School of Computer Science and Electronic Engineering

University of Essex

Thesis submitted as part of CE901 MSc Project and Dissertation

Using Machine Learning Techniques to Create Neural Network Driving Agents, with a Focus on Reinforcement Learning

Name: Drew A. J. Berry

Supervisor: Dr. Michael Fairbank

Submitted on 26th August 2024

Abstract

The Open Racing Car Simulator (TORCS) is a car racing simulation game, which allows a player to race against opponents that are simulated by the computer. In this project, several AI opponents for this game have been developed, using various forms of machine learning on neural networks, which can be observed or raced against. The differences and advantages of genetic and reinforcement learning algorithms have been explored in this racing game context. Each parameter and condition of these algorithms were experimented upon diligently, to create AI agents that could complete laps of a race efficiently and rival human players.

A racing simulation environment offers a unique set of challenges that are comparable to those faced by real-world autonomous vehicles, such as high-speed decision-making and spatial awareness. The exploration of this idea could uncover insights into the use of machine learning in such contexts.

This thesis illustrates all experiments and findings, demonstrating the competency of trained agents, while also explaining the technical aspects of how research was conducted. Both algorithms have created simple and complex racing behaviours, such as following the track axis and/or racing line. However, a final model trained using the genetic algorithm demonstrated the best performance.

Key Terms

Reinforcement learning, TORCS, Artificial Intelligence, PPO, Proximal Policy Optimization, GA, genetic algorithm

Acknowledgements

I would like to thank the following people who made this research project possible:

Bernhard Wymann and the other developers of TORCS, who created the simulation environment on which my research was conducted.

The developer of Gym-TORCS, Naoto Yoshida, for his reinforcement learning environment which was the base for my implementation.

My supervisor, Dr. Michael Fairbank, for his guidance and criticism of my work.

My family and close friends, for their support and encouragement during the four months of this project.

Table of Contents

1. Introduction

This project is centred on the research into the field of autonomous systems. “Autonomous [systems] … refers to artificial intelligence systems capable of performing tasks without human intervention” [d]. This field is interesting because it has the potential to enhance human lives and reduce human error. This extends to robot or self-driving cars. Advancements in this example are important to the safety of passengers and pedestrians. Reducing the likelihood of accidents requires extensive research into how the system can detect hazards and the approach to minimise the risk of a resolution. Once this challenge is overcome, these vehicles could greatly improve human punctuality and comfort. For example, using traffic reports to avoid busy areas, reducing travel time and road congestion.

Autonomous vehicles should demonstrate a few behaviours to perform reliably and efficiently. First, they should show spatial awareness to avoid obstacles and consider the laws of the environment, requiring complex sensors. Second, they should be capable of making complex decisions in high-speed scenarios, which contributes to risk minimisation. Finally, they should understand the signs of a hazard or future event to prepare for or predict the best course of action. These behaviours must also be considered for an AI racing agent so that it can drive correctly and rival opponents.

In machine learning, the exploration of different approaches is required to find an optimal learning method for the given context. Certain sub-categories or algorithms are only suited for certain problems. For example, supervised and unsupervised learning are used for classification and regression, whereas reinforcement learning is used to optimise a sequence of tasks. Finding a new optimal approach can lead to major advancements in the given field. This lends itself to this project as two different learning methods were experimented with to determine which was best for the problem.

2. Project Context

This section details the entire project context to provide an understanding before any research is explained. The game TORCS is the subject of the initial subsection, describing its mechanics and background. Following this, the machine learning subsection briefly explains the concept as a whole, while giving an in-depth description of each algorithm chosen for this project.

Section 2.1 is adapted and expanded from the project proposal submitted as a part of CE902 [1].

2.1. TORCS

Developed as an open-source project in C++, TORCS offers a unique physics-based experience with a multitude of features, having 42 different cars to choose from and 30 tracks to race them on. Lighting and particle effects, such as smoke and glowing brake discs, are some of the graphical features available that make this a life-like simulation. The mechanical features of TORCS also contribute to replicating real racing, where the cars have most of the properties and workings of a real car, with their own fuel tanks and the capability to sustain damage. The game demonstrates realistic interactions between cars and the world, such as collisions and aerodynamics. All of this provides an excellent simulation of racing for research or play, although now it looks slightly outdated based on today's standards in gaming. [2][3]

Figure 1. Shows the logo of TORCS, which is shown upon running the game

TORCS allows the player to race against friends in split screen, or against up to 39 computer-driven opponents. There are many different opponents to race against provided by the developers, and you can even make your own AI opponent, which is fundamental to this project and will be discussed in better detail later in Section 3.1. The game allows for multiple input devices, depending on a user's computer's peripheral support. These could be a steering wheel, for a more authentic experience, or a keyboard, amplifying the difficulty of correctly driving. [2][3]

TORCS provides a few game modes for the user to race in, the ones of note being *Championship* and *Practice*. *Championship* tasks the user to race in multiple races and keeps track of their performance with a leaderboard, whereas *Practice* acts as a time trial, where the user can test how fast they can drive a track. *Practice* is the game mode that will be used to train any agents in this project, and *Championship* will be a good demonstration of an agent's overall performance and robustness.

2.1.1. Graphics

TORCS emulates the real world by having some real racetracks for players and the computer to race on, although some tracks are made by the game's developers or users, via a track editor that is also available. The game also has 42 different car models to drive, based on real-world automobiles.

Figure 2. Shows an example of a track in the game and the different cars that can be driven [2]

The game has many graphical features to make the simulation realistic. Some of these are subtle and take effect on the car model: lighting and glowing brake disc effects. The others show up in the environment, being smoke or skid marks.

Figure 3. Shows the smoke and skid marks effects, from a video on the SourceForge page [2]

2.1.2. Mechanics

Before discussing the technical components and data structures of TORCS, the following are just a handful of features that this simulation brings to life.

Each of the 42 cars available is unique, having its own turning circle and acceleration rate. A car's wheels have all the properties of a real wheel, whether that be the suspension or the correct friction with the ground. Cars are influenced by aerodynamics, have spoilers to increase downforce, and can collide with barriers and other cars. Collisions cause damage to the player's car, and any other cars involved. Also, when braking too hard, a car's wheels lock up and steering becomes more difficult. [3]

TORCS has different types of tracks that have differing terrain. Road tracks have rough terrain outside of the track edges. If the car strays from the track, this causes the loss of traction, resulting in the car slowing down and being harder to control. Dirt tracks are entirely rough terrain but can be made easier with a different car choice. There are also oval tracks, which are the easiest, with no rough terrain, and can be used to simulate racing similar to NASCAR.

In the competition version of TORCS, detailed in this paper [4], there is a physical separation between the game engine and AI agents. An agent's understanding of the world and its available actions are defined by sensors and actuators, of which there are 19 and 7 respectively.

2.1.2.1. Sensors

The sensors act from the perspective of the car, not the world. This means that the x-axis always acts in the forward direction, while the y-axis acts sideways and the z-axis acts vertically. The tables below detail all 19 sensors, which were adapted from the competition paper [4] and include observations from experiments. Any sensor with an infinite range is defined as having no limit, but the world and physics engine only allow these values to reach a specific limit. Where this is unclear, the rough limits are explained.

Name	Range (unit)	Description
angle	$[-\pi, \pi]$ (rad)	The angle between the car's direction and the direction of the track axis. This sensor returns 0 when the car is facing in the direction of the track axis, $\pm \frac{\pi}{2}$ when the car is facing perpendicular to the track axis, and $\pm \pi$ when the car is facing the opposite direction.
trackPos	$[-\infty, \infty]$ (m)	The distance between the car and the track axis. A negative value represents being on the right side of the track axis. At the track edge, this distance is almost always ±1. The distance at the track edge can only be larger than ±1 when a pit lane is included in the track layout.
focus	[0, 200] (m)	A vector of 5 range finder lasers, which each return the distance to a track edge from the car up to 200 metres. These act in a 180-degree area in the direction that the car is currently facing, such that it cannot sense anything in the other direction. However, they do not occupy the whole 180-degree area during a single timestep. This works by sensing in a 1-degree area, then skipping 5 degrees and sensing again in a 1-degree area, repeating until 5 values are returned.

track	[0, 200] (m)	A vector of 19 range finder lasers, which each return the distance to a track edge from the car up to 200 metres. By default, these act in a 180-degree area in front of the car, each sensor equidistant from each other. However, the laser angles can be defined by the client when setting up a connection to the server, where each angle must be in degrees.

Table 1. Describes the sensors for locating the car in the environment

Name	Range (unit)	Description
distFromStart	[0, ∞] (m)	The distance driven from the start line in the current lap. This sensor has a maximum limit equal to the track length, before it is reset for the next lap.
distRaced	[0, ∞] (m)	The distance driven since the start of the race. This sensor is limited by the lap limit, with a maximum equal to the total laps multiplied by the track length.
curLapTime	[0, ∞] (s)	The time elapsed in the current lap.
lastLapTime	[0, ∞] (s)	The time elapsed in the previous lap.
opponents	[0, 200] (m)	A vector of 36 range finder lasers, each of which covers a 10-degree range around the car, scanning for the closest opponent in a 200-metre radius and returning the distance. This does not consider track edges.
racePos	{1,2,...,N}	The position that the car is in the race. N here is the last place in the race which can be up to 40.

Table 2. Describes the sensors for describing the bot's position in a race

Name	Range (units)	Description
speedX	[-∞, ∞] (km/h)	The speed at which the car is travelling on the x-axis. A positive value here denotes moving forward. This sensor never exceeds ±300.

speedY	[-∞, ∞] (km/h)	The speed at which the car is travelling on the y-axis. Values here denote moving sideways. This sensor usually has low values, never exceeding ±80.
speedZ	[-∞, ∞] (km/h)	The speed at which the car is travelling on the z-axis. A positive value here denotes moving upward. This sensor only returns a non-zero value while the car is off the ground, so is barely useful. It can be assumed that the value also never exceeds ±80.
rpm	[0, ∞] (rpm)	The number of rotations per minute of the engine. This sensor never exceeds 10,000.
gear	{-1,0,1,2,...,6}	The current gear that the car is in, where -1 is reverse, 0 is neutral, and 1 to 6 for the other gears.
fuel	[0, 100] (litres)	The current fuel level, where 0 is no fuel left and 100 is the maximum fuel. However, from my observations, *fuel* usually starts around 96. This could depend on the car chosen.
damage	[0, 100]	The current amount of damage sustained from collisions, where 0 is no damage and 100 is the maximum damage.
wheelSpinVel	[0, ∞] (rad/s)	A vector of 4 values, each being the rotation speed of a wheel.
z	[-∞, ∞] (m)	The distance from the car's centre of mass to the surface of the track along the z-axis. A negative value would mean that the car is below the track/world. When not in the air, this value is 0.2.

Table 3. Describes the sensors for describing the current conditions of the car

2.1.2.2. Actuators

The following figure describes all 7 actuators, which have also been adapted from the competition paper [4].

Name	Range	Description
accel	[0, 1]	A virtual accelerator pedal (0 meaning no accelerator, 1 full accelerator).

brake	[0, 1]	A virtual brake pedal (0 meaning no brake, 1 full brake).
clutch	[0, 1]	A virtual clutch pedal (0 meaning no clutch, 1 full clutch).
gear	{-1,0,1,2,…,6}	Like the sensor of the same name, with the same values, and is used to change the gear. Note that this can be done without the use of the clutch, making the clutch useless.
steer	[-1, 1]	Denotes the current steering angle of the car, where -1 is fully right and 1 is fully left.
focus	[-90, 90]	The direction that the *focus* sensor should face, allowing the focus sensor to change its angle during the race.
meta	{0, 1}	This holds a command for the server, where 0 is do nothing, and 1 is restart the race. This will return the game to its waiting-for-socket-connection state and a connection must be reconnected. These can be also defined as True or False.

Table 4. Describes the actuators that implement an agent's decided action

Mechanics continued

The competition paper [4] describes the interaction with the server best, "Every game tic (roughly corresponding to 20ms of simulated time), the server sends the current sensory inputs to each bot and then it waits for 10ms (of real time) to receive an action from the bot. If no action arrives, the simulation continues and the last performed action is used." This means that there are roughly 50 game ticks per second, which was helpful to know during my research. Also note that game ticks are referred to as timesteps in this thesis as they are easier to understand whilst working with OpenAI's Gym and StableBaselines3's reinforcement learning algorithms.

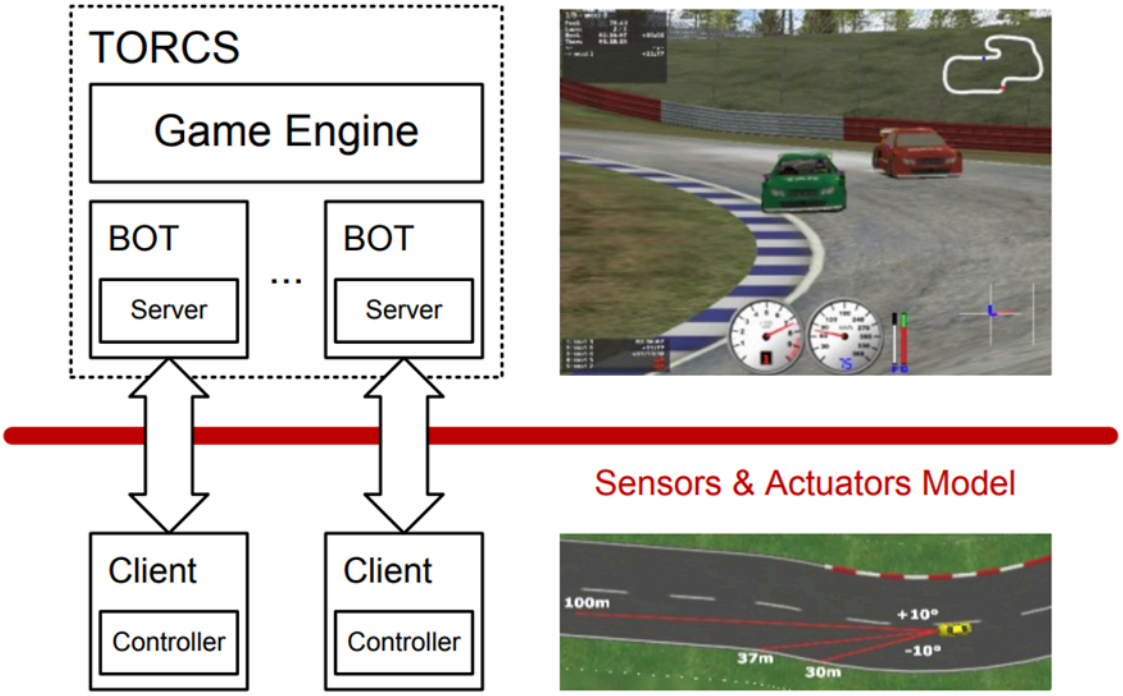


Figure 4. Shows how the bots interact with the server [4]

TORCS also allows the enabling of noisy sensors. This limits the range of the range finder lasers and creates noise in them to simulate real-world sensors. This is useful when trying to translate any AI in TORCS to real-world “robot” cars. [4]

2.1.3. History

TORCS started as a 2D racing game in 1997, by developers Eric Espie and Christophe Guionneau, called Racing Car Simulator (RCS). Later it became Open Racing Car Simulator (ORCS) when it transitioned to 3D. This version of the game was somewhat basic, cars didn’t even have engines! When engines were added to the game, it finally became The Open Racing Car Simulator (TORCS) and was released in 2000/2001. Development is currently headed by Bernhard Wymann, who has written a paper [5] on how AI “bots” can be added to the game and how to develop them correctly. [2][3]

Since the release, TORCS has had plenty of updates, adding many more features, such as multiple camera angles and texture mapping that gave cars more detail, and even allowing support for users to build their own AI opponents. However, the game hasn’t been updated since 2016. [2]

Due to TORCS not having network-based multiplayer, to compete against others, users were tasked with creating their own AI “bots” which would set completion times on tracks and other users would try to beat these times. This led to the Simulated Car Racing Championships, which were held between 2013 and 2017. This competition made users race their AI “bots” against each other’s creations in real time, using a patched version of the game that used a server that could be connected to via a client. All participants made AI agents in C++ or Java. [4]

TORCS, being an open-source project, has been used in the past for plenty of research into how AI agents can play a racing game, and how this translates to robot cars in the real world. The game has also been adapted into a few other projects, such as Speed Dreams [6].

TORCS has amassed a large following of players, reaching over 3 million downloads as of 2024 [2], and was even featured in a 2007 article by Linux Journal [7] that referred to the game as “the best open-source driving simulation”.

2.2. Machine Learning

“Machine learning [regards] the development … of computer systems that [are capable of learning and adapting] without following explicit instructions” [8], unlike traditional programming. These systems automatically improve through experience, finding patterns and relationships in data. This makes machine learning an effective tool for solving complex problems in various fields, such as autonomous systems, which relates to this project. [8][9]

Neural networks are an essential part of many advanced machine learning systems. Inspired by the structure of the human brain, neural networks consist of layers of interconnected nodes, named neurons. Inputs are passed through these layers, transforming the data into an output, which could be a single value, a set of values, or even a classification label. Neural networks learn by adjusting the weights of connections between neurons through various means, most notably backpropagation. [10][11]

This project aimed to develop neural network agents that have been trained using genetic and reinforcement learning algorithms. The following subsections define the distinction between both algorithms to explain why they were chosen for the research conducted.

Sections 2.2.1 and 2.2.2 are adapted and expanded from the project proposal submitted as a part of CE902 [1].

2.2.1. Genetic Algorithms

Genetic Algorithms are an optimization technique inspired by natural selection and genetics. They mimic the process of evolution to improve a population, this includes mutations and “breeding” to make offspring. Each individual in a population is a solution to a problem, and its chromosome determine its characteristics. These solutions are graded by a fitness function, which returns a value that represents how “healthy” a chromosome is. Certain individuals are chosen, based on their fitness, to create offspring via a crossover of their chromosomes. Once the offspring are created, they undergo random mutations to add subtle changes to their chromosome and are added back to the population to progress to the next generation. [12][13]

Usually, genetic algorithms are used to find the optimal solution to a problem. For example, the chromosome could represent the actions taken or the items that should be used for the problem, like in the knapsack problem [14]. However, for this project, the genetic algorithm was used to search for the optimal weights and biases of a neural network, such that each member in the population has a chromosome consisting of each weight and bias in the network. The algorithm was programmed from scratch and had to consist of components that were appropriate for the chromosomes, including

mutation that had to apply an appropriate amount of random noise to create small changes. The GA was chosen due to its simplicity and was initially used to generate a good baseline for an agent's performance. The experiments conducted with this algorithm are detailed in Section 4.4, while the implementation is detailed in Section 5.4.

2.2.2. Reinforcement Learning

Reinforcement learning (RL) algorithms are a set of machine learning techniques, where an AI-driven system, often called an agent, learns via a trial-and-error approach.

At any given timestep, an agent has a state. Every state has its own set of actions, collectively called the action space, that can be performed to move the agent into another state. Each following state has a reward tied to it, which explains how good or bad it is for the agent to move to said state. The policy chooses which actions to enact from any given state. An agent then utilises feedback from its actions, indicated through punishment or reward defined by the reward function, to optimise the policy. So, an RL algorithm determines the optimal policy for an agent, allowing it to select actions that maximise a cumulative reward. In the context of TORCS, the optimal policy would allow the agent to drive around the track at high speeds to finish laps in the fastest time. [15][16]

The policy samples from a probability distribution to decide the actions taken. When considering the feedback from its actions, the policy is updated to give a larger probability to the actions which give good rewards. This means that poor actions are less likely to be chosen in the future, eventually being filtered out, so that the agent learns to solve the problem effectively. [17][18]

Agents must also learn the trade-off between immediate and long-term rewards, as some rewards may be delayed, and actions can have long-term consequences. A preference towards either of these can be specified with a discount factor, where a value closer to 0 prioritises immediate rewards, and a value closer to 1 does the opposite. [15][19]

State/action spaces can be either discrete or continuous. Discrete refers to having distinct states/actions, for example, chess has specific board states and moves. Whereas, continuous refers to states/actions that can fall anywhere in a set interval. In TORCS, these spaces are continuous.

2.2.2.1. Policy Gradient Theorem

A sub-category of reinforcement learning concerns policy gradients. Policy gradient methods aim to model and optimise the policy directly, rather than learning a value function before deriving the policy from it, as seen in Q-Learning. This uses the gradient of expected rewards to iteratively improve the policy. This is considered “on-policy” since the existing policy generates the data necessary to update itself. Also unlike Q-Learning, policy gradient methods do not utilise a replay buffer to store past experiences, agents learn directly from what they experience in the environment. A batch of experience (states, actions, rewards, etc.) is discarded after each gradient update, making these methods typically less sample-efficient than Q-Learning. [20][21][22][23]

Policy gradient methods require two neural networks to operate. The first is the policy network, which outputs the policy. This gives a probability for each action that can be taken from a given state, directly dictating the behaviour of the agent. The second is the value network, which estimates the value of state-action pairs to evaluate the quality of actions performed by the policy network, which

the policy network uses to refine the policy. The output of the value network is the expected cumulative reward from the given state. [21][22][24][25]

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

This is the equation for the policy gradient loss, which is used to update the policy network. π refers to the output of the policy network, the probabilities of actions. A is the advantage, the estimate of the relative value of the selected action. This is comprised of the discounted sum of rewards minus a baseline estimate, which is the value from the value network. A negative advantage creates a negative gradient where the likelihood of the selected action is decreased, and vice versa for a positive advantage. [20][26]

Examples of policy gradient methods include:

- REINFORCE
- Trust Region Policy Optimization (TRPO) [27]
- Proximal Policy Optimization (PPO) [26]
- Deep Deterministic Policy Gradient (DDPG)

2.2.2.2. Proximal Policy Optimization

PPO, designed by OpenAI, aimed to balance an easy implementation, sample efficiency and ease of tuning which had been missing from TRPO. These algorithms improve training stability by avoiding parameter updates that change the policy too much at one step. TRPO aims to optimise the policy by maximising a surrogate objective function while constraining the policy update to stay within a "trust region". PPO simplifies this by using a clipped surrogate objective instead of the complex constraint while still retaining similar performance. [20][26][27][28][29]

For the formula of PPO, they [26] define the probability ratio between old and new policies as:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

This then creates the following formula for policy gradient loss:

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

The first term in the min( ) operator is the normal policy gradient objective. The second term is the clipped version of the objective, where the clip( ) function constrains the ratio to within the defined range, typically using an ε of 0.2. The min( ) operator then reduces the policy gradient loss to the smaller of the two, such that policy updates aren't too large and change the policy too much. [20][26][29]

So, PPO was chosen for this project due to its constrained policy updates. In a racing game context, major changes in the policy could cause an agent trouble when in high-speed scenarios, and with such a complex state space, policy updates should be smaller. The PPO algorithm wasn't made from scratch, the Stable Baselines3 library [30] was used for the implementation. This was to save time for experiments and to remove any human error in the logic of the algorithm.

Machine Learning continued

PPO is significantly different from a genetic algorithm, as the models learn from interactions in the environment, rather than complete randomness. The contrast between learning through experience and changing from a final fitness is used to compare the performances of agents trained by these algorithms.

3. Preliminary Work

This section briefly details the work done in preparation for this project and acts as the starting point of research. This includes the initial setup of the game and client.

Section 3 is adapted and expanded from the project proposal submitted as a part of CE902 [1].

3.1. Background Research

A presentation [31] on TORCS and its research capabilities served as the catalyst for this project. The speaker discussed the game engine and how Python “bots” work with the game. This also gave some examples of how to develop specific racing behaviours. However, they focused on preprogrammed intelligence rather than machine learning.

A SourceForge page [2] holds all the source code, documentation, and forums for TORCS. Each version of the game can be downloaded from this site, and there is plenty of debugging help on the forums.

The paper [4] written on the Simulated Car Racing Championship and the competition version of TORCS details plenty of the game’s inner workings while explaining how user-created “bots” can be tested. For the “bots” to communicate with the game, TORCS must be patched with the SCR server patch and users must connect to a socket via a client. The client receives the sensor data from this socket and returns the decided action to the socket to drive the car. This patch can be found on the SourceForge page [2] with the game.

3.1.1. Related Works

Plenty of research has been conducted into using machine learning in racing games, most commonly using reinforcement learning. TORCS has served as a platform for a reasonable amount of said research. One research paper [32], written by Ben Lau, demonstrates DDPG to create agents for TORCS. They trained a neural network to drive one of the harder racetracks in TORCS and were able to keep the car from spinning out for 3 minutes. In addition, Mohammed Abdou Tolba used both discrete and continuous action reinforcement learning algorithms to teach neural networks in TORCS, this is described in their thesis [33]. Alternatively, Muhammad Salman wrote their MSc thesis [34] on preprogrammed agents in TORCS, creating 9 different behaviours that would drive a car. Another researcher [35] created racing agents from pixels alone using imitation learning and a CNN.

3.2. Background Work

Due to the instructions of the competition paper, all research was conducted using TORCS version 1.3.4 and the SCR server patch version 2.0.

An initial investigation of the game engine was conducted to develop an understanding of a car’s interaction with the environment, providing context to how research should be conducted. This included identifying racetracks suitable for training. An optimal racetrack would allow the learning of complex behaviours while avoiding the potential for exploitation by agents.

Subsequently, a client was implemented to connect to the game server and run agents. The Gym-TORCS GitHub repository [36] provided examples of both a client, named snakeoil, and a custom reinforcement learning environment, named gym_torcs, developed with OpenAI's Gym. The SnakeOil client was originally made by Chris X Edwards [37]. Gym-TORCS states that the automation of training sessions without graphics is OS-specific. However, the process used in this project to run the game without graphics was unaffected by this (see Section 5.1.2 for additional information).

The snakeoil client included an example agent that accelerated up to a defined top speed and tried to follow the track axis as closely as possible, all through simple calculations using the sensor data. Due to the complex state and action space of TORCS, the agent eventually fails and is unable to navigate any difficult corners, drifting off the track and into barriers.

The SCR server patch adds a screen for selecting the drivers/agents that are included in a race. This screen includes the choice of a player-controlled car, developer-created agents or any client agents connected on certain sockets. These agents are named scr_server and a number between 1 and 10. This allows the player to race against any number of AI opponents, including those created by the player themselves. When training an agent, the correct socket must be selected here and connected to by the client.

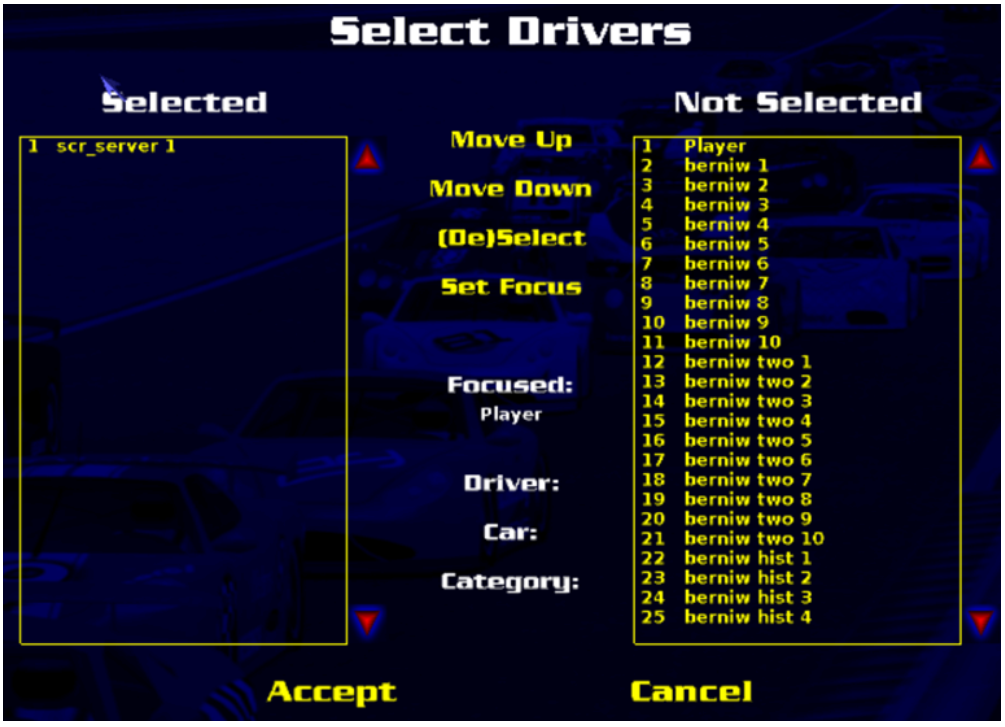


Figure 5. Shows the agent selection screen

3.3. Repositories

All the work completed during this project, including all code iterations, graphs, notes made, and videos, can be found in the GitLab CE901 repository. The links for these can be found in Appendix 1.

4. Research

This section details each experiment conducted in TORCS with their respective results and observations. The initial experiments focused on the game engine, which provided the understanding needed before proceeding to research both algorithms. Any technical information or implementations, for further context, can be found in Section 5.

4.1. Desirable Racing Behaviours

The goal in racing is to finish laps faster than opponents, but how should an agent drive to achieve this? Agents should accelerate on straight sections of the track and slow down for corners, utilising the brake effectively. In terms of steering, following the invisible line along the centre of the track (also known as the track axis) is an effective approach to keep within the track borders. However, to maximise speed when cornering, an agent should deviate from the track axis and follow the “racing line”. This means that while approaching a corner, the agent should move slightly outwards before turning into it, resulting in a better steering angle and retaining more speed. As a result, the agent will stay close to the inside of most corners, reducing the total lap distance.

4.2. TORCS

The following observations about the game were critical to know to avoid any crashing issues when training agents:

- The game-client connection allows for 50 timesteps a second of simulation time. If the client cannot keep up with this, the game-client connection crashes as the server doesn’t receive any actions when they are needed.
- When the car takes maximum damage, the race ends and the game sends a shutdown message to the client.

4.3. Data analysis

Before training any agents, the data of TORCS was investigated to decide which sensors and actuators should be used for the inputs and outputs of models. This section details each experiment conducted to gain this understanding. For these experiments, only the example agent provided by snakeoil was used.

4.3.1. Sensor/Actuator Data

Initially, a tool was developed to display sensor and actuator data, which can be monitored while an agent drives around the track. This is shown below. For the implementation details of this tool, see Section 5.2.1.

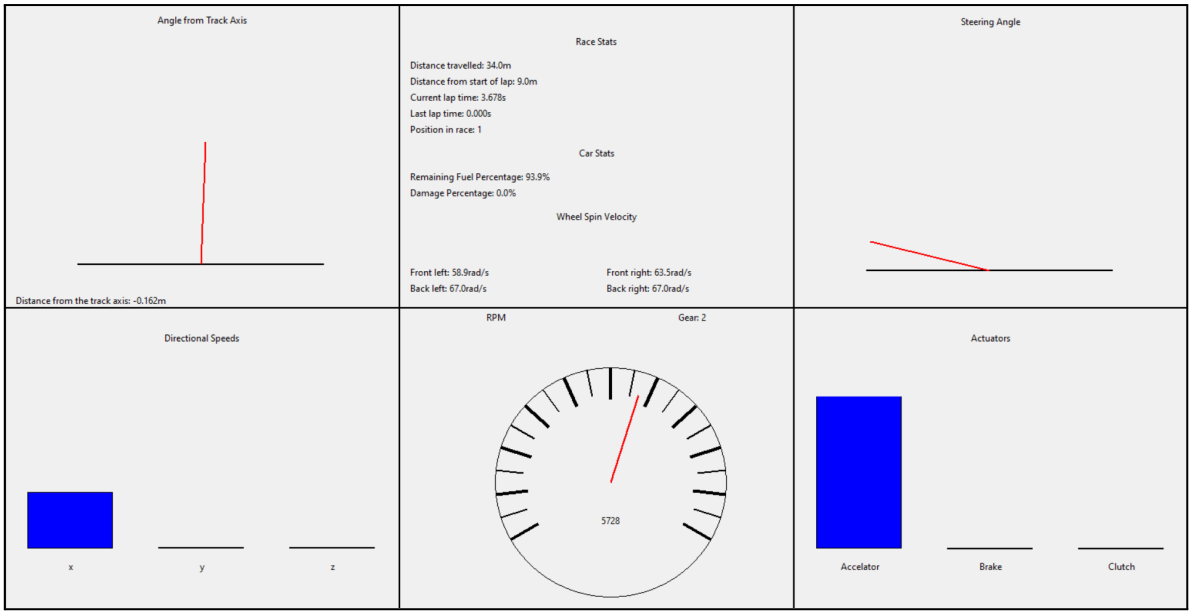


Figure 6. Shows an example of the tool

Any missing sensors/actuators here either couldn't be visualised or weren't important to track. Also, note that the angle from the track axis has been divided by 2 to fit on a 180-degree diagram, where the actual scale should be 360 degrees.

This experiment contributed to a deeper understanding of how the client communicates and shares data with the server, while also showing some important details/quirks of the sensors and actuators, the findings of which are explained below and in the tables in Sections 2.1.2.

Directional speeds are quite important to any agent, but only two of the three here have a use at all. *speedX* refers to the car's longitudinal velocity. This is usually the highest of the three speeds and can be used to change gears or understand the progress being made by an agent. *speedY* refers to the velocity perpendicular to the car's forward direction. This value increases when heavily steering at high speeds, indicating that the car is drifting or moving sideways, which does not occur very often. However, *speedZ* refers to the car's vertical velocity. First of all, TORCS's road tracks lack jumps, which are reserved for dirt tracks, hence the value is nearly always zero. Second, this speed would never be effective in developing any racing agents. If the value is greater than 0, the car is in the air, and the agent has no control over anything until it lands, at which point the value returns to zero.

When outside of the track, both the *track* and *focus* sensors become incredibly unreliable, returning values of -1 in most cases. This can ruin an agent's performance or allow for exploits, so allowing the car to go outside the track should be avoided.

When *angle* is 0, the car is facing in the exact direction of the track axis. When *trackPos* is 0, the car is directly on the track axis. Both *angle* and *trackPos* can be negative. When *angle* is larger than $\pm \pi/2$, the car is facing in the opposite direction to where it should be going, due to spinning out. When trackPos is negative, the car is on the right-hand side of the track axis. These are important in calculations of rewards or early stopping, where the absolute values should be used to avoid any unnecessary problems.

4.3.2. Drawing Tracks

TORCS has no sensors that give explicit data about where the car is in the world, only implicit distances of track edges or the track axis from the perspective of the car. This meant that trying to understand the world would either have to rely on a self-made coordinate system or no coordinate system at all. To test how reliable performing logic was, from the limited context that the sensors give, a coordinate system was defined to gradually draw the track as an agent drove it, using basic kinematics.

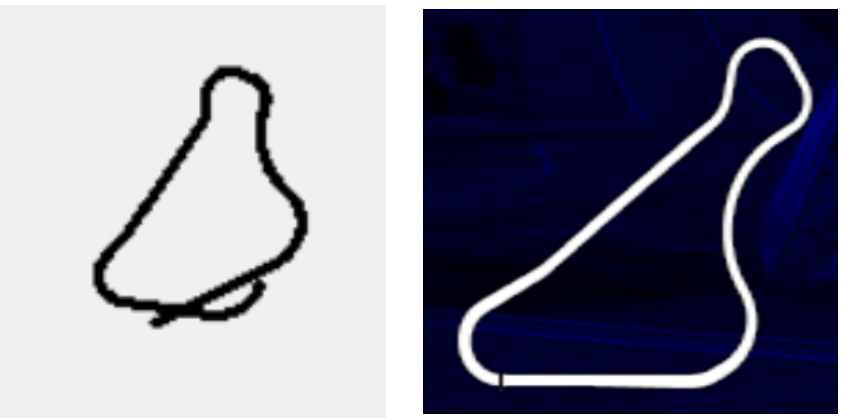


Figure 7. Shows the comparison between the drawing (left) and the actual track diagram (right)

Figure 7 shows that the track can be drawn mostly correctly, but becomes more unstable as the lap progresses, resulting in the second lap being slightly skewed. This is partially due to the car starting offset from the track axis but is mostly due to the difficult-to-interpret angles of the car.

Ultimately, this seemed to show that logic requiring explicit context from the world, such as determining corner properties, might be possible but would be unreliable. The *angle* sensor only describes the angle to the track axis, not the orientation in the world, so cannot be used with other sensors to decide where the car is on the track, i.e. you can't tell if the car is on a corner or a straight. Consequently, any similar calculations were avoided when training any agents.

4.3.3. Model Inputs

From the previous observations, the following sensors were determined as the optimal inputs for models:

- *angle* and *trackPos* to judge the direction of the track, so that the agent can learn to follow it and complete laps.
- *track* or *focus* return distances to the track edges, so that the agent can learn to stay within the boundaries of the track and avoid spinning out or losing speed. *focus* only has 5 sensors and acts over a small area so doesn't contribute much. *track* has 19 sensors, whose angles can be defined by the client, which proved more effective.
- *speedX* so the agent knows how fast it is going, which could teach it to slow for corners or speed up for straights.
- *speedY* so the agent understands that steering at high speeds can force the car outwards from its desired position on the track.

The combination of these is considered the state for reinforcement learning.

For the initial genetic algorithm, *focus* was used instead of *track*, due to the initial idea that it acted in a full 180-degree area in front of the car and that track acted in a complete circle around the car. This was determined to be false upon re-reading the competition paper, *focus* acts in a small area defined by the actuator of the same name. So all subsequent experiments used *track*, an actual 180-degree area ahead of the car, acting from -90 to 90 degrees.

The gym_torcs custom gym environment, adapted for this project, used additional sensors that were deemed unnecessary. These were *speedZ*, *wheelSpinVel*, and *rpm*. On a road track, *speedZ* is nearly always 0. *wheelSpinVel* may help the agent in correcting any drifting, however, drifting is quite unforgiving in TORCS, typically resulting in a crash or spin-out, which should be avoided entirely. *rpm* would have only been useful if the agent had to control its gearbox, which was avoided for simplicity.

The concept of racing with the awareness of opponents was ignored for this project, so the *opponents* sensor was not included as an input. Including the sensor would have made the models too complex for the scope of the project. The aim was to create agents that display good overall racing behaviours, such as taking the racing line. Having to learn where other cars are would have hindered this. For a racing agent that can rival human players, this would have to be included.

4.3.4. Model Outputs

After considering the actuators, only the *accel*, *brake*, and *steer* were necessary to create an agent capable of displaying positive racing behaviour. The values for these actuators had to be continuous to allow for using fractions of each one, for example, gradually steering for a large corner. Both *accel* and *brake* take values between 0 and 1, indicating the percentage of the actuator used, whereas steer takes values between -1 and 1, marking the range between fully turning right and fully turning left.

Giving the agent control of the gears would have been too complex to teach, so was ignored, and, by extension, the clutch was unnecessary. The sample agent came with an automatic gearing system, which simply set the gear based on the current speed. This did not need to use the clutch as the game engine automatically uses the clutch if the gear is changed. It was observed that agents tend to sit on the boundary of two gears, repeatedly shifting up and down, seemingly without reason.

The example agent also came with a traction control system, which was a small calculation that limited the accelerator if the back wheels were spinning too much. This was used for the initial genetic algorithm agent but was eventually scrapped for PPO and revising the genetic algorithm. The traction control imposed an excessive limitation on the agent, which should instead learn to manage traction independently.

4.4. Genetic Algorithm Driving Agents

The aim of implementing and testing the genetic algorithm was to create a simple baseline to compare to the reinforcement learning agents, due to the major differences in complexity between the two. This section details the experiments with the genetic algorithm using a simple neural network architecture, and later, a more complex architecture and fitness function. For the initial genetic algorithm, which is discussed in Sections 4.4.1 through 4.4.3, an architecture of 12 hidden neurons, with 9 inputs and 3 outputs, was used. This kept the search space, for the optimal weights and biases, small and aimed to avoid any complex behaviours being developed. Agents were trained on the tracks g-track-1 and forza here (see Appendix 3 for the track layouts).

4.4.1. Fitness Function

For the initial genetic algorithm, an agent's fitness was graded as the total distance travelled within 9,000 timesteps or roughly 3 minutes, using the *distRaced* sensor. This aimed to create agents that could stay on the track indefinitely and tried to promote going fast to get the furthest in the time limit. Because genetic algorithms do not learn directly from fitness (i.e., they do not collect rewards for specific actions, unlike reinforcement learning), keeping the fitness function simple would avoid agents finding a poor strategy while still being adequate for this problem. However, the genetic algorithm was revised late into the project to analyse whether the fitness function was too simple (discussed in Section 4.4.4).

Fitness was determined either at the end of the three minutes or when the agent was stopped early by a set of criteria. The criteria kept the agent from straying off of the track and driving slower than 20 km/h. This had a delay to allow the agent to accelerate to a speed above the threshold.

4.4.2. Single-Individual Population

A population with only one individual was the subject of the first experiment. This only included mutation and selection, no crossover. Mutation added a random noise chromosome to the individual's chromosome to create a child. The fittest chromosome between the two was then kept for the next generation. This generated agents that drove poorly and fell into local maxima constantly, as some good weights and biases could be lost between generations. This was largely due to the mutation limitations. The random noise had to be small enough to avoid missing maxima, however, this did not enable movement between maxima. This meant that most of an agent's training and fitness were almost completely reliant on the original random chromosome, even after training for 120 generations. Ten repeated experiments were conducted to validate this conclusion. The results are shown in the following graph.

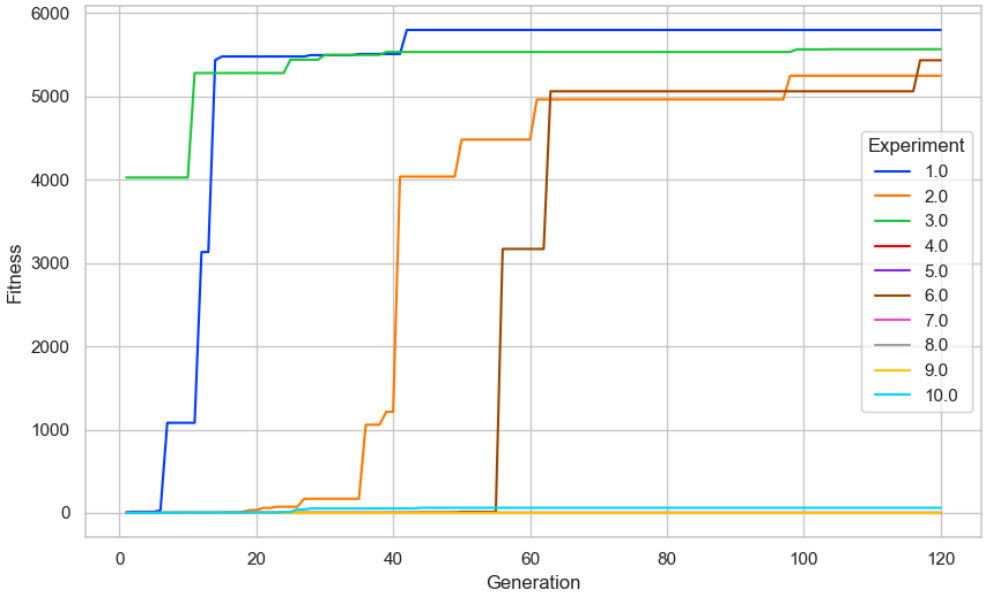


Figure 8. Shows the best fitnesses for ten experiments

As Figure 8 shows, some individuals can start with a large fitness due to the complete randomness of the chromosome. The long plateaus here show the local maxima that individuals get stuck in.

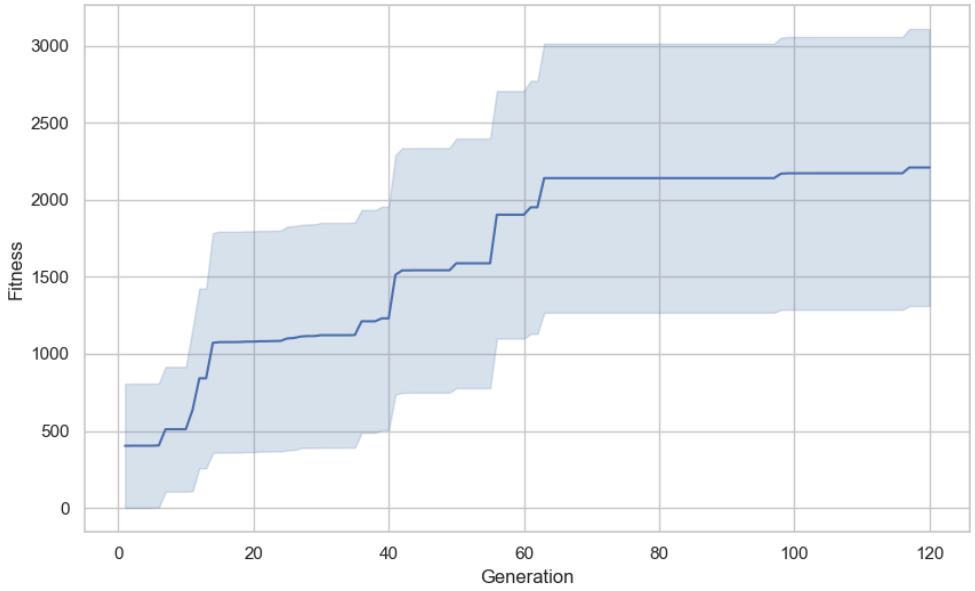


Figure 9. Shows the aggregate of best fitnesses for ten experiments

The large confidence ranges in Figure 9 show the inconsistency of having a single-individual population. Due to the random element of a genetic algorithm, the confidence ranges should be of a decent size, but this is too extreme.

To solve this problem, individuals should pass parts of their chromosomes onto the next generation to reduce the randomness of performance. This led to the inclusion of a multi-individual population.

4.4.3. Multi-Individual Population

A population of six individuals was used for these experiments. This size allowed for enough variety between individuals to develop appropriate behaviours, but not so many that each generation took too long. Using single-point crossover, the individuals create six children that are mutated and tested. The six fittest individuals of the total twelve were kept for the next generation. This allowed the passing of good “genes” onto the next generation.

After implementing this, it was observed that the initial randomness of the chromosomes no longer had a significant impact on overall performance as it had previously. Each experiment consistently achieved high fitness values by the end of 120 generations. The graph below demonstrates this, using data from ten repeated experiments.

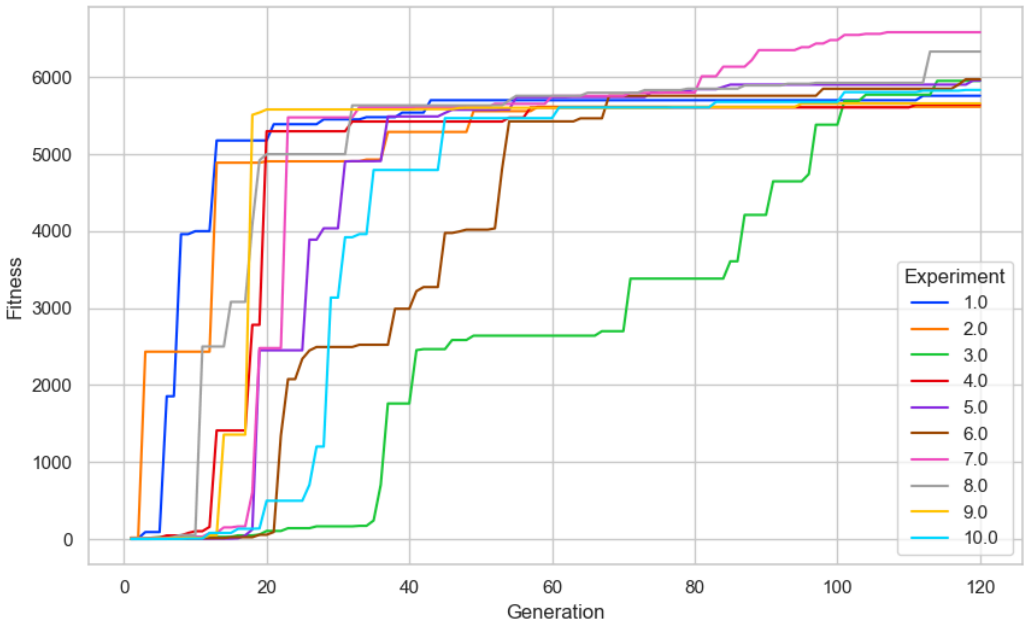


Figure 10. Shows the best fitnesses for ten experiments

As shown in Figure 10, most populations learned at a similar rate, reaching similar maxima. These create an individual that found a maximum that was global or close to it.

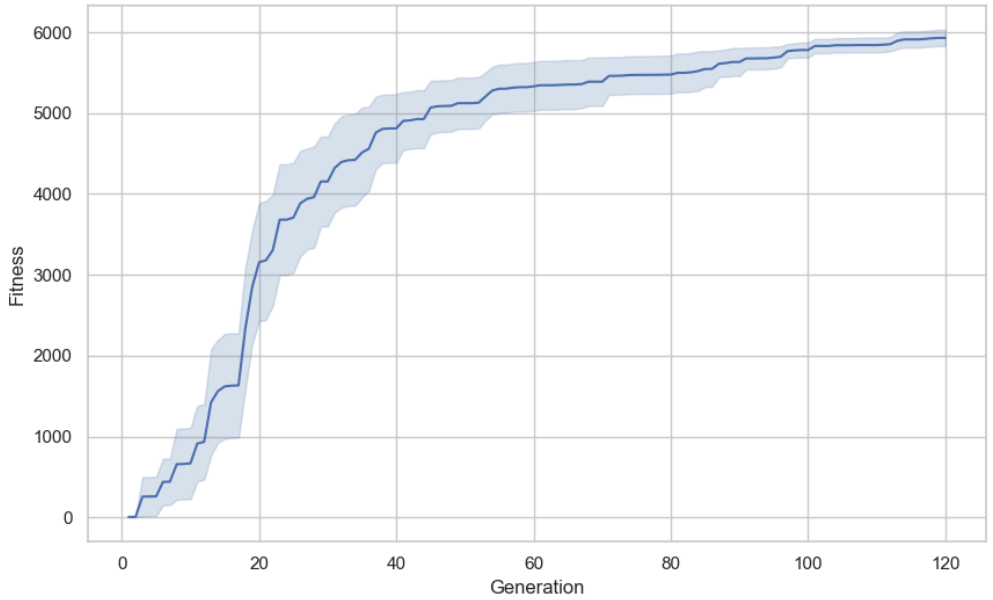


Figure 11. Shows the aggregate of best fitnesses for ten experiments

Figure 11 shows vastly smaller confidence ranges than Figure 9. This further illustrates the point that a multi-individual population is more consistent.

With multiple individuals in the population, an average fitness could be recorded at the end of every generation.

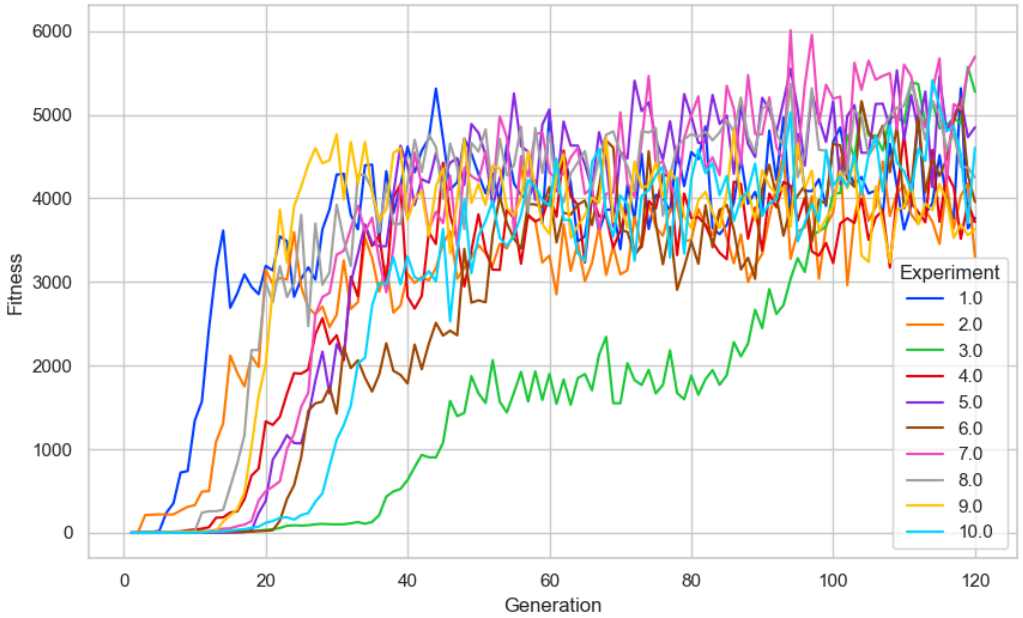


Figure 12. Shows the average fitnesses for ten experiments

Figure 12 is quite hectic but expertly demonstrates how the population grows together and benefits from each other's “genes”, shown by the upwards trend.

The best agents from these experiments demonstrate a simple behaviour of following the track axis and coasting at a steady speed. They also tend to weave from side to side of the track axis, rather than staying straight, resembling the oscillations of a sine wave. This seemed to be a limitation of using

such a simple neural network, as there weren't enough neuron connections to elicit complex behaviours. Experiments to support this hypothesis are detailed in the following section.

4.4.4. Revision of the Genetic Algorithm

The reason for returning to the genetic algorithm late into the project was to employ the newfound knowledge gained from experiments with PPO. This aimed to improve the performance seen previously by implementing a more complex neural network, and fitness function.

The population now consisted of 24 individuals, double that of the original. The maximum timesteps was reduced to conduct the experiments quickly, now for only 3,500 or roughly 70 seconds. The early-stopping criteria and generation limit of 120 were still in place. All individuals were tested on g-track-1.

The complexity of the neural network was increased by adding more hidden neurones and adjusting the inputs. The updated network now consisted of 36 hidden neurons and 23 inputs, as opposed to the original 12 and 9. The increase in inputs came from changing the *focus* sensor to the *track* sensor, as it acted over a larger area and was more reliable.

These experiments evaluated two fitness functions. The first was the original fitness function, that returns the distance travelled from the *distRaced* sensor. The second accumulated the fitness at each timestep into a total fitness value. This used an equivalent equation to the modified reward function discussed later in Section 4.5.1.2 and is shown below:

$$fitness = \sum \left(V_x \cdot \cos(\theta) - V_x \cdot \max(|trackPos| - 0.3, 0)^2 \right)$$

Using the original fitness function, agents now suprisingly learned to follow the racing line and drove at similar speeds to the agents trained using the initial genetic algorithm. This seems to be due to the greater complexity of the neural networks. They also drive much smoother now, with little to no weaving.

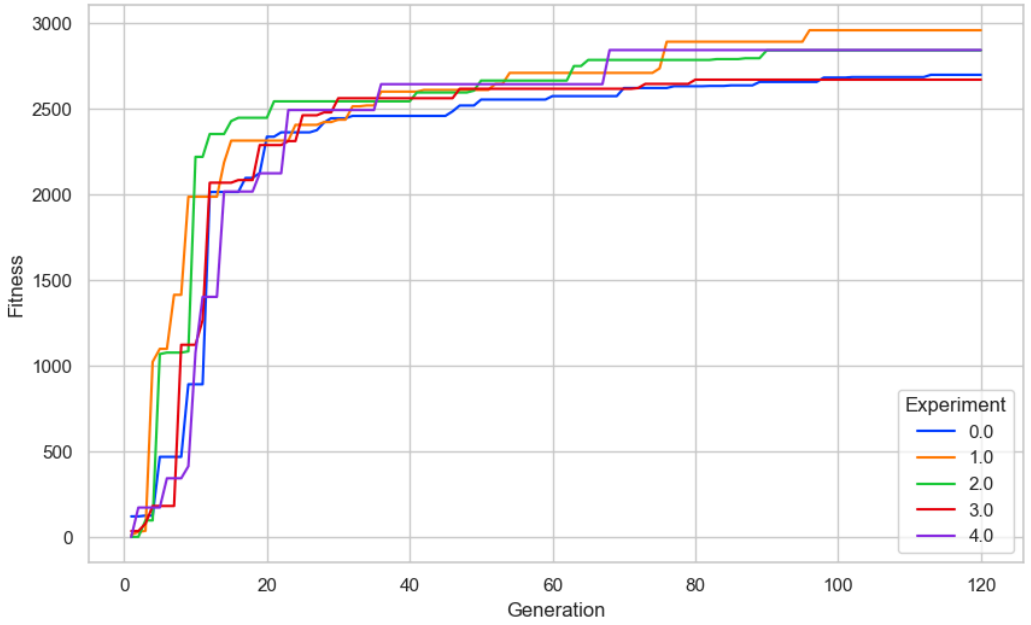


Figure 13. Shows the best fitnesses for five experiments using the original fitness function

Comparing Figure 13 to Figure 10 in Section 4.4.3, training is even more consistent now and the global maximum is found much faster. This further shows that the increase in complexity is beneficial.

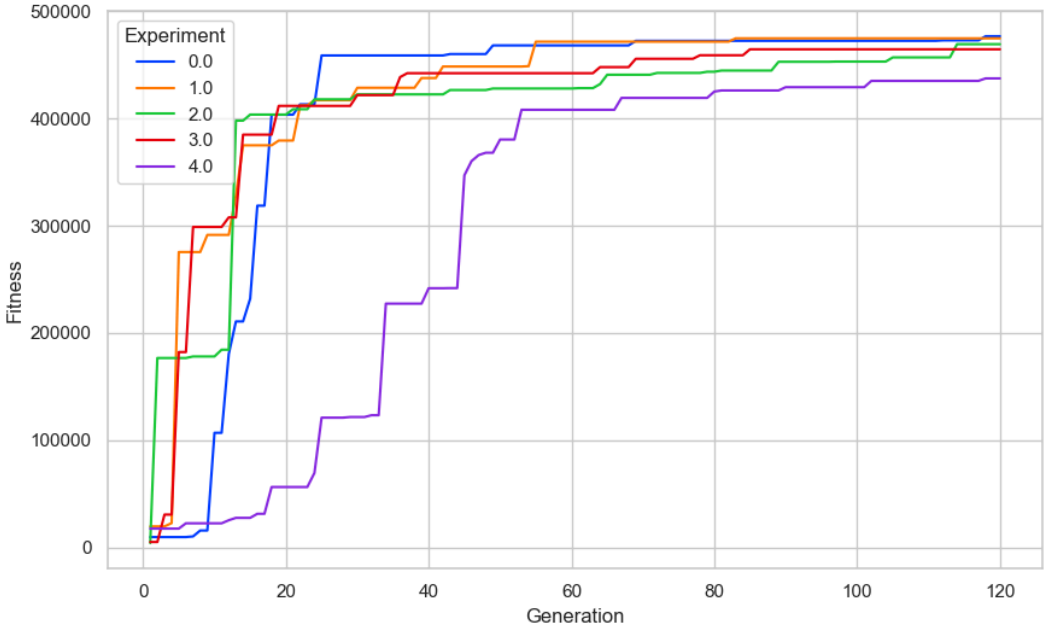


Figure 14. Shows the best fitnesses for five experiments using the modified fitness function

Using the modified fitness function, the population's fitness values are scaled vastly differently to the original, as shown in Figure 14. There is less consistency here but a maximum is still found quite quickly. Upon testing the best agent from these experiments, it can be observed trying to follow the track axis instead, and drives less smoothly than before. This is overall a poorer behaviour than seen in the previous experiment, and sometimes causes the agent to drive off of the track.

Both methods tend to accelerate to 140km/h and stay at that speed, where following the track axis becomes considerably more difficult. They also never use the brake. For a track as easy as g-track-1 and due to getting further with more speed, the brake is just ignored.

Comparing the results demonstrates the importance of the fitness function to an agent's overall performance. This could indicate that using simpler fitness functions is preferable for a genetic algorithm, or that the modified fitness function is simply inferior.

4.4.5. Final Agents

To conclude, the implementation of a population of multiple individuals was able to develop intermediate individuals for racing in TORCS. A single-individual population was not as effective and was too reliant on the random noise it was mutated with. As Figures 10 and 11 in Section 4.4.3 showed, the multi-individual population could achieve fitter individuals, more reliably.

Agents trained with the initial genetic algorithm can reach decent speeds, finishing laps in good time, but try to stick to the track axis as much as possible, weaving from side to side of the track axis. This leads to them getting overwhelmed on certain corners, which causes oversteering and spin-outs. The brake is barely used, agents instead learn to drive at a speed limit that doesn't cause them to drive off

the track, so tend to overfit the track they are trained on. If trained on an easy track, agents learn to drive too fast for more difficult tracks. As a consequence, the genetic algorithm was not used to train any agents on difficult tracks, as the agents would have learned to drive slowly to remain on the track.

The genetic algorithm was revised late into the project with newfound knowledge to create two final agents. Increasing the complexity of the neural network allowed for more complex positive behaviours to be learned, such as taking the racing line. It was also observed that using different fitness functions has a large effect on the performance of an individual and that a more complex fitness function does not necessarily generate better results. The trained individuals can finish laps of the g-track-1 race track in 55 seconds and can race nearly indefinitely, but don't ever use the brake during a lap. Driving is smooth, with no weaving around the track axis.

For videos demonstrating the two final agents, see Appendix 1.

4.5. PPO Driving Agents

This section details my extensive work with the PPO reinforcement learning algorithm. The models trained in this section have independent policy and value networks. Both share a common architecture, which consists of two fully connected layers with 64 neurons each and uses the Tanh activation function. The inputs are flattened to ensure that they are in an appropriate one-dimensional format for processing, and the final layer applies a linear transformation. See Appendix 2 for the raw representation of the network architecture.

4.5.1. Custom Gymnasium Environment

To understand the states and rewards of TORCS, a Gym/Gymnasium environment had to be created to supply them to the PPO algorithm. This environment is referred to by its class name, TorcsEnv, whose implementation is detailed in Section 5.5.1.

This was one of the two major research areas for PPO, alongside hyperparameters. The following subsections detail the key observations and results from experiments conducted regarding this environment.

4.5.1.1. Early-Stopping Criteria

To avoid agents learning any race-ending behaviours, a set of criteria should limit the length of an episode if certain states hinder training. For example, an agent may find that more reward can easily be earned if drives completely straight, even off of the track. This is bad if we want an agent that can complete laps by following the track and its corners. To solve this, a limit can be put on the car's displacement from the track axis. So in TORCS, if the *trackPos* sensor returns a value greater than 1, the car is off the track and the episode must end. This helps frequently during the early stages of training when the agent is testing which directions earn the most reward, as it shortens episodes to only the useful state/action pairs.

Some other early-stopping criteria were employed in these experiments to help with similar problems, each of which is checked at every timestep.

First is the collision detection criterion. TorcsEnv stores the observation/state of the current and previous timestep. These are used to compare damage percentages. If the damage percentages are different, the car must have collided with an object or opponent. In this instance, collisions always indicate that the car has hit a barrier outside the track. This criterion can either act as a backup or a replacement for the car displacement condition.

Next is the orientation criterion. This tracks cosine of the car's angle to the track axis. A negativc value refers to an angle of over ± π/2 or, more simply, facing the wrong direction. This is the result of spinning out, coming from poor use of any of the actuators and should end the episode.

For each of these criteria, deciding whether to deduct reward is important for the learning process. Initially, it might seem more beneficial to penalise the agent for the poor behaviour, but with a high discount factor, this can dramatically affect the rewards at multiple timesteps and the policy may update in unexpected ways. However, removing the reward deduction may not give the algorithm enough context as to which actions are bad and why episodes conclude early. It was decided that rewards should be deducted for these cases but only as a tiny negative reward of -1. This is relatively small compared to the reward function averages of 50. However, when using smaller discount factors, it is more beneficial to increase the deduction, so that it contributes more.

The final criteria ends an episode when the reward is too low. This forces the agent to keep trying to maximise near-immediate rewards, rather than doing less and generating only mediocre rewards that slowly build up due to not being stopped. In this case, driving slowly, without the risk of going off the track, earns good rewards overall but lap times would be significantly longer, most likely losing the race. Whereas, if a limit is put in place, the agent won't learn to drive so slowly. There should be a small delay imposed on this criteria to allow the agent to accelerate to enough speed where rewards are above the defined threshold, due to the car starting stationary. A reward deduction should never occur in this criteria.

4.5.1.2. Reward Functions

Reward functions are incredibly important for a reinforcement learning algorithm, but finding one that is effective and promotes the proper behaviour is difficult, especially for such a complex state space as TORCS. This section describes each reward function tested and explains the intention behind them.

In the reward function definitions below, V_x refers to the *xSpeed* value and θ refers to the *angle* value.

The two reward functions detailed below do not directly promote braking, which the agent should learn itself. The first being the original reward function from Gym-TORCS, proposed in this paper [32], and is the following:

$$reward_t = V_x \cdot \cos(\theta) - |V_x \cdot \sin(\theta)| - V_x \cdot |trackPos|$$

The first term is the forward speed of the car in the direction of the track, teaching the agent to follow the track around and accelerate. The second term reduces reward based on the speed of the car perpendicular to the track axis, with the third term reducing reward based on the car's displacement from the track axis. These both penalise the agent for straying from the centre of the track, more so at higher speeds. Agents learn to completely follow the track axis with this reward function, which isn't an optimal behaviour if the objective is to beat opponents and finish laps in the fastest time.

To incentivise using some of the width of the track to potentially take the racing line (a better behaviour), this reward function was modified to be the following:

$$reward_t = V_x \cdot \cos(\theta) - V_x \cdot \max(|trackPos| - 0.3, 0)^2$$

The first term has not changed from the original reward function. The second term of the original function was completely unnecessary to the change in behaviour, so was removed. The last term now has a threshold for the displacement from the track axis, only penalising the agent for straying too far out, allowing for some exploration. A potential problem in the logic of this was that the agent might learn to maximise rewards by just driving straight at a high speed, ignoring the fact it was off-centre, and flying off the track. However, this reward function did show the desired behaviour. Agents learned to follow the racing line. When approaching a corner, it would move slightly outwards before steering to create the best turning angle. This allowed for more speed whilst staying on the track, getting further into a lap.

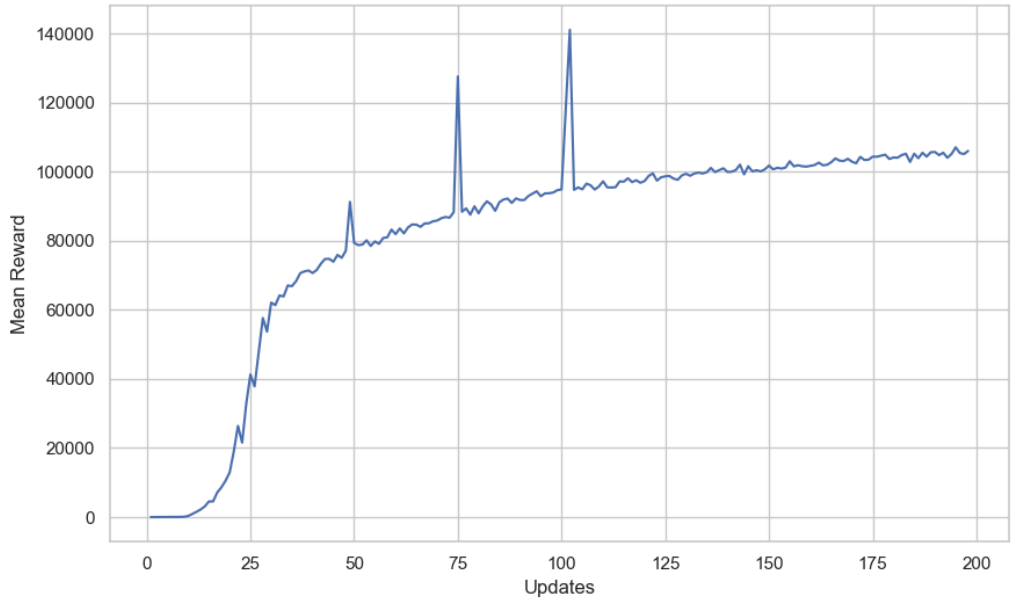


Figure 15. Shows the mean reward of each policy update using the original reward function

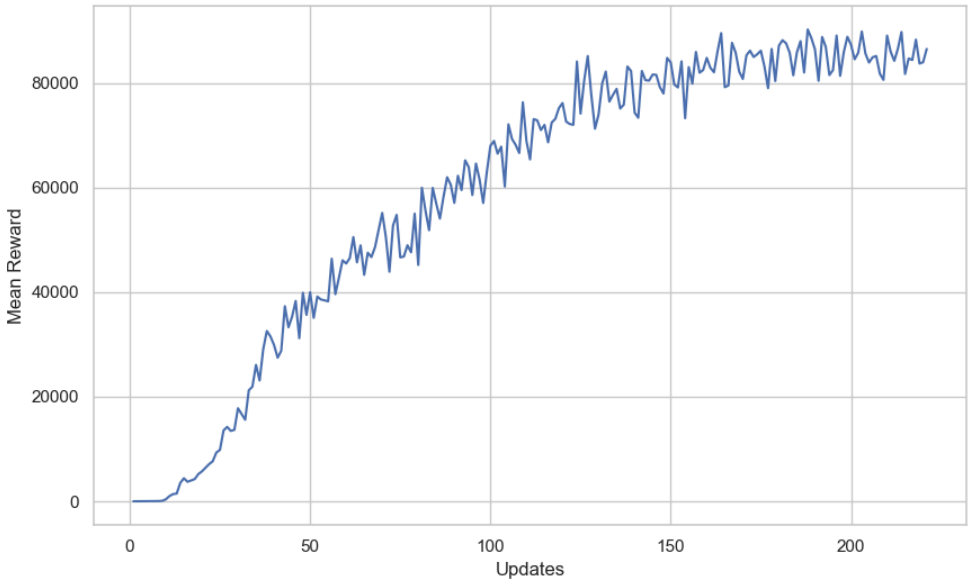


Figure 16. Shows the mean reward of each policy update using the modified reward function

Comparing Figures 15 and 16, the original reward function gets a higher total reward. This is due to it being more stable on the track, so can get further in a lap, having more chances to get rewards. The modified reward function created a behaviour that has more risk, so episodes would often end early. However, the behaviour observed from the modified reward function was more desirable for a racing agent.

Finding a balance between these wasn't the problem here. Both reward functions generated agents that constantly had trouble learning to brake effectively. So as soon as an agent approached a difficult corner, it would be travelling too fast and fly off of the track. For these two reward functions, braking was inherently bad for maximising reward. It decreases speed which decreases the positive reward given. This led to the creation of two reward functions that tried to directly promote braking, aiming to give the agents a reason to brake. The two reward functions detailed below used the modified reward function as the base and added other terms, so the common terms will not be explained.

The first aimed to give large rewards for braking at high speeds. The second term in the following reward function uses the *brake* actuator to increase rewards.

$$reward_t = V_x \cdot \cos(\theta) + V_x \cdot brake - V_x \cdot \max(|trackPos| - 3, 0)^2$$

This ended up being quite poor. The agents learned to briefly accelerate and then immediately hit the brake every split second. This achieved nothing, not even making it to the first corner before the maximum timesteps were reached. The agents would also not learn that going off the track was bad for continuing the episode and generating more rewards.

The second reward function aimed to reduce rewards for driving at high speeds while facing a track edge. This was to penalise an agent for not braking and steering while approaching a corner, resulting in the following:

$$reward_t = V_x \cdot \cos(\theta) - V_x \cdot (1 - track[9]) - V_x \cdot \max(|trackPos| - 3, 0)^2$$

The term "*track*[9]" here refers to the centre angle of the *track* sensor, which returns the distance of a track edge directly in front of the car. Minusing this from 1 gives a maximum value when close to a track edge, as "*track*[9]" is closer to 0.

Agents trained using this reward function didn't show any differences in mean rewards compared to the first two reward functions. However, upon testing, the agent found an exploit in this reward function. It found that if it stuck the front of the car just outside the track edge, to not trigger the out-of-bounds condition, it could nullify the reward loss. The agent would speed down the track edge to maximise reward until reaching the first corner, where it could not recover and went out of bounds. This seems to show that the *track* sensor returns the maximum value when there is no edge to detect, resulting in the second term equalling 0.

Many variations of the previous two reward functions were also tested, where limits were placed on each term, but these all failed to generate the desired behaviour. This shows that micro-managing a reward function does not give the intended results. Agents should learn the behaviours themselves from a general reward function, shown in the results from the first two reward functions. The majority of the reward functions tested in these experiments also showed that reinforcement learning models tend to find flaws in the reward function design. If there is something that the model can exploit for greater reward, it will. Also, if there are any difficult-to-exit local maxima, it will get stuck in them.

4.5.2. Hyperparameters

Throughout the experiments, it was found that four hyperparameters significantly impacted model performance, which are described below.

- n_steps: The number of timesteps to run the environment for each policy update
- Learning rate
- Batch size: The number of samples used in each iteration of gradient ascent
- Gamma: The discount factor

The n_steps was the least important of the four, altough it must be large enough to allow the agent to experience the track in multiple episodes. The state space is quite complex in TORCS, so large amounts of training data should be collected for effective policy updates. Both 10,000 and 100,000 were tested as values for n_steps, with minimal differences observed. Ultimately, an n_steps of 100,000 was selected to ensure that updates to the policy didn't occur too frequently.

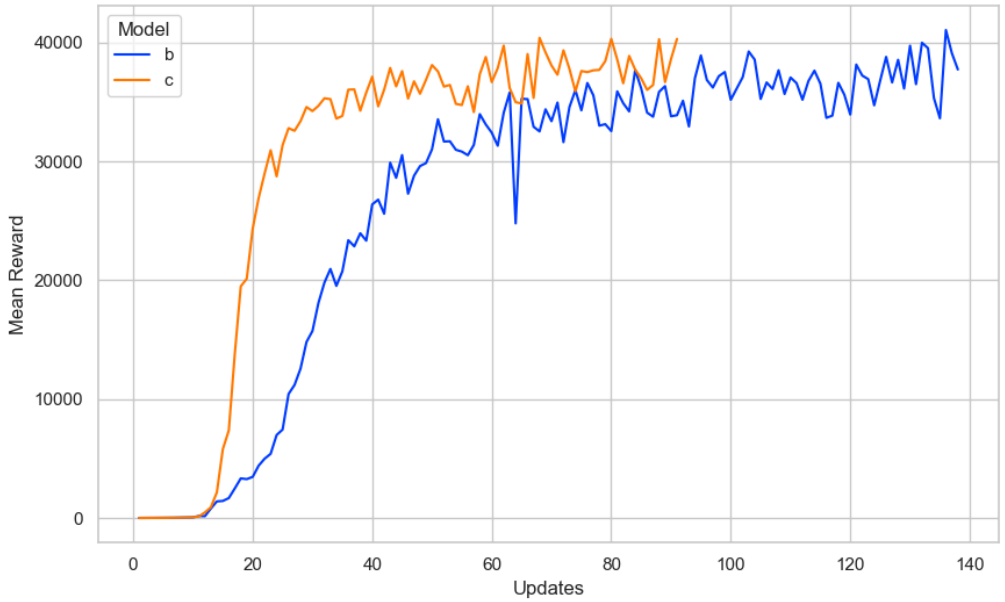


Figure 17. Compares the mean reward of each policy update for two models

In Figure 17, model b's n_steps is 10,000, whereas model c's is 100,000, and all other parameters are the same. These are difficult to compare because the total timesteps each model is trained for is vastly different, model c takes ten times as many timesteps to update the policy once. However, this does show that using more experience for a policy update allows agents to generate high rewards in less policy updates.

The learning rate also less important to an agent's performance. Any values between 0.0001 and 0.001 were adequate, anything outside of this range trained poorly. Too low would cause the agent to learn too slowly, needing too many episodes to learn any behaviour. Consequently, values above 0.001 were not tested, based on the assumption that such values would lead to drastic changes to the policy. Beyond the grid searches detailed in Section 4.5.2.2, the learning rate remained unchanged, with a value of 0.0003. This is coincidentally the default for the PPO algorithm.

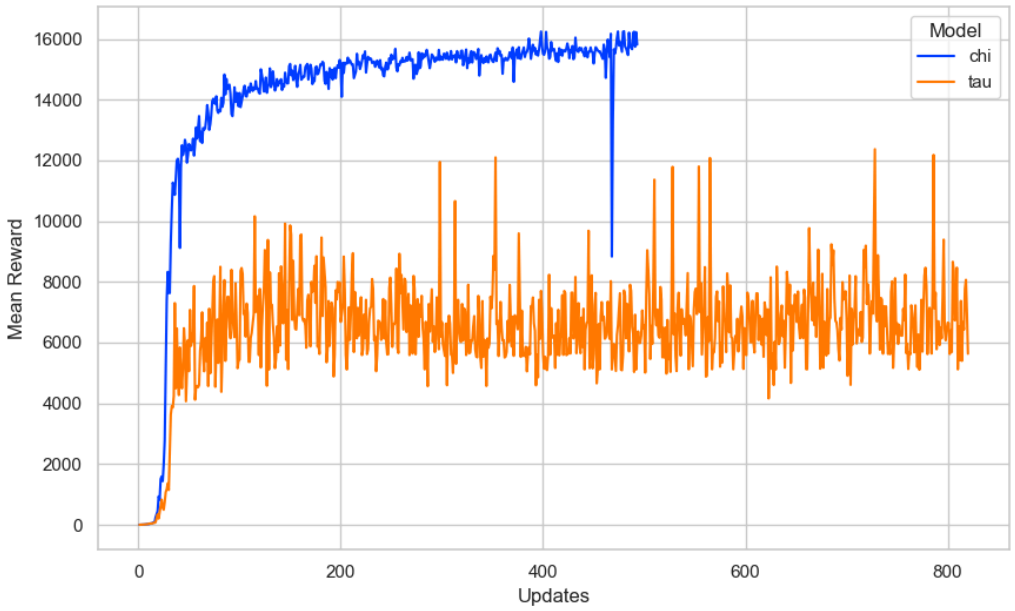


Figure 18. Compares the mean reward of each policy update for two models

In Figure 18, model tau's learning rate is 0.001, whereas model chi's is 0.0003, and all other parameters are the same. This shows having a learning rate that is too high can teach the model too well, where it falls into a poor local maximum.

The batch size was quite important in training a good agent. Having more data collected for gradient ascent and policy updates made an agent's final performance more stable, steering was less erratic, and seemed to look more 'human'. This came with a small downside, agents would take longer to train (more total timesteps), due to the more accurate policy updates. Values below 100 are too small. Ultimately, a value of 1,000 was deemed ideal.

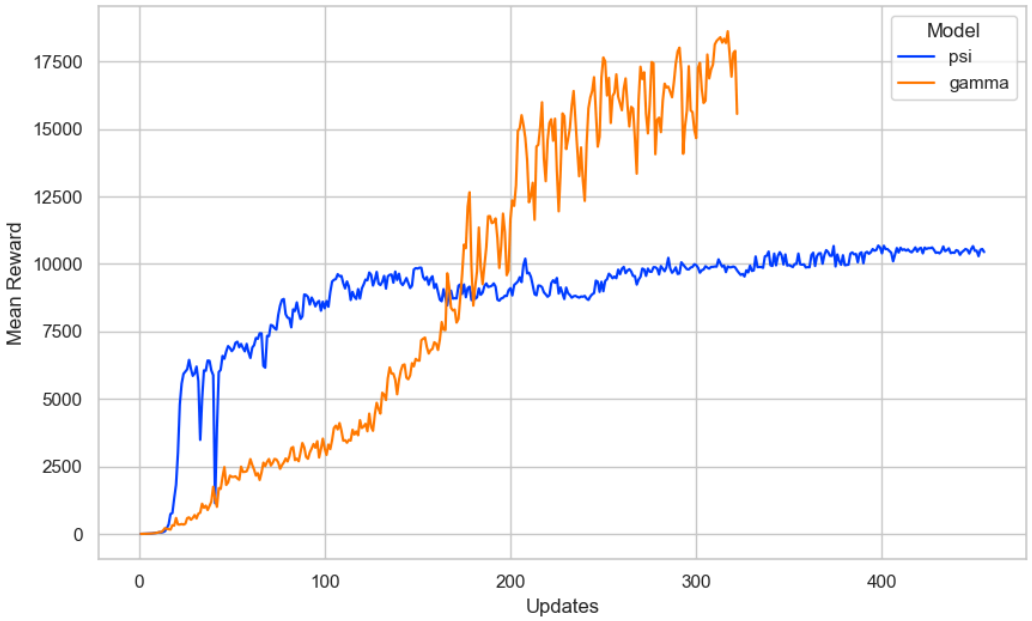


Figure 19. Compares the mean reward of each policy update for two models

In Figure 19, model psi's batch size is 100, whereas model gamma's is 1,000, and all other parameters are the same. This shows that models with higher batch sizes take longer to train but can perform much better, gaining larger total rewards, and can find better local maxima.

4.5.2.1. Discount Factor

Throughout the majority of the experiments with PPO, gamma values between 0.9 and 0.99 were used. This was the recommended range, so nothing was explored outside of this range. As a result, it was observed that agents would only use a maximum of 60% of the steer actuator, showing that agents were trying to maximise short-term rewards by staying fast, as steering slows the car. This priority for near-immediate rewards led to the conclusion that these values were too low.

As the game runs at 50 timesteps per second, later rewards can be lost if the discount factor is too low. For example, a corner usually takes 3-4 seconds to navigate, between the approach and the exit. With a low gamma, the last reward, that should be considered when cornering, would be scaled down to practically 0. Using 4 seconds as the time for a corner, the following calculations illustrate this point:

$$0.9^{4 \cdot 50} = 0.0000000007055$$

$$0.99^{4.50} = 0.13398$$

These show that to allow rewards after a corner to matter when an agent decides which actions to take, the discount factor should be considerably higher. The gamma value was changed to 0.9999 to resolve this, which gives the following scale factor:

$$0.9999^{4.50} = 0.9802$$

This shows that future rewards now have a large influence on reward calculations. Testing this value, it was observed that training was more efficient, agents learned to accelerate more often in earlier policy updates, to increase future rewards from already going fast. However, due to problems with braking (discussed in Sections 4.5.1.2 and 4.5.4), this ultimately didn't yield an agent that could corner effectively.

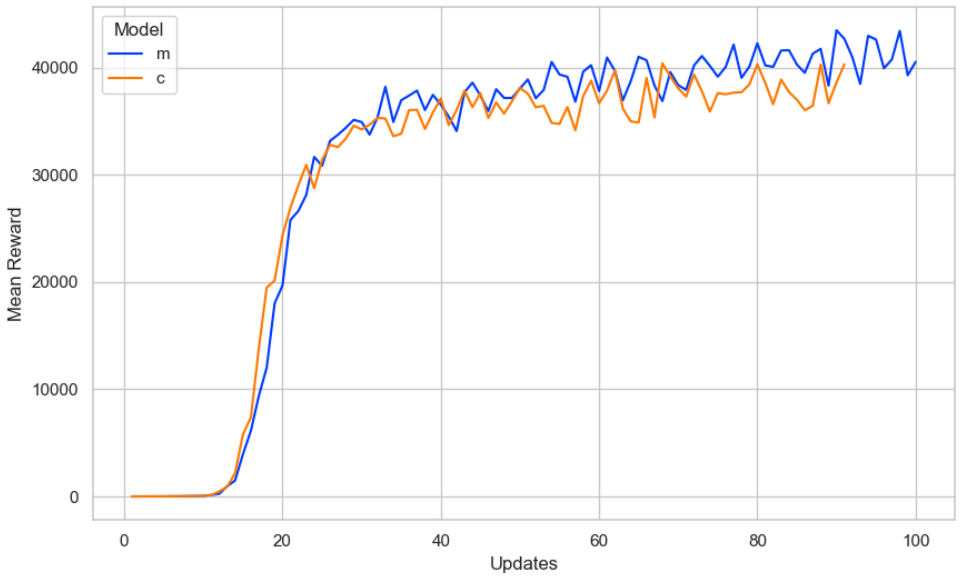


Figure 20. Compares the mean reward of each policy update for two models

In Figure 20, model c's gamma is 0.99, whereas model m's is 0.9999, and all other parameters are the same. These aren't particularly comparable as they are technically different reward functions, as a higher gamma will mostly return a larger reward. However, when comparing using a metric other than rewards, as shown below in Figure 21, there is still little difference, mostly due to the braking problem detailed in Sections 4.5.1.2 and 4.5.4.

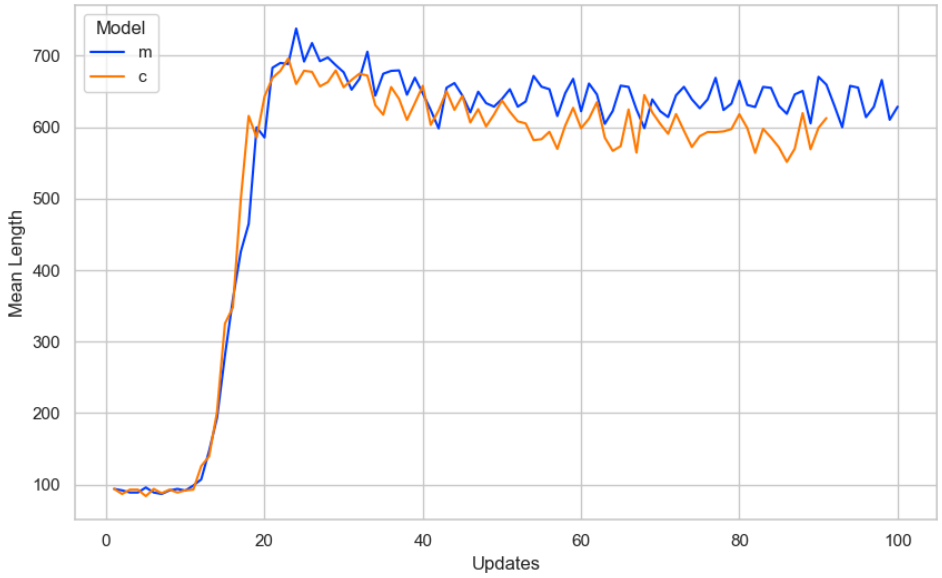


Figure 21. Compares the mean episode length of each policy update for two models

When increasing the discount factor, the early-stopping negative rewards had to be reconsidered, as discussed in Section 4.5.1.1. The agents became too "afraid" of the track edges when using such a large gamma, such that training didn't get anywhere.

4.5.2.2. Grid Searches

A grid search refers to testing every combination, or permutation, of parameters given a set of values for each parameter type. This finds the best permutation for training a model. These can be quite useful for determining rough values for each parameter that can be finely tuned to reach an optimal set of parameters.

This technique was to be used in two experiments. The first intended to check five values for each of the four previously-stated hyperparameters, but was ultimately scrapped. The five values were decided from the defaults of the PPO algorithm and rough recommendations from the book "Reinforcement Learning: An Introduction" by Sutton and Barto [19]. Each combination would be used to create a model which would train for a set number of timesteps, recording training metrics as it went. Five values each for four parameters is 5^4 permutations or 625. Each permutation would have to be run for an adequate time to see any differences in performance, making this task an extremely large commitment, roughly an hour each permutation.

After further consideration, values for n_steps and batch size were decided to limit the grid search to the more variable parameters, gamma and learning rate.

In the official grid search, some largely different values were tested for both parameters, to consider every range. The values {0.9, 0.95, 0.97, 0.99, 0.995} were tested for gamma, and the values {0.0001, 0.0003, 0.001} were tested for learning rate.

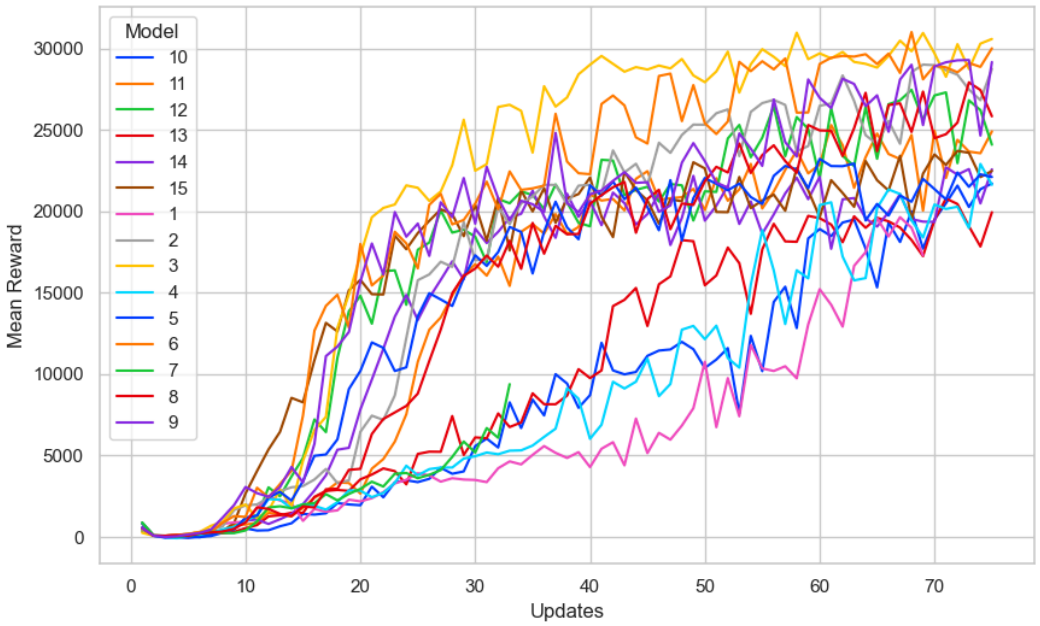


Figure 22. Compares the mean reward of each policy update for the fifteen permutations of the grid search

Figure 22 shows the results of this grid search, and was used to determine which set of parameters was the best, where model 3 had the {0.9, 0.001} parameter set, corresponding to gamma and learning rate respectively. This shows that there is little variation between most of the permutations.

After many revisions of the hyperparameters, this grid search was deemed useless, and the results weren't used in the final models. Each test was far too short to conclude anything. However, if the number of timesteps to train for was higher, the grid searches would take even longer. Also, each permutation should have been tested multiple times, as training with the same parameters isn't consistent.

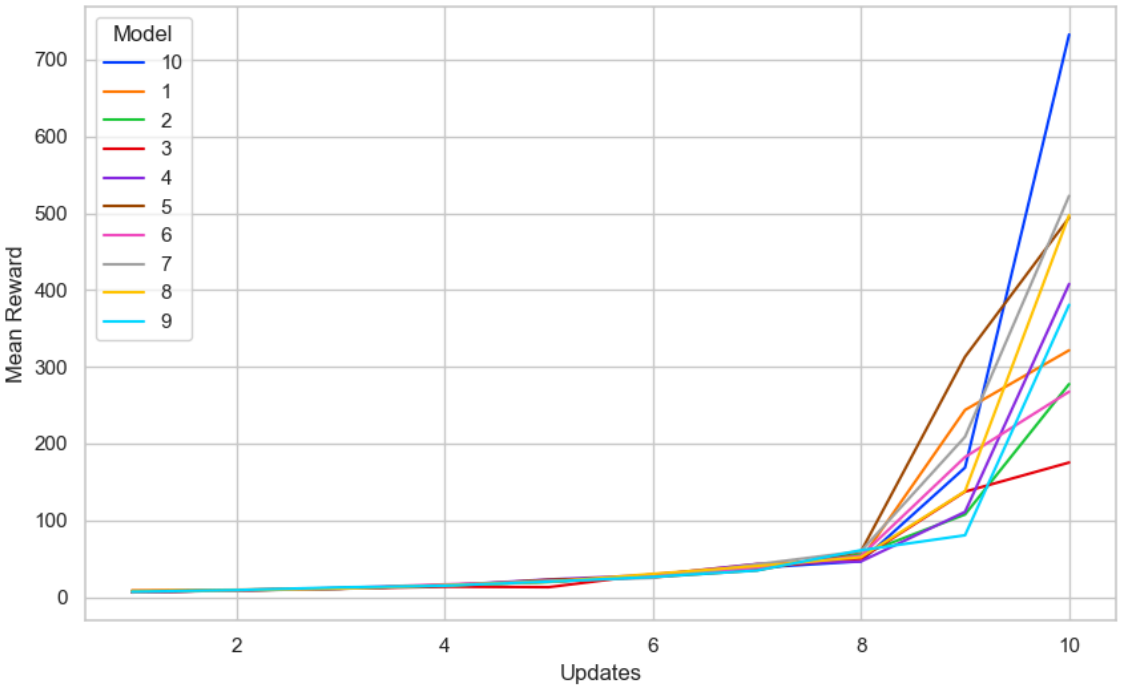


Figure 23. Compares the mean reward of each policy update for ten models

Figure 23 shows ten training sessions that use an identical parameter set. This shows that each training session can train at different rates, achieving largely different rewards until the models are fully trained, further validating the disadvantages of such a short grid search. Averages of multiple tests with identical parameter sets should be used to fully judge the parameter sets performance.

4.5.3. Multi-Track Training Sessions

Training a model on multiple tracks aimed to create an agent that was generally good for any track, testing the hypothesis that training on only a single track would cause overfitting. Ten tracks were chosen for this experiment, each having to be a road track of moderate difficulty, the ten are as follows: g-track-3, ruudskogen, street-1, aalborg, alpine-1, alpine-2, corkscrew, e-track-1, e-track-2, e-track-3. These are the file names for the tracks, as in-game they can be named slightly differently.

Upon numerous tests of using a new track after every episode, the agent seemed to not show any better behaviour or reward metrics, in some cases this was less effective. It seems the PPO algorithm can get confused if the reward space is changed often/suddenly.

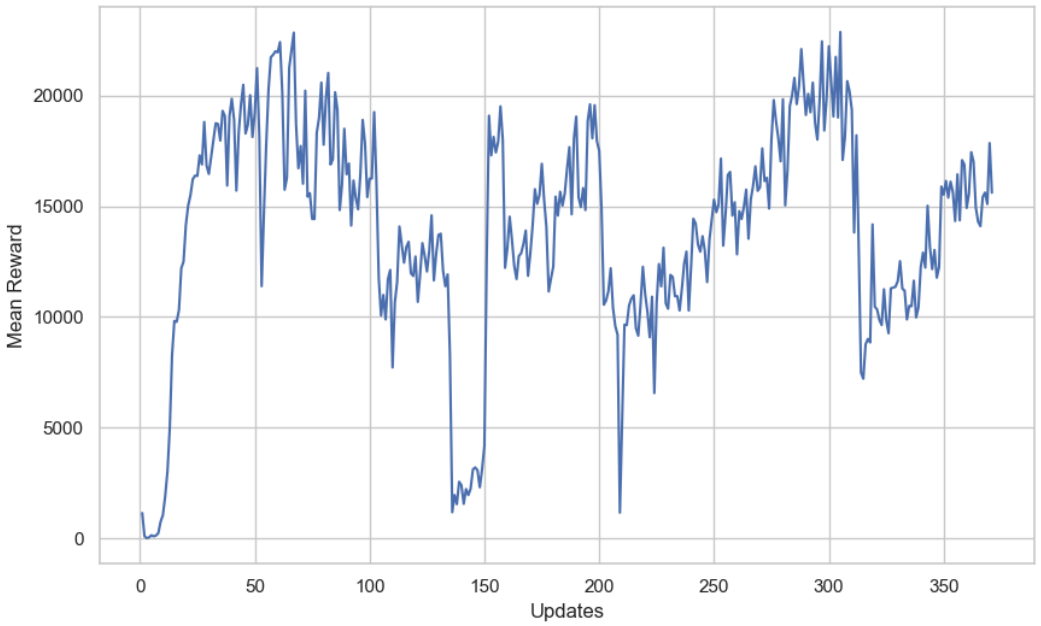


Figure 24. Shows the mean reward of each policy update for a model trained on multiple tracks

Figure 24 shows how training on multiple tracks can lead to fluctuations in rewards, which confuses the model. As each update gives an average of the reward of its episodes, this doesn't show that the agent gains vastly different rewards on each track, rather the large changes to the policy each update.

Images for the layout of each selected track, alongside some others, can be found in Appendix 3.

4.5.4. Stochastic Brake

As discussed previously, all the experiments with PPO show a major problem, agents have difficulty learning to brake effectively. In an attempt to solve this, a stochastic brake was implemented. This

forced the agent to brake at random intervals, aiming to teach the agent that slowing down for corners had the long-term benefit of gaining more reward later. Ben Lau's work [32] on TORCS coincidentally drew the same conclusion and had implemented it into their DDPG algorithm.

The stochastic brake required a few experiments to be implemented effectively. The implementation only changes the value for the brake actuator from a given action, using random number generation to act as a percentage chance.

The first experiment tested how often should this be used. Using the stochastic brake frequently would cause the agent to drive nowhere and learn very little. A chance of 20% or less was deemed ideal, settling on 10% to only have a slight effect on the agent.

The next experiment tested how much of the brake should be used when this activates. Originally, the *brake* value was set to a random number between 0 and 1. This aimed to teach agents that varying the use of the brake would have beneficial effects. However, this was too harsh for agents. When the *brake* value is above 0.5, the car can either spin out or the wheels can lock up, making steering difficult. This was changed to a static value of 0.2, aiming to lessen these issues and teach the agent as intended.

The final experiment gave the stochastic brake a delay before it activates. This depended on which track the agent was being trained on, as the delay should end before the first difficult corner. For the aalborg track, which was primarily used in these experiments, the delay was 500 timesteps.

Despite the logic behind this stochastic brake, all trained agents were still unable to use the brake effectively, with the brake mostly being ignored.

4.5.5. Further Observations

The PPO algorithm tends to generate models that seem to jerk from side to side when driving straight. This is due to the stochastic policy of PPO and the fact that TORCS doesn't limit the use of the steering wheel. An agent can turn fully left at one timestep and turn fully right at the next. In addition to this, models have a hard time overcoming oversteer. To recover from steering poorly, agents overcompensate by steering too much, making the problem worse. These problems lessen after extended training sessions, creating agents that drive smoother but not completely. This is again due to the stochastic policy which slowly phases out poor actions.

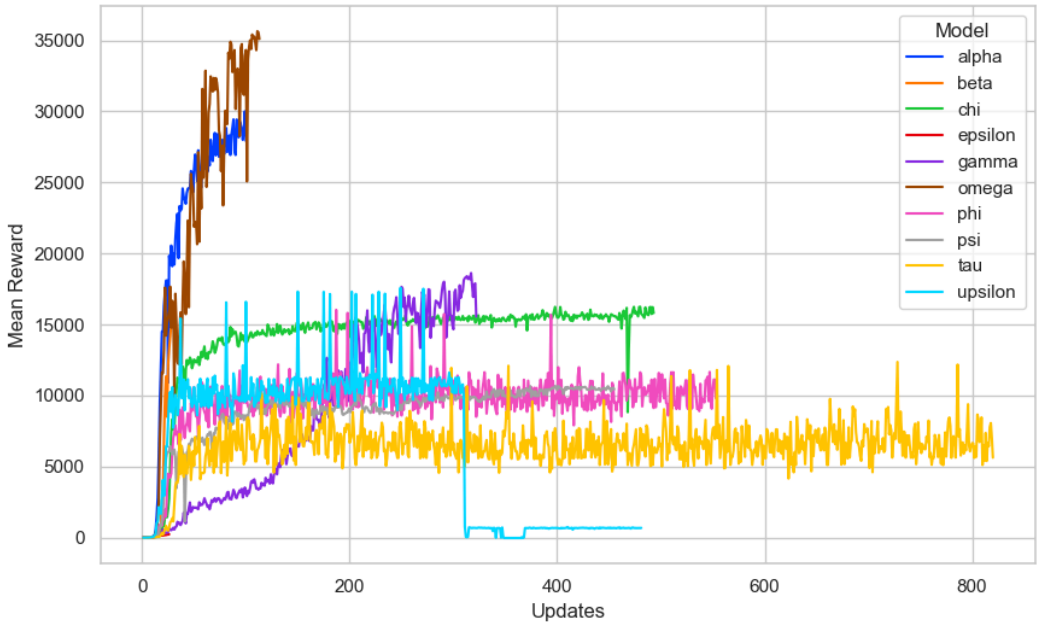


Figure 25. Compares the mean reward of each policy update for many models

Figure 25 shows that using various reward functions, hyperparameters, or tracks for a collection of models makes comparisons in a combined graph difficult. The number of episodes in each update or the mean reward of an update can have vastly different ranges. The total number of updates depends only on the n_steps parameter and how long the model is trained for. This means that for somewhat accurate comparisons, these have to be very similar.

4.5.6. Final Agents

Combining all previous observations, two final agents were trained to show the best behaviours that could be learned. Both agent use a discount factor of 0.9999, a learning rate of 0.0003, a batch size of 1,000, and a n_steps of 100,000.

The first model was trained on g-track-1 with the original reward function. This one learns to stick closely to the track axis and can reach speeds of 140km/h. The second agent was also trained on g-track-1 but with the modified reward function. This one starts to follow the racing line and can also reach speeds of 140km/h. Despite the second agent displaying a better racing behaviour, it is too volatile to consistently perform better than the first agent. The first agent reliably gets further into a lap than the second, and can even finish laps of g-track-1, with an average lap time of 55 seconds. Neither agent learns to use the brake at all, limiting them to simple tracks.

Many things contribute to an agent's ability to learn to brake, mainly the reward function and discount factor. No corner angles can be determined to aid in the reward function due to the limited sensor data. With this and the complex state/action space, training an agent that can swiftly complete laps while showing every desired behaviour is quite difficult. Each component needs excessive tuning to create a complete agent.

However, despite not achieving the goal of this project, this thesis has explored many of the components of PPO, determining good parameters for efficiently training for this problem. Every

behaviour shown, other than braking, can be easily trained by these parameters and in 20 hours of training, the agents drive smoothly and start to appear somewhat “human”.

Training a PPO agent requires a considerable amount of time to reduce erratic behaviour. Early into a training session, actions are mostly random, as the stochastic policy hasn’t reduced the chance of taking poor actions yet. This gives a somewhat shake to how the car drives, the agent repeatedly changes which way it is steering due to not learning that driving completely straight can maximise speed.

One hypothesis that wasn’t tested during this project was how training in batches could help the braking problem. This could include scheduling hyperparameters, for example, using a high learning rate early into training and slowly diminishing it over time. Also, changing reward functions in the middle of training could be considered. This would require much thought to ease the transition between reward functions, but agents could learn to drive straight first and then learn to brake after, rather than at the same time.

For videos demonstrating the two final agents, see Appendix 1.

4.6 Limitations

All programming, training, and observations were conducted on a single computer. This limited the progress that could be made at any given time. Agents took from 4 to 14 hours to train, where no programming could be done. Combine this with the crashing issues discussed in Section 5.1.3, and plenty of time was either wasted or spent working passively. Having other machines to conduct multiple experiments, each testing a different hypothesis, simultaneously, would have possibly allowed for an optimal reward function to be found within the timeframe of this project.

5. Implementations

This project was completed on the Windows 11 operating system, with Python as the only programming language.

This section details all code needed to implement the client, genetic algorithm and PPO algorithm to conduct the research detailed in Section 4. If any Python files are needed for additional context, see Appendix 1 for the repositories the files are stored in. The file names are included in the relevant sections.

5.1. Setup

This section details how the client was implemented to correctly communicate with the game server during training. This was adapted from the original snakeoil client given by Gym-TORCS [36].

5.1.1. Client

This class (see client.py) controls the running of the game and the connection to the server, which is quite complicated and is mostly unchanged from snakeoil.

In the setup_connection( ) method, a socket is created and an initial message is sent on port 3001, which determines the track sensor angles. The client then waits for an ***identified*** response from the game server. Then, at every timestep, get_server_inputs( ) is called to retrieve the string of data from the socket, to determine the current sensor data.

To accommodate cycling through different tracks, new practice mode files had to be created. This was simple, just create copies of the existing race configuration *.xml* file and change the track name inside each, making sure to name them differently. These are named 1 through 10 for easy iteration when a connection is identified. An example of a *.xml* file can be found in Appendix 4.

The following two subsections detail how the client handles the sensor inputs and the actuator outputs. Both subsections refer to a specific class.

5.1.1.1. ServerState

This class is used to interpret and store the sensor inputs from the game server. At the beginning of a timestep, the string retrieved from the socket is converted into a dictionary of sensor names and their values, such that a data entry can be accessed from its name. These could be lists or just single values and correspond to the sensors detailed in Section 2.1.2.1.

(see server_state.py)

5.1.1.2. DriverAction

This class holds all actuator values outputted by the agent before they are sent to the game server to enact the actions. These are stored in a dictionary similar to the ServerState one. Each actuator can be accessed by name. When responding to the game server, DriverAction translates the dictionary into the string representation needed. The class also clips any actions that are out of bounds of the ranges shown in Section 2.1.2.2, such that the game doesn’t reject them.

(see driver_action.py)

5.1.2. Running Headlessly

To run the game without the graphics, only in a text format, or “headlessly”, the competition paper details that you should run the game from the command line using the “-r” option followed by a .xml race configuration file and its path. The following is an example of said command:

wtorcs.exe -r config\\raceman\\practice.xml

A list of strings is held by Client and is combined into this command that is run using subprocess.Popen( ) [38]. Replacing *practice* in the command with a number between 1 and 10 allows for changing between tracks while training.

Running headlessly decreases training time considerably. It was observed that training took approximately 18 times less time to complete compared to running with the graphics enabled.

The primary issue with running the game this way was that restarting it through the *meta* actuator would cause a crash, terminating training. There is a bug in TORCS that causes the game to attempt to display its graphics when restarted despite being instructed not to. The installation of TORCS used in this project did not allow access to its source code, but the solution is explained in a SourceForge forum post [39]. This meant that the game would need to be reopened at every restart. The Gym-TORCS repository [36] uses a similar process, less frequently to solve a memory leak bug.

To run the game with graphics, run wtorcs.exe separately and connect to the server using the Client_w_GUI class (see client_w_gui.py). This was utilised to view an agent's performance rather than solely the training metrics.

5.1.3. Crashing Issues

Since the client code was adapted from snakeoil, identifying and resolving bugs was a long but necessary process. Several crash issues emerged during experiments, each of which were quite rare, halting training and resulting in hours of wasted training time. This section details each issue and its resolution.

The first issue occurred when the client attempted to obtain inputs from the server and discovered that there was no socket available. This occurred in the client's get_server_inputs( ) method, causing the client to enter an infinite loop while searching for socket data. When a try-except block detects this error, the solution is to close the socket and reconnect to the server.

Another issue was sometimes the client would give up trying to create a connection if the game took too long to identify the connection. This was solved by extending the wait time for the identification, which doesn't seem to have any computational consequences.

Finally, a handful of PPO models would accidentally finish the race and cause the game to shut down. The shutdown request from the server would end the client's execution and no new instance of the game was created. The learning algorithm did continue to learn but with 1-timestep episodes of minuscule rewards, ruining the training session. The solution is to reopen the game via the command and create a new connection when the client receives the ***shutdown*** string in the socket data in

get_server_inputs( ). The actual cause was later determined to be the car taking too much damage ending the race, rather than finishing all the laps.

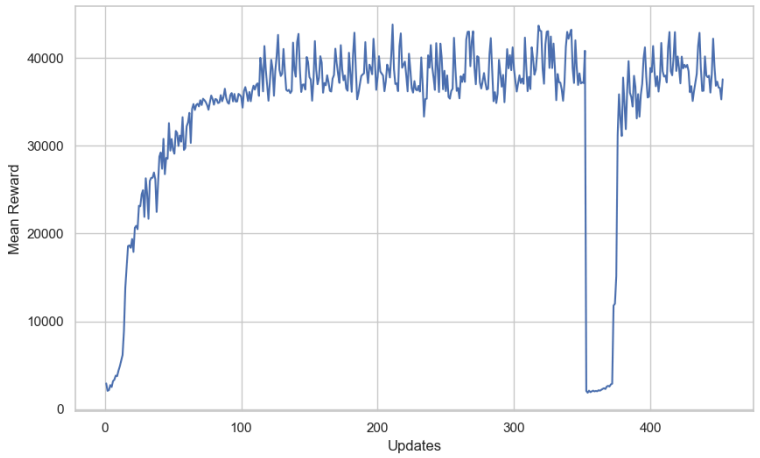


Figure 26. Shows the mean reward of each policy update for the model where the crash occurred

The significant drop in rewards shown in Figure 26 indicates where the client crash occurred, while the improved rewards afterward are the result of an additional training session following the fix.

Once these issues were resolved, training would run uninterrupted and did not need human supervision.

5.2. Data Analysis

Two tools were created to analyse the sensors and actuators of TORCS. Each tool is an additional window that can be shown when demonstrating an agent's performance. The windows run on separate threads and update automatically so as not to interfere with server communication. Both can be enabled/disabled separately using two boolean values in the Client class. Each window and its elements are created upon running the client, staying open until runtime is terminated, usually after a successful test of an agent.

The Tkinter library was used for both windows, with minimal documentation available here [40].

5.2.1. Statistics Window

The first tool is the sensor/actuator data visualiser (see stats_window.py). The window is split into six cells, which each hold a group of sensors or actuators.

The window execution is solely handled by the StatsWindow class, which holds the instance of the window and copies of both the ServerState and DriverAction dictionaries from the Client class. At every timestep, the dictionary copies are updated by the client, but the window itself is only updated 20 times per second, using Tk.after( ). These updates do not recreate each element in the window, just update the values or moving parts of any dials/bars.

Each of the six cells is a separate *Frame* instance, consisting of multiple other frames, each holding separate collections of statistics. This gives a recursive grid layout to hold and scale everything as

intended. Dials use the desired value from the dictionary as an angle combined with some trigonometry to find the endpoint coordinates of the needle from its base. Bars normalise the value from the dictionary and scale it to the needed size so that each is comparable to similar bars.

5.2.2. Track Drawing Window

To draw the track for the second tool (see track_window.py), a coordinate system is established with the race start point set to (0, 0). This utilises the car's *x* and *y* speeds, as well as an arbitrary angle based on the track axis and steering angle, to calculate a displacement vector from the previous timestep to the current one. The displacement is added to the old coordinates to find the new position of the car. The *track* sensor data is then drawn from this point as small dots.

The driving agent should stay within the track boundaries and follow the track axis for this to function properly. The window displays the drawing in real-time, updating 20 times per second.

5.3. Basic Agent

The original snakeoil client included an example drive( ) method. This took the dictionaries for the client's sensors and actuators as arguments and returned the new actuator dictionary after it had been modified for the action decision. The method was adapted into the neural network, keeping the main framework but updating the way driving decisions were made.

The example agent also included automatic traction control and gearing systems. The automatic traction control decreases the *accel* actuator value when the back two wheels are spinning at a higher speed than the front wheels, using the *wheelSpinVel* sensor. The automatic gearing system utilises TORCS's automatic clutch and just changes the gear value based on the *xSpeed*.

5.4. Genetic Algorithm

This section details each component of the genetic algorithm separately, which are combined to explain the loop for each generation in Section 5.4.7. For any code detailed in this section, other than the neural network, see training_multi.py or training_single.py.

5.4.1. Neural Network

The neural network architecture was created using NumPy, which was a simple implementation that could be easily manipulated. NumPy can perform matrix multiplication and has built-in activation functions, which can be used to convert the sensor data inputs into actions. GAs don't utilise error backpropagation, so this was sufficient.

The weights and biases for the neural network are initialised from a chromosome which must fit the following rule:

$$chromosome\ length\ =\ n\ inputs\ *\ n\ hids\ +\ n\ hids\ +\ n\ hids\ *\ n\ outs$$

Where each term represents the number of nodes in the input layer (*n_inputs*), the hidden layer (*n_hids*), and the output layer (*n_outs*). This creates a neural network with only one hidden layer, which was sufficient for this project.

Normalisation was required for the particular sensors used as the neural network's inputs, as the sensors have vastly different ranges. These values are multiplied by the input layer's weight matrix after being passed in as a one-dimensional matrix. The result is added to the input layer's bias matrix and is processed by the Tanh activation function. This is repeated for the hidden layer, giving outputs that are within the range of the Tanh function ($-1 \leq x \leq 1$). The outputs map to the *accel*, *brake*, and *steer* actuators. Both the *accel* and *brake* outputs have to be rescaled as these actuators take values between 0 and 1, using the following formula:

$$rescaled\ output = \frac{output}{2} + 0.5$$

The NeuralNet class handles all of these operations, an instance of which creates the agent when training or testing (see neural_net.py).

5.4.2. Initialising the Population

To generate an initial population of 6 chromosomes, 2 chromosomes are filled with only zeroes and the remaining 4 are randomly generated using NumPy's random.uniform( ) [41], achieving chromosomes filled with numbers between -1 and 1. Each chromosome is then run in TORCS to achieve a fitness value.

Chromosomes and their corresponding fitness are saved in a two-dimensional array and are returned to the main loop.

5.4.3. Testing a Chromosome

To run a chromosome in TORCS, a NeuralNet instance is created using the chromosome. A loop runs the neural network agent for up to 9,000 timesteps roughly 3 minutes. The loop consists of getting the server inputs, passing these inputs into the neural network, receiving actions from the neural network, and sending the actions to the game server via the client. The loop can end early due to early-stopping criteria. These criteria include when the agent's *xSpeed* is below 20 km/h or when the *trackPos* sensor is greater than 1. However, there is a 400-timestep delay where the agent is immune to these criteria.

5.4.4. Crossover

To perform crossover and generate children, pairs of parents are selected randomly from the population. After a parent has been used to create children, it is removed from the pool of parents. For this genetic algorithm, single-point crossover was used, where the point is chosen randomly, excluding the start and end indices of a chromosome. Then the two children are generated by slicing each parent and combining the opposite segments. Once all parents have been used for this task, only the children are returned to the main loop and progress to the next step, mutation.

5.4.5. Mutation

For each child, a random array in the shape of a chromosome is sampled from a NumPy random.normal( ) [42] distribution, where the mean is 0 and the standard deviation is 0.1. This array is then added to the child chromosome to mutate it. Once all children are mutated, they are returned to the main loop and are tested for their fitness value.

5.4.6. Saving

At the end of each generation, the state of training is saved. Using NumPy's savetxt( ) function [43], the population's chromosomes and fitnesses are saved so that the algorithm is crashproof and allows for batch training. The best and average fitnesses of the population are also saved to be used to visualise the training in graphs.

5.4.7. Generational Loop

Before the loop begins, the initial population is created. For a multi-individual population, the loop consists of the following:

1. Perform crossover to create 6 children
2. Mutate each child
3. Evaluate the fitness of each child by running each sequentially
4. Select the fittest 6 individuals from the total 12 population
5. Display the generation number and training duration, along with the best, average, and worst fitness values
6. Save the current population's chromosomes and the training metrics of the generation

For a single-individual population, the loop is simpler and consists of the following:

1. Perform mutation to create a child
2. Evaluate the fitness of the child
3. Compare the two chromosomes and keep the one with the best fitness
4. Display the generation number and training duration, along with the best fitness value
5. Save the current chromosome and the best fitness

5.5. Reinforcement Learning

This section details all the implementations needed for the reinforcement learning research in this project.

5.5.1. Custom Gymnasium Environment

Using the gym_torcs environment as a base, the TorcsEnv class was created to house the reinforcement learning environment that the PPO algorithm used for the states and rewards. The original needed plenty of modification, as it was unnecessarily complicated and had large sections of

code that were useless to the scope of this project. TorcsEnv was much simpler, making research easier.

This environment includes the action space for the PPO models, which ensures that the action space is continuous and within the correct boundaries.

An OpenAI Gym [44] environment must have two methods defined to operate. The first, step( ), is called every timestep and takes the model's action as an argument. This function should enact the actions and determine the reward from the new state. It also should decide when to reset the game. The second function, reset( ), is called to reset the game and any variables for the next training episode.

Upon testing TorcsEnv with Stable Baselines3's PPO, it was made apparent that to be compatible, TorcsEnv's API needed to switch to, the more up-to-date fork of OpenAI's Gym, Gymnasium [45]. This had small consequences on the parameters and returns of the step( ) and reset( ) methods, which were an easy fix.

5.5.2. RandomBrakeWrapper

This class extends Gymnasium.Wrapper [46] and is used to wrap the TorcsEnv environment to allow for modifying the model's actions. RandomBrakeWrapper only has two methods, step( ) and reset( ). step( ) is invoked upon a timestep, taking the action from the model and, if a random number is below 0.1, changing the brake value to 0.2, then passing the modified action to the step( ) function in TorcsEnv. It also counts each timestep to act as a timer for the random brake effect, where the random brake can only take effect after 500 timesteps. reset( ) simply calls the reset( ) method in TorcsEnv.

5.5.3. Data Collection

To save the main rollout metrics of PPO, a logger must be used. The following are the metrics the logger saves:

- *ep_rew_mean*: The mean reward of episodes
- *ep_len_mean*: The mean length of episodes in seconds
- *iterations*: The policy update iteration number
- *total_timesteps*: The number of timesteps elapsed since the start of training
- *time_elapsed*: The time elapsed since the creation of the logger
- Others include the loss from certain steps of the policy update, the clip range, and the fps

To save the PPO models and some custom metrics during training, a custom callback was created, which derives from StableBaselines3's BaseCallback [47]. This made training crash-proof, as at the end of every rollout, the model was saved to be loaded for further training.

When creating a callback instance, it takes file paths for saving metrics and models. In the constructor, these can be used to initialise the files if they do not already exist, or to load certain values from the old files to continue updating the existing files.

At every timestep (in the _on_step( ) function), the attribute *locals* is checked for the end-of-episode boolean value. If true, the time taken for the episode is saved to a .csv file and the *n_episodes* attribute is incremented. This function and the *locals* attribute can be slightly unreliable, usually due to

returning a 0 reward when it shouldn't. The _on_step( ) function increments the *n_calls* attribute when it is called, so there is no need to increment it again.

At a rollout end (in the _on_rollout_end( ) function), the logger is checked for its *ep_rew_mean*, which is compared to the *best_reward* attribute. This is used to save the "best" model so far, which is also quite unreliable. So, the model is also saved at a rollout end, acting as the last model for loading or testing. This function also calculates the number of episodes in the policy update and saves them to a .csv, using the *n_episodes* attribute.

5.5.4. PPO

The ppo.py file was used to run all training with PPO, and everything stems from there. This file includes multiple functions for different PPO experiment types, these are as follows:

- ppo( ) is the regular method that trains an agent from scratch.
- ppo_w_grid( ) performs a grid search, training many models with different hyperparameters sequentially. This was used to find a set of hyperparameters that were "optimal". Each model trained here is saved under a different name, corresponding to the permutation number.
- ppo_w_random( ) is equivalent to ppo( ) but wraps TorcsEnv in the RandomBrakeWrapper to train with the stochastic brake enabled.
- ppo_w_load( ) loads a previously trained agent and continues to train it, including any model originally trained in the grid search. This means that no parameters of the model can be changed. However, TorcsEnv can be modified here if the reward function or early-stopping criteria need changing to promote a different behaviour. Also, the RandomBrakeWrapper can be included if needed. The model loading uses the built-in load( ) from StableBaselines3's PPO.

The following details the main method used for training a PPO model:

1. Create an instance of the Gym/Gymnasium environment
2. Define a name for the model for easy recall and graph representation
3. Save the hyperparameters, reward function, and other training comments to a file
4. Create a logger to save the rollout metrics
5. Create the PPO model and set the desired hyperparameters
6. Assign the logger to the model
7. Create a callback instance for saving custom training metrics
8. Set the model to learn for a desired number of timesteps and pass in the callback

6. Conclusions

As shown by the vast research conducted, TORCS is an excellent environment to work with and allows for insight into how to teach AI driving agents correct behaviours for racing safely and efficiently. Agents can learn two distinct behaviours. The first is to follow the track axis, which keeps the agent on the track indefinitely but is majorly inefficient for racing. The second is to take the racing line, cutting corners to preserve speed and finish laps faster. However, the second is very situational and volatile, usually resulting in the agent losing control and coming off the track. Neither algorithm can learn to brake efficiently to adjust speed for difficult corners, even when reward functions are used to promote braking or using a stochastic brake. Despite this, both behaviours are positive and need merging, along with effective use of the brake, to make an agent capable of rivalling humans.

Both algorithms have major advantages and disadvantages. The genetic algorithm is good at training agents with simpler architectures which yield simpler behaviours in a short time, and even complex agents given significant knowledge of TORCS. The PPO algorithm creates agents with complex architectures, making training significantly more difficult to conduct correctly but showing more complex behaviours. PPO has significantly more parameters and conditions that need tuning and plenty of thought required to create agents that can race effectively. The genetic algorithm shows the effectiveness of random numbers for machine learning tasks but has less potential for expansive learning than reinforcement learning, which employs trial and error.

The genetic algorithm has generated significantly better final agents than the PPO algorithm. These agents show similar behaviours to those trained by PPO but are more reliable. To support this claim, ten experiments for each final agent were conducted to collect the distance raced within 3 minutes. The experiments were stopped early if the agent strayed from the track.

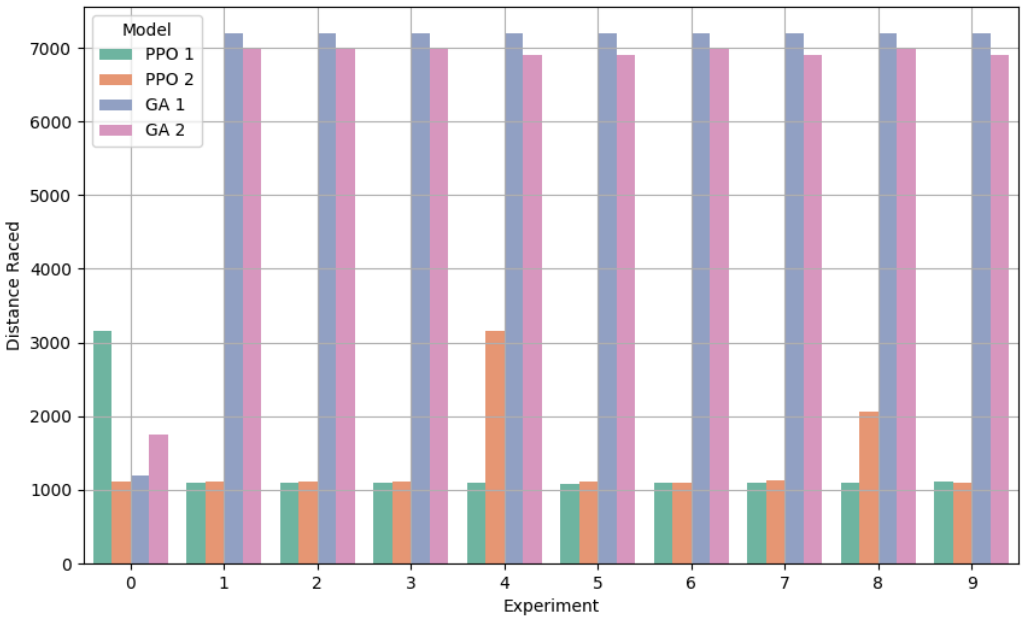


Figure 27. Compares the distance raced in ten experiments with each agent

In Figure 27, PPO 1 refers to the agent trained with the modified reward function, PPO 2 refers to the agent trained with the original reward function, GA 1 refers to the agent trained with the original fitness function, and GA 2 refers to the agent trained with the modified fitness function. This clearly shows that the genetic algorithm agents get significantly further in a race than the PPO agents, which

usually go out of bounds before finishing a lap. Also, the consistency in the results show the reliability of the agents and that the agents reached the 3-minute limit.

So, the best agent created in this project was trained using a genetic algorithm that used a simple fitness function, that returns the distance raced. This agent's neural network has an architecture of 23 inputs, 36 hidden neurons, and 3 outputs. This agent was trained for 120 generations as a part of a 24-individual population. This agent always takes the racing line while keeping enough distance from the track edge to not go out of bounds, and maintains a steady top speed of 140km/h, driving similarly to a human. Despite this agent showing good racing behaviours, it cannot speed up for straight sections and slow down for corners like a human player. This means that human players usually beat this agent. The genetic algorithm trains agents that are overfitted to the track they are trained on, where they find an optimal speed that won't cause them to drift off the track, so are far from optimal.

Due to being from the car's perspective, the inputs/outputs discussed in Sections 4.3.3 and 4.3.4 can easily be replicated in the real world, where a robot car could adapt the models in this thesis to drive. A track axis and edge would need to be defined and the car would need appropriate laser range finders. This suggests that machine learning for autonomous systems could benefit from using virtual simulations to explore and enhance concepts such as high-speed decision-making. However, a real-world counterpart comes with the added problem of sensor noise, as all research conducted in this project was noiseless.

In reinforcement learning, designing a good reward function is more than just promoting the main desired behaviours. It also must allow secondary behaviours to be learned that are less important but are needed for a complex agent. For example, while the reward function should encourage driving fast and using the track's width effectively, it should also allow agents to learn that braking for corners can be beneficial. However, the challenge is that some desirable behaviours may reduce rewards. As a result, agents might ignore these behaviours, limiting their performance (as shown by the reward functions detailed in Section 4.5.1.2). This shows that a more complex reward function should take the secondary behaviours into account, without being too explicit with what behaviours are promoted as micro-managing can have opposite effects. With this understanding, if further work is conducted after this project, producing and tweaking an optimal reward function should be prioritised. This would result in agents that can drive indefinitely on any track, accelerate for straight sections, and brake for difficult corners. From there, including opponent detection would allow agents to compete with human players or other AI agents. Also, experiments with other reinforcement learning algorithms, such as DDPG, may yield better results.

Appendices

Appendix 1 - Links

The following are links to the repositories where code/results can be found:

GitLab: 23-24 CE901-CE911-CF981-SU / 23-24_CE901-CE911-CF981-SU_berry_drew_a_j · GitLab (essex.ac.uk)

GA 1 (uses original fitness function): videos/GA_1.mp4 · master · 23-24 CE901-CE911-CF981-SU / 23-24_CE901-CE911-CF981-SU_berry_drew_a_j · GitLab (essex.ac.uk)

GA 2 (uses modified fitness function): videos/GA_2.mp4 · master · 23-24 CE901-CE911-CF981-SU / 23-24_CE901-CE911-CF981-SU_berry_drew_a_j · GitLab (essex.ac.uk)

PPO 1 (uses modified reward function): videos/PPO_1.mp4 · master · 23-24 CE901-CE911-CF981-SU / 23-24_CE901-CE911-CF981-SU_berry_drew_a_j · GitLab (essex.ac.uk)

PPO 2 (uses original reward function): videos/PPO_2.mp4 · master · 23-24 CE901-CE911-CF981-SU / 23-24_CE901-CE911-CF981-SU_berry_drew_a_j · GitLab (essex.ac.uk)

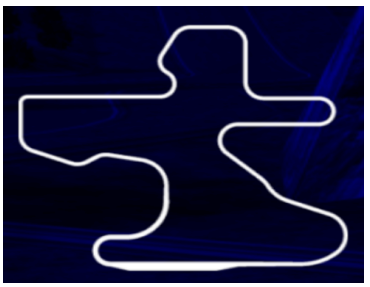
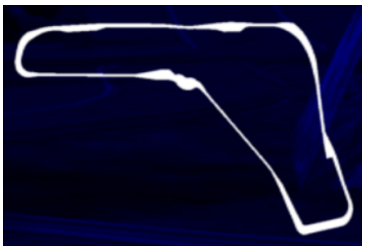
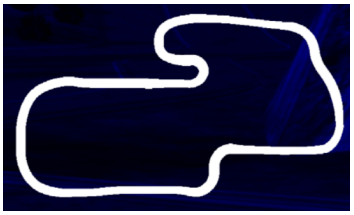
Appendix 2 - PPO Network Architecture

The following is the raw representation of the PPO action/value network architecture:

```
ActorCriticPolicy(
  (features_extractor): FlattenExtractor(
    (flatten): Flatten(start_dim=1, end_dim=-1)
  )
  (pi_features_extractor): FlattenExtractor(
    (flatten): Flatten(start_dim=1, end_dim=-1)
  )
  (vf_features_extractor): FlattenExtractor(
    (flatten): Flatten(start_dim=1, end_dim=-1)
  )
  (mlp_extractor): MlpExtractor(
    (policy_net): Sequential(
      (0): Linear(in_features=23, out_features=64, bias=True)
      (1): Tanh()
      (2): Linear(in_features=64, out_features=64, bias=True)
      (3): Tanh()
    )
    (value_net): Sequential(
      (0): Linear(in_features=23, out_features=64, bias=True)
      (1): Tanh()
      (2): Linear(in_features=64, out_features=64, bias=True)
      (3): Tanh()
    )
  )
  (action_net): Linear(in_features=64, out_features=3, bias=True)
  (value_net): Linear(in_features=64, out_features=1, bias=True)
)
```

Appendix 3 - Track Images

The following images are the layouts of each track used in the experiments of this project.

		
aalborg	alpine-1	alpine-2
		
corkscrew	e-track-1	e-track-2
		
e-track-3	forza	g-track-1
		
g-track-3	ruudskogen	street-1

Appendix 4 - Example .xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE params SYSTEM "params.dtd">

<params name="Practice">
	<section name="Header">
		<attstr name="name" val="Practice"/>
		<attstr name="description" val="Practice"/>
		<attnum name="priority" val="100"/>
		<attstr name="menu image" val="data/img/splash-practice.png"/>
		<attstr name="run image" val="data/img/splash-run-practice.png"/>
	</section>

	<section name="Tracks">
		<attnum name="maximum number" val="1"/>
		<section name="1">
			<attstr name="name" val="g-track-3"/>
			<attstr name="category" val="road"/>
		</section>
	</section>

	<section name="Races">
		<section name="1">
			<attstr name="name" val="Practice"/>
		</section>
	</section>

	<section name="Practice">
		<attnum name="laps" val="20"/>
		<attstr name="type" val="practice"/>
		<attstr name="starting order" val="drivers list"/>
		<attstr name="restart" val="yes"/>
		<attstr name="display mode" val="normal"/>
		<attstr name="display results" val="yes"/>
		<attnum name="distance" unit="km" val="0"/>
		<section name="Starting Grid">
			<attnum name="rows" val="1"/>
			<attnum name="distance to start" val="100"/>
			<attnum name="distance between columns" val="20"/>
			<attnum name="offset within a column" val="10"/>
			<attnum name="initial speed" unit="km/h" val="0"/>
			<attnum name="initial height" unit="m" val="0.2"/>
		</section>
	</section>
```

```
        <section name="Drivers">
                <attnum name="maximum number" val="1"/>
                <attstr name="focused module" val="human"/>
                <attnum name="focused idx" val="1"/>
                <section name="1">
                        <attnum name="idx" val="0"/>
                        <attstr name="module" val="scr_server"/>
                </section>
        </section>

        <section name="Configuration">
                <attnum name="current configuration" val="4"/>
                <section name="1">
                        <attstr name="type" val="track select"/>
                </section>

                <section name="2">
                        <attstr name="type" val="drivers select"/>
                </section>

                <section name="3">
                        <attstr name="type" val="race config"/>
                        <attstr name="race" val="Practice"/>
                        <section name="Options">
                                <section name="1">
                                        <attstr name="type" val="race length"/>
                                </section>
                                <section name="2">
                                        <attstr name="type" val="display mode"/>
                                </section>
                        </section>
                </section>
        </section>
</params>
```

References

1. CE902 MSc Project Proposal proposal and background work/2312089_Project_Proposal.pdf · master · 23-24 CE901-CE911-CF981-SU / 23-24_CE901-CE911-CF981-SU_berry_drew_a_j · GitLab (essex.ac.uk)
2. 'TORCS - The Open Racing Car Simulator'. SourceForge, 6 Feb. 2020, https://sourceforge.net/projects/torcs/
3. TORCS Robot Tutorial. https://www.berniw.org/tutorials/robot/tutorial.html. Accessed 20 Mar. 2024 (this link is now down but was used for the proposal [1])
4. Loiacono, Daniele, et al. Simulated Car Racing Championship: Competition Software Manual. arXiv:1304.1672, arXiv, 29 Apr. 2013. arXiv.org, https://doi.org/10.48550/arXiv.1304.1672
5. 'Torcs'. Pdfcoffee.Com, https://pdfcoffee.com/torcs-pdf-free.html. Accessed 23 Aug. 2024
6. Speed Dreams - Free and Open Source Motorsport Simulator. https://www.speed-dreams.net/#about. Accessed 23 Aug. 2024.
7. Free Games for Linux | Linux Journal. https://www.linuxjournal.com/content/free-games-linux. Accessed 23 Aug. 2024.
8. 'History and Evolution of Machine Learning: A Timeline'. WhatIs, https://www.techtarget.com/whatis/A-Timeline-of-Machine-Learning-History. Accessed 23 Aug. 2024.
9. Machine Learning Textbook. https://www.cs.cmu.edu/~tom/mlbook.html. Accessed 23 Aug. 2024.
10. LeCun, Yann & Bengio, Y. & Hinton, Geoffrey. (2015). Deep Learning. Nature. 521. 436-44. 10.1038/nature14539. (PDF) Deep Learning (researchgate.net). Accessed 23 Aug. 2024.
11. What Is a Neural Network? | IBM. 6 Oct. 2021, https://www.ibm.com/topics/neural-networks. Accessed 23 Aug. 2024.
12. Alam, Tanweer & Qamar, Shamimul & Dixit, Amit & Benaida, Mohamed. (2020). Genetic Algorithm: Reviews, Implementations, and Applications. 10.20944/preprints202006.0028.v1. (PDF) Genetic Algorithm: Reviews, Implementations, and Applications (researchgate.net). Accessed 23 Aug. 2024.
13. Simple Genetic Algorithm - an Overview | ScienceDirect Topics. https://www.sciencedirect.com/topics/computer-science/simple-genetic-algorithm. Accessed 23 Aug. 2024.
14. Backpack Problem | Brilliant Math & Science Wiki. https://brilliant.org/wiki/backpack-problem/. Accessed 23 Aug. 2024.
15. Brooks, Ruth. 'What Is Reinforcement Learning?' University of York, 20 Dec. 2021, https://online.york.ac.uk/what-is-reinforcement-learning/.
16. Part 1: Key Concepts in RL — Spinning Up Documentation. https://spinningup.openai.com/en/latest/spinningup/rl_intro.html. Accessed 23 Aug. 2024.
17. An introduction to Reinforcement Learning. YouTube, https://youtu.be/JgvyzIkgxF0?si=cH-9sLvV1eeeeqYo. Accessed 23 Aug. 2024.
18. Deep Reinforcement Learning: Pong from Pixels. https://karpathy.github.io/2016/05/31/rl/. Accessed 23 Aug. 2024.
19. Sutton & Barto Book: Reinforcement Learning: An Introduction. http://incompleteideas.net/book/the-book-2nd.html. Accessed 23 Aug. 2024.
20. An introduction to Policy Gradient methods - Deep Reinforcement Learning. YouTube, https://youtu.be/5P7I-xPq8u8?si=ofUggKZd0buXc5Jb. Accessed 23 Aug. 2024.

21. Policy Gradient Methods for Reinforcement Learning with Function Approximation. 464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf (neurips.cc). Accessed 23 Aug. 2024.
22. The Definitive Guide to Policy Gradients in Deep Reinforcement Learning: Theory, Algorithms and Implementations. Ar5iv, https://ar5iv.labs.arxiv.org/html/2401.13662. Accessed 23 Aug. 2024.
23. Karunakaran, Dhanoop. 'REINFORCE — a Policy-Gradient Based Reinforcement Learning Algorithm'. Intro to Artificial Intelligence, 8 June 2020, https://medium.com/intro-to-artificial-intelligence/reinforce-a-policy-gradient-based-reinforcement-learning-algorithm-84bde440c816.
24. Policy Gradient vs. Value Function Approximation: A Reinforcement Learning Shootout. tr-06-001.pdf (mcgovern-fagg.org). Accessed 23 Aug. 2024.
25. Doshi, Ketan. 'Reinforcement Learning Explained Visually (Part 6): Policy Gradients, Step-by-Step'. Medium, 24 Apr. 2021, https://towardsdatascience.com/reinforcement-learning-explained-visually-part-6-policy-gradients-step-by-step-f9f448e73754.
26. Schulman, John, et al. Proximal Policy Optimization Algorithms. arXiv:1707.06347, arXiv, 28 Aug. 2017. arXiv.org, https://doi.org/10.48550/arXiv.1707.06347.
27. Schulman, John, et al. Trust Region Policy Optimization. arXiv:1502.05477, arXiv, 20 Apr. 2017. arXiv.org, https://doi.org/10.48550/arXiv.1502.05477.
28. Proximal Policy Optimization. OpenAI, https://openai.com/index/openai-baselines-ppo/. Accessed 23 Aug. 2024.
29. Weng, Lilian. Policy Gradient Algorithms. 8 Apr. 2018, https://lilianweng.github.io/posts/2018-04-08-policy-gradient/.
30. PPO — Stable Baselines3 2.4.0a8 Documentation. https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html. Accessed 23 Aug. 2024.
31. Virtual robotic car racing with Python and TORCS. YouTube, https://youtu.be/BGOtqXA_y1E?si=1xF3-thLNAXuifHn. Accessed 23 Aug. 2024.
32. Using Keras and Deep Deterministic Policy Gradient to Play TORCS | Ben Lau. https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html. Accessed 23 Aug. 2024.
33. 'Torcs---Reinforcement-Learning-Using-Q-Learning/MSc_RL_Thesis.Pdf at Master · A-Raafat/Torcs---Reinforcement-Learning-Using-Q-Learning'. GitHub, https://github.com/A-Raafat/Torcs---Reinforcement-Learning-using-Q-Learning/blob/master/MSc_RL_Thesis.pdf. Accessed 23 Aug. 2024.
34. "Collision Detection and Overtaking Using Artificial Potential Fields in Car Racing game TORCS using Multi-Agent based Architecture" Thesis, https://www.diva portal.org/smash/get/diva2:830507/FULLTEXT01.pdf. Accessed 23 Aug. 2024.
35. 'Imitation-Dagger/README.Md at Master · Avisingh599/Imitation-Dagger'. GitHub, https://github.com/avisingh599/imitation-dagger/blob/master/README.md. Accessed 23 Aug. 2024.
36. "Gym-TORCS." GitHub, https://github.com/ugo-nama-kun/gym_torcs. Accessed 23 Aug. 2024.
37. SnakeOil. https://xed.ch/p/snakeoil/. Accessed 23 Aug. 2024.
38. 'Subprocess — Subprocess Management'. Python Documentation, https://docs.python.org/3/library/subprocess.html. Accessed 23 Aug. 2024.
39. TORCS - The Open Racing Car Simulator / Discussion / Help: Headless Torcs Crashes at Restart. https://sourceforge.net/p/torcs/discussion/11282/thread/2963ff7f/. Accessed 23 Aug. 2024.

40. ‘Tkinter — Python Interface to Tcl/Tk’. Python Documentation, https://docs.python.org/3/library/tkinter.html. Accessed 23 Aug. 2024.
41. Numpy.Random.Uniform — NumPy v2.1 Manual. https://numpy.org/doc/stable/reference/random/generated/numpy.random.uniform.html. Accessed 23 Aug. 2024.
42. Numpy.Random.Normal — NumPy v2.1 Manual. https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.html. Accessed 23 Aug. 2024.
43. Numpy.Savetxt — NumPy v2.1 Manual. https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html. Accessed 23 Aug. 2024.
44. Gym. GitHub, GitHub - openai/gym: A toolkit for developing and comparing reinforcement learning algorithms. Accessed 23 Aug. 2024.
45. Gymnasium Documentation. https://gymnasium.farama.org/index.html. Accessed 23 Aug. 2024.
46. Gymnasium Wrapper Documentation. https://gymnasium.farama.org/api/wrappers.html. Accessed 23 Aug. 2024.
47. Callbacks — Stable Baselines3 2.4.0a8 Documentation. https://stable-baselines3.readthedocs.io/en/master/guide/callbacks.html. Accessed 23 Aug. 2024.