School of Electrical and Computer Engineering

Diploma Thesis

Reinforcement Learning for Obstacle Overcoming using a Three-Dimensional Humanoid Model

Author

Ioannis Petroulakis

Committee

Prof. Michail G. Lagoudakis (Supervisor)

Prof. Michalis Zervakis

Prof. Thrasivoulos Spyropoulos

A thesis submitted in fulfillment of the requirements for the degree of Engineering Diploma from the School of Electrical and Computer Engineering

Chania, February 2024

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Ενισχυτική Μάθηση για Υπέρβαση Εμποδίων με χρήση Τρισδιάστατου Ανθρωποειδούς Μοντέλου

Συγγραφέας

Ιωάννης Πετρουλάκης

Εξεταστική Επιτροπή

Καθ. Μιχαήλ Γ. Λαγουδάκης (Επιβλέπων)

Καθ. Μιχαήλ Ζερβάκης

Καθ. Θρασύβουλος Σπυρόπουλος

Διπλωματική εργασία για την απόκτηση Διπλώματος Μηχανικού από τη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Χανιά, Φεβρουάριος 2024

Abstract

In the realm of artificial intelligence and robotics, the creation of agents capable of effectively overcoming obstacles is a great challenge. Reinforcement Learning has received substantial attention for its capacity to empower machines to learn and adapt within their surroundings, through interaction with their environment. This has led to groundbreaking advancements in the domain of autonomous agents. This diploma thesis embarks on a journey to harness the potential of Reinforcement Learning, with a specific focus on enabling obstacle overcoming through the utilization of a Three-Dimensional Humanoid Model, commencing from a walking learning example. Building on a comprehensive background, encompassing Unity Game Development, the ML-Agents toolkit, the Anaconda environment for streamlined dependency management, and the fundamental principles of Reinforcement Learning and the Proximal Policy Optimization (PPO) algorithm, the stage is set for a deep dive into the challenges of creating a model able to overcome obstacles. Through a series of experiments, the setup and progress are presented, along with the development of a reward function and the observation space for our agents. Changes in the environment are introduced to assess adaptability and resilience of our model, and PPO hyper-parameters are meticulously tuned for best results. This thesis concludes with promising outcomes, showcasing the creation of a fully functional model, adaptable to diverse environments. Furthermore, it outlines future directions for research and development, aiming to further the quest for intelligent agents capable of undertaking difficult challenges.

Περίληψη

Στον χώρο της τεχνητής νοημοσύνης και της ρομποτικής, η δημιουργία πρακτόρων ικανών να υπερβαίνουν εμπόδια αποτελεσματικά αποτελεί σημαντική πρόκληση. Η Ενισχυτική Μάθηση (Reinforcement Learning) έχει λάβει ιδιαίτερη προσοχή για την ικανότητά της να επιτρέπει στις μηχανές να μαθαίνουν και να προσαρμόζονται στο περιβάλλον τους μέσω της αλληλεπίδρασής τους με αυτό. Αυτό έχει οδηγήσει σε πρωτοποριακές εξελίξεις στον τομέα των αυτόνομων πρακτόρων. Η παρούσα διπλωματική εργασία εκκινεί ένα εγχείρημα να αξιοποιήσει τη δυναμική της Ενισχυτικής Μάθησης, εστιάζοντας στη δυνατότητα της υπέρβασης εμποδίων με χρήση ενός τρισδιάστατου ανθρωποειδούς μοντέλου, ξεκινώντας από ένα παράδειγμα μάθησης βαδίσματος. Χτίζοντας σε ένα ολοκληρωμένο υπόβαθρο, το οποίο περιλαμβάνει την πλατφόρμα Unity Game Development, την εργαλειοθήκη ML-Agents, το περιβάλλον Anaconda για βέλτιστη διαχείριση εξαρτήσεων και τις θεμελιώδεις αρχές της Ενισχυτικής Μάθησης και του αλγορίθμου Proximal Policy Optimization (PPO), οι προϋποθέσεις είναι έτοιμες για μια βαθιά κατάδυση στις προκλήσεις της δημιουργίας ενός μοντέλου ικανού να υπερβαίνει εμπόδια. Μέσα από μια σειρά πειραμάτων, παρουσιάζονται οι ρυθμίσεις και η πρόοδος, μαζί με τη δημιουργία μιας συνάρτησης ανταμοιβής και του χώρου παρατηρήσεων για τους πράκτορές μας. Εισάγονται αλλαγές στο περιβάλλον για την αξιολόγηση της προσαρμοστικότητας και της ανθεκτικότητας του μοντέλου μας και οι υπερ-παράμετροι του PPO ρυθμίζονται σχολαστικά για τη βελτιστοποίηση των αποτελεσμάτων. Η παρούσα εργασία ολοκληρώνεται με πολλά υποσχόμενα αποτελέσματα, παρουσιάζοντας τη δημιουργία ενός πλήρως λειτουργικού μοντέλου, προσαρμόσιμου σε διαφορετικά περιβάλλοντα. Επιπλέον, σκιαγραφεί μελλοντικές κατευθύνσεις για έρευνα και ανάπτυξη, με στόχο να ενισχύσει την ανάπτυξη ευφυών πρακτόρων ικανών να αντιμετωπίζουν δύσκολες προκλήσεις.

Acknowledgements

I would like to express my gratitude to Prof. Michail G. Lagoudakis for his guidance and advice and the opportunity he gave me to work on something I enjoyed experimenting with. I would also like to thank my family and friends for their assistance and support.

Contents

Chapter 1

Introduction

In the field of artificial intelligence and robotics, the endeavor to develop agents with the capacity to navigate within dynamic environments and conquer obstacles continues to present a substantial and enduring challenge.

Reinforcement Learning (RL) has gained considerable attention for its ability to enable machines to learn and adapt to their surroundings through interaction with the environment and has led to groundbreaking advancements in the field of autonomous agents.

One particularly intriguing application of RL in robotics is the development of Three-Dimensional Humanoid Models capable of overcoming obstacles in dynamic environments. These robots, inspired by the human form, represent a promising avenue for achieving advanced tasks in a wide range of settings, making them valuable assets in missions, where human lives could have been endangered.

1.1 Thesis Contribution

In this thesis, we study the problem of obstacle overcoming using a 3D humanoid model. Using Unity engine and the tools it provides, we manage to train a 3D humanoid model able to successfully overcome obstacles in environments that change arbitrarily at every try. Taking inspiration from the process of training a model to walk, we adapt various aspects of the Reinforcement Learning (RL) cycle. This involves significant modifications to the environment, the observations and the reward mechanisms. Moreover, we meticulously fine-tune the hyper-parameters of the Proximal Policy Optimization (PPO) algorithm to further maximize its performance. By implementing these strategic modifications, we develop a proficient model that excels in successfully tackling the designated challenge.

1.2 Thesis Outline

In Chapter 2, we dive into the foundational knowledge required to understand the subject of this thesis. We discuss the Unity Game Development Engine, the ML-Agents toolkit, Anaconda, Reinforcement Learning, and Proximal Policy Optimization (PPO). This background information aims to help readers comprehend the context and tools used in our research.

In Chapter 3, we focus on defining precisely the problem studied and provide an overview of prior work in the same area. We'll look into examples similar to our study and highlight the improvements we aim to make.

Chapter 4 outlines our approach to solving the identified problems. We discuss the setup, initial progress, reward function, observation space, environmental changes, and the tuning of PPO hyper-parameters. This section details the methodology, tools, and techniques employed in our research.

Chapter 5 presents the final results and findings of our research. We analyze the outcomes of our approach and provide insights into the practical complications of our work.

In the final Chapter 6, we draw conclusions based on our findings and results of our research. We also suggest potential directions for future work, focousing on areas where further research can build upon our findings.

Chapter 2

Background

2.1 Unity Game Development Engine

Unity is an industry-standard game development engine that has rightfully earned widespread popularity for its versatility, cross-platform compatibility, and extensive toolset. Initially designed for creating interactive video games, Unity's capabilities extend well beyond gaming, making it a versatile choice for researchers and developers in various domains, including robotics and artificial intelligence [4].

Unity engine has the capabilities to create realistic 3D environments, allowing researchers to simulate complex real-world scenarios for training RL agents. With its built-in physics engine [6], it allows for the accurate modeling of physical interactions, which is essential for simulating robot movements and dynamics. With the vast amount of published tutorials and documentations, as well as the well supplied asset store, it creates a perfect engine to simulate every imaginary or realistic scenario you can think of.

2.2 ML-Agents Toolkit

Unity's ML-Agents toolkit (Figure 2.1), an open-source project developed by Unity Technologies, serves as a bridge between Unity's rich simulation capabilities and the

field of Reinforcement Learning. ML-Agents empowers researchers and developers to train and evaluate RL agents in Unity environments efficiently [11].

Key components and functionalities of ML-Agents include:

-Unity Integration: ML-Agents seamlessly integrates with Unity, allowing the creation of custom learning environments using Unity's Editor and assets.

-Python API: ML-Agents provides a Python API for training and interacting with RL agents, enabling the use of popular RL libraries, such as TensorFlow and PyTorch.

-Customizable Agents: Researchers can define custom agent behaviors, including neural network architectures, observation spaces, and action spaces, tailored to specific tasks.

-Learning Algorithms: ML-Agents offers a variety of RL algorithms, including Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC), so that researchers can experiment with different algorithms to suit their objectives.

-Highly Scalable: ML-Agents supports distributed training, making it scalable for large-scale experiments that require multiple agents and simulations.

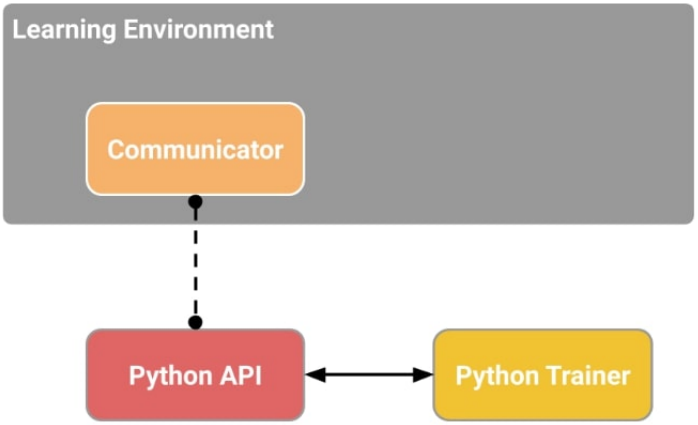

Figure 2.1: ML-Agents simplified [10]

2.3 Anaconda

Anaconda is a widely-used open-source distribution of software and libraries, designed to facilitate data science, machine learning, and scientific computing tasks. It tightly bundles a comprehensive suite of tools and libraries for data analysis, numerical computing, and machine learning into a single, easily manageable package [15].

Anaconda comes with its own package manager called "conda" which simplifies the installation, updating, and management of various data science and machine learning packages. Conda also manages dependencies, making it easier to set up and maintain complex software environments. It offers a great variety of library support and being able to create isolated virtual environments for different projects (making it possible to manage different project dependencies separately and avoid conflicts between packages), it becomes a great tool for a wide range of data analysis and machine learning tasks.

2.4 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning that focuses on enabling agents to learn very good (even optimal) decision-making strategies by interacting with an environment [5]. Unlike supervised learning, where models are trained on labeled data, RL agents learn through trial and error, receiving feedback in the form of rewards or punishments based on their actions. The fundamental goal of RL is to find a policy—a mapping from states to actions—that maximizes cumulative rewards over time.

Key concepts for RL [17]:

-Agent: The learner or decision-maker that interacts with the environment. The agent takes actions based on its current state and learns over time to maximize cumulative rewards.

-Environment: The external system with which the agent interacts. It includes all aspects of the problem, including states, actions, rewards, and transitions.

-State: A representation of the environment's current situation. States provide the context for the agent's decision-making.

-Action: The choices or decisions made by the agent. Actions influence the environment and can lead to transitions to different states.

-Reward: A numerical signal provided by the environment after each action, indicating the immediate desirability or quality of the agent's decision. The goal is to maximize the cumulative reward over time.

-Policy: A strategy or mapping that defines the agent's behavior, specifying which action to take in each state.

-Value Function: A function that estimates the expected cumulative reward an agent can achieve when starting from a particular state and following a specific policy. There are two types of value functions: state-value functions and state-action-value functions.

-Exploration vs. Exploitation: Agents face a trade-off between exploration (trying new actions to discover their consequences) and exploitation (choosing actions known to yield high rewards).

To sum up, the agent makes an action, checks the environment and depending on changes he gets a reward. Said reward is connected with its end goal, so the agent gradually learns from experience which action will lead to results with more reward to gain (Figure 2.2).

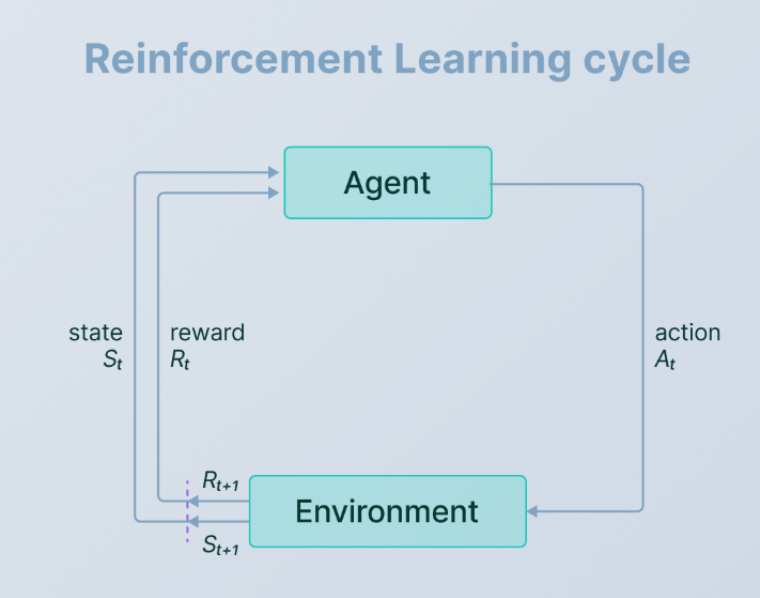

Figure 2.2: Here we can see the reinforcement learning cycle [18].

2.5 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) was introduced by OpenAI in 2017 [16] [14], focused on finding a stable and efficient way to optimize policies in reinforcement learning. It is a state-of-the-art reinforcement learning algorithm designed for training agents to learn policies that maximize expected cumulative rewards in a stable and sample-efficient manner.

PPO belongs to the family of policy gradient methods. In many of those, policy updates are unstable, because of the large step size, which leads to bad policy updates and when this new bad policy is used for learning, then it leads to even worse policy. If steps are small, then it leads to slower learning [1].

2.5.1 Key Concepts and Techniques

-Policy Optimization: PPO focuses on optimizing the policy, which is a mapping from states to actions, to improve the agent's decision-making. It seeks to find a policy that maximizes expected cumulative rewards.

-Trust Region Optimization: PPO employs a trust region approach to policy updates, ensuring that the new policy remains close to the old policy. This trust region constraint prevents overly large policy updates that can lead to training instability.

-Objective Function: PPO uses a surrogate objective function to approximate the policy improvement. It maximizes the expected advantage (the difference between the returns of the new policy and the old policy), while staying within the trust region.

-Actor-Critic Architecture: PPO often uses an actor-critic architecture, which consists of two neural networks. The actor network represents the policy and selects actions, while the critic network estimates the value (expected cumulative rewards) of state-action pairs.

-Clipping: PPO introduces a clipping mechanism in the objective function, limiting the extent to which the policy can change in each update. This clipping reduces the risk of policy updates that are too aggressive and destabilizing.

-Multiple Epochs: PPO typically conducts multiple optimization epochs per training iteration, using different batches of data to update the policy. This improves

stability and sample efficiency [12].

2.5.2 PPO Pseudo-Code

A simplified pseudo-code providing a basic outline for the PPO algorithm is presented below.

```
function PPO(environment, policy_network, value_network, optimizer,
         num_episodes, max_steps, gamma, lambda, epsilon_clip):

    for episode in range(num_episodes):
        state = environment.reset()
        episode_states = []
        episode_actions = []
        episode_rewards = []
        episode_probs = []
        episode_values = []

        for step in range(max_steps):
            #Actor-Critic Architecture: separate networks for the
            #policy  and the value function.
            action, prob, value = policy_network(state)
            next_state, reward, done, _ = environment.step(action)

            episode_states.append(state)
            episode_actions.append(action)
            episode_rewards.append(reward)
            episode_probs.append(prob)
            episode_values.append(value)

            state = next_state

            if done or step == max_steps - 1:
                last_value = 0 if done else value_network(next_state
                                                          )
                discounted_rewards = calculate_discounted_rewards(
                                    episode_rewards,gamma, last_value)
                advantages = calculate_advantages(discounted_rewards
                                                  ,episode_values)

                optimize_policy(value_network, policy_network,
                optimizer, episode_states, episode_actions,
                episode_probs, advantages, epsilon_clip)
```

```
                break

def calculate_discounted_rewards(rewards, gamma, last_value):
    discounted_rewards = []
    running_add = last_value

    for r in reversed(rewards):
        running_add = running_add * gamma + r
        discounted_rewards.insert(0, running_add)

    return discounted_rewards

def calculate_advantages(discounted_rewards, values, gamma, lambda):
    advantages = []
    last_advantage = 0
    for i in range(len(discounted_rewards)):
        delta = discounted_rewards[i] + gamma * values[i+1] - values
                                                [i]
        last_advantage = delta + gamma * lambda * last_advantage
        advantages.append(last_advantage)
    return advantages

#Policy Optimization: The policy network is updated using a
#combination of surrogate policy loss and value loss.
def optimize_policy(value_network, policy_network, optimizer, states
                    , actions, old_probs, advantages, epsilon_clip):

   #Multiple Epochs: Conduct multiple optimization epochs, iterating
   #over the dataset multiple times to update the policy.
    for _ in range(optimization_epochs):
        for i in range(len(states) // batch_size):
            start_idx = i * batch_size
            end_idx = (i + 1) * batch_size

            batch_states = states[start_idx:end_idx]
            batch_actions = actions[start_idx:end_idx]
            batch_old_probs = old_probs[start_idx:end_idx]
            batch_advantages = advantages[start_idx:end_idx]

            new_probs, new_values = policy_network(batch_states)

            ratio = new_probs / batch_old_probs

            #Trust Region Optimization and Clipping: the ratio of
            #new and old policy probabilities is clipped within a
            #certain range to ensure hat policy updates stay within
            #a certain range..
            clipped_ratio = clip(ratio, 1 - epsilon_clip, 1 +
```

```
                                                epsilon_clip)

            #Objective Function: The surrogate objective function is
            #implemented, where the goal is to maximize the expected
            #advantage within the trust region.
            surrogate_loss = -min(ratio * batch_advantages,
                                  clipped_ratio * batch_advantages)

            value_loss = mse_loss(new_values, batch_advantages)

            total_loss = surrogate_loss + value_loss

            optimizer.zero_grad()
            total_loss.backward()
            optimizer.step()

def clip(x, min_value, max_value):
    return max(min(x, max_value), min_value)

def mse_loss(predictions, targets):
    return ((predictions - targets) ** 2).mean()
```

2.5.3 PPO Advantages

The methods behind PPO arm PPO with a plethora of advantages [16], making it known for its sample efficiency. It can achieve good results with fewer samples compared to some other RL algorithms, making it suitable for real-world applications, where data collection may be costly or time-consuming. The trust region optimization and clipping mechanisms in PPO contribute to training stability. PPO tends to produce smoother learning curves and is less prone to divergence. Lastly, it is versatile and can be applied to both continuous and discrete action spaces, making it suitable for a wide range of RL problems. Summing up, PPO has gained its popularity for its robustness and ease of implementation.

2.5.4 PPO Hyper-parameters

PPO comes with a number of hyper-parameters [13] that the user can tune according to the exact problem and optimize the results.

Batch Size :
The batch size determines the number of data points used in each training iteration.

A larger batch size can enhance training stability, but demands more memory, impacting gradient estimate accuracy and training time.

Buffer Size :
The buffer size sets the maximum number of experiences (state, action, reward, next state) stored in the replay buffer. The replay buffer retains experiences over time and is sampled from during training. A larger buffer size improves the quality of experiences used for training.

Learning Rate :
The learning rate controls the step size at which the model's parameters are updated based on computed gradients. A higher learning rate accelerates learning progress, but may introduce instability or overshooting. Conversely, a lower learning rate ensures stability, but might decelerate learning progress.

Beta :
Beta is the coefficient for the entropy term in the loss function. Entropy encourages exploration by penalizing deterministic (low-entropy) actions. Smaller beta values promote more exploration, while larger values prioritize exploitation.

Epsilon :
Epsilon serves as the threshold parameter for the "clipping" step in the PPO loss function. Clipping prevents the policy update from becoming excessively large, enhancing stability. It defines a range within which the new policy's action probabilities are clipped based on the old policy's probabilities.

Lambda :
Lambda is the discount factor used to compute advantage estimates. Advantage estimates indicate how much better or worse a particular action is compared to the expected value. When lambda is low, the algorithm prioritizes the current understanding of the value function over actual rewards obtained from the environment, which can lead to bias in the learning process. Conversely, higher values of lambda prioritize the actual rewards obtained from the environment, potentially introducing more variance into the learning process.

Number of Epochs :
The number of epochs signifies how many times the entire dataset is used during one iteration of the training loop. More epochs yield more precise updates, but may extend training time, while fewer epochs expedite training, but can result in less accurate updates.

Gamma :
Gamma is the discount factor used to discount future rewards in reinforcement learning. It determines the importance of future rewards compared to immediate rewards. A value closer to 1 indicates that future rewards are highly important, while a value closer to 0 indicates that only immediate rewards are significant.

2.6 Catastrophic Forgetting

Catastrophic forgetting is a phenomenon in machine learning, where a model rapidly loses its previously learned knowledge, when it is trained on new data. This issue can occur when a model is updated or fine-tuned with new data, and the new information disrupts or erases the previously learned information, making it difficult for the model to retain knowledge about multiple tasks or domains over time.

Catastrophic forgetting can also affect PPO, especially when the agent needs to adapt to new tasks or environments, while retaining the knowledge of previously learned tasks [2].

Chapter 3

Problem Statement

The primary objective in this thesis is to train a simulated humanoid agent (Figure 3.1) to overcome obstacles using reinforcement learning. Our 3D humanoid agent has to be taught how to move forward in an obstacle course, having zero prior knowledge. Starting from the basics the agent has to learn how to walk and at the same time find a way to overcome multiple, static, but random, obstacles in a full realistic physics environment.

At our disposal we have observations, which provide the necessary information about the environment, actions, which encompass the set of permissible movements, and decisions the humanoid agent can undertake in response to its observations and the rewards offered to the agent during approved behavior. To do so, the model has to be designed using a physics engine, the environment has to be created and the algorithms have to be implemented.

The foundation of our environment is rooted in a pre-existing ML-Agents Unity environment known as "walker" and the algorithm chosen to achieve our mission is the Proximal Policy Optimization (PPO) algorithm. Together, these elements form the bedrock of our quest to transform an inexperienced 3D humanoid agent into a proficient obstacle runner through the power of reinforcement learning.

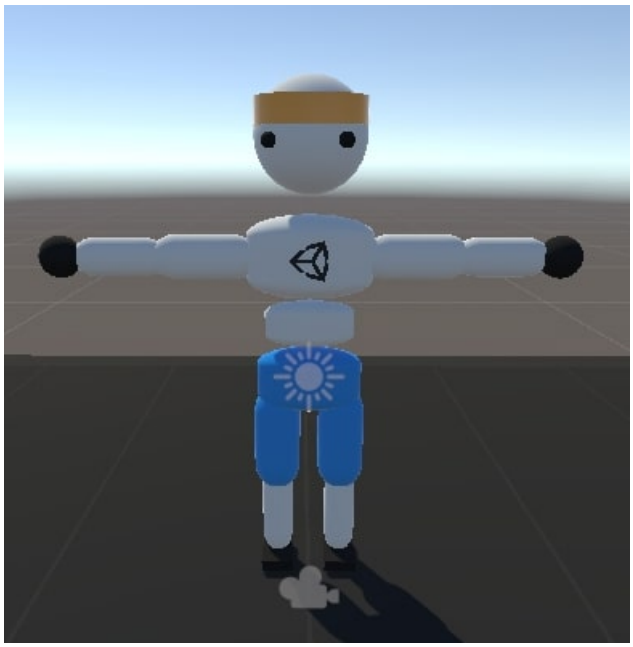

Figure 3.1: ML-Agents' walker agent.

3.1 Related Work

Ml-Agents in Unity offers some examples as reference, related to the one we're going to create, the most notable being the wall jump and walker ones.

The purpose in wall jump (Figure 3.2) is to train an agent to navigate environments, where wall jumps are required to reach certain locations or avoid obstacles.

The approach involves setting up a simulated environment, where the agent can interact with walls and learn to perform wall jumps. The agent is able to move forward, rotate, move sideways, jump and has knowledge of its current position, objects position and if it is grounded or not, at all times. Agent is rewarded every time he touches the goal, but loses reward as time passes by and if he falls from the platform.

The ML-Agents toolkit supports various RL algorithms, such as Proximal Policy Optimization (PPO), Trust Region Policy Optimization (TRPO), and others. PPO algorithm was used for this example proving to be up for the specific task, as the

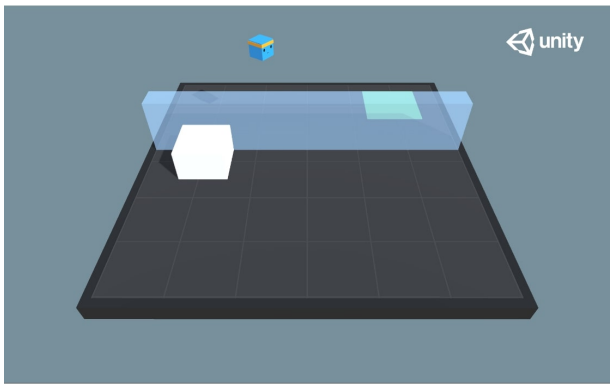

Figure 3.2: ML-Agents example wall jump environment.

agent was able to consistently succeed in given mission (Figures 3.3 and 3.4) .

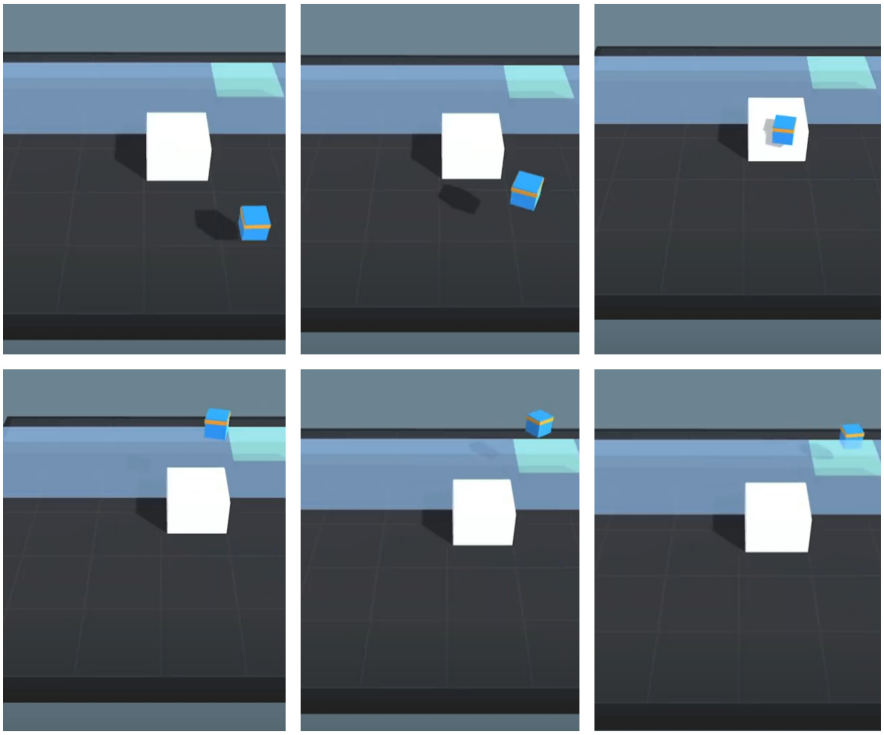

Figure 3.3: ML-Agents example wall jump agent overcoming a wall.

The purpose of the walker example is to train a simulated humanoid agent (Figure 3.5) to control the movements of its joints in order to learn to walk without falling and reach a target.

The approach involves setting up a 3D simulated environment where the agent (walker) can take actions to control its movements. The environment includes physics and dynamics that simulate the behavior of a realistic humanoid. The agent must learn a policy that allows it to walk or run towards a target.

The agent has knowledge of position, velocity and angular velocity of each limp along with the target direction. The agent is rewarded while moving towards the goal target, maximized when moving straight towards it. The agent can control the strength applied to its joints. Realistic physics are integrated within the simulation, using the Unity physics engine. Using the PPO algorithm to train our agent, we get descent results (Figure 3.6).

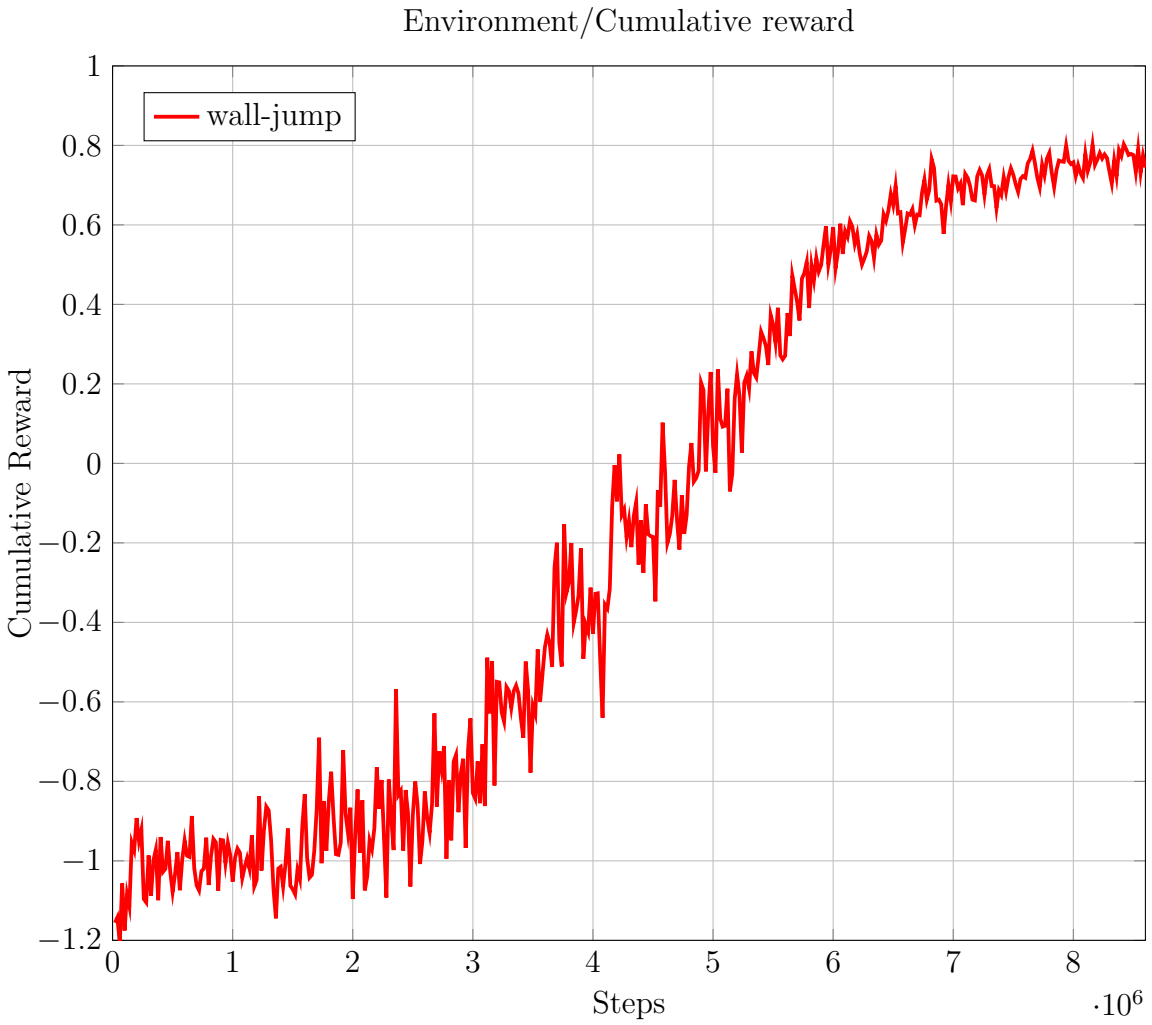

Figure 3.4: Environment/cumulative reward graph of wall-jump example

The key idea in this thesis is to get a combination of those two environments [7]. Having a complicated 3D humanoid agent learning how to walk and at the same time overcome the obstacles ahead.

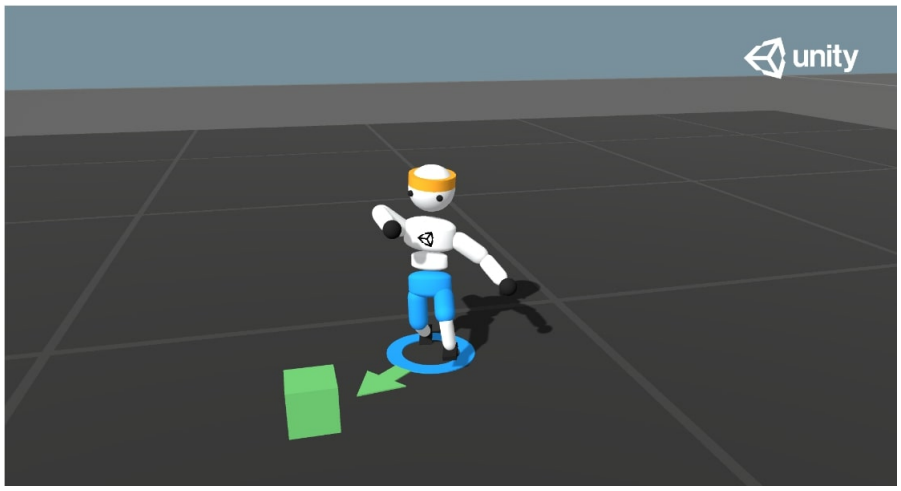

Figure 3.5: ML-Agents example walker environment.

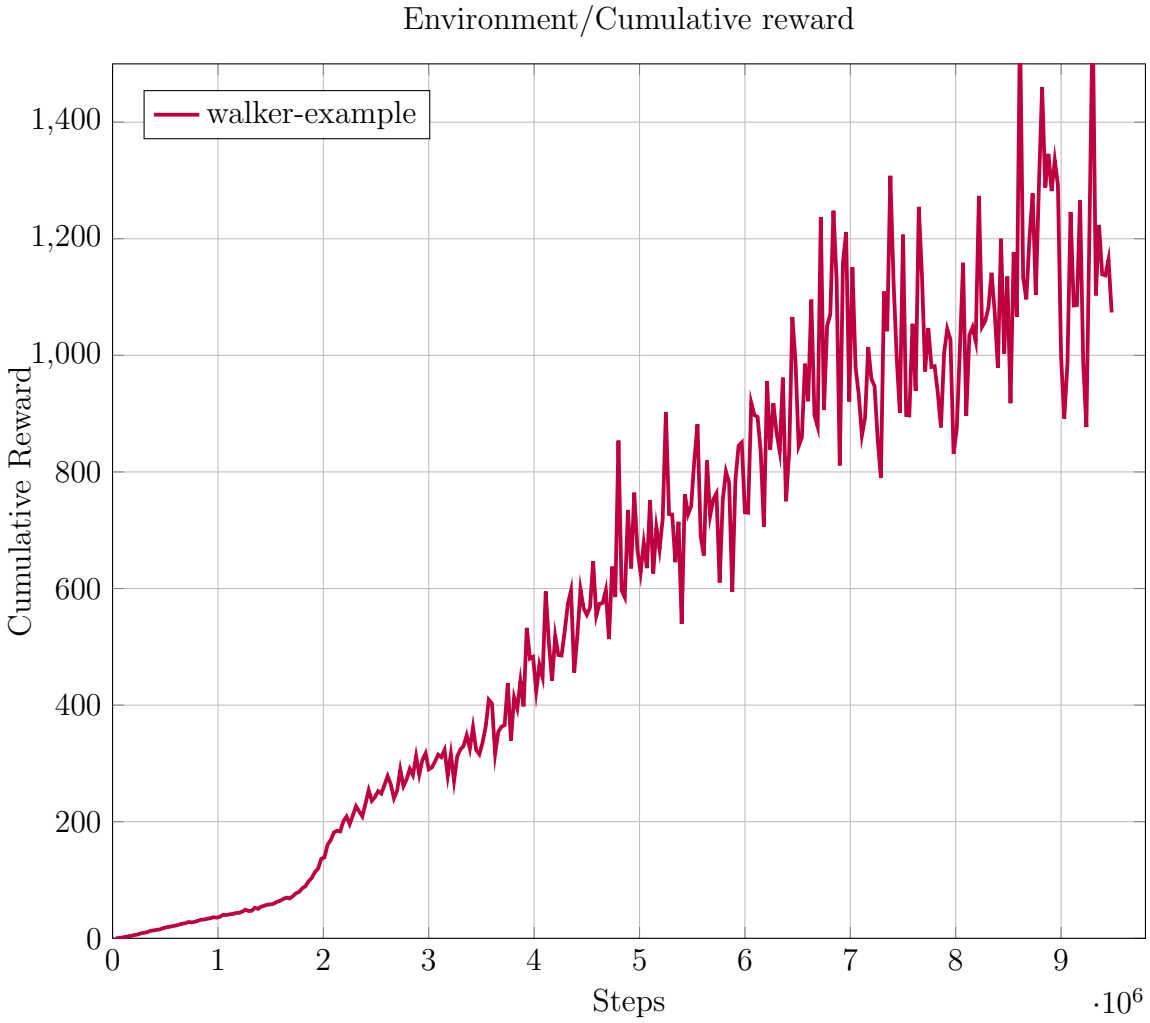

Figure 3.6: Environment/cumulative reward graph of walker example

Chapter 4

Approach

4.1 Setup

To commence our experiments several crucial steps need to be taken beforehand. We must ensure the successful installation of all required programs and libraries on our system. A need to establish a suitable environment is generated and training algorithms have to be implemented. This chapter will provide a thorough exploration of this process, addressing each step in detail.

4.1.1 Installation of Programs and Libraries

To initiate this process, several essential programs must be successfully installed on our system:

1. **Anaconda:** As mentioned earlier, Anaconda serves as our project manager. It allows us to create an isolated environment, in which all our libraries will be installed. This isolated environment helps manage dependencies and ensures the installation of the correct versions of libraries.

2. **Python:** Python is an exceedingly popular programming language, making it nearly indispensable for Reinforcement Learning, due to its vast array of libraries and comprehensive documentation.

3. **Unity:** We will be utilizing the Unity engine, renowned for its robust physics engine and the wide array of assets that enable us to create highly realistic simulations.

4. **ML-Agents:** ML-Agents is an open-source toolkit designed for Unity, functioning as a crucial link between Unity and Reinforcement Learning algorithms.

5. **Required Python Packages:** In addition to the above, several other Python packages are necessary for data analysis and graph design.

For complete documentation and tutorials on the installation of these programs and libraries, please refer to the provided links in the Bibliography section at the end of this thesis [8],[9].

4.1.2 Environment

Building upon the Walker example within ML-Agents and leveraging some of its valuable assets, we embark on the creation of the environment that will be key to our work.

Our initial requirement is a 3D humanoid agent, and the one offered by the Walker example (Figure 3.5) is perfect for our task: a physics-based humanoid agent with 26 degrees of freedom (DOFs). These DOFs correspond to the articulation of the following body-parts: hips, chest, spine, head, thighs, shins, feet, arms, forearms and hands.

The agent's action space remains unaltered. It is able to perform 39 continuous actions, corresponding to target rotations and strength applicable to the joints, rendering it an exceptionally realistic model. Same goes for the reward function, which remains for the moment unchanged, checking if the agent is facing towards the target goal and rewarding the agent if moving towards it. The reward is maximized, when the agent is moving straight towards it. Walker's observation space, consisting of 143 variables, is also used for our first try, which consists of information related to position, velocity and angular velocity of each limp along with the target direction. Our discount factor (gamma) is set to 0.995, close to 1 to make the agent value future rewards more, leading to more farsighted decision-making. It is worth noting that all our modifications and enhancements will be implemented on the foundation of the Walker example's assets, ultimately transforming its behavior to cater to our specific requirements.

Now, we require an obstacle course in which our agent can undergo training. Our initial obstacle course comprises 11 obstacles, strategically arranged with varying levels of difficulty. These obstacles are positioned statically, with a scalable level of challenge. They are structured such that a low obstacle (Figure 4.1) is followed by a high one (Figure 4.2), and as we approach the course's conclusion, the challenges become progressively more demanding. The low obstacles increase in height, while the high obstacles are adjusted downward. The courses length is parallel to axis z, courses width parallel to axis x and obstacles' height parallel to axis y (Figure 4.3).

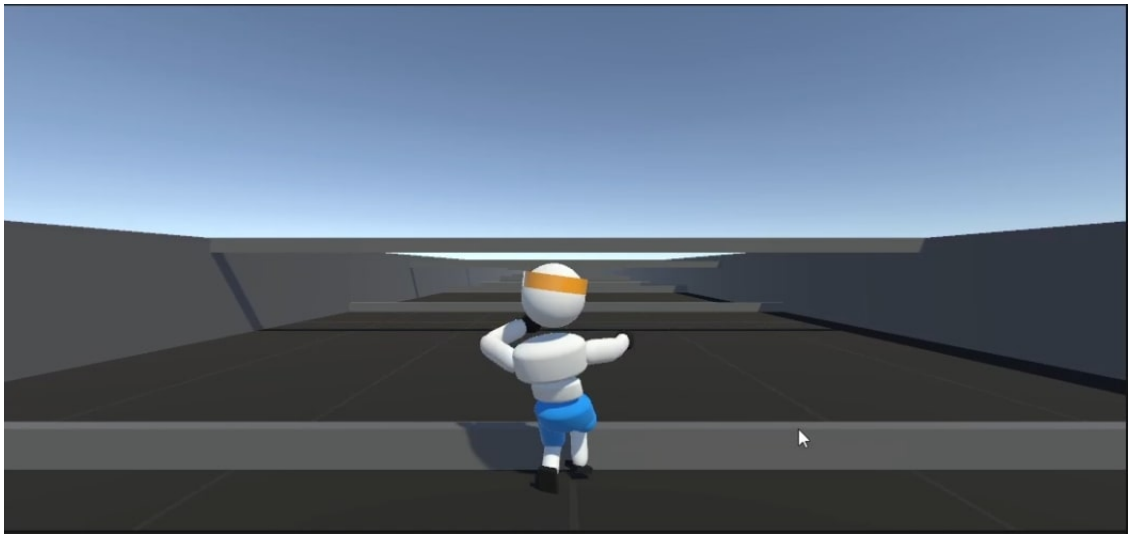

Figure 4.1: Our first environment's low obstacle.

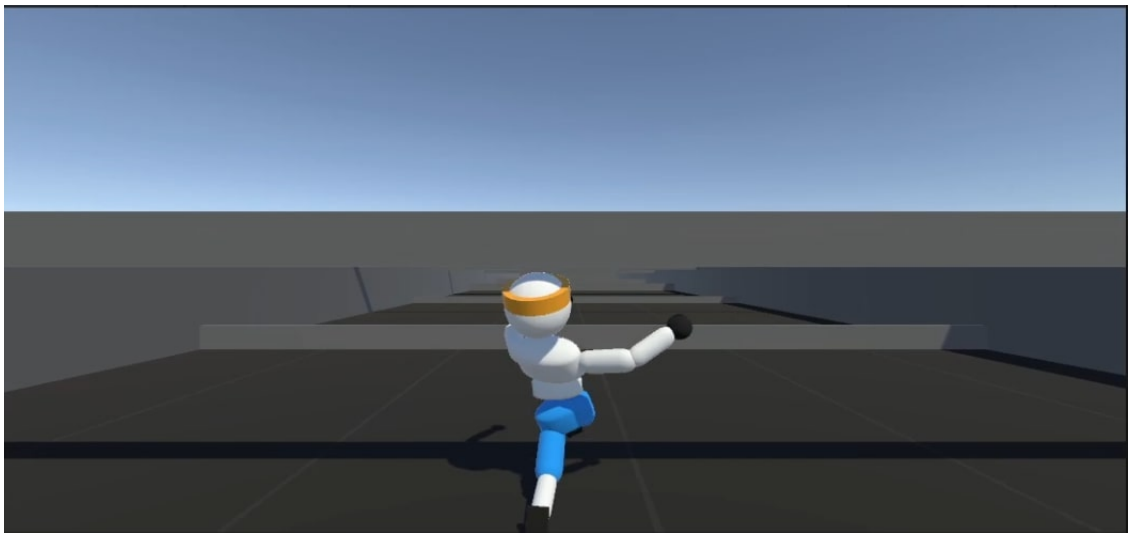

Figure 4.2: Our first environment's high obstacle.

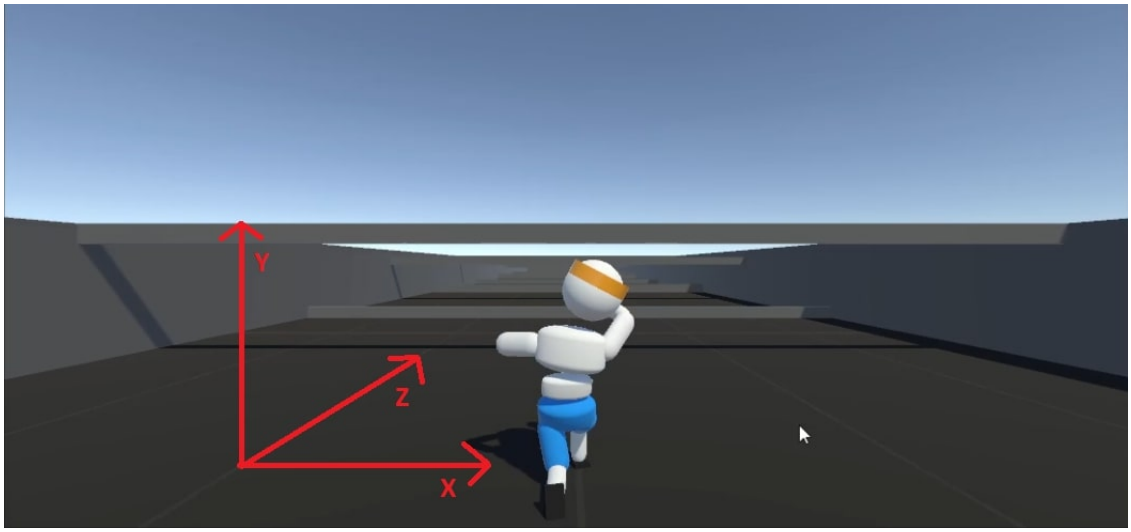

Figure 4.3: Our first environment's axis.

The interactive target, initially employed in the Walker's example for the agent to follow, has been transformed into a non-interactive static element, serving as the goal within our obstacle course. Due to parallelization and experience sharing multiple agents can interact with the environment simultaneously and share their experiences with each other, allowing each agent to explore the environment independently, thus

contributing different perspectives on how to solve the task. Recognizing that the use of multiple agents yields faster results, we opt to clone the agent 10 times, creating the first training environment for our agent (Figure 4.4).

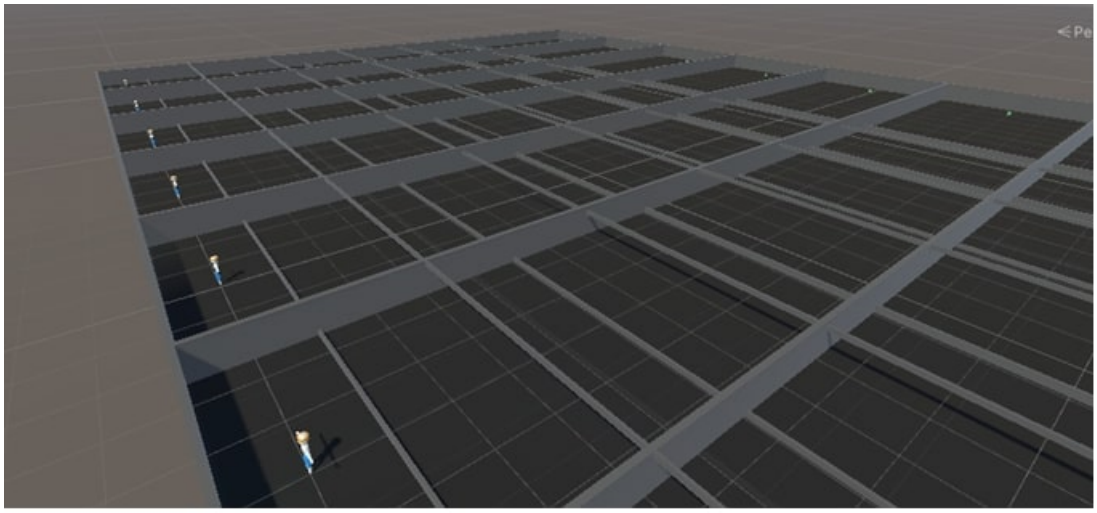

Figure 4.4: Our first environment created for agent's training sessions.

4.2 Initial Progress

For our agent, taking its first steps was relatively straightforward, as it was already pre-programmed to learn this fundamental skill. Learning how to walk was a rapid process, but challenges began to emerge upon encountering the first obstacle. Surprisingly, the agent quickly adapted to overcome this obstacle. However, after several hours of training, we observed a noteworthy outcome.

After approximately fifteen hours of training, our agent began to exploit the obstacles strategically, maximizing its rewards. Specifically, it devised a method to progress toward the target, while seemingly stuck in the initial obstacle of the course (video [3]). By doing so, it could achieve the maximum possible reward with each step as the agent according to Unity engine had speed towards the target. As the agent was blocked and leaning on the first obstacle, it never reached the target, nor lost balance. This approach resulted in remarkably long runs (reaching the max steps set for each simulation) and an unusually high accumulation of rewards (Figure 4.5).

While this behavior was technically correct, given our initial reward function, it did not align with our desired objectives. It was time to redesign our own reward function, and come up with one that would motivate the agent to learn the specific behaviors we required.

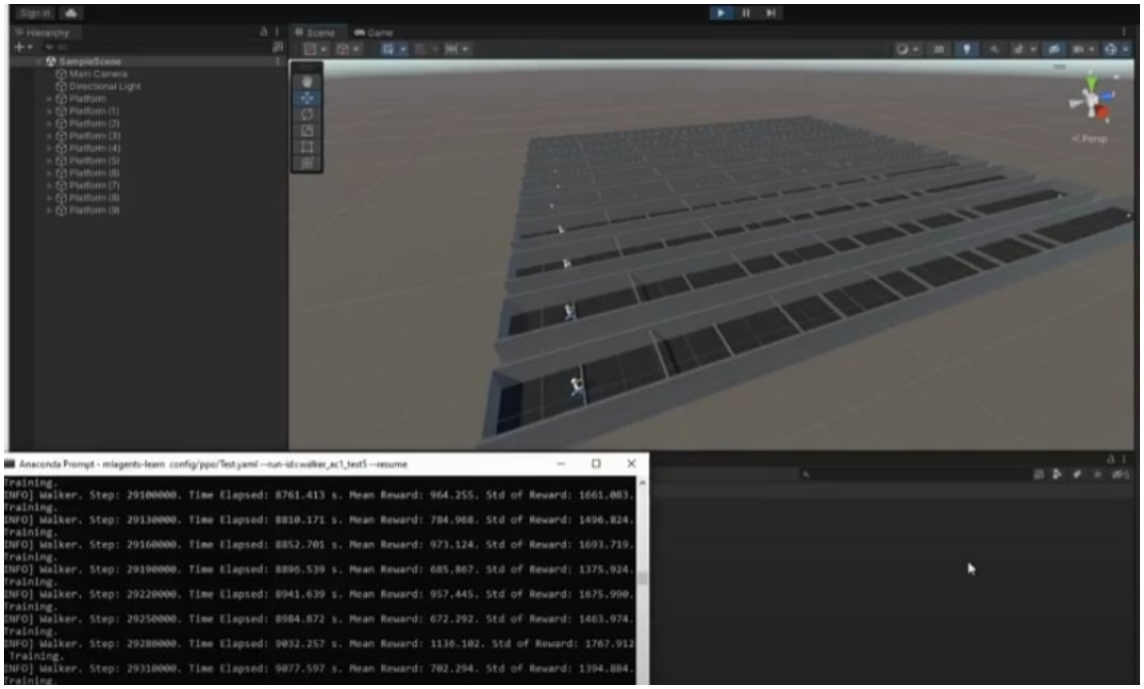

Figure 4.5: Agents getting stuck on first obstacle to maximize reward gained.

4.3 Designing a New Reward Function

Determining when and how we reward our agents stands as one of the critical aspects in reinforcement learning, and, in my experience, it is often one of the most challenging aspects to manage. Simplifying our reward function made it easier for the agent to exploit the function by learning unintended actions, while increasing its complexity, resulted in unexpected agent behaviors. Achieving the right balance in our reward function required a thorough understanding of the agent's goals and a process of trial and error, until we reached the desired outcome.

4.3.1 Reward Function Implementation

The precise objective for the agent is to reach the end of the obstacle course. Achieving this goal involves a combination of three essential components: directing the agent toward the endpoint, ensuring the agent maintains a certain level of velocity, and ensuring that the agent continuously progresses toward the goal. It is on these three fundamental principles that we have founded our reward function.

So our reward function consists of three parts according to these three rules:

1. **lookAtTargetReward :** This component provides a reward between zero and one, based on the alignment of the agent's vector with the direction of the goal. It is computed by taking the dot product of the agent's direction vector and the goal's direction vector. The dot product's range is from -1 to $+1$. Adding 1 shifts the range from 0 to 2, and multiplying by 0.5 scales it to a range from 0 to 1.
   Formula: $((|agentDirectionVector| * |goalDirectionVector| * cos\theta) + 1) * 0.5$
   where θ is the angle between the two vectors.

2. **matchSpeedReward :** This component delivers a reward between zero and one, dependent on how closely the agent's speed matches our desired speed (desired speed is set for our experiments to 5m/s). This allows us to control the agent's velocity. The reward calculation begins by measuring the Euclidean distance (denoted as D) between our agent's velocity and the desired goal velocity, limiting the range from 0 to the goal velocity. We then compute a value that decreases as the difference between the velocities increases.
   The formula employed is : $1 - (D/targetSpeed)^2$.
   Starting at 1, it decreases, as the discrepancy between the velocities widens. The square exponent emphasizes the reduction in reward as the velocities drift

further apart.
The final result represents a reward, value that approaches 1, when the actual velocity perfectly matches the goal velocity and approaches 0 as the deviation between the two velocities increases.

3. **distanceTraveledReward :** This component awards a reward between zero and one, based on the agent's proximity to the goal. The closer the agent is to the goal, the higher the reward, encouraging the agent to continuously move toward the end of the obstacle course. To calculate this reward, we divide the agent's z axis distance by the goal's z axis distance (goal is static so $goalZdistance$ is a fixed value), as our obstacle course is parallel to the z axis. A small constant (0.01) is added to prevent the reward value from starting at zero. The square root is applied to ensure that the reward does not approach zero too quickly at the beginning.
Formula: $\sqrt{agentZDistance/goalZdistance + 0.01}$

The mentioned components must be provided at the same time for our agent to work properly. So our final reward is the product of the three components, with a range from 0 to 1.

4.3.2 Outcomes of the New Reward Function

With the implementation of the new reward function, our agent demonstrated significant progress in learning and successfully overcoming static obstacles (Figures 4.7 4.8)

The plotted data (Figure 4.6) illustrates that the agent underwent training for 80 million steps (averaging approximately 2 million steps per hour, totaling around 40 hours of training). It achieved commendable results in terms of the cumulative reward gained. However, an anomaly is noticeable at around 60 million steps, as indicated in the graph, where there is a sudden drop in cumulative reward. This dip occurred due to a system crash that prevented the saving of the neural network tensor. After the system reboot, training resumed from a previous value.

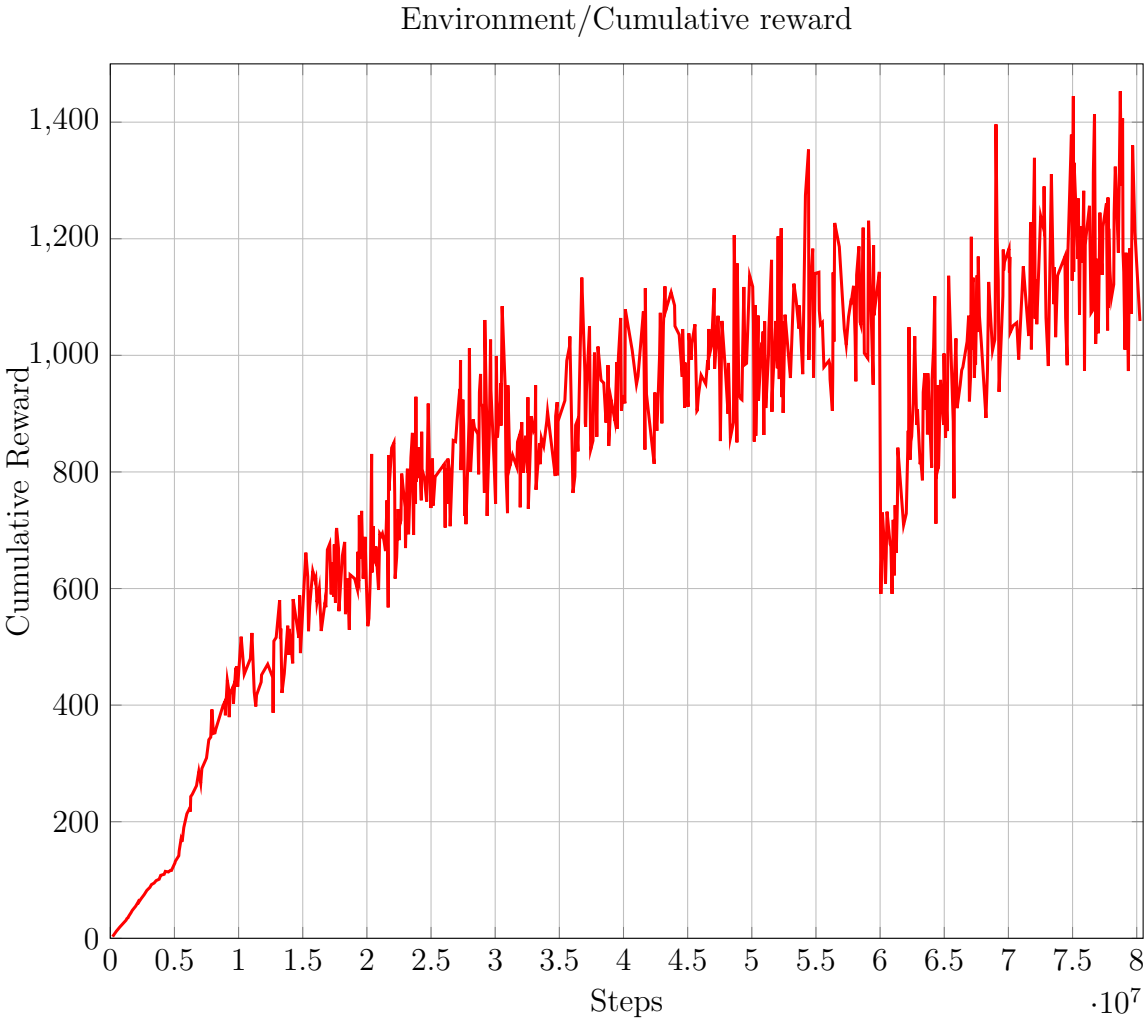

Figure 4.6: Environment/cumulative reward graph after reward function changes

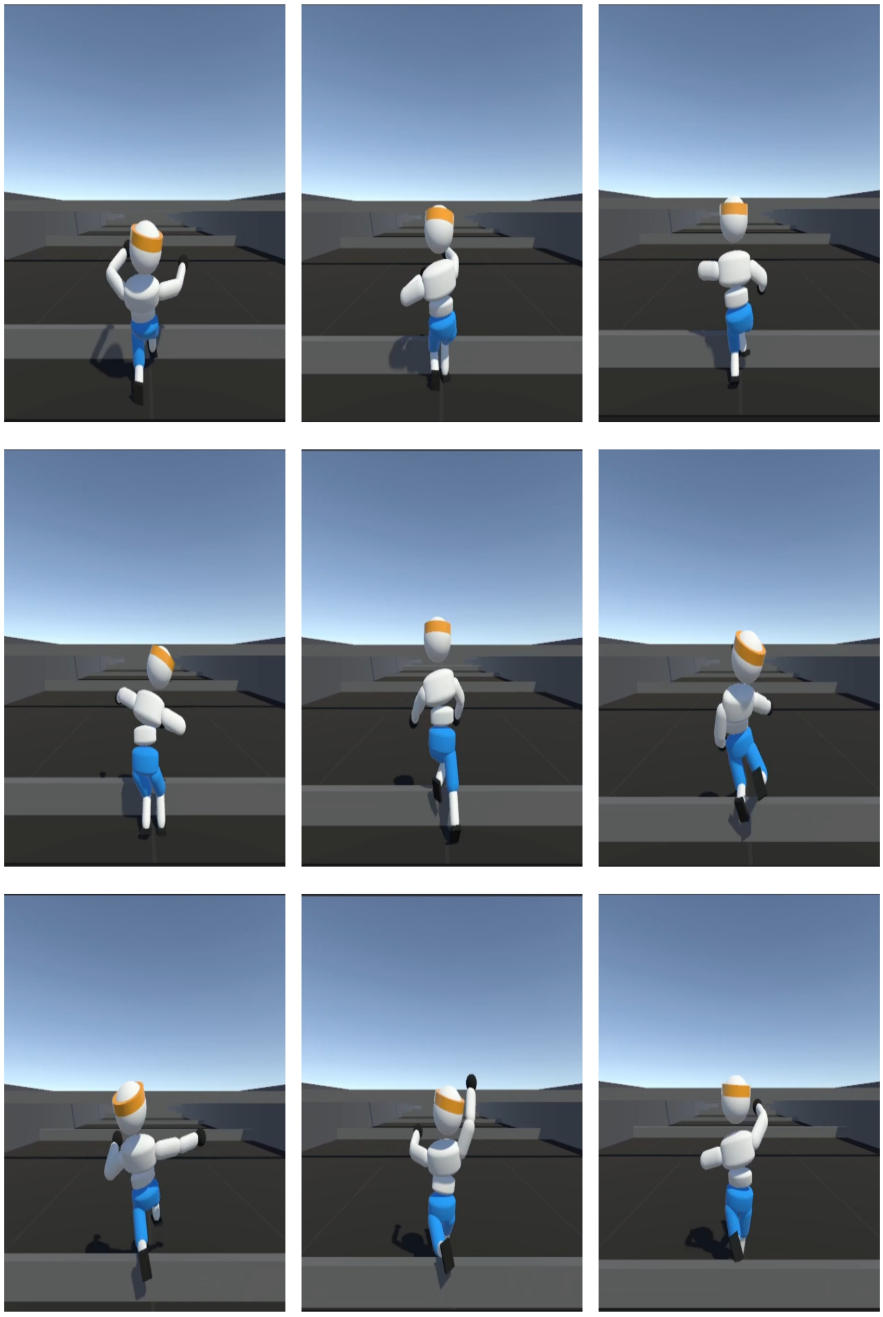

Figure 4.7: Reward function change. Low obstacle overcoming

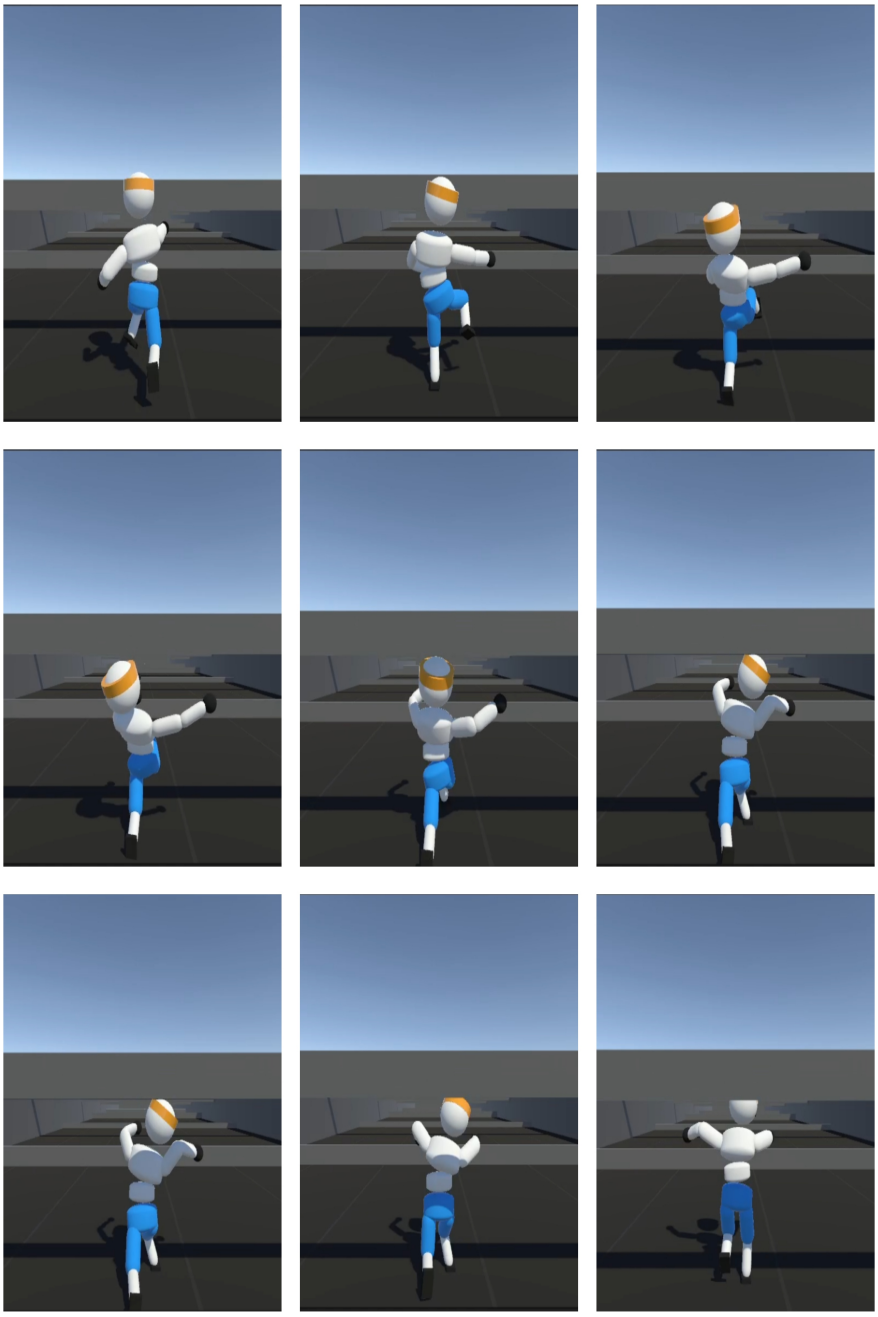

Figure 4.8: Reward function change. High obstacle overcoming

4.4 Observation Space

While our agent has become proficient at overcoming obstacles, a crucial consideration arises: What if the position of the obstacles were to change, even slightly?

Currently, our agent lacks the capability to discern the locations of obstacles, and its learning is solely based on repetitive exposure to the same course. Consequently, it remains ill-equipped to adapt to dynamic environments. To address this challenge and enable our agent to excel in dynamic settings, we must enhance its observation space, equipping it with the comprehensive information it requires.

4.4.1 Implementation of the Observation Space

In all living beings, there exists a crucial sensory ability, called proprioception. Although it may not be widely recognized, because we often take it for granted, it is the very sense that enables us to move and accomplish tasks. Proprioception, in essence, is the sensory capacity that allows us to perceive the location, movement, and actions of different parts of our body. It encompasses a complex range of sensations, including the perception of joint positions, movements, muscle forces, and efforts. These are precisely the elements our agent requires, in addition to information on the goal direction and awareness of the obstacles it must confront.

The extent to which our agent possesses information is crucial. Too little of it can lead to insufficient data, while an excess of information may necessitate significantly more training time, potentially hindering the learning process, due to an inability to discern which details are useful. So, we have to find a balance.

Consequently, our observation space consists of the following:

- For each part of the agent's body (16 body parts), we acquire its location (3), velocity (3), angular velocity (3), and whether it is in contact with the ground (1). For 13 of those (excluding the hips and the hands) we also acquire rotation (4) and strength applied (1).
  Total number of variables for body parts 225.
- The distance between our agent and the goal of the obstacle course (1).
- The goal position (3).
- The agent's average velocity (3).

- The desired velocity for our agent to match (3).
- The rotation of the agent's hips (4) and head (4).
- The position of the next obstacle in the course, considering its coordinates on the y and z axes, as well as its height (3).

This comprehensive observation space comprises a total of 246 variables, providing our agent with the necessary information for its task.

4.4.2 Observation Space Implementation Results

The enhancement of the observation space had a significant positive impact on the agent's training times. By providing information about obstacles, the training process became approximately three times faster, when utilizing the same static obstacle course. Importantly, the improvements did not seem to produce any noticeable alterations in the learned behavior of the agent.

The agent consistently attains an average reward of about 500, which signifies its capability to overcome obstacles effectively. With these enhancements, this level of proficiency can be achieved after only 5 million steps (equivalent to 2.5 hours of training), whereas without these improvements, it required approximately 13 million steps (or 7.5 hours) to reach a similar level of competence (Figures 4.9 and 4.10).

Figure 4.9 displays the cumulative reward in the environment, where the red curve represents the previous setup (Figure 4.6) and the green curve represents the same graph after the observation space changes.

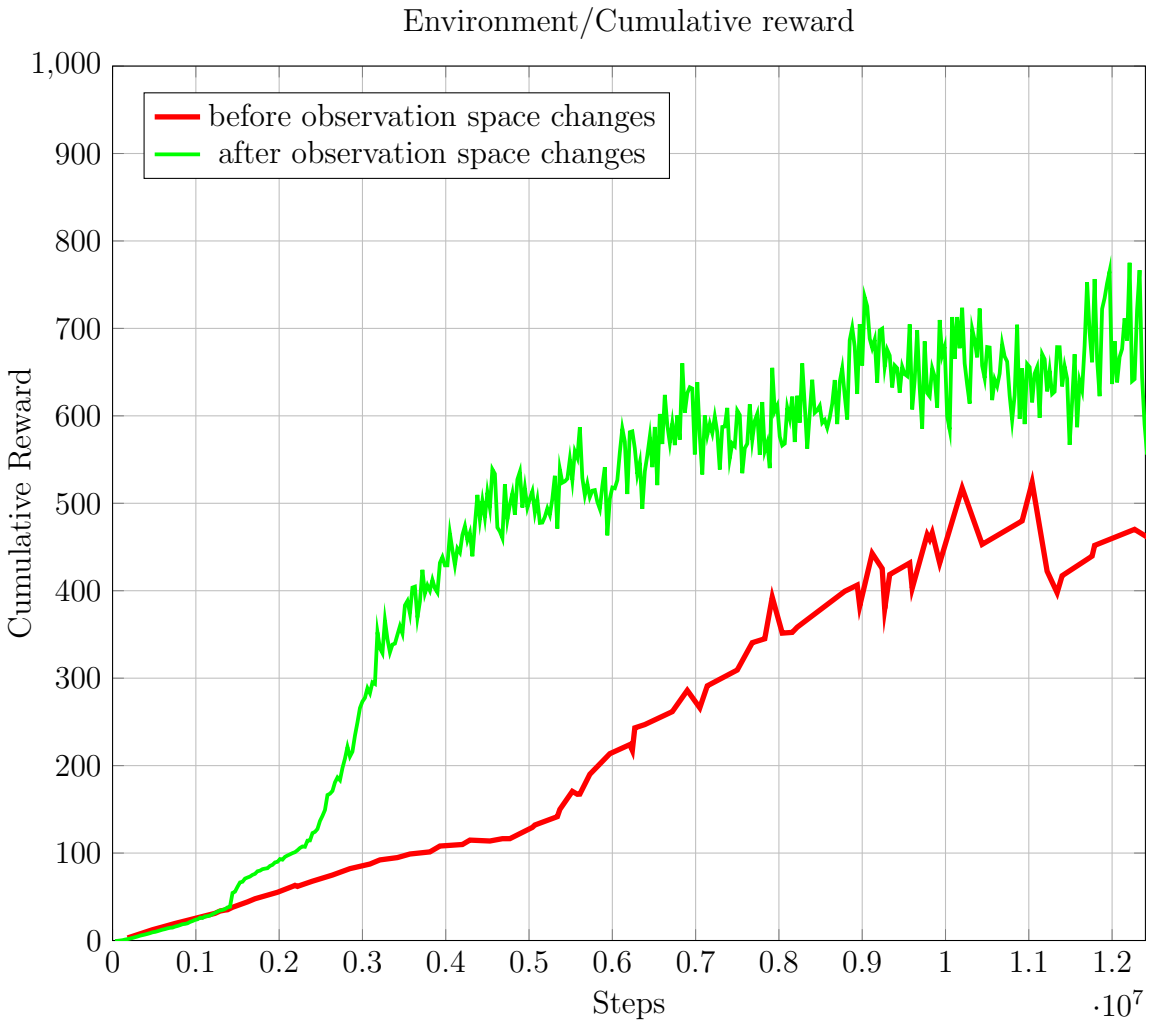

Figure 4.9: Comparison between reward accumulation with new and old observation space

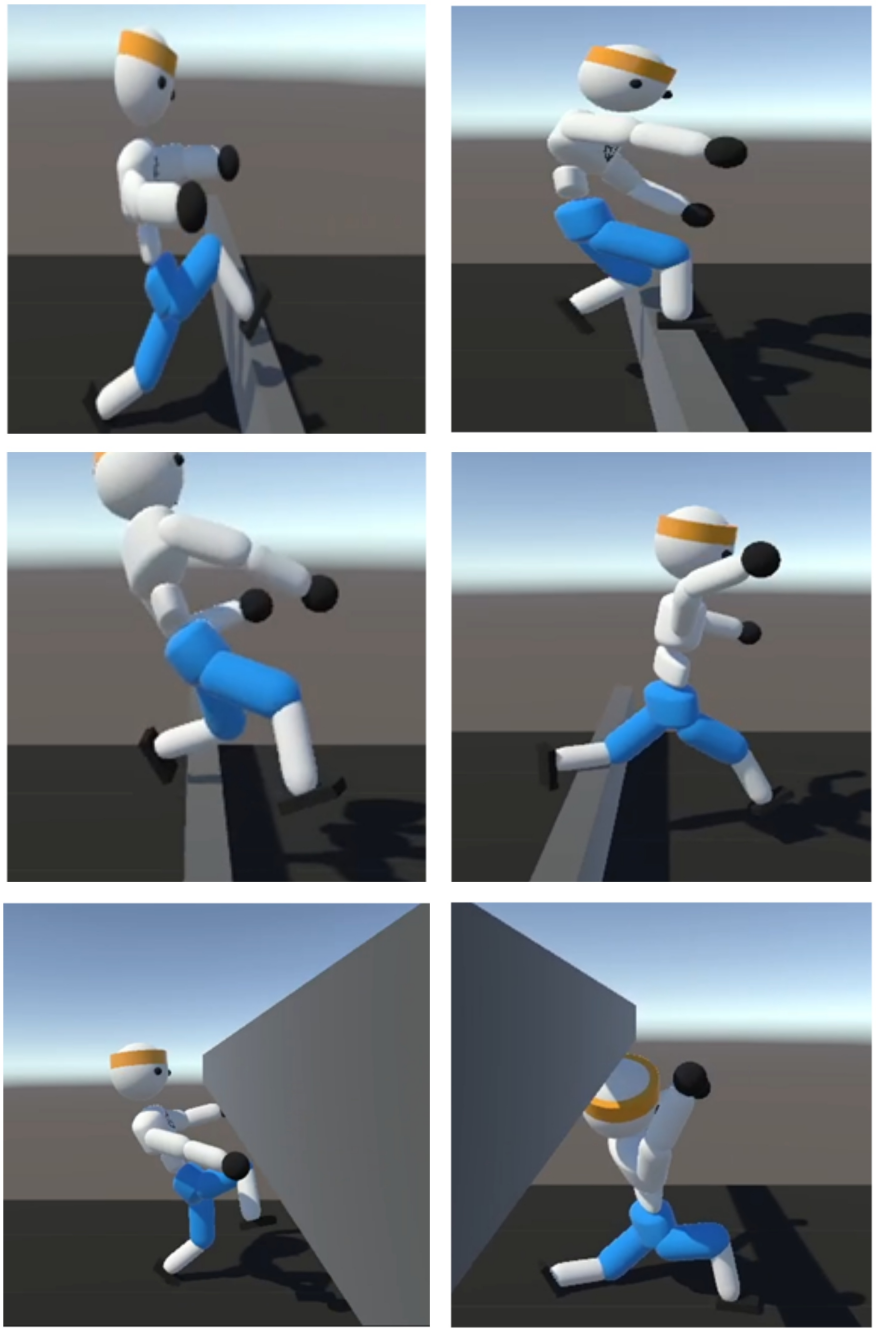

Figure 4.10: Agent getting over a low obstacle and under a high one

4.5 Implementation of Environmental Changes

A completely static environment does not align with our objectives, and it is essential to introduce alterations to our obstacle course. We aim for each episode to present a different obstacle course, necessitating our agent to learn a policy that relies on observations rather than relying solely on accumulated experience from the same course.

To achieve this, the positions and heights of our 10 obstacles are randomized every time the environment resets. This approach ensures that a distinct obstacle course is generated at the commencement of each episode. Each of the 10 obstacles individually receives values for these variables within the specified bounds, allowing us to assess entirely different scenarios as the need arises.

However, the last obstacle remains static, serving a dual purpose. Firstly, we check the height that our agent is not able to surpass due to its body properties. Overcoming this obstacle demands complex actions, and witnessing the agent successfully overcoming it would be a remarkable achievement. Secondly, it serves as an indicator of whether the agent successfully accomplishes its task, reaching the end of the obstacle course. The implementation of our observation space allows us to detect when the agent's torso surmounts this final obstacle, triggering a warning that indicates the absence of another obstacle in the sequence. As a side note, it's worth mentioning that printing information during training sessions is memory-intensive, so we've relied on Unity's warnings as a more resource-efficient option.

In Figure 4.11, we present an illustrative example of our new training environment.

These environmental changes have imposed a considerable challenge on our agent's performance, as the difficulty level has increased significantly. The graphs obtained after the implementation of these changes aptly reflect this heightened complexity.

In Figure 4.12, we observe the outcomes of four distinct training sessions using the same settings, as well as the graph after observation space changes from figure 4.9), to use as reference. It is immediately apparent that our learning time has significantly increased. This extended learning time is a foreseeable consequence, considering that our agent's task has become notably more challenging with the obstacle course changing in every episode.

Another noteworthy observation is the sharp decline in cumulative reward seen in training sessions 2 and 4. This decline is attributed to the phenomenon known as

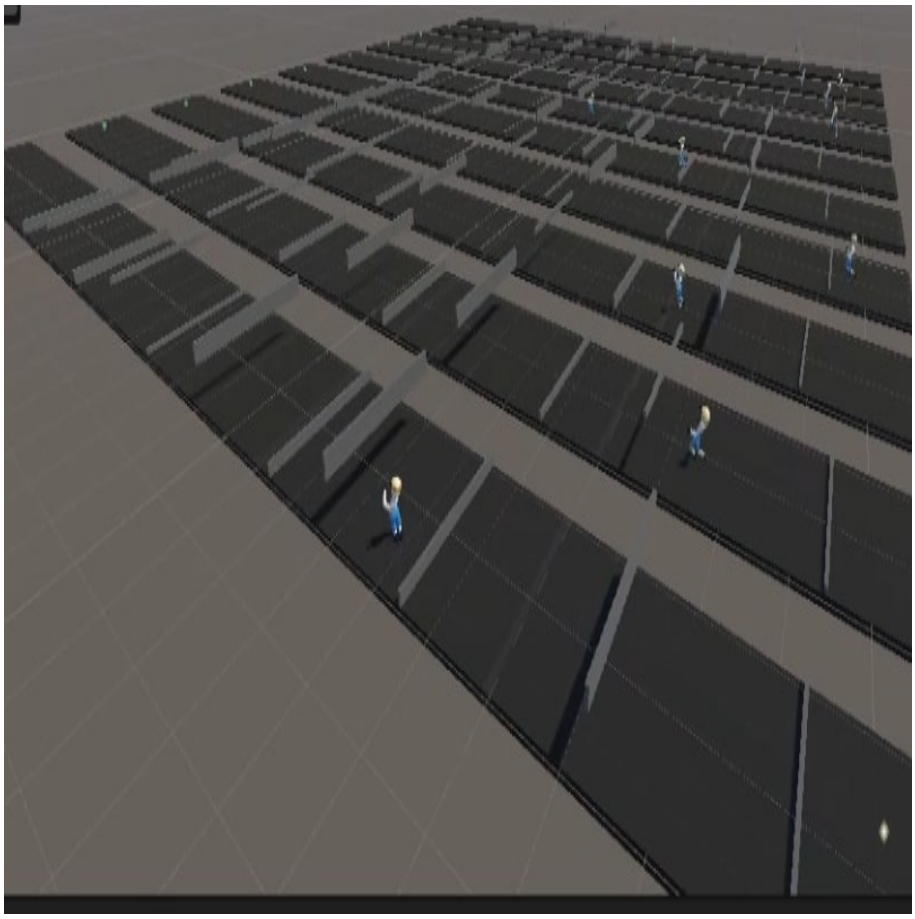

Figure 4.11: Different scale and position of obstacles for each agent

catastrophic forgetting, a challenge that the PPO algorithm can encounter (further elaborated in the background chapter).

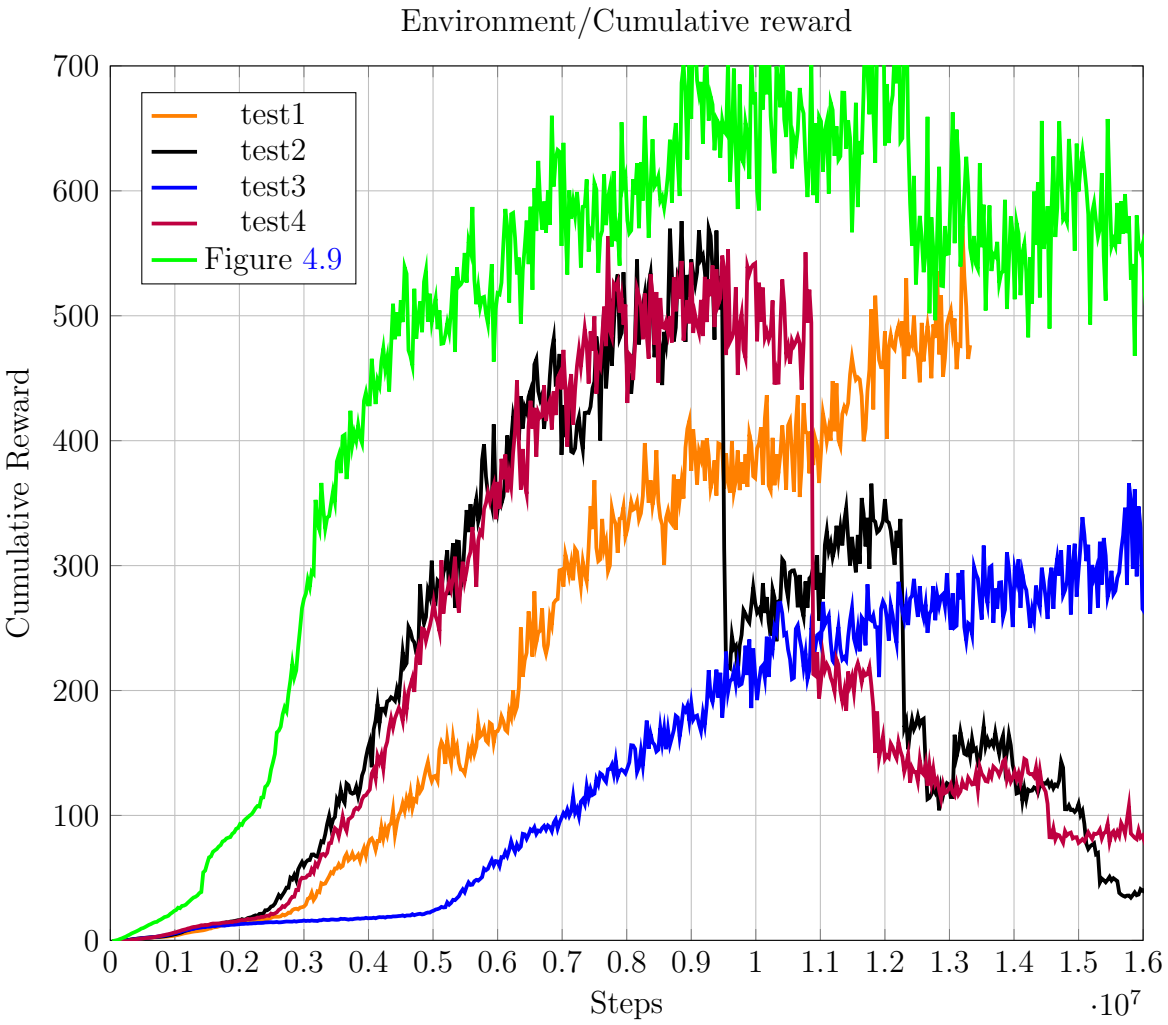

Figure 4.12: Training sessions after changes in obstacle course

4.6 Tuning PPO Hyper-parameters

To enable our agent to effectively adapt to the new challenges posed by the evolving problem difficulty, we must dive deeper into the optimization of the Proximal Policy Optimization (PPO) algorithm's hyper-parameters. Fine-tuning these hyper-parameters for our specific task is crucial for our agent's learning process. Although we discussed PPO hyper-parameters in the background section, we will now explore them from an experimental perspective and examine their precise roles.

Let's take a sample from our project (Figure 4.13) and analyse them with testing.

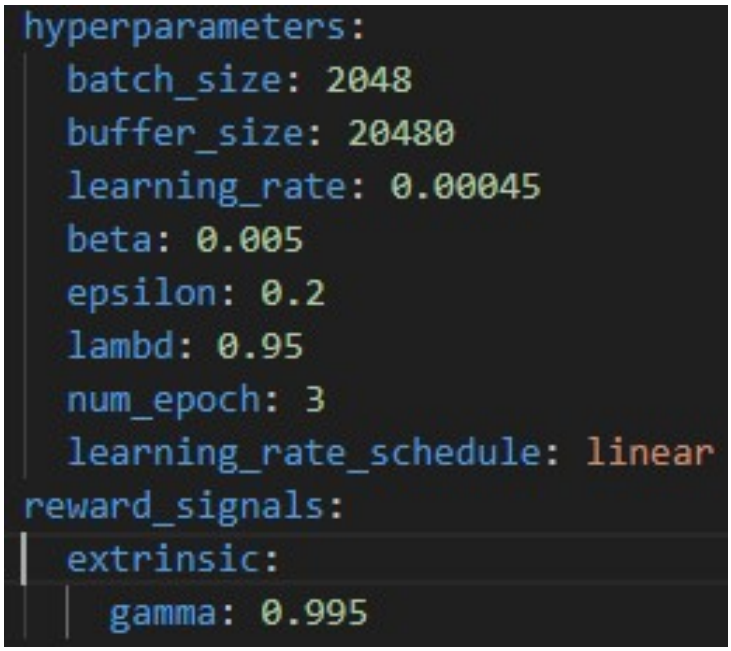

Figure 4.13: PPO hyper-parameters

The first three parameters (batch_size, buffer_size and learning_rate) are the most influential ones.

Batch Size (batch_size = 2048):
A larger batch size contributes to a more stable policy update. However, the chosen value of 2048 is also mindful of our system's memory constraints, ensuring that the batch size does not overextend to more than 80% of available memory. This balance aims to promote stable training while optimizing the utilization of system resources.

Buffer Size (buffer_size = 20480):
The buffer size determines the size of the experience replay buffer, which stores past experiences for training the policy. A larger buffer size allows the agent to learn from a more diverse set of experiences, which can improve sample efficiency and stability during training. The buffer size has to be $batch - sizeused * number\_of\_replays$, so $10\times$ provides a good balance between diversity and memory efficiency.

Learning Rate (learning_rate = 0.00045):
A smaller learning rate (0.00045) is chosen to ensure stability, allowing the model to make smaller, more controlled steps towards an improved policy. In Figure 4.14 we can observe the influence of the learning rate on our environment's cumulative reward graph. The purple curve represents our fastest learning rate (0.00099), causing our plot to rapidly ascend to its peak before undergoing a swift decline. In contrast, the orange curve ($learning\_rate = 0.00005$) illustrates the effects of our slowest learning rate, where minimal learning progress was evident, even after eight hours of training.

Beta (beta = 0.005):
The value of 0.005 is the median of allowed values, so it strikes a balance between promoting exploration and exploiting the current policy.

Epsilon (epsilon = 0.2):
The value of 0.2 helps to prevent overly large policy updates that might destabilize training. It ensures that the policy updates stay within a relatively safe range.

Lambda (lambda = 0.95):
With the value of 0.95 the algorithm gives more weight to the actual rewards when estimating advantages, while providing a reasonable balance between bias and variance.

Number of Epochs (num_epoch = 3):
A moderate value like 3 strikes a balance between using enough computation to learn from the data without over fitting to it.

Learning Rate Schedule (learning_rate_schedule = linear):
The learning rate schedule dictates how the learning rate changes over time. In this case, it follows a linear schedule, meaning the learning rate decreases linearly as training progresses. Such schedules help fine-tune the balance between exploration and exploitation over time.

Gamma (gamma = 0.995):
Gamma is the discount factor used to weigh the value of future rewards. It varies between 0 and 1. With a high gamma (0.995) we emphasize in long-term rewards instead of short term ones.

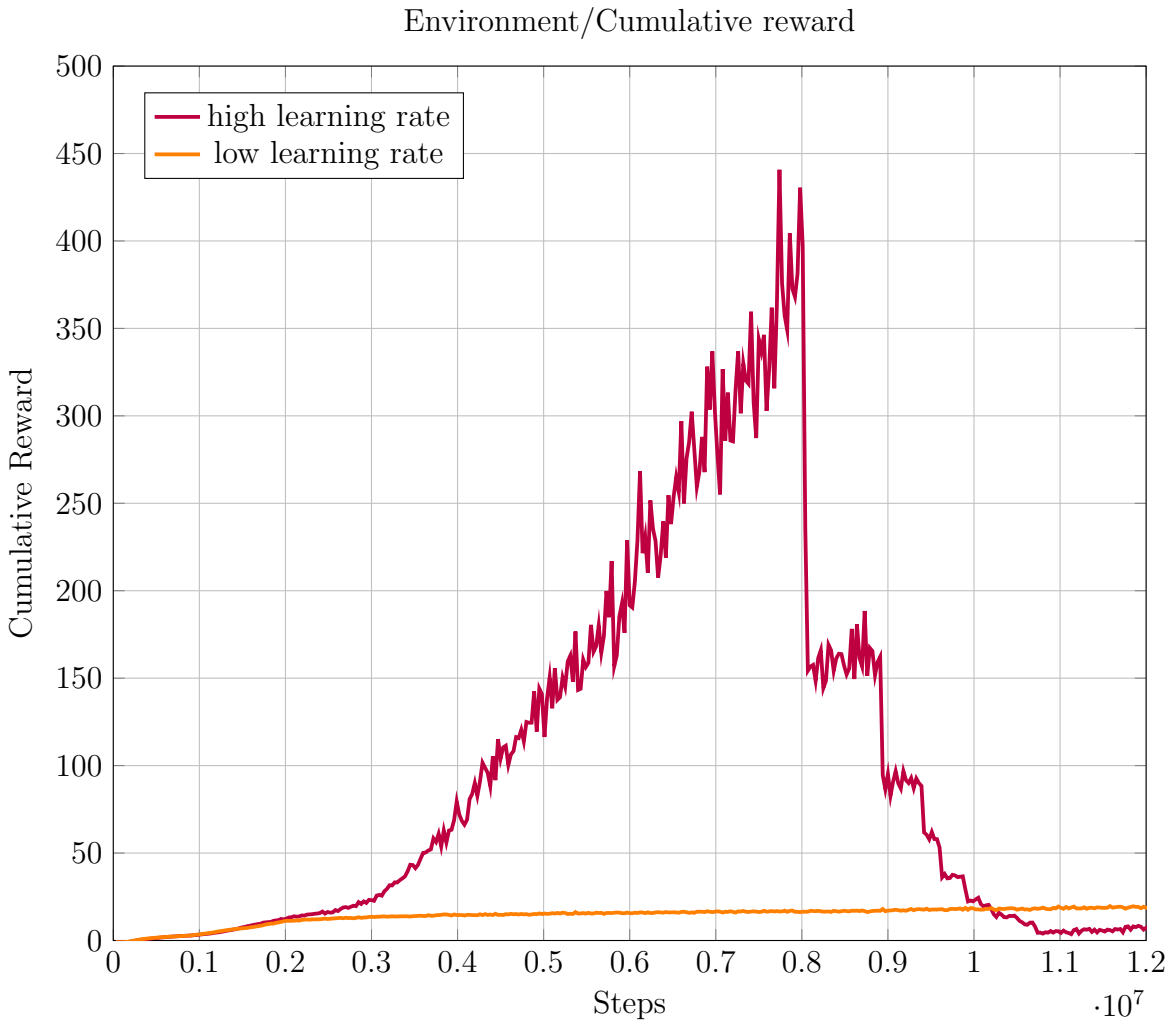

Figure 4.14: Influence of the learning rate on our environment's cumulative reward graph.

Chapter 5

Final Results

With the knowledge our experiments provided, extensive trial and error, and countless hours of simulations (Figure 5.1), we have succeeded in constructing several models with commendable performance.

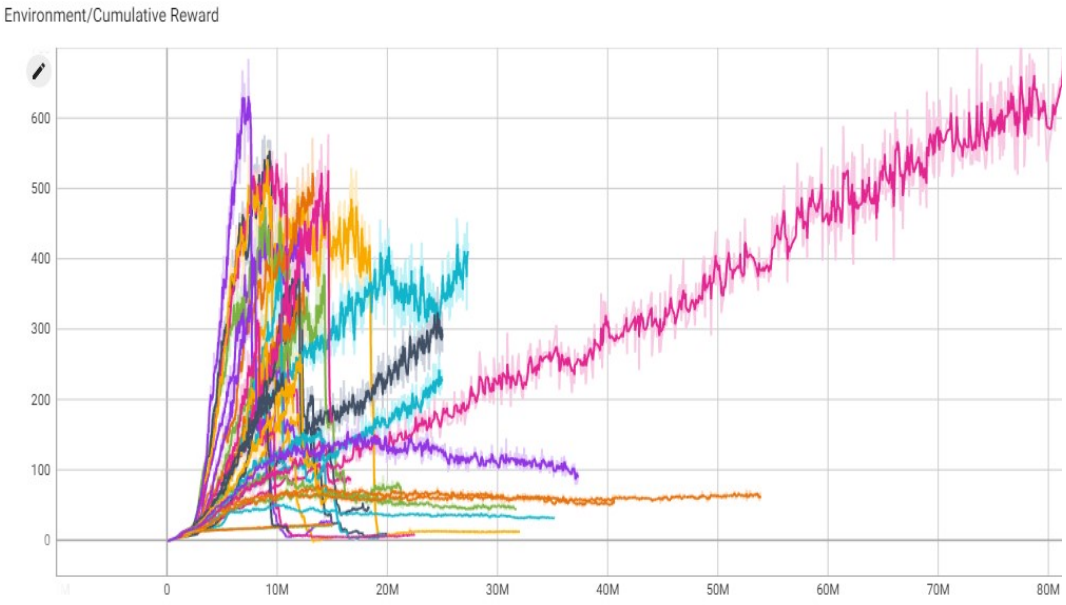

Figure 5.1: A fraction of the experimental simulations we run.

From those we will choose and analyze our best one. Having the most training hours and no interruptions or problems in the process, it's a model worth noticing (Figure 5.2).

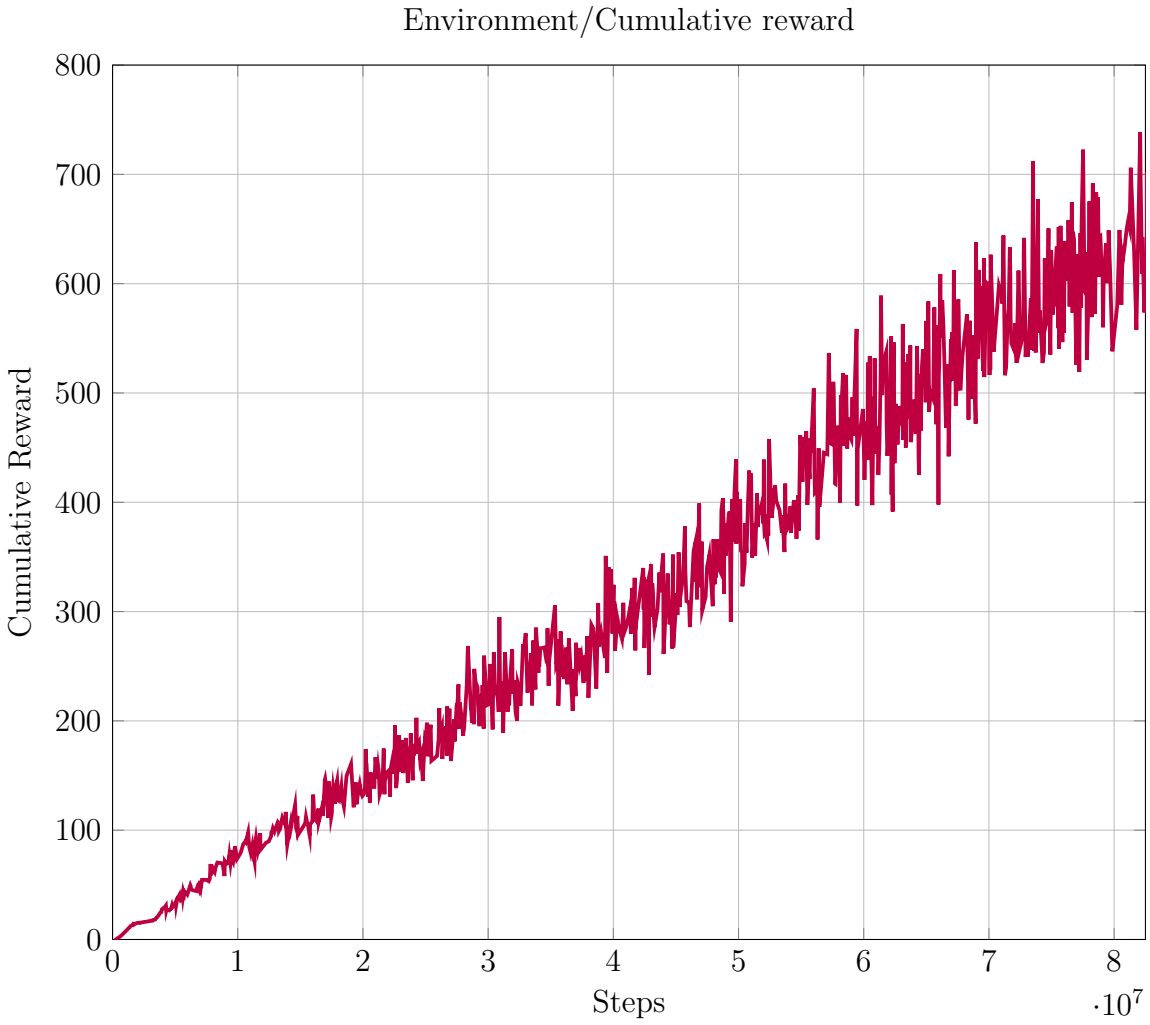

Figure 5.2: Our best training session (more than 40 hours of training).

After more than 40 hours of training, our agent demonstrates a consistent ability to overcome obstacles within the moderately challenging environment we've designed, effectively fulfilling its intended task.

Some samples of agent behavior are presented below, but there are more in the thesis video archive [3].

At the beginning of each episode, our agent tends to promptly drop to its knees (Figure 5.3). The reason for this behavior could be better stability or simply to navigate more efficiently under high obstacles, although we can only speculate.

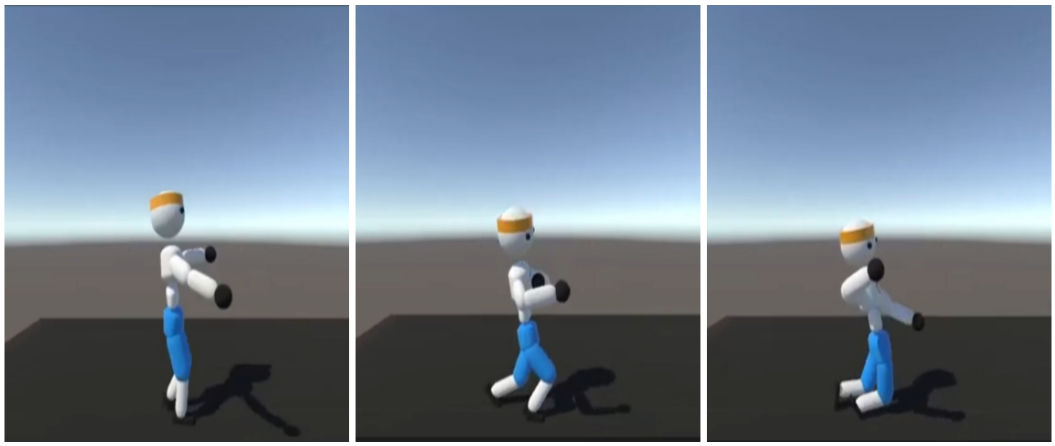

Figure 5.3: Agent's behavior when starting the course.

Our agent is already at its knees from the start, making it easier to deal with high obstacles. Notice that the agent is arching its head backward, followed by its torso, before reverting to its "normal" walking position (Figure 5.4).

When dealing with low obstacles, a fascinating observation emerges. Our agent, typically in a knelt position, anticipates the approach of a low obstacle and initiates a sequence of jumps even before reaching it, akin to a triple jump athlete. Following the obstacle's traversal, the agent seamlessly transitions back to its regular "walking" position (Figure 5.5).

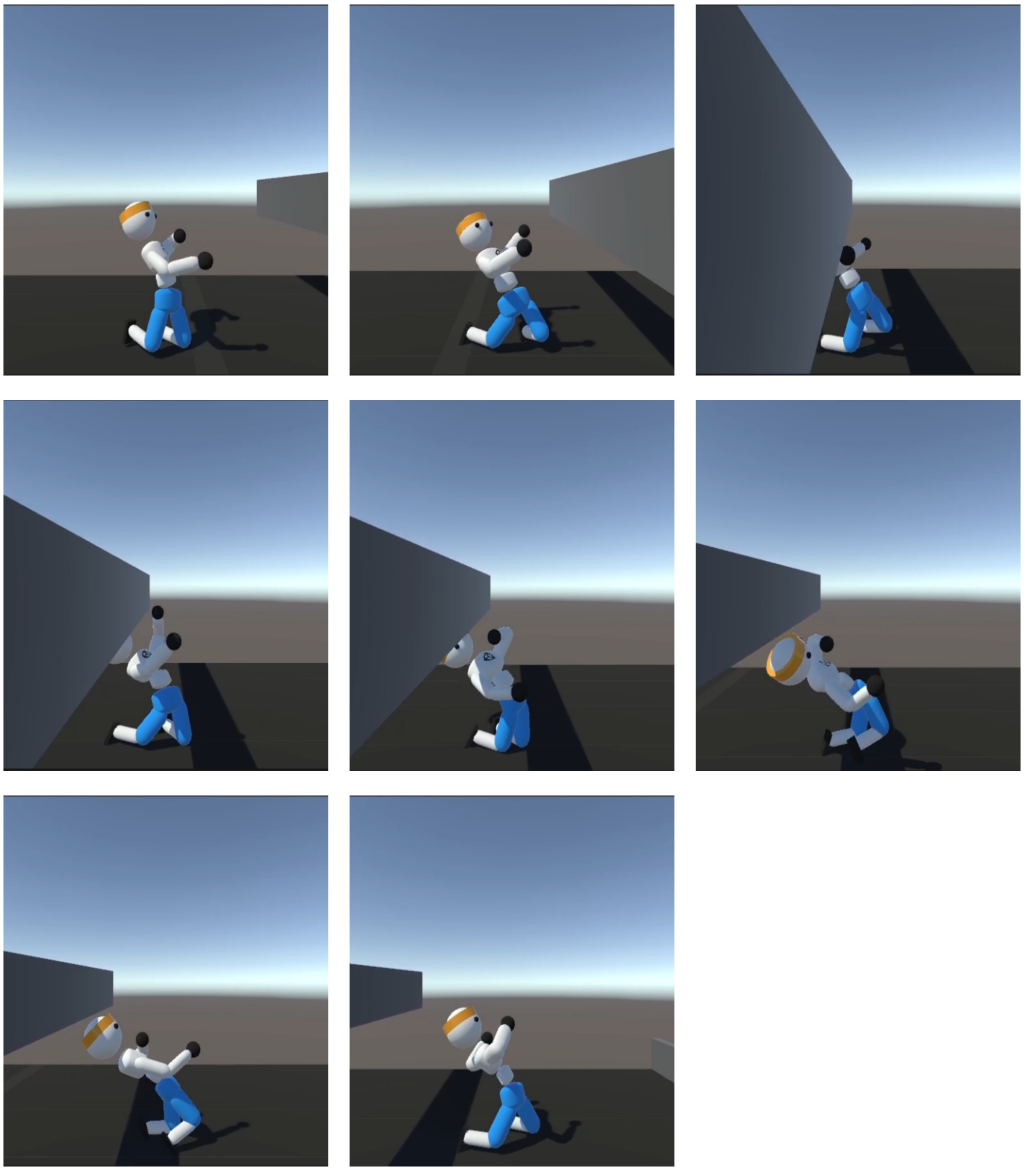

Figure 5.4: Overcoming a high obstacle.

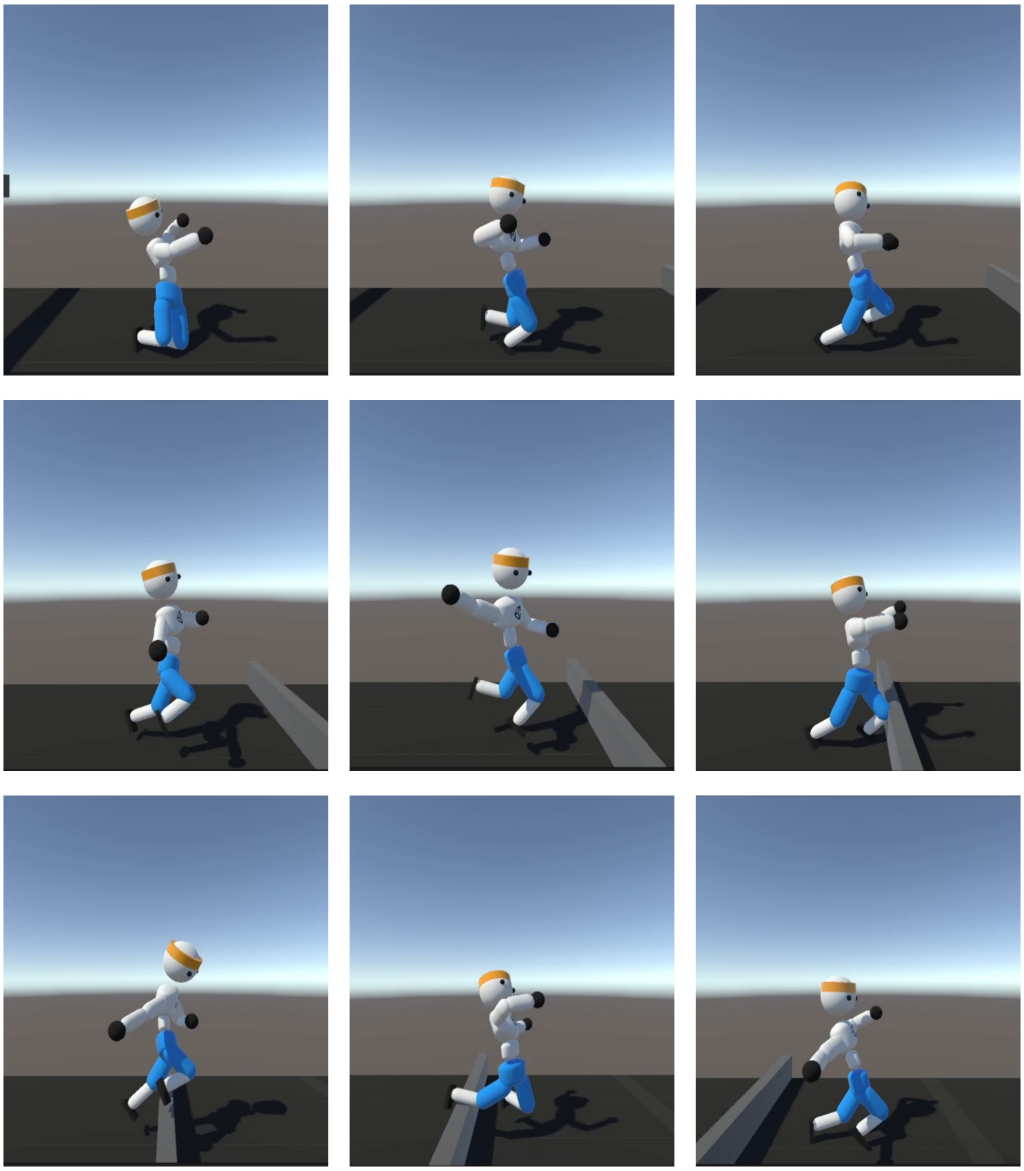

Figure 5.5: Low obstacle overcoming.

Chapter 6

Conclusion and Future work

6.1 Summary

In this concluding chapter, we revisit our endeavor into reinforcement learning for obstacle overcoming using a three-dimensional humanoid model. The goal of this thesis was to create a 3D model able to overcome obstacles. To do so, starting from a walking example, we created the training environment, set the goals and the rewards, and made every change needed for our agent to succeed in overcoming a course filled with obstacles. Based on our experiments, if the goals are clear, and the resources are sufficient, the capabilities of reinforcement learning seem limitless.

6.2 Future Work

The way the project is implemented, brings forward opportunities, where several aspects could be improved. Some future work would be to make our agent more like a human being. We could provide the agent with machine vision to identify the obstacles, get the information needed and then pass it to the observation space.

Another great improvement would also be to add different kinds of obstacles, so that our agent has the ability to learn how to navigate around them, when possible. This would make the task a lot harder though.

Lastly, a lot of similar stages can be set and a lot of similar tasks can be achieved, including taking trained models and applying them to real world. That would require a substantial amount of resources, so trained models' value has to be essential.

Bibliography

[1] Logan Engstrom et al. "Implementation matters in Deep RL: A case study on PPO and TRPO". In: *International conference on learning representations*. 2019.

[2] Robert M French. "Catastrophic forgetting in connectionist networks". In: *Trends in Cognitive Sciences* 3.4 (1999), pp. 128–135.

[3] *Google drive with this project's videos*. https://drive.google.com/drive/folders/15MFLiPteaG5NigCnQL6YCKsnQAoay9AR?usp=sharing.

[4] Arthur Juliani et al. "Unity: A general platform for intelligent agents". In: *arXiv preprint arXiv:1809.02627* (2018).

[5] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[6] V Krstić and I Mekterović. "Unity as a Physics Simulator: Calculating Mean Free Path for Hard Disk Gas". In: *2021 44th International Convention on In-*

formation, Communication and Electronic Technology (MIPRO). IEEE. 2021, pp. 613–616.

[7] *ML-Agents examples*. https://github.com/Unity-Technologies/ml-agents/blob/release_18_docs/docs/Learning-Environment-Examples.md.

[8] *ML-Agents getting started*. https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Getting-Started.md.

[9] *ML-Agents installation guide*. https://github.com/Unity-Technologies/ml-agents/blob/release_18_docs/docs/Installation.md.

[10] *ML-Agents overview*. https://github.com/Unity-Technologies/ml-agents/blob/release_18_docs/docs/ML-Agents-Overview.md.

[11] Abhishek Nandy et al. "Unity ML-Agents". In: *Neural Networks in Unity: C# Programming for Windows 10* (2018), pp. 27–67.

[12] *PPO explanation and example article*. https://towardsdatascience.com/proximal-policy-optimization-ppo-with-tensorflow-2-x-89c9430ecc26.

[13] *PPO hyper-parameters*. https://github.com/yosider/ml-agents-1/blob/master/docs/Training-PPO.md.

[14] *PPO Open-AI*. https://openai.com/research/openai-baselines-ppo.

[15] Damien Rolon-Mérette et al. "Introduction to Anaconda and Python: Installation and setup". In: *Quantitative Methods Psychology* 16.5 (2016), S3–S11.

[16] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

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

[18] *The Beginner’s Guide to Deep Reinforcement Learning [2023]*. https://www.v7labs.com/blog/deep-reinforcement-learning-guide.