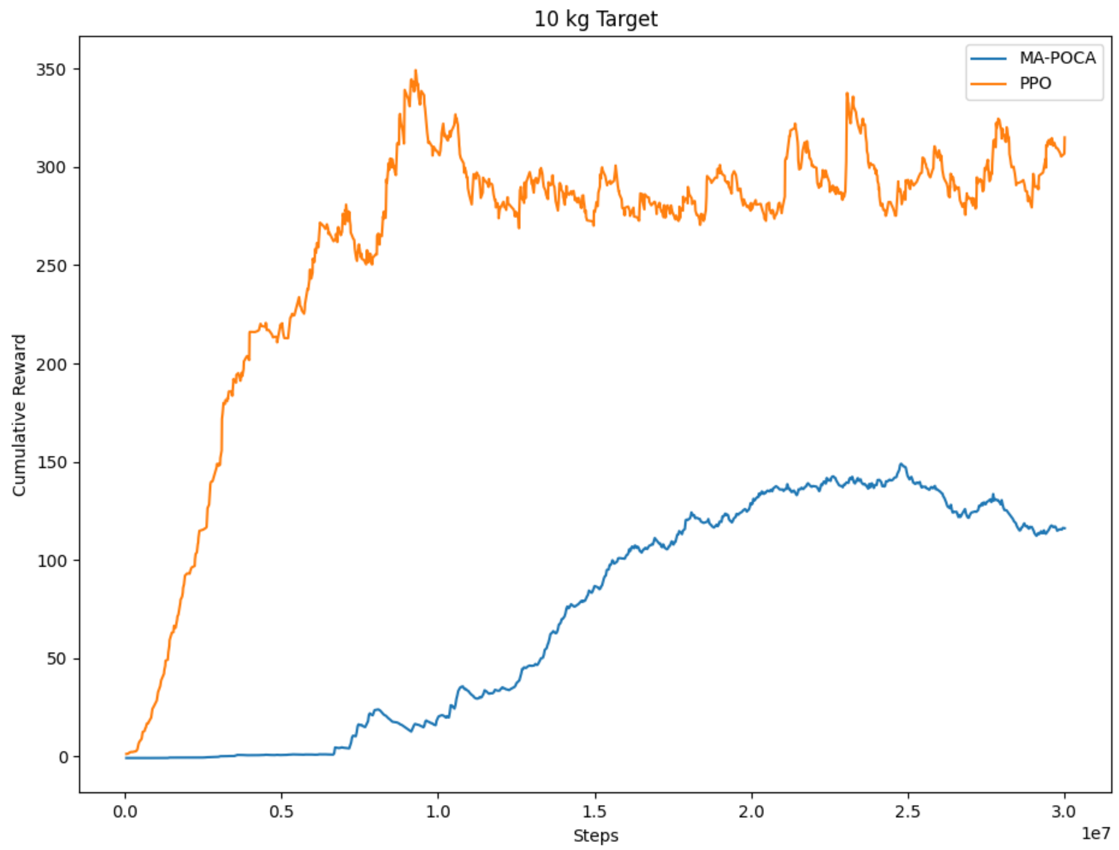


Fig. 7.2: MA-POCA and PPO 10 kg Target Results

7.1.2 15 & 20 kg Targets

As show in Figures 7.3 and 7.4 the training sessions with the 15 kg and 20 kg targets the results were very similar to the 10 kg with the PPO achieving a much higher reward than the MA-POCA. For these training session the target was too heavy for just one agent to push it and required the both agents to collaboratively push the target to a goal for success. The results show a slightly better reward for the 20 kg target scenario but that is believed to be a result of the randomness in the algorithms or a an error within the standard deviation. We suppose that if the training scenarios were run indefinitely we would expect the 15 kg target scenario to have a higher group cumulative reward but not by a significant amount.

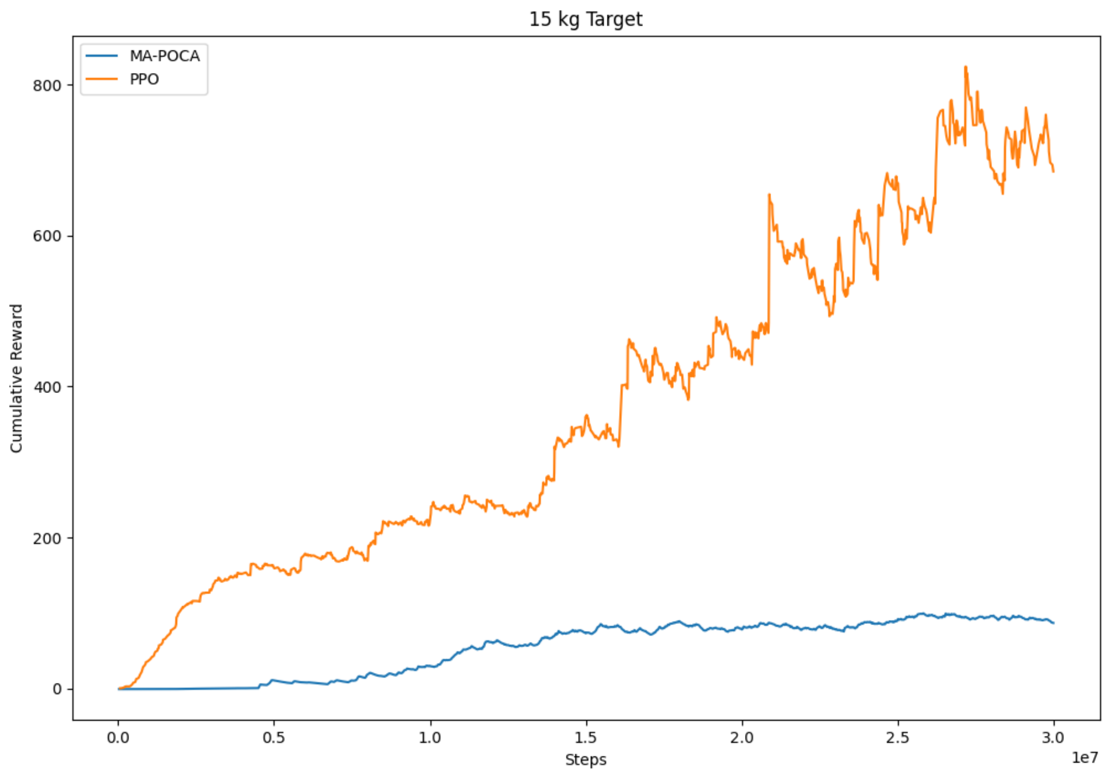


Fig. 7.3: MA-POCA and PPO 15 kg Target Results

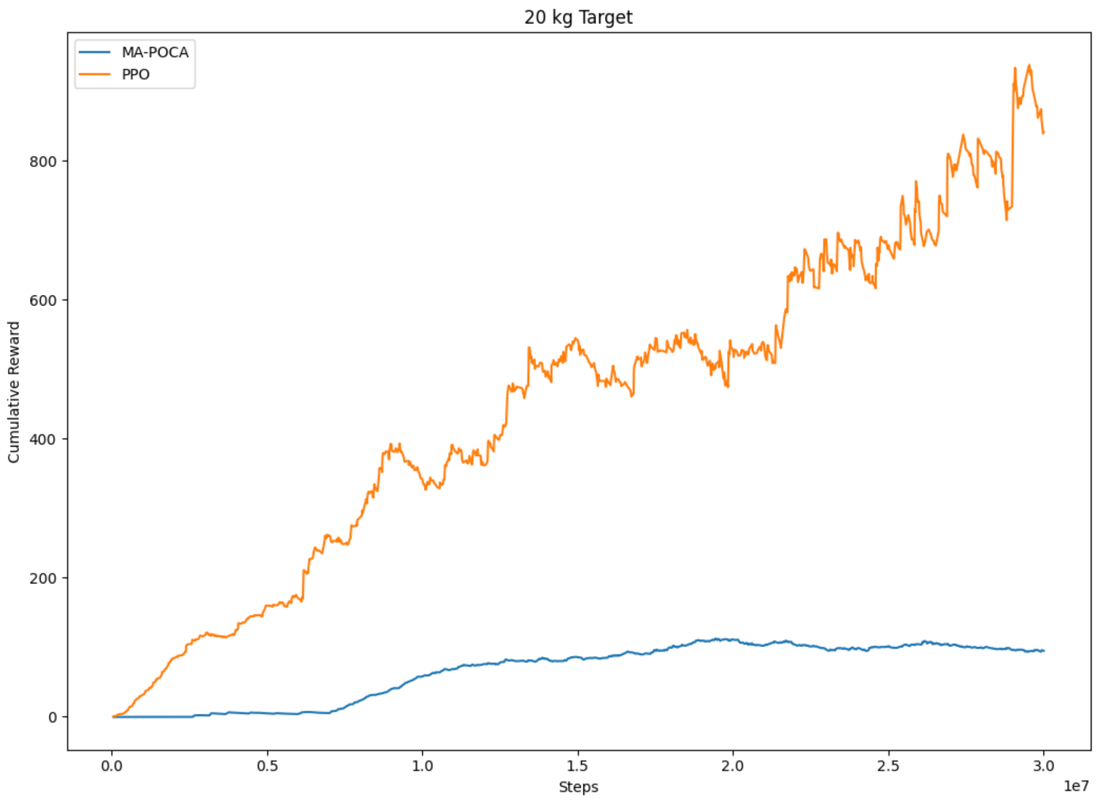


Fig. 7.4: MA-POCA and PPO 20 kg Target Results

CHAPTER 8
CONCLUSION

Through the research are able to learn many lessons and observations. First, we learn that the agents were given a reward to go towards the center of the target, this causes issues with the agents as they would trip over each other. This causes the agents to receive a negative reward and would overall discourage collaboration. Another thing we notice is at times one agent is pushing the other agent while that agent is pushing on the goal. Although this may be collaborative in nature it is not the most efficient way to push the target and could cause the agent getting pushed to fall over. Lastly we expected better success with the MA-POCA results as the algorithm is designed with agents collaborating. However where MA-POCA excels is with agents leaving and coming during training which was not the case in the training that was done. The PPO algorithm excels in handling complex high dimensional action spaces, the agents were able to learn how to walk better than the MA-POCA algorithm thus achieving better overall results.

In this work we address the problem of developing autonomous systems capable of exhibiting collaborative behaviours. To do this we implemented hexapedal ant-like robots in a RL simulation where we simulated collaborative task completion using multi-agent system. We discussed two RL algorithms MA-POCA and PPO that are commonly used in MARL. We ran simulation test on these two algorithms and discussed the results that we found.

CHAPTER 9
FUTURE WORK

In this work we proved that collaborative task completion is possible through RL. Although we were able to train simulated hexapod robots to collaboratively push a target to a goal there is still work to be done. The training environment goal placement was simplified and structured so that it did not matter if the agents were pushing the target either positively or negatively on the y axis as it was always towards a goal. Future work could only have one goal and give the agents rewards for pushing the target toward that goal. Future work could implement rather than agents going toward the center of the target they could go for any part of the target. This would fix the problem where the agents were too close to each other while pushing. The research would also be improved if the agents were able to grip the target with their claws allowing pulling and pushing motions. A significant improvement to the research would be refining the reward structure by adding more rewards or tweaking reward values to help the agents observe the environment better and make better decisions.

REFERENCES

[1] A. G. B. Richard S. Sutton, *Reinforcement Learning: An Introduction.* The MIT Press, 1992.

[2] "I want my own enormous robotic ant," Apr 2010. [Online]. Available: https://arthropoda.wordpress.com/2010/04/28/i-want-my-own-enormous-robotic-ant/

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[4] (2022) ml-agents/ml-agents-overview.md at main · unity-technologies/ml-agents. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md#training-in-cooperative-multi-agent-environments-with-ma-poca

[5] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2020. [Online]. Available: https://arxiv.org/pdf/1809.02627.pdf

[6] W. Smart and L. Pack Kaelbling, "Effective reinforcement learning for mobile robots," in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, vol. 4, 2002, pp. 3404–3410 vol.4.

[7] Z. Zhang, X. Luo, T. Liu, S. Xie, J. Wang, W. Wang, Y. Li, and Y. Peng, "Proximal policy optimization with mixed distributed training," in *2019 IEEE 31st international conference on tools with artificial intelligence (ICTAI).* IEEE, 2019, pp. 1452–1456.

[8] C. C.-Y. Hsu, C. Mendler-Dünner, and M. Hardt, "Revisiting design choices in proximal policy optimization," *arXiv preprint arXiv:2009.10897*, 2020.

[9] Y. Guan, Y. Ren, S. E. Li, Q. Sun, L. Luo, and K. Li, "Centralized cooperation for connected and automated vehicles at intersections by proximal policy optimization," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 11, pp. 12 597–12 608, 2020.

[10] A. Cohen, E. Teng, V.-P. Berges, R.-P. Dong, H. Henry, M. Mattar, A. Zook, and S. Ganguly, "On the use and misuse of absorbing states in multi-agent reinforcement learning," *arXiv preprint arXiv:2111.05992*, 2021.

[11] D. Kozlov, "Comparison of reinforcement learning algorithms in problems of acquiring locomotion skills in 3d space," in *2022 VIII International Conference on Information Technology and Nanotechnology (ITNT).* IEEE, 2022, pp. 1–5.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[13] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent autocurricula," *arXiv preprint arXiv:1909.07528*, 2019.

[14] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, pp. 9–44, 1988.

[15] X. Wang, C. Zhao, T. Huang, P. Chakrabarti, and J. Kurths, "Cooperative learning of multi-agent systems via reinforcement learning," *IEEE Transactions on Signal and Information Processing over Networks*, vol. 9, pp. 13–23, 2023.

[16] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," *Autonomous Robots*, vol. 27, pp. 55–73, 2009.

[17] L. Canese, G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Re, and S. Spanò, "Multi-agent reinforcement learning: A review of challenges and applications," *Applied Sciences*, vol. 11, no. 11, p. 4948, 2021.

[18] "Diy hexapod ant robot.step," https://thangs.com/designer/models/3d-model/183980, accessed: 2023-03-01.