Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

ReinforcementDrive

Project Engineering

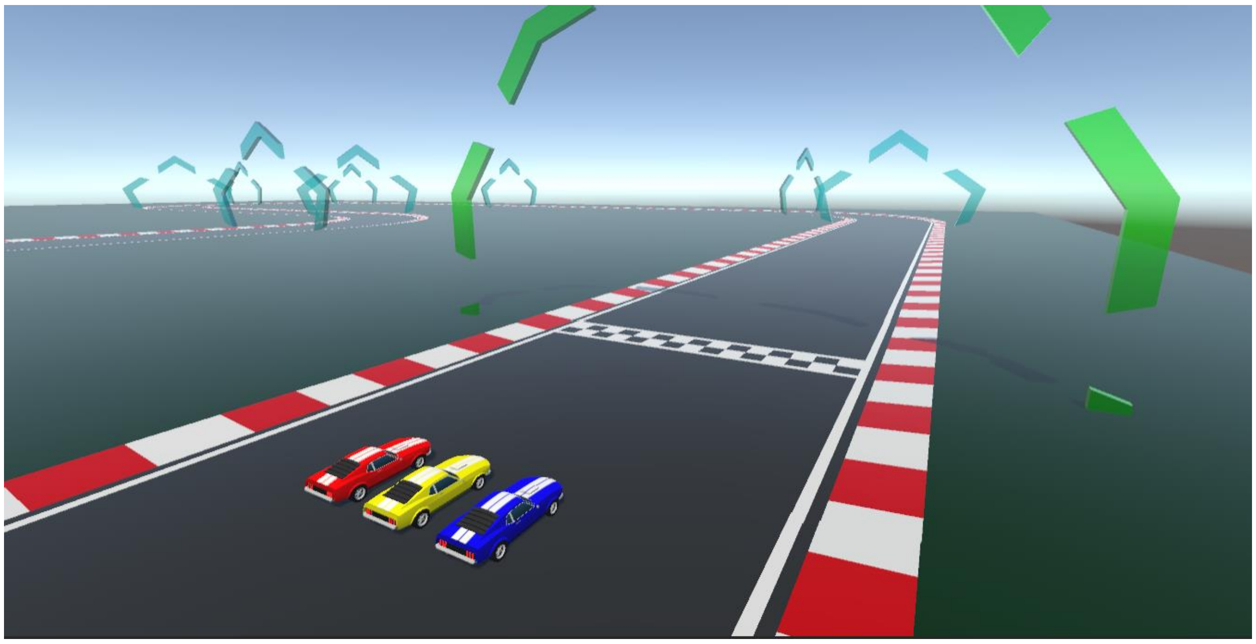
Year 4

Daniel McMorrow

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2022/2023



The Race Lineup

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technical University.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

____

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Michelle Lynch, for her unwavering support, guidance, and invaluable insights throughout the development of this project. Her expertise and dedication were instrumental in bringing this work to fruition, and I am grateful for the constant encouragement she provided.

I would also like to extend my gratitude to Paul Lennon, whose assistance and inspiration during the idea generation phase significantly contributed to shaping the direction of this project.

Special thanks go to Des O'Reilly for his constructive feedback on the project poster, which helped improve its overall presentation and impact.

Finally, I would like to express my appreciation to Lokesh Panti for his assistance in debugging and resolving issues encountered during the project's development. His knowledge and troubleshooting skills were immensely helpful in overcoming obstacles and ensuring the project's success.

Table of Contents

1 Summary

ReinforcementDrive integrates reinforcement learning and artificial intelligence in order to create a thrilling race environment. The main goal is to create a racing game where players can compete against AI-controlled opponents at various levels of difficulty, while simultaneously enhancing my grasp of AI and reinforcement learning concepts.

The project entails creating and implementing key game mechanics, such as AI behaviour, track creation, and car physics. ReinforcementDrive provides a realistic racing experience with two different AI difficulty settings, checkpoints, lap timers, and a ranking system. The AI system, built with pre-trained machine learning models, is a crucial component of the project, ensuring players of different skill levels receive a challenge which suits them. The models are trained using unique hyperparameters for both easy and hard game settings.

The development approach is modular and systematic, ensuring efficient organization. This method involves designing and incorporating individual components separately, starting with basic mechanics like track design and car physics, followed by the integration of AI elements and additional features.

ReinforcementDrive utilizes the robust Unity game engine and the C# programming language to create the game's interactions and logic. The AI system is developed using the ML-Agents plugin for Unity, which employs Proximal Policy Optimization (PPO) and reinforcement learning algorithms, enabling AI opponents to perform competently within the game.

The result is a fully functional racing game featuring versatile AI opponents that cater to a range of difficulty levels. The game includes an advanced checkpoint and lap system, as well as a dynamic ranking system that encourages players to progress.

ReinforcementDrive stands as a successful project that merges reinforcement learning and AI to deliver a compelling and challenging car racing experience. The use of ML-Agents, PPO, and reinforcement learning methods allow AI opponents to perform effectively, ensuring a fun experience for players of all skill levels. The modular and systematic approach to development

culminates in a polished and captivating final product that demonstrates the potential of AI and reinforcement learning in the realm of gaming.

2 Poster

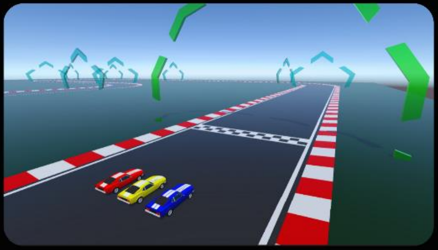
REINFORCEMENTDRIVE

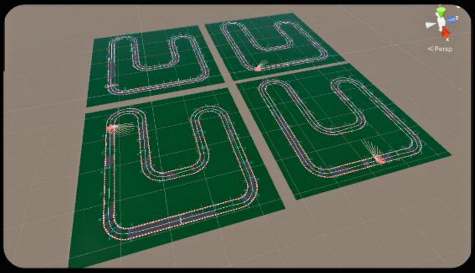
Daniel McMorrow | Software & Electronic Engineering | G00343022

Ollscoil Teicneolaíochta an Atlantaigh
Atlantic Technological University

Introduction

This project is focused on creating a racing game with a unique twist, by integrating Unity Game Engine and ML-Agents plugin. It uses a Reinforcement learning method called Proximal Policy Optimization (PPO) to teach the AI how to drive a car in the game. PPO helps the AI learn from what it knows and discover new techniques. Players can race against the AI, which learns to go faster around the track while avoiding crashes with the player's car. As the AI uses PPO, it keeps getting better at racing, offering players an exciting and challenging experience.



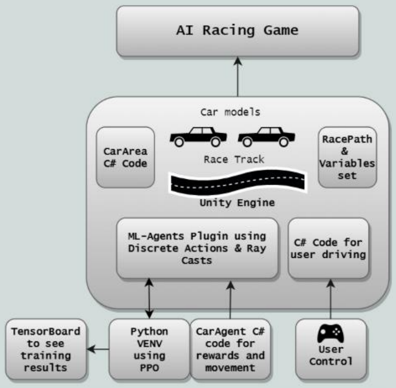


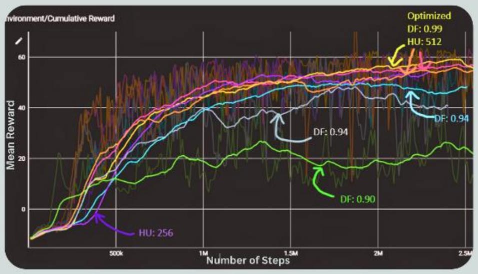
Training, Rewards & Object Detection

- Designed a reward system using a racepath, checkpoints, and invisible walls on the track.
- Checkpoints provided positive rewards, while collisions with invisible walls incurred negative rewards.
- Enhanced the agent's perception with raycasts, allowing it to detect checkpoints, walls, and track boundaries.
- Created multiple instances of the training track to accelerate the agent's learning process.
- Implemented a step timeout to penalize the agent for taking too long to reach the next checkpoint, resetting it to a random checkpoint if necessary.
- These improvements allowed the AI racer to learn from its actions and become a more skilled competitor in the racing game.

Technologies Used

Unity Engine: A powerful game engine & development environment used to create games and applications.
ML-Agents Plugin: Enables machine learning to train intelligent agents within the game.
Proximal Policy Optimization (PPO): A reinforcement learning algorithm used to train the AI agent, balancing exploration and exploitation for efficient learning.
TensorBoard: A visualization tool used to monitor the training process of the AI agent.
Python: Used to configure the training process and define the learning algorithm.
Visual Studio IDE & C#: Used to write necessary C# code for game creation.
Raycasts: Provide the AI agent with information about its environment.





Graph of learning improvements with parameter changes

Results

Initially, the AI agent struggled on the track, often making poor decisions and failing to follow the optimal racing line. However, by adjusting hyperparameters like hidden units and discount factor (gamma), the agent's learning improved. Balancing exploration (trying new actions) and exploitation (leveraging its current knowledge), the agent adapted its behavior and enhanced its racing skills.

These adjustments led to increased mean rewards per step, indicating that the agent was completing different tracks faster and with fewer mistakes. The progress highlights the reinforcement learning approach's effectiveness, with the learning curve eventually leveling out, as seen in the TensorBoard graph output.

Figure 2.1 – Project poster

3 Scripts UML Diagram

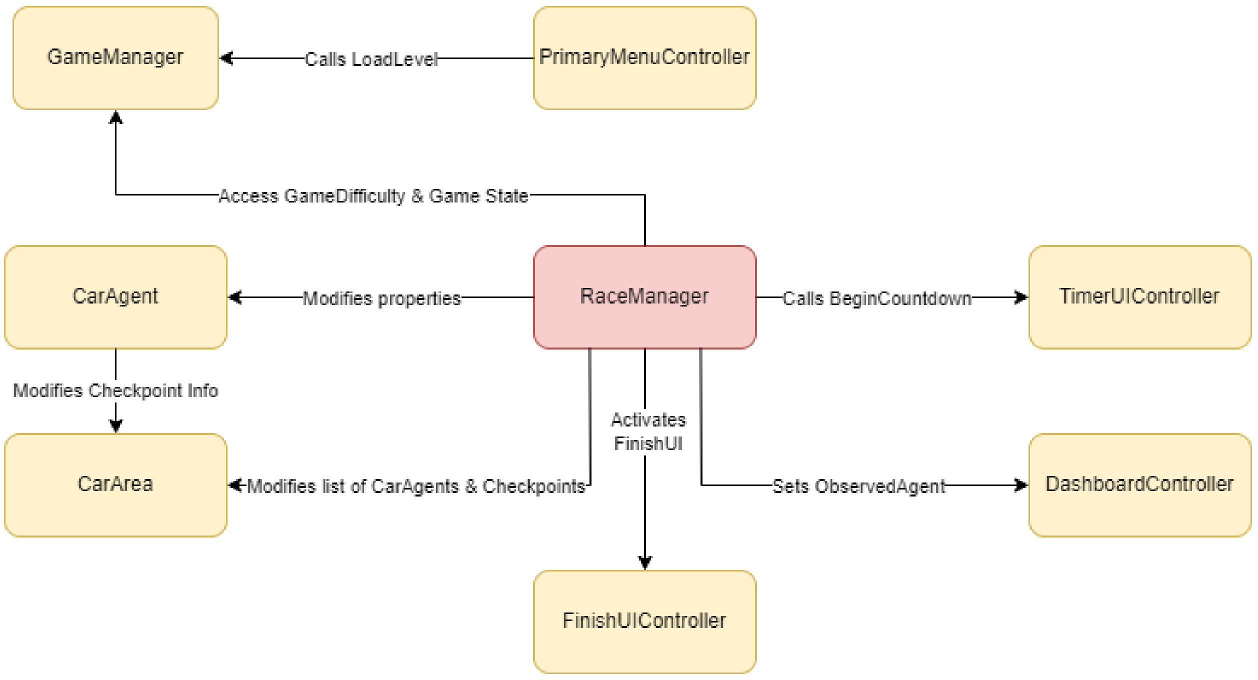


Figure 3.1 – Scripts UML Diagram

4 Introduction

During my student placement at Valeo, a leading company in self-parking and vision systems, I developed a keen interest in the potential of Artificial Intelligence (AI) in the realm of autonomous vehicles. The aforementioned encounter catalysed the inception of the ReinforcementDrive initiative, aimed at comprehending the intricacies of reinforcement learning and developing a game that showcases its prowess in a high-speed racing milieu. The objective of the project is to establish the foundation for prospective practical implementations through the development of a computer-generated representation of an artificial intelligence-powered racing video game.

The scope of ReinforcementDrive is extended by emphasising several crucial domains of motivation. These encompass the creation of diverse artificial intelligence difficulty levels, the integration of AI behaviour, the investigation of reinforcement learning principles, and the formulation and execution of core game mechanics. The project integrates supplementary gaming features such as checkpoints, lap timers, and a ranking system to provide a comprehensive racing experience. The project involves a crucial aspect of exploring potential practical applications of ReinforcementDrive's reinforcement learning methodologies in self-driving systems. Finally, comprehensive documentation and analysis of the project's development lifecycle, utilised technologies, and acquired knowledge will provide invaluable insights into the project's achievements and areas for potential enhancement.

5 Choosing an engine

Before initiating the project, an investigation was conducted on two prominent game development engines, namely Unreal Engine and Unity Engine. After careful consideration of various factors, including the availability of the ML-Agents package and its ease of use, Unity was selected as the engine of choice for the ReinforcementDrive project, despite both engines having their respective pros and cons.

The Unreal Engine developed by Epic Games is renowned for its robust graphical functionalities and superior visual rendering, rendering it a favoured option for AAA game development companies [1]. Programming in the project is accomplished through the utilisation of Blueprints, a proprietary scripting language, and C++. Although Unreal Engine is highly regarded in the gaming sector, it may present a more challenging learning experience for novice game developers [2].

The Unity Engine, developed by Unity Technologies, is renowned for its intuitive interface and ease of use [3]. C# is a commonly acknowledged programming language utilised for scripting purposes. ReinforcementDrive's selection of Unity can be attributed to the ML-Agents package, which provides a robust set of tools for developing AI agents within the Unity environment. The toolkit aligns well with the project's objectives as it provides a robust structure for implementing reinforcement learning.

Furthermore, Unity boasts a substantial community of developers and a comprehensive asset store, offering novice developers a plethora of resources, add-ons, and assistance [5]. This ecosystem facilitates the acquisition of Unity by developers and enables them to efficiently construct their projects, thanks to its manageable learning curve.

The ReinforcementDrive project opted for the ML-Agents package and Unity Engine due to the latter's more accessible learning curve, as opposed to Unreal Engine. The salient features that rendered Unity the optimal choice for developing the AI-based racing game are its robust community backing and resource availability.

6 Project Architecture

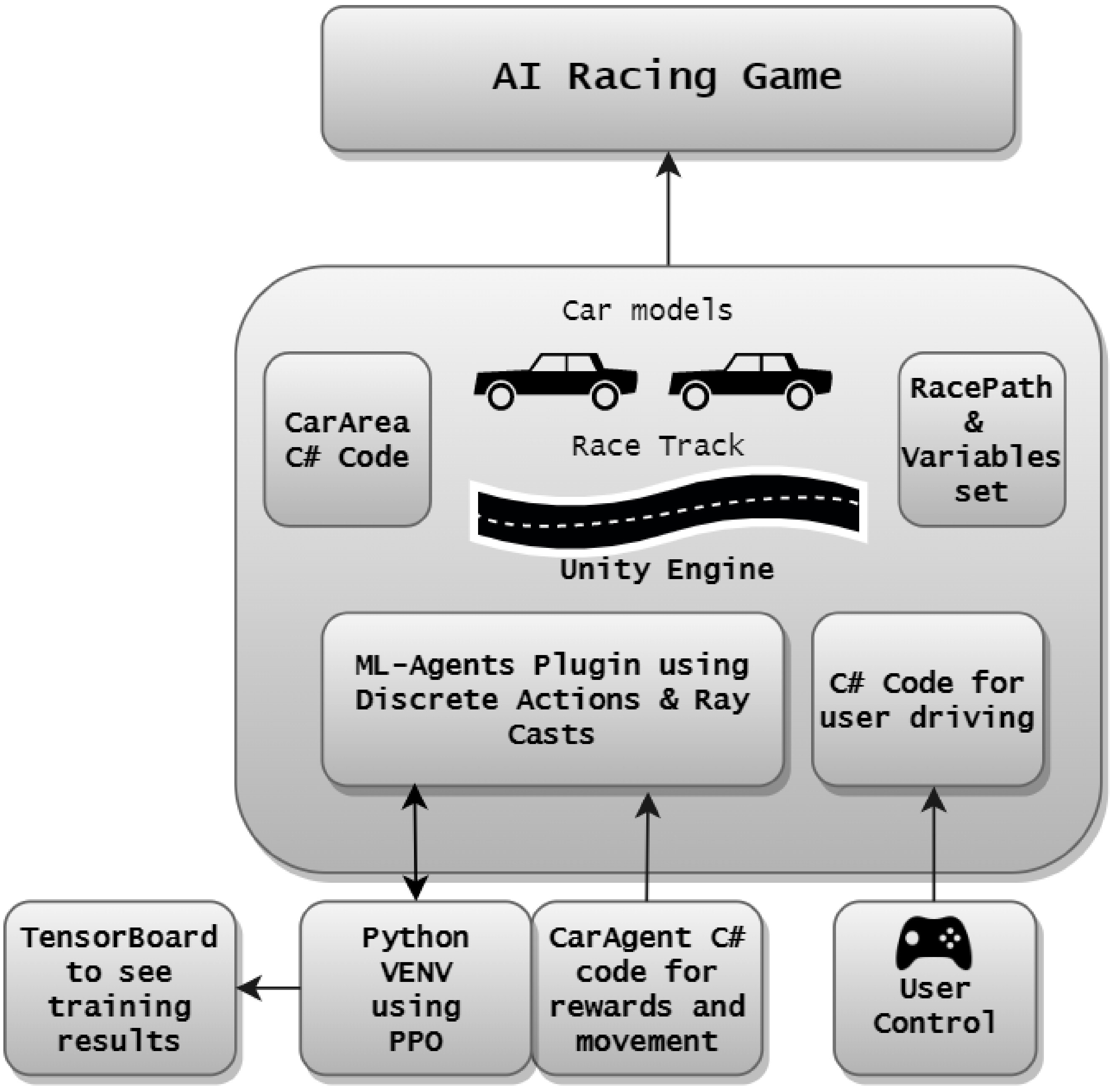


Figure 6-1 Architecture Diagram

7 Project Plan

7.1 Setup Procedure

The initial phase of the project plan entails configuring the essential tools and procuring the requisite software. The essential software prerequisites comprise the Unity game engine, C# programming language, Anaconda for my virtual environment, Jira for project management and the ML-Agents plugin designed for Unity. To ensure a streamlined development process, it is imperative to validate the compatibility and currency of all software.

7.2 Creation of a Virtual Environment

After the installation of requisite tools and software, the subsequent course of action involves the establishment of a virtual environment for the game using Anaconda. This platform provides a basis for executing the training of the AI.

7.3 Fundamental AI Training Environments

Following the establishment of the virtual environment, attention is directed towards the development of rudimentary training environments for AI training. This involved many parts including the tweaking of game mechanics, the creation of race tracks, the manipulation of vehicle physics, and the programming of artificial intelligence actions. The development of a verisimilar and captivating milieu is imperative in augmenting the holistic gaming encounter. This phase is critical in determining the optimal hyperparameters and reward mechanisms for AI adversaries in the gaming environment. Through experimentation with diverse configurations and parameters, the AI can be optimised to provide a suitable degree of difficulty for individuals possessing different levels of proficiency.

7.4 Integration of Artificial Intelligence Technology

Upon identification of the optimal hyperparameters and reward systems, the AI components can be seamlessly integrated into the game environment. The AI system is created using the ML-Agents plugin for Unity, which utilises reinforcement learning algorithms. Achieving optimal integration of artificial intelligence components is imperative to provide a captivating gaming experience.

7.5 Game Logic and Menus

This refers to the programming and design elements that govern the rules and user interface of a video game. Following the integration of AI components, the subsequent phase involves the development of game logic and menus. This entails coding game interactions and developing user interfaces for the checkpoint and lap system, ranking system, and AI difficulty settings. The appropriate integration of game logic and menus is a crucial aspect that enhances the overall quality and usability of the end product.

7.6 Quality Assurance and Assessment

During the software development lifecycle, it is imperative to conduct thorough testing and evaluation to detect potential defects and implement required modifications. Through the implementation of frequent testing and the acquisition of user feedback, the game may undergo refinement to guarantee a pleasurable and engaging experience for players across all proficiencies.

7.7 Completion and Distribution

Upon completion of comprehensive testing and optimisation of all constituent parts, the ultimate iteration of ReinforcementDrive can be constructed and dispatched. This involves the process of game packaging for distribution.

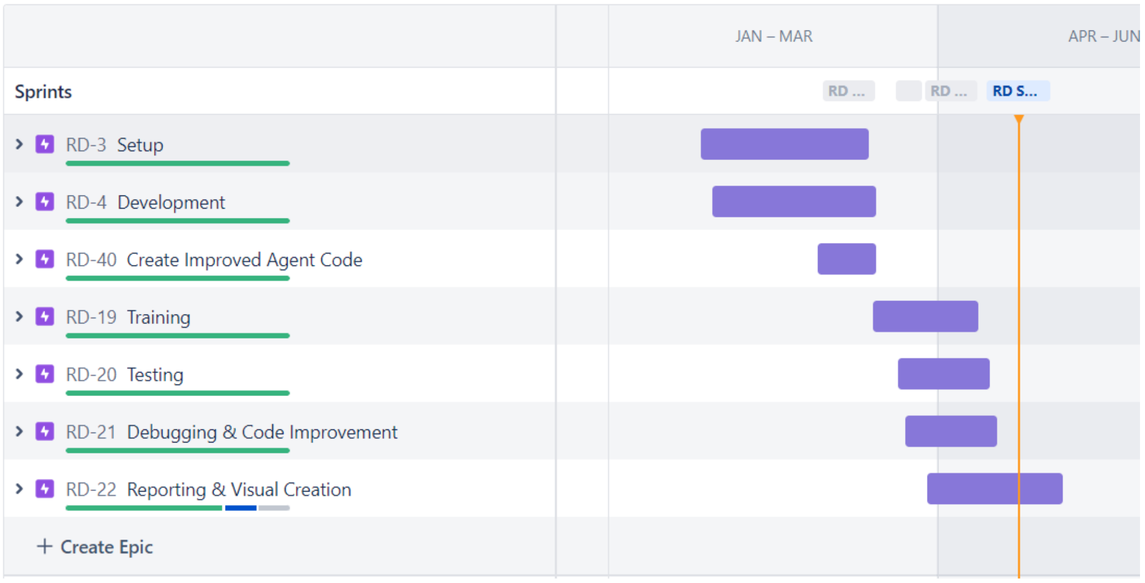


Figure 7.1 - Jira Project Roadmap

8 Learning Unity Basics

Before diving into the development of ReinforcementDrive, it was essential to become familiar with the Unity game engine and its features. To achieve this, time was invested into watching instructional videos and taking courses on game development and artificial intelligence within the Unity framework.

8.1 Watching Instructional Videos

To acquire knowledge about the Unity development process, I engaged in the consumption of diverse video resources about game development within the Unity framework. The aforementioned videos offered significant perspectives on the engine's functionalities, the user interface, and fundamental constituents for game creation. A highly beneficial asset was a compilation of videos endorsed by GameDev Academy, providing a thorough primer on Unity and artificial intelligence within the context of game development[6].

8.2 Unity Courses on AI for Beginners

To further enhance my knowledge of Unity and AI, I enrolled in an online course offered by Unity titled "Artificial Intelligence for Beginners"[7]. The curriculum furnished a comprehensive comprehension of artificial intelligence principles, methodologies, and resources that are exclusive to the Unity platform. The curriculum encompassed concepts including pathfinding, navigation, and machine learning through the utilisation of ML-Agents, all of which were fundamental in the creation of ReinforcementDrive.

Through the consumption of educational videos and enrolment in introductory Unity courses focused on artificial intelligence, I obtained the fundamental understanding required to effectively create ReinforcementDrive. The available resources facilitated a thorough comprehension of the Unity game engine and its AI functionalities, empowering me to proficiently integrate reinforcement learning and artificial intelligence into the game.

9 Understanding Reinforcement Learning, Markov Decision Process

A comprehensive grasp of the fundamentals of reinforcement learning and the Markov Decision Process (MDP) was imperative for the effective integration of artificial intelligence in ReinforcementDrive. These fundamental concepts constitute the basis of the artificial intelligence system implemented in the game.

9.1 Reinforcement Learning Basics

Reinforcement learning is a specialised area of machine learning that encompasses an agent's capacity to attain decision-making skills by means of interactions with an environment [8]. The autonomous agent is programmed to employ reinforcement learning techniques to enhance its performance. It receives feedback in the form of positive or negative reinforcement for each action it executes and adjusts its behaviour accordingly to maximise the overall reward. Reinforcement learning is a crucial component in the training of AI opponents in ReinforcementDrive. This is because it enables the AI to acquire knowledge and adjust to various game scenarios and player proficiencies.

9.2 Markov Decision Process

The Markov Decision Process (MDP) is a mathematical model used in reinforcement learning to describe the decision-making process of an agent in a given environment[9] Markov Decision Processes (MDPs) are comprised of a collection of states, actions, rewards, and transition probabilities that are utilised to define the underlying dynamics of a given environment. The objective of the agent is to determine a policy that can efficiently map states to actions, thereby maximising the expected cumulative reward over a period of time[9]. I've created Figure 9.1 to illustrate this for my project.

9.3 Exploration and Exploitation

Exploration and exploitation are fundamental strategies employed by an agent in reinforcement learning to maintain a balance between information acquisition and decision-making based on prior knowledge [10]. In the context of reinforcement learning, exploration pertains to the process wherein the agent undertakes random actions to uncover novel states and rewards. On the other hand, exploitation pertains to the strategy of selecting the most optimal action based on current knowledge to maximise immediate rewards. Optimising the trade-off between exploration and exploitation is a pivotal aspect of training the AI adversaries in ReinforcementDrive, as it facilitates their ability to acquire knowledge from their interactions and enhance their decision-making skills during gameplay.

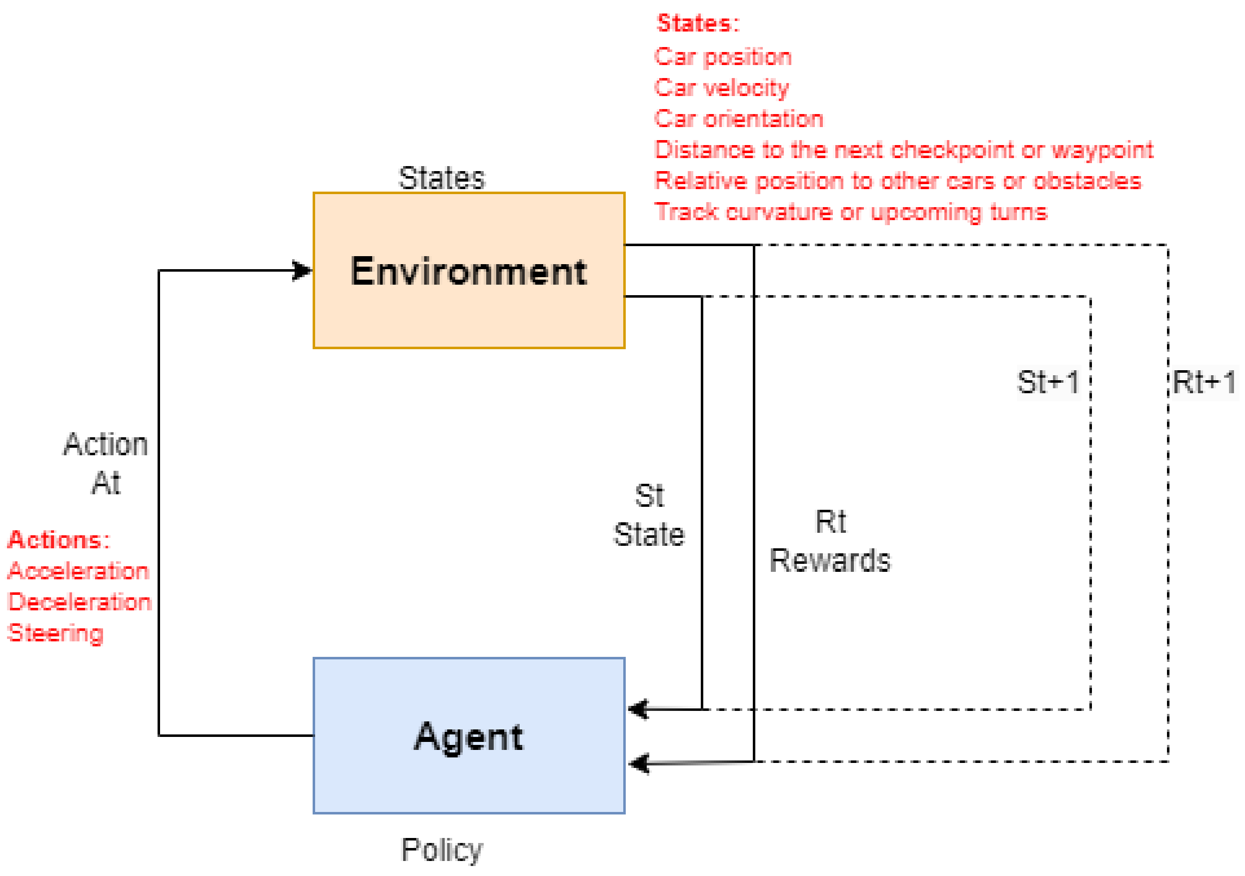


Figure 9.1 – Makarov Decision Process

10 Getting Started with ML-Agents

The AI system in ReinforcementDrive was developed utilising Unity's ML-Agents plugin, a robust solution that facilitates the amalgamation of machine learning and reinforcement learning algorithms into Unity projects. To get started with ML-Agents, the official "Getting Started" guide provided by Unity Technologies[11] was followed.

10.1 Exploring the 3D Balance Ball Example Environment

To become familiar with the ML-Agents framework, the learning process was initiated by conducting an investigation of the 3D Balance Ball example environment that has been included in the ML-Agents repository. The environment consists of a ball that must learn to balance on a platform. The provided instance offered a valuable initiation for comprehending the fundamentals of configuring and educating artificial intelligence agents utilising ML-Agents within the Unity platform.

10.2 Executing a trained model

Upon completion of analysing the 3D Balance Ball illustration, the pre-existing trained model was executed to witness the functionality of the ML-Agents plugin. The model that underwent training exhibited proficient utilisation of reinforcement learning, as the artificial intelligence agent adeptly maintained equilibrium of the ball on the platform. This step allowed me to gain a deeper understanding of how ML-Agents can be used to train AI opponents in ReinforcementDrive.

10.3 Model Training Procedure

Having gained enhanced comprehension of the ML-Agents framework, I moved on to training a new model for the 3D Balance Ball example. By changing the model, I was able to witness the training process of the AI agent as it acquired the skill of ball balancing through trial and error. The practical involvement in instructing artificial intelligence agents via ML-Agents was highly beneficial in implementing and training AI opponents in ReinforcementDrive.

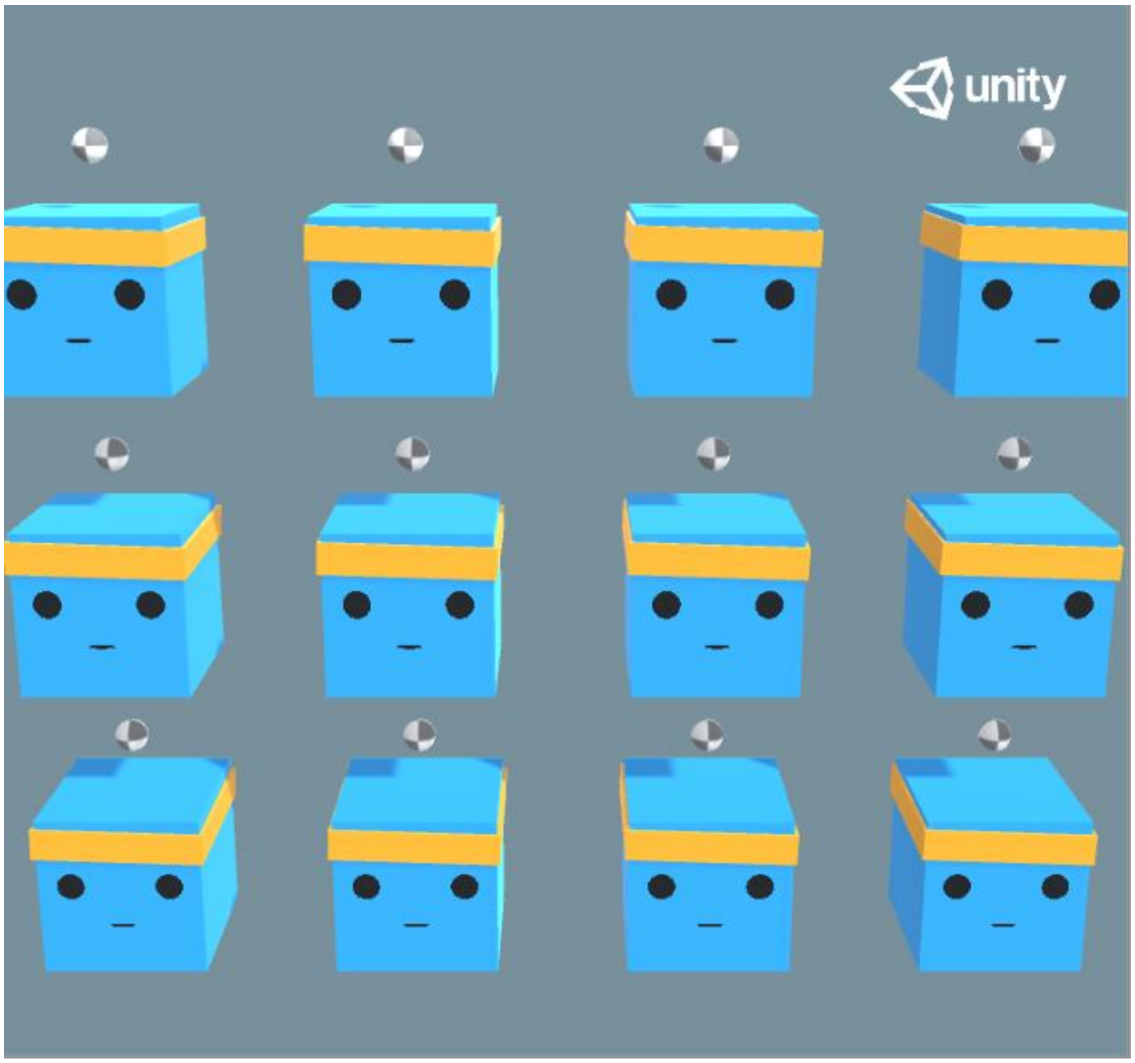


Figure 10.1 – 3D Ball Example[11]

11 Initial Environment Creation

The instantiation of a foundational setting was a pivotal milestone in the evolution of ReinforcementDrive. Through the establishment of a rudimentary environment for the AI agents' interaction, it was possible proficiently train and adjust their conduct prior to advancing to intricate scenarios. The Unity Asset Store was leveraged to procure car and track models for expediting the creation of the initial environment.

11.1 Acquiring Car and Track Models

In order to develop the initial environment, 3D models for the car and the track were needed. found suitable assets on the Unity Asset Store that met my specific requirements. The Prometeo Car Controller asset[12] which has a realistic car model was opted for. The Modular Lowpoly Track & Roads Free asset[13] was chosen for the track, which offers a customizable modular track system that allowed for the creation of a variety of track layouts easily.

11.2 Creating a Simple Track

I chose to create an initial environment featuring a linear track without any bends. The rudimentary setting proved to be optimal for instructing and fine-tuning artificial intelligence agents during their initial phases, as it allowed them to concentrate on fundamental steering actions prior to advancing to more intricate racecourse configurations. By starting with a straightforward course, ensure that the AI agents developed a solid foundation before encountering more complex racing scenarios.

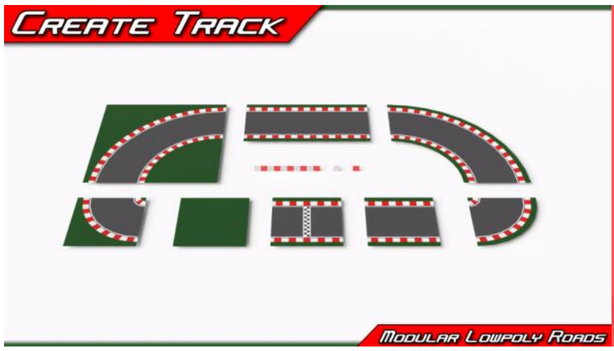


Figure 11.1 – Modular Track[12]

Figure 11.2 – Car Model[13]

12 Creating the User Car Controller

The development of a car controller was an essential element in ReinforcementDrive, enabling the user to exercise effective control over their vehicle. To initiate this process, I by following to a tutorial video[14] to create a basic car controller. Subsequently, I proceeded to advance the code to incorporate additional features and optimizations that enhanced the driving experience.

12.1 Advanced Car Controller

The AdvancedCarController script, shown below, builds upon the initial car controller by adding several features and improvements. The code is organized into various sections, each responsible for a specific aspect of the car's behaviour.

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody))]
public class AdvancedCarController : MonoBehaviour
{
    [Header("Car Properties")]
    [SerializeField] private float motorForce = 1500f;
    [SerializeField] private float brakeForce = 3000f;
    [SerializeField] private float maxSteeringAngle = 30f;
    [SerializeField] private float downForceCoefficient = 10f;
    [SerializeField] private float steerAssistCoefficient = 0.2f;

    [Header("Wheels")]
    [SerializeField] private WheelCollider[] driveWheels;
    [SerializeField] private WheelCollider[] steerWheels;
    [SerializeField] private WheelCollider[] brakeWheels;

    private float horizontalInput;
    private float verticalInput;
    private bool isBraking;
    private Rigidbody carRigidbody;

    private void Start()
    {
        carRigidbody = GetComponent<Rigidbody>();

        // Lower the center of mass
        carRigidbody.centerOfMass = new Vector3(0, -0.7f, 0);

    }

    private void FixedUpdate()
    {
        ProcessInput();
        ApplyMotorTorque();
        ApplySteering();
        ApplyBrakes();
        ApplyDownForce();
        ApplySteerAssist();
    }

    private void ProcessInput()
    {
        horizontalInput = Input.GetAxis("Horizontal");
        verticalInput = Input.GetAxis("Vertical");
        isBraking = Input.GetKey(KeyCode.Space);
    }

    private void ApplyMotorTorque()
    {
        float motorTorque = verticalInput * motorForce;
        foreach (var wheel in driveWheels)
        {
            wheel.motorTorque = motorTorque;
        }
    }

    private void ApplySteering()
    {
        float steerAngle = maxSteeringAngle * horizontalInput;
        foreach (var wheel in steerWheels)
        {
            wheel.steerAngle = steerAngle;
        }
    }

    private void ApplyBrakes()
    {
        float appliedBrakeForce = isBraking ? brakeForce : 0f;
        foreach (var wheel in brakeWheels)
        {
            wheel.brakeTorque = appliedBrakeForce;
        }
    }

    private void ApplyDownForce()
    {
        carRigidbody.AddForce(-transform.up * carRigidbody.velocity.magnitude *
downForceCoefficient);
    }

    private void ApplySteerAssist()
    {
        Vector3 localVelocity =
transform.InverseTransformDirection(carRigidbody.velocity);
        carRigidbody.AddForceAtPosition(transform.right * -localVelocity.x *
steerAssistCoefficient, transform.position);
    }
}
```

Figure 12.1 – AdvancedCarController Script

12.2 Techniques and Algorithms

The AdvancedCarController script employs various techniques and algorithms to implement the basic car behaviours. Here is a comprehensive breakdown of how each function fulfils its intended objective:

12.3 ProcessInput

This function uses Unity's Input class to retrive user input. The GetAxis("Horizontal") and GetAxis("Vertical") methods provide a range of values from -1 to 1, denoting the steering and acceleration inputs. The method Input.GetKey(KeyCode.Space) yields a boolean value denoting whether the brake key (Spacebar) is being pressed.

12.4 ApplyMotorTorque

This function calculates the motor torque by multiplying the input provided by the user with the motorForce variable. The algorithm performs an iterative process on the driveWheels array, assigning the calculated motor torque to each corresponding wheel. This property regulates the magnitude of the force applied to the wheels to enable either an increase or decrease in speed.

12.5 ApplySteering

This function calculates the steering angle by performing a multiplication between the user's horizontal input and the maxSteeringAngle. The program iterates through the steerWheels array and assigns the calculated steering angle to each wheel, thereby regulating the wheels' turning direction.

12.6 ApplyBrakes

This function begins by verifying the isBraking boolean to determine if the user is currently engaging the brakes. If the condition is true, the value of the brakeForce variable is assigned to the brake force; otherwise, the brake force is assigned a value of 0. The algorithm iterates over the brakeWheels array and allocates the calculated brake force to each wheel, executing the braking action when necessary.

12.7 ApplyDownForce

This function computes the downforce through the multiplication of the magnitude of velocity, which represents the car's current speed, and the downForceCoefficient variable. The downforce is subsequently exerted on the rigidbody of the vehicle as a force that operates in the Y direction's negative axis (i.e., downward). This enhances the downforce of the vehicle, thereby improving its grip and steadiness.

12.8 ApplySteerAssist

This feature takes the vehicle's global speed and converts it into local space, calculating the vehicle's speed in its own reference frame. The sideways force on the car is found by multiplying a steering assistance variable with the local sideways speed. The force is applied in the opposite direction of the sideways movement, improving the vehicle's handling and response to the player's steering input.

By using these algorithms, the AdvancedCarController script has smooth and responsive control, enhancing the driving experience for the user.

Although this script was not used in the final implementation of the game it served as a base for the CarAgent script.

13 Comparing Reinforcement Learning Algorithms: PPO, SAC, MA-POCA, and Self-Play in ML-Agents

Reinforcement learning algorithms have been a topic of continuous investigation within the field of artificial intelligence, with a multitude of algorithms developed to tackle specific problems. The objective of this section is to perform a comparative evaluation of Proximal Policy Optimisation (PPO) [15], Soft Actor-Critic (SAC) [16], Multi-Agent Partially Observable Counterfactual Advantage (MA-POCA) [17], and Self-Play to ascertain the most optimal selection for the ReinforcementDrive. Based on the analysis, PPO has been identified as the most appropriate choice.

13.1 Proximal Policy Optimization (PPO) [15]

The Proximal Policy Optimisation (PPO) algorithm is a reinforcement learning technique that has been introduced in reference [15]. Proximal Policy Optimisation (PPO) is a prevalent and extensively utilised algorithm in the field of reinforcement learning, which aims to achieve a trade-off between exploration and exploitation. The Proximal Policy Optimisation (PPO) algorithm amalgamates the benefits of trust region optimisation techniques with the ease of first-order optimisation techniques, leading to consistent learning and expedited convergence. The PPO algorithm has demonstrated efficacy across multiple fields, such as robotics, autonomous vehicles, and game artificial intelligence.

13.2 Soft Actor-Critic (SAC)[16]

This is a reinforcement learning algorithm that aims to maximise the expected cumulative reward of a stochastic policy in continuous control tasks. It achieves this by optimising a soft Q-function and a policy that is regularised by an entropy term. SAC has been shown to outperform other state-of-the-art algorithms on a variety of benchmark tasks. The Stochastic Actor-Critic (SAC) is a reinforcement learning algorithm that operates without a specific policy and model and integrates the benefits of both policy gradients and Q-learning. The Stochastic Actor-Critic (SAC) algorithm is formulated to enhance a stochastic policy by minimising the divergence between the policy and target distribution. This target distribution is obtained from the Q-function. Even though SAC has shown superior performance in continuous control tasks, it needs greater computational resources when compared to PPO and may exhibit reduced efficiency in some scenarios.

13.3 Multi-Agent Partially Observable Counterfactual Advantage (MA-POCA) [17]

MA-POCA is a computational technique that employs multiple agents and reinforcement learning to address challenges posed by partially observable environments. The algorithmic framework is augmented with a centralised critic that approximates counterfactual advantages, thereby expanding the scope of the Counterfactual Multi-Agent (COMA) algorithm. The MA-POCA algorithm is optimised for multi-agent settings and may not be optimal for single-agent assignments such as ReinforcementDrive.

13.4 Self-Play

Self-Play is a technique in reinforcement learning where an autonomous agent improves its performance by participating in a sequence of games against itself and then iteratively updating its policy. The utilisation of Self-Play has been efficaciously implemented in two-player games such as Go, chess, and poker. However, its applicability in single-agent environments, such as a car racing simulator, is limited.

13.5 Choosing an algorithm

ReinforcementDrive has opted for the Proximal Policy Optimisation (PPO) reinforcement learning algorithm on account of its stability, rapid convergence, and extensive adoption across various domains. The PPO algorithm's capacity to maintain a balance between exploration and exploitation makes it a feasible alternative for training artificial intelligence adversaries in the game. Although SAC, MA-POCA, and Self-Play have their own distinct advantages, they are comparatively less suitable for this specific task in contrast to PPO.

While SAC has demonstrated exceptional proficiency in tasks involving continuous control, the decision to employ PPO was predominantly influenced by the project's limited computational resources, which were restricted due to the development being carried out on a laptop. The computational requirements of SAC were deemed to be excessive, thereby making PPO a more feasible alternative. Since the development was executed on a laptop, the utilisation of an algorithm such as SAC could have led to extended training durations and conceivably impeded the advancement of the project. Moreover, the PPO algorithm has exhibited dependable efficacy and expedited convergence in diverse single-agent and multi-agent settings, rendering it a more fitting choice for the racing game milieu, wherein numerous AI agents are competing against the player.

The selection of PPO over SAC was based on its optimal balance of performance, computational resource requirements, and adaptability in managing single-agent and multi-agent environments, rendering it a more suitable choice for ReinforcementDrive.

In addition to this, due to SAC's remarkable efficacy in tasks involving continuous control. This attribute could potentially enable AI agents to attain a high level of proficiency in the domain of racing. Although the attribute is typically favourable, it has the potential to render triumph over AI adversaries unattainable for the majority of players, thereby reducing the game's overall level of amusement.

Through the utilisation of PPO, it is possible to train AI to attain an optimal level of proficiency that maintains a delicate equilibrium between presenting a formidable challenge to players and ensuring that they are still afforded a reasonable opportunity to achieve success. The equilibrium achieved in the game design fosters an immersive and pleasurable gameplay encounter for individuals possessing diverse levels of skill.

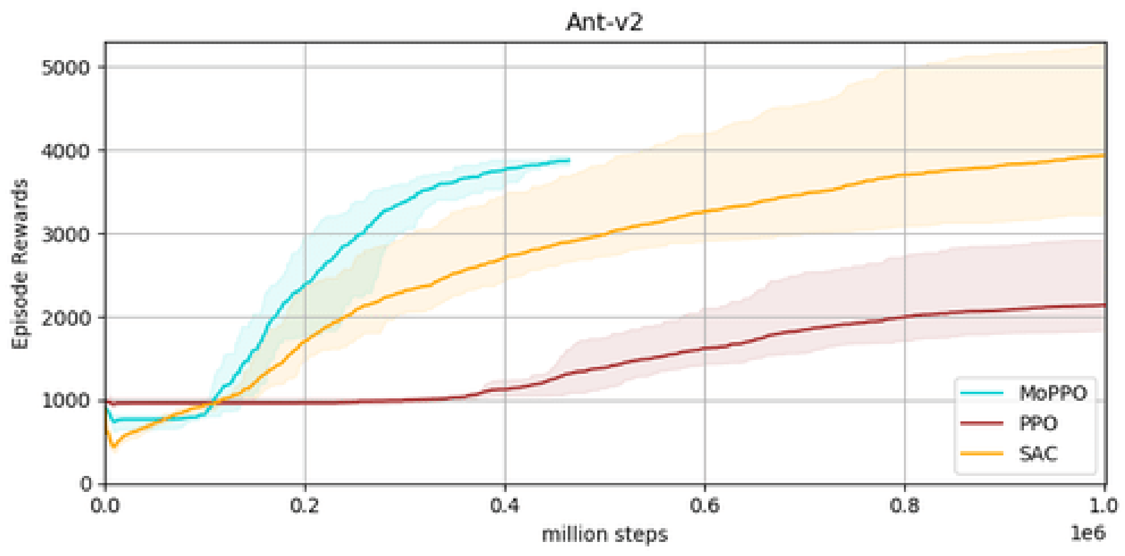


Figure 13.1 – PPO vs SAC vs MoPPO[18]

14 Exploration of Deep Neural Networks in Relation to PPO

Deep Neural Networks (DNNs) have played an important role in the progression AI and reinforcement learning, especially in Proximal Policy Optimisation (PPO). The PPO algorithm leverages deep neural networks to estimate the policy and value functions, enabling the algorithm to learn complex and non-linear relationships between states and actions [15]. This section discusses the importance of DNNs in PPO and how they contribute to the learning process.

14.1 Policy and Value Function Approximation

PPO utilises DNNs as function approximators for the policy (π) and value (V) functions [19]. The policy function, which is commonly referred to as the actor, performs the task of mapping states to a probability distribution over possible actions, while the value function (also called the critic) estimates the expected cumulative reward of each state. DNNs facilitate the representation of intricate and non-linear relationships between states, actions, and rewards, thereby enabling efficient learning in complex environments [15].

14.2 DNN Architecture for PPO

The architecture of DNNs used in PPO typically consists of multiple fully connected layers, followed by non-linear activation functions [19]. Depending on the goal of the network, the input layer receives the state information, while the output layer generates either the value function estimates or the action probabilities for the policy. Depending on the difficulty of the problem and the available computational resources, the number of hidden layers and neurons within each layer can be changed [20]. To find the ideal DNN configuration in practice, researchers frequently use trial-and-error or rely on prior successful architectures.

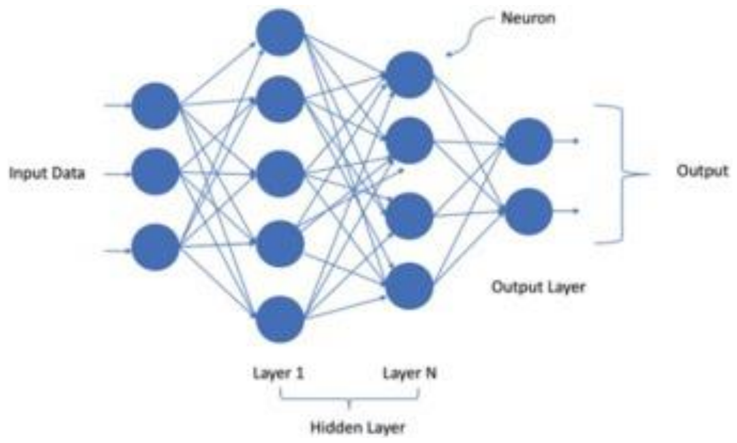


Figure 14.1 - Deep neural network with "N" hidden layers [21]

14.3 DNN Training with PPO

To enhance the policy and value functions, it is necessary to update the network parameters, such as weights and biases, of the deep neural networks trained for PPO. The PPO algorithm incorporates a surrogate loss function that discourages significant policy updates and fosters more consistent learning [22]. The optimisation process employs stochastic gradient ascent for the policy network and stochastic gradient descent for the value network [15] to update the parameters in a manner that maximises the anticipated cumulative reward.

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

Figure 14.2 – Clipped surrogate function [15]

The Clipped Surrogate Objective function is used in the training process to update the policy network.

14.3.1 Trust Region [22]

The trust region method is an optimisation technique that is commonly employed in reinforcement learning algorithms to guarantee the stability of policy updates. The trust region is executed by utilising a surrogate loss function and a clipping mechanism. The PPO algorithm is designed to optimise policy parameters by maximising the expected cumulative reward. It achieves this by constraining the magnitude of the updates to prevent instability during the learning process. In the course of optimisation, the probability ratio of the new and old policies is subject to clipping, which restricts it to a specific range defined by (1 - ε) and (1 + ε). The implementation of a policy constraint mechanism is crucial to maintain the stability of the learning process by confining policy updates within a reliable region. This approach mitigates the risk of significant deviations from the previous policy, which could potentially lead to unstable learning outcomes.

Note that ε was set to 0.2 in hyperparameters meaning that the deviation from one policy to another is up to 20%.

14.3.2 Advantage Function

The advantage function is a function used to estimate the advantage (Value) of taking a particular action in a given state compared to other possible actions. The function At quantifies the relative improvement of a given action a_t with respect to the average action within the corresponding state s_t. The PPO algorithm leverages outcome-driven action selection to facilitate policy updates. This facilitates the AI opponents' training process in ReinforcementDrive to enhance their performance. In ReinforcementDrive the state could be position, velocity, or orientation as inputs. The actions on the other hand could be acceleration, steering or breaking.

14.3.3 The probability ratio

In Figure 14.3 ‘rt’ is commonly referred to as the probability ratio. It represents the ratio between the probability of taking action a_t in state s_t according to the new policy (πθ(a_t|s_t)) and the old policy (πθ_old(a_t|s_t)).

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, \text{ so } r(\theta_{\text{old}}) = 1.$$

Figure 14.3 – Probability ratio [15]

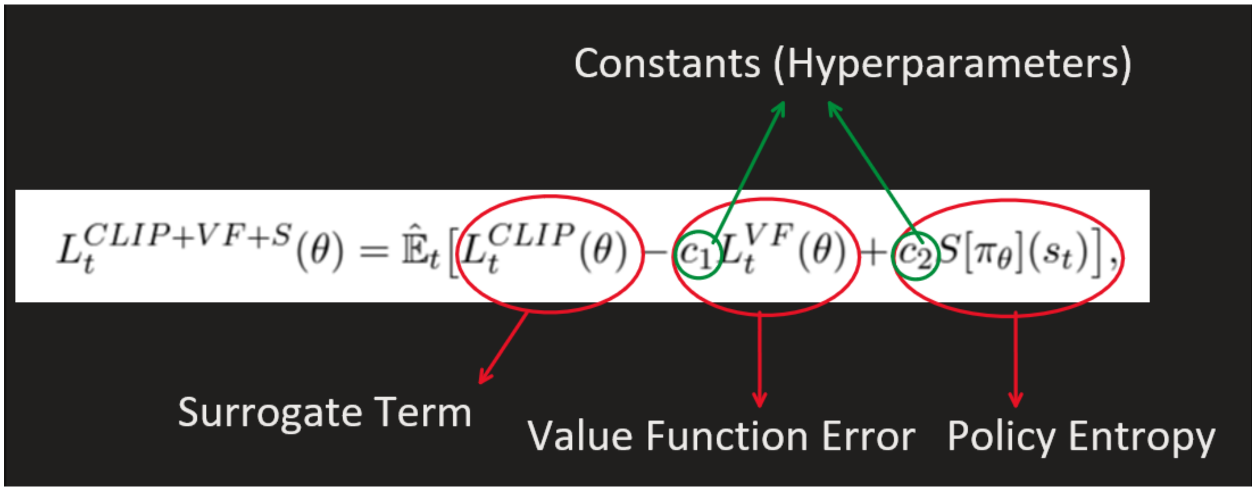

Constants (Hyperparameters)
$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right],$
Surrogate Term
Value Function Error
Policy Entropy

Figure 14.3 – Final loss function

14.3.4 Value Function Error

The value function error term updates the value function by minimizing the difference in the anticipated values and the target values. By doing this, the value function is improved as an estimator of the cumulative total reward. In ReinforcementDrive, the value function is used to predict the expected cumulative reward from the current state of the car (e.g., position, velocity, orientation).

14.3.5 Policy Entropy

The PPO algorithm incentivizes the policy to efficiently explore the action space and avoid early convergence to suboptimal solutions by integrating the entropy term into the objective function. In ReinforcementDrive this helps the AI agents to discover better strategies and adapt to different scenarios. [23]

14.4 Regularization and Generalization

The concepts of regularisation and generalisation hold significant importance in the field of machine learning. Regularisation is a technique in machine learning that involves the incorporation of a penalty term into the loss function. The purpose of this technique is to mitigate the risk of overfitting, which occurs when a model becomes too complex and begins to fit the training data too closely. By adding a penalty term, the model is encouraged to prioritise simpler solutions that generalise better to new data. Generalisation pertains to the capacity of a model to exhibit high performance on new data that is different from the training set. This implies that the model can effectively generalise its learned patterns and relationships to new, unseen data. Deep neural networks have been observed to display exceptional performance in reinforcement learning tasks when subjected to appropriate regularisation techniques [24]. Furthermore, these networks have demonstrated remarkable generalisation abilities when presented with previously unseen states. To improve the generalizability of Deep Neural Networks (DNNs) and mitigate the issue of overfitting, several methods can be utilised, including but not limited to batch normalisation, L2 regularisation, and dropout [25]. The integration of entropy regularisation into the Proximal Policy Optimisation algorithm is a technique that promotes exploration and improves generalisation to new situations by constraining the policy from becoming excessively deterministic [16].

The utilisation of Deep Neural Networks (DNNs) as function approximators for both the policy and value functions is a pivotal aspect of the Proximal Policy Optimisation algorithm. The capacity of these models to depict intricate and non-linear relationships renders them suitable for efficient learning in intricate environments, whereas regularisation techniques facilitate generalisation.

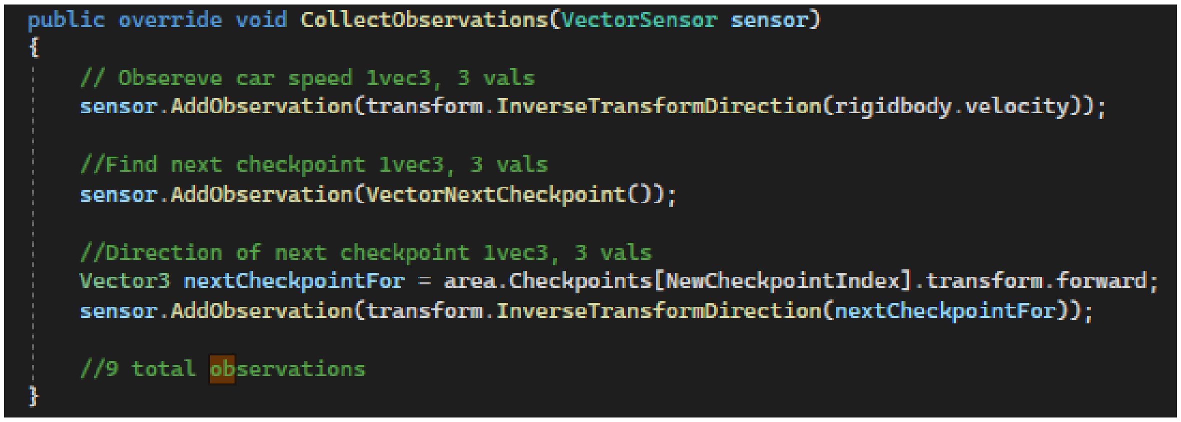
15 Creating an Agent

The following section pertains to the CarAgent script, which assumes the responsibility of managing the car agent within the simulation. Throughout the scripting process, a series of iterations were assessed in order to determine the optimal methodology for conducting training. This involved the execution of numerous iterations through Unity to evaluate the resultant output, while concurrently effecting modifications to the underlying code.

15.1 Observations

The agent collects observations from the environment in order to make informed decisions. Apart from the velocity and checkpoint data, the Agent uses Raycasts to collect environmental information. Raycasting involves projecting a series of invisible rays from a designated origin in a particular direction to identify any obstructions that may be present in their trajectory. The incorporation of these sensors in this project is crucial as it facilitates the agent's ability to detect and respond to nearby objects, thereby optimising its navigation on the track.

15.1.1 Observations in code



```
public override void CollectObservations(VectorSensor sensor)
{
    // Obsereve car speed 1vec3, 3 vals
    sensor.AddObservation(transform.InverseTransformDirection(rigidbody.velocity));

    //Find next checkpoint 1vec3, 3 vals
    sensor.AddObservation(VectorNextCheckpoint());

    //Direction of next checkpoint 1vec3, 3 vals
    Vector3 nextCheckpointFor = area.Checkpoints[NewCheckpointIndex].transform.forward;
    sensor.AddObservation(transform.InverseTransformDirection(nextCheckpointFor));

    //9 total observations
}
```

Figure 15.1 – Code for observations

In the CollectObservations() method, the agent gathers the following information:

Its own speed:

sensor.AddObservation(transform.InverseTransformDirection(rigidbody.velocity))

The vector to the next checkpoint:

sensor.AddObservation(VectorNextCheckpoint())

The direction of the next checkpoint:

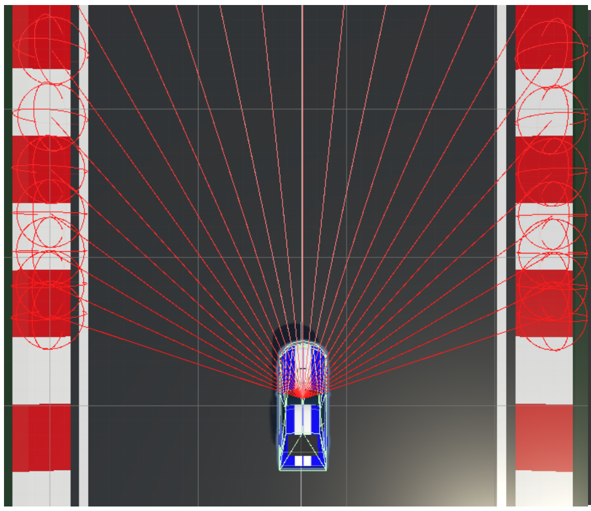
sensor.AddObservation(transform.InverseTransformDirection(nextCheckpointFor))

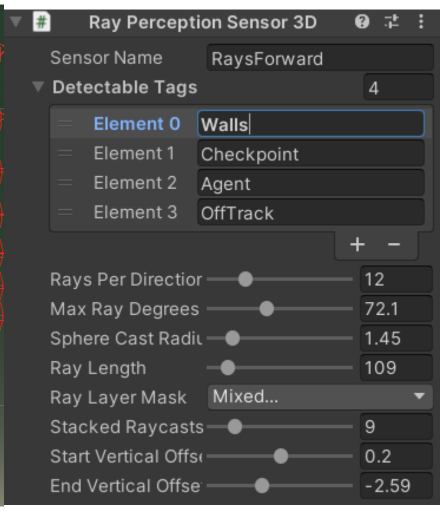
Each of these observations holds a value of 3 for the x, y, and z positions.

15.1.2 Raycasts

The RayPerceptionSensor3D is a vital component in the Agent by enabling ray casting to additional supplementary observations, which enhances the Agent's decision-making capabilities. Ray casting is a computational method that involves the projection of rays emanating from a specific origin in diverse directions. The algorithm detects any instances of intersection between the rays and objects present within the environment. Within the scope of our project, the agent employs the technique of ray casting to perceive its environment, including its distance from walls, other agents, checkpoints, and areas outside the designated track.

The RayPerceptionSensor3D relies on detectable tags to enable object recognition and differentiation within the environment. The RayPerceptionSensor3D can obtain observations by detecting the interactions between rays and designated objects, which are identified by specific tags assigned to them. The data is subsequently utilised by the agent to acquire knowledge on optimising its navigation within the given environment.





Ray Perception Sensor 3D

Sensor Name	RaysForward
Detectable Tags	4
Element 0	Walls
Element 1	Checkpoint
Element 2	Agent
Element 3	OffTrack
Rays Per Directior	12
Max Ray Degrees	72.1
Sphere Cast Radiu	1.45
Ray Length	109
Ray Layer Mask	Mixed...
Stacked Raycasts	9
Start Vertical Offs	0.2
End Vertical Offse	-2.59

Figures 15.2 & 15.3 – Raycast and Sensor settings

In this project, the following detectable tags are used:

Walls: These tags are assigned to the barriers surrounding the track. The agent learns to avoid collisions with walls to minimise negative rewards.

Agent: This tag is assigned to other agents in the environment. The agent learns to avoid crashing into other agents to prevent negative rewards and better navigate the track.

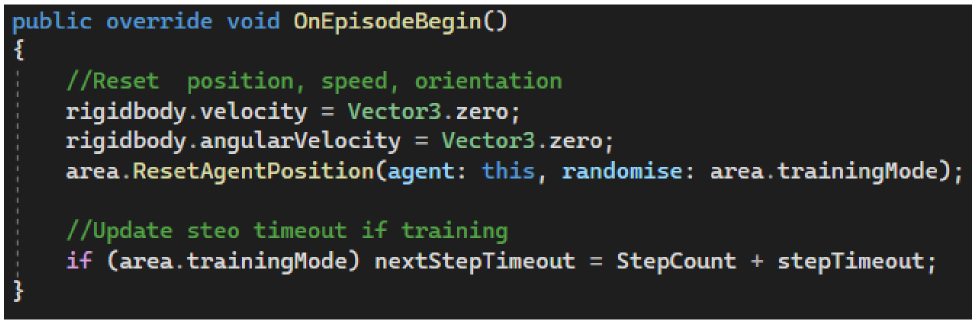
OffTrack: The off-track areas are tagged to help the agent understand the track boundaries and learn to stay on track.

Checkpoints: This tag is assigned to checkpoints so that the agent can see the correct path.

Through the utilisation of the RayPerceptionSensor3D component and detectable tags, our agent gains an enhanced comprehension of its environment. As a result, the system attains proficiency in mitigating negative reinforcement and selecting the most efficient route through iterative experimentation.

15.2 Training setup

In the course of the CarAgent script development, it was discovered that the incorporation of spawn location randomization and a step timeout led to a signified enhancement in the training process. The utilisation of randomised spawn locations is crucial in facilitating effective driving by the agent across diverse scenarios, as opposed to mere memorization of a particular track configuration. Through consistent exposure to various scenarios, the agent's adaptability, and ability to navigate diverse tracks post-training are enhanced.



```
public override void OnEpisodeBegin()
{
    //Reset  position, speed, orientation
    rigidbody.velocity = Vector3.zero;
    rigidbody.angularVelocity = Vector3.zero;
    area.ResetAgentPosition(agent: this, randomise: area.trainingMode);

    //Update steo timeout if training
    if (area.trainingMode) nextStepTimeout = StepCount + stepTimeout;
}
```

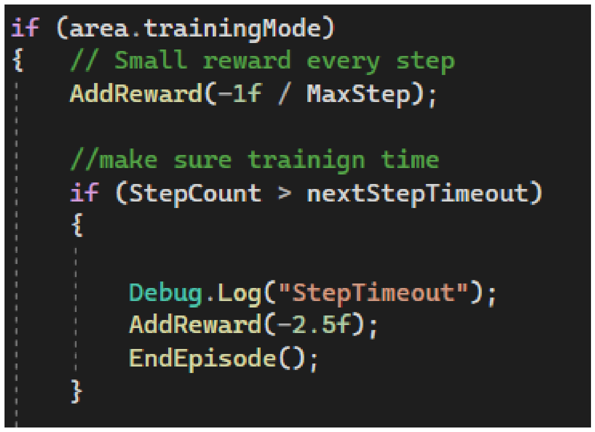
Figure 15.4 – OnEpisodeBegin

The randomization of the agent's spawn location is accomplished through the OnEpisodeBegin() method, which resets the agent's position, velocity, and orientation before the start of each new episode.

15.3 Rewards

In reinforcement learning, rewards are possibly the most important component for facilitating the agent's comprehension of the outcomes of its actions. The CarAgent script implements a reward system that assigns rewards to the agent based on its actions. Let us delve into a comprehensive analysis of each individual reward.

15.3.1 Negative reward for each step taken & timing out



```
if (area.trainingMode)
{   // Small reward every step
    AddReward(-1f / MaxStep);

    //make sure trainign time
    if (StepCount > nextStepTimeout)
    {

        Debug.Log("StepTimeout");
        AddReward(-2.5f);
        EndEpisode();
    }
```

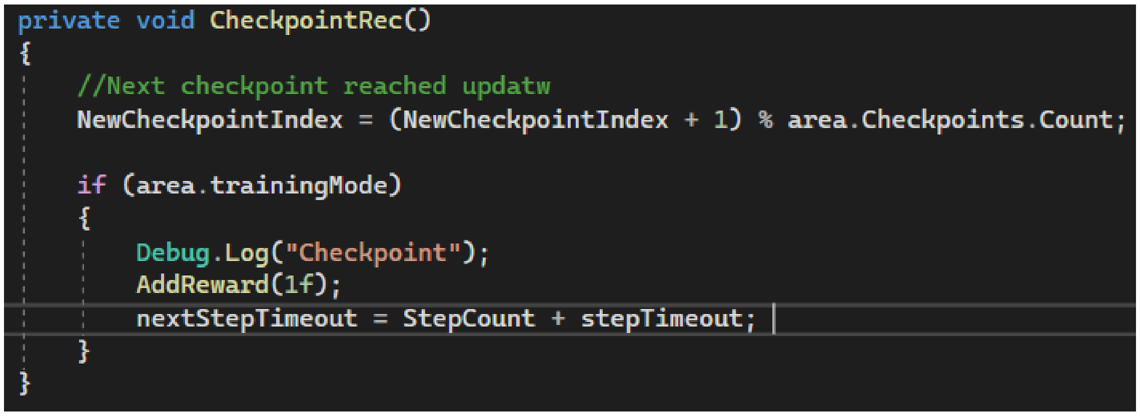
Figure 15.5 – OnActionsRecieved Rewards

The first reward shown encourages the agent to complete the task quickly. The agent receives -1f / MaxStep reward for each step.

The second reward pertains to StepTimeout, -2.5f, which is given in the event that the Agent fails to reach its objective within a specified number of steps, thereby necessitating a reset of the episode. This encourages exploration instead of remaining static.

15.3.2 Positive reward for reaching checkpoints

The agent is programmed to receive a reward of 1f when it reaches the next checkpoint in the correct order successfully. This creates a system of incentives for the agent to adhere to the correct path and successfully navigate through subsequent checkpoints. The code for this reward is in the CheckpointRec() method which is linked to the CarArea script:

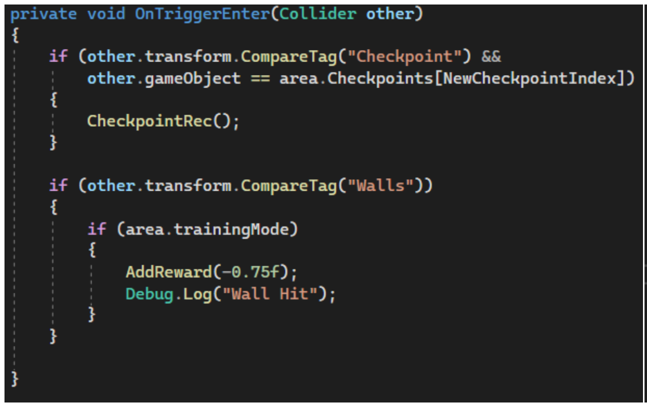


```
private void CheckpointRec()
{
    //Next checkpoint reached updatw
    NewCheckpointIndex = (NewCheckpointIndex + 1) % area.Checkpoints.Count;

    if (area.trainingMode)
    {
        Debug.Log("Checkpoint");
        AddReward(1f);
        nextStepTimeout = StepCount + stepTimeout;
    }
}
```

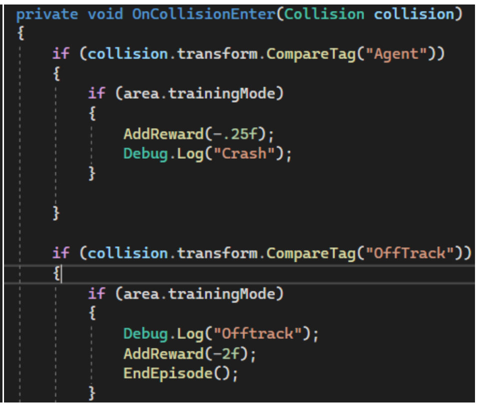
Figure 15.6 – CheckpointRec

15.3.3 Negative Rewards for Collisions

The agent incurs negative rewards when it collides with walls, other agents, or going off the track. This serves as a deterrent for the agent to commit navigation errors. The implementation for the rewards is present in the OnTriggerEnter() and OnCollisionEnter() functions:



```
private void OnTriggerEnter(Collider other)
{
    if (other.transform.CompareTag("Checkpoint") &&
        other.gameObject == area.Checkpoints[NewCheckpointIndex])
    {
        CheckpointRec();
    }

    if (other.transform.CompareTag("Walls"))
    {
        if (area.trainingMode)
        {
            AddReward(-0.75f);
            Debug.Log("Wall Hit");
        }
    }

}
```



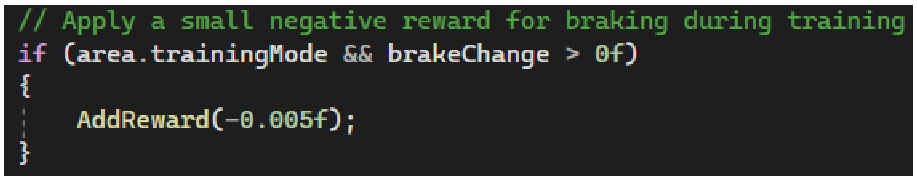
```
private void OnCollisionEnter(Collision collision)
{
    if (collision.transform.CompareTag("Agent"))
    {
        if (area.trainingMode)
        {
            AddReward(-.25f);
            Debug.Log("Crash");
        }

    }

    if (collision.transform.CompareTag("OffTrack"))
    {
        if (area.trainingMode)
        {
            Debug.Log("Offtrack");
            AddReward(-2f);
            EndEpisode();
        }
```

Figure 15.7 & 15.8- OnTriggerEnter and OnCollisionEnter

The implementation of walls was important as it allowed the Agent to see the racetrack boundaries and avoid them. It being a trigger meant it was able to still pass through the walls.

15.3.4 Small negative rewards for braking and not moving during training

The agent receives a small negative reward of -0.005f for braking or not moving during training. This discourages excessive braking, promoting smoother driving. The code for this reward is in the ApplyBrakes() and ApplyMotorTorque() methods.



```
// Apply a small negative reward for braking during training
if (area.trainingMode && brakeChange > 0f)
{
    AddReward(-0.005f);
}
```

Figure 15.9 – Negative reward in applying brakes

15.4 Discrete Actions & Heuristics

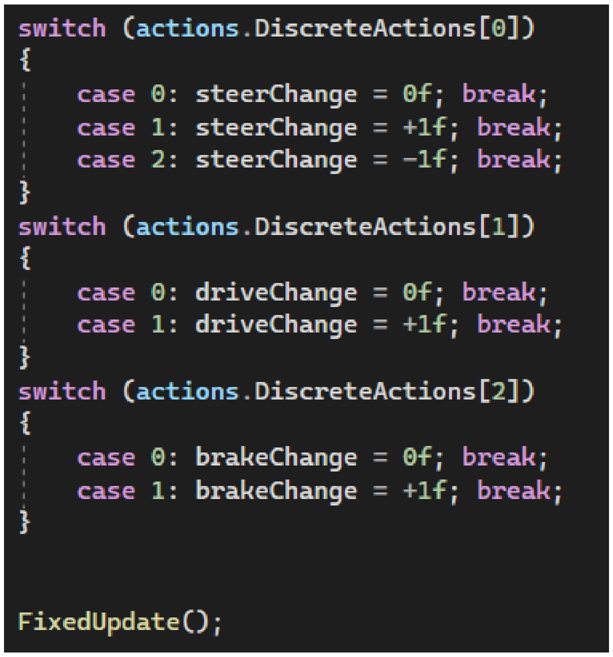
Discrete actions refer to a specific type of action space that an agent can utilise to engage with the environment. Within the Unity ML-Agents framework, discrete actions are encoded as integral values and possess a limited set of potential outcomes. This is in contrast to continuous actions, which represent a continuous range of values. Discrete actions are applicable to scenarios where the agent has a finite number of actions at its disposal, such as steering left, right, or abstaining from turning altogether.

The CarAgent script comprises three discrete actions, each denoting a particular control input.

Steering: The agent can either steer left, steer right, or remain straight. This is denoted by 0 (no steering), 1 (steer right), and 2 (steer left).

Throttle: The agent can either apply throttle or not apply throttle. This is denoted by 0 (no throttle) and 1 (apply throttle).

Brake: The agent can either choose to apply brakes or not. This is denoted by 0 (no brakes) and 1 (apply brakes).



```
switch (actions.DiscreteActions[0])
{
    case 0: steerChange = 0f; break;
    case 1: steerChange = +1f; break;
    case 2: steerChange = -1f; break;
}
switch (actions.DiscreteActions[1])
{
    case 0: driveChange = 0f; break;
    case 1: driveChange = +1f; break;
}
switch (actions.DiscreteActions[2])
{
    case 0: brakeChange = 0f; break;
    case 1: brakeChange = +1f; break;
}


FixedUpdate();
```

Figure 15.10 – Discrete Actions code

The discrete actions undergo processing in the OnActionReceived function of the CarAgent script, wherein the agent executes the corresponding action depending on the received integer value.

Heuristics serve as a means of supplying manual control inputs to the agent for the purposes of testing or demonstration. The Heuristic method is utilised in Unity ML-Agents to implement heuristics. This approach enables software engineers to establish a correspondence between keyboard inputs or alternative control schemes and the action space of the agent. The CarAgent script utilises the Heuristic technique to translate user input received from the keyboard into discrete actions of the agent. This facilitates direct control of the car by the user through the agent script.

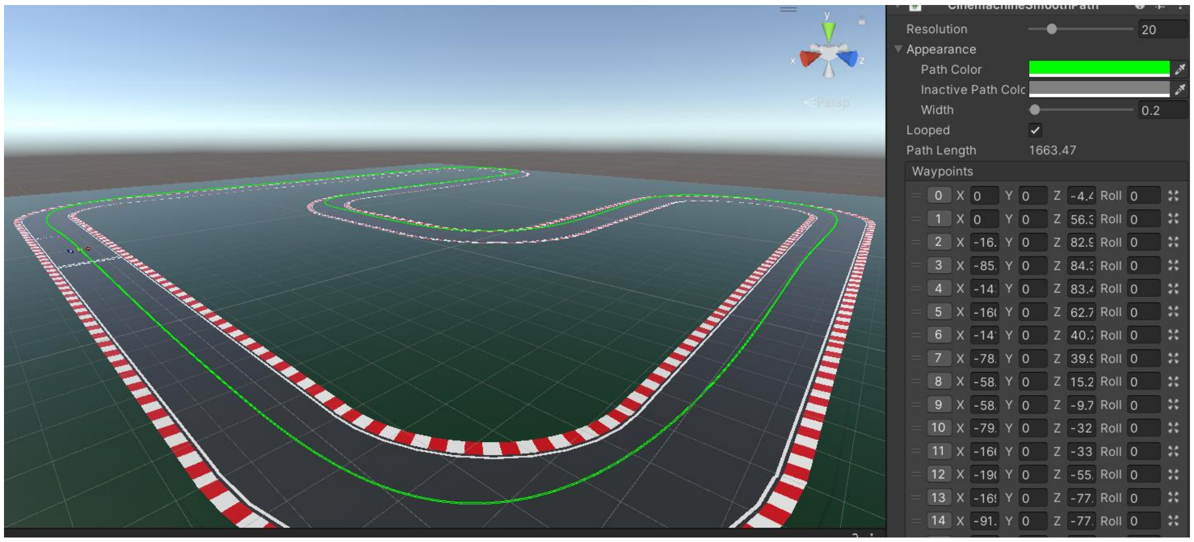
After the careful combination of observations, actions, rewards, and heuristics, the CarAgent script analysis shows how a reinforcement learning agent that can successfully navigate challenging environments can be built. The development process, which was based on trial and error, was essential in optimising the rewards system and locating the best rewards.

16 Checkpoint Setup

The section of the project is centred on the setup of checkpoints along the track by utilising Cinemachine race paths and waypoints. The evaluation of an agent's progress and determination of its position on the track heavily relies on checkpoints.

16.1 Cinemachine Race Path

Cinemachine is a robust camera package designed for Unity, which enables seamless and dynamic camera management. It has the capability to generate complex paths and waypoints for the agents to follow within the racing environment. The race path is defined using a CinemachineSmoothPath component in this project. This module facilitates the creation of a seamless and continuous path by utilising a series of waypoints, that establish the foundation for the checkpoints.



Resolution	20
Appearance	
Path Color	🟩
Inactive Path Colc	▨
Width	0.2
Looped	✓
Path Length	1663.47

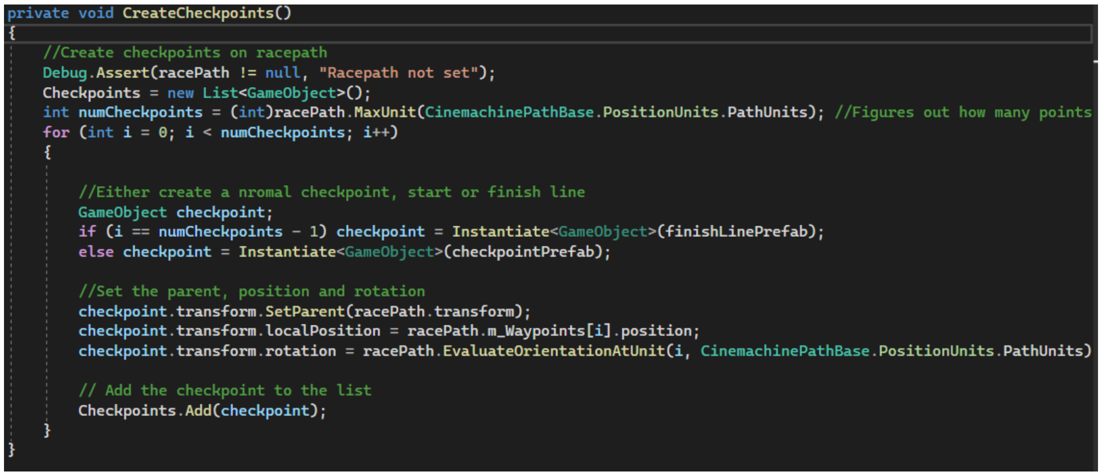
Waypoints

0	X	0	Y	0	Z	-4.4	Roll	0
1	X	0	Y	0	Z	56.3	Roll	0
2	X	-16.	Y	0	Z	82.9	Roll	0
3	X	-85.	Y	0	Z	84.3	Roll	0
4	X	-14	Y	0	Z	83.4	Roll	0
5	X	-16	Y	0	Z	62.7	Roll	0
6	X	-14	Y	0	Z	40.7	Roll	0
7	X	-78.	Y	0	Z	39.9	Roll	0
8	X	-58.	Y	0	Z	15.2	Roll	0
9	X	-58.	Y	0	Z	-9.7	Roll	0
10	X	-79.	Y	0	Z	-32	Roll	0
11	X	-16	Y	0	Z	-33	Roll	0
12	X	-19	Y	0	Z	-55.	Roll	0
13	X	-16	Y	0	Z	-77.	Roll	0
14	X	-91.	Y	0	Z	-77.	Roll	0

Figure 16.1 – RacePath in green

16.2 Checkpoint Creation

The CarArea script is accountable for generating the checkpoints in accordance with the CinemachineSmoothPath. Within the CreateCheckpoints() method, the code utilises a loop to iterate through the waypoints that have been defined by the CinemachineSmoothPath. Depending on the index of each waypoint, the script will instantiate either a checkpoint or the finish line.



```
private void CreateCheckpoints()
{
    //Create checkpoints on racepath
    Debug.Assert(racePath != null, "Racepath not set");
    Checkpoints = new List<GameObject>();
    int numCheckpoints = (int)racePath.MaxUnit(CinemachinePathBase.PositionUnits.PathUnits); //Figures out how many points
    for (int i = 0; i < numCheckpoints; i++)
    {

        //Either create a nromal checkpoint, start or finish line
        GameObject checkpoint;
        if (i == numCheckpoints - 1) checkpoint = Instantiate<GameObject>(finishLinePrefab);
        else checkpoint = Instantiate<GameObject>(checkpointPrefab);

        //Set the parent, position and rotation
        checkpoint.transform.SetParent(racePath.transform);
        checkpoint.transform.localPosition = racePath.m_Waypoints[i].position;
        checkpoint.transform.rotation = racePath.EvaluateOrientationAtUnit(i, CinemachinePathBase.PositionUnits.PathUnits)

        // Add the checkpoint to the list
        Checkpoints.Add(checkpoint);
    }
}
```

Figure 16.2 – CreateCheckpoints

To summarise, the establishment of checkpoints is a critical component in directing the agent's navigation within the racing environment. The Cinemachine package within Unity is employed to generate a seamless racing trajectory with designated waypoints, subsequently partitioned into checkpoints via the CarArea script.

17 Training the Agents

Before diving into the specifics of the training process, it's important to understand the overarching goal: to create a well-trained, adaptable agent capable of navigating diverse tracks with ease. This was achieved through a combination of carefully designed environments and fine-tuning of hyperparameters.

17.1 Environment setup

After the development of CarAgent and CarArea scripts, the agent could begin training on more intricate tracks that encompassed walls and turns in place of the straight track. The establishment of walls was imperative in facilitating the agent's understanding of its spatial constraints and enhancing its ability to manoeuvre within them proficiently. To enhance the efficiency of the training process, the track was replicated multiple times within the Unity environment, enabling the concurrent learning of multiple agents.

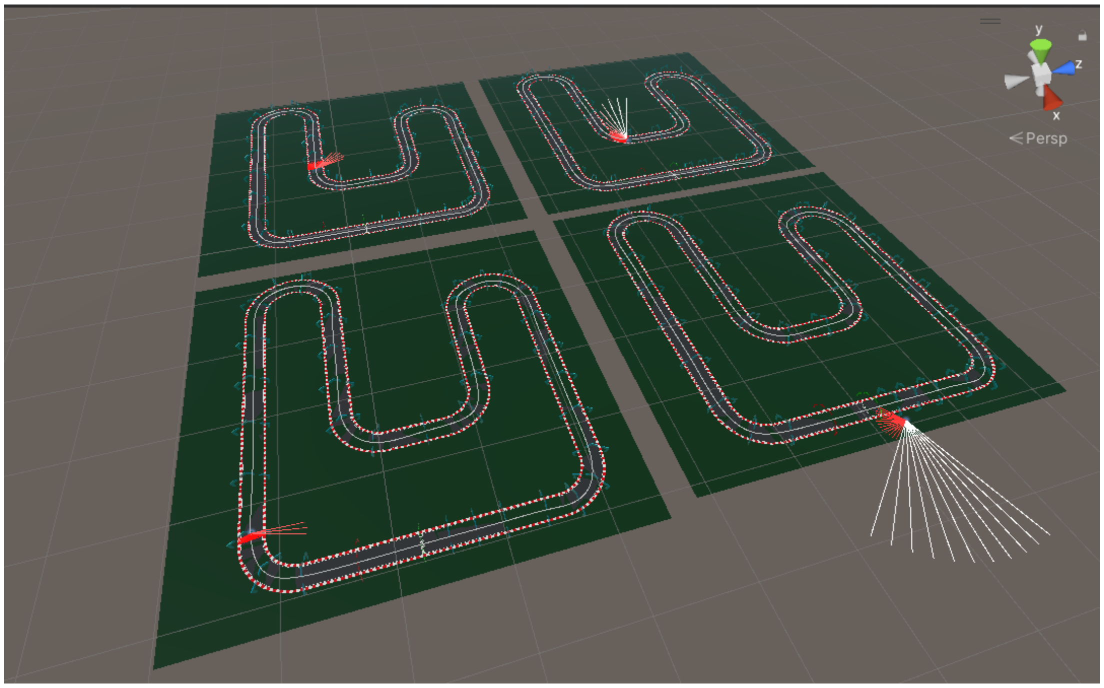
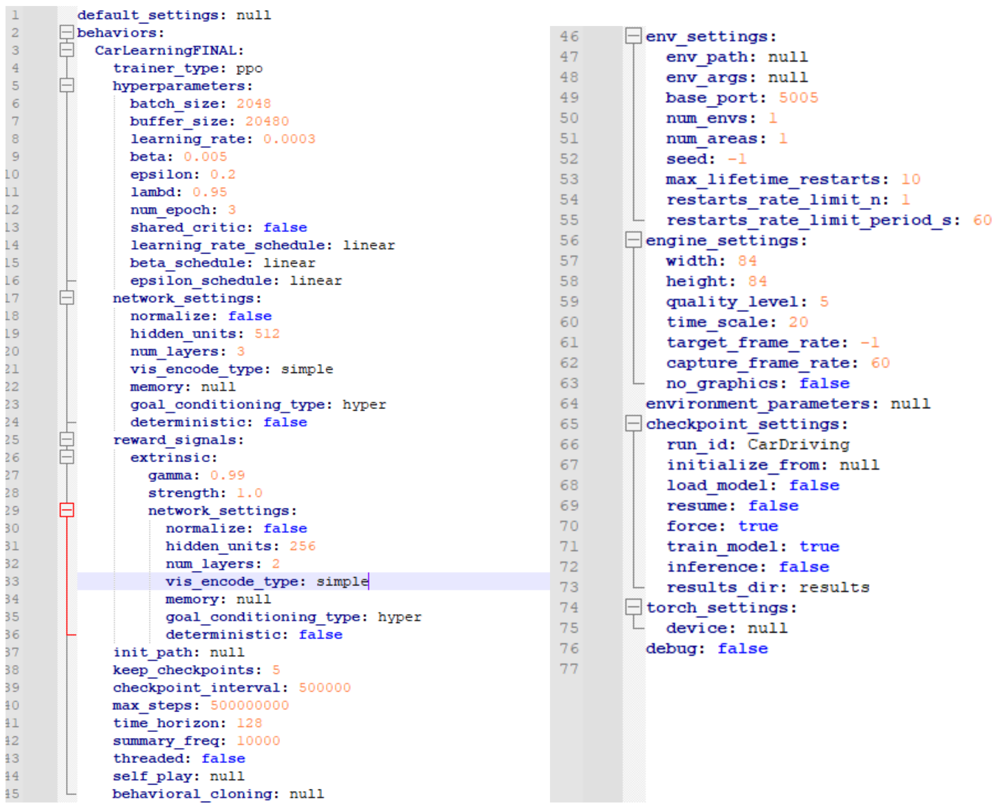


Figure 17.1 – Training Setup

17.2 Finding the Optimal Hyperparameters

In order to find the best configuration for training the agents, several hyperparameters were tweaked to optimize the learning process.



```yaml
default_settings: null
behaviors:
  CarLearningFINAL:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      shared_critic: false
      learning_rate_schedule: linear
      beta_schedule: linear
      epsilon_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 512
      num_layers: 3
      vis_encode_type: simple
      memory: null
      goal_conditioning_type: hyper
      deterministic: false
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
        network_settings:
          normalize: false
          hidden_units: 256
          num_layers: 2
          vis_encode_type: simple
          memory: null
          goal_conditioning_type: hyper
          deterministic: false
    init_path: null
    keep_checkpoints: 5
    checkpoint_interval: 500000
    max_steps: 500000000
    time_horizon: 128
    summary_freq: 10000
    threaded: false
    self_play: null
    behavioral_cloning: null
env_settings:
  env_path: null
  env_args: null
  base_port: 5005
  num_envs: 1
  num_areas: 1
  seed: -1
  max_lifetime_restarts: 10
  restarts_rate_limit_n: 1
  restarts_rate_limit_period_s: 60
engine_settings:
  width: 84
  height: 84
  quality_level: 5
  time_scale: 20
  target_frame_rate: -1
  capture_frame_rate: 60
  no_graphics: false
environment_parameters: null
checkpoint_settings:
  run_id: CarDriving
  initialize_from: null
  load_model: false
  resume: false
  force: true
  train_model: true
  inference: false
  results_dir: results
torch_settings:
  device: null
debug: false
```

Figure 17.2 – Optimal Configuration

17.2.1 Batch Size

In the context of the training process, the batch size denotes the quantity of samples utilised for every update. The enhanced learning efficiency of the agents was achieved by augmenting the batch size, which facilitated the utilisation of a greater amount of data for each update. A batch size of 2048 was selected in this instance.

17.2.2 Buffer Size

The buffer size refers to the upper limit of experiences that can be stored within the replay buffer. Increasing the buffer size can potentially enhance the learning process by enabling the agent to access a wider range of experiences. The project's buffer size was initialised to 20480.

17.2.3 Discount Factor (Gamma)

Gamma is the discount factor that determines the importance of future rewards in the learning process. A higher gamma value gives more weight to future rewards, while a lower gamma value places more emphasis on immediate rewards. In this project, a gamma value of 0.99 was chosen after testing.

The graph in Figure 17.3 illustrates the impact of the discount factor. The green line depicts a discount factor of 0.90, indicating the prioritisation of immediate rewards by the agent. Conversely, the purple line exhibits a discount factor of 0.99. The graph demonstrates that the lower discount factor exhibits a faster attainment of initial rewards compared to the higher discount factor, albeit peaking at a mean reward of approximately 30. The utilisation of a discount factor of 0.99 results in a significant increase in mean rewards per episode, nearly doubling the value achieved without it.

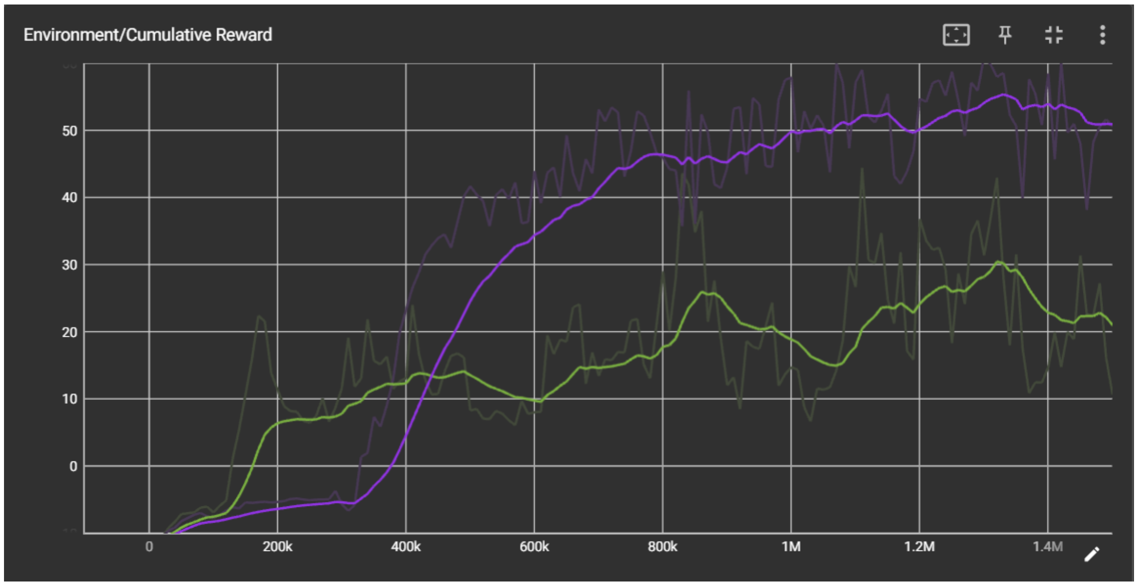


Figure 17.3 – Discount factor alterations

17.2.4 Learning rate

Modifying the learning rate parameter impacts the speed at which the agents acquire knowledge from their past interactions. The learning rate is a hyperparameter that specifies the step size for each update in the optimization process. Increasing the value of the learning rate can expedite the convergence process, whereas decreasing the learning rate can enhance the stability of the system with the trade-off of being slower. The project's optimal learning rate was determined to be 0.0003.

"Slow and steady wins the race"

17.2.5 Hidden Units

The quantity of neurons present in every hidden layer of the neural network utilised by the agent is referred to as hidden units. Increasing the quantity of hidden units can enhance the network's capacity, enabling it to acquire intricate patterns and correlations within the data. However, increasing the quantity of hidden units can also escalate the network's vulnerability to overfitting which may necessitate additional training duration.

17.2.5.1 Overfitting

Overfitting occurs in machine learning when a model achieves a high level of accuracy using the training data but is not able to generalise its performance for new data. In ReinforcementDrive, overfitting may present itself as the agent exhibiting optimal driving behaviour on a particular track utilised during the training phase while encountering difficulties in achieving comparable performance on new tracks. I have sketched Figure 17.4 to show this.

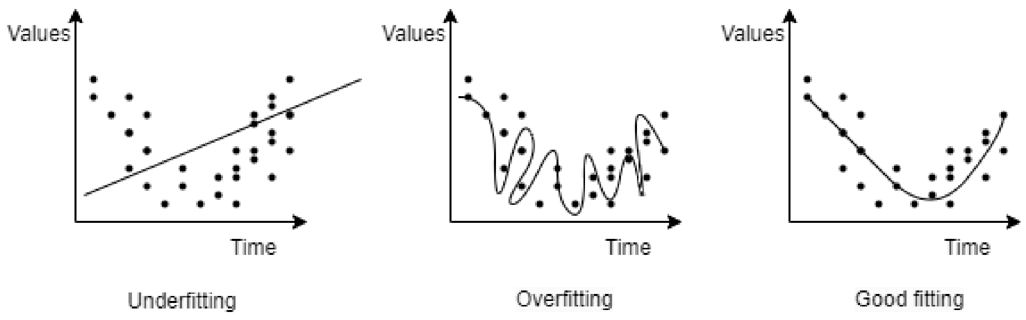


Figure 17.4 – Overfitting Illustration

I believe the image in Figure 17.5 to be an example of overfitting in my agents due to the drop off of the graph.

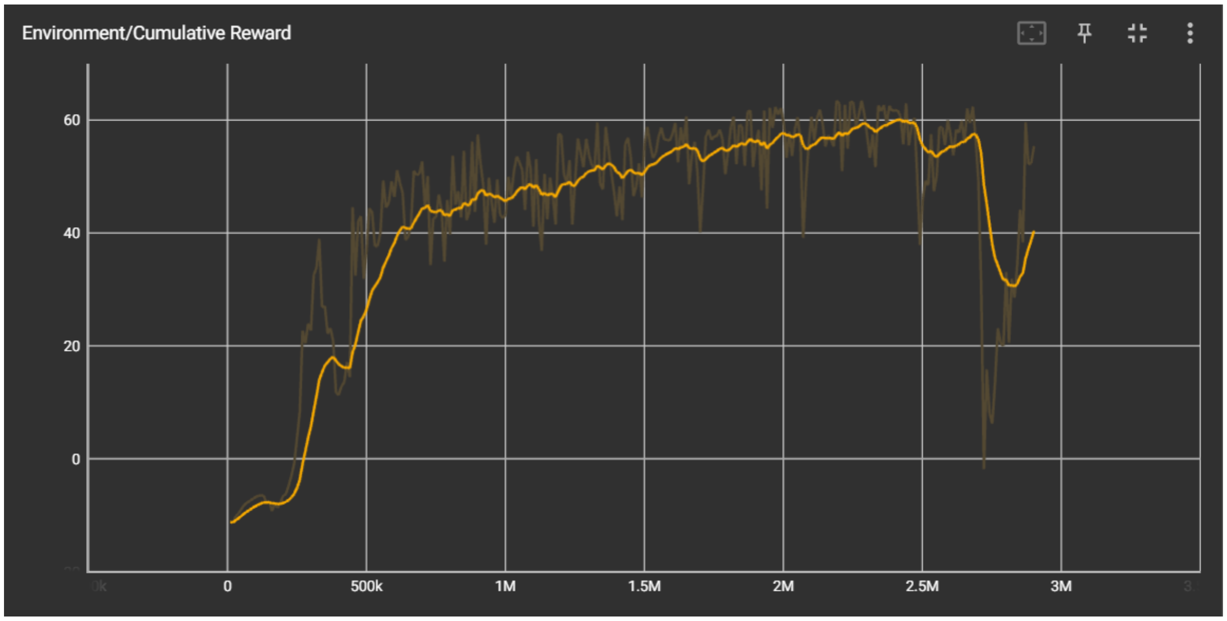


Figure 17.5 – Overfitting in ReinforcementDrive

The project determined that the ideal quantity of hidden units is 512. The utilisation of the neural network enabled the agent to acquire knowledge of the complexities of driving while simultaneously preserving an efficient training process.

17.3 Time Horizon & Time Scale

The time horizon determines the number of steps used to estimate the future rewards in the advantage calculation. By manipulating the time horizon, the training procedure can be expedited, as the agent can acquire a greater understanding of the surroundings in a reduced time frame. ReinforcementDrive opted for a time horizon of 128.

The time scale is a multiplier for how quickly the training runs. A time scale of 20 was chosen so the Agents would train 20 times faster.

17.4 Results of optimal parameters

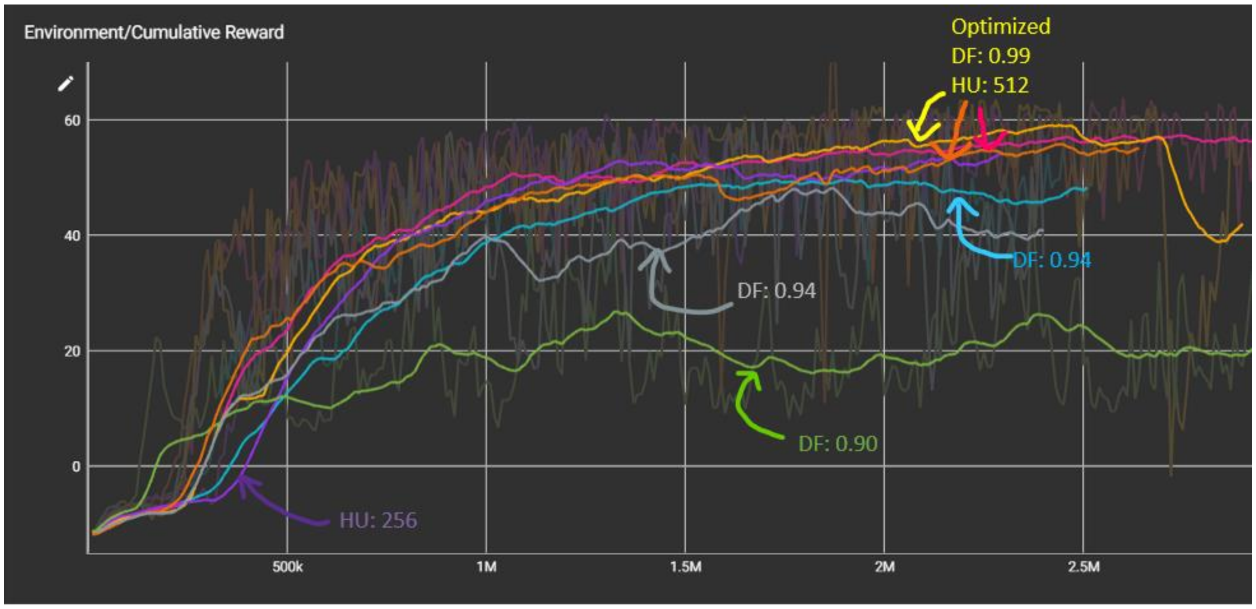


Figure 17.6 – Optimised training

Figure 17.6 shows how a variety of Hyperparameters were examined before the ideal values for the best training logic were found. Discount factor and hidden units had the biggest effect here.

18 Testing & Validation

ReinforcementDrive underwent rigorous testing and validation to ensure that the game's mechanics, agents, and output brains functioned as intended and contributed to the game's overall success. The efficacy of AI training and the attainment of a smooth gaming experience was established through these testing and validation procedures.

18.1 Mechanics Testing

The game mechanics, including the steering, checkpoints, and racing environment, were subjected to comprehensive testing across diverse scenarios. This was executed to guarantee the game's functionality, the user interface was highly reactive, and the checkpoints were correctly positioned. To ensure optimal game performance, various edge cases that may hinder gameplay, such as collisions and off-track occurrences, were taken into account.

18.2 Agents Testing

Following training, the agents underwent testing across a range of race scenarios, encompassing diverse track layouts, mirrored tracks, car configurations, and opponent behaviours. This facilitated the detection of any unusual patterns in the agent's decision-making and adaptability. This also facilitated the chance to generate additional tracks for ultimate integration into the game.

19 Game Logic & Menus

19.1 RaceManager

The RaceManager script controls the core racing logic, such as laps, checkpoints, and agent statuses. It also handles the observed agent, manages the in-game timer, and calculates the positions of the cars during the race.

19.2 GameManager

The GameManager is a script that follows the singleton design pattern and is responsible for managing the overall state of the game. The system monitors the present status of the game (such as Preparing, Playing, or Gameover) and the chosen level of difficulty. This script is accountable for managing alterations in state and subsequently initiating corresponding events.

19.3 PrimaryMenuController

The PrimaryMenuController script is responsible for the management of the main menu in the game. The code initialises the level and difficulty selection dropdown menus and handles button presses for start and quit the game.

19.4 TimerUIController

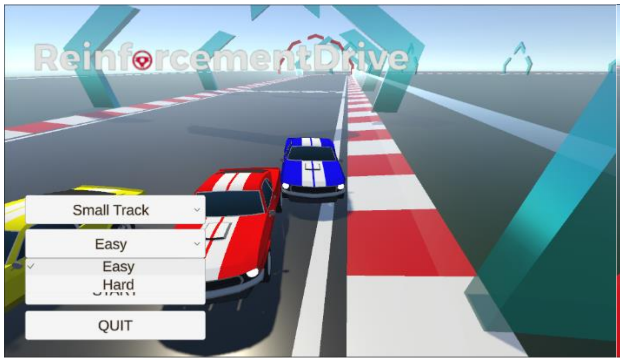
The TimerUIController script handles the countdown timer displayed at the beginning of the race. It uses a coroutine to show the countdown on the UI before the start.

19.5 DashboardController

The DashboardController script is responsible for displaying information about the player on the user interface. It shows the player’s current lap, position, and time remaining.

19.6 FinishUIController

The FinishUIController script is responsible for the management of the game over screen. It displays the final results, such as the player's position, and provides options to restart or quit the game.



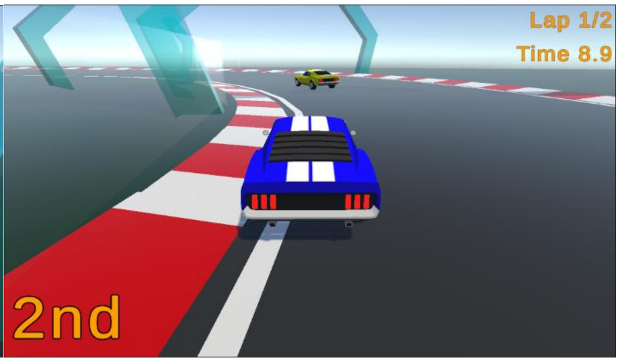


Figure 19.1 – Main Menu

Figure 17.5 – In-Game UI

Together these scripts result in a cohesive game experience by managing the essential game elements and user interfaces. The software components collaborate seamlessly to facilitate a seamless progression between menu screens, gameplay, and game-over states, all while delivering an immersive and pleasurable racing experience.

20 Ethics

Within the domain of artificial intelligence and reinforcement learning, ethical considerations hold substantial importance in ascertaining the conscientious advancement and implementation of AI systems. ReinforcementDrive, while small in scale, is no exception.

The process of training artificial intelligence models, specifically those utilising deep learning techniques, necessitates a substantial allocation of computational resources. These resources have a high energy demand, resulting in elevated carbon emissions and contributing to global climate change. While the ReinforcementDrive may be limited in scope and its ecological footprint relatively insignificant when juxtaposed with more expansive artificial intelligence systems, it is imperative to acknowledge the possible ramifications of scaling up the project or applying similar techniques in broader contexts. Researchers and developers should consider the energy consumption involved in AI training and strive to optimise their models and algorithms to reduce their ecological impact.

21 Conclusion

The objective of ReinforcementDrive was to design an AI-based car racing video game that leveraged reinforcement learning methodologies to generate flexible and captivating agents. The outcome of the project is a fully operational prototype game that features a demonstrable system, highlighting the capabilities of reinforcement learning in game development and its applicability in real-world scenarios.

The project effectively integrated diverse game mechanics, such as car controls, race tracks, and user interface elements, to establish an engaging gaming experience. The reinforcement learning agents underwent training via Unity ML-Agents and were customised to suit the racing environment, showcasing their aptitude in track navigation and competitive gameplay against the player. Furthermore, the project incorporated a modular architecture, facilitating additional scalability and personalization of the game.

Although ReinforcementDrive has accomplished its main objectives, are opportunities for additional development and enhancement. Future work may encompass the amalgamation of more diverse and challenging racing settings, the progression of sophisticated reinforcement learning methodologies like SAC, and the investigation of multi-agent frameworks for both competitive and collaborative gameplay.

In summary, ReinforcementDrive demonstrates the incorporation of reinforcement learning methodologies within the gaming sector. The prototype showcases the capabilities of AI-powered gaming experiences and adds to the overall comprehension of AI implementations in game design.

22 References

[1] Epic Games, "Unreal Engine," [Online]. Available: https://www.unrealengine.com/.

[2] D. Buckley, "Unity vs. Unreal: Choosing a Game Engine," GameDev Academy, 2022. [Online]. Available: https://gamedevacademy.org/unity-vs-unreal/.

[3] Unity Technologies, "Unity," [Online]. Available: https://unity.com/. [Accessed: 03-Apr-2023].

[4] Unity Technologies, "ML-Agents," [Online]. Available: https://unity3d.com/machine-learning.

[5] S. Kaur Arora, " Unity vs Unreal: Which Game Engine Should You Choose?," Hackr.io, 2022. [Online]. Available: https://hackr.io/blog/unity-vs-unreal-engine.

[6] GameDev Academy, "Best Unity AI Tutorials," GameDev Academy, [Online]. Available: https://gamedevacademy.org/best-unity-ai-tutorials/.

[7] Unity Technologies, "Artificial Intelligence for Beginners," Unity Learn, [Online]. Available: https://learn.unity.com/course/artificial-intelligence-for-beginners.

[8] JM. Carew, "Reinforcement Learning," TechTarget, [Online]. Available: https://www.techtarget.com/searchenterpriseai/definition/reinforcement-learning.

[9] Deeplizard, "Markov Decision Processes (MDPs) - Structuring a Reinforcement Learning Problem," YouTube, 20 September 2018. [Online]. Available: https://www.youtube.com/watch?v=my207WNoeyA.

[10] A. Yang, "What is Exploration vs. Exploitation in Reinforcement Learning?,", 25 July 2022. [Online]. Available: https://angelina-yang.medium.com/what-is-exploration-vs-exploitation-in-reinforcement-learning-a3b96dcc9503.

[11] Unity Technologies, "Getting Started with ML-Agents," GitHub, [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Getting-Started.md.

[12] NixArt, "Modular Lowpoly Track & Roads Free," Unity Asset Store, [Online]. Available: https://assetstore.unity.com/packages/3d/environments/roadways/modular-lowpoly-track-roads-free-205188.

[13] Prometeo, "Prometeo Car Controller," Unity Asset Store, [Online]. Available: https://assetstore.unity.com/packages/tools/physics/prometeo-car-controller-209444.

[14] RoBust Games, "How To Make A Simple Car Controller In Unity," YouTube, [Online]. Available: https://www.youtube.com/watch?v=QEqhaxGg_EE.

[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," arXiv:1707.06347 [cs], 20 Jul. 2017.

[16] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," arXiv:1801.01290 [cs], 4 Jan. 2018.

[17] A. Cohen, E. Teng, V-P. Berges, R-P. Dong, H. Henry, M. Mattar, A. Zook, S. Ganguly, "On the Use and Misuse of Absorbing States in Multi-agent Reinforcement Learning," arXiv:2111.05992 [cs.LG], Nov. 2021.

[18] E. Merdivan, S. Hanke and M. Geist, " Modified Actor-Critics.”, arXiv:1907.01298 [cs.LG], 2 Jul 2019.

[19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," arXiv:1509.02971, 9 Sep 2015.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," arXiv:1312.5602, 19 Dec 2013.

[21] J. Chaki, “Brain Tumor MRI Image Segmentation Using Deep Learning Techniques”. MIT Press, 2021.

[22] Arxiv Insights, “An introduction to Policy Gradient methods” [Online]. Available: https://www.youtube.com/watch?v=5P7I-xPq8u8.

[24] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization, arXiv:1502.05477 [cs.LG], 19 Feb 2015.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, 2014.