F21MP - Masters Project and Dissertation

Autonomous Vehicle AI for Simulated Traffic Environment

Authored by:

Gunabalan Lingam (M.Sc. Artificial Intelligence)[H00423885]

Supervisor:

Dr Jamie Gabbay

HERIOT WATT UNIVERSITY
UK | DUBAI | MALAYSIA

Student Declaration of Authorship

Course code and name:	F21MP- Masters Project and Dissertation
Type of assessment:	**Individual**
Coursework Title:	**Autonomous Vehicle AI for Simulated Traffic Environment**
Student Name:	Gunabalan Lingam
Student ID Number:	H00423885

Declaration of authorship. By signing this form:

- **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section.

- I confirm that I have read, understood and followed the University's Regulations on plagiarism as published on the University's website, and that I am aware of the penalties that I will face should I not adhere to the University Regulations.

- I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on Academic Integrity and Plagiarism

Student Signature *(type your name):* Gunabalan

Date: 17/04/2024

HERIOT WATT UNIVERSITY
UK | DUBAI | MALAYSIA

Abstract

This project aims to develop an autonomous vehicle agent capable of navigating a simulated traffic environment. Leveraging Unity's ML-Agents framework, the autonomous vehicle agent learns to navigate a track. Through integration with Unity's Navmesh agents, a traffic simulation is developed, and the autonomous vehicle agent is trained to navigate the traffic simulation environment.

Source code to the project is linked here.

The demonstration video of the project is linked here.

HERIOT WATT UNIVERSITY
UK | DUBAI | MALAYSIA

Contents

HERIOT WATT UNIVERSITY
UK | DUBAI | MALAYSIA

List of Figures

Abbreviations

CNNs **C**onvolutional **N**eural **N**etworks

RNNs **R**ecurrent **N**eural **N**etworks

DRL **D**eep **R**einforcement **L**earning

DQN **D**eep **Q**-**N**etworks

PPO **P**roximal **P**olicy **O**ptimizations

SAC **S**oft **A**ctor **C**ritic

AVs **A**utonomous **V**ehicles

ML **M**achine **L**earning

RL **R**einforcement **L**earning

1. Introduction

The development of autonomous vehicles is a significant challenge within the automotive industry, offering immense potential to revolutionise traffic safety. Current statistics indicate that over 90% of car accidents result from human errors, underscoring the urgent need for innovative solutions in vehicle navigation [1]. Major industry leaders such as Google, Toyota, Nissan, and Audi have intensified their efforts by investing substantially in creating autonomous vehicle prototypes[2].

DRL, functioning on the premise of learning through trial and error, equips vehicles with the capability to make dynamic real-time decisions, adapting seamlessly to unpredictable road scenarios. This paradigm has significantly evolved in recent years, marking a pivotal era in artificial intelligence [47]. Fusing reinforcement learning principles with deep neural networks has empowered the reinforcement learning community with advanced tools to confront complex challenges efficiently.

The use of the Unity ML-Agents toolkit for training autonomous vehicles through deep reinforcement learning (DRL) is indeed substantiated by various studies and projects that employ these technologies to develop intelligent systems capable of navigating complex environments. For instance, research has shown that using DRL within Unity's ML-Agents can effectively train agents to navigate a racing track, highlighting the feasibility of deploying such technology for more complex tasks like autonomous driving. [63]

Further, the Unity ML-Agents toolkit is noted for its robustness in stimulating environments where agents can learn various behaviours through interaction, which is critical for the training of autonomous vehicles. The toolkit's versatility allows it to be used in a range of applications, from simple game environments to complex simulations involving autonomous agents. [64]

Moreover, various DRL algorithms, including Proximal Policy Optimization and others, have been successfully applied using Unity ML-Agents to train agents for tasks like obstacle avoidance and lane navigation, further proving the capability of DRL in this domain.

An integral part of this progress involves the utilisation of simulated environments for experimentation and testing. Platforms like the Unity game engine provide an immersive and realistic space for researchers and developers to model intricate scenarios where autonomous vehicles can operate, learn, and adapt. Unity's capabilities facilitate the replication of diverse real-world scenarios, making it an ideal platform for exploring and advancing reinforcement learning applications.

1.1 Aim

There are three aims for this project:

1. Develop a vehicle agent in the Unity game engine capable of autonomous navigation using Unity ML-Agents Deep Reinforcement learning algorithms.
2. Create a traffic simulation system within Unity which satisfies the following:
    a. Modular road sections with randomised navigation.
    b. Junction logic, including crossroads and pedestrian crossings.
    c. Obstacle logic for realistic navigation challenges.
    d. Pedestrian traffic with point-to-point randomised navigation and junction crossing logic.
3. Utilise Navmesh agents in Unity to facilitate the development of the traffic simulation. Subsequently, train the autonomous vehicle agent to navigate efficiently within this dynamic traffic environment.

1.2 Objectives

1. ***Objective 1***: Establish the environment for the car agent within the Unity game engine, encompassing the design of a simple track.

2. ***Objective 2***: Integrate the Unity ML-Agents framework into the project to set up the training process for the autonomous vehicle agent. Define the observation space, action space, and reward system to facilitate effective learning.
3. ***Objective 3***:Initiate training of the car agent in a simple looped track, utilising reinforcement learning algorithms such as Deep Q-Networks (**DQN**)/Proximal Policy Optimization (**PPO**)/ Soft Actor-Critic (**SAC**) to facilitate learning through multiple episodes.
4. ***Objective 4***: Develop a traffic simulation system within Unity, featuring a road network with modular road sections and various types of junctions including crossroads and pedestrian crossings. Implement randomised navigation for both road vehicles and pedestrians, incorporating junction logic and obstacle logic. Utilise Navmesh agents in Unity to simulate realistic movement patterns, ensuring effective collision detection and avoidance mechanisms for vehicles and pedestrians alike.
5. ***Objective 5***: Monitor, record, and evaluate the performance of the autonomous vehicle agent during training, fine-tuning parameters, and reward functions to enhance navigation behaviour and efficiency. Optimise the training process to expedite learning and effectiveness.
6. ***Objective 6:*** Ensure the autonomous vehicle agent is trained to navigate the traffic scenarios and assess the effectiveness of different Deep Reinforcement Learning (**DRL**) algorithms. Conduct comprehensive testing to determine the optimal algorithm for achieving reliability and autonomy.

The ***Minimum Viable Product*** of this project is achieving Objectives 2,3 and 4 to establish a basic functional prototype.

2. Literature Review

2.1 History

The concept of vehicle automation dates back to 1918 [3], with General Motors showcasing an automated vehicle concept in 1939 [4]. This early vision was publicised via television in 1958[5]. During the 1950s, General Motors, Radio Corporation of America, and Sarnoff Laboratory initiated joint research and development (R&D) in vehicle automation [4]. This collaboration laid the groundwork for future advancements in the field.

In 1988, significant progress was made with Carnegie Mellon's NAVLAB vehicle demonstrating lane-following using camera images [6]. This development marked an important step in applying computer vision to vehicle automation.

The early 2000s saw increased R&D activities in the U.S., Europe, and Japan. These Projects focused on automated bus and truck platoons, super-smart vehicle systems, and driving scene recognition through video image processing [4]. They diversified the applications and technological breadth of autonomous vehicles. The U.S. Defence Advanced Research Projects Agency (**DARPA**) launched its Grand Challenges Program in 2004, accelerating autonomous vehicle research. The program led to successful demonstrations of autonomous vehicles in desert terrains (2005 and 2007) and later in urban settings through the **DARPA** Urban Challenge Program[3][4]. This period marked significant advancements in autonomous vehicle technology in academic and industrial sectors.

Volvo began its autonomous driving project in 2006 and unveiled a fully autonomous test vehicle in 2017. Similarly, Google initiated its autonomous vehicle project in 2009, and its subsidiary, Waymo, launched the first shared autonomous vehicle fleet, Waymo-One, in 2018 (https://waymo.com/). Tesla's announcement in 2014 about its vehicles' self-driving capabilities was another significant milestone. Today, all Tesla models are equipped

with self-driving features, reflecting the rapid integration of autonomous vehicle technology in the consumer market.

The development of autonomous vehicles has progressed from early conceptualizations to sophisticated demonstrations on full-scale platforms. Current research continues to focus on achieving full autonomy, emphasising safety and reliability.

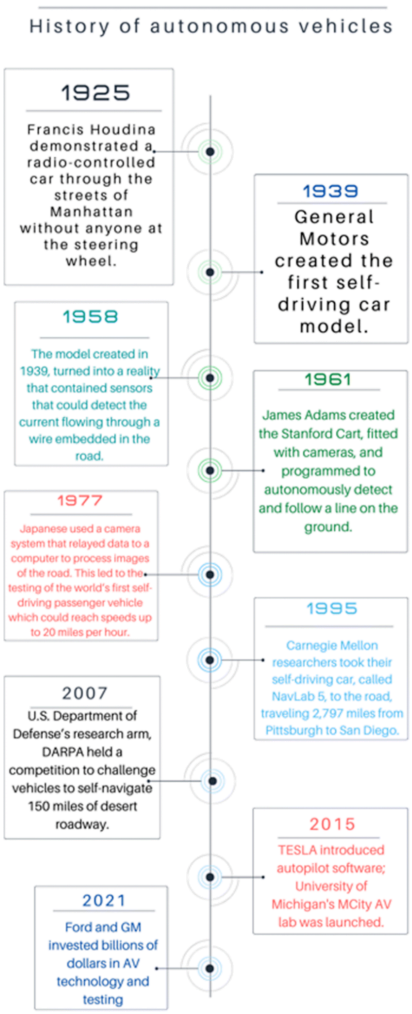


Figure 1.1 : History timeline of Autonomous Vehicles[46]

2.2 Technology advances in autonomous vehicles

2.2.1 Autonomous Vehicles Classification

Autonomous vehicles (**AVs**) have undergone significant technological advancements in recent years, transitioning the concept of self-driving cars from a theoretical idea to a tangible reality. A key factor driving the development of **AVs** is their potential to enhance road safety.

Beyond safety, **AVs** offer environmental benefits. They are projected to play a role in reducing carbon emissions, thereby contributing to environmental protection [31]. Self-driving cars have the potential to improve traffic flow and increase productivity, which could lead to significant economic impacts.

The Society of Automobile Engineers (**SAE**) classifies automated vehicles into six levels, ranging from level 0 (complete driver control) to level 5 (full vehicle control) [32]. Levels 2 and 3 of automation have been integrated into some commercial vehicles. Examples include GM's Cruise, Tesla's Autopilot, and BMW, which feature autonomous functionalities such as adaptive cruise control, automatic braking, and lane-keeping systems.

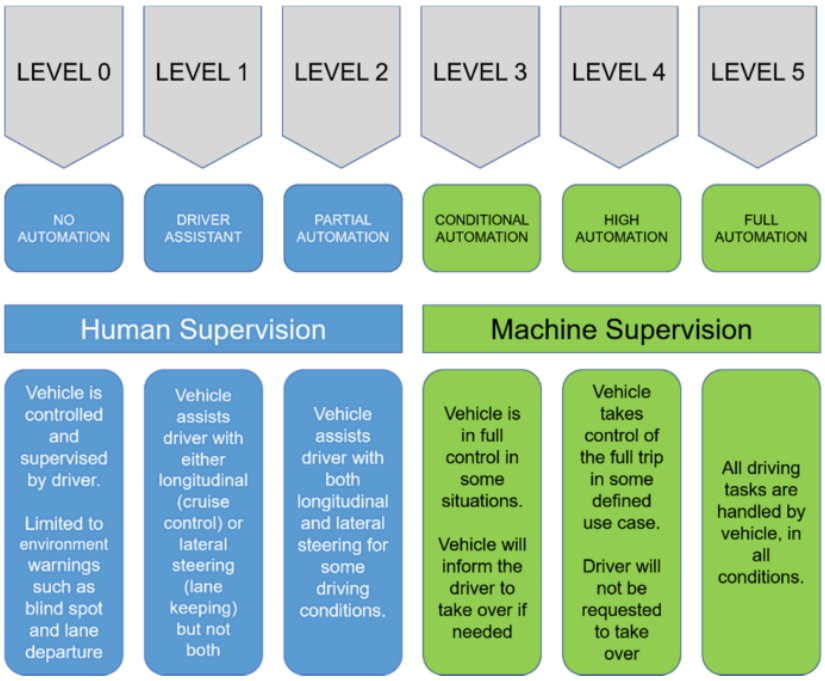


Figure 2.1 : The six levels of autonomous vehicles as described by the Society of Automobile Engineers (**SAE**) [32]

2.2.2 Advances in Autonomous Vehicle Software Systems

Autonomous vehicle software systems are organised into three core competencies: *perception*, *planning*, and *control*. These competencies interact with each other and the vehicle's external environment. Additionally, Vehicle-to-Vehicle (**V2V**) communications are recognized for their potential to enhance perception and planning through cooperative interaction between vehicles.

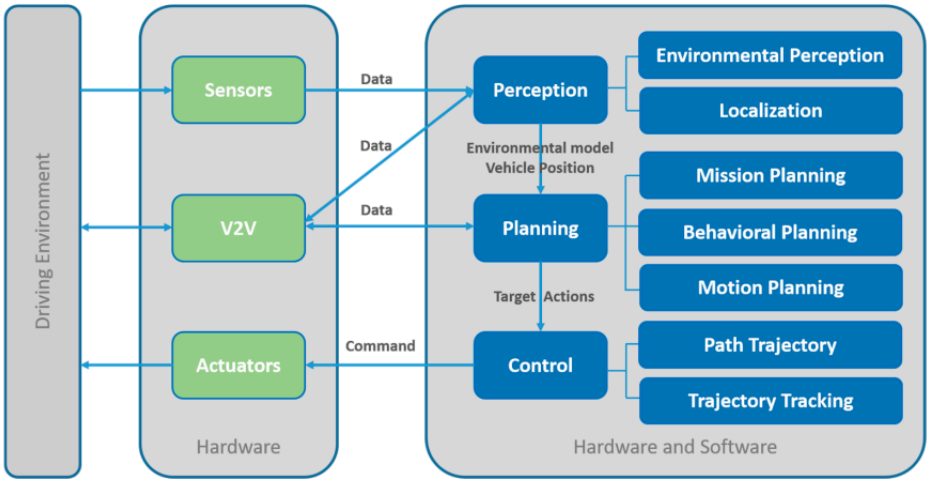


Figure 2.2 : Typical Autonomous Vehicle System

Perception: It is the process of gathering and interpreting environmental information. This encompasses environmental perception, which involves understanding the surroundings, such as locating obstacles, detecting road signs, and categorising environmental data semantically. One aspect of perception is localization, which is the vehicle's capability to determine its position relative to its environment.

Planning: It involves making strategic decisions to achieve higher-level objectives. In the context of autonomous vehicles, this typically means navigating from a starting point to a destination, avoiding obstacles, and optimising various performance metrics. This process is central to the operation of autonomous vehicles.

Control: Control refers to the execution of planned actions. This competency enables the vehicle to follow the determined path and adhere to the decisions made during the planning phase.

Let us delve into the details of these three competencies to understand better their roles and interactions within the autonomous vehicle software system.

2.2.2.1 Perception

Environmental perception is critical in autonomous vehicles, providing essential information about the driving environment. This includes identifying accessible drivable areas, the location and velocity of surrounding obstacles, and potentially predicting their future states. The primary tools for environmental perception are **LIDARs** and *cameras*, often used in conjunction with or complemented by other sensor technologies like radars and ultrasonic sensors. The tasks in environmental perception involve *road surface extraction* and *on-road object detection*.

LIDAR, standing for Light Detection and Ranging, plays a pivotal role in object detection for autonomous vehicles. It emits numerous light pulses per second and creates a dynamic, three-dimensional map of the surroundings.

Figure 2.3 illustrates the optimal detection outcomes achieved using 3D **LIDAR**, showcasing the identification of all moving objects in the scene.

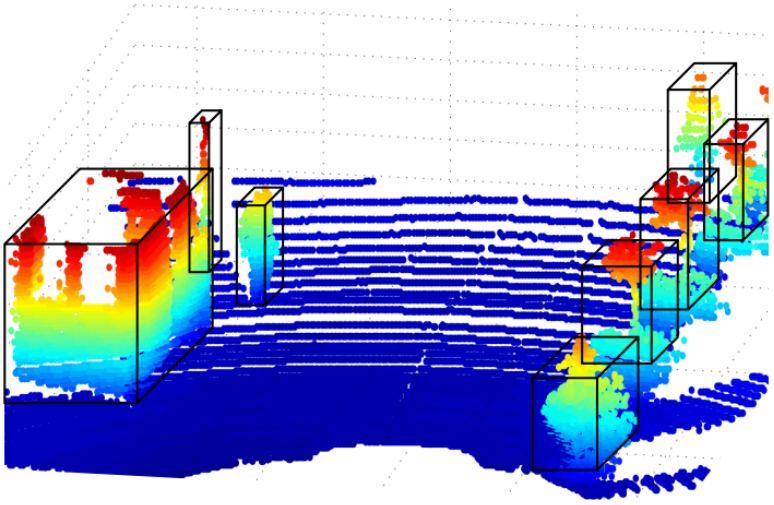


Figure 2.3 : Result from 3D **LIDAR**, adapted from [26]

LIDAR outputs are represented through sparse 3D points, with each point signifying the location of an object's surface relative to the **LIDAR**. Three main approaches to representing this data are: *point clouds, features, and grids*. [25]

1. Point cloud approaches use raw sensor data, providing detailed environmental representation but at the cost of increased processing time and memory. To optimise this, a voxel-based filtering mechanism is often applied to reduce the number of points.
2. Feature-based approaches extract parametric features from the point cloud to represent the environment. While this method is memory-efficient, it may need more accuracy due to its abstraction and dependence on the nature of the point cloud.
3. Grid-based approaches discretize space into grids, filling each grid with information from the point cloud to establish a neighbourhood of points. This method balances memory efficiency and feature dependence but faces challenges in determining optimal discretization sizes.

2.2.2.2 Planning

The development of planning systems in autonomous vehicles has evolved significantly over time. Initially, self-driving vehicles (**SDVs**) exhibited semi-autonomous capabilities, primarily executing basic functions like lane following and adaptive cruise control [27]. A notable advancement in this field was showcased in the 2007 **DARPA** Urban Challenge (DUC) [28], which demonstrated the potential of SDVs to navigate complex urban driving scenarios. Even though only six out of 35 entrants completed the final event, this challenge marked a pivotal point in establishing the feasibility of autonomous urban driving. It highlighted key research challenges in the domain [29].

The top competitors of the DUC utilised a three-level hierarchical planning framework consisting of a *mission planner, behavioural planner,* and *motion planner*. An alternative approach implemented a two-level planning system comprising a navigator and a motion planner. The navigator combined the roles of both *mission* and

behavioural planners [30]. The *mission planner* focuses on high-level objectives such as route selection and task assignments. In contrast, the *behavioural planner* makes real-time decisions for interaction with other agents and adherence to traffic rules, thereby setting local objectives like lane changes or intersection navigation. The *motion planner* generates paths or actions to accomplish these local objectives, typically aiming to reach a designated goal region without colliding with obstacles.

2.2.2.3 Control

The control aspect, often termed motion control in autonomous systems, is crucial for translating plans into actionable movements. This process involves converting the strategic intentions of the planning phase into physical actions by providing necessary inputs to the vehicle's hardware. This results in the generation of desired motions. Control in autonomous vehicles bridges the cognitive aspects, like navigation and planning, which primarily focus on velocity and position, with the physical world's dynamics of forces and energy.

A necessary function of the control system is to assess the vehicle's performance in real time. It uses various measurements to determine the system's behaviour, allowing the controller to respond to any disturbances or changes in conditions. This response is crucial to maintain or alter the vehicle's dynamics to achieve the desired state. Furthermore, detailed models of the autonomous system are employed to describe and execute the desired motion with greater precision and effectiveness. These models ensure the motion execution aligns with the intended navigational and planning decisions.

2.3 Simulation Environment

Selecting an appropriate simulation environment is important in both the development phase and problem definition of autonomous vehicle (**AV**) projects. The chosen simulator influences the data obtained from the model and shapes the problem's framework.

AV simulation encompasses a broad spectrum of study areas, including machine learning (**ML**), vehicle dynamics, and traffic simulation, leading to the creation of numerous simulators, each tailored to specific aspects of these fields.

2.3.1 Selecting Unity as Simulation Platform

In this section, I will list the reasons for selecting Unity game engine as my simulation platform.

There are simulators particularly designed for training autonomous systems. I have listed some of the popular simulators below:

1. Airsim - an open-sourced system from Microsoft AI & Research, designed to train autonomous systems. AirSim provides realistic environments, vehicle dynamics, and multi-modal sensing for researchers building autonomous vehicles that use **AI** to enhance their safe operation in the open world.
2. Apollo - Created by Baidu, is a high performance, flexible architecture which accelerates the development, testing, and deployment of Autonomous Vehicles. Apollo provides an open, reliable and secure software platform for its partners to develop their own autonomous driving systems through on-vehicle and hardware platforms.
3. Carla - Developed from the ground up by Intel Labs, Toyota Research Institute and CVC Barcelona, to support development, training, and validation of autonomous urban driving systems. CARLA has been built for flexibility and realism in the rendering and physics simulation. It is implemented as an open-source layer over Unreal Engine 4.

But for this project I am going to use Unity game engine and its ML Agents toolkit. I have listed the reasons below:

1. Unity Game Engine is a General platform that is capable of creating environments with arbitrarily complex visuals, physical and social interactions, and tasks.

2. It fulfils my project requirement, and it requires lower hardware specifications which fits well with my working environment.
3. Unity's environment has a user-friendly interface, extensive documentation, and thus the implementation process seems straightforward.
4. It has its scripting system in C# allows for creating varied gameplay and simulation dynamics.
5. The Unity engine's asynchronous physics and rendering processes allow for accelerated simulation speeds. Its ability to run simulations without rendering, control over frame rates, and quality settings enable efficient large-scale simulations.

2.3.2 The Unity Platform

In this section, I will give a brief outline about the Unity Engine and outline how the interface looks.

Unity is a 3D development platform, encompassing a rendering and physics engine and a user interface known as the Unity Editor.

A Unity Project comprises Assets, which are typically file-based and include Scenes, the environmental or level definitions of a Project. Scenes consist of GameObjects, representing objects in the environment, whose behaviours are dictated by attached components. Unity provides numerous built-in components like Cameras, Meshes, and RigidBodies and supports custom components via C# scripts or external plugins.

Figure 2.4 shows the interface of the engine.

Figure 2.4 : Unity Editor Window[33]

2.3.3 The Unity ML-Agents Toolkit

I will leverage the ML-Agents toolkit in Unity for training my vehicle AI. This section gives a brief introduction to ML-Agents.

The Unity ML-Agents Toolkit is an open-source initiative that allows researchers and developers to create simulated environments within the Unity Editor and interact with them using a Python API. This toolkit includes the ML-Agents SDK, encompassing the necessary tools to define environments and build a learning pipeline using core C# scripts. Its versatility is showcased by the inclusion of state-of-the-art Reinforcement Learning (RL) and Imitation Learning (IL) algorithms, along with support for advanced features like Self-Play and extensions such as the Intrinsic Curiosity Module (ICM) and Long-Short-Term Memory Cells (LSTM).

2.3.4 Algorithms

Key features of the ML-Agents Toolkit include a variety of example environments and two Reinforcement Learning Algorithm:

- ***Soft Actor-Critic (SAC)*** [34] and,
- ***Proximal Policy Optimization (PPO)*** [35].

In Section 6.1.1, I will evaluate these two algorithms to determine which one more effectively suits our project's needs.

In the domain of Imitation learning, it incorporates

- ***Generative Adversarial Imitation Learning (GAIL)*** [36] and,
- ***Behavioural Cloning (BC)*** [37].

The toolkit also supports Self-Play in both symmetric and asymmetric games [38], and offers extensions like **ICM** [39] and **LSTM** [40], with plans for future algorithm and model expansions.

2.3.5 ML-Agents SDK Components

The ML-Agents SDK comprises three core entities: ***Sensors***, ***Agents***, and an ***Academy***.

Agents assigned to GameObjects are central in collecting observations, executing actions, and receiving rewards. They utilise a range of ***sensors*** to gather diverse forms of data, including images, ray-cast results, or vector information. Each agent operates under a policy with a unique behaviour name, allowing for the creation of complex multi-agent scenarios. Policies can be linked to decision-making mechanisms, such as player inputs, scripts, neural network models, or Python API interactions. Developers can customise the frequency of decision requests from agents.

The learning dynamics are driven by a reward function, modifiable at any point during simulation through Unity scripts. The simulation can enter a 'done' state individually for agents or the entire environment, triggered by script calls or reaching a maximum step count.

The Academy, a singleton in the simulation, manages these aspects, tracking simulation steps and overseeing agents. It also facilitates the adjustment of environment parameters in real-time, controlling elements like physics, textures, and GameObject dynamics. This dynamic adjustment capability is crucial for creating diverse training scenarios and implementing curriculum learning strategies [41].

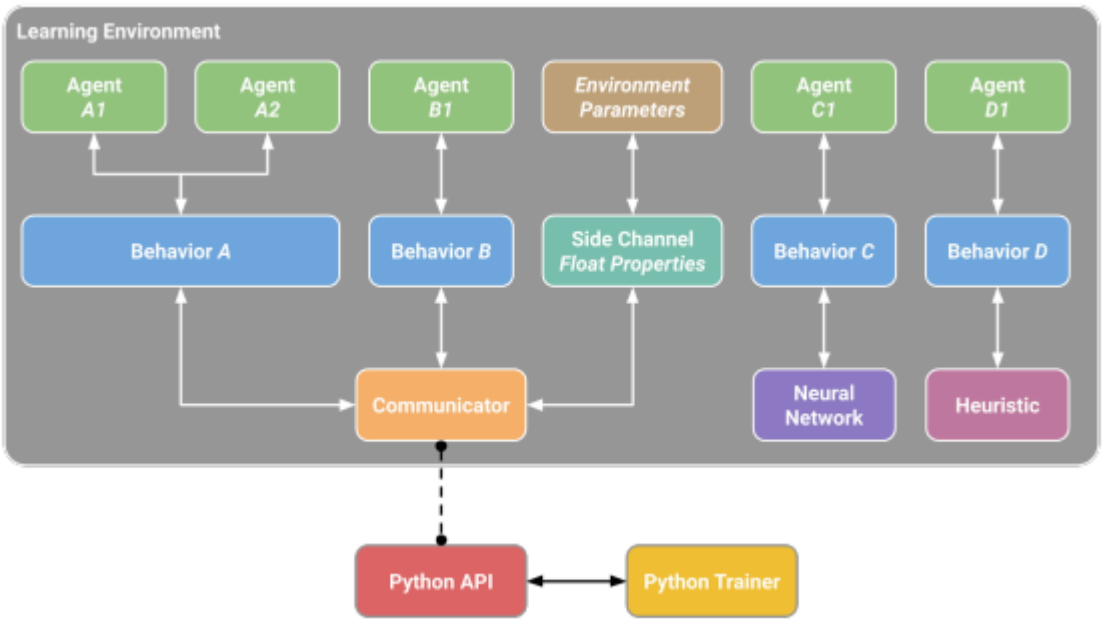


Figure 2.5 : A Learning Environment created using the Unity Editor contains Agents and an Academy. The Agents are responsible for collecting observations and executing actions. The Academy is responsible for global coordination of the environment simulation.

2.4 Deep Reinforcement Learning

2.4.1 Introduction

Reinforcement learning (**RL**) is about an agent interacting with the environment, learning an optimal policy, by trial and error, for sequential decision-making problems in a wide range of fields in both natural and social sciences, and engineering.[47]. Deep Reinforcement Learning (**DRL**) is an advanced subset of reinforcement learning (**RL**), distinguished by its use of deep neural networks. These networks are crucial for representing states and approximating functions within value and policy decision-making frameworks. At its core, **DRL** thrives on learning through interaction with the environment, echoing the natural learning processes observed in humans.

Learning through Interaction

The central premise of **DRL** is learning via environmental interaction, often likened to the 'trial and error' method. This approach is significant in understanding how an agent, such as a self-learning program, discovers the sequence of actions that yield the maximum cumulative rewards, accounting for both immediate and future outcomes.

Key Components of DRL

The architecture of **DRL** involves two components: the *agent* and the *environment*.

1. The *agent* is responsible for making decisions under uncertainty,
2. While the *environment* defines the context and consequences of those decisions.

This interaction is pivotal to the agent's learning process, encompassing the state of the environment and the agent's observations.

Reward Mechanism and Learning Cycle

In **DRL**, the reward system is fundamental. The environment provides feedback on the agent's actions, guiding it towards achieving optimal outcomes. This learning cycle begins with the agent's perception of the state and reward, followed by decision-making, action implementation, and the resulting state transition in the environment.

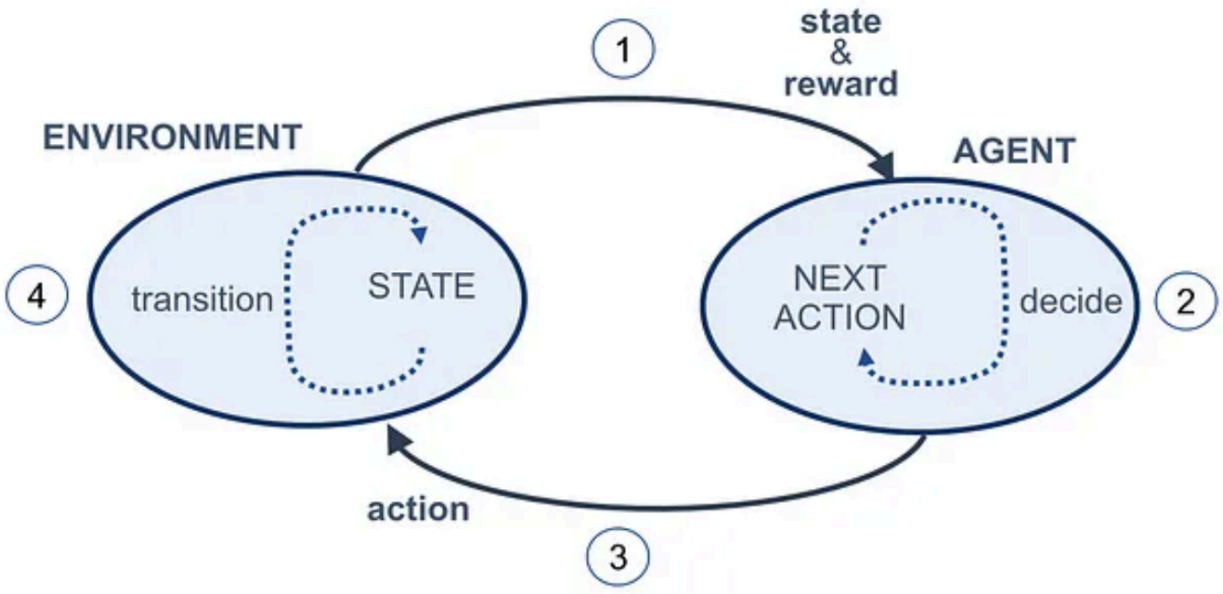


Figure 2.6 : Reinforcement Learning cycle.

The cycle starts with the Agent observing the Environment and obtaining a state and a reward (step 1). Based on this information, the Agent decides on its next action (step 2). The chosen action is then implemented in the Environment (step 3). As a result, the Environment undergoes a transition, altering its internal state due to the Agent's action (step 4). This process then repeats continuously.

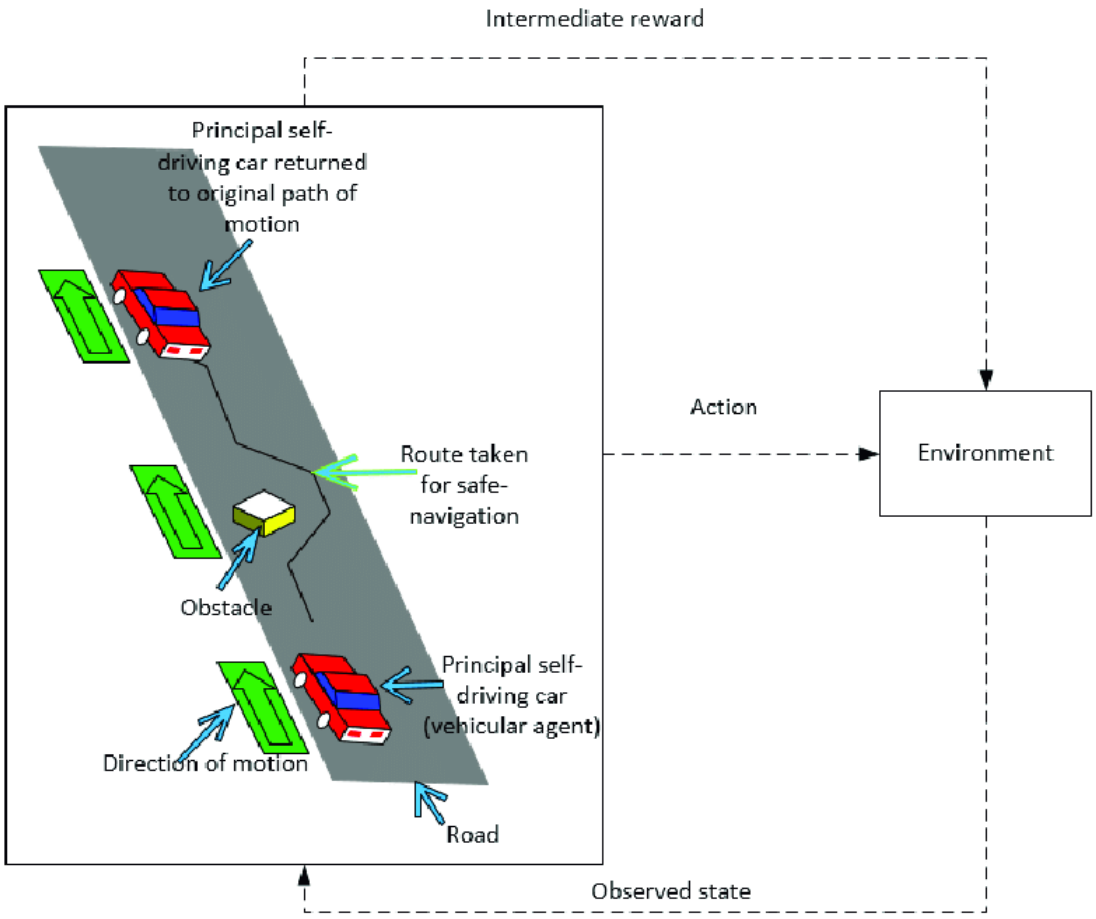


Figure 2.7: Deep reinforcement learning (**DRL**) for self-driving cars

Figure 2.7 demonstrates the **DRL** cycle in an autonomous vehicle context, showing the principal self-driving car (agent) navigating around an obstacle, receiving intermediate rewards for successful manoeuvres, and the subsequent action-response with the environment.

Episodic and Continuous Tasks

DRL categorises tasks into *episodic* (with natural endings) and *continuous* (without clear endings). The concept of return, or the total reward gathered in an episode, serves as a critical measure of the agent's performance.

The Exploration-Exploitation Dilemma

A crucial aspect of **DRL** is balancing exploration (trying new actions) and exploitation (using known beneficial actions). This trade-off is essential for maximising rewards and is a focal point in **DRL** research, representing the balance between immediate gains and potential future benefits.

Having explored Deep Reinforcement Learning, we now turn our attention to its application in the field of autonomous driving.

2.4.2 Deep Reinforcement Learning in autonomous vehicle systems

Autonomous driving represents a pinnacle objective in Artificial Intelligence (**AI**), demanding exceptional skill, attention, and experience – qualities typically associated with human drivers. Unlike humans, computers excel in continuous attention and focus, yet achieving complete autonomy in driving requires an **AI** sophistication not yet attained.

The process of developing an autonomous driving system can be divided into three distinct categories: ***Recognition***, ***Prediction***, and ***Planning***. Each category is critical in enabling a vehicle to navigate autonomously and safely.

1. ***Recognition***:

   It involves the identification of environmental elements. This includes detecting pedestrians and recognizing traffic signs, tasks that Deep Learning (DL) algorithms have significantly advanced. These algorithms, particularly Convolutional Neural Networks (CNNs), have achieved, and in some cases surpassed, human-level capability in object detection and classification [7], [8], [9], [10]. Their success, marked by triumphs in competitions like the ImageNet challenge since the introduction of AlexNet [7], extends to autonomous driving applications like lane and vehicle detection [11].

2. ***Prediction***:

   It extends beyond mere environmental *Recognition*. Autonomous agents must construct internal models to foresee future environmental states. This involves tasks such as environment mapping and object tracking. For effective *Prediction*, the integration of historical data is required, making recurrent neural networks (**RNNs**) and long short-term memory (**LSTM**) networks [12] indispensable. Their application has been notable in end-to-end scene labelling systems [13] and in enhancing object tracking in models like DeepTracking [14].

3. ***Planning***:

   This is the most challenging category, as it involves creating an efficient model that unifies *Recognition* and *Prediction* capabilities to strategize future driving actions for safe and successful navigation. The complexity lies in combining environmental understanding and dynamic Prediction to formulate action plans that avoid risks and achieve goals. In this realm, the Reinforcement Learning (**RL**) framework [15] mixed with DL, has shown promise in achieving human-level control in various applications, including the Deep Q Networks (**DQN**) model demonstrated on Atari games [16]. The incorporation of **RNNs** addresses partially observable scenarios [17].

Figure 2.8 illustrates the flow of tasks in a **DRL**-based autonomous driving system, beginning with real-world driving without prior information, progressing through the capture of driving information, and culminating in the achievement of an optimal state defined by a cost function that balances learning efficiency and safe driving.

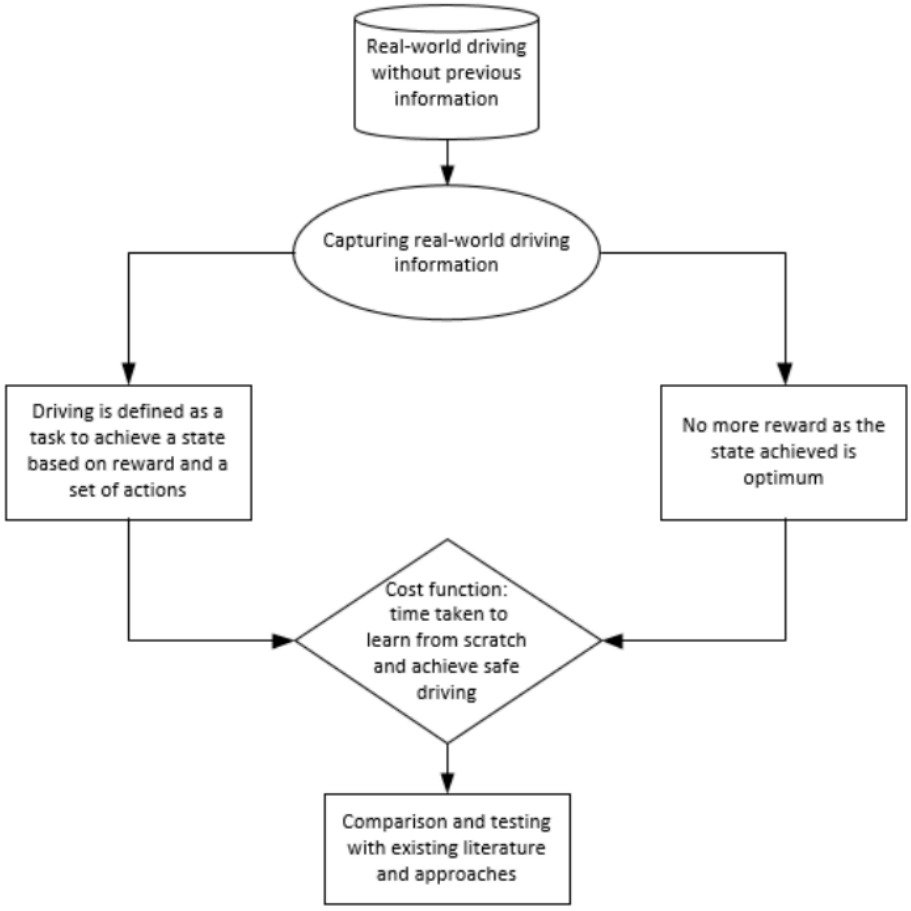


Figure 2.8 : Deep reinforcement learning based approach to autonomous driving

The integration of diverse sensor data is important in autonomous driving. Sensors range from low-dimensional (like **LIDAR**) to high-dimensional (like cameras). However, the truly relevant information for driving tasks could be much higher-dimensional. Key aspects include moving vehicles, available road space, and curb positions. Attention models filter out irrelevant data, enhancing system accuracy and efficiency, and have succeeded in image recognition [18]. Their integration into **DQN** [16] and Deep Recurrent Q Networks (**DRQN**) [17] models has been explored, suggesting their potential in autonomous driving systems.

Since I am going to work with Unity ML-Agents toolkit, I will survey the two **DRL** algorithms it is using.

2.4.3 Proximal Policy Optimization (PPO)

PPO belongs to a category known as policy gradient methods, which are used to train the decision-making function, or policy network, of a computer agent. **PPO** was developed by John Schulman in 2017 [52].

The key to **PPO** is its moderate approach to updating the policy network. If updates are too large, they could push the policy in the wrong direction, making it hard to correct later. Conversely, very small updates might slow down the learning process. To address this, **PPO** uses a technique called "*clipping*" to keep updates within an optimal range, ensuring steady progress towards the best solution.

In reinforcement learning, a policy maps states to actions, guiding an agent's decisions. Policy Gradient methods, which use gradients to update policies based on the log probabilities of actions, are central to training these agents. However, these methods are sensitive to hyperparameters and lack sample efficiency. Proximal Policy Optimization (**PPO**) addresses these challenges by optimising a balance between ease of implementation, tuning, and efficiency while ensuring updates do not deviate significantly from the previous policy, thus maintaining low training variance.

Proximal Policy Optimization (**PPO**) is an on-policy algorithm suitable for environments with both discrete and continuous action spaces. It can be efficiently parallelized using the Spinning Up implementation with MPI. **PPO** updates policies using a specific approach to ensure that the new policy does not deviate significantly from the previous one.

The key update formula for **PPO** is expressed as follows:

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[L(s, a, \theta_k, \theta) \right],$$

where the objective function $\boldsymbol{L}$ is defined by:

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \quad g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right)$$

with the function $\boldsymbol{g}$ defined as:

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

This approach ensures that the new policy remains close to the old policy, thereby limiting dramatic shifts and promoting gradual improvement. The ϵ hyperparameter effectively acts as a regulator, restricting the extent to which updates can modify the policy. This strategy enhances the algorithm's stability and prevents excessive changes in policy direction, which could negatively impact learning performance.

1: Initialize actor $\mu: S \rightarrow R^{m+1}$ and $\sigma: S \rightarrow diag(\sigma_1, \sigma_2, \ldots, \sigma_{m+1})$
2: **for** i = 1 to M **do**
Run policy $\pi\theta \sim N(\mu(s), \sigma(s))$ for T timesteps and collect (s_t, a_t, r_t)
Estimate advantages $\hat{A}_t = \sum_{t'>t} \gamma^{t'-t} r_{t'} - v(s_t)$
Update old policy $\pi_{old} \leftarrow \pi_0$
3: **for** j = 1 to N **do**
Update actor policy by policy gradient:

$$\sum_i \nabla_\theta L_i^{CLIP}(\theta)$$

Update critic by:

$$\nabla L(\phi) = -\sum_{t=1}^{T} \nabla \hat{A}_t{}^2$$

4: **end for**
5: **end for**

Figure 2.9 : Pseudocode for Proximal policy optimization (**PPO**)[53]

In the context of our project, ML-Agents employs an implementation of the Proximal Policy Optimization (**PPO**) algorithm. **PPO** utilises a neural network to approximate the ideal function that maps an agent's observations to the most advantageous action in a given state. The implementation of **PPO** within ML-Agents is based on TensorFlow and operates in a separate Python process. This process communicates with the active

Unity application via a socket connection, allowing for real-time data exchange and interaction between the software components. [54]

2.4.4 Soft Actor Critic (SAC)

The Soft Actor-Critic (**SAC**) algorithm represents a significant advancement in reinforcement learning, optimising a stochastic policy in an off-policy manner. Developed collaboratively by researchers at UC Berkeley and Google [55], **SAC** is distinguished by its incorporation of entropy regularisation. This technique encourages the policy not only to maximise expected return but also to maintain a degree of randomness or entropy, enhancing exploration capabilities.

SAC is inherently suited for environments with continuous action spaces, as evidenced by its implementation in our project. However, it is adaptable; a modified version of **SAC** can accommodate discrete action spaces by altering the policy update rule. Notably, the Spinning Up implementation of **SAC**, used in our analyses, does not support parallel processing, which may limit its scalability.

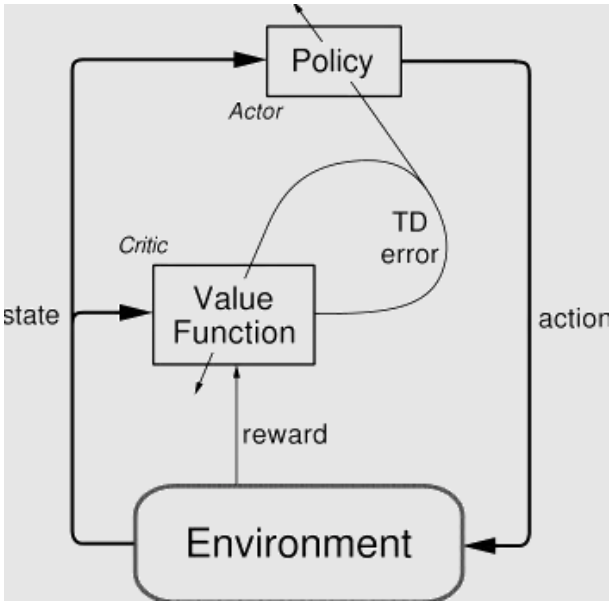


Figure 2.10 : Actor-Critic Architecture [56]

This algorithm is based on the principles of maximum entropy reinforcement learning, aiming to identify an optimal policy that maximises both the expected long-term reward and entropy. Its efficiency and effectiveness have rendered it particularly valuable in practical applications, such as real-world robotics, where such dual optimization promotes robust and versatile behavioural strategies.

$$J(\pi_\theta) = E_{\pi_\theta}[\sum_{t=0}^{T-1} \gamma^t R(s_t, a_t) + \alpha H(\pi(.|s_t))]$$

The Soft Actor-Critic (**SAC**) algorithm is a state-of-the-art reinforcement learning approach that optimises a stochastic policy in an off-policy manner [57]. Developed to enhance exploration through entropy regularisation, **SAC** simultaneously learns a policy $\boldsymbol{\pi_\theta}$ and two Q-functions $\boldsymbol{Q_{\phi 1}}$, $\boldsymbol{Q_{\phi 2}}$. This method balances expected return with a measure of randomness in the policy's actions, increasing the robustness of learned behaviours.

The SAC algorithm has evolved over time. Initially, it also learned a value function $\boldsymbol{V_\psi}$, but current implementations, including Spinning Up, focus solely on the policy and Q-functions, omitting the value function for simplicity. The primary learning mechanism for the Q-functions resembles that of TD3, using *Mean Squared Bellman Error (MSBE)* minimization with some distinctions. Both algorithms utilise a shared target computed by polyak averaging and the clipped double-Q trick, but SAC integrates entropy regularisation directly into the target:

$$Q^\pi(s,a) \approx r + \gamma \left(Q^\pi(s', \tilde{a}') - \alpha \log \pi(\tilde{a}'|s')\right), \qquad \tilde{a}' \sim \pi(\cdot|s').$$

The loss functions for the Q-networks in SAC are defined as:

$$L(\phi_i, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_i}(s,a) - y(r,s',d) \right)^2 \right]$$

where the target y is given by:

$$y(r,s',d) = r + \gamma(1-d) \left(\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s',\tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s') \right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s').$$

In policy learning, **SAC** aims to maximise both the expected future return and entropy, redefining the traditional value function $V_\pi(s)$ as:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi} \left[Q^\pi(s,a) - \alpha \log \pi(a|s) \right].$$

Utilising the reparameterization trick, **SAC** employs a squashed Gaussian policy, parameterized to account for state-dependent variations, enhancing action sampling:

$$\tilde{a}_\theta(s,\xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi \right), \qquad \xi \sim \mathcal{N}(0,I).$$

Policy optimization is thus conducted according to:

$$\max_\theta \mathbb{E}_{s\sim\mathcal{D},\xi\sim\mathcal{N}} \left[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s) \right].$$

This formulation highlights **SAC's** unique approach to incorporating entropy in the optimization process, promoting a balance between exploration and exploitation crucial for effective reinforcement learning in complex environments. [58]

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Figure 2.11 : Pseudocode for Soft Actor Critic (**SAC**) [58]

In our project, ML-Agents incorporates an implementation of the Soft Actor-Critic (**SAC**) algorithm. Unlike Proximal Policy Optimization (**PPO**), **SAC** operates on an off-policy basis, allowing it to utilise experiences gathered at any previous point in time. This is facilitated through the use of an experience replay buffer, where experiences are stored and subsequently sampled randomly during the training process. Such a mechanism significantly enhances the sample efficiency of **SAC**, often reducing the need for samples by five to ten times compared to **PPO** for learning comparable tasks. However, this efficiency comes at the cost of increased frequency in model updates.

SAC is also characterised by its "*maximum entropy*" framework, which intrinsically promotes exploration by optimising a trade-off between expected return and entropy in the policy's action distribution. This feature makes **SAC** particularly suited for environments that are either heavy or slow (approximately 0.1 seconds per step or slower).

To effectively train an agent using **SAC**, it is essential to provide one or more reward signals that the agent aims to maximise. This approach not only drives the agent towards achieving specific objectives but also aligns with the broader goal of enhancing performance through strategic exploration and exploitation. [59]

2.5 Related Work

The incorporation of Unity and ML-Agents for simulating and training autonomous vehicle agents has seen significant development, as explored in various recent studies. Here, we review key contributions in this area, highlighting their relevance to the project's focus on leveraging Unity's ML-Agents for developing an autonomous vehicle capable of navigating a simulated traffic environment.

Zhijing Jin and their colleagues have significantly contributed to this field with their study on "*3D Traffic Simulation for Autonomous Vehicles in Unity and Python*" [60]. They utilise Unity to create a 3D simulation that integrates real-time traffic data, enhancing the realism and applicability of the training environment for autonomous vehicles. This approach not only allows for dynamic and flexible training scenarios but also supports the implementation of deep reinforcement learning algorithms. A similar simulation environment is required to be achieved for our project as it provides a robust platform for developing and testing the autonomous vehicle agent's navigation capabilities in varied and unpredictable traffic conditions, mirroring the real-world complexities that autonomous vehicles will face.

Reza Mahmoudi and Armantas Ostreika's research [61] focuses on the use of the Proximal Policy Optimization (**PPO**) algorithm and Generative Adversarial Imitation Learning (**GAIL**) within Unity to train agents for obstacle avoidance. Their findings underline the effectiveness of combining behaviour cloning with

reinforcement learning to enhance decision-making in dynamic scenarios. This study is particularly relevant to our project's objective to train an autonomous vehicle agent using Unity ML-Agents. Implementing such advanced learning techniques could significantly improve the agent's ability to understand and react to sudden changes in the environment, such as unexpected obstacles or changes in traffic patterns.

Lastly, the work by Jurij Kuzmic and Günter Rudolph [62] on using Unity 3D for simulating autonomous motorway traffic and emergency corridor management offers insights into managing complex traffic patterns and emergency scenarios in a simulated 3D environment. Their research supports the feasibility of using such simulations for comprehensive training and testing of autonomous systems, which is similar to our goal of creating a traffic simulation that includes modular road sections, junction logic, and realistic navigation challenges.

The studies reviewed provide a comprehensive understanding of the current capabilities and innovations in using Unity for autonomous driving simulations. By integrating these insights, especially the advanced reinforcement learning strategies and the effective simulation of real-world traffic scenarios, our project can enhance the robustness and realism of the autonomous vehicle agent's training process. This would not only ensure a more effective learning environment but also contribute to the broader field by applying these technologies to meet specific navigation challenges within simulated traffic systems.

2.6 Social and Ethical implications of autonomous vehicles

The emergence of autonomous vehicles (AVs) introduces complex ethical considerations. Ethical debates in the AV industry, as reported in scientific literature and industry narratives, primarily centre around safety, cybersecurity, and moral decision-making in extreme traffic situations. These discussions often reference the trolley problem, a moral dilemma used to explore the decision-making process of AVs in life-threatening scenarios.

2.6.1 Safety and Cybersecurity

Recent developments in the safety and cybersecurity of autonomous vehicles (AVs) continue to focus on enhancing their reliability and addressing vulnerabilities. Researchers are exploring hybrid systems where remote human operators can intervene in complex scenarios like highway merging to ensure safety. This method considers the coordination between autonomous vehicles to minimise human intervention, potentially allowing for a scalable system where a small number of supervisors could manage multiple vehicles effectively (MIT News). In the realm of cybersecurity, the automotive industry faces significant challenges due to the increasing connectivity and complexity of AV systems. New regulatory frameworks, such as those proposed by the World Forum for Harmonization of Vehicle Regulations, are being developed to establish cybersecurity as a critical aspect of automotive quality. This includes comprehensive standards for hardware and software security throughout a vehicle's lifecycle (McKinsey & Company). Moreover, the concept of "Poltergeist" attacks has been identified, where adversarial machine learning techniques can induce 'hallucinations' in autonomous systems, creating false perceptions from actual visual inputs. This highlights the growing need for sophisticated cybersecurity measures that can protect against both conventional and novel cyber threats (Northeastern Global News).

These efforts collectively aim to enhance the trustworthiness and functional safety of autonomous vehicles as they become increasingly integrated into everyday transportation.

2.6.2 Moral Decision-Making

Researchers are recognizing that the binary choices presented by the trolley problem do not accurately represent the complexity of real-world driving scenarios. Instead, they are creating experiments to understand and quantify moral decisions in everyday driving situations, such as choosing whether to run a red light or exceed speed limits under different circumstances. To achieve this, researchers at North Carolina State University have developed a series of virtual reality experiments to collect data on how humans make moral judgments in low-stakes traffic

situations. This involves scenarios with varying degrees of urgency and consequence, helping to build a robust dataset to train **AI** algorithms for moral decision-making in AVs. The ultimate goal is to incorporate these findings into **AI** systems that can mimic human moral judgments, ensuring that autonomous vehicles act in ways that align with societal values and ethics. These studies use a framework called the Agent Deed Consequence (ADC) model, which considers the character or intent of the decision-maker (agent), the action they take (deed), and the outcome of that action (consequence). By creating multiple variations of each scenario, researchers can gather comprehensive insights into the moral evaluations of different actions, which can then be translated into algorithmic guidelines for autonomous driving systems. The outcome of this research is expected to lead to more ethically aware algorithms that guide AVs in making decisions that are not only safe but also morally sound, addressing a broader spectrum of everyday driving decisions rather than extreme life-or-death scenarios (NC State News) (ScienceDaily) (TechXlore) (Autonomous Vehicle International).

2.6.3 Accountability and Human Oversight

As autonomous vehicle (AV) technology advances, the shift in responsibility from drivers to system designers and remote operators raises significant accountability and oversight issues. Recent legislative and regulatory developments reflect these concerns. For instance, states like Pennsylvania and Mississippi have enacted laws to enable broader use of AVs without human drivers, focusing on ensuring these systems meet safety and operational standards (Faegre Drinker Biddle & Reath LLP). Similarly, the National Highway Traffic Safety Administration (NHTSA) in the U.S. is being urged by labour groups to enhance oversight to address safety concerns and establish clear accountability measures for AV manufacturers and operators (National Law Review). Internationally, the European Union is proposing the Artificial Intelligence Act, which includes strict regulations for high-risk AI systems, such as autonomous vehicles. This act is designed to ensure rigorous testing, proper documentation, and an accountability framework that emphasises human oversight (World Economic Forum). This kind of regulation aims to balance innovation with safety and ethical considerations, ensuring that AVs operate under stringent ethical guidelines and are accountable for their performance in real-world scenarios.

These developments indicate a global trend towards greater regulatory scrutiny and standardised accountability in the rapidly evolving field of autonomous vehicles.

2.6.4 Employment and Economic Impact

The rise of autonomous vehicles (AVs) brings significant implications for employment and the economy, particularly within the driving sectors. The transition to widespread use of AVs is likely to impact a range of jobs, from long-haul trucking to taxi services. According to a report by Deloitte, advancements in self-driving technology could eliminate the need for approximately 380,000 long-haul truck drivers over the next five years. This shift is anticipated due to the increased efficiency and safety of AVs, potentially leading to a decrease in demand for human drivers (Deloitte United States). Moreover, the broader adoption of autonomous vehicles could transform the economic landscape significantly. Research by ARK Investment Management suggests that the shift to autonomous taxis alone could add approximately $26 trillion to global GDP annually by 2030. This dramatic increase is attributed to the cost savings from reduced need for personal vehicle ownership and the conversion of unpaid driver time into productive activity. However, this shift also entails potential GDP losses from decreased sales of fuel and gas-powered vehicles (Ark Invest). In light of these changes, strategies to mitigate job losses are crucial. Policy interventions might include retraining programs, support for transitioning workers into new roles within the AV ecosystem, and broader economic measures to harness the productivity gains from autonomous technology while safeguarding against adverse social impacts (U.S. Department of Commerce) (OSU DESIS Lab).

These developments highlight the dual-edged nature of technological advancement in the autonomous vehicle sector—promising significant economic benefits and efficiencies while posing challenges for workforce adaptation.

2.6.5 Traffic Management and Environmental Impact

Autonomous vehicles (**AVs**) offer promising benefits for traffic management and environmental impact, but they also present challenges that need careful consideration and balance. Research indicates that **AVs** can significantly improve traffic efficiency and reduce emissions by optimising driving patterns and reducing idling times at intersections. For instance, studies have demonstrated that connected autonomous vehicles can be taught to minimise stops at red lights, communicating with traffic signals to ensure smoother flows. This not only reduces fuel consumption but also decreases carbon emissions substantially, with estimates suggesting a potential reduction in fuel consumption by up to 18% and carbon dioxide emissions by 25% in scenarios where **AVs** dominate the roads (MIT News). However, there are concerns about the potential increase in traffic volumes as AVs become more prevalent. The convenience and accessibility of **AVs** could lead to higher vehicle usage, potentially offsetting some of their efficiency gains. Additionally, the energy demands of the computing power required to operate these vehicles could be substantial, contributing significantly to greenhouse gas emissions if not managed with renewable energy sources (MDPI). Moreover, the integration of **AVs** in mixed traffic conditions, especially under adverse weather, shows that while they can improve overall traffic performance, the full benefits will only be realised at higher market penetration rates. This suggests a gradual transition phase where autonomous and human-driven vehicles will need to coexist, which could complicate traffic management and safety strategies (MDPI).

In essence, while **AVs** hold the potential to transform our driving experiences and environmental footprint positively, realising these benefits without exacerbating traffic and environmental issues will require careful planning, robust policy frameworks, and continued technological advancement.

2.7 Summary

Our literature review commenced with an exploration of the historical evolution of autonomous vehicle (**AV**) technology, charting its beginnings in the early 20th century and progressing through to the present day. This narrative started by marking key milestones and influential developments that have significantly shaped the trajectory of **AV** research and application, providing a foundational understanding for the subsequent technical advancements.

We then transitioned to examining the current technological landscape of **AVs**, specifically the classification systems defined by the Society of Automobile Engineers (**SAE**) and the critical technical components involved in **AV** operations—*perception, planning, and control*. This section covered the integration of complex software systems that enable vehicles to interact intelligently with their environments.

Deep reinforcement learning (**DRL**) was the focus of the next part of our review. Here, we discussed how **DRL** enhances autonomous navigation capabilities through sophisticated learning mechanisms that allow vehicles to adapt and make decisions dynamically. The relevance of **DRL** in developing practical, efficient, and safe **AV** systems was highlighted, emphasising its role in advancing autonomous driving technologies.

Finally, our review detailed the selection of Unity as the simulation platform, particularly for utilising its ML-Agents toolkit. This section underscored the advantages of Unity in providing robust tools for the simulation and testing of **AV** technologies, including the use of advanced reinforcement learning algorithms such as Proximal Policy Optimization (**PPO**) and Soft Actor-Critic (**SAC**).

In summary, this literature review has meticulously outlined the progression, challenges, and technological breakthroughs in **AV** development, illustrating the critical role of simulation technologies like Unity in fostering the next generation of autonomous vehicles. Through this comprehensive examination, we have set the stage for further explorations and innovations in the field of autonomous driving.

3. Requirement Analysis and Evaluation

The requirement analysis section of our project outlines the specific needs and functionalities necessary to achieve the defined objectives. Considering the project's *context* and *objectives*, we will list functional and non-functional requirements using a requirements review table such as MoSCoW.

3.2 MoSCoW

M	Must Have
S	Should Have
C	Could have
W	Won't have
F	Functional Requirement
NF	Non-Functional Requirement

3.3 Functional Requirements

ID	Description	MoSCoW	Objective
F1	Integrate Deep Reinforcement Learning techniques with UnityMLAgents framework to develop autonomous vehicle AI.	M	Objective 1, Objective 2
F2	Select and implement a Reinforcement Learning algorithm	M	Objective 2

	in Unity.		
F3	Create a traffic simulation in Unity, including roads, traffic signals, and pedestrians.	M	Objective 3
F4	Integrate the autonomous vehicle with the traffic simulation in Unity.	M	Objective 4
F5	Implement coding parameters for autonomous vehicle decision-making in simulated traffic.	S	Objective 4
F6	Conduct training sessions to improve autonomous vehicle performance in different traffic conditions.	S	Objective 5
F7	Document behaviours, readings, and outcomes from training to refine autonomous vehicle for real-world use.	M	Objective 5

3.4 Non-Functional Requirements

ID	Description	MoSCoW	Objective
NF 1	The autonomous vehicle model should demonstrate efficient decision-making, adaptability, and safety in traffic conditions.	M	Objective 5
NF 2	Ensure scalability for future enhancements in traffic scenarios and autonomous vehicle capabilities.	S	Objective 2
NF 3	Develop a user-friendly interface for interaction with the autonomous vehicle and simulation.	S	Objective 3,Objective 4
NF	Maintain system stability, reliability, and robustness during	M	Objective 3,Objective 4

4	training and evaluation.		
NF 5	Establish documentation for project findings, insights, and improvements.	M	Objective 1,Objective 2, Objective 4,Objective 5

3.5 Evaluation Metrics

It's essential to establish criteria that will effectively measure the success of the implementation and results of our project. Evaluation metrics are key tools that provide measurable ways to check a system's performance and quality.Some Key metrics to consider:

1. Performance Metrics:

- *Cumulative Rewards*: Measure the cumulative rewards over time to evaluate how well the reinforcement learning algorithms, such as Proximal Policy Optimization (**PPO**) and Soft Actor-Critic (**SAC**), are adapting and optimising behaviour based on the environment. This metric helps assess the convergence rate, reward stability, and final performance of the algorithms.
- *Loss Metrics*: Track the mean magnitude of the loss function over time to determine the learning stability and the effectiveness of the policy update mechanism. This includes monitoring changes in policy loss to gauge how closely the model's predictions align with expected outcomes.
- *Policy Entropy*: Use entropy as a measure of randomness in the decision-making process of the trained models. A decrease in entropy over time would typically indicate that the model is learning to make more confident and consistent decisions, transitioning from exploration to exploitation.

2. Hyperparameter Optimization:

- *Convergence Speed*: Measure how quickly the reinforcement learning algorithm converges to an optimal or near-optimal policy based on different sets of hyperparameters. This metric is crucial in evaluating the

effectiveness of the selected hyperparameters like batch size, buffer size, learning rate, and others in achieving faster and more stable learning outcomes.

- *Sensitivity Analysis*: Conduct a sensitivity analysis to determine the robustness of the learning algorithm to changes in hyperparameters. This involves systematically varying each hyperparameter (such as learning rate, epsilon, and beta) within a certain range and observing the impact on performance metrics like cumulative rewards and loss. This analysis helps identify which hyperparameters are most influential in improving or degrading the performance of the agent.

3. Pooling Optimization:

- *Memory Usage*: Monitor the memory usage of Unity, particularly how it varies before and after implementing object pooling techniques. Object pooling is critical in managing memory efficiently by reusing objects rather than continuously creating and destroying them, which is especially important in simulations with a high instantiation rate of objects like vehicles, pedestrians and buildings.
- *Instantiation and Destruction Rates*: Evaluate the rates of object instantiation and destruction to assess the impact of pooling on reducing the computational overhead associated with these operations. Lower rates of destruction and instantiation when using pooling indicate effective memory management, which can lead to smoother performance and reduced lag.
- *Garbage Collection Impact*: Analyse the frequency and performance impact of garbage collection events in the Unity project. Effective pooling should minimise the need for garbage collection by reducing the creation of temporary objects, thereby enhancing performance stability.

4. Project management

4.1 Professional, Legal, Ethical and Social issues

4.1.1 Professional issues

In this project, the code and algorithms for the autonomous vehicle agent are developed adhering to high professional standards. The coding practices align with guidelines set forth by authoritative bodies in computing and software engineering. All external resources, such as libraries and frameworks, are appropriately credited and used in compliance with their respective licences.

4.1.2 Legal issues

The project strictly respects legal aspects, particularly concerning data privacy and security. It ensures compliance with intellectual property laws, especially when incorporating third-party software or data sets. The usage of all external resources is within the bounds of their licensing agreements.

4.1.3 Ethical issues

Given the project's focus on simulation, direct ethical issues related to data collection or human subjects are absent. However, for broader ethical considerations related to the deployment of autonomous vehicles in real life, refer to section 2.6(Social and Ethical implications of autonomous vehicles in Literature review) of the report.

4.1.4 Social issues

Since this project is based on simulations and does not involve direct interaction with individuals or social groups, it does not present immediate social issues. Nonetheless, the potential social implications upon real-world application are discussed in section 2.6(Social and Ethical implications of autonomous vehicles in Literature review), considering the broader impact of autonomous vehicle technology.

4.2 Project plan

The project is structured into two phases: Deliverable 1 (**D1**) and Deliverable 2 (**D2**).

Gantt charts are utilised to illustrate the timeline for both **D1** and **D2** below.

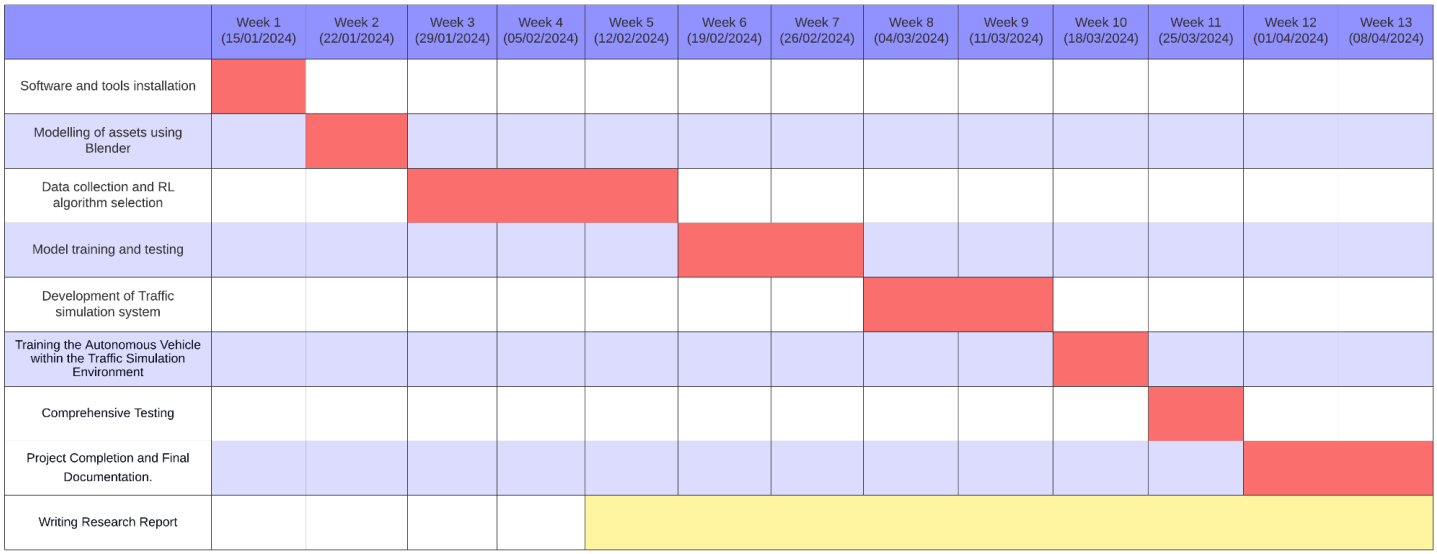
Timeline for **D1**:

	Week 1 (11/09/2023)	Week 2 (18/09/2023)	Week 3 (25/09/2023)	Week 4 (02/10/2023)	Week 5 (09/10/2023)	Week 6 (16/10/2023)	Week 7 (23/10/2023)	Week 8 (30/10/2023)	Week 9 (06/11/2023)	Week 10 (13/11/2023)	Week 11 (20/11/2023)
Project Kickoff											
Preparatory Tasks											
Literature Review											
Methodology Development											
Risk Analysis and Mitigation Strategies											
Research Report proofreading and feedback collection											
Finalization of planning report											
Writing the Report											

In the first phase, **D1**, the focus is on the project's planning stage, which includes developing the project plan and writing a planning report.

Timeline for **D2**:

The subsequent phase, **D2**, is dedicated to the actual implementation of the project and the composition of the final thesis. It is anticipated that the deadline for **D2** will fall towards the end of the third week in April, although the exact date will be confirmed at the onset of Semester 2.



4.3 Risk Analysis and Mitigation

This table presents a comprehensive risk analysis for the project, outlining potential risks and their respective mitigation strategies.

Risk ID	Risk Category	Likelihood	Impact	Mitigation
R1	Algorithm Complexity	Medium	High	Start with simpler models, progressively advance to complex algorithms.
R2	Technology Limitations	Low	High	Research alternative technologies; maintain flexibility in approach.
R3	Integration Issues	Medium	High	Conduct robust testing phases; seek expert advice for integration.
R4	Data Privacy Concerns	Medium	High	Adhere to data protection laws; use anonymized datasets.
R5	Project Scope Creep	Low	Medium	Regularly review objectives; stick to the initial project scope.
R6	Hardware Availability	Low	Medium	Identify and secure necessary hardware early.
R7	Software Bugs and Errors	High	Medium	Implement continuous testing and debugging processes.
R8	Simulation Accuracy	Medium	High	Regularly validate simulation results with real-world data; adjust models as needed.
R9	Illness of Student/Supervisor	Low	Low	Establish contingency plans for project continuity, such as remote collaboration tools.

5. Implementation

5.1 Technology Stack

1. Development Environment:
    - Integrated Development Environment (**IDE**): Visual Studio Code used for editing, debugging, and development features.
    - Anaconda: Serves as the environment manager for Python.
2. Version Control:
    - GitHub: Employed for source code management and deploying the build files.
3. Coding Languages:
    - C# and Python
4. Machine Learning Libraries:
    - Reinforcement Learning Libraries: Includes libraries specifically for implementing reinforcement learning algorithms such as Proximal Policy Optimization (**PPO**) and Soft Actor-Critic (**SAC**).
5. Simulation:
    - Unity Game Engine: Used as our simulation environment.
    - Unity ML-Agents Toolkit: Integrates machine learning technologies with Unity.
    - Unity's NavMesh Agents: Utilised for AI pathfinding for the agents within the simulation.
6. Data Analysis and Visualization:
    - TensorBoard: Employed as the primary tool for analysing and visualising the training data from trained machine learning models, helping to track metrics and performance of the algorithms.
7. Model Creation and Animation:
    - Blender: Used for animating the pedestrian model
    - MagicaVoxel: Utilised for crafting pedestrian models.

And models required for simulation named Racing Kit, Car Kit, City Kit(Suburban), City Kit(Commercial) and Modular Buildings were downloaded from:

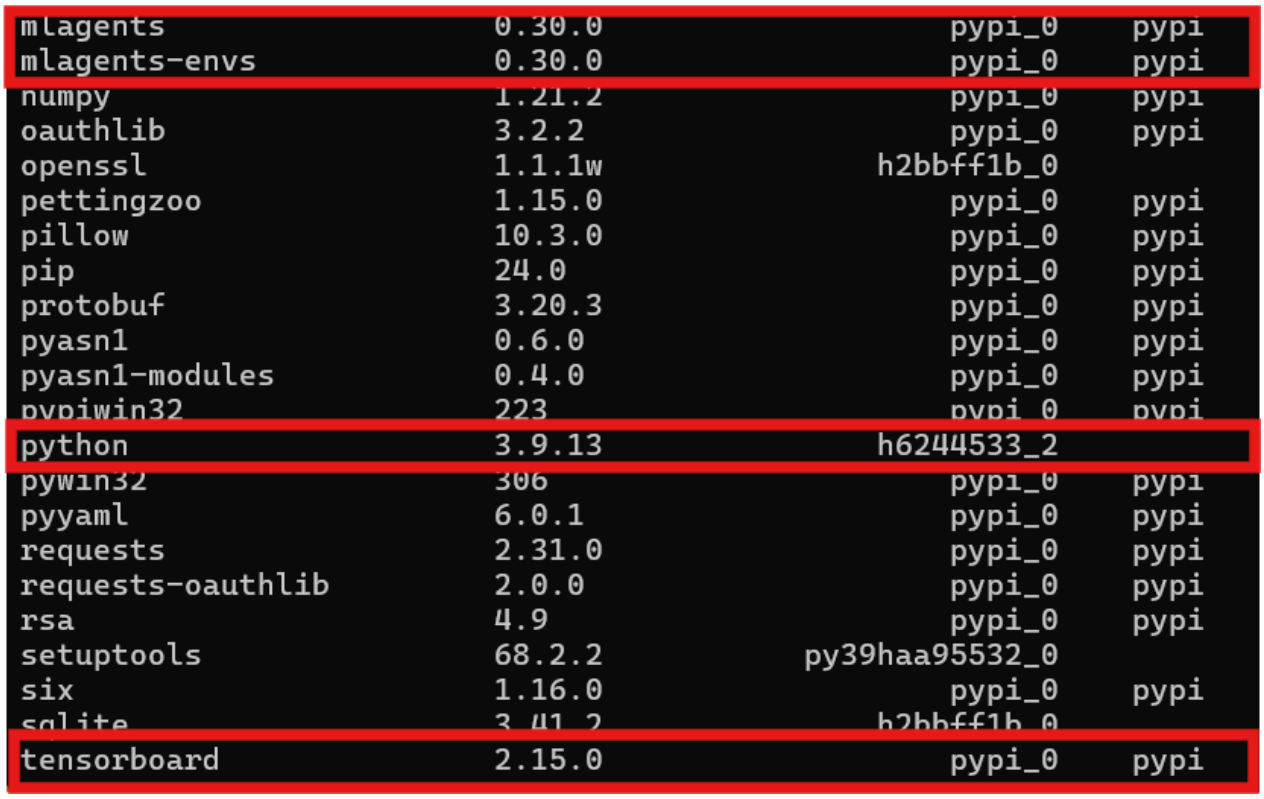
- Creator: Kenny
- Licence: Creative Commons CC0 (Public Domain)

5.2 Preliminaries

The setup for the project involved the following steps:

1. Within Anaconda Navigator, I created a new environment named "*mlagents*." This environment was activated using the Anaconda Prompt, where I subsequently installed the ML-Agents package.
2. A new Unity project was created and titled '***F21MP Autonomous Vehicle Traffic Simulation***.' To this project, I added both the ML-Agents Unity package and the Navmesh agents Unity package through the package manager.

The project environment has been successfully configured, and all necessary packages are installed, with the versions detailed in Figure 5.1.



mlagents	0.30.0	pypi_0	pypi
mlagents-envs	0.30.0	pypi_0	pypi
numpy	1.21.2	pypi_0	pypi
oauthlib	3.2.2	pypi_0	pypi
openssl	1.1.1w	h2bbff1b_0	
pettingzoo	1.15.0	pypi_0	pypi
pillow	10.3.0	pypi_0	pypi
pip	24.0	pypi_0	pypi
protobuf	3.20.3	pypi_0	pypi
pyasn1	0.6.0	pypi_0	pypi
pyasn1-modules	0.4.0	pypi_0	pypi
pypiwin32	223	pypi_0	pypi
python	3.9.13	h6244533_2	
pywin32	306	pypi_0	pypi
pyyaml	6.0.1	pypi_0	pypi
requests	2.31.0	pypi_0	pypi
requests-oauthlib	2.0.0	pypi_0	pypi
rsa	4.9	pypi_0	pypi
setuptools	68.2.2	py39haa95532_0	
six	1.16.0	pypi_0	pypi
sqlite	3.41.2	h2bbff1b_0	
tensorboard	2.15.0	pypi_0	pypi

Figure 5.1 : ML-Agents and Python Versions used for the project

The project is divided into three segments:

1. The development of a self-driving car using reinforcement learning.
2. The creation of a traffic simulation.
3. Enhancements to the self-driving car to enable it to navigate the traffic simulation.

While it is not possible to address every detail, the focus will primarily be on the self-driving car development. Detailed descriptions of all processes will be provided in the subsequent sections.

5.2.3 Part 1 - Self Driving Car with Reinforcement Learning

Environment Setup:

The first segment of our project involves developing a self-driving car capable of navigating a simple looping track. In my Unity project '***F21MP Autonomous Vehicle Traffic Simulation***', I initiated a new scene and constructed a basic track layout, which is illustrated in Figure 5.2.

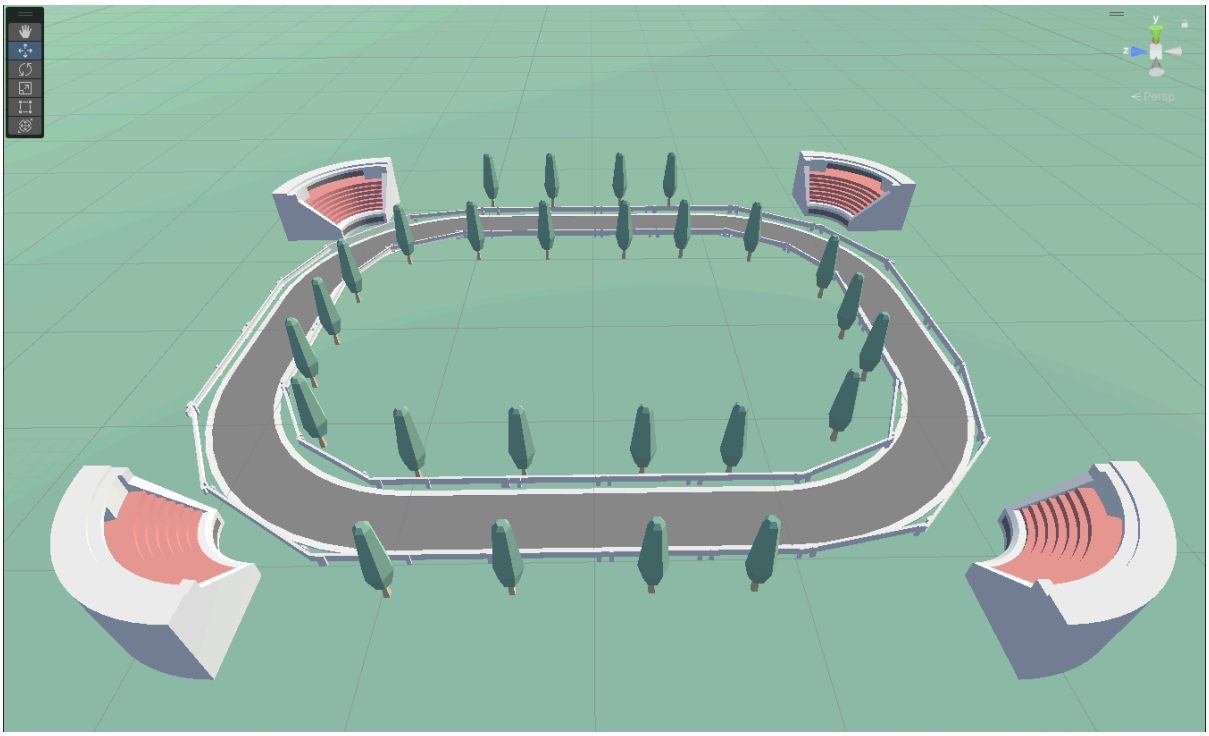


Figure 5.2 : Track layout

Car Agent Setup:

The agent consists of a basic car model equipped with controls and sensors necessary for movement during and after training:

- *Control*: These dictate how the agent traverses in the environment. Key components include colliders on each wheel and the car body to manage physical interactions with the road and other obstacles.

- *Sensors*: The car is outfitted with three sensors (ForwardDetection, DownwardDetectionX, and DownwardDetectionZ) utilising Unity's RayPerceptionSensorComponent3D to detect boundaries and rewards.

Figure 5.3 displays the car agent model with the configured colliders and sensors.

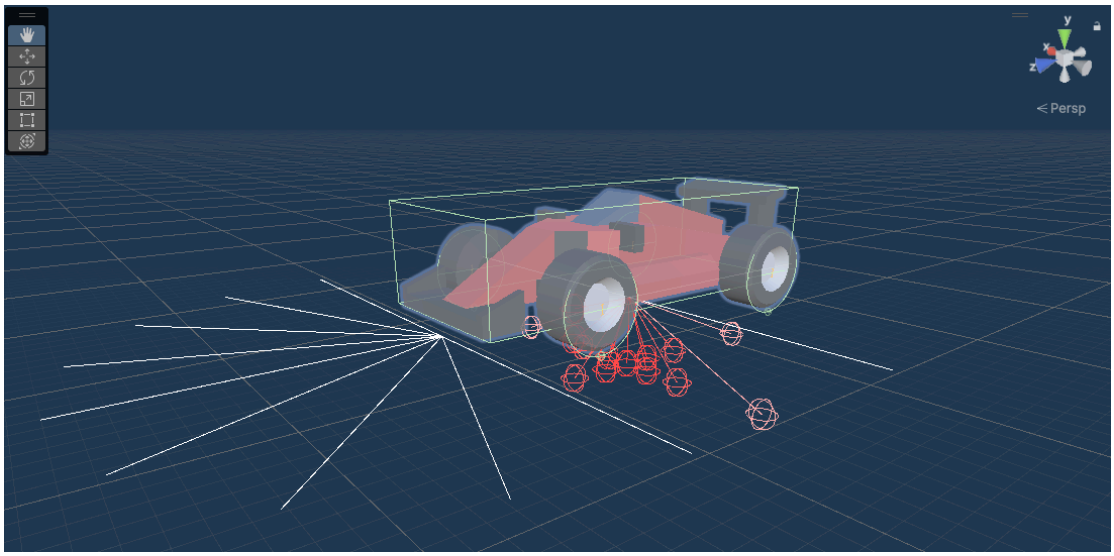


Figure 5.3 : Car Agent with colliders and sensors

The logic controlling the car is encapsulated in CarAgentController.cs, attached to the car. This script manages key vehicle parameters such as maximum motor torque (_maxMotorTorque) and steering angle (_maxSteeringAngle), which are adjustable through serialised fields. These parameters significantly influence the car's response to controls during simulations.

```
private void Move(float horizontal, float vertical)
{
    float motorInput = _maxMotorTorque * vertical;
    float steeringAngle = _maxSteeringAngle * horizontal;

    throttleInput = motorInput;
    steeringInput = steeringAngle;

    foreach (AxleInfo axleInfo in _axleInfos)
    {
        if (axleInfo.steering)
        {
            axleInfo.leftWheel.steerAngle = steeringAngle;
            axleInfo.rightWheel.steerAngle = steeringAngle;
        }
        if (axleInfo.motor)
        {
            axleInfo.leftWheel.motorTorque = motorInput;
            axleInfo.rightWheel.motorTorque = motorInput;
        }

        UpdateWheelVisuals(axleInfo.leftWheel);
        UpdateWheelVisuals(axleInfo.rightWheel);
    }
}
```

Figure 5.4 : Move function in the script.

The ***Move()*** function translates the learning algorithm's outputs into physical actions, adjusting motor torque and steering angle. The ***UpdateWheelVisuals()*** method ensures the wheels' visual states correspond to their physical positions, enhancing simulation realism.

```
public override void CollectObservations(VectorSensor sensor)
{
    var relativeVelocity = transform.InverseTransformDirection(rigidBody.velocity);
    sensor.AddObservation(relativeVelocity.x);
    sensor.AddObservation(relativeVelocity.z);
    sensor.AddObservation(rigidBody.velocity.magnitude);
}

0 references
public override void Heuristic(in ActionBuffers actionsOut)
{
    ActionSegment<float> continuousActionsOut = actionsOut.ContinuousActions;

    continuousActionsOut[0] = 0f;
    continuousActionsOut[1] = 0f;

    continuousActionsOut[0] = Input.GetAxis("Horizontal");
    continuousActionsOut[1] = Input.GetAxis("Vertical");

}
0 references
public override void OnActionReceived(ActionBuffers actionBuffers)
{
    Move(actionBuffers.ContinuousActions[0], actionBuffers.ContinuousActions[1]);
}
```

Figure 5.5 : *CollectObservations*, *Heuristic* and *OnActionReceived* functions in the script.

The car's sensors and observations are pivotal for its learning process. The ***CollectObservations()*** method is essential as it gathers and relays critical data such as relative velocity and speed to the learning algorithm. This function is central to determining the observations the agent uses to make decisions within the environment. Within this function, I begin by calculating the agent's local velocity by transforming its global velocity to the local coordinate system using the ***InverseTransformDirection()*** method. Subsequently, three observations are added to the *VectorSensor* object named "sensor":

1. The first two observations represent the local x and z components of the velocity, providing insights into the agent's movement along its local axes.
2. The third observation quantifies the agent's velocity magnitude, offering a comprehensive measure of its speed.

These observations ensure that the agent possesses accurate information about its movement dynamics, which is crucial for its decision-making during both training and real-time operations. The ***OnActionReceived()*** and

Heuristic() methods are responsible for processing the actions determined by either the **AI** or a human controller, effectively translating these decisions into the simulation.

The most important part is implementing a reward system. This is crucial for reinforcement learning. The track has boundary colliders, and the road game object has three reward colliders inside it. The car receives negative rewards for colliding with the track boundaries and positive rewards for collecting specified objects along the track. These rewards signal to the learning algorithm how well the car is performing, helping it to adjust its strategies for improved performance.

After setting up, I integrated the BehaviourParameters script as the agent's brain and the DecisionRequester script to manage decision-making frequency.

Training Process:

The car agent and environment is set up. The next step is to start the training process. For training, I utilised Unity's reinforcement learning algorithms, **PPO** and **SAC**, with separate configurations detailed in ***trainer_config_ppo.yaml*** and ***trainer_config_sac.yaml***. The hyperparameters used are shown in Figure.

```yaml
hyperparameters:
  batch_size: 120
  buffer_size: 12000
  learning_rate: 0.0003
  beta: 0.001
  epsilon: 0.2
  lambd: 0.95
  num_epoch: 3
  learning_rate_schedule: linear
network_settings:
  normalize: true
  hidden_units: 256
  num_layers: 2
  vis_encode_type: simple
reward_signals:
  extrinsic:
    gamma: 0.99
    strength: 1.0
keep_checkpoints: 5
max_steps: 2000000
time_horizon: 1000
summary_freq: 12000
threaded: true
```

Figure 5.6 : Hyperparameters for training.

We finally have all the components to run the training process. The training will benefit from multiple agents to leverage Unity's multi-agent capabilities, requiring the duplication and spatial arrangement of the tracks to avoid overlaps.

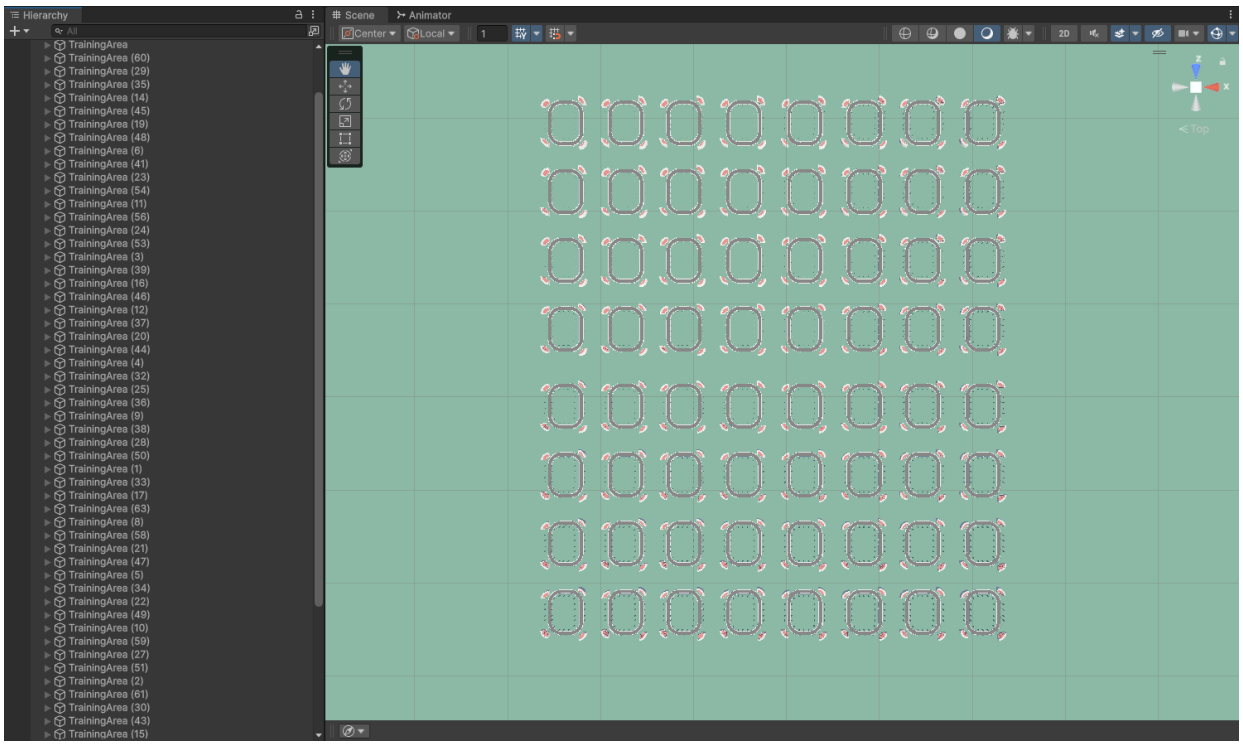


Figure 5.7 : 64 instances of training area

Training was executed through the Anaconda command prompt and successfully completed, enabling the car to autonomously navigate the track. Detailed discussions on algorithm utilisation and selection will be presented in Section 6.1.1. The build file for this part of the simulation is linked here.

5.2.4 Part 2 - Traffic Simulation System

The objective is to develop a simple traffic simulation utilising Nav Mesh to enforce constraints and manage the movement of NavMeshAgents within designated lanes and navigation sections. Traffic flows continuously, with vehicles and pedestrians moving towards randomly selected destinations.

For road vehicle traffic, the simulation will feature modular road sections with randomised navigation, integrated junction logic for managing intersections, and obstacle logic to handle other cars and pedestrians in the road. Pedestrian traffic is simulated using point-to-point navigation where pedestrians move randomly between locations, incorporating junction crossing logic to manage their interactions with roadways at crossroads and pedestrian crossings. The junctions within the simulation will include crossroads, which dictate vehicular movement and interactions, and pedestrian crossings are designed to facilitate pedestrian transit across roads.

The simulation includes four types of roads: Straight Road, Corner Road, Crossroad, and Pedestrian Crossing Road.

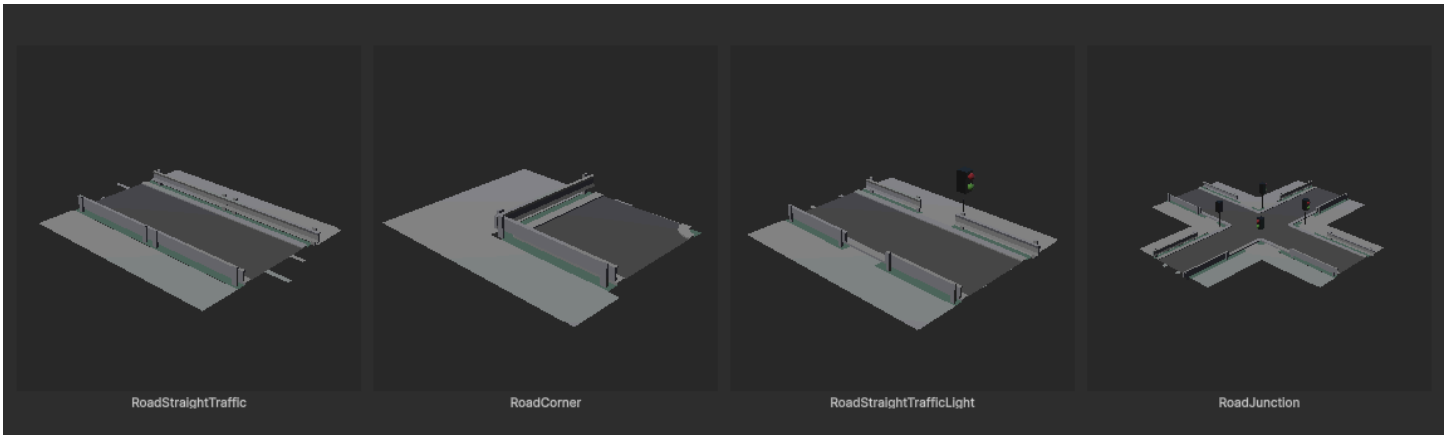


Figure 5.8 : Road types

To manage the logic for these roads, I developed three scripts: NavSection.cs, Road.cs, and Junction.cs.

NavSection.cs serves as the base class for all navigable segments within the simulation, defining essential properties and methods common to roads and junctions. This class is critical for enabling movement across different road sections or junctions.

Additionally, I implemented the NavConnection.cs script to facilitate the transition between navigable segments. This script maintains references to one or more *NavSection* instances via properties like *outConnections*, which act as pointers to subsequent sections that a vehicle or pedestrian may enter after completing the current segment. It provides vital navigation data, such as the required position and orientation for agents transitioning to a new section, ensuring proper alignment and continuous navigation paths.

The NavConnection.cs also features a dynamic path selection function, ***GetOutConnection()***, which randomly selects from the available outgoing connections. This function mimics the decision-making process of agents when choosing their next route in the simulation.

```
2 references
public NavConnection GetOutConnection()
{
    if (outConnections.Length > 0)
    {
        int index = UnityEngine.Random.Range(0, outConnections.Length);
        return outConnections[index];
    }
    return null;
}
```

Figure 5.9 : ***GetOutConnection()*** method

The NavSection.cs script will hold an array of *NavConnection* objects, which will have potential paths that vehicles can take when they reach the end of a nav section. These are crucial for defining the road network and allowing dynamic pathfinding. In this script, we also check for available vehicle spawn points during its initialization. This is achieved in the ***InitializeConnections()*** method, where connections to other road sections

are established, and the availability of vehicle spawn points are verified. Additionally, I provided mechanisms to register and deregister vehicles as they enter or leave a section.

Road.cs is a specialised version of *NavSection* that specifically represents road segments. It extends *NavSection* by adding pedestrian spawn points and a function to find an available pedestrian spawn point on the road.

Junction.cs extends *NavSection* and represents the intersections where multiple roads converge. Junctions will have the logic for crossroads and pedestrian crossing, which includes managing traffic lights. To support this, I developed a simple traffic light model in Unity using basic objects like Cubes and Spheres.

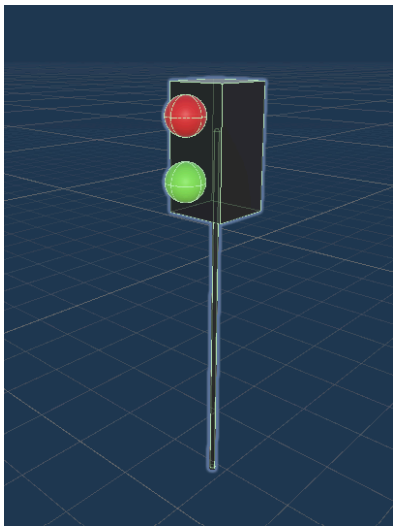


Figure 5.10 : Traffic light

I made a structured system to manage traffic flow at intersections through the use of *Phase*, WaitZone.cs, and TrafficLight.cs classes. This is to simulate traffic lights and pedestrian crossing behaviour. To achieve this, I designed the *Phase* class to handle different traffic conditions at intersections by dividing traffic signals into different operational stages or "phases." Each phase specifies which parts of the junction are open or closed to traffic and pedestrian movement. This division into phases allows for complex traffic light sequencing and pedestrian crossing management.

The Logic of Phases is that:

- Positive Zones and Lights: These are traffic lanes and corresponding traffic lights that are activated during a phase, allowing vehicles and pedestrians to pass.
- Negative Zones and Lights: Conversely, these are lanes and lights that are deactivated, thereby stopping traffic and pedestrian movements.

The method ***Enable()*** in the Phase class sets *canPass* on *WaitZone* instances and switches *TrafficLight* states.

```
2 references
public void Enable()
{
    foreach (WaitZone zone in positiveZones)
        zone.canPass = true;
    foreach (TrafficLight light in positiveLights)
        light.SetLight(true);
    foreach (WaitZone zone in negativeZones)
        zone.canPass = false;
    foreach (TrafficLight light in negativeLights)
        light.SetLight(false);
}
```

Figure 5.11 : ***Enable()*** method in Phase class

The WaitZone.cs class is created to designate specific areas where vehicles or pedestrians must wait before proceeding. Each *WaitZone* is linked to a type of traffic (either pedestrian or vehicle) and can dynamically change its state based on the current phase. Agents (vehicles and pedestrians) in the simulation check the state of *WaitZone* they encounter and if *canPass* is true, agents continue moving; if false, they stop and wait. This logic is crucial for ensuring that traffic adheres to the signals given by the traffic lights. The TrafficLight.cs class manages the visual representation of traffic signals at intersections. It controls a set of mesh renderers that change colours according to the state indicated by the active phase. The ***SetLight(bool input)*** method adjusts the colours of the traffic lights.

So, using these logics, the Junction.cs cycles through its phases based on time This cyclic operation is managed through a timer that tracks the duration of each phase and switches to the next one upon expiry.

This concluded the development for roads. The next step in the project involved creating the layout for the simulation within the scene. Then I developed the TrafficSystem.cs class to serve as the central management system for all dynamic elements within the simulation. This class is responsible for initialising and managing the traffic flow, building instantiation, and dynamic spawning of vehicles and pedestrians based on predefined configurations. I made a list of spawn points in the scene where buildings will be spawned. In the script used ***SpawnAllBuildings()*** method where each building prefab in the list is instantiated at corresponding spawn points. Similarly, for spawning vehicles and pedestrians, the system selects a random road from the available list and attempts to spawn a vehicle or pedestrian if the respective spawn point is available.

```
private void SpawnAllBuildings()
{
    foreach (Transform spawnPoint in buildingSpawnPoints)
    {
        var prefab = buildingPrefabs[Random.Range(0, buildingPrefabs.Count)];
        Instantiate(prefab, spawnPoint.position, spawnPoint.rotation);
    }
}
3 references
private void SpawnVehicle(bool reset)
{
    if (reset)
        vehicleSpawnCount = 0;

    int index = Random.Range(0, roads.Count);
    if (roads[index].GetVehicle(out VehicleSpawn spawn))
    {
        var vehiclePrefab = carPrefabs[Random.Range(0, carPrefabs.Count)];
        var newVehicle = Instantiate(vehiclePrefab, spawn.spawn.position, spawn.spawn.rotation, pool).GetComponent<Vehicle>();
        newVehicle.Initialize(roads[index], spawn.destination);
    }
    else if (++vehicleSpawnCount < roads.Count)
    {
        SpawnVehicle(false);
    }
}

3 references
private void SpawnPedestrian(bool reset)
{
    if (reset)
        pedSpawnCount = 0;

    int index = UnityEngine.Random.Range(0, roads.Count);
    if (roads[index].GetPed(out Transform spawn))
    {
        var pedestrianPrefab = pedestrianPrefabs[Random.Range(0, pedestrianPrefabs.Count)];
        var newAgent = Instantiate(pedestrianPrefab, spawn.position, spawn.rotation, pool).GetComponent<NavAgent>();
        newAgent.Initialize();
    }
    else if (++pedSpawnCount < roads.Count)
    {
        SpawnPedestrian(false);
    }
}
```

Figure 5.12 : Spawn logic functions in TrafficSystem.cs

To facilitate dynamic and continuous interaction within the simulation, I added functionality to spawn vehicles and pedestrians not just at the start but also on demand during the simulation runtime. This functionality was implemented in the Update() function, which handles the spawning of agents: pressing the 'P' key spawns a pedestrian, and the 'V' key spawns a vehicle. With the road network and spawning logic now established, Figure 5.13 displays the traffic simulation in the game view.

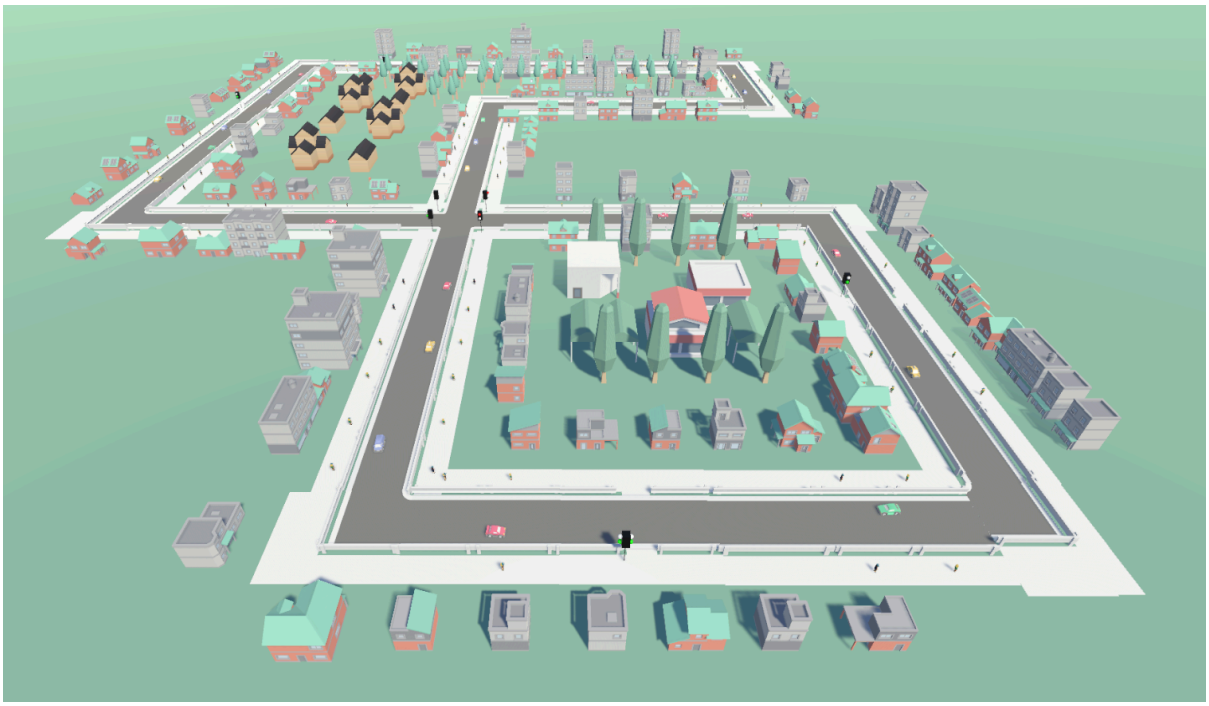


Figure 5.13 : Traffic Simulation System

Next, I developed the logic for vehicle and pedestrian AI using NavMesh agents. Figure 5.14 illustrates the different types of vehicles and pedestrians implemented in the simulation.

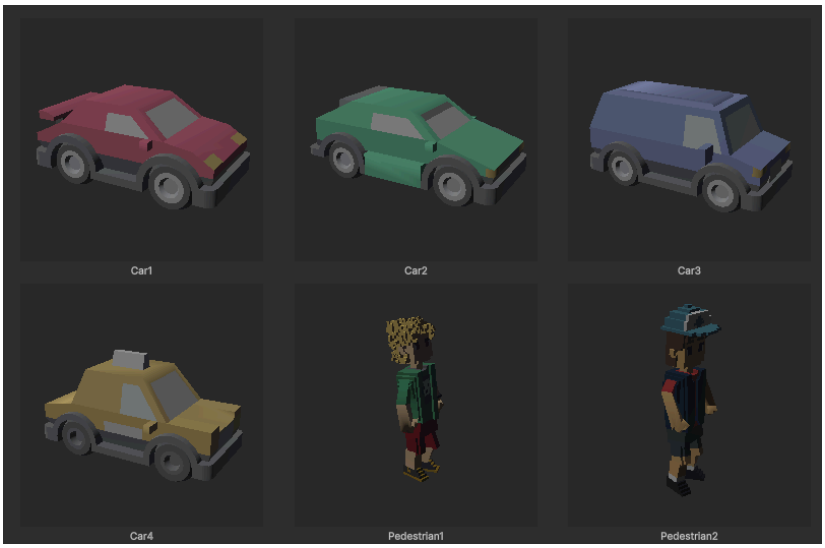


Figure 5.14 : Car and Pedestrian types used in the simulation

I created the NavAgent.cs class that forms the core logic for all moving agents in the simulation, encompassing both pedestrians and vehicles. It employs Unity's *NavMeshAgent* for pathfinding across the navigable environment. The ***SetDestination()*** method dynamically assigns a new target location fetched from ***TrafficSystem.Instance.GetPedAgentDestination()***, guiding agents towards their intended destinations. During each frame update (***Update()*** method), the agent's movement is adjusted based on its current environmental context. It checks whether the agent is on a *NavMesh* and not required to stop (***CheckStopConditions()***). If conditions like being in a wait zone where passage is not allowed (*isWaiting*) are met, the agent's velocity is set to zero, effectively making the agent stop.

The Vehicle.cs class extends NavAgent.cs is created for vehicle agent. Custom Behaviours for Vehicles: Vehicles continuously check for obstacles in their path using the ***CheckForObstacles()*** method. This method utilises ray casting to detect if any object is within a critical distance ahead. If an obstacle is detected, the vehicle stops to avoid collisions. When a vehicle encounters a road connection (*NavConnection*), it triggers the ***ChangeRoad()*** method if it's transitioning to a different section. This involves deregistering from the current section and re-registering with the new one, followed by setting a new destination based on the outgoing connection, ensuring seamless traffic flow across different road segments.

```
1 reference
private void ChangeRoad(NavConnection newConnection)
{
    RegisterVehicle(currentNavSection, false);
    speed = TrafficSystem.Instance.GetAgentSpeedFromKPH(Mathf.Min(newConnection.navSection.speedLimit, maxSpeed));
    agent.speed = speed;
    currentNavSection = newConnection.navSection;
    RegisterVehicle(currentNavSection, true);
    currentOutConenction = newConnection.GetOutConnection();
    if (currentOutConenction != null)
        agent.destination = currentOutConenction.transform.position;
}
```

Figure 5.15 : ***ChangeRoad()*** method in Vehicle.cs

With this, our traffic simulation implementation is done. The build file for this part of the simulation is linked here.

5.2.5 Part 3 - Self Driving Car in Traffic Simulation

In this part, I intended to extend the capabilities of a foundational autonomous driving model implemented in the CarAgentController.cs script, aiming to enhance its decision-making processes within a dynamic traffic simulation environment. The objective is to equip the self-driving car with advanced learning capabilities, enabling it to adapt to the traffic scenarios in the simulation, including interactions with other vehicles, pedestrians, and compliance with traffic signals. The CarAgentController.cs script served as the initial framework for the autonomous driving agent. It provided basic vehicle control and learning mechanisms, such as handling throttle and steering inputs, through the Unity Machine Learning Agents Toolkit (ML-Agents). This script had fundamental driving tasks and simple reward-based interactions, such as collecting rewards and avoiding boundaries. Building upon this foundation, I developed the MLCarTraffic.cs script that has these enhancements:

- Obstacle Detection: Utilising raycasting to detect and respond to static and dynamic obstacles in the vehicle's path.

- Enhanced Navigation: Integrating with a traffic system that includes roads (*Road*) and navigation connections (*NavConnection*), allowing for context-aware navigation decisions.
- Adaptive Behaviour in Traffic: Reacting to traffic signals and dynamically changing traffic conditions, which requires the car to learn from a variety of traffic-related scenarios and adjust its behaviour accordingly.

The reward system is designed to incentivize the self-driving car to make decisions that ensure safe, efficient, and rule-compliant driving. Rewards and penalties are assigned based on the car's interactions with its environment, which are critical for the car's learning process.

1. Speed Management: The car is penalised if it moves backwards, which is typically an undesired behaviour in traffic settings. This is implemented to discourage reverse movements and encourage forward progression, reinforcing the car's ability to navigate traffic efficiently.

```
localVelocity = transform.InverseTransformDirection(_rb.velocity).z;

//If its going backwards
if (localVelocity < -0.1f)
{
    SetReward(-0.5f);
}
```

Figure 5.16 : Negative reward for moving backward

2. Obstacle Detection and Avoidance: The car uses raycasting to detect obstacles within a certain distance. If an obstacle is detected, the car must stop, and it receives a scaled reward based on how early the obstacle is detected. The closer the obstacle when detected, the higher the reward, encouraging the car to maintain a safe stopping distance.

```
RaycastHit hit;

Vector3 forward = transform.forward;
Quaternion startingAngle = Quaternion.AngleAxis(-rayAngleSpread / 2, transform.up);
Vector3 rayDirection = startingAngle * forward;
float angleIncrement = rayAngleSpread / numRays;

for (int i = 0; i <= numRays; i++)
{
    Debug.DrawRay(transform.position, rayDirection * detectionRange, Color.red, 0.1f);

    if (Physics.Raycast(transform.position, rayDirection, out hit, detectionRange))
    {
        // Check if the ray hit a pedestrian or car
        if (hit.collider.CompareTag("Gib") || hit.collider.CompareTag("Unit"))
        {
            Debug.Log("Obstacle detected: " + hit.collider.tag);
            // HandleObstacleDetected(hit.distance);
            Debug.DrawRay(transform.position, rayDirection * hit.distance, Color.green, 1f);
            obstacleDetected = true;
            normalizedDistanceToObstacle = hit.distance / detectionRange;

            // Updated reward calculation based on urgency
            float rewardValue = Mathf.Clamp(0.5f * (1f - normalizedDistanceToObstacle), 0.1f, 0.5f);
            SetReward(rewardValue);

            _rb.velocity = Vector3.zero;// Stage3_PPO1

            Debug.Log($"Stopping due to obstacle, reward: {rewardValue}");

            break; // Stop checking other rays once an obstacle is detected
        }
    }

    // Rotate the ray direction
    rayDirection = Quaternion.AngleAxis(angleIncrement, transform.up) * rayDirection;
}
```

Figure 5.17 : Reward based on obstacle stopping distance

3. Lane Keeping and Turn Management: When the car approaches a curve or an intersection, it is crucial for the model to adjust its steering to align with the road. The steering adjustments are calculated based on the difference between the car's current heading and the required heading to stay on the road or make the turn. This part of the reward system incentivizes the car to minimise steering errors and follow the road geometry accurately.

```
// Check if the car is on a curve and needs to adjust its steering
if (_currentRoad != null && _currentRoad.isCurve && _currentOutConnection != null)
{
    // Calculate desired alignment angle for curves based on stored angleToConnection
    float currentAngle = transform.localEulerAngles.y;
    float targetAngle = Mathf.DeltaAngle(currentAngle, angleToConnection);

    // Adjust the steering to align with the target angle
    // This might involve some form of proportional control or a direct setting based on the angle
    float steeringAdjustment = -Mathf.Clamp(targetAngle * _curveSteeringGain, -_maxSteeringAdjustment, _maxSteeringAdjustment);
    Move(horizontal, vertical, steeringAdjustment);

    // Optionally, reset angleToConnection to zero once aligned
    if (Mathf.Abs(targetAngle) < 5.0f) // Threshold for 'alignment'
    {
        angleToConnection = 0;
    }
}
else
{
    Move(horizontal, vertical);
}
```

```
if (other.CompareTag("Reward"))
{
    LaneReward rewardLaneName = other.GetComponent<LaneReward>();
    if (rewardLaneName != null)
    {
        if (rewardLaneName.laneName == laneName)
        {
            // Correct lane
            //SetReward(1f);  // Adjust reward magnitude as needed
            //Debug.Log("Correct lane! Positive reward.");

            // Calculate deviation from the lane center
            float deviation = CalculateDeviationFromLaneCenter(transform.position, rewardLaneName.transform.position);

            // Calculate reward based on deviation
            float reward = CalculateReward(deviation) *10f;

            SetReward(reward);
            Debug.Log($"Correct lane! Reward: {reward} based on deviation: {deviation}");
        }
        else
        {
            // Wrong lane
            SetReward(-2f);  // Adjust penalty magnitude as needed
            Debug.Log("Wrong lane! Negative reward.");
        }
    }
```

Figure 5.18 : Reward system for lane management

4. Traffic Rule Compliance: The car is rewarded for stopping at traffic signals and penalised for moving in non-pass zones, ensuring compliance with traffic rules.

```
if (other.CompareTag("WaitZone"))
{
    WaitZone waitZone = other.GetComponent<WaitZone>();
    if (waitZone != null)
    {
        if (!waitZone.canPass)
        {
            _rb.velocity = Vector3.zero; //Stage3_PPO1
            SetReward(-1f); // Penalize for entering a non-passable waitzone
            Debug.Log("Entered non-passable WaitZone! Negative reward.");
            // Optional: Add functionality to stop the car
        }
        else
        {
            SetReward(1f); // Reward for correctly navigating through a passable waitzone
            Debug.Log("Entered passable WaitZone! Positive reward.");
        }
    }
}
```

Figure 5.19 : Reward system for Waitzone

After setting up the reward system and collecting observations, the next step is to train it. I have used the PPO algorithm for this training, as it proved to be better as discussed in this part of section 6.1. The hyperparameters used for training is shown in Figure 5.20.

```
trainer_type: ppo
hyperparameters:
  batch_size: 120
  buffer_size: 12000
  learning_rate: 0.0003
  beta: 0.001
  epsilon: 0.2
  lambd: 0.95
  num_epoch: 3
  learning_rate_schedule: linear
network_settings:
  normalize: true
  hidden_units: 256
  num_layers: 2
  vis_encode_type: simple
reward_signals:
  extrinsic:
    gamma: 0.99
    strength: 1.0
keep_checkpoints: 5
max_steps: 300000
time_horizon: 1000
summary_freq: 12000
threaded: true
```

Figure 5.20 : Hyperparameters used for training in Traffic system

After completing the training, the neural network file was imported into the car agent to assess its ability to navigate the traffic environment. The agent demonstrated learning outcomes such as complying with traffic signals, maintaining its lane, and stopping upon detecting other cars or pedestrians. However, the driving behaviour observed did not meet the desired standards for our virtual roads. The car exhibited erratic movements, with wheels turning left and right in a seemingly frantic manner and displaying confusion. This behaviour will be further discussed in Section 6.2. The build file for this part of the simulation is linked here.

6. Discussion and Results

6.1 Results

6.1.1 Algorithm Selection of Self Driving Car

Following our implementation of section 5.2.3, in this section I will evaluate the performance of the Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) algorithms in training a self-driving car on a track. I decided to evaluate based on three statistical graphs: Environment/Cumulative, Loss/Policy and Policy/Entropy.

Environment/Cumulative:

The Environment/Cumulative rewards graph shows the mean cumulative episode reward over all agents. For analysis, we consider three points [51]:

1. **Convergence Rate:** This is how quickly the algorithm reaches a stable high reward, indicating effective learning and adaptation to the environment.
2. **Reward Stability**: After reaching a high reward level, the consistency of maintaining or improving that reward is crucial. Fluctuations might indicate instability in the learning process.
3. **Final Performance**: The maximum and average rewards achieved towards the end of training give an indication of the overall success of the training process.

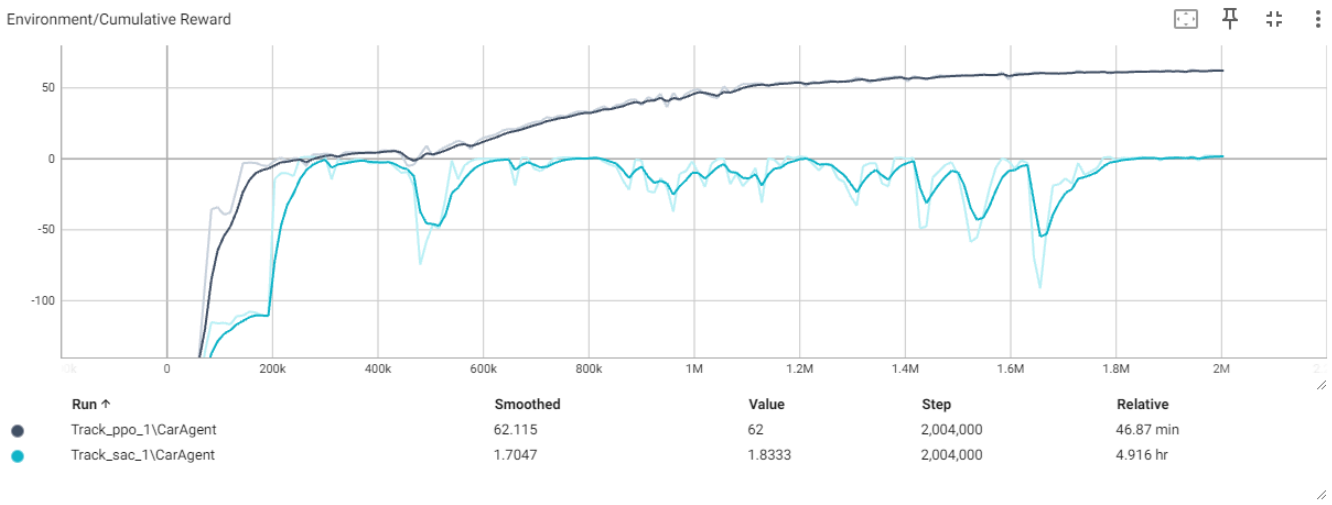


Figure 6.1 : Environment/Cumulative Graph for PPO and SAC

Evaluation Metric	PPO	SAC
Early Training	Initially shows a negative reward, indicating struggles in the early adaptation phase.	Starts with negative rewards but the values are generally lower, suggesting that the algorithm had a rougher start or that it explores more risky strategies initially.
Progress Over Time	The reward trajectory shows improvement over time, with a noticeable trend towards positive values as training progresses. By the later stages, the rewards stabilise and consistently remain in a higher range, peaking at around values of 61.	The rewards improve but with considerable fluctuations. There are several dips and rebounds, indicating potential instability in the policy or higher sensitivity to environmental dynamics. Highest reward value is 1.83.
Final Stages	Achieves consistently high rewards, suggesting that the model has learned a stable policy for navigating the track effectively.	Achieves positive rewards but they seem less consistent compared to PPO, with some values dipping back into negative territory shortly before recovering.

Loss/Policy:

This is the mean magnitude of the policy loss function. Correlates to how much the policy (process for deciding actions) is changing. For analysis, we consider three points [51]:

1. **Loss Magnitude**: Lower loss values typically indicate better policy performance, as the agent's actions align more closely with expected rewards.
2. **Loss Trend**: A decreasing trend in loss over training episodes suggests improvement in learning, whereas fluctuations or increases may indicate instability.
3. **Stability and Convergence**: The steadiness of the loss reduction or its convergence to a minimal value are indicators of training stability and the robustness of the learning algorithm.

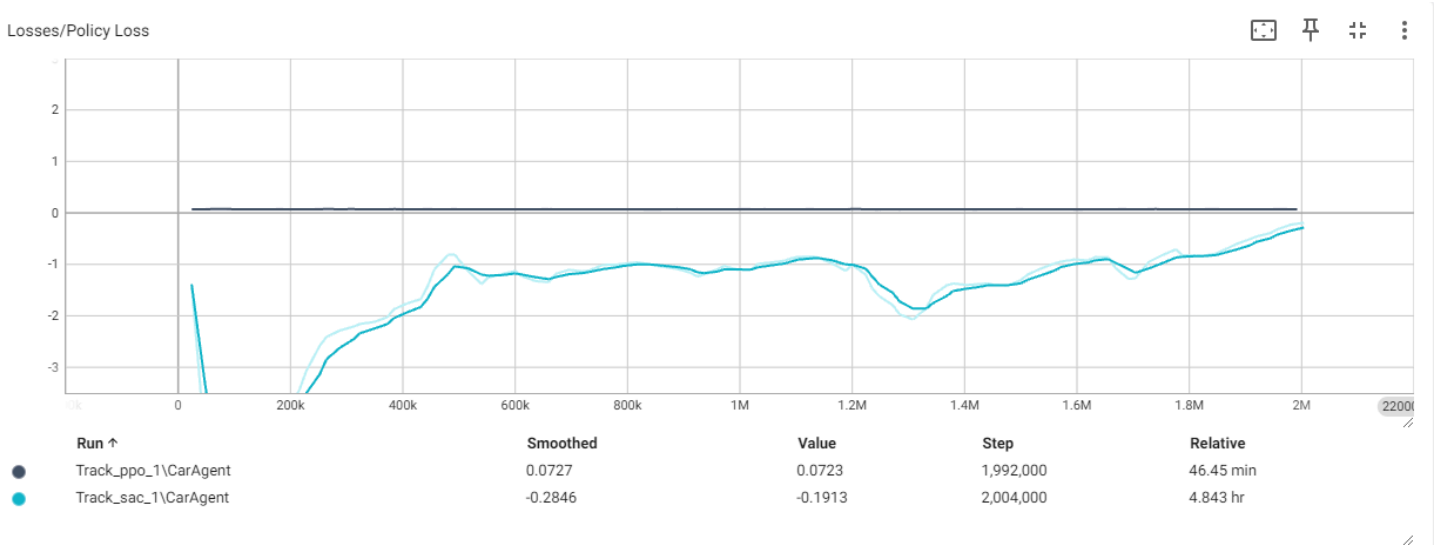


Figure 6.2 : Loss/Policy Graph for PPO and SAC

Evaluation Metric	PPO	SAC
Initial Loss Values	Starts with a policy loss value around 0.070, indicating an initial decent performance.	Begins with significantly higher loss values, around -1.39, indicating a rough start or possibly a different scale or interpretation of policy loss.
Progress Over Time	The loss generally remains stable and exhibits a gradual decrease, dipping to around 0.066, which suggests incremental improvement in the policy's decision-making capability.	Loss values show a significant decreasing trend, improving from -1.39 to about -0.19. The negative sign might imply a different loss calculation or optimization direction, which is characteristic in actor-critic methods.
Variations and Convergence	Shows minor fluctuations but maintains a narrow range, mainly between 0.066 and 0.076. This range indicates consistency and a reliable convergence of the policy loss, which is desirable for stable training outcomes.	Steady improvement but the range of the loss is quite wide from -1.39 to -0.19. Although improving, the negative values and wide range suggest higher initial uncertainty or instability in learning the optimal policy.

Policy/Entropy:

This determines the relative importance of the entropy term. This value is adjusted automatically so that the agent retains some amount of randomness during training. For analysis, we consider three points [51]:

1. **Entropy Values**: High initial values are typical, indicating initial exploration. Lower values over time suggest the algorithm is learning to exploit its environment more effectively.
2. **Trend in Entropy Over Time**: Decreasing entropy values generally reflect an algorithm's growing confidence in its learned policy, focusing more on exploitation.
3. **Stability of Entropy**: Large fluctuations in entropy can suggest varying levels of uncertainty in action choices, while stable decreasing trends indicate consistent learning.

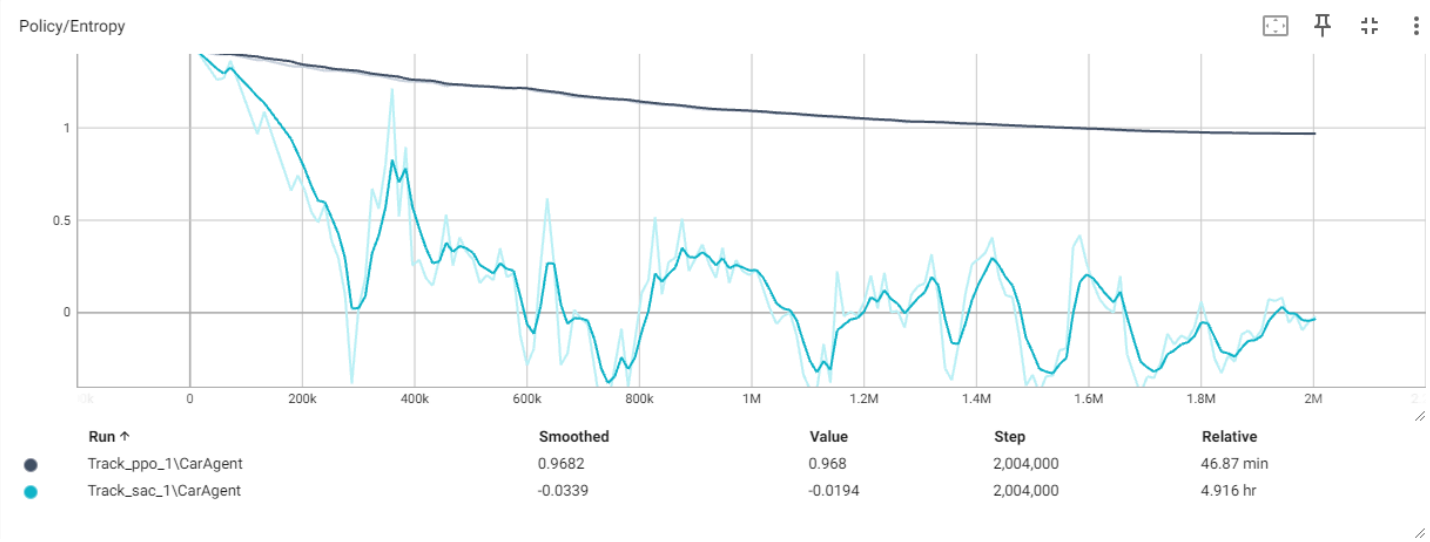


Figure 6.3 : Policy/Entropy Graph for PPO and SAC

Evaluation Metric	PPO	SAC
Initial Entropy Values	Starts with relatively high entropy, indicating a significant level of initial exploration.	Starts with high entropy. However, its initial values are slightly lower compared to PPO, suggesting a less explorative approach initially.
Progress Over Time	There is a noticeable steady decrease in entropy, suggesting a gradual shift from exploration to exploitation as the training progresses. The entropy values reach lower levels, indicating that the policy becomes more certain of the best actions to take.	Shows a more dramatic decrease in entropy, particularly noticeable in the steeper drops, which suggests periods of rapid learning. However, unlike PPO, SAC experiences several periods where the entropy increases again, indicating potential rediscoveries or corrections in the policy.
Fluctuations and Stability	The entropy reduction is fairly smooth with minor fluctuations, indicating stable learning	Exhibits more variability in entropy values, with several spikes and drops, reflecting a less stable learning process

	and consistent improvement in policy certainty.	compared to PPO. This might indicate ongoing adjustments and refinements in its policy.

Conclusion - PPO vs SAC:

The recorded observations in previous sections provide an insight into which algorithm might be better suited for practical implementations in autonomous driving scenarios.

Graph	PPO	SAC
Environment/Cumulative Rewards	Exhibited a more stable learning curve with consistent improvements in rewards and fewer fluctuations, indicative of its effective policy optimization and reliability.	Demonstrated potential to reach comparable reward levels but with greater volatility, which might introduce risks in predictable and safety-critical environments like autonomous driving.
Loss/Policy	Showed a stable and consistent policy loss, suggesting better and more reliable policy optimization, particularly beneficial in structured environments where consistent performance is crucial.	While improvements were evident, the learning was less stable, marked by wider ranges in loss values—potentially complicating deployments in environments requiring quick and reliable policy stabilisation.
Policy/Entropy	The gradual and smooth reduction in entropy with PPO points to a steady convergence to an optimal policy, advantageous in environments demanding dependable behaviour.	The more pronounced fluctuations in entropy suggest ongoing adjustments in policy, which, while beneficial for complex scenario adaptation, may delay reaching a stable policy state.

Also, PPO required significantly less training time, approximately 46.87 minutes, to complete 2000k steps. SAC, on the other hand, took about 4.916 hours to complete the same number of steps.This stark difference in training time efficiency underscores PPO's advantage in scenarios where time and computational resources are constraints.

Given the metrics analysed and the training time, PPO emerges as the more suitable algorithm for training self-driving cars on a predetermined track. Its stability in learning, reliable convergence in policy optimization, and significantly faster training times make it a safer and more efficient choice for autonomous driving applications. I also used the PPO algorithm to train the car in a traffic simulation for the next part of the training.

6.1.2 Observations of Self Driving Car in Traffic Simulation

Following our implementation of section 5.2.5, in this section I will evaluate the training performance of our self-driving car in the traffic simulation. I decided to evaluate based on three statistical graphs: Environment/Cumulative, Loss/Value and Policy/Entropy.

Figure 6.4 : Environment, Loss and Policy graph for car agent in traffic simulation using PPO

Environment/Cumulative Rewards Data: The cumulative reward is a sum of the rewards the model receives at each step [51]; in this scenario, the rewards likely represent adherence to traffic rules. Initially, the cumulative rewards start negative (indicating non-compliance or mistakes), but over time they become less negative and start showing positive values, which suggests improving compliance and decision-making by the model.

Loss and Value Graph Data: The Loss values represent the error between the predicted outcomes by the PPO model and the actual outcomes [51]. The graph shows fluctuations but a general decrease in loss over time, which suggests that the model's predictions are aligning more closely with the actual outcomes, indicating learning and adaptation. Value, in the context of reinforcement learning, generally represents the expected return from a particular state; the data shows this value fluctuating but tending to increase, which implies the model is learning to predict more valuable actions over time.

Policy Entropy Data: Entropy in this context measures the randomness of the policy's action distribution. High entropy means the policy is exploring a wide range of actions, while low entropy suggests it is exploiting known good actions [51]. The data shows a gradual decrease in entropy, suggesting that the model is moving from exploring a broad set of potential actions to exploiting a more refined set of actions it deems valuable based on its learning.

The PPO model used for the car agent in traffic simulation appears to be successfully learning and adapting over time. This is evidenced by:

- Improving cumulative rewards, which suggests better compliance with traffic rules and smarter decision-making.
- Decreasing loss values, indicating the model's predictions are becoming more accurate.
- Decreasing policy entropy, showing a transition from exploration to exploitation of better-understood actions.

The agent demonstrated learning outcomes such as complying with traffic signals, maintaining its lane, and stopping upon detecting other cars or pedestrians. Figure X shows the trained car agent navigating the traffic simulation environment.

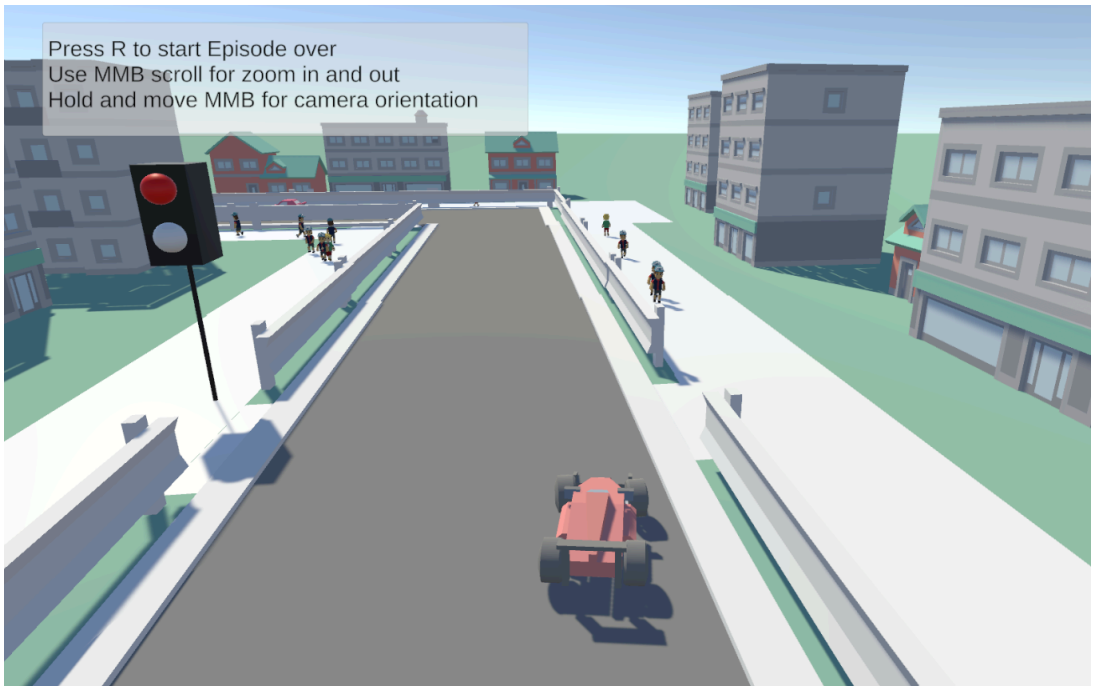


Figure 6.5 : Trained Car agent in Traffic simulation Environment

6.2 Discussion

In developing the self-driving car agent within a simulated traffic environment using Proximal Policy Optimization (PPO) algorithm, several issues were observed that impeded the desired learning outcomes and performance standards. This section explores the discrepancies between expected and actual behaviours, discusses underlying issues in both the MLCarTraffic.cs script configuration and the PPO model's training dynamics, and examines how these factors contributed to the limitations in the agent's learning and execution capabilities.

There were some limitations in MLCarTraffic.cs configuration which possibly led the agent to not understand the traffic environment fully.

Action Space and Control Dynamics

The erratic behaviour observed, particularly the frantic turning of the wheels, can be attributed to the steering action. The direct application of neural network outputs to the vehicle's steering controls without adequate smoothing or filtering have led to over-sensitive responses to slight changes in action values. This sensitivity likely caused the rapid oscillation of steering directions, preventing the agent from maintaining stable and smooth trajectories.

Sensory Configuration and Obstacle Interaction

The configuration of sensory inputs, particularly the implementation of raycasting for obstacle detection, was potentially insufficient or misaligned with the requirements of navigation tasks such as cornering or lane switching. If the rays did not adequately cover the necessary spatial range or were too aggressively tuned to react to detected obstacles, the agent might receive limited or misleading feedback about its environment.

Reward Function Design

The comprehensive reward system I designed, intended to guide the learning process of the autonomous car, did not yield the expected results. The complexity of the reward structure, aimed at capturing and differentiating between various critical driving behaviours, may have been challenging for the agent to interpret and learn effectively. Despite efforts to finely grade rewards to encourage driving manoeuvres, such as smooth cornering or precise lane switching, the agent struggled to decode and respond to these signals effectively. The multifaceted nature of the rewards could have led to an overload or confusion, where the agent found it difficult to ascertain which actions were truly beneficial, thereby affecting its learning trajectory.

There were also some model limitations when using PPO, whose results are discussed in section 6.1.2.

Exploration and Exploitation Balance

The PPO algorithm's characteristic high initial entropy intended to foster exploratory behaviours might have been maintained too long, delaying the transition to exploitation of learned behaviours. This prolonged

exploration phase could manifest as persistent erratic behaviours, as the agent continued to test a wide range of actions, some of which were counterproductive or inefficient for the task at hand. The slow decrease in entropy and the corresponding slow convergence of the policy could suggest that the agent struggled to identify and reinforce optimal strategies within the simulated traffic environment.

Model Prediction Stability

Fluctuations in the model's loss values indicated ongoing challenges in aligning the predicted outcomes with actual desirable actions. These fluctuations suggest a variability in the agent's learning process. The stability of the value function, critical for estimating the expected returns from various states, might have been insufficiently robust, leading to poor decision-making in scenarios that required advanced understanding and responses, such as navigating through traffic signals or executing safe turning manoeuvres.

7. Conclusion

7.1 Summary of our work

In this project, we embarked on developing an autonomous vehicle agent capable of navigating through a simulated traffic environment, utilising Unity's ML-Agents framework. Our implementation was structured into three main sections, detailed in the Implementation chapter. Section 5.2.3 outlined the initial development of the self-driving car, where a basic track environment was set up for the vehicle equipped with necessary sensors. This environment served as the testing ground for applying reinforcement learning algorithms, specifically Proximal Policy Optimization (**PPO**) and Soft Actor-Critic (**SAC**), which guided the vehicle's learning through dynamic reward systems.

Further, in Section 5.2.4, we expanded the scope to include a comprehensive traffic simulation system within Unity. This system featured advanced scripting for managing road segments, intersection logic, and traffic flows, emulating real-world urban traffic scenarios. This setup tested the car agent's capability to adhere to complex traffic regulations and interact safely with an array of dynamic environmental elements such as other vehicles and pedestrians.

Lastly, Section 5.2.5 integrated the autonomous vehicle with the developed traffic simulation to further enhance and test its navigation capabilities under more realistic and challenging conditions. This final integration phase was crucial for refining the car agent's decision-making processes, ensuring its ability to interpret and react appropriately to real-time changes in traffic conditions and adhere strictly to traffic laws.

Throughout these stages, the project not only demonstrated the car agent's growing proficiency in handling isolated and complex driving tasks but also underscored the potential and effectiveness of simulation-based training environments for developing sophisticated autonomous navigation systems. The successful implementation of these components lays a solid foundation for future explorations into autonomous vehicle behaviour in increasingly complex scenarios.

7.2 Future work

The project opens several avenues for future expansion to enhance the autonomous vehicle AI's understanding and interaction with its environment:

1. Imitation Learning: Introduce imitation learning to leverage pre-recorded driving data, such as recordings of a human navigating the simulated environment. This approach will help the autonomous agent learn optimal driving behaviours more efficiently by mimicking successful manoeuvres, reducing the learning curve typically associated with reinforcement learning from scratch.
2. Advanced Learning Algorithms: Exploring more advanced or hybrid learning algorithms could further enhance the AI's performance. For instance, integrating newer variants of policy gradient methods or exploring architectures that combine multiple learning strategies might yield better decision-making capabilities and faster adaptation to complex traffic scenarios.
3. Enhanced Simulation Realism: Further development can also include enhancing the traffic simulation with more realistic scenarios, such as varying weather conditions, unexpected roadblocks, or emergency driving situations, to test and improve the AI's responses under diverse conditions.

By expanding on these areas, the project can significantly push forward the capabilities of the autonomous vehicle in understanding and safely navigating complex, real-world traffic environments.

References

[1]S. Singh. Critical reasons for crashes investigated in the national motor vehicle crash causation survey. Technical report, 2015

[2] P. Bansal, K. M. Kockelman, and A. Singh. Assessing public opinions of and interest in new vehicle technologies: An austin perspective. Transportation Research Part C: Emerging Technologies, 67:1–14, 2016.

[3]Pendleton, S. D., Andersen, H., Du, X., Shen, X., Meghjani, M., Eng, Y. H., & Ang, M. H. (2017). Perception, planning, control, and coordination for autonomous vehicles. Machines, 5, 6.

[4]Shladover, S. E. (2018). Connected and automated vehicle systems. Journal of Intelligent Transportation Systems, 22, 190–200.

[5]Clemmons, L.; Jones, C.; Dunn, J.W.; Kimball, W. Magic Highway U.S.A. Available online: https://www.youtube.com/watch?v=L3funFSRAbU

[6]Thorpe, C.; Hebert, M.H.; Kanade, T.; Shafer, S.A. Vision and navigation for the Carnegie-Mellon Navlab. IEEE Trans. Pattern Anal. Mach. Intell. 1988, 10, 362–373.

[7]Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, (pp. 1097-1105).

[8]Badrinarayanan, V., Kendall, A., & Cipolla, R. (2015). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. arXiv preprint arXiv:1511.00561 .

[9]Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., & LeCun, Y. (2013). Overfeat: Integrated recognition, localization and detection using convolutional networks. arXiv preprint arXiv:1312.6229 .

[10]Kendall, A., Badrinarayanan, V., & Cipolla, R. (2015). Bayesian SegNet: Model Uncertainty in Deep Convolutional EncoderDecoder Architectures for Scene Understanding. arXiv preprint arXiv:1511.02680.

[11]Huval, B., Wang, T., Tandon, S., Kiske, J., Song, W., Pazhayampallil, J., et al. (2015). An Empirical Evaluation of Deep Learning on Highway Driving. arXiv preprint arXiv:1504.01716 .

[12]Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation , 9 (8), 1735-1780.

[13]Pinheiro, P. H., & Collobert, R. (2013). Recurrent convolutional

[14]Ondruska, P., & Posner, I. (2016). Deep tracking: Seeing beyond seeing using recurrent neural networks. arXiv preprint arXiv:1602.00991.

[15]Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. Machine learning , 3 (1), 9-44.

[16]Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., et al. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 .

[17]Hausknecht, M., & Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. arXiv preprint arXiv:1507.06527 .

[18]Xu, K., Ba, J., Kiros, R., Courville, A., Salakhutdinov, R., Zemel, R., et al. (2015). Show, attend and tell: Neural image caption generation with visual attention. arXiv preprint arXiv:1502.03044 .

[19]Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., et al. (2015). Human-level control through deep reinforcement learning. Nature , 518 (7540), 529-533.

[20] Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y., & others. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. NIPS, 99, pp. 1057-1063.

[21]Sorokin, I., Seleznev, A., Pavlov, M., Fedorov, A., & Ignateva, A. (2015). Deep Attention Recurrent Q-Network. arXiv preprint arXiv:1512.01693 .

[22]Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning , 8 (3-4), 229-256.

[23]Abbeel, P., & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. Proceedings of the twenty-first international conference on Machine learning, (p. 1)

[24]Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., et al. (2016). End to End Learning for Self-Driving Cars. arXiv preprint arXiv:1604.07316 .

[25]Asvadi, A.; Premebida, C.; Peixoto, P.; Nunes, U. 3D Lidar-based static and moving obstacle detection in driving environments: An approach based on voxels and multi-region ground planes. Robot. Auton. Syst. 2016, 83, 299–311

[26]Fisher, A. Google's Self-Driving Cars: A Quest for Acceptance. Available online: http://www.popsci.com/cars/article/2013-09/google-self-driving-car

[27]Ozguner, U.; Acarman, T.; Redmill, K. Autonomous Ground Vehicles; Artech House: Norwood, MA, USA, 2011.

[28]Buehler, M.; Iagnemma, K.; Singh, S. The DARPA Urban Challenge: Autonomous Vehicles in City Traffic; Springer: Berlin, Germany, 2009; Volume 56.

[29]Urmson, C.; Anhalt, J.; Bagnell, D.; Baker, C.; Bittner, R.; Clark, M.N.; Dolan, J.; Duggins, D.; Galatali, T.; Geyer, C.; et al. Autonomous driving in urban environments: Boss and the Urban Challenge. J. Field Robot. 2008, 25, 425–466.

[30]. Leonard, J.; How, J.; Teller, S.; Berger, M.; Campbell, S.; Fiore, G.; Fletcher, L.; Frazzoli, E.; Huang, A.; Karaman, S.; et al. A perception-driven autonomous urban vehicle. J. Field Robot. 2008, 25, 727–774.

[31]Olia, A.; Abdelgawad, H.; Abdulhai, B.; Razavi, S.N. Assessing the Potential Impacts of Connected Vehicles: Mobility, Environmental, and Safety Perspectives. J. Intell. Transp. Syst. 2016, 20, 229–243.

[32]Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles (J3016 Ground Vehicle Standard)—SAE Mobilus. Available online: https://saemobilus.sae.org/content/j3016_201806

[33]Technologies, U.*Unity's interface*, *Unity*. Available at: https://docs.unity3d.com/Manual/UsingTheEditor.html.

[34]Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. arXiv preprint arXiv:1801.01290.

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

[36]Ho, J. and Ermon, S. (2016). Generative adversarial imitation learning. In Advances in neural information processing systems, pages 4565–4573.

[37]Hussein, A., Gaber, M. M., Elyan, E., and Jayne, C. (2017). Imitation learning: A survey of learning methods. ACM Computing Surveys (CSUR), 50(2):21.

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

[39]Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In International Conference on Machine Learning (ICML), volume 2017.

[40]Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8):1735–1780.

[41]Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In Proceedings of the 26th annual international conference on machine learning, pages 41–48. ACM.

[42]Kalra, N., & Paddock, S. M. (2016). Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? Transportation Research Part A: Policy and Practice, 94, 182–193.

[43]Parkinson, S., Ward, P., Wilson, K., & Miller, J. (2017). Cyber threats facing autonomous and connected vehicles: Future challenges. IEEE Transactions on Intelligent Transportation Systems, 18(11), 2898– 2915.

[44]Goodall, N. J. (2014a). Ethical decision making during automated vehicle crashes. Transportation Research Record, 2424(1), 58–65.

[45]Matthias, A. (2004). The responsibility gap: Ascribing responsibility for the actions of learning automata. Ethics and Information Technology, 6(3), 175–183.

[46]Rana, Kritika & Gupta, Gaurav & Vaidya, Pankaj & Khari, Manju. (2023). The perception systems used in fully automated vehicles: a comparative analysis. Multimedia Tools and Applications. 1-23. 10.1007/s11042-023-15090-w.

[47]Sutton, R. S. and Barto, A. G. (1998). Reinforcement Learning: An Introduction. MIT Press.

[48]Johnson, M., Hofmann, K., Hutton, T., and Bignell, D. (2016). The malmo platform for artificial intelligence experimentation. In IJCAI, pages 4246–4247.

[49]Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, Amir Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., and Petersen, S. (2016). Deepmind lab. arXiv preprint arXiv:1612.03801

[50]Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. (2016). Vizdoom: A doom-based AI research platform for visual reinforcement learning. In Computational Intelligence and Games (CIG), 2016 IEEE Conference on, pages 1–8. IEEE.

[51]Technologies, U. *Using tensorboard to observe training*, *Using TensorBoard to Observe Training - Unity ML-Agents Toolkit*. Available at: https://unity-technologies.github.io/ml-agents/Using-Tensorboard/

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

[53]Day, Min-Yuh & Yang, Ching-Ying & Ni, Yensen. (2023). Portfolio dynamic trading strategies using deep reinforcement learning. Soft Computing. 10.1007/s00500-023-08973-5.

[54]Yosider. (n.d.). *ml-agents-1/docs/Training-PPO.md at master · yosider/ml-agents-1*. GitHub. https://github.com/yosider/ml-agents-1/blob/master/docs/Training-PPO.md

[55]Gupta, R. (2018, December 14). *Soft Actor Critic—Deep Reinforcement Learning with Real-World Robots*. The Berkeley Artificial Intelligence Research Blog. https://bair.berkeley.edu/blog/2018/12/14/sac/

[56]*6.6 Actor-Critic methods*. (n.d.). http://incompleteideas.net/book/first/ebook/node66.html

[57]Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1801.01290

[58]*Soft Actor-Critic — Spinning Up documentation*. (n.d.-b). https://spinningup.openai.com/en/latest/algorithms/sac.html?highlight=SAC

[59]Yosider. (n.d.-b). *ml-agents-1/docs/Training-SAC.md at master · yosider/ml-agents-1*. GitHub. https://github.com/yosider/ml-agents-1/blob/master/docs/Training-SAC.md

[60]Jin, Z., Swedish, T., & Raskar, R. (2018, October 30). *3D traffic simulation for autonomous vehicles in Unity and Python*. arXiv.org. https://arxiv.org/abs/1810.12552

[61]Mahmoudi, Reza, and Armantas Ostreika. "Reinforcement learning for obstacle avoidance application in unity ml-agents"

[62]Kuzmic, Jurij & Rudolph, Günter. (2020). Unity 3D Simulator of Autonomous Motorway Traffic Applied to Emergency Corridor Building. 197-204. 10.5220/0009349601970204.

[63]Savid, Y.; Mahmoudi, R.; Maskeliūnas, R.; Damaševičius, R. Simulated Autonomous Driving Using Reinforcement Learning: A Comparative Study on Unity's ML-Agents Framework. Information 2023, 14, 290. https://doi.org/10.3390/info14050290

[64]*Deep Reinforcement Learning framework for Autonomous Driving*. (n.d.). Ar5iv. https://ar5iv.labs.arxiv.org/html/1704.02532