Entertainment through Deep Reinforcement Learning on Gran Turismo for the PlayStation

Nadir Zaman Syed

MSc in Computer Science
The University of Bath
2023

Entertainment through Reinforcement Learning on Gran Turismo for the PlayStation

Submitted by: Nadir Zaman Syed

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in Computer Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

This works soughent to develop an interface between a Python environment and the Gran Turismo racing game running on an unmodified PlayStation emulator. The interface allows real-time observation of the game in both visual and parametric values. For the goal of training an agent through reinforcement learning to play Gran Turismo in time-trial on the High Speed Ring track. Work involving complex retro-games with 3D environment such as Gran Turismo, treated as a real-time problem using reinforcement learning is an approach not found in literature.

To achieve these goals, it was necessary to reverse engineer Gran Turismo sufficiently to expose necessary parameters to aid the agent to interpret the game environment state sufficiently to satisfy the Markov Decision Process criteria. Parameters needed to be found through various reverse engineering. Parameters are transmitted to the Python environment over a TCP socket through Protobuf encoding by Lua scripts that run on the emulator PCSX-Redux. The agent was then training using the Soft-Actor-Critic (SAC) method. The agent is rewarded for progressing through synthetically generated checkpoints along the track. The agent performs actions by means of a virtual gamepad utilising both continuous, or discrete actions, without alterations to the SAC algorithm. The design is such that the agent's actions are acting on the environment in real-time without pausing the emulator or skipping an exact number of frames.

The ideal setup was found to be when the agent was able to have receive the game screen as four stacked greyscale images of 128 by 128 pixel combined with vehicle parameters. Under ideal conditions, the agent was able to achieve a lap-time of 63.137 seconds, over half a second faster than expert human performance.

Contents

List of Figures

List of Tables

Acknowledgements

Thanks to my supervisor Benjamin Ralph for his encouragement, advice, and guidance, not only in this work but also throughout my studies at the University of Bath for the modules that set the foundation for artificial intelligence which inspired my love for the subject.

Thanks to Nicolas Noble, the main author of PCSX-Redux as well as the rest of the maintainers of the project for providing the emulator, prioritising bug-fixes and feature requests required to conduct this work.

Thanks to Yann Bouteiller in sharing their reinforcement learning experience on Track Mania and allowing themselves to be available to bounce-off ideas and understand their code so that it can be adapted for my work.

Thanks to Leo Jimenez (known as Submaniac in the Gran Turismo game-modification community) for helping me confirm certain in-game memory addresses while reverse engineering, as well as providing a public replay of themselves driving an MR2 on the high-speed ring.

Thanks to the individual known as Squaresoft74 for without their help to figure out the decompression algorithm in MIPS assembly code for Gran Turismo, I would not have been able to make any headway in reverse engineering of the game.

Thanks to Daniel Stolpmann who supported me to understand the Ray RLlib API for my early work and for their moral support and encouragement.

Many thanks to the many more individuals have supported me through motivation, or technical support along the way, or simply conducted research or application work in a similar domain, for which their contribution has allowed me to base my work upon.

I want to thank my wife and kids, who had to tolerate my absence not only for my day-job and regular business trips, but also for allowing me the time to focus on this project. I thank my wife for encouraging me to pursue an MSc in Computer Science as opposed to my original intention to study an MSc in Psychology.

Finally, I want to thank Polyphony Digital. Their work on Gran Turismo for the PSX, inspired me to shift my career focus to the real automotive world, their release of Gran Turismo 6 inspired me to take on this Masters in Computer Science at the University of Bath, and my failed interview motivated me to take on this project topic.

Chapter 1 Introduction

Artificial Intelligence (AI) is a very prominent topic as of January 2023, GPT-3, Dall-E and Midjourney are three examples of well projects that are sparking public moral and legal debate in the subject, most recently in the field of art, where some question if these tools are replacing artists or perhaps destroying art itself (Goudarzi, 2023). AI is also being used to compete against humans.

Use of AI to beat or surpass humans has been pursued for a long time, one of the earliest popular successes was that of IBM's Deep Blue II famous victory over Garry Kasparov in 1997 (Hsu, 1999). Deep Blue II and other similar AI of the time often utilised search algorithms (Campbell, Hoane and Hsu, 2002) which made retrospective understanding of the agent's decisions possible (Campbell, Hoane and Hsu, 2002; Rogers, 2019). Modern achievements continue to be made with recent examples such as AlphaGo's defeat of Ke Jie in 2017 (Rogers, 2019), Sony AI's Sophy, was able to out-perform top human racers (Wurman et al., 2022) and OpenAI Five that was able to defeat the world champions at an esports event in 2019 (Berner et al., 2019).

AlphaGo and Sophy captivate interest through their remarkable ability to perform at either a human or super-human level in exceedingly intricate games. Notably, the game of Go, characterized by an average of 250 possible moves per turn, surpasses the complexity of chess, rendering conventional search algorithms ineffective (Levinovitz, 2014). Whereas the environment of motor-racing can be said to contain near infinite number of possible states, when one considers the various vehicle parameters. A modern racing game such as that of Gran Turismo Sport (GTS) may be discrete by nature of running on a game-console, however the complexity shares similarities with real racing, due to the similar continuous bounded parameters such as track position in 3-dimensions (3D), with different possible velocity and acceleration vectors for each vehicle. Sony's AI for GTS, Sophy engages in competitive racing against human players in such a complex environment even amid the added complexity of imperfect human players on track simultaneously (Wurman et al., 2022). This is a demonstration of the power of modern reinforcement learning (RL) methods.

This study shares a comparable focus to that of Sophy by concentrating on both the technical implementation and reinforcement learning (RL) methods to train an AI agent for time-trial races in an early 3D racing game – Gran Turismo (GT). GT is a 3D-based racing game environment released in December 1997 in Japan (Sony, 2008), featuring advanced physics in contrast to prior games of the genre. The game is world-renown and has old 10 million copies worldwide (Sony, 2008). In contrast to prevalent literature, this research uniquely approaches this retro-gaming AI challenge as a real-time robotics environment. This deliberate choice aims to enhance the transferability of the findings to diverse applications and enable application in games featuring time-step fluctuations, including online gaming. The engineering approach therefore also hat to fit the requirements of a real-time method, making the combination of technical implementation and RL methodology novel.

The work did not make use of a custom emulator such as work done on an earlier PSX RL study(Purves et al., 2019) but rather relies on a modern emulator that still receives daily updated. The emulator instead interacts with the Python environment used for RL, in an asynchronous mode over TCP sockets to deliver the observations. This was achieved by means of the developed Lua-scripts to run on the emulator's Lua engine. The common approach of frame-skipping (Severo, 2023b; Mnih et al., 2013) or

emulation pausing (Costa, 2022; Fletcher, 2017) cannot be applied to real-life problems since it is not possible to pause the real-world nor is it possible to skip exact frames. The only possibility is to sample in real-time at some fixed sampling rate with some tolerance on the timing. This is also through for scenarios such as online games, where the latency of the WiFi, internet or game server may fluctuate. For these reasons, this work aimed to develop a technical implementation and agent trained on the premise of real-time conditions.

The research also delved into possible difference occurring from diverse action-spaces that also apply to human players, such as the use of analogue or digital controllers.

To utilise Gran Turismo for this work, a portion of reverse engineering of the game was necessary. Such an endeavour has not been done or published. The main game executable is a decompressing binary that was compiled for the PSX's MIPS processor and lacked any debug labels. The methods for reverse engineering are applicable to other games on the PSX.

The study initially explored the use of RL frameworks such as Ray RLlib and made some initial success applying proximal policy optimization (PPO) to simpler observations. However, due to the limited examples in literature or documentation, as well as the limited timeframe to adapt such frameworks for real-time environment it was necessary to develop a more bespoke pipeline for this research. The RL related code was written in Python using the PyTorch library.

The primary formal motivation for this work on GT was to address the under-performing built-in AI used in GT and its sequel GT2, which limits the value of replaying the game in a single-player mode. The inadequacy of the built-in AI in certain situations, such as veering off the track or experiencing spin-offs are often subject of ridicule in online content (Joe Fox, 2018; MattJ155, 2021). This work would also serve as a foundation for acquiring knowledge on the inner workings of RL-trained AI in modern games, which is also just now emerging in retail games. One such recent example is that of Sophy which has been added to GT 7 as a game feature (Spranger, 2023).

1.1 The Platform

The more detailed justification for focusing on the PSX is provided here. RL on games tend to predominantly focus on two ends of twhichaming spectrum, very primitive 2D games such those of Atari (Mnih et al., 2013), or latest-generation games such as StarCraft II (Vinyals et al., 2017), of which the latter tends to be developed by large teams.

In the case of games intended for computers that were optimised for 2D games, there are several well documented projects on the retrospect use of AI in substitute of a human player in videogames. An example of such work applies convolution networks and Q-learning algorithms to train an agent to play and excel at Atari 2600 games using raw video data (Mnih et al., 2013).

Another recent example of reinforcement learning in retro-gaming is on Steet Fighter II on the Super Nintendo Entertainment System (SNES), which was demonstrated at the Samsung Developer Conference (Fletcher and Mortensen, 2018). The authors utilised custom emulators, speed run tools, the Gym library (Gym is now superseded by Gymnasium) and Python with Keras to develop an agent that was able to achieve an 80% win-rate. Street Fighter II has a significantly faster gameplay in comparison to Atari 2600 games.

The SNES is a dramatic improvement of the graphical capability in comparison to an older Atari 2600, but is still a predominantly a 2D platform, and thus its environments are similarly simple. It was

possible to apply Canny Edge to SNES games with barely any parameter tuning, as shown in Figure 1, which demonstrates that the game highly contrasting colours which make edges unambiguous.

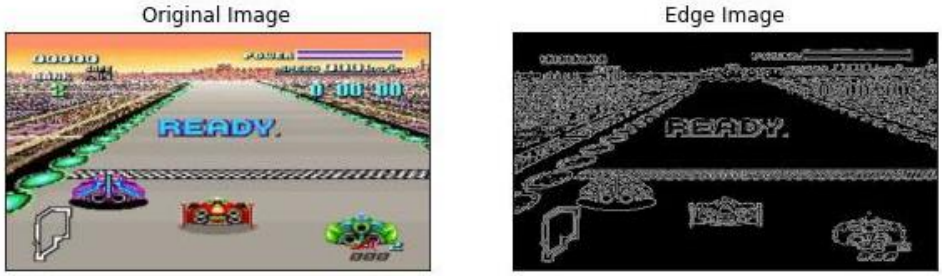


Figure 1 - F-Zero with basic Canny Edge detection

Current generation gaming consoles can render photo-realistic images at 4K in games such as Gran Turismo 7 on the PlayStation 5 (Jupp, 2022). Such consoles are unfortunately economically expensive and cannot yet be emulated. Even if a reliable emulator were to be available, it would most likely not be possible to emulate on an average desktop computer, let alone on the additional computational effort required for the AI RL in parallel.

Modern gaming platforms were also be considered as they permit access to a wide selection of games of varying degree of visual capability and physics accuracy. However, such work has already been done on several racing games such as Track Mania with use of an extension called OpenPlanet (Boyer, 2020; Bouteiller, Geze and GobeX, 2023b), as well as rally-based games such as World Rally Championship 6 (Jaritz et al., 2018). Several driving simulators such as TORCS have also been adapted for the purpose of RL (Cai et al., 2020; Peng et al., 2021; Loiacono et al., 2010; Mnih et al., 2016), however such simulators do not have the wide-entertaining appeal of a racing videogame. In many of these cases, special hidden APIs, custom builds or third-party tools were used to extract some or all of the observation parameters.

Therefore, it was decided to select a platform that had the lowest computational power while being able to provide 3D environments with realistic physics-dynamics, as that would secure the highest ease of emulation, provide a novel research opportunity, and hopefully appeal to a wider audience. The PSX is considered by some to be one of the earliest consoles that democratised such 3D (Hester, 2019), for which a suitable racing game exists. The low cost of the PlayStation hardware and software in the used market means they are more easily accessible, making it feasible for others to purchase the game for such studies without a large economic cost.

The work conducted on GT Sport demonstrated the ability to achieve top-human level capabilities in a complex modern 3D racing game. They claim to have trained an agent able to match the top 95% of human time trial track records within only a few hours of training. The computation power required to achieve such results is not well document (Wurman et al., 2022). The only similar study conducted with Sony's indicates that several PlayStations and computers were utilised for at least one such GT Sport related work (Song et al., 2021; Fuchs et al., 2021). Those authors also required access to GT Sport's proprietary APIs granted by the authors from Sony (Wurman et al., 2022).

1.2 The Game

GT popularised racing games due to its relatively realistic physics and 3D environment, as suggested by its sales record. GT has grown to the extent that it had an official FIA Championship and featured at the 2020 Tokyo Olympics (in 2021 due to the COVID19 pandemic). Therefore, to select a game from the popular series seems like an interesting choice. The age of the game presents primitive 3D graphics by modern standards, which adds to the challenge of this research since it is not pixel-perfect like an Atari game and nor is it photo-realistic like GTS, but rather in between.

Other racing games with similar graphics such as Option Tuning Battle were considered, but ultimately ignored as they are far less renown than GT and would have less potential of growing into a RL community, as has become for TrackMania with the TrackMania Roborace League (Bouteiller, Geze and GobeX, 2023a).

Furthermore, selecting a game such as GT would permit a wider audience to critic or related to this work, as well as being able to compare the performance of agent against other players' lap-times.

1.3 Ethical Consideration

AI has been discussed in relation to ethics and governance in publications such as Forbes (Vander Ark, 2018; Gordon, 2022). Neural networks are said to be of concern by some due their black box nature leading to unexplainable and potentially dangerous decision making when applied to dangerous situations (Rogers, 2019). This is a reasonable concern since it may be hard to understand what a neural network is doing and how it achieves the result it does (Mohamed, 2008; Rogers, 2019), but that is why a virtual racing game is the perfect environment for such development, as there is no risk of harming anyone even if erroneous results were achieved. In addition, this form of study did not make use of any personal data.

The other ethical consideration of this work is the electrical power consumption to perform RL and the CO2 impact. The estimated power consumption of the computers used was around 250~350 Watts each during training. Considering that for much of the training this work required the computers to run continuously, one can estimate 7.2kWh per day per computer. To offset the CO2 impact (as well as the economical one) that may be caused by this work, two 400 Watt-peak photovoltaic panels were installed. This would allow 100% renewal energy consumption during optimum sunshine hours at least, but not enough to cover the total CO2 impact.

1.4 Objectives

Formally the objects are as follows:

1. Develop or adapt a method for a Python environment to observe the GT environment (or observable environment).
2. Develop or adapt a method for a Python environment to be able control the player's vehicle (take actions).
3. Implement and train an AI agent in Python to manoeuvre a vehicle around the High Speed Ring in a time-trial.

These objectives are further described and defined below.

1.4.1 Observation

GT will be run on an emulator running on Windows 10 (my own computer). A method to transfer the necessary relevant in-game parameters and/or images as part of the observation, to a Python environment is necessary. Each set of parameters for a given game step will be referred to as a *frame*.

It is expected that higher order parameters such as acceleration or trajectory will be inferred from the difference between stacked images or stacked parameters.

If necessary, parameters shall be extracted from GT via the emulated memory.

1.4.2 Control

The Python environment will need to control the vehicle in GT via the emulator. This could potentially be achieved by one of the following means:

- A virtual gamepad that has a Python API library.
- A custom pipeline for Python to directly control the gamepad status in the emulator.
- A custom pipeline for the Python environment to directly overwrite the relevant variables in the GT game-engine's memory.

The mandatory controls as part of the control-space would be:

- Steering as Discrete (Digital Control) or Continuous (Analogue Control)
- Accelerator as Discrete or Continuous
- Brake as Discrete or Continuous

The PlayStation's original launch controller only had digital controls, but early in its life cycle it was then replaced by the Dual Shock controller which provided two analogue inputs. Thus, several games were design to be playable by both digital and analogue controls. Analogue mode on GT, would use one analogue input for steering, and one analogue for both the accelerator and brake. In analogue mode it would not be possible to brake and accelerate (as shown on the left side of Figure 2). The ability to brake simultaneously while accelerating is known as left foot braking and is mainly used to transfer vehicle weight towards the front tyres which may be preferable under certain racing conditions (Perkins, 2017).

Figure 2 - Control Settings in GT

For analogue controls, this limitation was respected as it also applies to human players that would have to play this game with the first-generation Dual Shock controller.

1.4.3 Implement an Agent and Training

Having achieved a way of receiving observations and performing actions on the emulated environment, it would then be necessary to implement a neural network that can be trained by means of reinforcement learning (RL) through use of reward or penalty functions.

This would therefore require three features to be implemented:

- A deep neural network.
- A reward function (to reward or penalise the agent during training).
- A reinforcement learning algorithm.

Chapter 2 Literature Review

In this section a literature review is presented in two main categories, the first is from literature that describe the RL branches, and literature that focus on solutions related to relevant games.

2.1 Reinforcement Learning Algorithms

The main elements of reinforcement learning are policy, rewards, value and in some cases a model of the environment (Sutton and Barto, 2018, p. 2). Another essential element is the environment or the observable environment, since in some cases the full environment might not be observable. For the problem proposed in this work, some effort is spent on reviewing what is or should be observable and potentially unobservable in contrast to a human agent. In some cases, it may be that the accuracy of observation differs in comparison to humans, such as measuring tyre slip in a road-car. Driver's may perceive it and estimate it, but not observe it as an exact percentage.

A selection of RL algorithms are chosen from a list provided by OpenAI shown in Figure 3, which organises some of the recently proposed algorithms of several authors (Mnih et al., 2016; Schulman et al., 2017; Schulman et al., 2015; Lillicrap et al., 2015; Fujimoto, Van Hoof and Meger, 2018; Haarnoja et al., 2018; Mnih et al., 2013; Bellemare, Dabney and Munos, 2017; Dabney et al., 2017; Andrychowicz et al., 2017; Ha and Urgen Schmidhuber, 2018; Racanière et al., 2017; Nagabandi et al., 2017; Feinberg et al., 2018; Silver et al., 2017). These algorithms are not an exhaustive list but covers most of the popular methods which are found in most reinforcement learning frameworks or libraries such as Stabe-baselines3 (Stable-baslines3, 2023), Ray RL Library (Ray, 2023b) and Keras-rl (keras-rl, 2019).

These algorithms were reviewed in their original publications and later as exploited use cases in scenarios such as autonomous driving, racing games, robotics or videogames.

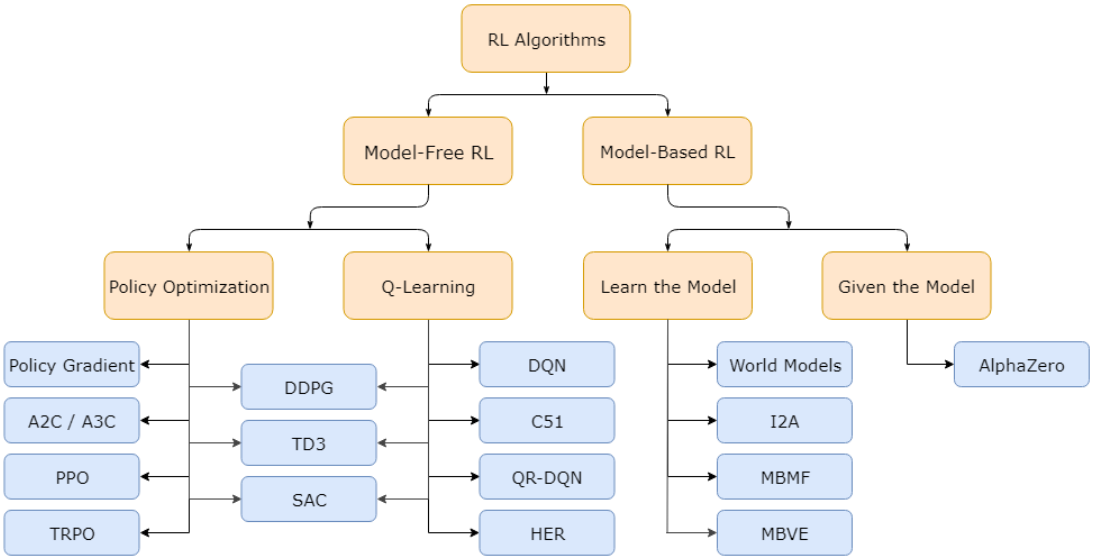


Figure 3 - A Taxonomy of algorithms in modern RL (OpenAI, 2022)

2.1.1 Model-Based Algorithms

The AlphaZero algorithm is shown to train an agent through self-play in a short time (the authors claim 24hrs) to achieve superhuman capability in Chess, Shogi and Go, without requiring manual re-tuning of any hyper parameters. However, AlphaZero has perfect knowledge of the rules of each game, and a noise is applied during training proportionate to the typical number of moves for that game to encourage exploration.

Dependency on knowledge of the rules makes this method inappropriate for a racing agent, since that would require implementing knowledge nearly equivalent to that of the game's physics, vehicle characteristics and track properties, in addition to driving techniques for all scenarios. As a simple example, applying the brakes on the road would behave differently depending on the friction of the various road properties modelled in the game. The effect of an action also depends on the state of the vehicle that is controlled by the agent, as well as behaving differently according to vehicle properties such as wheel torque capacity. Therefore, the techniques employed by AlphaZero were deemed inappropriate, as the effort to model these factors would be cumbersome or impractical.

It is also uncertain, how fast can AlphaZero infer an action for a given observation since the authors evaluated it on turn-based board games. In racing, the observation state is also a function of time due to the vehicle's motion. In a PSX game, we can assume the desirable time step to be between ideally between 0.02s (the frame-period of a PAL game on the PSX) and 0.3s comparable to an F1 driver's reaction at the start of a race (McLaren, 2020).

While model-based methods seem to be less common in literature for real-time game problems, a paper published recently (towards the end of this work) titled *Mastering Diverse Domains through World Models* (Hafner et al., 2023) seems to effectively overcome the issue of re-implementing the physics. Their method referred to as Dreamer V3, uses reinforcement learning to learn the dynamics of the environment. The model of the environment is then utilised to simulate and train an actor-critic pair of neural networks. It is claimed that this solution is beneficial for various problems in terms of the type of action space, the size of the observation space and the sparseness of the rewards. The authors prove the scalability of Dreamer v3 on a sparse reward problem through mining diamonds in Minecraft. Since this publication emerged late in this work, it was not thoroughly considered, but noted to exist.

2.1.2 Non-Model Based Algorithms

One of the most significant methods of non-model-based learning was that of Q-learning by Watkins in 1989 (Sutton and Barto, 2018, p. 131). The method relies on learning the action value per each state-action pair, which can be considerably large for complex problems that have a large state-space and or large action-space. In case of a continuous action, in the pure mathematical sense, an infinite series of actions are possible which would not allow for a finite Q-table. One could approximate a continuous action by restricting the resolution such as to discretise the actions, but this may lead to a "dimension explosion problem" (Anonymous authors, 2022).

Several algorithms have been developed upon the concepts of Q-learning. The most relevant in video games is that for Deep Q-Networks (DQN) which was utilised with a convolution neural network (CNN) to learn to play 49 Atari games (Mnih et al., 2013). This application is said have kick-started the application of RL in video games (Hessel et al., 2017). One of the features introduced by Q-learning is that of off-policy learning. By this method, the policy that is being optimized can be distinct from the policy being used to generate episode steps (Sutton and Barto, 2018, p. 131). DQN algorithms have been improved by variations or extensions such as the Double DQN to compensate for bias, or A3C to promote exploration (Hessel et al., 2017). An interesting and recent study was done to combine the different elements of strictly value-based Q-learning algorithms into what was named Rainbow which was evaluated at Atari games (Hessel et al., 2017).

Other methods often cited in literature that are said to help overcome the issues of continuous action spaces, include, Trust Regions Policy Optimization (TRPO) albeit incompatible with noisy environments and Proximal Policy Optimisation (PPO) which was shown by its authors to outperform methods such as the Advantage Actor Critic (A2C) in complex benchmark examples such as the Walker 2D problem (Schulman et al., 2017).

One surprising observation in literature is that authors often benchmark their work on Atari games. An example of such comparison is that of the Policy Gradient and Q-learning (PGQL) against Asynchronous Advantage Actor Critic (A3C) and traditional Q-learning (O'donoghue et al., 2017). This highlights the value of considering video games in literature, however also casts doubt on the relevance of such simpler games.

The focus on Atari-games as a benchmark is perhaps insufficient to judge their applicability to 3D racing games, as their scalability to problem complexity and larger continuous space are not well demonstrated in their papers. A game such as space invaders has an action space of four actions (left, right, no motion with shoot or not shoot), and since the actions are pairs, they can be considered as two dimensions, where the first domain is *left*, *right*, or nothing, and the second domain is *shoot* or n*ot shoot*. In comparison, a game such as Street Fighter II for the SNES, non-binary dimensions available. The first dimension has 8 directions and the second has 6 action buttons with on or off states, leading to 512 (2^8) actions. Utilising known mechanics of the game and gamepad limitations, the action space could be reduced to 35 (Fletcher, 2017), which is still nearly 6 times greater than Atari 2600's Space-Invaders. The low computation effort to emulate such games, means that the possibility to run parallel environments is also more feasible (Mnih et al., 2013). In the case of emulating a PSX racing game, it was necessary to consider literature that consider at least higher complexities than Atari games and that might deliver positive results without parallelisation.

2.2 Application Specific Methodologies

A thorough effort was made to review literature that applied to specific problems that were deemed analogous to a racing scenario, either by nature of the game or by the complexity of the environment observation, reward functions and action mechanisms. Here also the technical implementation described in literature is also reviewed.

At the start of this work there were only two PSX related AI projects, one of which is based on a 3D ball game and was discovered before commencing this work and will be reviewed as part of the methodology proposal (Purves et al., 2019; Purves, 2019b; Purves, 2019a). The second is final work is undocumented but successfully achieves a 100%-win rate in Crash ball (Costa, 2022).

During this work, a third RL application on the PSX emerged for a 2D game Mega Man X (Severo, 2023b). The author got in contact to learn from this project's early work as inspiration. Since they completed their work ahead of this, albeit without formal publication, they were included as part of the literature.

2.2.1 PSXLE

The first and oldest work focusing on the PSX is *The PlayStation Reinforcement Learning Environment* (PSXLE). The author and authors related to it, focus on the engineering implementation necessary to provide the OpenAI Gym API interface (Purves, 2019b; Purves et al., 2019) for which the source code is also available (Purves, 2019a). The author provides a single use case as evidence of their success however this review will highlight inadequacies or misleading claims made.

The original author of PSXLE modified the PCSX-Reloaded (PCSX-R) emulator, which is an opensource evolution of PCSX (Linuzappz et al., 2023). The emulator itself is said to be used by Sony themselves in their PlayStation Classic (a recently officially licensed emulation device imitating the shape of the original PSX in a smaller form-factor) from 2018 (Kohler, 2018). One of the contributors of the original PCSX developers having experienced the PCSX-R on the PlayStation Classic said:

> *"I came to realize two things: first, the state of the PlayStation emulation isn't that great, and second, the only half-decent debugging tool still available for this console is that old telnet debugger I wrote eons ago, while other emulators out there for other consoles gained a lot of debugging superpowers… also felt I had a responsibility to cleaning up some of the horrors I've introduced myself in the codebase long ago, and that made me cry a little looking at them."*
>
> (Noble, 2023a)

From such comments one can infer that PCSX-R might not emulate the real hardware faithfully. Deviation from emulation with real hardware which could lead to issues such as the compatibility of replays generated in GT. Compatibility of GT replays are beneficial for others to race against as a ghost. The authors of PSXLE created their own fork of PCSX-R, which at the time of writing had not been updated in four years. Following their approach may lead to further issues such as incompatibility with more recent Windows or Linux operating systems.

The authors of PSXLE also implemented emulation control such as to pause and resume emulation as a synchronous control between the RL environment and the emulation frames.

In earlier work on PSXLE (Purves, 2019b) it was said that OpenCV is utilised to crop and separate the red-green-blue (RGB) components of the game screen, however it also can process the audio from the game to detect specific events in the game Kula such as “coin collection”. The use of audio seems to be novel in comparison to the literature that has been reviewed, however its benefits may be limited to the simplicity of the Kula game. The authors of PSXLE demonstrated to have produced positive RL results in the game Kula World utilising DQN and PPO.

In the later publication, the authors suggest that their work is applicable to any PlayStation game (Purves et al., 2019), a suggestion emphasised in the source code’s readme (Purves, 2019a). Assumptions by the authors will be critically reviewed to only be true for games with similar simplicity to Kula. This is not a complete dismissal of their work, as it was the first true publication involving the PSX.

Kula World is a ball game in a 3D world; however the game is better described as a board game. As shown in the following Figure 4, the 3D world is made of discrete positions. That means after a player performs an action (such as pressing forward), the ball will roll perfectly from 1 position to another by a fixed animation duration, and thus all meaningful states can be represented as a simple 2D array (see Figure 4).

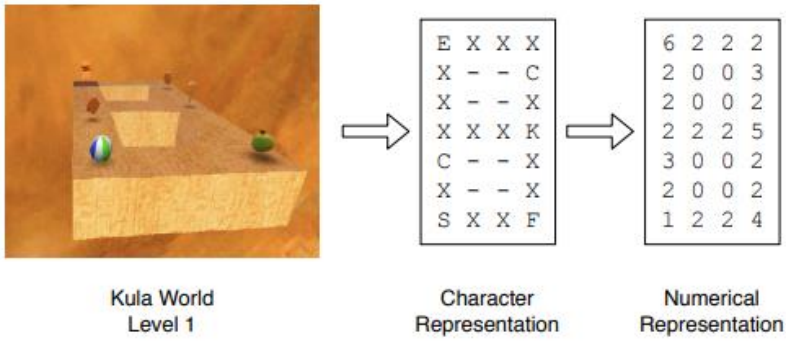


Figure 4 - 2D Representation of a Kula World Level (Purves, 2019b)

This means the time-dimension of the game for an optimal policy depends on two only actions, that of the selected action and the time idling until a next action is computed. Figure 5 demonstrates a slow (upper) and faster (lower) gameplay assuming the same sequence of actions, the time inbetween actions are fixed by their animation, it is only the idle that can be reduced for the same sequence of actions. Furthermore, an action leads to an asserted state without uncertainty, since the time the ball is in motion is pre-determined and unconditional to any other parameter. The animations are not meaningful game states.

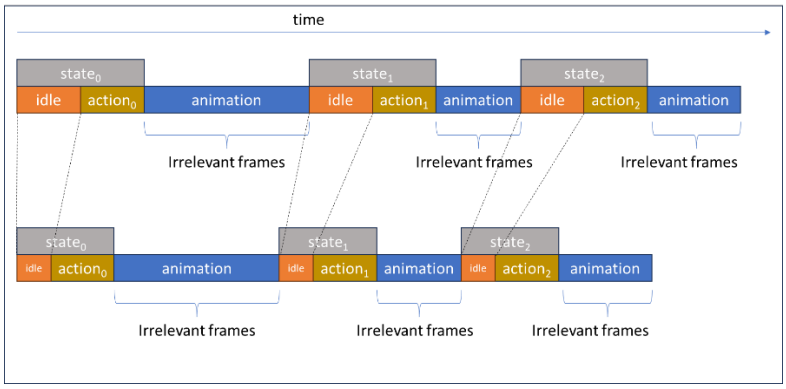


Figure 5 - Time mechanics in Kula World

The authors introduce and use the term asynchronous mode, which could be misleading, as their framework and pipeline still relies on allowing the agent to synchronize with each meaningful game state. The asynchronous notation refers to the varying frame-skips due to the varying animation duration, rather than fixed frame-skipping as is used on Atari games work (Mnih et al., 2013).

In racing game such as GT, every frame is valuable while the vehicle or engine are in transient conditions. Satisfying the Markov Decision Process (MDP) from a visual spatial position alone may be insufficient, since if a vehicle is stationary at a given x-y-z position since a vehicle accelerating would lead to different states in a next observation at a different acceleration term. Games like GT cannot be reduced to a 2D observation space from visuals alone in the same way that Kula was.

This consideration suggests that PXLE's asynchronous method would not be beneficial for this work since there are no ideal frames to skip. Furthermore, due to the emulator choices, and limited applicability, PSXLE was judged as insufficient to deal with a complex matter such as that of the GT problem.

2.2.2 Crash Bash & Mega Man X4

Another PSX RL project is RillAI (Costa, 2022), hereafter referred to as Crash Bash due to the focus on the PSX game Crash Bash. Crash Bash is a game containing several mini games, where characters throw balls at each other to eliminate the other players (IGN, 2023) as shown in Figure 6.

Figure 6 - Crash Bash (IGN, 2023)

The author of the Crash Bash project utilised the emulator BizHawk, an emulator developed for speed running due to its record and playback features and its Lua interpreter (TASEmulators, 2023). The implementation does not implement any known API such as Gymnasium. The RL algorithm used is DQN and relies on triple-frame stacking. The observation passed to the neural network is purely parametric. The observation and reward are communicated to the Python environment as simple byte-string encoding (in the UTF-8 format) over a TCP socket. Despite the data being small due to the lack of a graphical frame, the author still implemented emulation pausing. The TCP socket was also used to transmit actions to the emulator.

The implementation was such that the emulation paused after each observation transfer during the on-policy training, as well as pausing until the next action is determined. In conversation with the author, it was revealed that during training, the game ran at approximately half speed (full speed being 60 frames per second). The policy could be used for forwarded passes at nearly full game-speed when using to determine the next action after each observation (at around 20 observations per second due to 3-frame stacks and the opted 2 frame-skips). The neural network had an input layer of 34 nodes fully connected sequentially to 2 layers each of 64 nodes with 4 outputs, resulting in a total of 7,348

trainable parameters. The low quantity of parameters is a possible reason that the game could run at a normal 60 frames per second render (as said by the author). This approach suggests that emulation pausing is not triggered in forwarded passes (albeit unconfirmed by the author). It was deemed smooth enough to allow playing against a human. In doing so the record holder of Crash Bash was defeated during 1 hour of gameplay (Costa, 2022). Had the emulation pauses been significant enough to reduce the frame rate, the human player might have reported discomfort. The fact that the agent could observe every frame as a set of 3 frames, which would equal to a reaction every 50 milliseconds, which is faster than the average typing speed of 260 characters per minute (Szeto, Straker and O'Sullivan, 2005).

Another RL application to a PSX game is that done on Mega Man X4 game 4(Severo, 2023b). The Mega Man X4 work has similarities to that of Crash Bash in that it also utilised BizHawk and a TCP socket to exchange data also with a simple byte-string encoding (in the UTF-8 format). One noticeable difference to that of Crash Bash but corresponding to that on Atari games by Mnih et al., (2013) is the use of a CNN. Each observation contains a stack of three display frames that have been converted to greyscale and resized to 84 by 84 pixels, after which six frames are skipped. The emulator also sends over TCP socket in addition to the display stack, player's health, enemy's health, and score. The parametric values were only to calculate a reward. The TCP socket was also used to transmit button presses back to the emulator to execute the actions.

The Mega Man X4 work relied on emulation pausing to keep the agent in synchronisation with the emulator environment for training. This work also implemented parallelisation to speed up learning by running multiple emulation session at once. As for Crash Bash, the Mega Max X4 author revealed in an online conversation that the game "seemed" to run at a smooth 60 frames per second when the agent was used and not trained. This also suggests that emulation pausing was unnecessary in this case or minor. The author also said that they attempted applying DQN learning algorithm before settling for the PPO algorithm (another on-policy alogirthm). The trained agent was able to beat all the games main bosses of Mega Man X demonstrated in a YouTube video (Severo, 2023a).

Both the Crash Bash and Mega Man X4 utilised the BizHawk emulator and a TCP socket to transfer necessary data to the Python environment for both transfer of actions and observations. Both implementations relied on some form of frame stacking and pausing, whereas Mega Man X also relied on frame skipping. The pausing seems to have a noticeable difference only in training. Both games are more complex than that of Kula World in that the time-steps between frames alter the state of the environment in meaningful ways, but such differences are cancelled out by pausing. What both works demonstrate, is that even for certain complex environments, a DQN or PPO methods with appropriate network architecture can result in a policy that can keep up with the games at real-time speed.

The methodology of emulation pausing is against the real-time requirement of this project. Both works passed few parameters to the Python environment, which meant that a simple assumed timing of the byte-string was sufficient. This limits the feasibility for larger data or the flexibility to easily adapt the method to other games. A different solution than that of this works for transmitting observations and actions was deemed necessary.

2.2.3 Street Fighter

This work has applied RL to play Street Fighter on the SNES (Fletcher and Mortensen, 2018; Fletcher, 2017). While this is not a 3D or a PSX game, the game is very high paced. Unfortunately, this work shares several similarities with those of Crash Bash and Mega Man X works. The BizHawk emulator is applied, however instead of utilising the Lua interpreter to implement the TCP socket communication, the authors compiled a custom fork of BizHawk to implement TCP sockets natively via C# code to achieve the pipeline shown in Figure 7 (Fletcher and Mortensen, 2018).

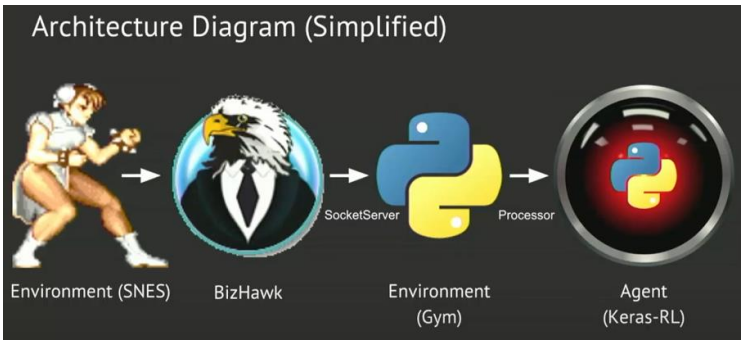


Figure 7 - Street Fighter II Reinforcement Learning (Fletcher and Mortensen, 2018)

In this work the authors also relied on emulation pausing whenever the Python computation was not ready. Like the Mega Man X work, the Street Fighter II implementation followed the Gym API which allowed the use of the Keras-RL library (Fletcher and Mortensen, 2018).

One notable feature of this implementation is that the AI agent is limited to taking an action every 20 frames, which equates to 333 milliseconds at 60 frames per second (Fletcher and Mortensen, 2018) and is in the range of an F1 driver's reaction time (McLaren, 2020). They were also able to utilise a Google cloud service to perform parallel training of multiple agents at 450 frames per second for 3 hours. That resulted is 7.5 times the game's normal running speed of 60fps (Fletcher and Mortensen, 2018). This meant that while pausing was implemented according to a snippet of their code shown (Fletcher and Mortensen, 2018), the higher computational power available for training could make up for the lower experience accumulation from the high number of frame-skips.

This work had two incompatible choices, which is to utilise frame-skipping and to make use of forks of an emulator (such as in the PSXLE work).

2.2.4 Gran Turismo Sport

The publication about Sophy for Gran Turismo Sport (GTS) on the PlayStation 4 (Wurman et al., 2022) is about a methodology of training a real-time AI agent for motorsport racing. The publication seems best categorised as experimental research because it adapts state of the art concepts, in combination with a training regime in a virtual environment. The work only explicitly focuses on training and evaluating the work in a simulation environment, albeit claiming the applicability to real world scenarios. The context of their work is based on concepts of model-free, off-policy and deep RL for training in both single and multi-agent environments. The reward and penalty systems were parameter based, but subjectively evaluated by humans. Specifically, certain parameters were set subjectively to deliver human-like sportsmanship. The authors initially attempted to implement a model to predict blame, however had to discard this method as it resulted in an agent that was too aggressive in the eyes of human evaluators which were also test-drivers. In motor racing against other players, racing incidents may occur for which it may be challenging to assert blame objectively according to the authors. In GTS an un-sportsmanship fault, results in a 100 km/h temporary speed restriction. (Wurman et al., 2022).

The authors claim that they utilised the Gran Turismo environment because it is an ideal realistic simulation environment that can "*faithfully*" (Wurman et al., 2022) reproduce non-linear difficulties of real race racers in a multi-agent environment. The authors implement a variation of the soft actor-critic named QR-SAC. This algorithm will later be described to be part of a common group of algorithms when dealing with racing scenarios. Most of the training was done by the proposed QR-SAC algorithm, which trains an agent to learn a policy, called the actor, which selects actions based on the agent's environmental understanding, and a value function, called the critic. It is stated that this algorithm is an extension of existing actor-critic approaches, in that it is modified to represent to expected future rewards value by probability distributions. The QR-SAC methodology trains the neural network asynchronously using sample data from an experience replay buffer. Actors generate driving experience using the recent policies further filling the experience buffer. The authors demonstrate that QR-SAC permitted an 0.6% improved lap-time on the Maggiore track (Wurman et al., 2022) in comparison to SAC. This suggests that traditional SAC is perhaps sufficient for being competitive (albeit not necessarily outpacing the best human players).

The setup of the research is only briefly described by the researchers (Wurman et al., 2022) as they could make use of a special API to Gran Turismo Sport. The developers utilised PlayStations for running the simulations while the agents were running on computational GPUs. The neural networks updated asynchronously as previously described and interacted with the game-observation at 10Hz (or every 100 milliseconds), which is said to be by the authors, the professional limit, which is at least in the scale of the 300 milliseconds referenced earlier for F1 drivers.

The authors also accelerated learning by partially optimising exposure. The authors give the example of a slingshot pass, a manoeuvre for which the right conditions rarely arise. The AI was exposed to what was called mixed-scenario training and manually selected driving conditions by a retired professional GT driver, to identify such scenarios. The authors also utilised proportional, integral, derivative (PID) controllers to optimise slipstreaming behaviour (Wurman et al., 2022). From these aspects of selective exposure which the authors claim to not be a form of curriculum training since there was no hierarchal bias for training (Wurman et al., 2022). These unique adjustments highlight that in motor-racing there are events that are more rare and thus more difficult to learn. A human player is perhaps able to learn these manoeuvres faster through coaching or scenario specific tutorials.

The experiences generated from the 4 PlayStation 4s, were transferred back to a centralised trainer asynchronously to revise the policy and Q-function networks (Wurman et al., 2022). Relying on algorithms that benefit or depend on parallel environment for training adds engineering challenges to the project and may exhaust the available computational power, and therefore such methods are restrictive to those with limited resources.

Conveniently but unsurprisingly, the environment setup for Sophy was utilising an API in Gran Turismo to provide access to all necessary in-game parameters as well as controls. Interestingly the authors decided to configure the acceleration and braking control on the same dimension [-1,1], which effectively prevents the agent from braking while simultaneously applying the accelerator. The justification given by the authors is that it is "rare" to apply those actions simultaneously (Wurman et al., 2022). As described in the introduction, the GT and default PSX controllers only allows simultaneous brake and accelerator actions using the digital pad (discrete actions), however the standard PS4 controller allows for continuous analogue accelerator and brake control to be applied simultaneously in GT Sport. Furthermore, it seems biased by the authors to judge that this rare human action is not part of a superior policy (this knowledge could perhaps be used against Sophy on tracks where braking while accelerating is known to be beneficial). More likely, this choice was taken to reduce the action space, despite that each trainer node had access to an NVIDIA V100 or half an NVIDIA A100 with 8 virtual GPUs and 55GB of memory.

The agent was able to receive from the game engine all other special parameters such as 3D linear and angular velocity as well as 3D acceleration, tyre loads, tyre slip, and relative track position as visualised in Figure 8. One key parameter that was not mentioned in the original publication detailing Sophy's observation space but described in Sony AI's blog, is the ability to observe the track ahead as 6 points with the 6th point relating to six seconds based on the vehicle's real time speed. This feature seems to be unique in literature (Subramanian, Fuchs and Seno, 2022) except in one publication (Neinders, 2023), which focused on verifying such advantages utilising the game Track Mania, which was researched after the publication of Sophy in the mentioned literature and blog.

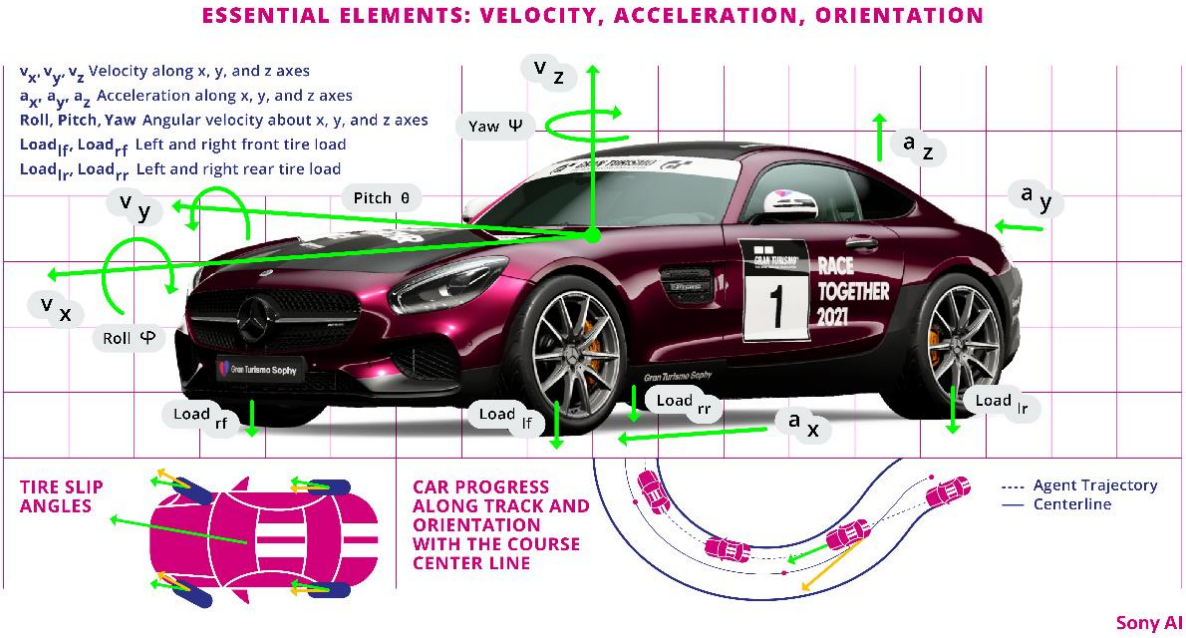


Figure 8 - Sophy's Essential Observed Parameters (Subramanian, Fuchs and Seno, 2022)

Another complex element to Sophy is the reward function, which is composed of eight elements – the course progress, off-course penalty, a wall penalty, tyre slip penalty, passing bonus, any-collision penalty, rear-end penalty and unsporting-collision penalty (Wurman et al., 2022). Of all the elements making up the total reward, the tyre slip is most surprising, as one may have assumed that the conditions that cause slip to a detrimental level would be learned. However, the details of the condition under which the reward is applied is more specific in that negative reward applies when the slip is in a direction different from the direction pointed by the tyre. It is suspected that the imposition of penalties for veering off course and colliding with walls may have been unnecessary, except to guarantee that the agent avoids agent does not find any short-cut way of reward hacking itself or to keep the sportsmanship fair. In the real-world drivers may avoid strategies such as riding the wall to avoid damaging their vehicle, which is not an obvious concern for an indestructible virtual vehicle. In GTS there are natural penalties when hitting the wall or cutting corners even for human players, whereas in GT there are none, and hence human players tend to tap the wall while achieving competitive lap-times.

In addition to the publication by Sony AI in Nature (Wurman et al., 2022) there are two publications from the University of Zurich that are co-authored by members from Sony AI and predate the Nature publication. Due to the support from Sony, they also had access to Gran Turismo Sport's API (Song et al., 2021; Fuchs et al., 2021).

The first of the Zurich publications focuses on lap times similar to the time trial objectives of this work. The setup utilized 4 PlayStation 4 consoles to collect observation sample data from 20 vehicles running on each console. The observation is used to generate trajectories that are also stored in a replay buffer which can be used for training a soft actor critic algorithm (SAC) by utilising mini-batch sampling from said buffer as shown in Figure 9 (Fuchs et al., 2021).

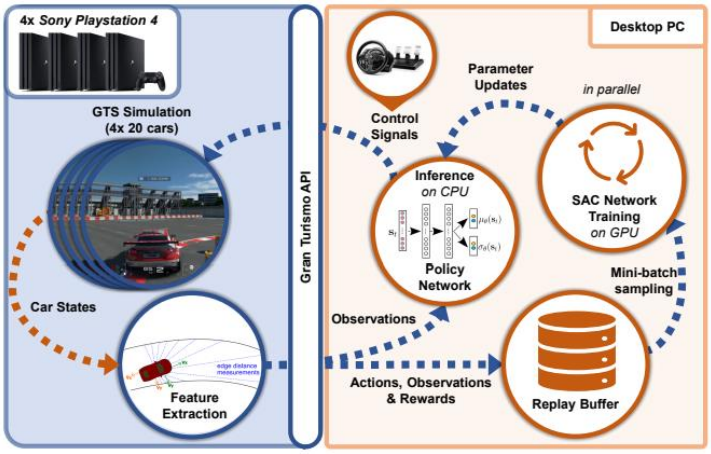


Figure 9 - Setup for Training in Gran Turismo Sport (Fuchs et al., 2021)

Just like the official Sophy publication, this work emphasised the importance of low-dimensional state representation despite the available resources (Fuchs et al., 2021).

The authors utilised a very simple reward that is based on the progress of the vehicle along the track, subtracted a penalty for the case the vehicle contacted the wall such that is proportional to the square of vehicle's velocity as per equation 1.

$$r_t = r_t^{prog} - \begin{cases} c_w \|v_t\|^2 & \text{if in contact with the wall} \\ 0 & \text{otherwise} \end{cases} \qquad \text{Equation 1}$$

The reward component r_t^{prog} is the progress of the vehicle along the track's centre line based on the intersection between the track centreline-profile and a perpendicular line from the centre of the vehicle perpendicularly projected from the centre of the vehicle as shown in Figure 10.

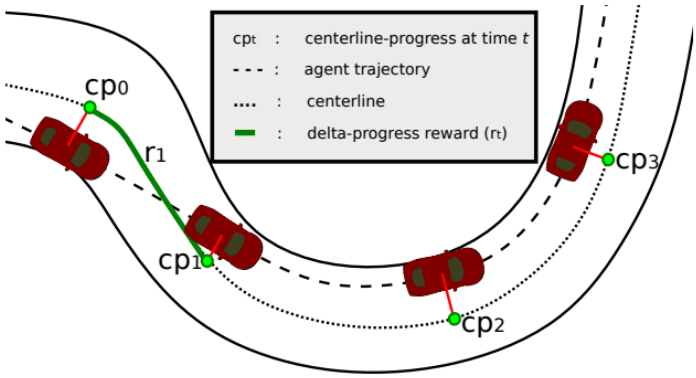


Figure 10 - Vehicle Progress (Fuchs et al., 2021)

The neural network is not provided any visual data and instead contains 7 state parameters for a given time step t, $[v_t, \dot{v}_t, \theta_t, d_t, \delta_{t-1}, w_t, c_t]$which represent the 3D velocity vector, 3D acceleration vector, Euler angle, range-finder distances (akin to lidar distance measurements as shown indicated by the blue lines in Figure 11), the steering angle at the previous observation, a binary flag for wall contact and a description of the curvature along the centre line in the near future (Fuchs et al., 2021). Most the parameters are not dissimilar to those described for Sophy (Wurman et al., 2022).

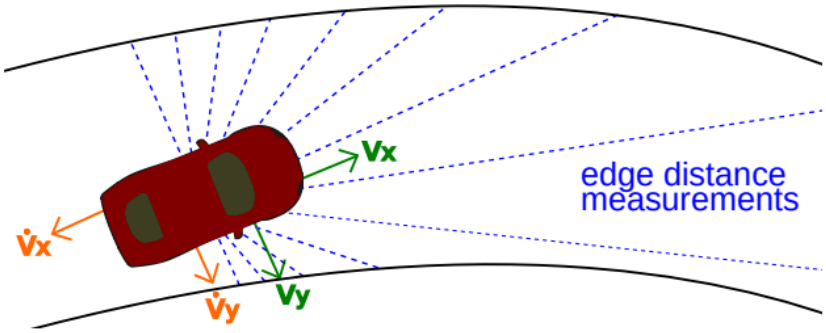


Figure 11 - Distance Measurements (Fuchs et al., 2021)

Like the work for Sophy (Wurman et al., 2022), these authors also set the control action space such that one dimension is available for both acceleration and braking as a value from positive 1 to negative 1, however the justification was not that it is rare, but rather that analysis of fast human recordings did not utilise simultaneous braking and accelerating. Unlike the neural networks utilised for Kula World (Purves et al., 2019; Purves, 2019b), the authors from Zurich utilised a larger neural network containing 2 hidden layers with 256 nodes for each of their network branches (policy network, two Q-networks, and a state-value network). Their setup utilised a normal desktop computer to train an agent with super-human abilities (around 40 to 400 milliseconds faster that fastest human data available) within 73 hours of training. Their setup is able infer an action from an observation and communicate with the PlayStation within 4 milliseconds, which the author admits, is an advantage over their recognition of human response time at around 200 milliseconds. A phenomenon was the influence of the delays in inference on the resulting lap-times (Fuchs et al., 2021) as shown in Figure 12, which shows a near flat relationship up to 50ms and what seems like a liner relationship from 100 to 200ms.

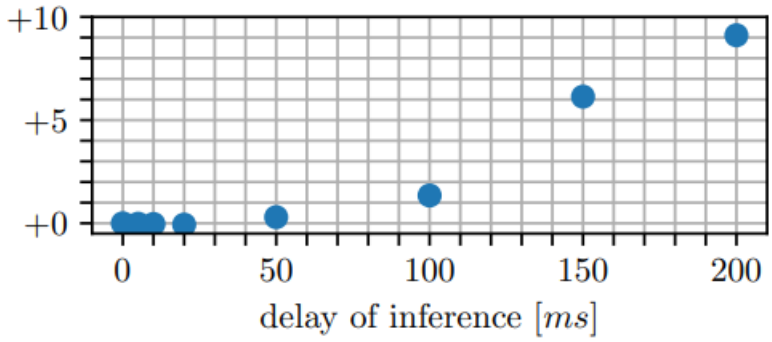


Figure 12 - Lap Time Loss (seconds) due to Inference Delay (Fuchs et al., 2021)

The second publication from Zurich university (Song et al., 2021) focuses on utilising SAC and a curriculum reinforcement learning to train an agent at overtaking. The problem as described is that overtaking is a compromise on the traditional reward of progressing through the track as fast as possible (such as time-trial). The curriculum method proposed by the authors is said to allow faster convergence in contrast to traditional deep RL.

The observation parameters are slightly different from the ones proposed in the prior time-trial based efforts(Fuchs et al., 2021). The parameters are now revised as $[v_t, \dot{v}_t, \theta_t, d_t, \delta_{t-1}, f_t, f_c, c_L]$, where c_L seems to be equivalent to the previous c_t term that provided curvature information of the centreline 0.2~0.3 seconds ahead. The wall collision flag w_t is now replaced by two flags f_t and f_c representing wall and car collisions respectively. Interestingly, all terms are z-score normalised except for the lidar distance d_t which is said to be min-max normalised instead. Justification was not provided (Song et al., 2021) but perhaps one reason is that lidar distances are always positive and the max possible value can vary depending on the track.

The reward function is replaced with two functions instead. The first is the same as that used in the prior work of the previously review literature (Fuchs et al., 2021), referred to as the racing reward.

The second reward function is expanded to include additional terms to encourage overtaking the opponent and achieve a better position in the race order. When the distance between the agent and a forward vehicle is below a certain value (which is said to be a hyper-parameter), the reward function utilises more parameters to encourage the agent to get closer to the vehicle ahead (Song et al., 2021). This second reward function is referred to as the overtaking reward.

The other novelty of this work (Song et al., 2021) is that of the curriculum-based training to overcome the “extrapolation error” caused by falsely estimating the state-action pair value for rare events such as overtaking. Their method uses a randomly initialised policy and randomly initialised neural network that is first trained in time-trial setting utilising the racing reward function.

After the agent can perform well at solo driving, it is then trained in custom scenarios where the agent is placed 200 metres behind a single vehicle (that is controlled by the game’s default AI), while maintaining the original replay buffer but re-initialising the weight of the exploration terms of the policy. During this second phase of training, the overtaking reward function is used (Song et al., 2021). The agent is then trained on new samples with a higher penalty term for collisions while still maintaining a fixed replay buffer with a first-in-first-out (FIFO) sequence to gradually shift out earlier experiences (Song et al., 2021). The authors were demonstrated that this method was successful at training an agent to overtake earlier in a race than their reference human-baseline data, by exploiting different overtaking strategies.

The curriculum-based training is appealing too because it may be relatable to how humans might learn to race, that is that they would likely involve learning to race around a track alone, before experiencing interactive racing, during which the reward and learning focus is also shifted to overtaking and defending.

2.2.5 WRC 6

In literature another example for actor-critic method to train an agent to drive a virtual racing environment, this time utilising the game World Rally Championship 6 (WRC 6), which provides a variety of tracks and scenery (Jaritz et al., 2018). In their work, they utilise asynchronous advantage actor critic (A3C) to train an agent in various conditions which means the environment looks different and behaves differently in terms of try grip, thus making A3C better suited due to the lack of replay buffers.

The authors utilised the dedicated API of WRC6, to extract the RGB image of the game, the reward and to transfer back the actions. Unlike the GT Sport related work, the authors here separated the accelerator and brake actions from the action space in addition to the steering, with all of them being discretised continuous actions. The authors also provided the agent with the ability to actuate the handbrake as a binary action (Jaritz et al., 2018).

Just like prior racing games reviewed, an artificial reward had to be introduced since the traditional reward (the final lap-time) would be too sparse to train with. Instead, a reward function was shaped as follows (Jaritz et al., 2018):

$$R = v(cos\,\alpha - d) \qquad \textit{Equation 2}$$

Where the reward R depends on the vehicle's velocity v and the difference between the vehicle's heading and the road direction α. The distance between the centre of the road and the vehicle position d was added as a penalty to encourage the agent not to slide along the guard rails (Jaritz et al., 2018).

The authors demonstrated that it is also beneficial to initiate episodes at different check-point positions rather than the start of the track to encourage generalisation. The training was done utilising one computer for running the algorithm with 9 simultaneous environments running on 2 sperate computers. The authors utilised the front-bumper-view without any extra information such as heads-up display. Episodes were terminated when the vehicle goes off track or travels in the wrong direction.

The result of this method was an agent that can drive well, and even utilise the handbrake to drift around certain corners but was unable to achieve racing performance because of two main reasons: a) the agent is unable to forecast the road ahead (Jaritz et al., 2018) unlike Sophy (Subramanian, Fuchs and Seno, 2022) and b) the agent was penalised for deviating for the centre of the track which is not always to optimum racing line. To overcome the the, issue the authors evaluated two alternative rewards functions that took into the account the road width (r_w) to calculate the penalty component. The better of the two was one that utilise the sigmoid function to the penalty resulting in the following reward function (Jaritz et al., 2018):

$$R = v\left(cos\,\alpha - \frac{1}{1+e^{-4(|d|-0.5r_w)}}\right) \qquad \textit{Equation 3}$$

The authors were able to develop an agent that was generalised sufficiently to handle different courses in a subjective and qualitative sense (Jaritz et al., 2018). While the agent performance in the race is not comparable to humans, it is worth appreciating how successful the agent was considering

the simplicity of the observation and the simplicity of the reward function. The reward function in equation 2 may have unintentionally and naively trained the agent to try and follow the track centre at the highest velocity possible, but the minor modification introduced by equation 3 helped overcome that naiveness by allowing the agent deviate from the centre for half the track width. Nevertheless, the final setup still seems to have a built-in bias towards the centreline such as in the work on GT Sport by Zurich University.

2.2.6 TMRL

Trackmania (TM) is a modern racing game currently available for the PC but also expected to be released on modern game consoles such as the PlayStation 5 and Xbox One in 2023 (Ubisoft, 2023) and is an appropriate candidate for RL since it is available as a free to play version (an edition of the game that has a full sub-set of the game features to play without cost) with time trial mode (Epic Games, 2023). The authors of the TMRL project did exactly that, as well as providing a TrackMania Roborace league, a ranking of AI agents (Bouteiller, Geze and GobeX, 2023b), for which an agent is only allowed to receive the screen image and base vehicle parameters - speed, gear, rpm (Bouteiller, Geze and GobeX, 2023a).

Several other authors have also utilised TM for RL purposes such as PedroAI (Porcher, 2023) and TMFORGE (Boyer, 2020). Another work building directly off from the TMRL one that studied some of the differences between Sophy and the default TMRL method (Neinders, 2023) which was reviewed as part of the GT Sport section of literature.

The PedroAI work does not provide an explanation of how they interfaced the game with reinforcement learning environment, whereas the other works mentioned make use of an extension called Openplant which permits the extraction of basic vehicle parameters for defining the state as well as following the Gym or Gymnasium API (Boyer, 2020; Bouteiller, Geze and GobeX, 2023b).

TMRL provides two default environments. The first method relies on pseudo light detection and ranging (LIDAR) as shown in Figure 13. It is not through LIDAR since the length of the lines are not proportionate to each other since it is utilising the driving view to project them. Lines close to the vertical axis can be longer distances than those on the horizontal axis. Nevertheless, the method seems to work well based on my own tests. This pseudo-LIDAR method relies on a convenient feature of the track, which is that the track border is black, which allows for simply detecting the black pixels from the projected rays.

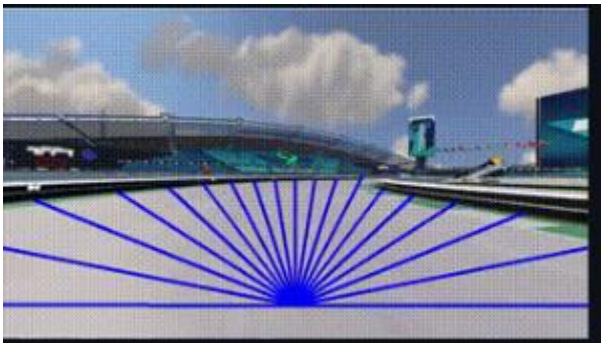


Figure 13 - LIDAR mode (Bouteiller, Geze and GobeX, 2023b)

The second environment does not use pseudo- LIDAR, but instead utilises the full display image scaled to 64 by 64 pixels and in greyscale, in combination with a CNN.

There are several novel features that TMRL employs, mostly stemming from the real-time-gym (rtgym) library. The rtgym library is meant to provide features that make it possible to perform RL on real-time or robotic environments (Bouteiller, 2023a). The choice to do real-time based RL means that there are less transitions between states to observe and learn, which is why the replay buffer of the SAC used in TMRL is more beneficial than in a non-real-time environment (Bouteiller, Geze and GobeX, 2023b). If time were a factor for decision making, then essentially the Markov Decision Process (MKD) principle cannot apply, since the transition to the next state would be dependent on the time elapsing before taking an action. Most of the work in literature reviewed above, suffer from this property but overcome by frame-skipping or environment pausing (through emulation pausing), brute-force of computational power, or a combination of these methods, such as to be ready for the next observation in time. TMRL uses rtgym to accept and deal with the time-domain differently. It instead elastically constraints each time-step to fixed boundaries (a maximum delay for which to assume an action is still meaningful to take). The observation also contains a pre-determined quantity of previous actions, which serves to help predict the state the game will be in when the action that is currently computed is finally applied. This is influenced by the time required to compute an action and the time required to acquire an observation (Bouteiller et al., 2021).

TMRL also utilises a history of images or of LIDAR measurements to observe transient conditions, the default history is set as 4 state frames, which would suggest that the differential of acceleration can be recognised. The actions are performed using a virtual gamepad python library, capable of emulating both an Xbox 360 or Dual Shock 4 controller (Bouteiller, Geze and GobeX, 2023b), that is unfortunately was only available for Windows (Bouteiller, 2023b) at the time of this work. This is beneficial as it means actions are subject to most overhead processing a human player might experience when playing, as opposed to using a custom direct pipeline to inject actions into the game.

The use of rtgym breaks compatibility with certain RL frameworks such as Stabelines3, as they expect the observation space to be a dictionary which rtgym does not provide. It should be possible to achieve with additional wrappers, but such efforts were not investigated as no example of using such wrappers exist in literature.

From a subjective perspective, the pre-trained agent provided by TMRL performs nicely, and is only unable to elegantly deal with a couple of corners (nearly hitting the wall). The work of (Neinders, 2023), shows that the agent suffers around those tight hairpins or double-s corners, which can be improved by providing the agent with future information about the track’s shape as was similarly done for the work on Sophy.

2.2.7 Grand Theft Auto V

Grand Theft Auto V (GTA V) is an open world adventure game that involves riding vehicles and is why some of have chosen to use GTA V for RL such as the work by Kinsley and Kukiela (2017). Their method relies on utilising a mixture of masking the screen, applying the Canny Edge algorithm to detect borders of objects such as the road lines, and then utilising Hough Lines algorithm to complete and enhance the imperfect road lines from the Canny Edge as shown in Figure 14.

Figure 14 - Canny Edge and Hough Lines in GTA V (Kinsley, 2017)

The method for detecting the vehicle's relative position is similar to that of other real-life application work such as that shown in Figure 15.

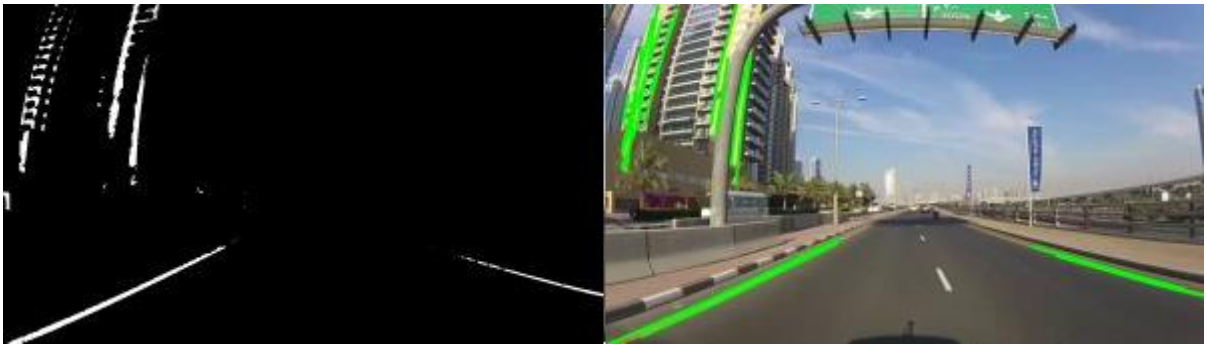


Figure 15 - Road Detection by Canny Edge and Hough Transform (Canu, 2023)

2.2.8 Doom

Doom is considered by some to be the most famous game even amongst non-gamers, and one of the earlier 3D based first person shooters albeit with animated 2D sprites for enemies (Thubron, 2021; Shoemaker, 2006), and thus from a 3D graphical maturity stage one may consider some similarities to GT. A study on the use of RL on Doom using raw pixel data was therefore relevant to review, since such 3D map environments have large state-spaces which are not entirely observable (Hafner, 2016). In case of a race in GT, one may argue that the environment is only partially observable, as one cannot see when an opponent vehicle is far up the road, or the shape of the future corners, however racing normally involves memorising the track, and time-trial does involve other any other agents or vehicles. Thus, it can be said that Doom has some additional challenges to overcome. Another similarity to GT is the sparse and delayed reward, in that the reward to a human player is at the end of a level or race. To overcome this, the authors utilised the death match mode of the game (the goal is to defeat as many players before a time limit or being defeated), and in such a mode, they shaped the reward such as to provide a positive reward for each enemy defeated. There were no negative rewards used, and neither were there any rewards at the end of the episode (Hafner, 2016, p. 23).

The author utilised down-sized, greyscale screen frame images as stacked frames (Hafner, 2016) which is like the DQN pipeline applied to Atari games (Mnih et al., 2013). The authors also experimented with providing frames that only contained the difference between frames (referred to as delta-frames) by the author. The author also reduced the action-space to speed-up learning. The unique discoveries from the work were that delta-frames did not add value in conjunction to frame stacking, and that an agent trained using DQN and an agent trained using Asynchronous Advantage Actor-Critic algorithm using Long Short Term Memory Networks (LSTM-A3C) had similar limited success at shooting forward facing enemies. The former observation suggests that stacked frames contain the necessary data that would be determined from delta-frames, and the former observation suggests that a stacked observation encodes sufficient information to satisfy the MDP criteria making the LSTM unnecessary.

2.3 Literature Survey Summary

From the work found in literature, one can find similar problems, with different boundary definitions and solutions. From an engineering viewpoint, while almost all seem to utilise Python for the training, some are using a direct access APIs such as the work done on GT Sport, whereas others have adapted existing emulators to provide the required interface, and a final subset developed a custom interface to provide a Gym or Gymnasium compliant environment. A variety of RL algorithms were used; however, the most commonly encountered ones were actor-critics, DQN and PPO (and variations of them).

Almost all work utilised frame skipping or pausing of the environment, whereas only the work on TMRL seems to have documented attempts to focus on real-time problems.

The form of input for RL are various, from pure parametric features as a fully connected layer direct from the game environment, to fully screen vision-based inputs utilizing a CNN. In between there are also pre-processed methods from the screen to create a LIDAR-like method as well as road edges by means of Canny Edge and Hough Lines.

Chapter 3 Technical Setup & Methods

In this chapter provides a description of the different technical elements for setting up the RL pipelines, as well as well as the boundaries that need to arbitrarily set in one way, or another are justified.

3.1 The Emulator

In literature there are 2 PSX emulators used in the domain of RL, which is a modified version of PCSXR (Purves, 2019b; Purves et al., 2019) and BizHawk (Costa, 2022; Severo, 2023a). To avoid being responsible to maintain a fork of a custom emulator as it is a form of re-inventing the wheel, a readily available and actively updated emulator was preferred. In doing so, it would make this work more accessible for future work by others. General purpose emulators such as BizHawk, might lack the focus of a dedicated emulator such as hardware accuracy. Nevertheless, the emulator selected would require features that are beneficial for reverse engineering, as is the case in BizHawk.

The emulator chosen was PCSX-Redux (Noble et al., 2023a) which is a collection of an emulator and some additional tools and libraries. The focus is on being hardware accurate (the authors even provide an opensource bios that is able to run on actual hardware if flashed to an EPROM and swapped to the PSX motherboard). PCSX-Redux provides a lot of additional features including emulation of the PSX serial port, GNU Project Debugger (GDB) server functionality for debugging custom code, memory and VRAM inspection, breakpoints, command-line flags, assembly viewer and a Lua engine that gives a lot of access to the insides of the game and redux (Noble et al., 2023b), most of these features will be critical to my project. A few of the features are described here.

Familiarity with PCSX-Redux from personal experience of using it as a development tool for PSX homebrew games, such as that of a CS50 Harvard project submission (Syed, 2020) and a game-jam development submission, Buzzy Bird (Syed, 2021), a natural biased towards the emulator may have existed. Finally, the core author of PCSX-Redux, is a good online friend, who has also supported me through serious health and life issues that have occurred in the time-frame of this project. The author of PCSX-Redux was kind enough to prioritise fixing or adding certain features to the emulator for the benefit of this project, such as the implementation of Protobuf (described in further detail in later sections).

3.2 Reverse Engineering

Before developing the RL environment, focused was made to reverse engineer GT. The minimum goal was to extract fundamental parameters such as vehicle speed, whereas ideally the whole game engine could be reverse engineered. However due to the time limitation, the focus was to achieve sufficient internal understanding to identify parameters that would be beneficial for an AI agent to perceive as part of the observation such as tyre slip, as well as internal game variables, such as the flag that detects the start and finish of a race.

The following descriptions are based on my knowledge through several informal discussions in related Discord groups or personal experience. Were feasible citations to sources have been provided but acknowledgment is made to potential inaccuracy of the claims (or subjectivity derived from personal experience). Games from consoles prior to the PSX mostly were written in assembly, directly exploiting the hardware capability such as those on the SEGA Mega Drive (Coding Secrets, 2020; Coding Secrets, 2021). On the other hand, the PSX was able to be coded in C with provided compilers based on early GNU GCC and libraries, but still required a lot of hardware understanding to exploit since a proper operating system is not available on the PSX. Basic functions such as saving your progress in the game, required custom code to store the game-save to the proprietary memory card (Gavin, 2020; Korth, 2022).

The co-founder of Naught Dog, the studio that released Crash Bandictoot themselves did not utilise the official libraries provided by Sony due to the limitations (Gavin, 2020). The PSX has a 2x speed CD-ROM and limited memory size – 2MB in total for code, video memory (VRAM) and sound memory (SPU) (Korth, 2022). To work around such limitations, assets are often loaded as binary overlays or compressed binary overlays, as is the case in GT. For decompiling, Ghidra was utilised. Ghidra is a software reverse engineering tool that can generate C code from a binary that also supports the MIPS CPU (NSA, 2023) which is utilised by the PSX. Ghidra's output would not be directly understandable due to the absence of labels. Figure 16 demonstrates that in some instances the generated C code can look like the original so long as it possible to anticipate the code structure based on an expected function. It was rarely the case, and the ways to overcome it are described in later sections.

Figure 16 - Own PSX Game Compiled and then Decompiled in Ghidra

The workflow for the achieved decompiling of GT was an iterative approach, at high level was made of the following steps. That is to first reconstruct the decompressed main executable binary of the game-engine to load into Ghidra to have a C-like decompiled version of the engine. Then the emulator would be used to identify variables in memory by searching in memory for parameters of known values such as vehicle speed, for which a reasonable variable type can be assumed. Using said variables, the Ghidra-generated code would be updated to better understand the functions as well as expose memory-structures such as the vehicle-state variables. Then using peek and poke methods, less obvious parameters would be manually understood through trial and errors. Whenever variable or parameter functions were understood, the Ghidra-generated code was updated in order to build a larger understanding of the game's inner workings. A description of the methods used are described in the following sections with examples.

3.2.1 Main Executable

GT is a game that seems to have exploited a lot of the PSX hardware features beyond the basic library. This is unsurprising since the developer studio Polyphony is part of Sony.

GT is split into 4 core binaries – GTOS.EXE, GTMAIN.EXE, GTEND.EXE, and GTMAIN.EXE. GTMAIN.EXE is the main engine loaded for a race (Squaresoft74, 2022), however the code itself is compressed. The original binary and resulting decompiled code are very compact (338 KB). It would then be discovered that the actual engine is decompressed at runtime (Squaresoft74, 2022).The algorithm seems to be a variant of LSZ according to one source (Baiter, 2017).

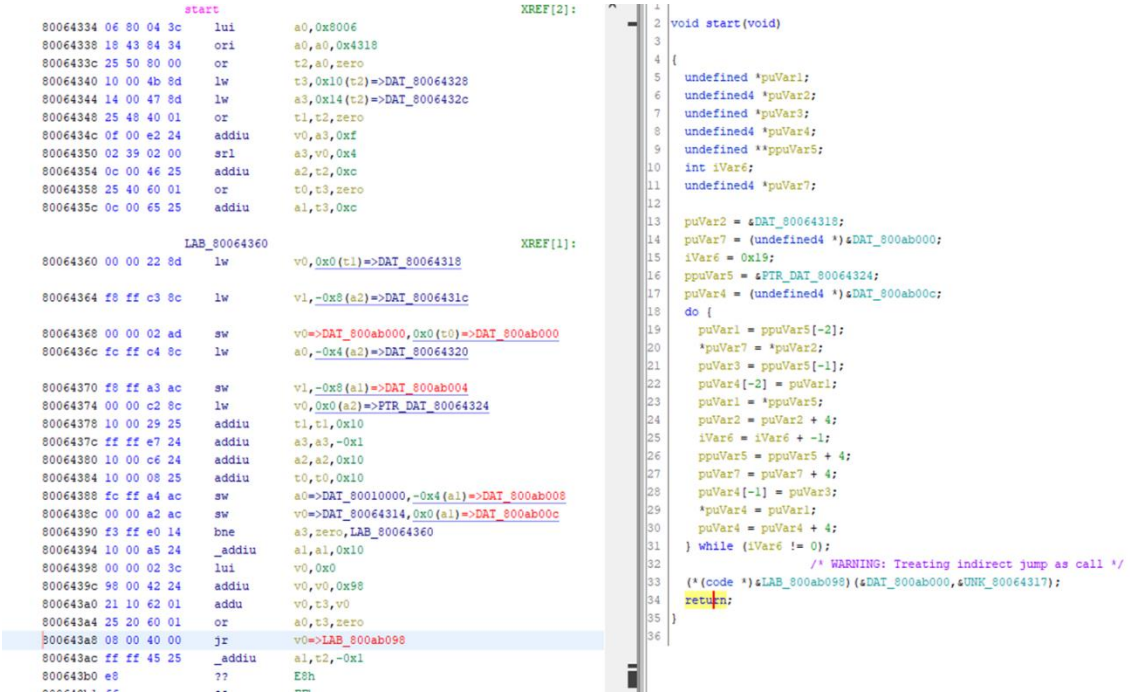


Figure 17 - GTMAIN.EXE Ghidra Decompilation

Even without fully understanding the mentioned code, it was possible to estimate from the decompiled code generated by Ghidra in Figure 17, that the looped layout sole use of pointers and variables that this is some form of data-reconstruction routine. Furthermore, all executable files for all region versions of the game, contain the string PSLZ (Figure 18), and 16bits further down is another constant that describes the size of the decompressed code.

```
                    P  S  L  Z DAT_80063bc8                                XREF[1]:     start:80063c10(R)
80063bc8 50 53 4c 5a     undefined4  5A4C5350h

                         DAT_80063bcc                                      XREF[1]:     start:80063c14(R)
80063bcc ff 37 0b 80     undefined4  800B37FFh
Size of decompressed code
                         DAT_80063bd0                                      XREF[1]:     start:80063c1c(R)
80063bd0 00 38 0a 00     undefined4  000A3800h
```

Figure 18 - Decompression Strings

By setting a breakpoint at 0x80010000 in PCSX-Redux (which is where most games seem to start actual execution from), it was possible to acquire directly a dump of the decompressed code from PCSX-Redux by utilising its web server API and requesting *http://localhost:8080/api/v1/cpu/ram/raw*.

From the raw-dump it was then possible to extract the code from address 0x1000 up to the length of the decompiled code, in the case of GTMAIN.EXE PAL edition that is 0xA3800 as shown in Figure 18. This block of code contained all the relevant code but would not be understandable by Ghidra or any PSX emulator since it is missing the header. Using a third tools such as PSFlab by Corlett (2011), it was possible to generate a generic PSX executable and paste the block of code at the typical PSX offset 0x8001000.

3.2.2 Identifying Parameters

There were several methods utilised to uncover parameters of the game.

3.2.2.1 Observing Memory for Known Values

Starting from low hanging fruits, the memory observer was used to search for most of the values displayed on the HUD outlined in red in Figure 19.

Figure 19 - Heads-up display

Using vehicle speed as an example, at an instant in the game when the displayed vehicle speed is 0 km/h, using the memory observer a search for a memory address containing the same value for an assumed variable type. The assumptions on the data type based on the possible logical values and the assumption that the developers were cautious with how they used the limited available memory. As an example, speed is never displayed as a negative in GT, and 8 bits is unlikely as it would be insufficient as the largest possible value would be 255, which is smaller than the highest speeds observed while playing. Therefore, the minimum would likely have to be unsigned 16 bits, since 32bit may be wasteful.

After an initial search such as the 0 value for 0km/h, multiple memory address would match. The game state was then altered by playing to alter the parameter to another value, such as 54 km/h. By searching through the initial list of address, it was then possible to find which address(es) would have changed to 54, as shown in Figure 20, which was found at address 0x800b66ec.

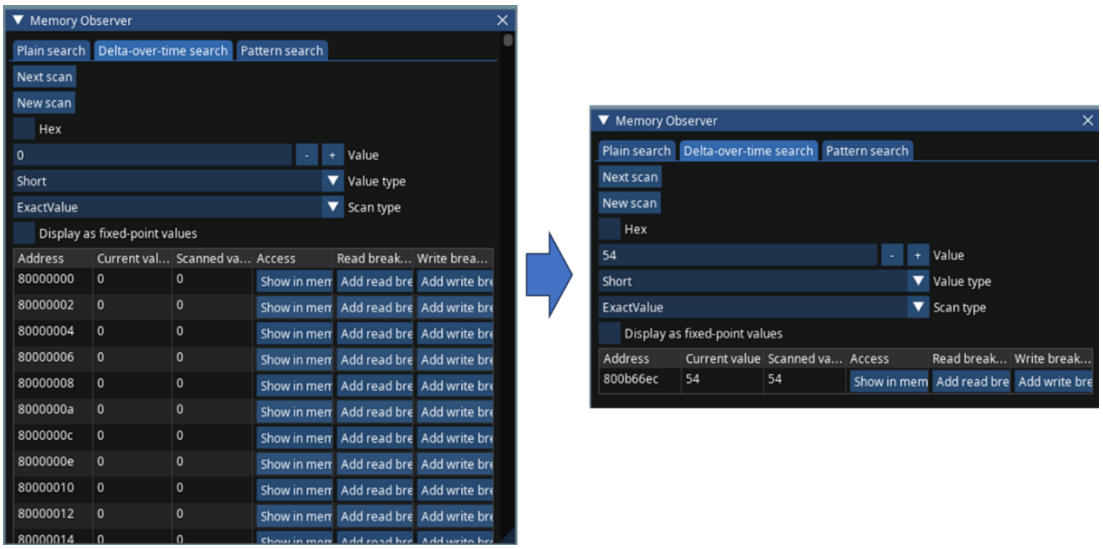


Figure 20 - Discovering Vehicle Speed

3.2.2.2 Reviewing Ghidra Generated Code

As more parameters were discovered through observation, and updating of the Ghidra-generated code, it was possible to make better educated guess at what other variables or functions are. An example would be the following snippet:

```
if ((DAT_800b6226 == 0) && (DAT_800bd990 < displayedVehicleSpeed)) {
  DAT_800bd990 = displayedVehicleSpeed;
}
```

Without understanding DAT_800b6226, it is recognisable that DAT_800bd990 is holding the maximum of the displayed vehicle speed.

```
if ((DAT_800b6226 == 0) && (hiddenMaxSeenSpeed < displayedVehicleSpeed)) {
  hiddenMaxSeenSpeed = displayedVehicleSpeed;
}
```

While searching for functions that read or write to the address of hiddenMaxSeenSpeed, it was possible to find the relevant code sections that displayed the end of race results. The max speed achieved during a race was only displayed for drag races but seems to be computed for any race.

3.2.2.3 Generating Widgets to Peek and Poke

In the previous example, the function was understood, but one data address was not yet understandable (DAT_800b6226). Using the Lua engine, scripts to display slider widget were created (as shown in Figure 21), that could be used to both display (Peek) and alter the value (Poke) utilising the code shown below.

```
function doSliderInt(mem, address, name, min, max, type)
    address = bit.band(address, 0x1fffff)
    local pointer = mem + address
    pointer = ffi.cast(type, pointer)
    local value = pointer[0]
    local changed
    changed, value = imgui.SliderInt(name, value, min, max, '%d')
    if changed then pointer[0] = value end
end
…
doSliderInt(mem, 0x800b6226, 'unknown', -50, 50, 'uint16_t*')
```

It was then possible to either observe the change in value while playing or slide the slider to force a new value.

Figure 21 - Example Slider

Often poking at memory to a value that the programmers did not expect would cause the game to crash. In this case, it was discovered that DAT_800bd990 is a form of race-state flag. Setting the value to 9, would bring up the race results graphics even if the race is still going (as shown in Figure 22). This would prevent the max speed variable from being altered before or after the race. This flag was used for this RL project to start and terminate episodes.

Figure 22 - Forced Race Results

Through more tests it was then possible to determine the enumerated meaning of DAT_800bd990 allowing for the code to be updated to:

```
if ((raceMode == Racing) && (hiddenMaxSeenSpeed < GlobalCar.currentSpeed)) {
  hiddenMaxSeenSpeed = GlobalCar.currentSpeed;
}
```

The reverse engineering of this section of code is provided in Appendix A as an example of a nearly completed reverse engineering of a function using the described methods.

3.2.2.4 Summary

By iteratively applying previous described methods it would be possible to gradually recognise structs due to code such as the following:

```
puVar6 = &DAT_800b66a0;
iVar7 = 0;
if (0 < DAT_800bbff6) {
  do {
    FUN_80020b5c(puVar6);
    iVar5 = 0;
    iVar2 = 0;
    do {
      bVar1 = puVar6[(iVar2 + iVar5) * 4 + 0xad];
      if ((0x7f < bVar1) && (iVar2 = FUN_8002e554(), iVar2 != 0)) {
        FUN_8002e600(iVar2);
        uVar3 = *(undefined4 *)(puVar6 + iVar5 * 0x18 + 0x7ec);
        uVar4 = *(undefined4 *)(puVar6 + iVar5 * 0x18 + 0x7f0);
        *(undefined4 *)(iVar2 + 0xc) = *(undefined4 *)(puVar6 + iVar5 * 0x18 + 0x7e8);
        *(undefined4 *)(iVar2 + 0x10) = uVar3;
        *(undefined4 *)(iVar2 + 0x14) = uVar4;
        *(ushort *)(iVar2 + 0x1e) = (ushort)bVar1 << 4;
      }
```

Referring to the above snippet, a pointer puVar6 points to the address of &DAT_800b66a0, in the loop, references to puVar6 multiplied by iVar5 number of offsets of 0x18, suggests that this is an array or more likely a struct. As more of the obvious struct fields were discovered, it was then possible to observe the memory addresses that correspond to possible struct fields and utilise the described methods to investigate such remaining unknown structs.

An example of this is the collision bitmask that was discovered to be 4 bits long, where each bit corresponds to each of the four vehicle corners in contact with the wall of the track as follows:

```
0001 = Front left
0010 = Front right
0100 = Rear left
1000 = Rear right
```

Combination of any two of those bits would indicate a face collision (example 0001 + 0010 = 0011, which is a front impact).

The journey to reverse engineer is one that was not complete, however almost all of the common parameters used in literature were discovered and monitored utilising widget elements as shown in Figure 23. Parameters such as wheel slip, tyre position (on road, kerb, grass or dirt) were exciting and opening more can of worms than anticipated as it allowed for many more “what if?” questions for considering in the RL problem.

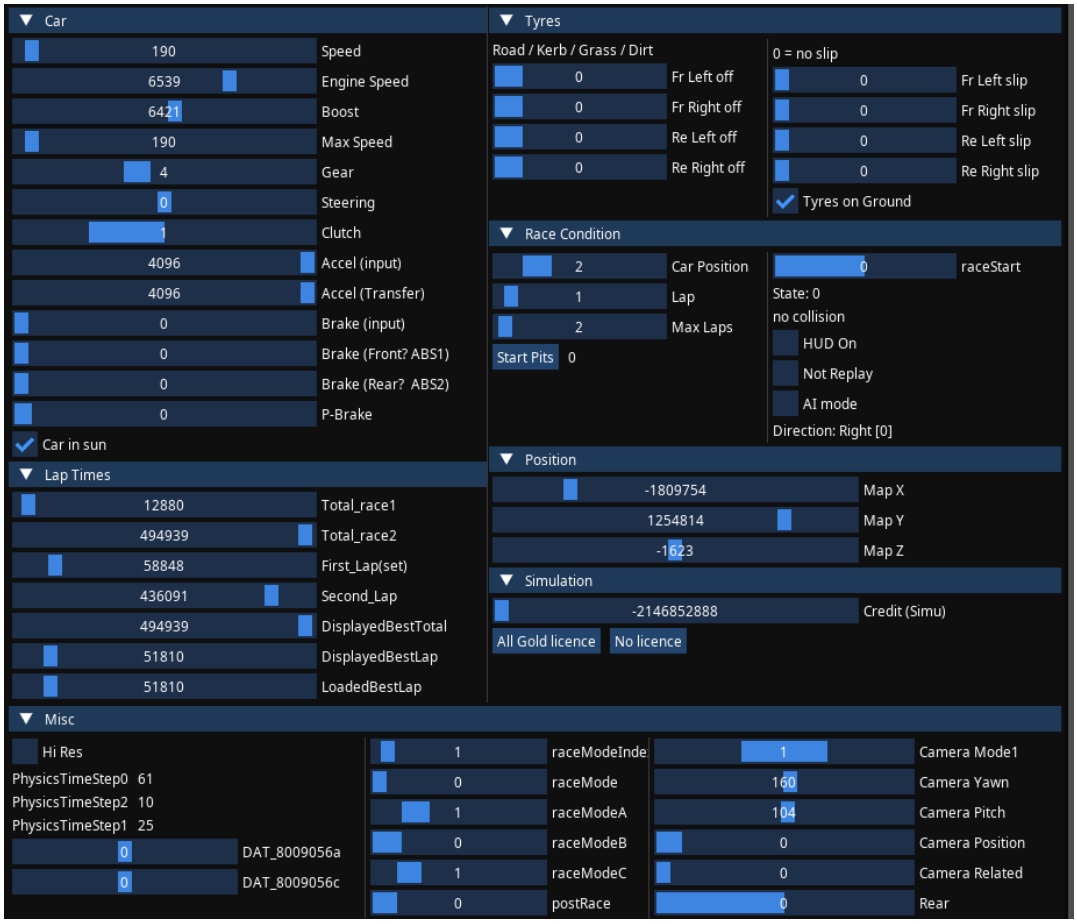


Figure 23 - Reverse Engineered Parameters in Widgets

Other flags discovered, such as the built-in AI flag, or the HUD display flag, were useful for other activities such as generating a track progress and centreline like co-ordinate index (described in later sections) of the track for similar purposes as those done in GT Sport and TMRL (described in the literature review).

3.3 Canny Edge

Based on the work in literature, it was noted that minimising the state-space should be beneficial in RL (Fletcher and Mortensen, 2018; Fletcher, 2017; Costa, 2022; Wurman et al., 2022; Song et al., 2021; Fuchs et al., 2021), therefore there was hesitancy to rely on the raw screen data (albeit finally implemented successfully).

Examples such as those on GTA V (Kinsley and Kukiela, 2017; Kinsley, 2017), seem to work on real car road footage (Canu, 2023), suggest the use of simplifying the image through CannyEdge(OpenCV, 2023a) and Hough Line Transform (OpenCV, 2023b) from the OpenCV library, which could then perhaps be used to form LIDAR like processing as was done directly in the LIDAR mode of TMRL (Bouteiller, Geze and GobeX, 2023b). It was assumed to be necessary in the absence of the ideal parameters extracted directly from the game engine like the work on Sophy (Wurman et al., 2022)and GT Sport (Song et al., 2021; Fuchs et al., 2021). Unfortunately, this method would fail my attempts, perhaps due to some unique difficulties with the PSX hardware.

3.3.1 The Problem

The problem was not analytically defined, but empirically investigated and subjectively concluded that there are three core difficulties.

With reference to the left capture in Figure 24, the first issue was that the pixelated and stretched textures on the road surface cause a lot of false edge detection since there are some intensity gradients. The second issue shown in the right capture in Figure 24, is that trying to adjust the Canny Edge parameters to avoid detecting those gradients, causes issues on situations such as the first corner of the high speed ring, where a strong edge is detected on the shadow-line rather than the edge next to the yellow and black wall warning. The third issue was that the low resolution of the PSX means there are several discontinuous lines, such as the blue and white billboard fence on the right of the start finish-line, or the transparent mesh-fence on the left of the same track section.

This was seemingly a unique problem in comparison to the simplicity of implementing LIDAR on TMRL (Bouteiller, Geze and GobeX, 2023b).

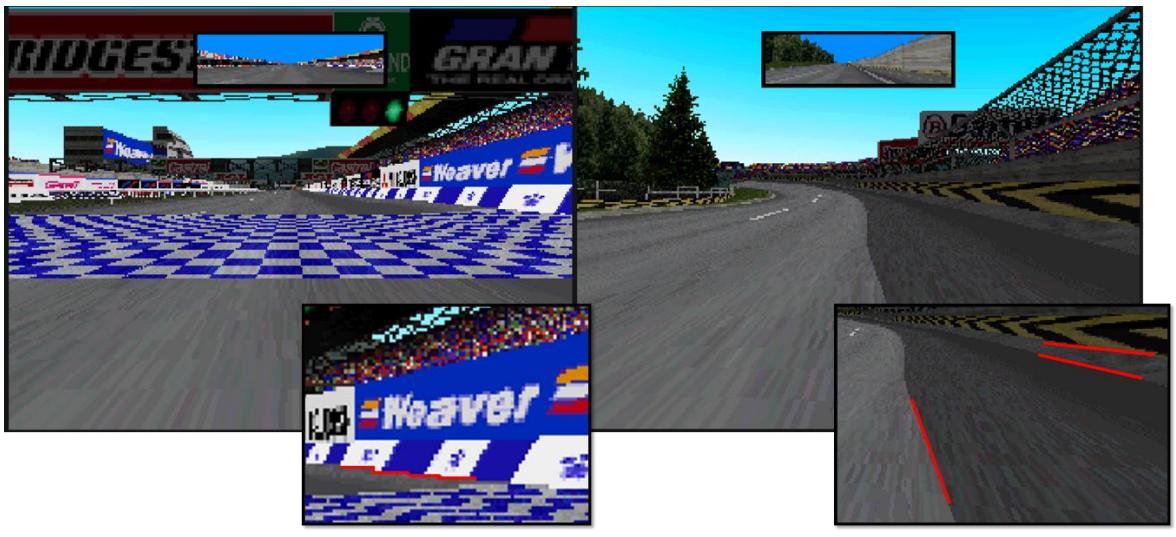


Figure 24 - High Speed Ring Start-Finish (left) and the First Corner (right)

3.3.2 Attempts to Process the Screenshot

Several attempts to compensate for this problem were made, including modifying the thresholds, working with grey images, applying additional gaussian blurs with different kernel size equalising the greyscale image and altering the order of normalizing and blurring.

An evaluation and comparison with another emulator with upscaling features (DuckStation) was also used as shown in Figure 25. The assumption was that a higher resolution image would generate less discontinuous lines which should be beneficial for detecting edges and cause less pixelation of textures.

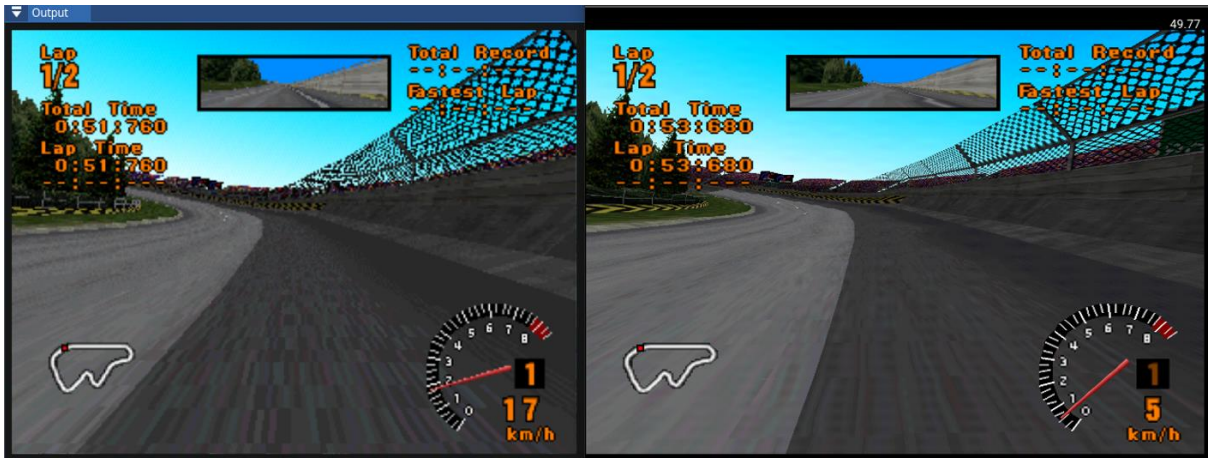


Figure 25 - PCSX-Redux (left) compared to an upscaled DuckStation (right)

DuckStation's upscaling shows a slight alleviation of the issues but was not able to solve the problem consistently in all parts of the track as shown in Figure 26.

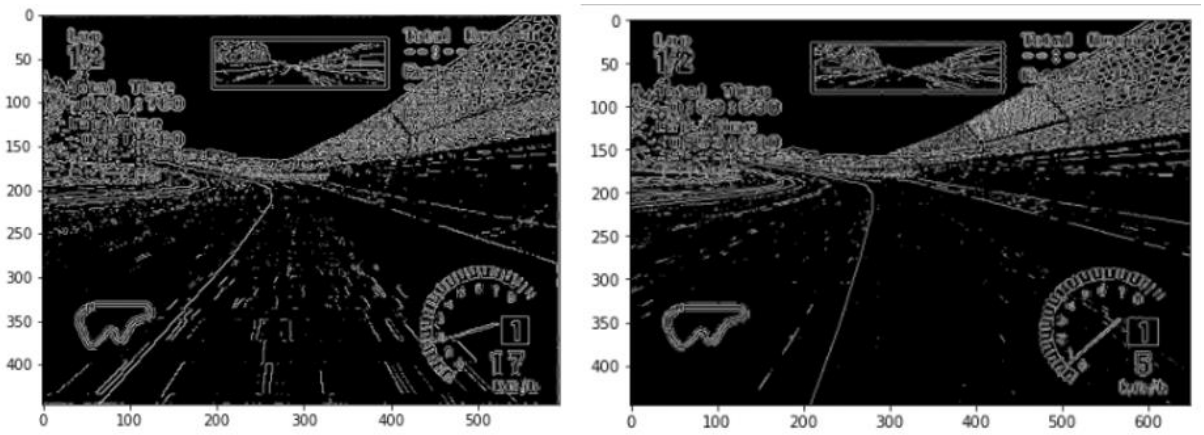


Figure 26 - CannyEdge on the first corner using PCSX-Redux (left) against DuckStation (right)

A brief summary of some of the attempts at adjust the Canny Edge parameters on both PCSX-Redux and DuckStation is shown in Figure 27. The first two rows show the start finished, and the last two rows the first corner. Original refers to the PCSX-Redux screenshot, whereas the upscaled image refers to DuckStation. One can visually observe that the condition that eliminates the noise on the start-finish made it harder to impossible to detect the wall of the first corner. Subjectively it seemed that an upscaled image performed marginally better, indirectly confirming that low resolution of the PSX may be contributing to this problem. More comparisons are provided in Appendix E .

Figure 27 - Comparison of some of the Parameters of Canny Edge

Relying on DuckStation's upscaling would hinder the possibility of applying this work's results to real hardware in the future. Thus textures modification to reduce the impact of this issue was considered.

3.3.3 Texture Modification

PCSX-Redux provides the ability to use the webserver to POST. One such POST API is (Noble, 2023c):

/api/v1/gpu/vram/raw?x=<value>&y=<value>&width=<value>&height=<value>

The PSX sprites or textures utilise colour palettes, which are known as colour look up tables (CLUT) and are stored separately from the texture itself in VRAM (nocash and Noble, 2023). An example is shown in Figure 28.

Figure 28 – VRAM and CLUT are of various textures

Through the POST API it was possible to upload a custom image to the video memory (VRAM) of the virtual PSX to either overwrite the CLUT, or the texture itself in such a way that the texture image is appeared as single flat colour. The Python code to overwrite the VRAM for this purpose (with help from a PSX Discord server member familiar with the PSX texture colour encoding, known by their alias Spicyjpeg).

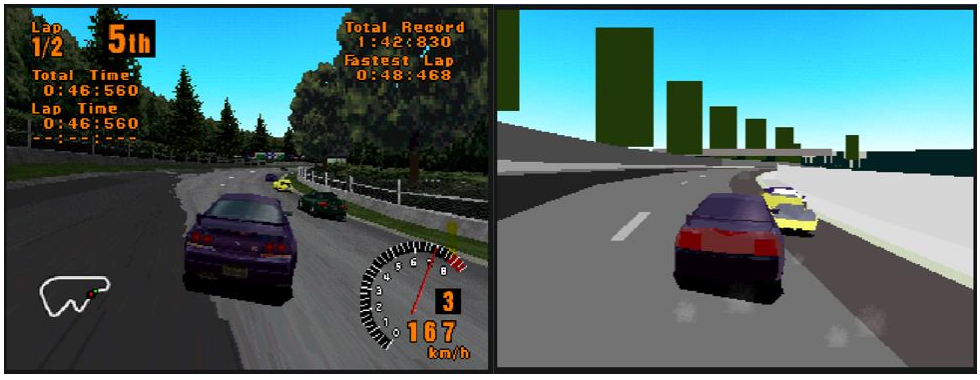


Figure 29 - Flattened Textures

This method produced the results as shown in Figure 29, which makes the game look as if it were running on an older generation of consoles. It was expected that with a bit more effort it would have been possible to alter the colours to make it convenient to detect edges through the Canny Edge algorithm, but this approach was withdrawn as it was deemed as a major departure from the goal of focusing on GT.

3.3.4 Summary

Having failed to implement a method of detecting the track edges from the visual data meant that there were only two possibilities to pursue.

The first would be to rely on purely in-game parameters which is proven to work on GT Sport (Wurman et al., 2022; Fuchs et al., 2021; Song et al., 2021), however it was anticipated that this would be unrealistic given the level of reverse engineering achieved or perhaps at all achievable within the time-frame.

The second would be to rely on the screen display, which is also proven to be successful, such as TMRL (Bouteiller, Geze and GobeX, 2023b) and WRC 6 (Jaritz et al., 2018) works described in the literature review.

The assumption was that the augmented observation would be required for general accuracy of the observation due to the inferior resolution of GT and the elastic time-step that will occur from threating the problem as a real-time problem.

3.4 Screen Capture

To implement an agent that relied on the screen image (or with some post processing) in conjunction with a CNN or by utilising some measurements, such as the LIDAR TMRL method (Bouteiller, Geze and GobeX, 2023b), it was necessary to establish a way of capturing the screen.

It was initially assumed that the matter would be trivial since screenshot libraries exist and have been used for similar projects such as those on GTA (Kinsley and Kukiela, 2017). Two of the most common libraries for screen capture were the Pillow/PIL library's ImageGrab, or pyautogui's screenshot methods. However, initial attempts at capturing the screen and re-rendering them in a window using cv2.imshow seemed to render at a much slower frame rate compared to the original emulator's output. This would mean that even without the computational overhead of the NN and RL, significant computational power would be consumed consume on this foundational matter. Tests were performed to evaluate the impact of the different libraries and methods.

3.4.1 Available Libraries

Concerned with speed of screen capture, in addition to Pillow (PIL), two additional libraries focusing on screenshot performance were found, mss and d3dshot (which internally had more than one way of doing the task). Through sequentially benchmarking in automated test, it was possible to estimate the frames per second (FPS) based on the average of 120 frames. While this would not objectively indicate variance of the method, it would give a reliable way of comparing the methods.

The results of the benchmark (code used in 0Appendix B – Screenshot Benchmark) are provided in the following table:

Table 1 - Comparison of Screen Capture methods on a 50 FPS Game

Method	*Average FPS*
PIL	26.6
pyautogui	25.5
mss	40.8
d3d_sc	53.6
d3d_buff	20.2

It was meaningful to search for another method such as d3dshot, since the d3d_sc method was about 30% to 200% the performance of mss and PIL respectively. However, the results still showed potential slow down for the relatively trivial task of observing the screen.

Furthermore, a practical inconvenience with screen captures is that they tend to require the coordinates and size of the rectangle to capture from the monitor area, which would have to be re-adjusted if the emulator window were moved. Additionally indexing monitors in a multiple order seemed inconsistent depending on the way monitor ordering is configured. Finally, the best performing method of d3d_sc, could only be utilised in a single instance, attempting to run the same

method twice would provide the warning *"Only 1 instance of D3DShot is allowed per process! Returning the existing instance..."* thus preventing the use of this method for running parallel environments on the same machine (an option that was desirable, but found to be not possible due to other issues.).

Considering that it would be necessary to also retrieve the memory data from the emulator to the Python environment to augment the vision-based observation similar to the TMRL solution (Bouteiller, Geze and GobeX, 2023b) or the work of Street Fighter 2, an alternative pipeline would be necessary anyway, so that pipeline was then developed to transfer both images and parametric values. Most of the work covered in the literature review relied on some form of TCP socket transport such parametric and/or visual data.

3.4.2 TCP Sockets Way

The use of a sockets to communicate over a TCP is not novel even in this domain of RL for game environments. The high-level concept was to implement the observation exchange according to Figure 30.

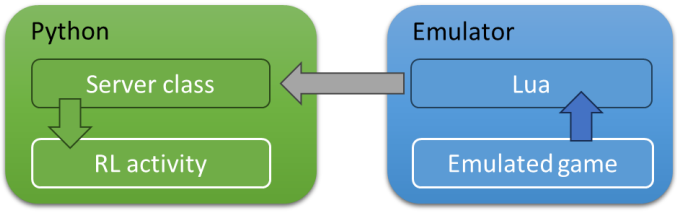


Figure 30 - Basic Conncetion for Observation

However, most methods are primitive in that they rely on pausing the emulator, so the emulator would only send a packet of data containing an image or parameters or both, when the Python environment is ready for it and after the Python environment has computed the action is the emulator allowed to advance, and since often frame-skipping is introduced, more time is elapsed between frames that rely on the exchange of data (Severo, 2023b; Costa, 2022). Emulation pausing, if noticeable is not akin to a normal human experience, and if unnoticeable it would be a result of the brute-force computational power the machine happens to have which may go against the real-time problems that humans and real environments face and robots face.

The development direction was to allow for frame-skipping but avoid it if necessary and to avoid synchronisation between the Python environment and the game.

The first implementation naively transmitted a frame of data including the screenshot at best at each frame or every n^{th} frame.

This could be implemented by a Lua script function that is run at the end of each render function call in the emulator (which could thus be executed each game-frame). The screenshot transported is the native display render output from the emulated PSX. This means that there is no significant overhead to capture the screenshot, and that the size is the true pixel size as if running on the PSX hardware, independent of the actual scaled the emulator window in Windows.

The result was a smooth synchronised frame exchange to Python, so long as the Python environment is idling in between frames. This would later be found to be inappropriate while adding computation workload, as this allowed the python environment to fall behind in receiving observations from the emulator. Python would process the stream of data in a first-in, first-out (FIFO) manner, which would

be problematic since the Python environment would be observing a passed environment state, and actions an agent would take would not be applied to the current state of the environment rather than the observed one. An example as an external appendix is provided at https://youtu.be/-YR5Gd1PeFI. If the Python environment were consistently slower than the emulator, this lagging effect would get progressively larger, further worsening the problem.

Therefore, development approach had to be altered such as read all data from the socket stream overwriting the latest frame buffer in Python. The framebuffer is decoded and utilised only used when an exception is raised due to the stream being empty. In addition, the emulator would not attempt to transmit data unless it has received a ping message to avoid filling up the stream unnecessarily but would not pause the game if a ping is not received. The data of each frame transmitted would be constructed as shown in the following figure.

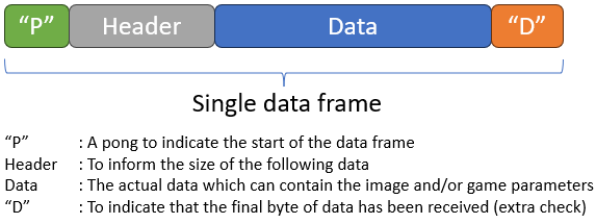


Figure 31 - Single Data Frame Description

For a continuous stream, the framebuffer state could be represented as shown in Figure 32 (the diagram is simplified since the server-side response is omitted, which was later developed to act as a ping or as a game-save state load request). The result of this method was that the Python environment would only process the last complete frame received, which would then be decoded using Protobuf (described later) and transferred to the RL environment. An example of this as an external appendix is provided at https://youtu.be/Pvm_LA9sfGA. Additionally, the way that the server received the 'data' section of the frame, could be in any chunk size, which in the ideal case can be within one message, or as part of several messages.

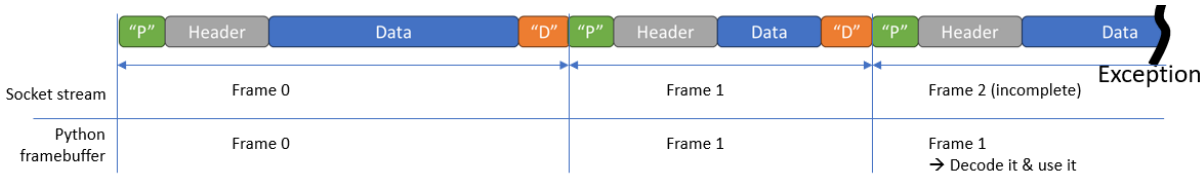


Figure 32 - Python Framebuffer against Stream of Data

Thanks to this implementation, with the emulator running at 49.99 FPS, it was able to measure consistent frame rates close to that of the native display of the emulator even when re-drawing the frame in Python as shown in the following table. Note that these results in comparison to those from table 1, include additional overhead for decoding the Protobuf data (described later).

Table 2 - Performance of Socket Method

Re-Render the frame	*120 frames*	*500 frames*
Not Rendered	49.3	49.8
Rendered	49.7	49.9

In parallel to this activity, implementing Protobuf decoding on the server side to encode the data in a struct like manner was necessary to easily alter the quantity and contents of parameters transported without hard-coded relative byte positions.

3.4.3 Protobuf

Protocol Buffer, also known as Protobuf, is a language agnostic way of serialising data. Libraries to encode and decode data are provided for by Google for multiple languages including Python, but not Lua. Protobuf, requires a language specific definition to decode messages. Such definitions are created using the proto compiler for the target language. An advantage of Protobuf is that the encoded data for transmission can be smaller than the original data type, since 0 values are omitted from the message (Google LLC, 2023). Conveniently Noble (2023b) added a third party Protobuf library from Wang (2023).

Other socket implementations such as that used for MegaManX (Severo, 2023b), define early on the observable parameters, or modify the server-client relevant code for the exact byte size and sequence of data exchanged. Relying on Protobuf allows the possibility to easily modify which parameters and how many parameters are transmitted. In my method, the server does not need to be hardcoded for the correct number or type of parameters being received, this is instead handled by the protobuf definition which is compiled once. An example of one such Protobuf definitions used in my work is provided in Appendix C – Protobuf Definition. Furthermore, Protobuf uses dictionaries to create struct-like data structures.

For the reason that the content of the message was not pre-constrained at this stage of development, and because the size of a Protobuf message can vary (due to overhead and due to the non-encoding of zero-values), it was necessary to include a header to inform the server the size of data for each message as shown in Figure 31, and is why the stream in Figure 32 shows different data sizes for sequential messaged frames.

The Lua code for the transmission of data and the server-side Python code for receiving data is provided in Appendix D – Server & Client Connection. This was later evolved further to allow for certain emulator setting manipulations, however the concept remained the same. Note that the server side has also additional functionality, such as the ability to receive exactly one frame (used for debugging) or continuously receive frames. There are additional helper functions to decode the screenshot data.

3.5 Control

There were two main control-flows to implement, the first being to tell the emulator to load a save state of the game at the start of training episode as part of the Gym API's reset state (more details in a later section), and to convert the agent's actions into in-game vehicle actuations. The former was done by altering the ping-pong exchange between the server and the emulator. Specific ping-characters would trigger the emulator to load a state instead of preparing data for transmission.

For the agent's controls, utilising the TCP interface to transmit gamepad states to the emulator were but were later abandoned in favour of the same solution as that of TMRL (Bouteiller, Geze and GobeX, 2023b) which utilised the virtual gamepad library (Bouteiller, 2023b). The main reason for doing this is that the agent would be able to actuate the controller at any time of the main game-loop. This might mean that a new action, is not recognised until the following main game-loop, since the game might only check the controller's state at a given instance in each main game-loop. While this might seem like a disadvantage, which is different from how a human player is interacting with a PSX. A human action is likely to persist until they update their action, and the start of an action is unlikely to be frame perfect (the action could be as late as nearly 1 game-frame). Furthermore, by emulating a PlayStation DualShock 4, the emulator had the correct default gamepad bindings.

The complete server-emulator and operating system (OS) interface is visualised by Figure 33.

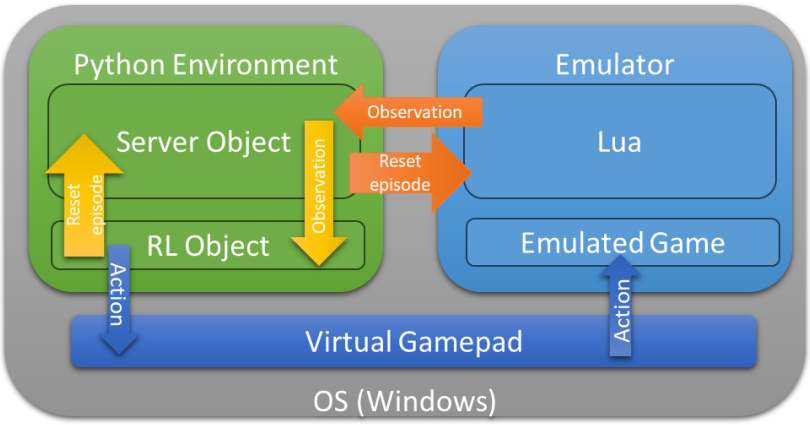


Figure 33 - Server-Emulator-OS Communications

Through this library it was possible to utilise both analogue controls such as the analogue sticks or digital ones such as the directional pad. Both control methods were studied in the experimental part of this work.

It would later be found that the virtual gamepad library in combination with PCSX-Redux would make it impossible to deploy multiple rollout workers on a single machine. This is because each virtual controller would appear under the device manager with the same hardware identifier and description. Thus it would not be possible to detect which controller belongs to which rollout worker and it would also not be possible to make sure that PCSX-Redux was using the correct controller.

3.6 Generating a Track Pseudo-Centreline

In the traditional time-trial racing sense, the reward is the lap-time at the end of the race. To reward an agent in this manner would be far too sparse to have efficient learning.

This is probably why several authors (Jaritz et al., 2018; Bouteiller, Geze and GobeX, 2023b; Wurman et al., 2022; Fuchs et al., 2021; Subramanian, Fuchs and Seno, 2022; Neinders, 2023) dealing with a similar racing problem all required a method for determining how far along the track a vehicle was. This is necessary to generate a dense reward function.

The centreline is generated in the work on GT Sport (Wurman et al., 2022; Fuchs et al., 2021) or WRC 6 (Jaritz et al., 2018) are likely given by the game-engine's API. The authors of the TMRL documented their method for generating this without such an API. Instead, a human player is required to drive a lap, during which the vehicle's position (as x-y pairs) is recorded in what is referred to as a "demo trajectory" as shown in Figure 34. The trajectory is recorded at a fixed timing interval, which means the data points themselves are not equally distanced due to the varying speed. The data points are then re-processed to generate interpolated equidistant points. Then an index is used to track which pair of x-y co-ordinates are closer to the vehicle's current position. This is computationally lighter than calculating the distance between the vehicle and all x-y co-ordinates in the table.

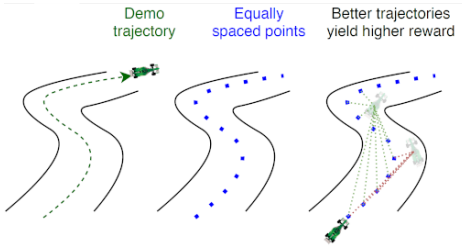


Figure 34 - TMRL Centreline Generation Method (Bouteiller, Geze and GobeX, 2023b)

The pseudo code of how the equidistant points are as follows (ignoring the start of the race):

```
# index is last track position relative to equidistant points table.
Loop until race is over:
        distance_1 = calculate distance between vehicle position and x-y pair at [index-1] position
        distance_2 = calculate distance between vehicle position and x-y pair at [index] position
        distance_3 = calculate distance between vehicle position and x-y pair at [index-1] position
        if distance_1 < distance_2:
                index = index -1 # going backwards
        else if distance_3 < distance_2:
                index = index +1 # going forward
```

A similar method was developed for this work on GT. However, instead of asking for a human player to drive round the track, the flag identified through reverse engineering for activating the game's default AI was utilised to drive around the track. To avoid too-sparse, the engine-speed variable was also hacked to force the car to drive at a low enough speed such that the simple-AI would drive at a quasi-constant speed.

Unlike the examples provided in TMRL (Bouteiller, Geze and GobeX, 2023b), the High Speed Ring is a looped track which provides an additional unique challenge. The race starts on lap 0, before the start-finish line, and the race ends on the start-finish line of lap 2. For simplicity all measurement points from lap 0 till the end of lap 2 as a one continuous event.

Post-processing was then applied to interpolate between data points in order to generate a denser equidistant check points along the pseudo-centreline, similar to the methods utilised in TMRL (Bouteiller, Geze and GobeX, 2023b). An example of the resulting effect of this processing method is shown in the following figure.

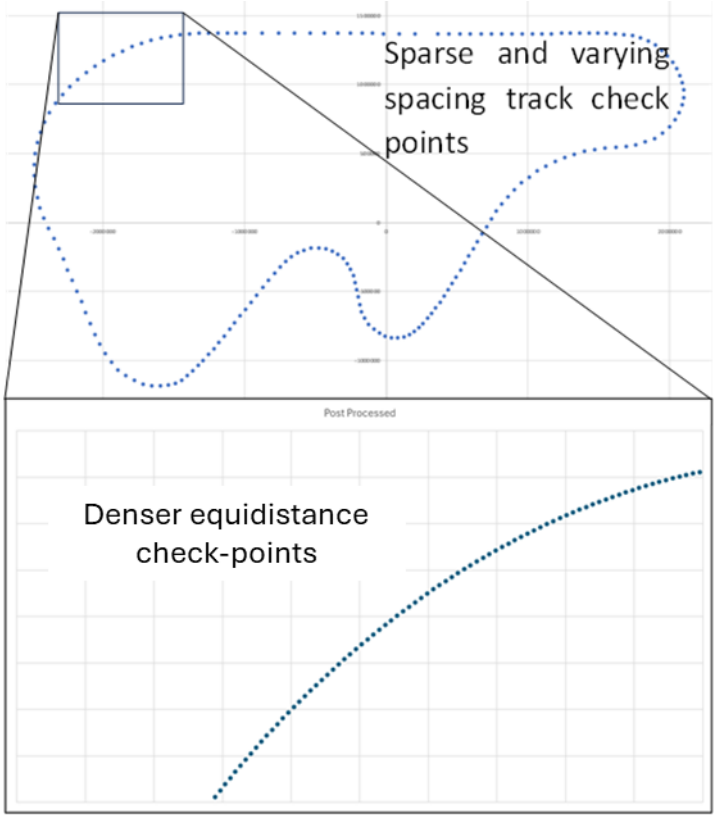


Figure 35 - Post Processing of Track Checkpoints

3.7 Reinforcement Learning and Frameworks

It was not within the scope of this project to study the detailed implementation of different RL algorithms, but rather to evaluate RL algorithms on the problem of time-trial in GT. Therefore, a review of available frameworks that provided the expected reinforcement learning algorithms. In terms of standardisation, it was intended that this work will follow the Gymnasium API through the rtgym library which will be described first.

3.7.1 Realtime Gym

The rtgym library is written in Python and inherits the class methods from the Gymnasium library. Some of the methods have to be manually created for custom environments as was the case with GT. The rtgym library automatically provides the clocked functionality for the elastic time constraints. The class methods that have to be implements are: *__init__(), get_observation_space(), get_action_sapce(), get_default_action(), send_control(), reset(), get_obs_rew_terminated_info()* and *wait()*. The *render()* function was also implemented for debugging.

All methods were implemented such that at runtime, a configuration can be used to determine:

a) The track to utilise for training (the high-speed ring or the straight-line drag strip)
b) The algorithm's restrictions (mainly the use of continuous or discrete control space)
c) The reward function (in case of the drag strip maximising forward speed in all steps might be enough)
d) The observation space dimensions (to test various combinations of parameters).

A lot of helper functions and classes had to be added to support with receiving the observation from the emulator via the server class and decoding of the Protobuf data as well as instantiating the controller with the appropriate analogue or digital setup.

A distinction of a rtgym environment is that the action and observation spaces are tuples which differs from Stable-baselines3 (Stable-baslines3, 2023), which requires a dictionary instead.

3.7.2 Frameworks Considered

For the final experimentation work, Ray and RLlib was utilised. It is a framework that is meant to provide machine learning capabilities without having to be too conscious of the inner workings that allows for scaling over distributed systems without having to develop the necessary code (The Ray Team, 2023a).

Another feature of the Ray framework is that of Ray RLlib which provides the necessary APIs for RL on a wide variety of problems such as multi-agent problems or training from offline data, furthermore RLlib supports and provides algorithms for TensorFlow 1.x and 2.x as well PyTorch (The Ray Team, 2023b). There are further packages and modules as part of the Ray framework that seem attractive for future work, such as Ray Tune, which allows for experiment definitions to help tune hyperparameters (Liaw et al., 2018).

Other frameworks were considered, such as the Stable-baselines3 (Stable-baslines3, 2023) however Stable-baselines3 and most Gymnasium default environment examples are not naturally capable of dealing with real-time environments since with all the examples provided, the environment steps are controlled by the RL algorithm, that is to say the environment does not change until the step() method is not called on the environment. By the authors of Stable-baslines3's publication it can be seen in table 3 (Raffin et al., 2021, p.4) that RLlib is actually more frequently updated than Stable-baslines3, the only drawback is the lack of pretrained models, which is not a concern in this case since the GT problem as a real-time problem is very unique.

Table 3 - Comparison of Stable-Baslines 3 with other Frameworks

	SB3	OAI Baselines	PFRL	RLlib	Tianshou	Acme	Tensorforce
Backend	PyTorch	TF	PyTorch	PyTorch/TF	PyTorch	Jax/TF	TF
User Guide / Tutorials	✓/ ✓	✗/ –	–/ ✓	✓/ ✓	–/ ✓	–/ ✓	✓/ –
API Documentation	✓	✗	✓	✓	✓	✗	✓
Benchmark	✓	✓	✓	✓	–	–	–
Pretrained models	✓	✗	✓	✗	✗	✗	✗
Test Coverage	95%	49%	?	?	94%	74%	81%
Type Checking	✓	✗	✗	✓	✓	✓	✗
Issue / PR Template	✓	✗	✗	✓	✓	✗	✗
Last Commit (age)	< 1 week	> 6 months	< 1 month	< 1 week	< 1 month	< 1 week	< 1 month
Approved PRs (6 mo.)	75	0	13	222	85	5	7

Another argument to focus on RLlib is that it also supports Unity 3D environments (The Ray Team, 2023b), which is in line with one of my personal objectives of acquiring the knowledge and experience for the possibility of applying it towards commercial games in the future. RLlib is built with the possibility for real-time in mind because it comes with an additional server/client setup for interacting external environments in which:

"*it does not make sense for an environment to be "stepped" by RLlib… [or] external simulators (e.g. Unity3D, other game engines, or the Gazebo robotics simulator)*"

(The Ray Team, 2023b, External Agents and Applications section)

Therefore, Ray seemed to be a viable for the purpose of being restricted to real-time externally actuated environments other than manually implementing algorithms as the authors of TMRL (Bouteiller, Geze and GobeX, 2023b).

In Both Ray and the TMRL pipeline rely on similar concepts, a trainer performing the reinforcement learning from the observations received, rollout workers which performs actions on the environment, and a server connecting the worker to the trainer for sending back observations to trainer, and for receiving the revised policy.

The Ray framework was tested on a simpler problem in GT. That was to train an agent to drive in a straight line on a drag track (a track and type of race where the goal is to race in a straight line). The setup utilised a single worker that synchronously sent samples to the trainer and synchronously received revised weights as shown in Figure 36.

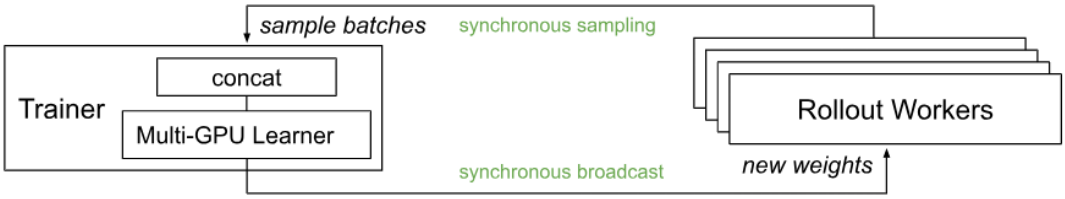


Figure 36 - RLlib PPO Worker and Trainer (Ray, 2023a)

The agent was given the ability to accelerate, brake and steer. Utilising the PPO algorithm and only parametric observations together with a very naïve reward that was proportional to the vehicle speed, the agent was able to learn that the best policy at any state is to press the accelerator. Figure 37 demonstrates an example of success. It can be seen that after 300 thousand training episodes, the minimum reward seen in an episode stabilises to a high positive value (top diagram) while the episode length average is diminishing to a stable value of around 320 game steps, because the agent is racing to the finish line faster and consistently so.

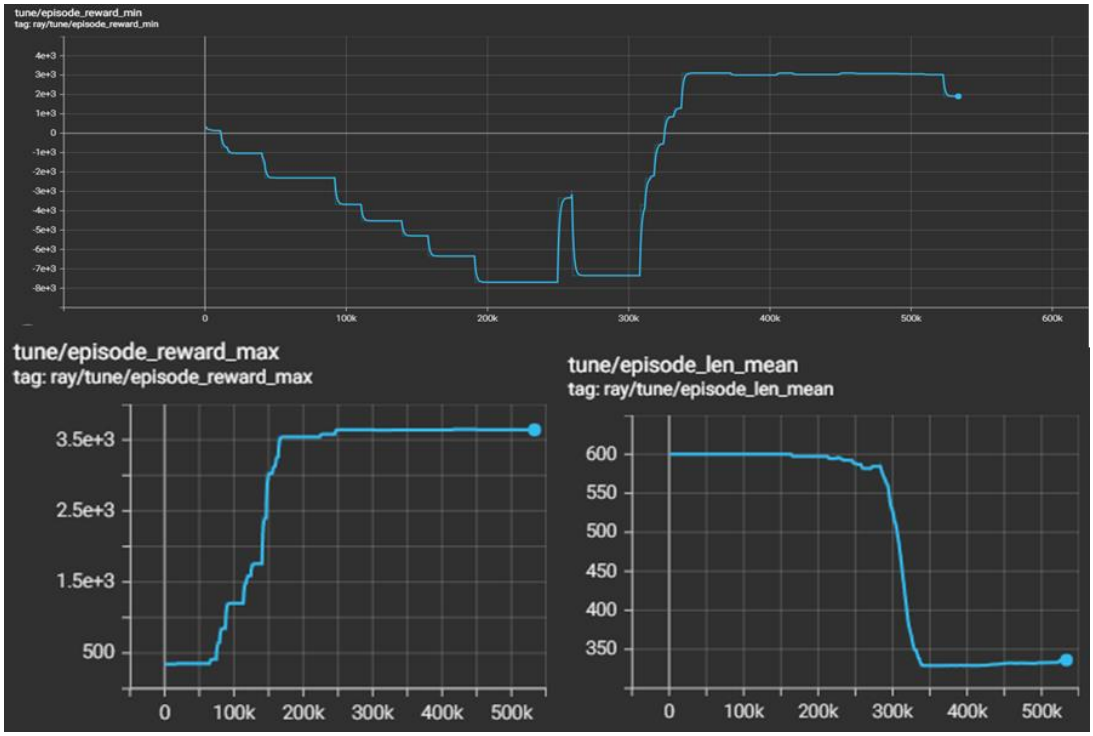


Figure 37 - Training to Drag Race with PPO in Ray

To achieve the objectives of this project, it was assumed that a mixture of images and variable parameters might need to be provided to the neural network model. Therefore, the model utilised earlier would not be sufficient. Unfortunately, Ray would later prove to be more challenging to in such case. The issues encountered were that while Ray provides a ComplexNet, a model that should be automatically selected to handle mixed observation types (images, one-hot encoded parameters, general variables) when applying PPO, APPO or IMPALA algorithms (Ray, 2023d). It could be confirmed to automatically work when trying IMPALA but not with PPO. It is possible that the issue was due to the Ray and RLlib versions tested (Ray 2.4 and Ray 2.6). The latest version at the time of this attempt was 2.7.0, which was not possible to update to due to other unsatisfiable dependencies. The current latest version as of writing is 2.9.0.

To overcome this issue a hack-workaround was done by copying the code of ComplexNet and introducing it as a custom model into my setup. Unsurprisingly, this did not work, since PPO is an on-policy algorithm, and when performing learning on the complex observation space, the trainer cannot keep up with the pace of the real time environment timing. This is not the fault of Ray's RLlib since all the default use-cases provided are intended for environments that can be stepped in a non-real-time fashion. Ray and RLlib provided a server-client mechanism to allow for a worker to run remotely and interact with an external environment that cannot be stepped. Furthermore, by caching the policy on the remote worker as shown in Figure 38, the rollout worker does not have to wait on the trainer to provide the next action. This methodology was attempted but without being able introduce the ComplexNet or SAC.

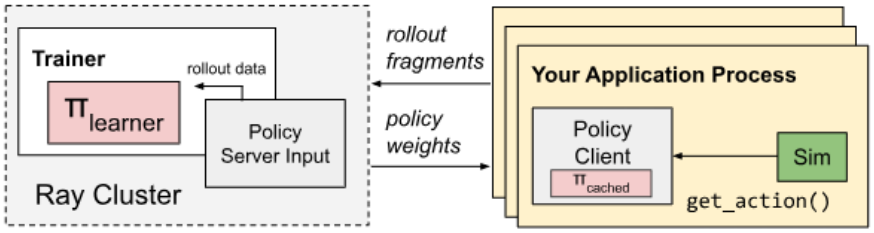


Figure 38 - Ray Remote External Environment (Ray, 2023c)

Considering the time constraint of this project, the decision was taken to start over utilising an adaptation of the TMRL methodology of implementing SAC. The rationale was that this methodology must be technically feasible since it is already applied to a similar real-time problem. The TMRL methodology is just that, and thus a reimplementation of that method by means of code the pipeline would be necessary. It does not provide any convenient pre-processing tools like Ray does (or states to do), and thus would also require manual variations to handle different observation setups. Hereafter the application of the TMRL methodology to GT will be referred to as the "TMRL pipeline". One of the authors of the TMRL work (Yann Bouteiller), was very responsive on their Github page, and was extremely helpful in assisting with issues faced in adapting their work.

The TMRL pipeline operated similar to Ray's server-client concept, allowing for multiple rollout workers running their own cached policy. When three rollout workers were deployed on three separate machines, the configuration would resemble the following figure. While emulating multiple PSX sessions on a single computer is feasible, it is also difficult to make sure each emulator session is operated by the correct virtual gamepad since all gamepads on a machine have the same hardware name identifier.

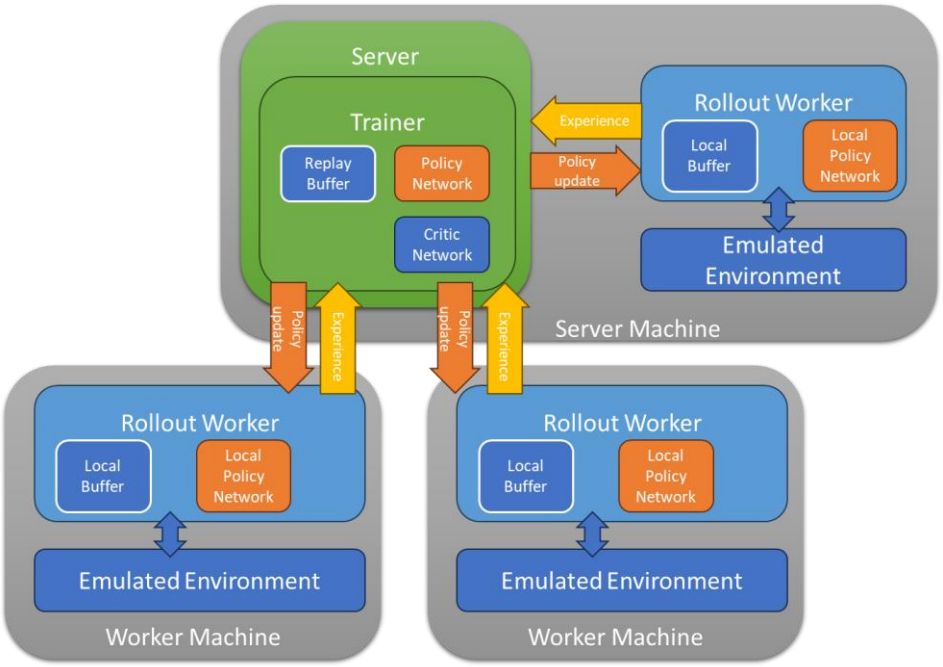


Figure 39 - TMRL Pipeline Setup

Chapter 4 Experimental Results

In this section the experimental environment and parameters are described before presenting the results. The actual process for this experimental phase was not entirely linear, since during the experimentation phases, observations lead to tuning of the experimental setups or development of new ones.

4.1 Experimental Setup

This study tried to cover several state and action space combinations, however, not all combinations were investigated in part due to unplanned modifications of the hardware and software setup

For most of the work (except the last 4 weeks of the experimentation phase) an unknown memory leak bug would cause workers to crash and training to eventually stop, occasionally even crashing the displayer driver of the machine. This occurred roughly every 5 hours. It was found to be a bug in the emulator version being used (which has since been fixed). Unfortunately, this incident severely restricted my ability to run a large matrix of experiments or to attempt tuning hyper parameters. Furthermore, certain elements of the results would not be objectively comparable due variations of background workload on the computers used.

A training session would often require roughly a minimum of 6 hours to 2 days to converge onto a good result.

Two forms of workloads were existing, that of a trainer, and that of a rollout-worker. The trainer is that which processes experiences samples and performs learning to hopefully derive an improved policy, whereas a rollout-worker makes use of a policy to determine the beset next action.

The hardware setup initially focused on a single desktop computer working as both the trainer and rollout-worker, with key specs - AMD 3700X (8x2 logical processors), 32 GB DDR4-3600, 4GB GTX1650 Super. By the end of this work, the setup was evolved over stages to the final configuration below:

Trainer (CUDA) & Rollout Worker (CPU): Desktop: AMD 3700X (8x2 logical processors), 64 GB DDR4-3600, 24GB RTX3090.
Rollout Worker (CPU): Laptop: Intel 12650H 32GB DDR5-4800, 8GB RTX4070.
Rollout Worker (CUDA): Desktop: Intel Xeon E5-2695, 32GB-2133, 4GB GTX1650 Super.

The Intel Xeon machine was too slow to compute the required action from the policy via a forward-pass when relying on the CPU without breaking the target real-time elastic limit. For this reason, the GPU was used instead for forward-passes on that machine.

4.1.1 Control Modes

Control modes are the various methods by which the action spaces were defined. Some of the setups were established after analysing early test results.

Table 4 - Control Modes

Control Mode	**Accel**	**Brake**	**Accel & Brake Exclusive**	**Steering**
Control 0	Discrete	Discrete	No	Discrete
Control 1	Discrete	Discrete	Yes	Discrete
Control 1.5	Continuous	Continuous	Yes	Continuous
Control 1.6	Continuous	Not Applicable	No	Continuous
Control 2	Discrete	Not Applicable	No	Discrete
Control 2.5	Discrete	Not Applicable	No	Discrete

Table 4 lists the control methods. Most of the methods have been evaluated with the SAC setup, whereas only control mode 1 was used for PPO. PPO is said to be able to work on environments with discrete and continuous action spaces, while the used SAC implementation is said to be only capable of working with continuous action spaces (Spinning Up, 2023). A modified version of SAC specifically for discrete actions is existing in literature (Christodoulou, 2019), however it will be shown as a result of this work that it was possible to achieve similar results with either discrete or continuous actions in this work.

Where continuous actions are used, the settings in Gran Turismo would be such that the left analogue stick is for steering (-1 is full left and +1 is full right) and the right analogue stick is for accelerating (+1 is full acceleration) and braking (-1 is full braking). It is impossible to accelerate and brake with the analogue stick in GT.

The assumption made to develop these modes was that an agent that can only accelerate (and not brake) like in control 1.6 or greater, would have the best chance of learning to get some sort of reward on the straight start-finish part of the track. That an agent that can apply the brakes and accelerate simultaneously like in control 0, would take longer to learn that under most circumstances it is not beneficial to use both. That an agent with analogue control such as control 1.5 and 1.6 would have superior accuracy and perform better than the result of a discrete ones such as control 1, 2 and 2.5.

Control 0

This action space has 3 dimensions, 1 for accelerating, 1 for braking and 1 for steering each with a value from -1 to 1 inclusive. The outputs of these dimensions were collapsed to discrete actions as follows:

If dim1 > 0 then accelerate else release accelerator
If dim2 > 0 then brake else release brake
If dim3 > 0 then turn right, else if dim3 < 0 then turn left, else go straight

Control 1

This action space has 2 dimensions, 1 for accelerating or braking and 1 for steering each with a value from -1 to 1 inclusive. The outputs of these dimensions were collapsed to discrete actions as follows:

If dim1 > 0 then accelerate else release accelerator else if dim1 < 0 then brake else release brake
If dim2 > 0 then turn right else if dim2 < 0 then turn left

Control 1.5

This action space has 2 dimensions, 1 for accelerating or braking and 1 for steering each with a value from -1 to 1 inclusive. The outputs of these dimensions were mapped as follows:

Right analogue stick = dim1
Left analogue stick = dim2

Control 1.6

This mode is similar to 1.5 except that dim1 is scaled and shifting to eliminate braking. This action space has 2 dimensions, 1 for accelerating and 1 for steering each with a value from -1 to 1 inclusive. The outputs of these dimensions were mapped as follows:

Right analogue stick = max(dim1,0) *clipping off negative values*
Left analogue stick = dim2

Control 2

This action space has 2 dimensions, 1 for accelerating or braking and 1 for steering each with a value from -1 to 1 inclusive. The outputs of these dimensions were collapsed to discrete actions as follows:

If dim1 > 0 then accelerate else release accelerator else release accelerator
If dim2 > 0 then turn right else if dim2 < 0 then turn left

Control 2.5

This action space has 2 dimensions, 1 for accelerating or braking and 1 for steering each with a value from -1 to 1 inclusive. The outputs of these dimensions were collapsed to discrete actions as follows:

If dim1 > 0 then accelerate else release accelerator else release accelerator
If dim2 > 0.2 then turn right else if dim2 < -0.2 then turn left *dead zone between -0.2 and 0.2*

4.1.2 Observation Modes

Table 5 is a description of all potential parameters. Some categorical parameters (indicated by an *) were treated as continuous in the SAC setup due to time constraints to modify the code.

Table 5 - Available Parameters

Parameter	**Description**	**Data Type**	**Values**
*rState**	Describes if the race is in the "3-2-1-GO" stage, racing, or race finished.	Unsigned 8 bit integer	[0,5]
*eClutch**	Describes if the clutch is engaged at idle, release, or engaged for shifting gears.	Unsigned 8 bit integer	[0,3]
eSpeed	Engine speed in revolutions per minute	32 bit signed integer	[0,10000]
eBoost	Engine turbo-charger boost (if using a turbo-charged vehicle) in a non-physical dimension	32 bit signed integer	[0,10000]
*eGear**	Vehicle drive gear	Unsigned 8 bit integer	[0,6]
vSpeed	Vehicle speed	32 bit signed integer	[0,500]
vSteer	Steering angle of wheels	32 bit signed integer	[-1024, 1024]
*vDir**	A flag to indicate if facing the wrong direction	Unsigned 8 bit integer	[0,1]
*vColl**	Indicate which corner or face of the vehicle is in contact with a wall	Unsigned 8 bit integer	[0,32]
*fLcoll**	Flag to indicate front left contact with a wall	Unsigned 8 bit integer	[0,1]
*fRcoll**	Flag to indicate front right contact with a wall	Unsigned 8 bit integer	[0,1]
*rLcoll**	Flag to indicate rear left contact with a wall	Unsigned 8 bit integer	[0,1]
*rRcoll**	Flag to indicate rear right contact with a wall	Unsigned 8 bit integer	[0,1]
fLeftSlip	The slip ratio of the front left tyre	Unsigned 8 bit integer	[0,255]
fRighttSlip	The slip ratio of the front right tyre	Unsigned 8 bit integer	[0,255]
rLeftSlip	The slip ratio of the rear left tyre	Unsigned 8 bit integer	[0,255]
rRightSlip	The slip ratio of the rear right tyre	Unsigned 8 bit integer	[0,255]
*fLWheel**	Road, Kerb, Grass, Dirt contact on front left wheel	Unsigned 8 bit integer	[0,4]
*fRWheel**	Road, Kerb, Grass, Dirt contact on right left wheel	Unsigned 8 bit integer	[0,4]
*rLWheel**	Road, Kerb, Grass, Dirt contact on rear left wheel	Unsigned 8 bit integer	[0,4]
*rRWheel**	Road, Kerb, Grass, Dirt contact on rear right wheel	Unsigned 8 bit integer	[0,4]
Images	A stack of the last 4 display frames in greyscale (including the current frame)	64 x 64, Unsigned 8 bit integer array	[0,255]

The parameters were then included or excluded in different combinations to form different observation spaces for testing. The modes are shown in Table 6. Parameters with a '✓' mark indicated that they were part of a particular observation mode.

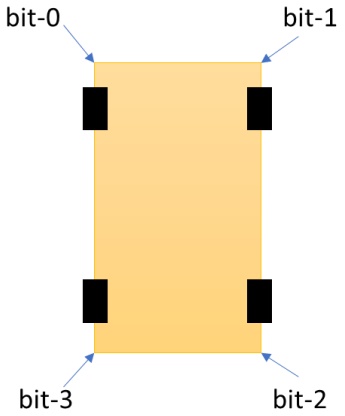
In addition to all the observable parameters, the previous two actions were appended to each observation.

Table 6 - Observation Modes

Parameter	**Mode 0**	**Mode 1**	**Mode 2**	**Mode 3**
*rState**		✓	✓	✓
*eClutch**		✓	✓	✓
eSpeed		✓	✓	✓
eBoost		✓	✓	✓
*eGear**		✓	✓	✓
vSpeed		✓	✓	✓
vSteer		✓	✓	✓
*vDir**		✓	✓	✓
*vColl**		✓	✓	
*fLcoll**				✓
*fRcoll**				✓
*rLcoll**				✓
*rRcoll**				✓
fLeftSlip			✓	✓
fRighttSlip			✓	✓
rLeftSlip			✓	✓
rRightSlip			✓	✓
*fLWheel**			✓	✓
*fRWheel**			✓	✓
*rLWheel**			✓	✓
*rRWheel**			✓	✓
Images	✓	✓	✓	✓

The assumptions made in developing the above modes were such that mode 0 would represent the minimalistic setup that is like prior work on Atari games. However, in this case it would be uncertain if the non-pixel perfect images of the PSX, the elastic delays due to real-time learning, and the lack of sound would collectively make it too challenging for an agent to learn.

Mode 2 and mode 3 are variations of each other and contain the richest observation. The only distinction is how collisions with the wall are observed. In mode 2, a single categorical variable *vColl* represents a 4-bit mask, where each bit represents contact between a corner of the vehicle and the wall as shown in the following figure. Only 2 adjacent bits can be true, in the case of a face collision, which means that not all 4-bit values are possible, and that magnitude of the resulting integer does not represent any meaning.



Bit-3	Bit-2	Bit-1	Bit-0	Value	Collision
0	0	0	0	0	No collision
0	0	0	1	1	Front Left
0	0	1	0	2	Front right
0	0	1	1	3	Front face
0	1	0	0	4	Rear right
0	1	1	0	6	Right face
1	0	0	0	8	Rear left
1	0	0	1	9	Left face
1	1	0	0	12	Rear face

Figure 40 - Collision Bit Mask

Since the SAC algorithm implemented, does not encode this parameter as a categorical one, it was assumed that the agent might try to associate meaning to the proportionality of this parameter which would not have meaning. In case of the PPO implementation with categorical parameters, mode 2 is valuable. Mode 3 was then developed to re-split all for corners as individual parameters so that the agent would potentially learn any relationships with contacts on any 1 or 2 corners. Thus, it was also assumed that mode 3 would perform better than model 2.

Mode 1 is somewhere in between mode 0 and mode 2/3 and is the closest to a human experience, because it is assumed that a human player does not know the slip ratio of a wheel as accurately as an in-game parameter other than a sense of sliding from the sound and the pixelated smoke that appears and was thus excluded. From personal experience it is challenging to perform well in GT without sound to hear the engine speed and tyre slip. This mode would help validate if knowledge of tyre slip is critical for training an AI agent that does not have the ability to hear tyre slip. Similarly, the tyre contact information was also eliminated because it is believed that this information should be observable by the reduction in vehicle speed or the oscillation of the vehicle body as it traverses non-road surfaces. The parameter vColl was retained in this setup to focus the changing points on the tyre slip.

4.1.3 Vehicles

Two vehicles were used, a Toyota MR2 GT-S which has a turbo-charged engine producing around 240hp and a heavily modified Toyota Supra with around 900hp.

Using data recording (by means of a script to log the relevant parameters) of both vehicles launching off from the start after revving the engine during the pre-race-start-countdown and of an acceleration from standstill on a random part of the track after the race has started, it is possible to highlight the key differences in the vehicles. Figure 41 shows these differences.

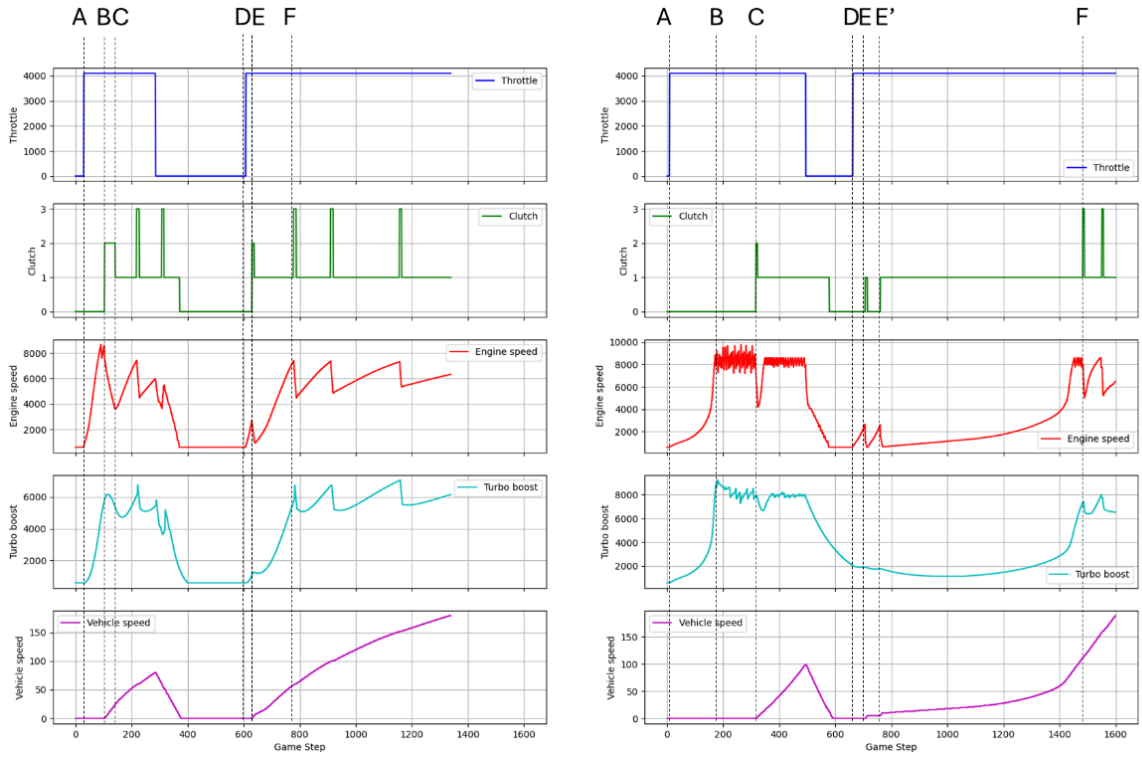


Figure 41 - MR2 and Supra Comparison (Top-to-Bottom: Throttle, Clutch, Engine Speed, Turbo Boost, Vehicle Speed)

Pre-race-start-countdown

From time-step A the accelerator is held, causing the engine to reach the maximum engine speed and boost, this is during the phase that the clutch is held at state 0 (which means the wheels are not connected to the engine – a special circumstance only applicable during the pre-race-start-countdown. The MR2 takes less than 100 time-steps, whereas the Supra requires nearly 200 time-steps.

At time-step B, the pre-race-start-countdown ends, and the race is started, and the clutch jumps to state 2 which to my best estimate is a clutch slip simulation, since vehicle such as the Supra exhibit a shorter period with the clutch at state 2.

From time-step C, the clutch turns to state 1, representing a normal torque transfer from the engine to the wheels.

Standstill-Start

Time-step D represents applying the accelerator to a vehicle that was stationary. Time-step E, represents when the clutch engaged (state 1) for a launch. In the case of the MR2, the clutch remained engaged until time-step F, which is the first gear change from 1st gear to 2nd gear (represented by a clutch state 3). In case of the Supra, due to the insufficient engine speed and/or boost, the Supra stops

decelerating at time-step E, and the vehicle returns to a clutch state 0, and attempts a second launch at time-step E'. It is only several time steps later that the Supra manages to reach 2nd gear.

Analysing the data from both vehicles and both starts (pre-race-start-countdown and standstill-start), it can be observed that, the MR2, has a more direct response to accelerator actions at any condition, whereas the Supra struggles to move unless the engine speed and boost were sufficiently high when the clutch is engaged (state 1 or 2). Once sufficient engine speed or boost is available the rate of change of speed of the Supra is nearly triple that of the MR2, but nearly half MR2's rate of change of speed at engine speed or boost.

The differences were expected to play a role in driving situations, however, were shown to prove additional difficulties with starting the race. Which lead to some modifications to the reward function or the timing at which the AI agent takes over interaction with the environment.

4.1.4 Reward Functions

Various reward functions were developed and tweaked during this work; however they can be grouped under two main types. The first type refers to a *naive speed* reward functions – a function that rewards proportionally to the driving speed of the vehicle. The second type refers to a function that rewards based on how far forward along the track (hereafter referred to as the *checkpoint reward)* the vehicle has moved since a previous observation. This latter type requires the use of the pseudo-centreline described in section 3.6.

The naïve speed reward function was predominantly utilised with the RLlib framework on simpler problem. It was assumed that if the whole pipeline and code was working, this reward would generate some meaningful learning, so long as the agent could observe the vehicle speed parametrically and corelate this results with the accelerator action (at least in a straight line).

The checkpoint reward method was the base method as this method was most supported in literature by similar problems, including the GT Sport related work and the related TMRL works. The variations of this reward function can be summarised by the following equation.

$$R_{Total} = R_{Checkpoint} + R_{Direction} + R_{Contact} + R_{speed} + K_{Constant} \quad \textit{Equation 4}$$

With each of the summed term are functions defined as:

$$R_{checkpoint} = K_{factor}\left(checkpoint_{current} - checkpoint_{previous}\right) \quad \textit{Equation 5}$$

$$R_{Direction} = \begin{cases} 0 & \text{When facing forward} \\ K_{Direction} & \text{When facing backwards} \end{cases} \quad \textit{Equation 6}$$

$$R_{Contact} = \begin{cases} 0 & \text{When moving without contact} \\ K_{Contact} & \text{When contacting the wall} \end{cases} \quad \textit{Equation 7}$$

$$R_{Speed} = \begin{cases} 0 & \text{When moving above a certain speed} \\ K_{Speed} & \text{When moving slower} \end{cases} \quad \textit{Equation 8}$$

The $R_{checkpoint}$ results are a reward proportional to the number of checkpoints passed since the previous observation, when travelling backwards, this reward is negative The K_{factor} is a simple scaling factor which was varied during different training sessions. This was used mainly to try and maintain the maximum reward between observations to less than approximately 1.0.

$R_{Direction}$ is a function designed to make facing backwards or moving backwards more disproportionately penalising than moving forward. This was not applied in all training sessions, and when applied the constant $K_{Direction}$ was varied amongst different training sessions.

$R_{Contact}$ is a function that applies a penalty when a vehicle contacts the wall. This was not applied in all training sessions, and when applied the constant $K_{Contact}$ was varied amongst different training sessions.

R_{Speed} is a function that applies a penalty when a vehicle is driving too slowly. This was not applied in all training sessions, and when applied the constant K_{Speed} was varied amongst different training sessions.

$R_{PreStart}$ is a function that was developed later as an alternative to wait for the agent to learn how to behave during the pre-race-start-countdown. The pre-race-start-countdown in GT has a different mechanism from the rest of the race. During this phase of the game, the clutch remains disconnected and allows a player to use the accelerator to raise the engine speed and turbo-boost (if applicable to the vehicle). It is possible to change the steering angle but without any meaningful effect since the vehicle is station. The consequence of starting the race with the incorrect steering angle and low engine speed or boost are only observable once the race started. That is to say, the potential for recognising the higher or lower reward is at a single time-step.

It was observed in early training attempts, when the AI agent was operating the Supra, due to the very large turbo lag (as described in 4.1.3), it was rarer to observe the rewarding time-step since not only did the agent have to learn to associate the accelerator with a rewarding action, but it was necessary to recognise that a history of holding the accelerator was beneficial. As an alternative to this issue, an additional artificial reward $R_{PreStart}$ which would encourage the agent to hold a high engine speed, a high turbo-boost and a small steering angle. It was a fair approach akin to the human scenario of instructing a human to hold the accelerator pedal down and not to touch the steering wheel until the race starts.

$$R_{PreStart} = \frac{EngineSpeed}{20000} + \frac{TurboBoost}{20000} - \left|\frac{SteeringAngle}{2048}\right| \qquad \textit{Equation 9}$$

In later phases of this work, a third approach was developed that eliminated the pre-race-start-countdown which is described in section 4.1.5.

4.1.5 Episode Start and Duration

The start of the episode was initially set to the beginning of the pre-race-start-countdown (which is around 8 seconds long). However, this meant that the worker would accumulate 160 observations (at 50ms intervals) at a part of the game where reward is impossible to experience except when applying equation 9 from section 4.1.4.

The first optimisation attempt was to start the episode 2 seconds before the race-start. This would allow sufficient time for the MR2 to raise its engine speed and boost before the start once the agent had learnt to do so. However, this time parameter would not work for the Supra due it requires more time to ramp up the engine speed and boost (see section 4.1.3).

The second optimisation attempt was to start the episodes with the vehicle already in motion. This method seemed to produce a capable agent with fewer training steps, however caused temporary policy collapsed when later exposed to pre-race-starts.

The third optimisation attempt that was retained for this work was to eliminate the pre-race-start-countdown by giving the vehicle controls to the game's built-in AI and handing over controls to the neural network AI at the instance of the race start. This method was justified since the race start is a trivial challenge which the neural network was able to learn given the time and is not particularly challenging even for an inexperienced human player (one must only press the accelerator and wait).

The duration of episodes presented different issues. The TMRL pipeline worked in manner such that episode experience was transmitted back to the trainer on the server each time an episode terminated (by completing the lap) or truncation (terminating after a fixed time duration).

Without truncation, there is an almost certain possibility that the agent would never terminate based on the initial random policy, and thus no experience could be collected for training. Therefore, initially training was conducted with periodically expanding the permitted episode length as shown in Figure 42 (x-axis represents thousands of training steps, and y-axis represents number of observation game-steps). In this manner, the episode length was truncated to 1000, 1500, 2000 and finally 3000 steps. The rationale is that the by limiting the episode length the agent can learn how to maximise the reward for the available observable episode length.

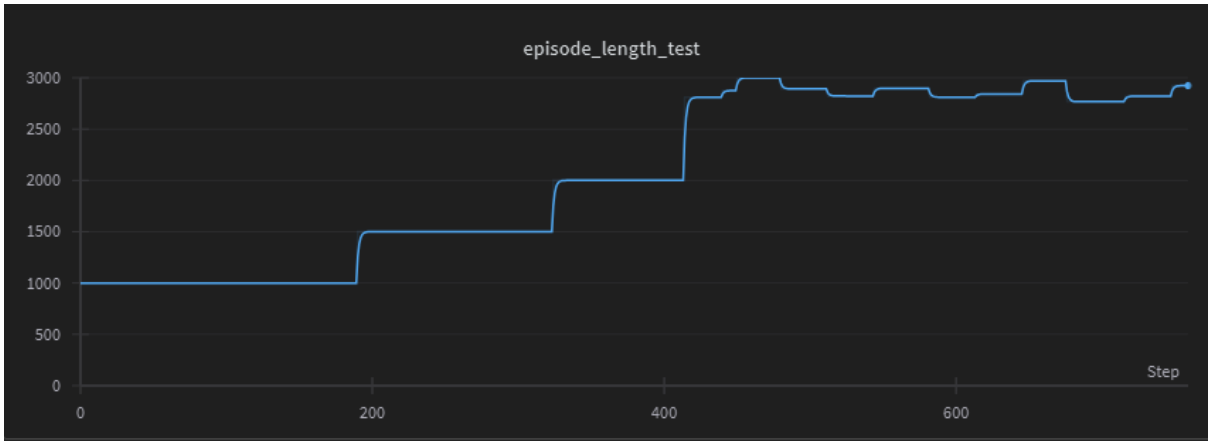


Figure 42 - Episode with Fixed Duration

Once the agent can achieve a stable reward, it would then be exposed to more of the track. In the latter section of training, the episodes are truncated by the successful completion of the race.

Minimising the episode length at a maximum reward is then the target of training as shown in Figure 43 (note the period of negative reward was caused by computer-crash) corresponding to the episode length shown in Figure 42.

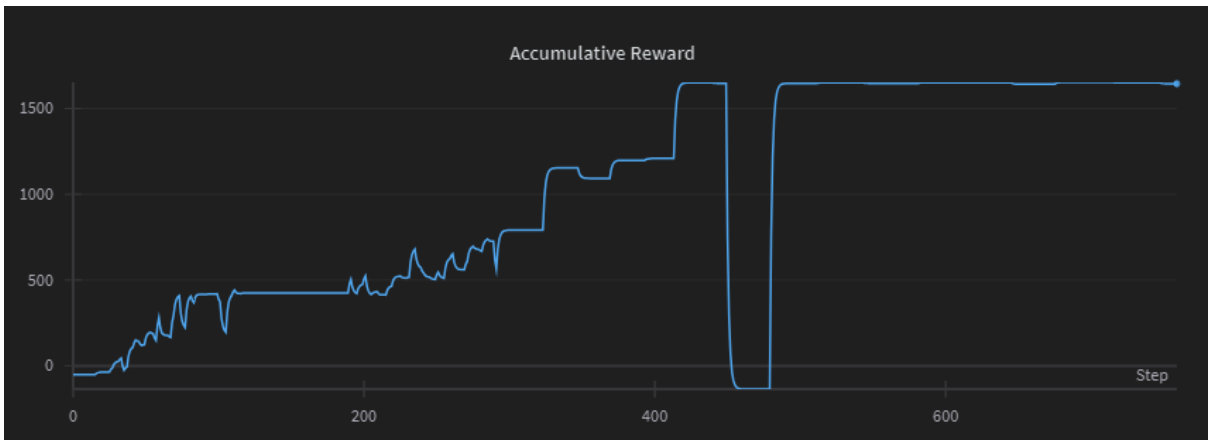


Figure 43 - Reward during Training

The demerit of this method of truncating episodes is that it requires detection of a saturated reward and allows the agent to drive the vehicle in the wrong direction. The agent is able to recognise it is facing the wrong direction in Modes 1 to 3 (which include the direction flag), and thus it learns to stop on track to avoid further negative reward. Therefore, this method is wasteful in that non-beneficial experience can be accumulated until the next episode is started.

To alleviate these issues a final method of limiting episode length was developed. This involves automatically truncating the episode if the vehicle has faced the wrong direction for 20 game-observation steps or has been at a velocity below 15 km/h (an arbitrarily low speed) for more than 40 game observation steps. This method forces the worker to accumulate useful experience and update the policy more frequently during initial training when the agent is still learning how to accelerate off the starting line.

4.1.6 Track

The track used for the significant majority of the training was the High Speed Ring. The layout is shown in the following figure. For some initial training with the naïve speed reward function the straight drag-race track was used.

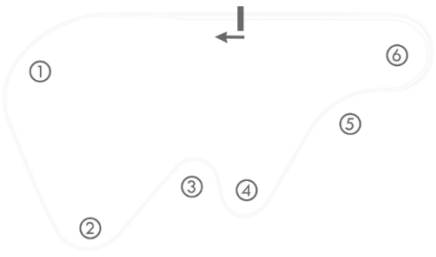


Figure 44 - High Speed Ring Track Layout (Gran-Turismo Fandom, 2023)

The high speed ring is mostly made of right turn banked, with walls on the other side of the corners. The banking allows for higher speeds especially through the first wide corner, but also causes asymmetric behaviour with left and right steering, with a tendence to oversteer when turning left and understeer when turning right. All other corners are tighter and are impossible to drive through

without braking or tapping the wall in a controlled manner (personal preference is to tap the wall by the rear outside corner of the vehicle to not spin-out. This method is similarly fast to Submaniac's lap-time which is used as a reference expert human player). Corner three is the first and only significant banked right-turn that directly feeds into another banked left-turn. The tightness of corner 3 and 4 and their high banking make it easier than other corners to fall into a spin when hitting the wall at speed. Corner three is the location at which the agent would have to first hopefully learn how to turn right. Furthermore, it would be necessary to do so in a manner to not compromise the fourth corner which is the final part of the only S-bend of the track. It was already observed in TMRL literature, that without forward forecast of the track, S-bend corners can be tricky for agents to learn, and therefore this section of the track posed a genuine concern for the success of this work.

The straight section between 4 and 5 is the only section that is driven through a tunnel, which means the lighting of the track is completely different as demonstrated by the following image. The tunnel section is also the only straight section with grass on either of the track.

Figure 45 - Tunnel Section Compared to Banked Corner

Having managed to complete the mild right corner 5 and the final banked corner 6, another challenge at the start of the start-finish straight is the nearly perpendicular fence to the pit entrance as shown in the following figure. Hitting this blocked-off pint entry will almost always stop the vehicle to nearly 0 km/h or initiate a spin.

Figure 46 - Closed Pit Entry

A second lap on the high-speed ring is almost identical to the first except that the entry speed to the first banked corner can be higher.

4.1.7 SAC Flavour

The example SAC implementation in the TMRL pipeline work was done exclusively with SAC. The implementation is based on code provided by Spinning Up. Specifically, this is the version that does not utilise a value function in addition to the two Q-function (Spinning Up, 2023) for the critic branch which is based on later publications on SAC (Haarnoja et al., 2019).

Furthermore, both the TMRL authors and Spinning Up provide two variants of SAC, one with a fixed entropy regularisation and another with an entropy constraint, for this work only the traditional fixed entropy was experimented with due to time constraints. The constrained method is said to be more effective to encourage exploration where uncertainty is still high and stabilise regions of the policy where the preferred actions have been found (Haarnoja et al., 2019).

The model of the TMRL-pipeline passes images through four convolution layers, which are then flattened and concatenated to the other observed parameters. This concatenated layer is then passed through 2 hidden layers of 256 nodes. It was surprising to find most authors of such as those of TMRL (Bouteiller, Geze and GobeX, 2023b), GT Sport (Fuchs et al., 2021) and the ComplexNet (Ray, 2023a) all applied a default 2 hidden layers of 256 nodes, and therefore the same architecture was also adopted.

4.2 Results and Analysis

This section will focus exclusively on the work done using the TMRL pipeline. The results of all successful and failed training sessions have been logged with *Weights & Biases*. This work adopted the hyper-parameters utilised in the TMRL work as the following in conjunction with a custom reward function that is similar to that of the TMRL and related works in literature. These are listed in the following table.

Table 7 - Default SAC Parameters Used

Parameter	Value	Comment
Discount Factor	0.995	Future reward consideration
Alpha	0.01	Encourage exploration
Learning rate for Critic	5E-5	
Learning rate for Actor	1E-5	
Polyak	0.995	Exponential averaging of the two target networks
Timestep	0.05s	Real-time environment target time step
Elastic Limit (Factor)	1.0	Factors of time-step for the real-time limit
Image size	64x64	Greyscale
Image stack	4	Each observation contains the current display output together with the 3 previous observation-step's respective display output
Action history	2	Each observation contains the 2 previous actions as part of the observation
Reward Function	$R_{Total} = R_{Checkpoint} + R_{Direction} + R_{speed}$ where: $R_{checkpoint} = 0.05\left(checkpoint_{current} - checkpoint_{previous}\right)$ $R_{Direction} = \begin{cases} 0 & \text{When facing forward} \\ -0.02 & \text{When facing backwards} \end{cases}$ $R_{Speed} = \begin{cases} 0 & \text{When moving above or equal to 15km/h} \\ -0.02 & \text{When moving slower than 15km/h} \end{cases}$ *Note: In the following plots, mention of Reward 4.58 or R4.58 or 4.58 in the legend refers to this configuration of the reward function.*	

following sections of this report are based on post-analyses of the best training results. This involves re-loading the weights and allowing the agent to drive the track for eight episodes and logging the agent's lap-times, vehicle parameters and driving position along the track, from which some objective and subjective analysis is done on the quality of the driving.

Finally, a comparative analysis is done by looking at the training logs across some of the training sessions will be presented in the final section of this chapter.

4.2.1 Benchmark Performance with the MR2

When driving the MR2, the agent's performance was objectively compared to three performances – the game's built in AI and that of Submaniac (a well known GT player hereafter referred to as *the expert*). Hereafter the RL trained agent is generically referred to as SOFIA (Switft Overtaking Futuristic Intelligent Automaton).

Table 8 - Performance Benchmark

Player	**First Lap (min:sec:ms)**	**Second Lap (min:sec:ms)**	**Total Race Time (min:sec:ms)**
Default AI	1:07:343	1:15:343	2:23:046
Submaniac (analogue controls)	1:11:076	1:02:695	2:13:771

Lap times within 5 seconds of the default AI would be considered as ***acceptable***. Lap times in between the two references would be considered as ***good*** and lap times better than that of the expert would be considered ***great*** (which may be an ambitious goal since claims that they have over two decades of practice). Agents that cannot complete a lap consistently are obviously considered bad.

The following section will present some performances from each category of ***good***, ***acceptable*** and ***bad***. Unfortunately, no result achieved the ***great*** level.

4.2.2 Good Results

The best results of the various training sessions of SOFIA were consistently above the default AI's performance, and occasionally very close to that of the expert.

The configurations that produced an agent with *good* performance relied on mode 2 or mode 3 with any control settings except control 0, are highlighted in the following section as examples.

4.2.2.1 Mode 3 and Control 1.6

An example of the SOFIA's performance on one of the most recent training sessions using mode 3, with Control Mode 1.6 is given in the following Figure 47.

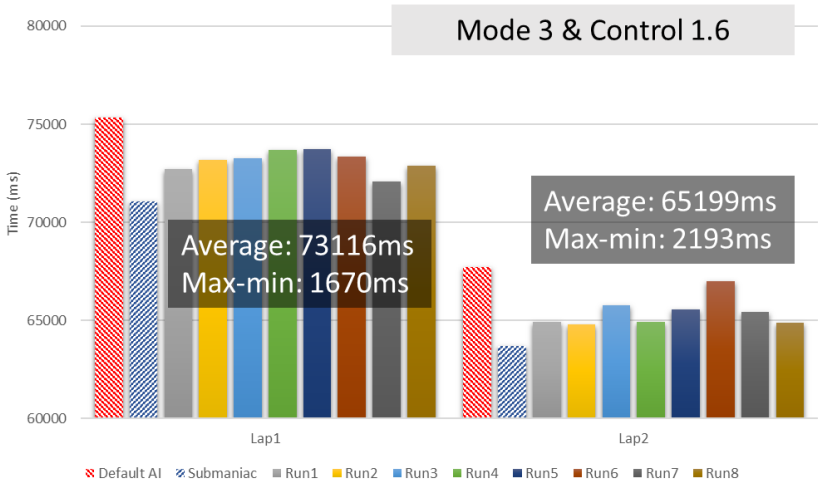


Figure 47 - SOFIA Mode 3 and Control 1.6

It is remarkable that SOFIA is always able to beat the built-in AI's performance, while having a performance that is in between that of the built-in AI's and that of the expert.

Figure 48 provides a comparison between SOFIA and the expert's vehicle speed and throttle control during the race.

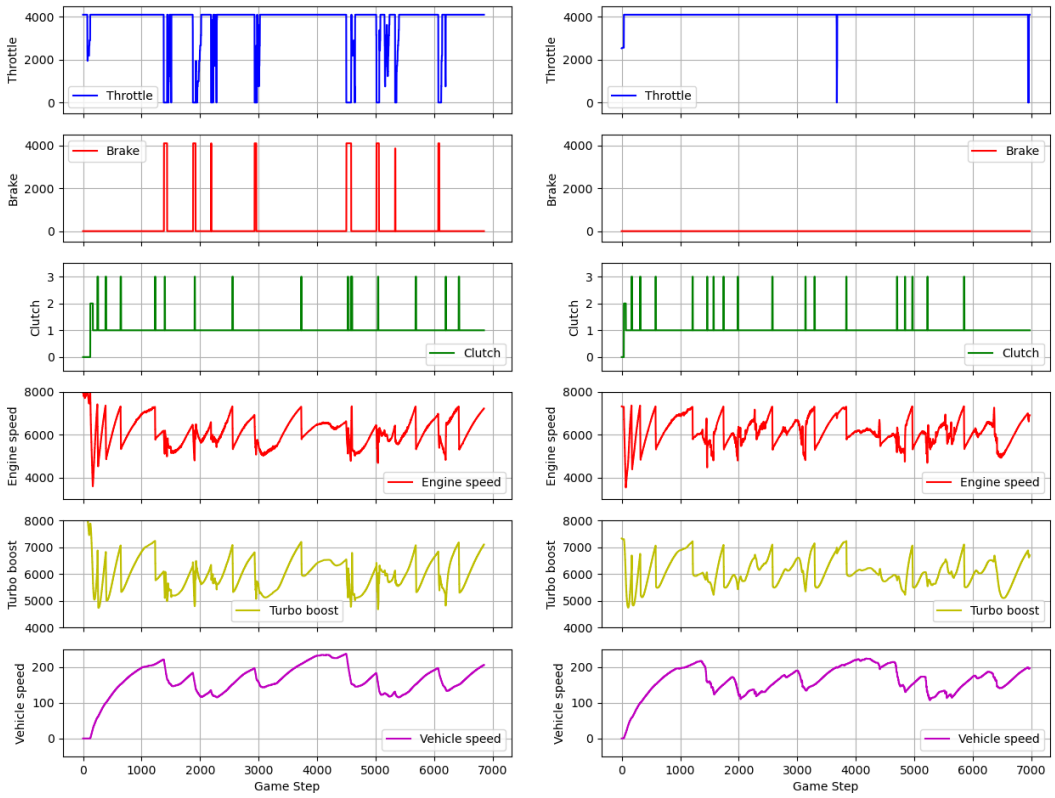


Figure 48 - Submaniac (left) against SOFIA [Mode 3 & Control 1.6] Driving Style (right)

The speed carried out through the race is similar, however the expert's lack of contact with the walls seems to lead to less noise speed profiles. the expert is also applying the brake (not shown in the data) into certain corners, which is evident from the sudden rapid decrease of vehicle speed at certain areas of the track. In this configuration SOFIA does not have the ability to brake, but it will be shown in later results, that even when SOFIA is provided the ability such as in control mode 1.5, SOFIA will still converge to the same driving style as of control 1.6. Furthermore, the expert also makes use of partial throttle in some sections of the race, whereas SOFIA saturates the throttle control to that of discrete action space and performs the race continuously on full throttle. SOFIA only lifts the throttle once on the start-finish line. This behaviour seems to be learnt and unlearnt continuously during training. The reason for this was not understood, however two hypotheses are formulated.

The first hypothesis is that the blue and white checked marking on the start finish line (see Figure 49) resembles other features such as the side barriers on the same track section. This could perhaps be activating parts of the CNN and hidden layers which had learnt to lift the throttle when facing a wall close to perpendicular to the wall.

Figure 49 - Start Finish Line

Figure 50 demonstrates the vehicle speed and the driving line of the expert and SOFIA. It can be observed that while the average and top speeds are similar, SOFIA is less straight when driving on the main straight. This is likely because it is hard to establish a more reward line and the because the entropy term of SAC is encouraging this dither.

In the first corner, both agents take a similar smooth driving line carrying similar speeds, however the entries to corners, seem to be more reactive in the case of SOFIA. This is most noticeable when comparing the driving line for corner two. SOFIA steers right (against the corner) before steering left (into the corner). This may be due to the fact that SOFIA has recognised that this manoeuvre of shifting the vehicle weight to the opposite direction before turning in can help the vehicle slide into the corner better, or this behaviour may be since SOFIA has limited foresight.

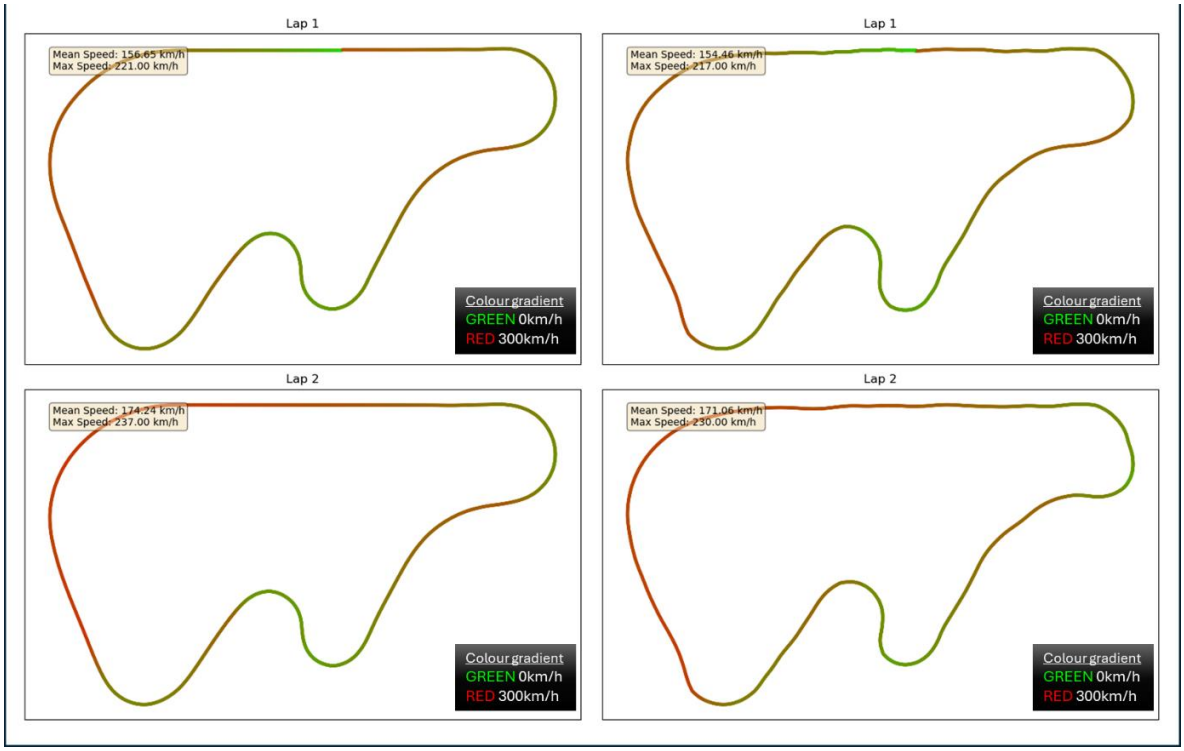


Figure 50 - Driving Line Comparison (Left: The Expert; Right: SOFIA [Mode 3 & Control 1.6])

The fact that the first corner is the smoothest, may also in part be since the first corner is the most experienced corner during training.

4.2.2.2 Mode 3 and Control 1.5

This version of SOFIA performs like the previous version, but able to occasionally perform even closer to the expert time for the first lap. Figure 51 demonstrates the lap times of this version of SOFIA.

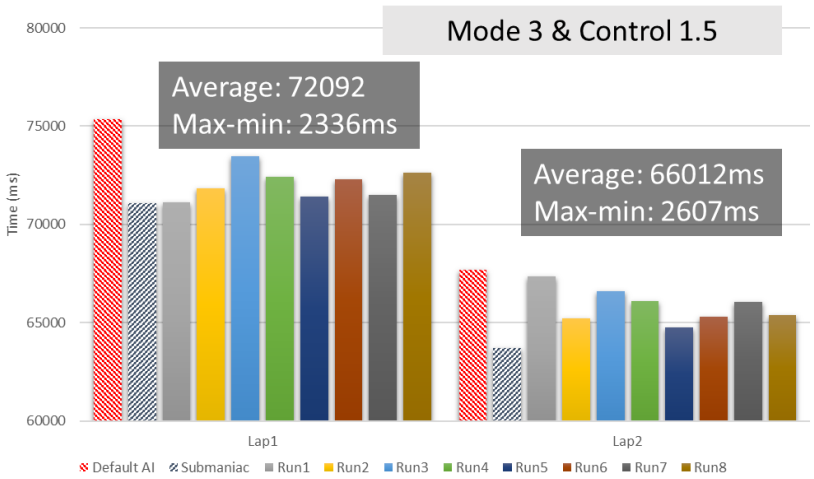


Figure 51 - SOFIA Mode 3 and Control 1.5

This agent can brake, but never uses that ability except for one blip a few seconds before entering the first right bend of the S-curve.

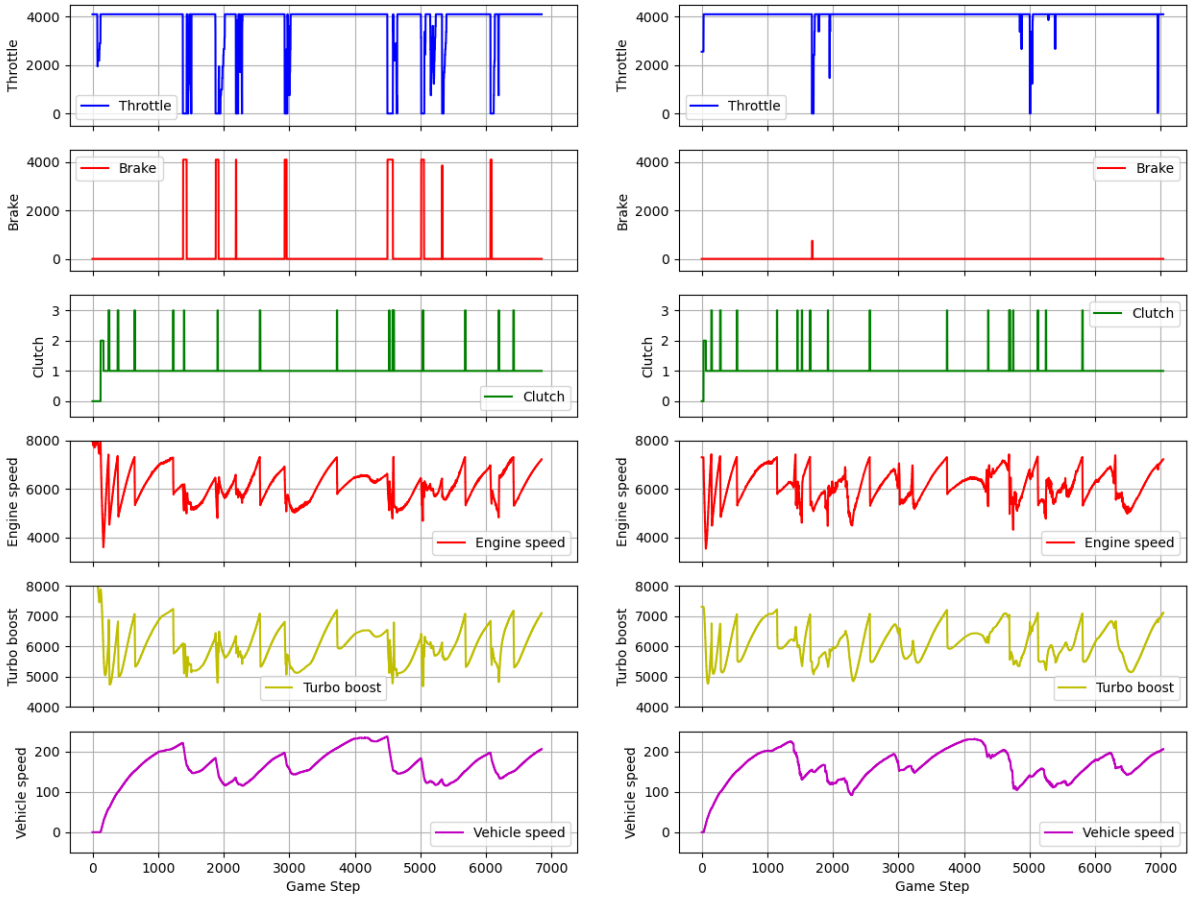


Figure 52 – The expert (left) against SOFIA [Mode 3 & Control 1.5] Driving Style (right)

The data visualised in Figure 52, shows that this SOFIA exhibits moments of throttle lift or partial throttle. However, there is insufficient data to assert that this is the result of the different control mode. One hypothesis is that this agent was not trained as sufficiently as much as the previous version and thus had not yet learnt to completely avoid lifting the throttle. Some evidence to this effect can be seen when comparing the range of lap-time differences on the first lap. SOFIA from 4.2.2.1 exhibited a smaller variation in lap-times when compared to this SOFIA. From experience of observing various agents being trained, the longer the training to more reliable they are at being consistent. Nevertheless, the sample size of episodes considered for both agents are statistically unreliable to assert this hypothesis. Due to time constraints, it was not feasible to re-train this agent to monitor its improvement.

Another hypothesis is that random variation of the initial policy allowed it to converge to a slightly better. This would require training several agents under the same conditions and observing their differences. A famous Youtuber that has described the results of similar work on Track Mania, suggests that they had to train several agents under specific parameters before pruning the agents and selecting the best one to carry forward into further training sessions (Yosh, 2023).

4.2.2.3 Mode 2 and Control 1

This version of SOFIA utilises mode 2 and control 1. performance of this agent is reflected by the lap times in Figure 53.

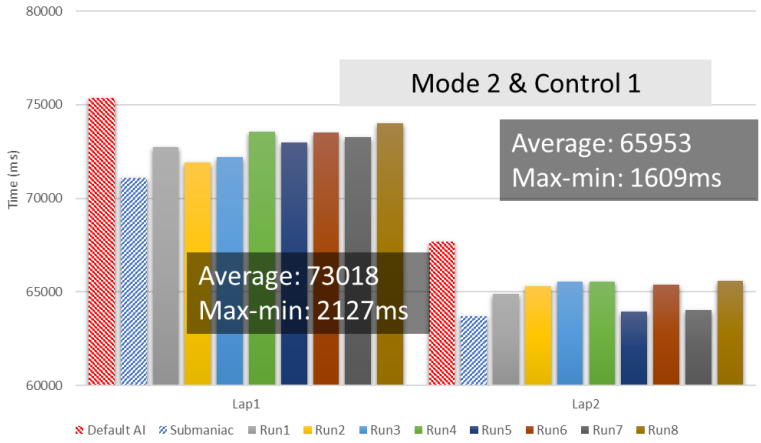


Figure 53 - SOFIA Mode 2 and Control 1

The lap-times are like the previous examples, however what is remarkable, is that the policy formed by this agent is similar to one that has the does not have the ability to brake but has continuous analogue controls available for both steering and the accelerator. This is demonstrated by the profiles in Figure 54. The vehicle speed profiles are nearly identical except at the end of the final lap (which is an abnormal behaviour observed in other agents).

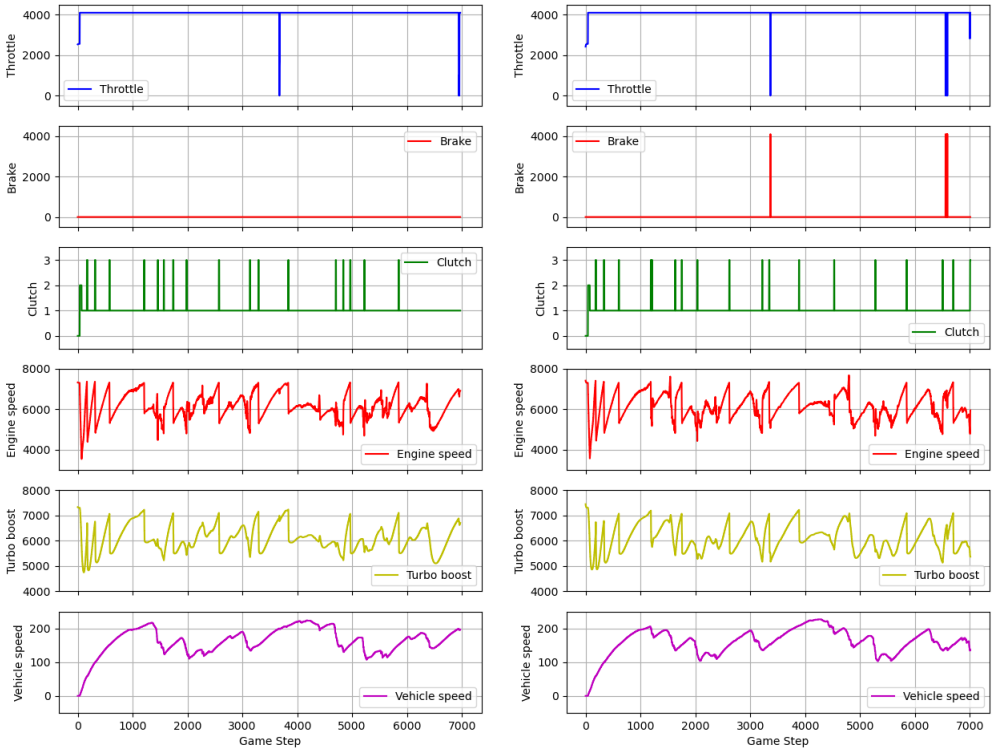


Figure 54 - Control 1 (Left) compared to Control 1.6 (Right)

This result also demonstrates another trend of using mode 2 is that there is a tendency for it to take a larger number of training steps to converge as highlighting by .

Two different configurations using mode 3 show that the average reward is converging towards to one equivalent to an agent able to complete 2 laps. Whereas the trainings conducted with mode 2 seem to get trapped into a global non-optimum policy and take approximately 100,000 to 200,000 training steps more to converge to a better global-optimum policy.

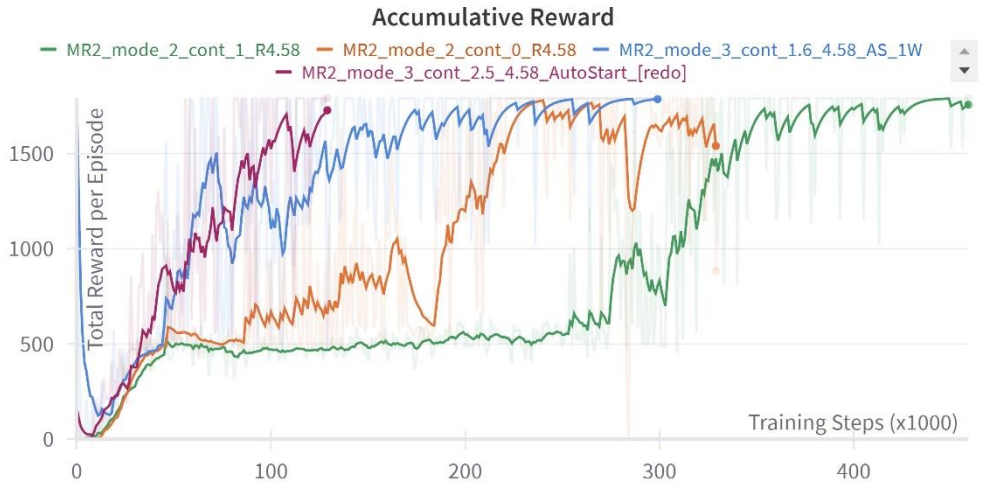


Figure 55 - Mode 2 against Mode 3 Training Efficiency

It is suspected that it is caused by the way mode 2 encodes wall contacts as a single variable. During the first two corners, the vehicle will tend to contact the wall by the right side of the vehicle. The experience may be encouraging the agent to turning left when it detects a wall contact. However, at the third corner – the first right corner, this policy is not effective. During this ineffective state, it was possible to observe the agent turning further into the wall. Therefore, it is suspected that these longer training steps are required to unlearn that relationship and form a new healthier one.

This result demonstrates that a larger a number of observation dimensions such as mode 3, may be more time-efficient, than one with fewer dimension such as mode 2 which is more memory-efficient. Ideally the wall collision would have been encoded as a one-hot vector rather than a continuous parameter.

4.2.3 Almost Great Results

One version of SOFIA that emerged as the best performer during post analysis. That is a version that utilised Mode 3 and Control 2, but with an increased observation image to 128 by 128 pixels (a quadruple increase in pixel quantity from the default 64 x 64).

4.2.3.1 Mode 3 and Control 2 and Higher Resolution

Figure 56 demonstrates that this agent was able to beat the expert's first lap-time once (Run1), and the second lap-time three times (run 1, run 2 and run 6), from a total of 8 episodes.

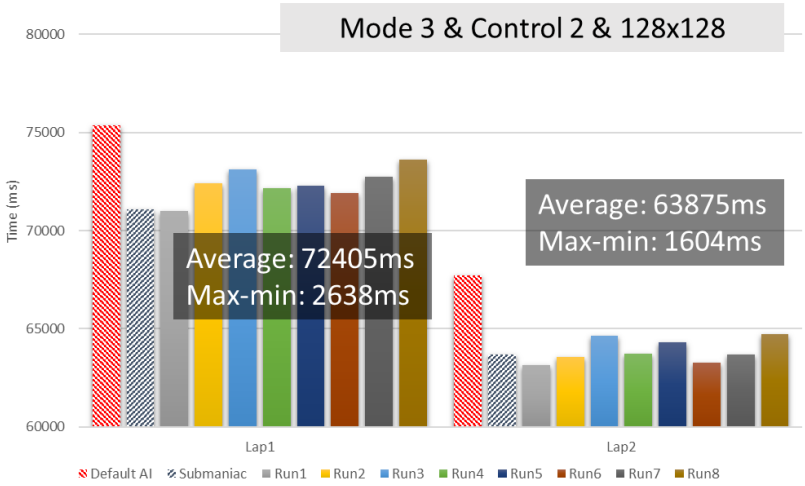


Figure 56 - SOFIA Mode 3 and Control 2 with 128x128 Pixels

The best second lap performance of SOFIA was that of 63.137 seconds, 558ms faster than the expert lap-time. The phenomenal aspect of this positive result is that this version of SOFIA can only utilise discrete actions and does not have the ability to brake. The major difference of this agent in comparison to others is that it has a higher resolution visual observation of 128 by 128 pixels instead of 64 by 64 pixels. This result was not initially expected since 64 by 64 seems to have been sufficient for other similar work in literature such as that on TMRL, however during post analysis, it is evident that 128 by 128 provided a clear advantage, especially when one considers that lap-time improvements are harder to achieve the faster the lap is. This is perhaps why in racing games, the gap between a silver and gold award is less than that between a bronze and silver one. Figure 57 demonstrates the differences of the two resolutions when up-scaled back to the same resolution. The lower resolution image has lost much detail from the image.

Figure 57 - 64 by 64 (left) versus 128 by 128 (right)

It is therefore perhaps not that surprising that this version of SOFIA is able to perform so well without lifting the throttle at all as shown in Figure 58.

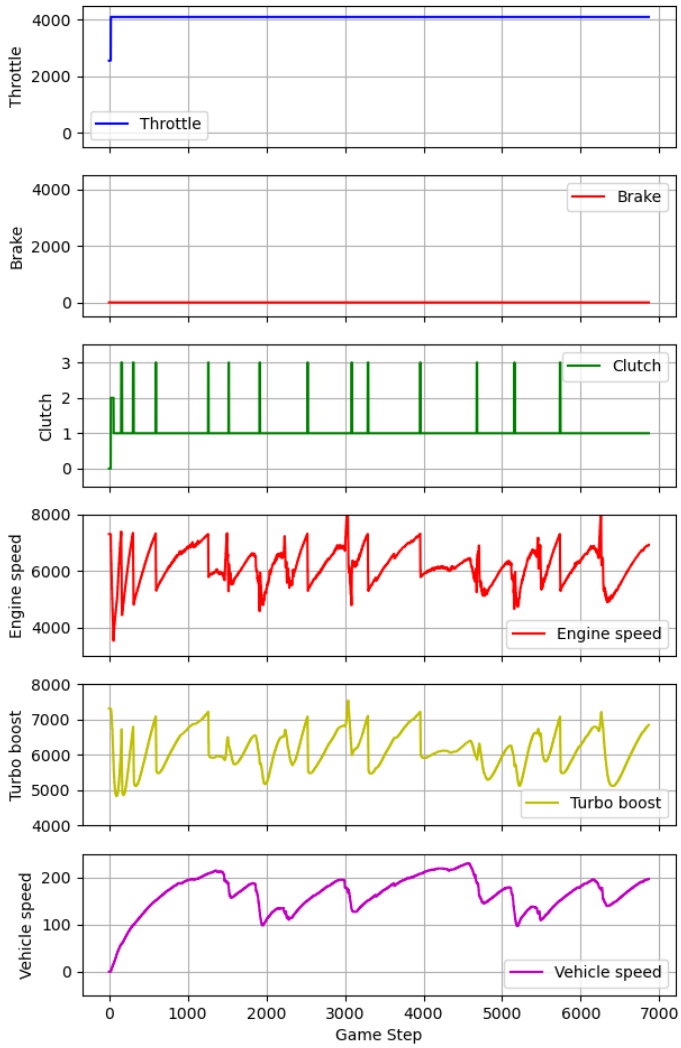


Figure 58 - 128 by 128 Pixel Driving Style

This test also demonstrates that it is possible to train an agent which has its continuous controls collapsed to a discrete one in both acceleration and steering. In fact, looking at the driving line taken as shown in Figure 59, it might be .said that it is as smooth if not smoother looking than the line taken by the SOFA in Figure 50.

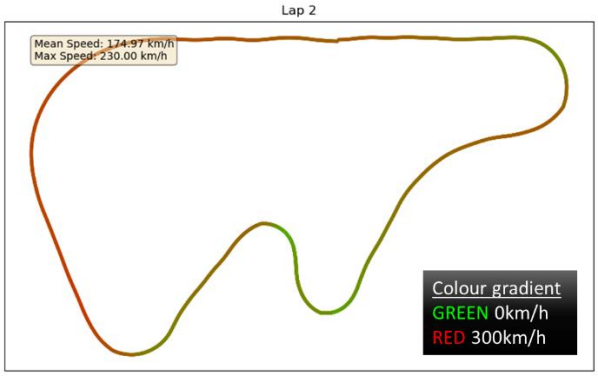


Figure 59 - Driving Line of SOFIA with mode 3 and control 2 with 128 by 128

4.2.4 Acceptable Results

During the definition of modes, one hypothesis previously stated was the assumption that tyre slip may be an important parameter, to test that hypothesis an agent in mode 1 and model 0 were trained.

4.2.4.1 Mode 1

The results support the hypothesis that tyre slip are necessary for a good or better performance. Figure 60 demonstrates that this configuration allows an agent to successfully learn how to complete the race, but on average it is approximately 2 seconds slower than the game's built in AI for the first lap and comparable on the second lap.

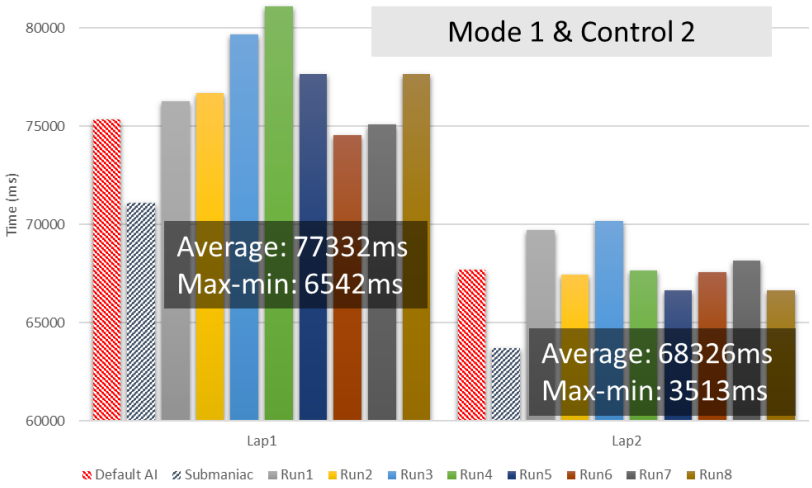


Figure 60 - SOFIA Mode 1 and Control 2

These results were partially surprising because while it was expected that SOFIA might struggle without accurate tyre information, it was not expected to do as well. The training had been conducted using three rollout workers. Assuming that this configuration is more unstable or challenging to converge to an optimal solution (suggested by the higher lap-time variation between runs), which lead to further hypothesised that the variation of hardware performance of the three rollout workers could have further disturbed convergence. Another attempt to re-train an agent using the same mode and control with three rollout workers was done.

The result of the second training attempt was not significantly different in performance from the first. Both trainings resulted in similar lap-times indicated by a similar episode length as shown in Figure 61 for the same maximum possible reward with similar episode length as shown in Figure 62, when using three rollout workers.

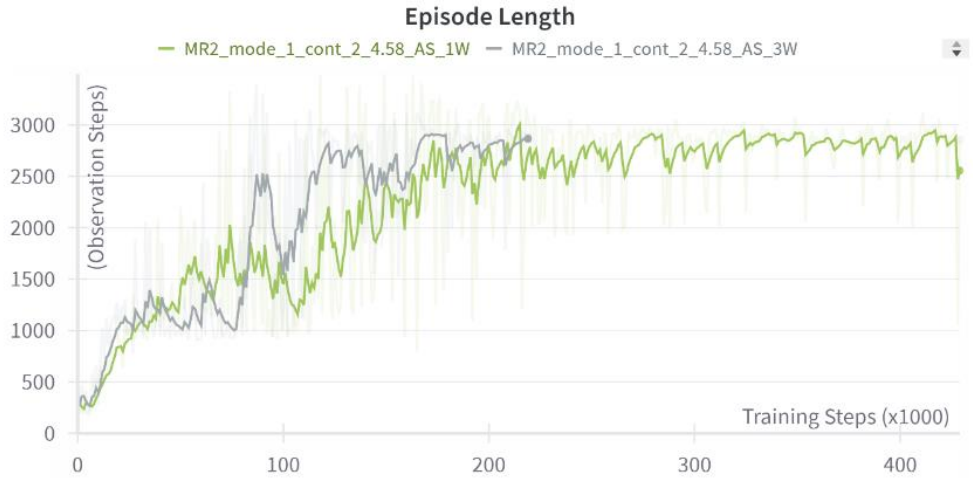


Figure 61 - Mode 1 Episode Length

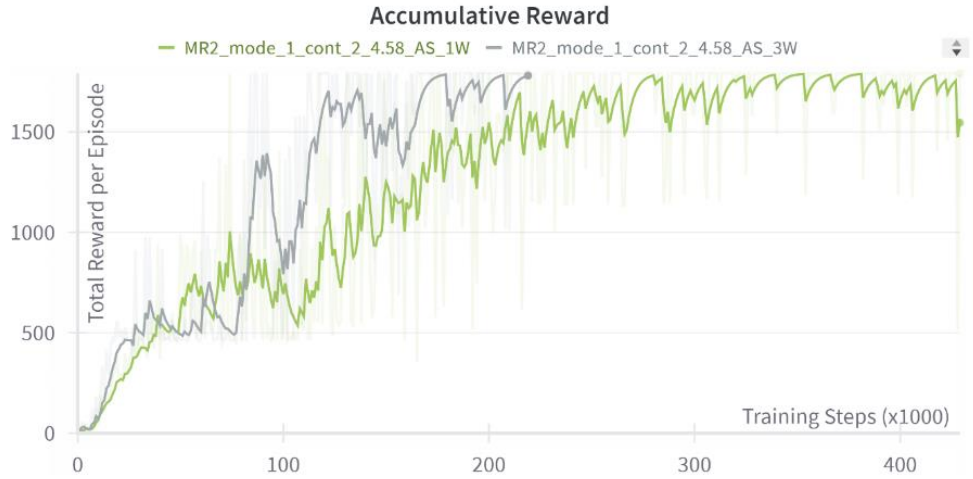


Figure 62 - Mode 1 Episode Reward

It is unsurprising that this version of SOFIA is less smooth around the track as shown in Figure 63.

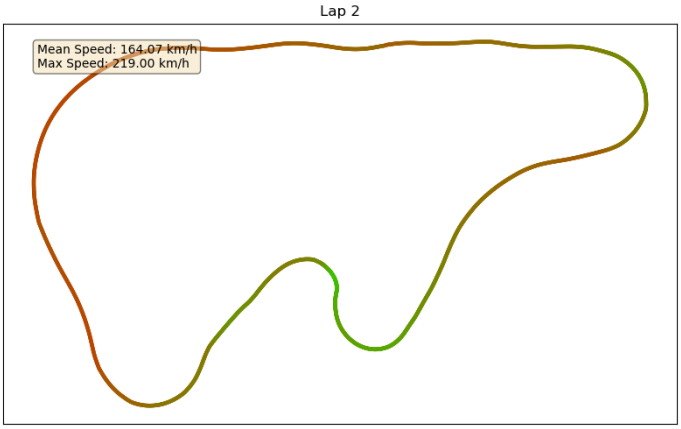


Figure 63 - SOFIA Mode 1 and Control 2

To further investigate the difficulty an agent has with performing well without sufficient reliable parameters, a version of SOFIA without any vehicle or engine parameters was also studied.

4.2.4.2 Mode 0

Mode 0 is the lightest model since it only has a stack of four images, and the last two actions appended to the observation. This version of SOFIA would manage to learn to drive the track but the performance is barely acceptable since as Figure 64 demonstrates, the average lap-time is within 5 seconds of the built-in AI, but the range of lap-times includes laps that are up to 14.6 seconds off.

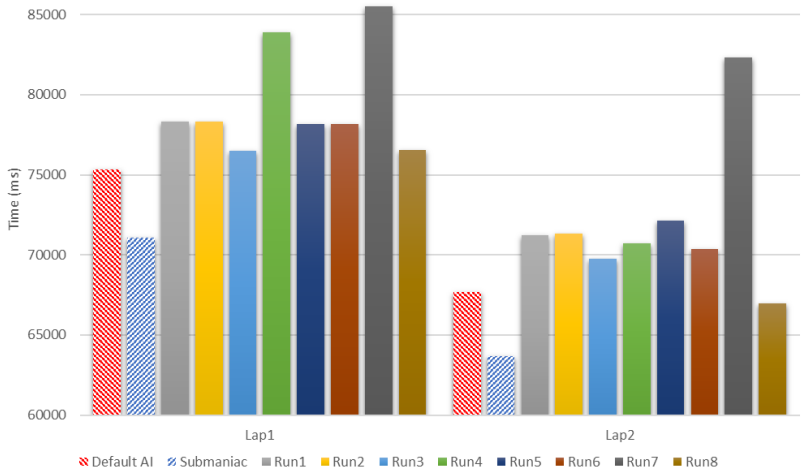


Figure 64 - SOFIA Mode 0 and Control 1.5

Another characteristic of training this configuration of SOFIA was the training-step inefficiency to converge compared to other successful ones.

4.2.4.3 Training Efficiency of Mode 0 and Mode 1

Figure 65 demonstrates the difficulty to train the agent with reduced observation spaces such as mode 1 and mode 0 in comparison to mode 3.

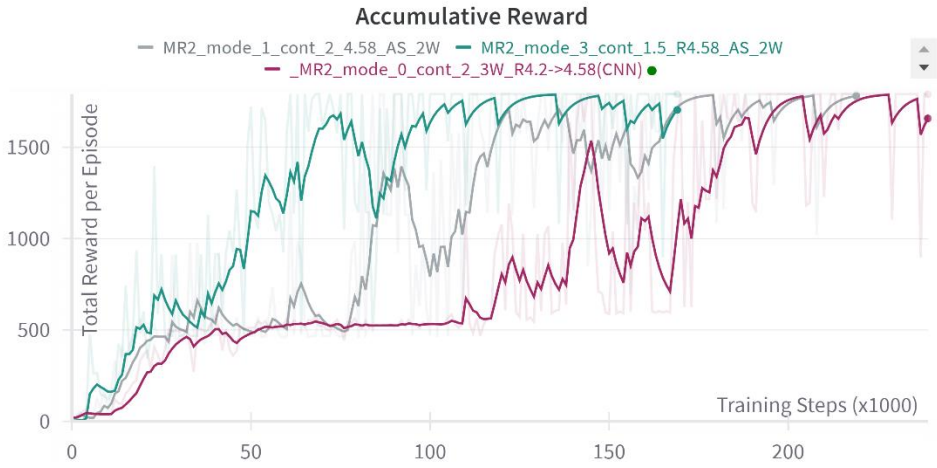


Figure 65 - Comparison of Mode3, Mode1 and Mode 0

In terms of actual process time, the single rollout worker in Mode 3 took approximately 15 hours to reach an acceptable performance with two rollout workers. Mode 1 required the same amount of time despite having the same amount of rollout worker, whereas mode 0 took 60 hours to achieve the previous mentioned performance despite deploying three rollout workers for more than half of the training. The number of rollout workers were reduced to one to improve stability of the agent.

With regards to mode 0, it is not possible to assert that the delayed convergence was not in partly or wholly in influenced by the number of rollout workers, however Figure 65 does suggest a trend of scaling difficult when reducing the observation domain space.

Other factors may have influenced computational performances such as the temperature in the room which may have influenced the degree of the computer's CPU and GPU clock-frequency boosting. An attempt to reduce doubts of such computation power fluctuations by repeating mode 1 based training two weeks apart, while also changing the number of rollout workers Figure 66. The figure illustrates that the rate at which the replay buffer is filled-up is proportionate to the number of rollout workers.

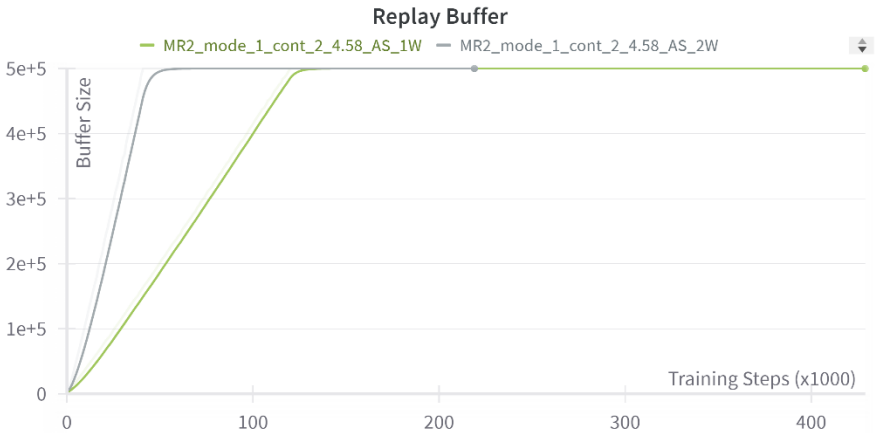


Figure 66 - Replay Buffer with 1 or 2 Rollout Workers

The training with two roll-out workers seems to have started to converge to a stable accumulated reward at 63% of the training steps required for the single rollout worker as shown in Figure 67.

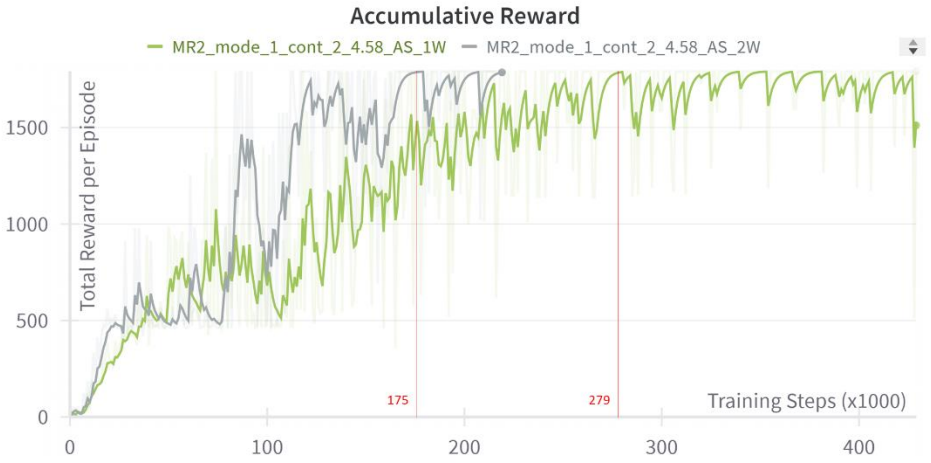


Figure 67 - Accumulative Reward of 1 or 2 Rollout Workers

4.2.4.4 Mode 2 Control 0

Under most conditions, mode 2 and 3 seemed to produce a good performaning agent with any control setting, except control 0. Control 0 is a three-dimension control, which allows for simultaneous braking and acceleration. It is possible that configurations that utilise mode 0 would require longer training sessions, since larger action space is larger when applied, or may require alternative hyper-parameters settings (such as different target entropies or learning rates). Training was stopped at 300,000 steps which was a similar duration to that for most other configurations.

The fact that two of the actions are fundamentally contradictions – accelerate is to move forward, brake is to stop moving forward, may have made it harder for the agent to explore and determine the impact of these actions when applied separately and simultaneously. Comparing the driving style against the reference expert data, this agent behaves differently in that it also was lifting throttle more often as well as braking simultaneously to applying the accelerator as can be seen in Figure 68.

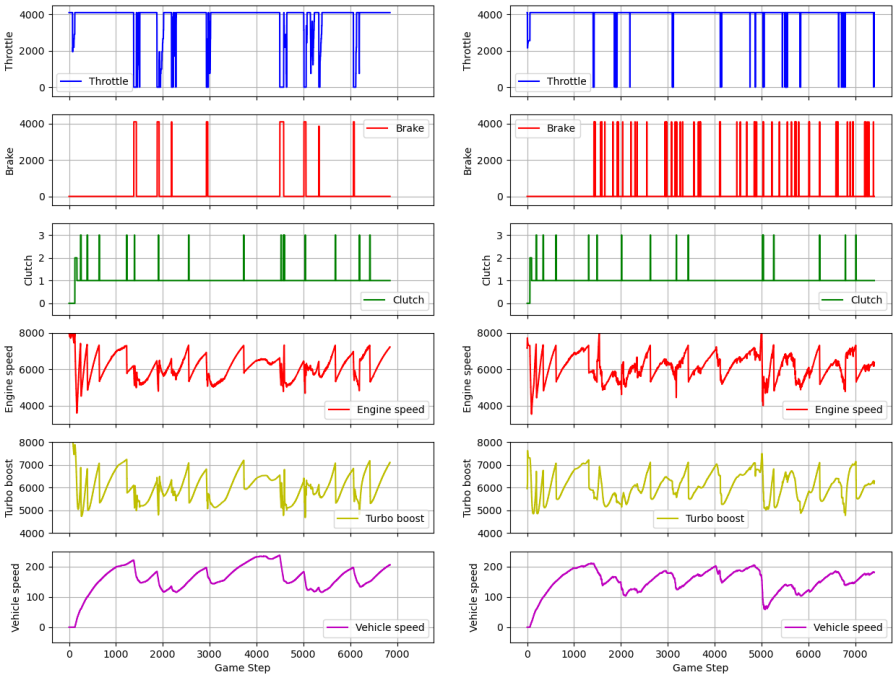


Figure 68 - Submaniac (left) against SOFIA [Mode 2 & Control 0] Driving Style (right)

Nevertheless, this agent seemed to show some potential for elegance to try and avoid hard taps with the wall and in some corners, any taps at all. With such style improvement not exhibited by other configuration, it is hard to dismiss the potential that may exist from using this control configuration. Figure 69 demonstrates that on average this agent is close to the default AI.

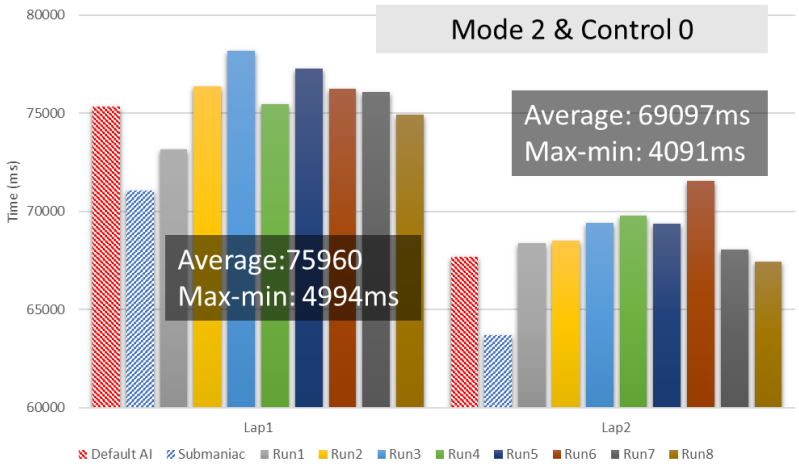


Figure 69 - Mode 2 and Control 0

4.2.5 Bad Results

The previous results have all been above acceptable with the MR2, but certain configurations generated very uncompetitive or incompetent agents. Some of those results will be reviewed in this section.

4.2.5.1 Initial Hyper-Parameter Tuning Attempt

In Figure 70 and Figure 71 examples representing each form of tuning that resulted in failed training attempts.

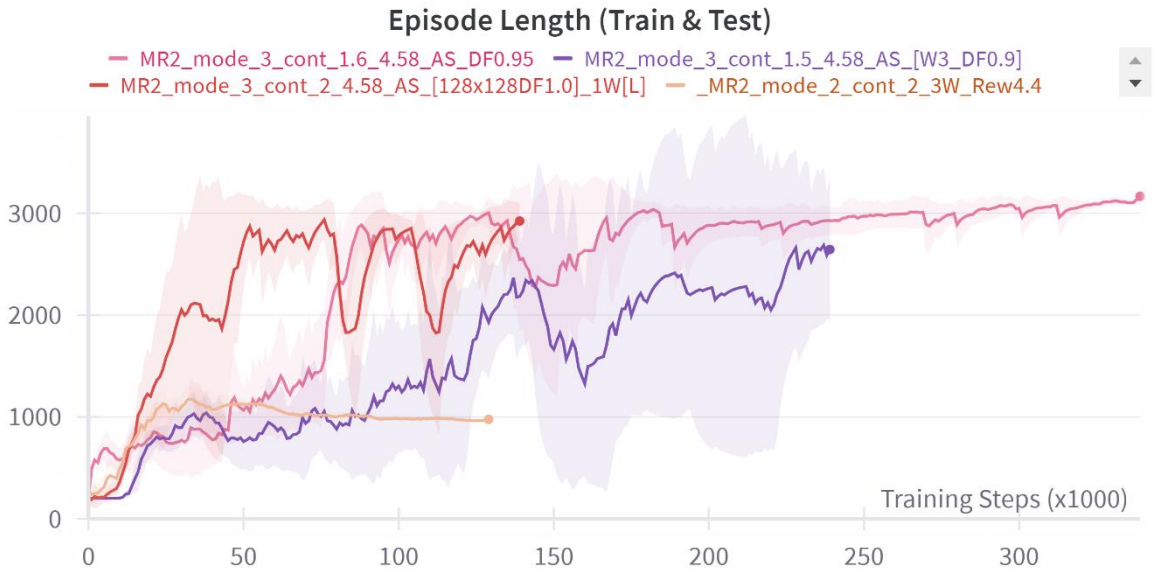


Figure 70 - Episode Length of Failed Training

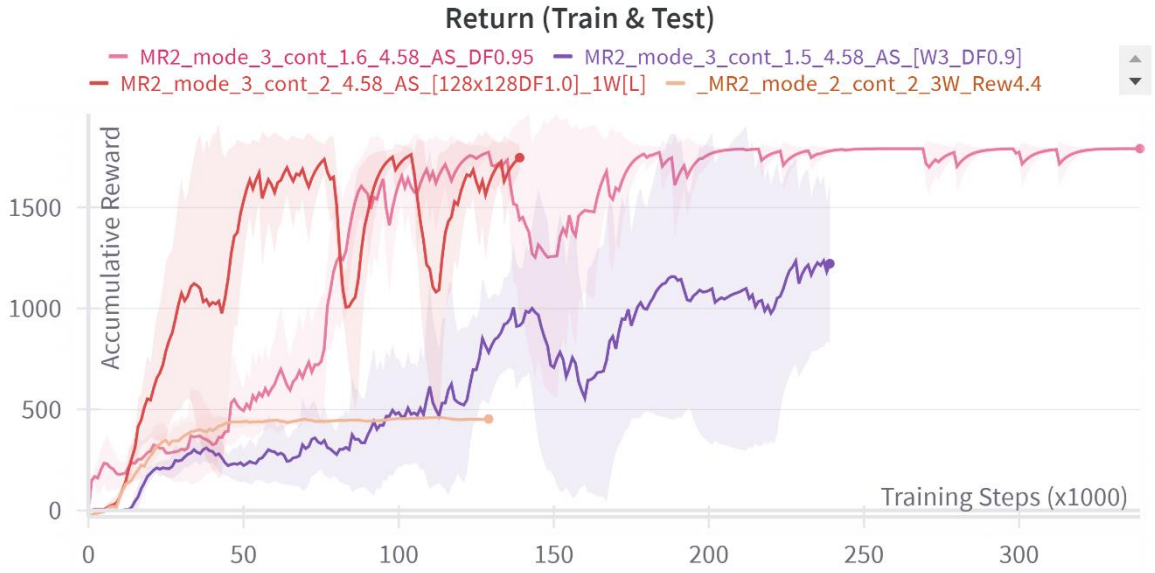


Figure 71 - Accumulative Reward of Failed Training

The pink lines training with mode 3, controls 1.6, had its discount factor slightly decreased to 0.95 (from 0.995). This created an agent that was able to complete two laps indicated by the saturated accumulative reward but does so at a slower lap-time indicated by a larger episode length, which seems to further worsen with more training steps.

Reducing the discount factor further to 0.9 on the run with mode 3 and controls 1.5, as shown by the purple lines produced an unstable policy. This is indicated by the large spread of episode lengths and accumulative reward indicating that the agent cannot always complete a lap. In attempt to log eight consecutive runs of this configuration, it only managed to complete a first lap in two of the eight runs. The lap times were of between 1 min at 33 seconds and 1min and 38 seconds, which is far slower than the default AI capability.

Increasing the discount factor to 1.0 (from the default 0.95) on the run with mode 3 and controls 2 with the image sizes of 128 by 128 produced a policy that seems converge more rapidly than the default discount factor, but also suffers from instability and temporary policy collapse during training as indicated by the two dips in episode length and reward at 84000 training steps and 113000 training steps.

The training with a different reward system 4.3 (as opposed to the final 4.58), cause the policy to converge (in a stable fashion) to one that does not attempt to go round the right corner and rather selects to stop on the track before the corner. In this configuration, the reward function had the $R_{Contact}$ (see equation 8 in 4.1.4) component applied with $K_{Contact}$ set to -0.5. This most likely encouraged that agent to conclude that the probability of contacting the wall during the S-curve part of the track was too high, and thus that it would be a safer strategy to stop.

Unfrotunately due to time limitations it was not possible to conduct a systematic sweep of all parameters in the reward function or the SAC hyperparameters. This was especially due to the fact that for most of the duration of this work, the memory-leak that was occuring due to a bug in the emulator meant that training sessions would need to be manually restarted after a computer crash due to lack of system memory.

4.2.5.2 Supra Attempts

Figure 72 represents one of the final attempts at training an agent to drive the Supra in comparison the MR2. The method of launching the car in motion was developed in hope of overcoming the high lag of the Supra, however, even then agent could not learn to drive more than the first and second corner as it would often spin out. It could not be determined if further training steps may have permitted it to learn to navigate this track section.

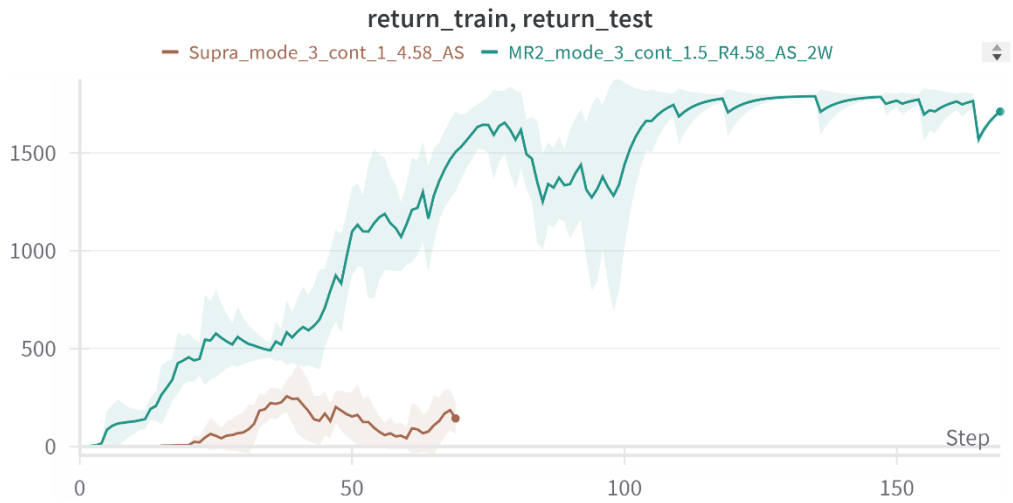


Figure 72 - Supra Training Attempt

The Supra is extremely fast and lightweight, meaning the strategy of hitting the wall as with the MR2 is has a higher likelihood of spinning out. Overcoming this issue was one of the reasons the $R_{Contact}$ function was introduced which obviously with the $K_{Contact}$ value of -0.5 did not work for the Supra as it also did for the MR2.

Chapter 5 Conclusions

This work set out to develop methods for interfacing a PSX emulated environment with Python to allow an RL agent to control a vehicle in the emulated game GT. The objective was to demonstrate that the agent can successfully complete a lap around the High Speed Ring track.

The result is that all objectives have been achieved and the fundamental question about the feasibility of using RL on such a retro game as GT on the PSX have been resolved, despite all the graphical limitations of the console's generation. The results of the agent trained have gone beyond the simple goal of completing a lap, but rather was able to compete against a human expert player lap-time when operating a Toyota MR2 around the targeted High Speed Ring track.

The methodology of interfacing between the emulator and the Python environment can handle both graphical and parametric features. This setup does not require a custom compiler as is the case with PSXLE (Purves et al., 2019), nor does it require external libraries for screen-capturing the visual data as is the case with TMRL (Bouteiller, Geze and GobeX, 2023b) but rather uses a custom lean method of acquiring the very latest video-frame and observation as soon as it is requested by the Python environment. This is done utilising the Lua engine built into PCSX-Redux. This method was proven to exceed the performance that could be achieved by existing screen capture libraries. The data transferred from the emulator to the Python environment is done via TCP, however, unlike other RL work for the PSX such as that with Crash Bash (Costa, 2022) or Mega Man X4 (Severo, 2023b), no assumption is made on the size of the data or the transmission timing of the data. The data is encoded and decoded using Protobuf. The contents of the transmission can be altered to contain different types and size of data while only requiring minor modifications. The TCP connection is also able to transmit requests to the emulator, such as loading a save-state to start a new training episode, or to alter memory variables. The actions are received by the emulator via a virtual gamepad that is simulating a PlayStation 4 controller.

The nature of this architecture allows for operations such as frame-skipping or emulation pausing are possible, or as was used in this work, as a real-time problem simulating a minimum delay with an elastic limit. Such a choice creates a more complex RL problem to be studied but has further reaching applications such for online gameplay where the lag may fluctuate. A real-time approach has not been studied in PSX related work or most RL work involving video games (Mnih et al., 2013; Raffin et al., 2021; Purves, 2019a; Fletcher, 2017) but has only been in found in the work on TRML (Bouteiller, Geze and GobeX, 2023b).

The RL framework is a minimalist one that has few dependencies because it adopts the TMRL library in combination with manually implemented RL algorithms while following the OpenAI Gym API. This means the setup is RL library agnostic and can make use of any library as was done here with the use of an SAC implementation in PyTorch. The code developed for both observing the environment and SAC allows for several rollout-workers to interact with their own GT environment on several machines, further speeding up training.

The environment was altered to allow for variations in observation and action space definitions to reveal similarities and differences of resulting agents. Furthermore, it was possible to use a version of SAC that is said to be exclusively for continuous actions (Spinning Up, 2023) with a discrete action space, without changes to the SAC algorithms themselves without any noticeable difference in performance. The use of SAC without modification to perform discrete actions seems novel in

literature. Through this work it was possible to train agents that relied solely on the display by means of a CNN or on an augmented observation using game-state parameters. To acquire such parameters, as well as to recognise episode relevant parameters such as the race start and finish, a level of reverse engineering GT was required and achieved.

Variations of the action-space and observations are not often highlighted in literature, as most often the literature is highlighting the superiority of the author's algorithm or the success of their agent. Here the main trends from this work are highlighted.

5.1 Main Outcome

The work revealed that, it is possible to train an AI agent to race around the High Speed Ring in time-trial using the MR2 GT-S using only visual data from the game, as demonstrated by mode 0. Conversely it was demonstrated that the visual data alone does not produce a competitive agent, to improve it the agent's observation was best augmented with parametric data. From the studied scenarios, tyre information seemed to be most critical in improving the performance.

This work also uncovered that despite the parametric augmentation, the quality of the visual input to the network in terms of image resolution also had a significant impact. The only setup able to beat the expert human player, was achieved by using 128 x 128-pixel images.

The work also uncovered that for almost all setups, the agent would collapse controls to steering and accelerator (even when braking was permitted as a mutually exclusive accelerator action). When braking was allowed as a separate action, the agent did not achieve similar performance within the permitted training steps.

It was also found that hyper-parameters from a similar game, TrackMania, a game from a completely different generation with superior graphics, resulted in positive racing performance on GT. This may suggest that some of the hyper-parameters, such as the discount factor, have some objective universal meaning in terms of racing.

The work made some arbitrary decisions, such as limiting time-steps to be between 50 and 100ms. Humans are expected to have capability that is in the scale of 200-300ms. While this is perhaps an unfair advantage compared, this setting is like that of other work in literature. This raises the question of how humans perform so well with a lower action speed. One possibility is that humans can forecast future game-states by simulating the environment's behaviour. Humans seem to put their hand out to catch a baseball based on its expected trajectory and not based on the real-time position of the ball. This suggests that humans have built a physics model in their minds to understand how the environment behaves.

This may suggest that perhaps a model-based approach such as that of Dreamer V3 may be more comparable to the way humans perform. Another possibility could be to pass more past actions as part of the observation which might lead to better future forecasting.

The work highlighted that developing a larger observation space (mode 3) that can allow the neural network to more easily converge on an optimum policy, is more time-efficient than an observation space that is smaller (such as mode 2) but is more challenging to converge for.

While the work is a success and introduces several novel components, it also failed to suffice for an attempt using a higher performance and higher turbo-lag vehicle such as the Toyota Supra. This may have been due to hidden variables that broke the MDP criteria, incompatible hyper-parameters, or insufficient training time. The use of a modified Toyota Supra was not part of the formally defined

objectives but serves as a reminder that even in the case of GT, there is scope for further improvements and exploration which can be easily pursued thanks to this novel foundational work.

5.2 Suggestions for Future Work

While this study has achieved its defined objectives, it also lays the groundwork for numerous avenues of future research and potential applications.

A particularly noteworthy prospect is the in-depth exploration of hyper-parameter optimization, with a specific emphasis on achieving optimal performance with the Supra. Addressing this could involve meticulous examination and fine-tuning of hyper-parameters to enhance the overall capabilities of the system.

With the Supra in focus and the broader goal of enhancing general agent performance, there are additional parameters that merit consideration. One such parameter involves introducing an acceleration scaler into the observation. Given that acceleration is a derivative of velocity, relying on two game frames solely from visual observations poses challenges of fluctuations, considering the elastic timing of observations. However, a potential solution lies in calculating acceleration through a Lua script on the emulator side.

Another possible parameter input would be to provide track-foresight. One such implementation could be to compute the normalised velocity vector from the positions of the 2 consecutive observations (as shown by the orange circles and arrow in Figure 73) and scale it by the parametric vehicle speed before passing it to the neural network. This would minimize the error caused by observation delays. Also by taking a number of checkpoints to compute directional vectors (as shown by the blue circles and arrows in Figure 73) and passing it to the neural network, the agent might be able to plan the optimal position to enter the corner rather than reacting to the corner. This approach is similar in concept to the foresight developed for Sophy in GT Sport, but considering an alternative approach that does not require relative track position.

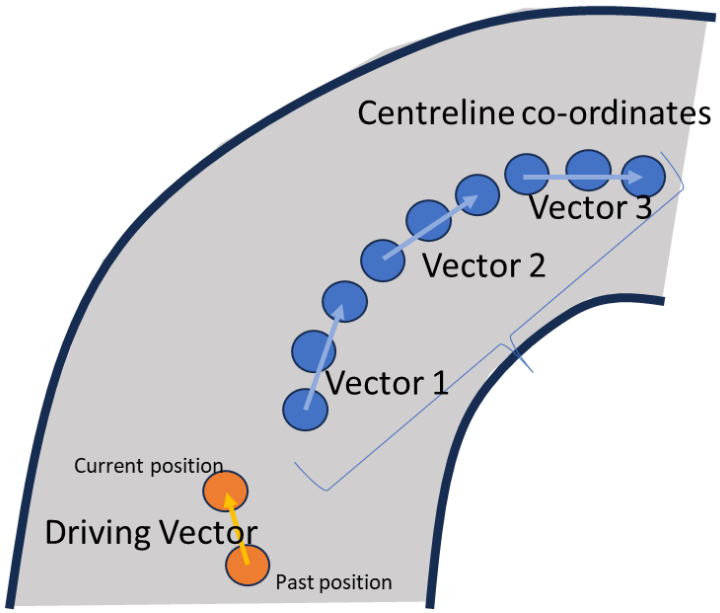


Figure 73 - Potential Foresight

Further investigation into optimal image resolution for feeding the neural network and exploring alternatives to image inputs with parametric values are avenues that warrant exploration in future research.

Code optimization is a critical consideration for enhanced time-step performance. Minimizing the use of *if-else* statements and streamlining the neural network architecture, potentially deviating from the default dual hidden layers of 256 nodes, are areas for potential improvement. Such optimizations align with literature findings related to GT Sport, suggesting potential gains in agent performance.

Considering the unique challenges faced by the agent on the High Speed Ring, it is essential to assess the generalizability of the methodology to other tracks. Examining tracks with features such as sand-trap-like areas and night settings with increased reflections could provide valuable insights.

Lastly, extending the methodology to other PSX racing games presents an intriguing avenue for research, offering the opportunity to gauge the applicability and effectiveness of the developed approach in diverse gaming environments.

5.3 Final Remarks

The project achieved and exceeded the expectations set out by the objectives. Interfaces were implemented that provided novel and flexible ways of fulfilling their purpose with an agent trained via SAC that matched in some instances the performance of a human expert.

Taking on a passion project as this came with several feasibility concerns. The project was relying on overcoming several walls sequentially, being reverse engineering, building the interfaces, and implementing the SAC algorithm. No matter how well time was planned and budgeted, it was not possible to assert in what time frame each objective could be achieved, due to the lack of experience in RE and TCP. The backup alternatives were to focus on a different environment and/or implement primitive inter-process communications. However, such alternatives seemed redundant in comparison to the current state of the art in literature.

That is, it was possible to reverse engineer GT sufficiently to acquire the desired parameters. A robust interface for connecting the emulator environment to Python, by means of Lua scripts on the emulator side and Python code in a real-time fashion was established, and a modern SAC algorithm was implemented and successfully trained.

Two-thirds of the work was spent on the engineering aspects of the project, since prior work such as that of PSXLE contained too many restrictions that limited the value of their application. The limited resources reverse engineer for a platform such as the PSX, could be one reason that such work has rarely been tackled. The RL learning aspects were also more time consuming than anticipated since the expected plug-and-play like selling point of Ray's RLlib did not stand true in this case. This is admittingly possibly due to library dependencies that could not be resolved when deploying the latest versions of Ray.

Murphy's law was not absent in this work, there was a memory leak that occurred during this work which was thought to be due a NN parameter explosion. It would later be found to be a bug in the build of the emulator being used. After recognising from other emulator users that this was a known and solved issue, training sessions could be extended to continuous sessions rather broken into 6 to 8 hour sessions proceeded by a computer crash.

While application of RL to games are various, few focus on the boundaries adopted by this work such as real-time and focus on the PSX. Most of the relevant work on PSX was not documented in the form of publications, and therefore a lot of effort was required to go through the available source code to understand what sort of algorithm or methodology was implemented, as was the case with Crash Bash.

During the work, often new emerging literature would be uncovered. This felt like a moving goalpost and a pressure to re-consider stated approaches. The recent video on YouTube summarising three-years' worth of work on TrackMania (Yosh, 2023) would have been valuable, but at least serve to confirm in retrospect some of the trends that were being observed in this work.

Another such instance is that of Dreamer V3 (Hafner et al., 2023) which demonstrates the ability to deliver fast and efficient RL utilising model methods. Unfortunately, that work could not be considered in time and is inciting to be considered for future work.

Through this work's YouTube updates and GitHub repository, it was possible to inspire others to take on similar projects such as that on MegaMan X albeit it finally taking a different direction. Another such work one which has not yet been made public, relates to RL for Tekken 3 on the PSX. This unreleased work is said by the author to have been inspired by this work's use of PCSX-Redux.

The decision to be open and report the work did not serve the community alone but was also self-serving through unexpected ways. Through contacts with literature authors and discussion on forums, it was possible to develop a network of individuals from all sorts of fields, including a prominent member who has worked on the original Sophy work as well as one of the authors of the TMRL work.

Furthermore, as PCSX-Redux continues to evolve, challenges faced may have been diminished. In a later version of PCSX-Redux, tested in October 2023 (nearly half a year after the reverse engineering phase was predominantly completed new features beneficial to RE were added. One such example is the ability to freeze memory address as shown in Figure 74, unlike what was available and described in section 3.2.2.1.

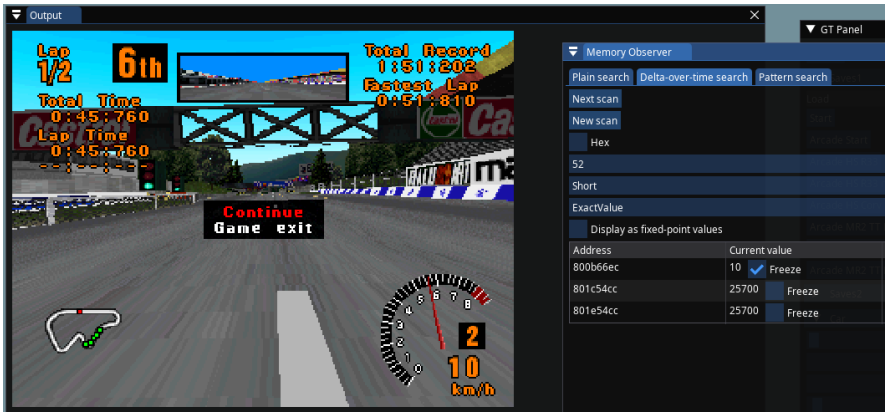


Figure 74 - Freezing the Memory Address Responsible for Displayed Speed

Considering that the engineering aspect of this project was by far the most time-consuming, one must also say that making use of third-party libraries had its own challenges. Updates to different libraries often broke compatibility. One such example, as Gym was superseded by Gymnasium, some of the API calls were altered, and a distinction was introduced to differentiate between a truncated and a terminated state, which was not natively possible in Gym. Therefore, the decision to avoid larger toolchains or libraries such as Ray may have been beneficial for the longer term implications of this work. However, even smaller libraries such as that used for creating the virtual gamepad provided a major compatibility breaking in a version update released during my work. The update inverted the y-axis of analogue sticks. For this reason, despite recognising the memory leak, it was preferable to avoid upgrading libraries mid-work so avoid introducing new problems or sources of variation.

It was also a positive confirmation that it was possible to transfer replays generated onto a real PSX memory card to play against as a ghost in GT. This has finally brought back entertainment value to playing GT after having completed the game.

Nevertheless, this work has managed to be completed beyond personal and objective expectations with an above expert human performance. It was in the end a worthwhile work that expanded person horizons at the least.

Bibliography

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P. and Zaremba, W., 2017. Hindsight Experience Replay. *Advances in Neural Information Processing Systems*, 2017-December, pp.5049–5059. Available from: https://doi.org/10.48550/arxiv.1707.01495 [Accessed 25 January 2023].

Anonymous authors, 2022. ADDRESSING HIGH-DIMENSIONAL CONTINUOUS AC-TION SPACE VIA DECOMPOSED DISCRETE POLICY-CRITIC. *ICLR*. Available from: https://arxiv.org/pdf/2109.05490.pdf [Accessed 29 August 2023].

Vander Ark, T., 2018. *Let's Talk About AI Ethics; We're On A Deadline* [Online]. *Forbes*. Available from: https://www.forbes.com/sites/tomvanderark/2018/09/13/ethics-on-a-deadline/?sh=61573b112e21 [Accessed 23 January 2023].

Baiter, A., 2017. *Gran Turismo (PS1) - PSLZ compression - XeNTaX* [Online]. Available from: https://forum.xentax.com/viewtopic.php?p=129687#p129687 [Accessed 9 September 2023].

Bellemare, M.G., Dabney, W. and Munos, R., 2017. A Distributional Perspective on Reinforcement Learning. *34th International Conference on Machine Learning, ICML 2017*, 1. International Machine Learning Society (IMLS), pp.693–711. Available from: https://doi.org/10.48550/arxiv.1707.06887 [Accessed 25 January 2023].

Bouteiller, Y., 2023a. *yannbouteiller/rtgym: Easily implement custom Gymnasium environments for real-time applications* [Online]. Available from: https://github.com/yannbouteiller/rtgym [Accessed 9 September 2023].

Bouteiller, Y., 2023b. *yannbouteiller/vgamepad: Virtual XBox360 and DualShock4 gamepads in python* [Online]. Available from: https://github.com/yannbouteiller/vgamepad [Accessed 9 September 2023].

Bouteiller, Y., Geze, E. and GobeX, A., 2023a. *tmrl/readme/competition.md at master · trackmania-rl/tmrl* [Online]. Available from: https://github.com/trackmania-rl/tmrl/blob/master/readme/competition.md [Accessed 17 August 2023].

Bouteiller, Y., Geze, E. and GobeX, A., 2023b. *trackmania-rl/tmrl: Reinforcement Learning for real-time applications - host of the TrackMania Roborace League* [Online]. Available from: https://github.com/trackmania-rl/tmrl [Accessed 11 June 2023].

Bouteiller, Y., Ramstedt, S., Beltrame Polytechnique Montreal Christopher Pal Mila, G. and Montreal Jonathan Binas, P., 2021. REINFORCEMENT LEARNING WITH RANDOM DELAYS. In: Y. Bouteiller, S. Ramstedt, G. Beltrame, C. Pal and J. Binas, eds. *ICLR*. Available from: https://arxiv.org/pdf/2010.02966.pdf [Accessed 8 January 2023].

Boyer, T., 2020. *TheoBoyer/TMForge: An opensource tool for reinforcement learning on Trackmania 2020* [Online]. Available from: https://github.com/TheoBoyer/TMForge [Accessed 24 April 2022].

Cai, P., Mei, X., Tai, L., Sun, Y. and Liu, M., 2020. High-speed Autonomous Drifting with Deep Reinforcement Learning. *IEEE robotics and automation letters*, 5(2), pp.1246–1253. Available from: https://sites.google.com/view/autonomous-drifting-with-drl/ [Accessed 3 April 2022].

Canu, S., 2023. *Lines detection with Hough Transform – OpenCV 3.4 with python 3 Tutorial 21 - Pysource* [Online]. Available from: https://pysource.com/2018/03/07/lines-detection-with-hough-transform-opencv-3-4-with-python-3-tutorial-21/ [Accessed 9 September 2023].

Christodoulou, P., 2019. *Soft Actor-Critic for Discrete Action Settings* [Online]. Available from: https://arxiv.org/abs/1910.07207v2 [Accessed 24 December 2023].

Coding Secrets, 2020. *The Coding Secrets hidden in 'Sonic the Hedgehog' - YouTube* [Online]. Available from: https://www.youtube.com/watch?v=Xv88_vTkQFo [Accessed 9 September 2023].

Coding Secrets, 2021. *Tricking the SEGA Genesis to Rotate the Screen Like a SNES - YouTube* [Online]. Available from: https://www.youtube.com/watch?v=Q5iqSzKYZoQ [Accessed 9 September 2023].

Corlett, N., 2011. *PSFLab - Neill Corlett's Home Page* [Online]. Available from: https://web.archive.org/web/20110611190823/http://www.neillcorlett.com/psflab/ [Accessed 10 September 2023].

Costa, M.F., 2022. *RillAi: Teaching a computer to play Crash Bash* [Online]. Available from: https://github.com/mateusfavarin/RillAi [Accessed 22 January 2023].

Dabney, W., Rowland, M., Bellemare, M.G. and Munos, R., 2017. Distributional Reinforcement Learning with Quantile Regression. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp.2892–2901. Available from: https://doi.org/10.48550/arxiv.1710.10044 [Accessed 25 January 2023].

Epic Games, 2023. *Trackmania | Download and Play for Free - Epic Games Store* [Online]. Available from: https://store.epicgames.com/en-US/p/trackmania [Accessed 9 September 2023].

Feinberg, V., Wan, A., Stoica, I., Jordan, M.I., Gonzalez, J.E. and Levine, S., 2018. *Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning* [Online]. Available from: https://doi.org/10.48550/arxiv.1803.00101 [Accessed 25 January 2023].

Fletcher, A., 2017. *How We Built an AI to Play Street Fighter II — Can you beat it? | by Adam Fletcher | GyroscopeSoftware | Medium* [Online]. Available from: https://medium.com/gyroscopesoftware/how-we-built-an-ai-to-play-street-fighter-ii-can-you-beat-it-9542ba43f02b [Accessed 20 August 2023].

Fletcher, A. and Mortensen, J., 2018. Using Python to build an AI to play and win SNES StreetFighter II - PyCon 2018 - YouTube. *PyCon*. Cleveland. Available from: https://www.youtube.com/watch?v=NyNUYYI-Pdg [Accessed 24 April 2022].

Fuchs, F., Song, Y., Kaufmann, E., Scaramuzza, D. and Dürr, P., 2021. Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning. *IEEE ROBOTICS AND AUTOMATION LETTERS.* Available from: https://doi.org/10.1109/LRA.2021.3064284 [Accessed 23 January 2023].

Fujimoto, S., Van Hoof, H. and Meger, D., 2018. Addressing Function Approximation Error in Actor-Critic Methods. *35th International Conference on Machine Learning, ICML 2018*, 4, pp.2587–2601. Available from: https://doi.org/10.48550/arxiv.1802.09477 [Accessed 25 January 2023].

Gavin, A., 2020. *How Crash Bandicoot Hacked The Original Playstation | War Stories | Ars Technica - YouTube* [Online]. Available from: https://www.youtube.com/watch?v=izxXGuVL21o&t=1393s [Accessed 9 September 2023].

Google LLC, 2023. *Techniques | Protocol Buffers Documentation* [Online]. Available from: https://protobuf.dev/programming-guides/techniques/ [Accessed 11 September 2023].

Gordon, C., 2022. 2023 Will Be The Year Of AI Ethics Legislation Acceleration. *Forbes*. Available from: https://www.forbes.com/sites/cindygordon/2022/12/28/2023-will-be-the-year-of-ai-ethics-legislation-acceleration/?sh=498c2349e855 [Accessed 23 January 2023].

Goudarzi, S., 2023. *Special: Will AI destroy art? Or just change it? - Bulletin of the Atomic Scientists*. *Bulletin of the Atomic Scientists*. Available from: https://thebulletin.org/2022/10/will-ai-destroy-art-or-just-change-it/ [Accessed 21 January 2023].

Gran-Turismo Fandom, 2023. *High Speed Ring | Gran Turismo Wiki | Fandom* [Online]. Available from: https://gran-turismo.fandom.com/wiki/High_Speed_Ring?file=Old+high+speed+ring+map.png [Accessed 16 December 2023].

Ha, D. and Urgen Schmidhuber, J.¨, 2018. *World Models* [Online]. Available from: https://doi.org/10.5281/ZENODO.1207631 [Accessed 25 January 2023].

Haarnoja, T., Zhou, A., Abbeel, P. and Levine, S., 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *35th International Conference on Machine Learning, ICML 2018*, 5, pp.2976–2989. Available from: https://doi.org/10.48550/arxiv.1801.01290 [Accessed 23 January 2023].

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P. and Levine, S., 2019. *Soft Actor-Critic Algorithms and Applications* [Online]. Available from: https://arxiv.org/abs/1812.05905v2 [Accessed 23 December 2023].

Hafner, D., 2016. *Deep Reinforcement Learning From Raw Pixels in Doom* [Online]. Available from: [Accessed 30 August 2023].

Hafner, D., Pasukonis, J., Ba, J. and Lillicrap, T., 2023. *Mastering Diverse Domains through World Models* [Online]. Available from: [Accessed 13 August 2023].

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2017. *Rainbow: Combining Improvements in Deep Reinforcement Learning* [Online]. Available from: www.aaai.org [Accessed 19 August 2023].

Hester, B., 2019. *How PlayStation democratized 3D video games - Polygon* [Online]. Available from: https://www.polygon.com/features/2019/12/5/20997745/how-playstation-democratized-3d-video-games [Accessed 22 January 2023].

IGN, 2023. *Crash Bash - IGN* [Online]. Available from: https://www.ign.com/games/crash-bash [Accessed 28 August 2023].

Jaritz, M., De Charette, R., Toromanoff, M., Perot, E. and Nashashibi, F., 2018. *End-to-End Race Driving with Deep Reinforcement Learning* [Online]. Available from: http://team.inria.fr/rits/drl. [Accessed 23 January 2023].

Joe Fox, 2018. *Gran Turismo 1 - AI Fail - YouTube* [Online]. Available from: https://www.youtube.com/watch?v=ESGiM841RGw [Accessed 27 February 2023].

Jupp, E., 2022. *[Video] Gran Turismo 7 vs. real life | GRR* [Online]. Available from: https://www.goodwood.com/grr/road/news/2022/3/video-gran-turismo-7-vs.-real-life/ [Accessed 22 January 2023].

keras-rl, 2019. *GitHub - keras-rl/keras-rl: Deep Reinforcement Learning for Keras.* [Online]. Available from: https://github.com/keras-rl/keras-rl [Accessed 13 August 2023].

Kinsley, H., 2017. *Line Finding with Hough Lines - Python plays Grand Theft Auto 5 p.5 - YouTube* [Online]. Available from: https://www.youtube.com/watch?v=lhMXDqQHf9g [Accessed 9 September 2023].

Kinsley, H. and Kukiela, D., 2017. *Sentdex/pygta5: Explorations of Using Python to play Grand Theft Auto 5.* [Online]. Available from: https://github.com/Sentdex/pygta5 [Accessed 9 September 2023].

Kohler, C., 2018. *PlayStation Classic Plays Fine, But It's A Bare-Bones Experience* [Online]. 8 November. Available from: https://kotaku.com/playstation-classic-plays-fine-but-it-s-a-bare-bones-e-1830294616 [Accessed 26 August 2023].

Korth, M., 2022. *Memory Map - PlayStation Specifications - psx-spx* [Online]. Available from: https://psx-spx.consoledev.net/memorymap/ [Accessed 10 April 2022].

Levinovitz, A., 2014. The Mystery of Go, the Ancient Game That Computers Still Can't Win | WIRED. *WIRED Magazine*, 12 May. Available from: https://www.wired.com/2014/05/the-world-of-computer-go/ [Accessed 22 January 2023].

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J.E. and Stoica, I., 2018. Tune: A Research Platform for Distributed Model Selection and Training. *Arxiv*. Available from: https://arxiv.org/abs/1807.05118v1 [Accessed 16 September 2023].

Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D., 2015. Continuous control with deep reinforcement learning. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*. Available from: https://doi.org/10.48550/arxiv.1509.02971 [Accessed 25 January 2023].

Linuzappz, Shadow, Bernett, P., NoComp, Nik3d and Akumax, 2023. *pcsxr/AUTHORS at master · iCatButler/pcsxr · GitHub* [Online]. Available from: https://github.com/iCatButler/pcsxr/blob/master/AUTHORS [Accessed 26 August 2023].

Loiacono, D., Prete, A., Lanzi, P.L. and Cardamone, L., 2010. Learning to overtake in TORCS using simple reinforcement learning. *2010 IEEE World Congress on Computational Intelligence, WCCI 2010 - 2010 IEEE Congress on Evolutionary Computation, CEC 2010*. Available from: https://doi.org/10.1109/CEC.2010.5586191 [Accessed 23 January 2023].

MattJ155, 2021. *Gran Turismo 2: The top 5 best AI fails - YouTube* [Online]. Available from: https://www.youtube.com/watch?v=5bKp24HcXJU [Accessed 27 February 2023].

McLaren, 2020. *McLaren Racing - Lights out and away we go?* [Online]. Available from: https://www.mclaren.com/racing/team/carlos-sainz-lando-norris-reaction-time/ [Accessed 25 January 2023].

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing Atari with Deep Reinforcement Learning. *arXiv*. Available from: https://arxiv.org/pdf/1312.5602.pdf [Accessed 24 April 2022].

Mnih, V., Puigdomènech Badia, A., Mirza, M., Graves, A., Harley, T., Lillicrap, T.P., Silver, D. and Kavukcuoglu, K., 2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv*. Available from: [Accessed 23 January 2023].

Mohamed, A., 2008. *Artificial Intelligence in Racing Games* [Online]. Available from: https://www.cs.bham.ac.uk/~ddp/AIP/RacingGames.pdf [Accessed 9 April 2022].

Nagabandi, A., Kahn, G., Fearing, R.S. and Levine, S., 2017. *Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning* [Online]. Available from: https://sites.google.com/view/mbmf [Accessed 25 January 2023].

Neinders, L., 2023. *Improving Trackmania Reinforcement Learning Performance: A Comparison of Sophy and Trackmania AI* [Online]. University of Twente. Available from: https://essay.utwente.nl/96153/1/Neinders_BA_EEMCS.pdf [Accessed 30 June 2023].

Noble, N., 2023a. *GitHub - grumpycoders/pcsx-redux: The PCSX-Redux project is a collection of tools, research, hardware design, and libraries aiming at development and reverse engineering on the PlayStation 1. The core product itself, PCSX-Redux, is yet another fork of the Playstation emulator, PCSX.* [Online]. Available from: https://github.com/grumpycoders/pcsx-redux [Accessed 26 August 2023].

Noble, N., 2023b. *Loaded libraries - PCSX-Redux* [Online]. Available from: https://pcsx-redux.consoledev.net/Lua/libraries/#lua-protobuf [Accessed 12 September 2023].

Noble, N., 2023c. *Web server - PCSX-Redux* [Online]. Available from: https://pcsx-redux.consoledev.net/web_server/ [Accessed 16 September 2023].

Noble, N., Herben, A., Ryusan, Peach, wheremyfoodat, Mariano, Baumann, J., Yates, C. and Favarin, M., 2023a. *grumpycoders/pcsx-redux: The PCSX-Redux project is a collection of tools, research, hardware design, and libraries aiming at development and reverse engineering on the PlayStation 1. The core product itself, PCSX-Redux, is yet another fork of the Playstation emulator, PCSX.* [Online]. Available from: https://github.com/grumpycoders/pcsx-redux [Accessed 9 September 2023].

Noble, N., Herben, A., Ryusan, Peach, wheremyfoodat, Mariano, Baumann, J., Yates, C. and Favarin, M., 2023b. *PCSX-Redux* [Online]. Available from: https://pcsx-redux.consoledev.net/ [Accessed 9 September 2023].

nocash, M. and Noble, N., 2023. *Graphics Processing Unit (GPU) - PlayStation Specifications - psx-spx* [Online]. Available from: https://psx-spx.consoledev.net/graphicsprocessingunitgpu/#texture-palettes-clut-color-lookup-table [Accessed 16 September 2023].

NSA, 2023. *Ghidra* [Online]. Available from: https://ghidra-sre.org/ [Accessed 9 September 2023].

O'donoghue, B., Munos, R., Kavukcuoglu, K., Volodymyr, & and Deepmind, M., 2017. COMBINING POLICY GRADIENT AND Q-LEARNING. *arXiv*. Available from: [Accessed 19 August 2023].

OpenAI, 2022. *Part 1: Key Concepts in RL — Spinning Up documentation* [Online]. Available from: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html [Accessed 25 January 2023].

OpenCV, 2023a. *OpenCV: Canny Edge Detection* [Online]. Available from: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html [Accessed 12 September 2023].

OpenCV, 2023b. *OpenCV: Hough Line Transform* [Online]. Available from: https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html [Accessed 12 September 2023].

Peng, B., Sun, Q., Shengbo, ·, Li, E., Kum, D., Yin, Y., Wei, J. and Gu, T., 2021. End-to-End Autonomous Driving Through Dueling Double Deep Q-Network. *Automotive Innovation*, 4, pp.328–337. Available from: https://doi.org/10.1007/s42154-021-00151-3 [Accessed 19 October 2022].

Perkins, C., 2017. *Left-Foot Braking - How and Why to Left Foot Brake in Racing*. *Road & Track*. Available from: https://www.roadandtrack.com/car-culture/videos/a32553/how-and-why-to-left-foot-brake-racing-rally/ [Accessed 13 August 2023].

Porcher, P., 2023. *Pedro AI - How PedroAI works* [Online]. Available from: https://www.trackmania.ai/blog/posts/pedroai-rl/pedroai-rl.html#reward-function [Accessed 9 September 2023].

Purves, C., 2019a. *GitHub - carlospurves/psxle: A Python interface to the Sony PlayStation console.* [Online]. Available from: https://github.com/carlospurves/psxle [Accessed 26 August 2023].

Purves, C., 2019b. *The PlayStation Reinforcement Learning Environment* [Online]. Jesus College. Available from: https://catalina17.github.io/files/cp614-dissertation.pdf [Accessed 24 April 2022].

Purves, C., At˘, C., Cangea, A. and Veličkovi´c, V., 2019. The PlayStation Reinforcement Learning Environment (PSXLE). *arXiv preprint arXiv:1912.06101*. Available from: https://github.com/pcsxr/PCSX-Reloaded/ [Accessed 20 August 2023].

Racanière, S., Weber, T., Reichert, D.P., Buesing, L., Guez, A., Rezende, D., Badia, A.P., Vinyals, O., Heess, N., Li, Y., Pascanu, R., Battaglia, P., Hassabis, D., Silver, D. and Wierstra, D., 2017. Imagination-Augmented Agents for Deep Reinforcement Learning. *Advances in Neural Information Processing Systems*, 2017-December, pp.5691–5702. Available from: https://doi.org/10.48550/arxiv.1707.06203 [Accessed 25 January 2023].

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M. and Dormann, N., 2021. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*, 22, pp.1–8. Available from: https://github.com/DLR-RM/stable-baselines3. [Accessed 16 September 2023].

Ray, 2023a. *Algorithms — Ray 2.9.0* [Online]. Available from: https://docs.ray.io/en/latest/rllib/rllib-algorithms.html [Accessed 23 December 2023].

Ray, 2023b. *Algorithms — Ray 3.0.0.dev0* [Online]. Available from: https://docs.ray.io/en/master/rllib/rllib-algorithms.html [Accessed 13 August 2023].

Ray, 2023c. *Environments — Ray 2.9.0* [Online]. Available from: https://docs.ray.io/en/latest/rllib/rllib-env.html [Accessed 23 December 2023].

Ray, 2023d. *RL Modules (Alpha) — Ray 2.9.0* [Online]. Available from: https://docs.ray.io/en/latest/rllib/rllib-rlmodule.html [Accessed 24 December 2023].

Rogers, A., 2019. What Deep Blue And AlphaGo Can Teach Us About Explainable AI. *Forbes*. Available from: https://www.forbes.com/sites/forbestechcouncil/2019/05/09/what-deep-blue-and-alphago-can-teach-us-about-explainable-ai/?sh=1699bd6952fd [Accessed 21 January 2023].

Schulman, J., Levine, S., Moritz, P., Jordan, M. and Abbeel, P., 2015. Trust Region Policy Optimization. *32nd International Conference on Machine Learning, ICML 2015*, 3, pp.1889–1897. Available from: https://doi.org/10.48550/arxiv.1502.05477 [Accessed 25 January 2023].

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. *Proximal Policy Optimization Algorithms* [Online]. Available from: https://doi.org/10.48550/arxiv.1707.06347 [Accessed 25 January 2023].

Severo, V., 2023a. *INTELIGÊNCIA ARTIFICIAL AMASSANDO os CHEFES de Mega Man X4 - YouTube* [Online]. Available from: https://www.youtube.com/watch?v=uYRemfDmwTk [Accessed 26 August 2023].

Severo, V., 2023b. *victorsevero/bizket* [Online]. Available from: https://github.com/victorsevero/bizket [Accessed 22 January 2023].

Shoemaker, B., 2006. *The Greatest Games of All Time: Doom - GameSpot* [Online]. 2 February. Available from: https://www.gamespot.com/articles/the-greatest-games-of-all-time-doom/1100-6143094/ [Accessed 30 August 2023].

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. and Hassabis, D., 2017. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm* [Online]. Available from: https://doi.org/10.48550/arxiv.1712.01815 [Accessed 25 January 2023].

Song, Y., Lin, H., Kaufmann, E., Dürr, P. and Scaramuzza, D., 2021. Autonomous Overtaking in Gran Turismo Sport Using Curriculum Reinforcement Learning. *IEEE International Conference on Robotics and Automation (ICRA)*. Available from: https://youtu.be/e8TVPv4D4O0 [Accessed 11 June 2023].

Sony, 2008. *GRAN TURISMO™ SERIES SHIPMENT EXCEEDS 50 MILLION UNITS WORLDWIDE | PRESS RELEASES / Sony Computer Entertainment Inc.* [Online]. Available from: https://www.sie.com/en/corporate/release/2008/080509.html [Accessed 22 January 2023].

Spinning Up, 2023. *Soft Actor-Critic — Spinning Up documentation* [Online]. Available from: https://spinningup.openai.com/en/latest/algorithms/sac.html [Accessed 23 December 2023].

Spranger, M., 2023. *From Research to Deployment in One Year - GT Sophy Is Now Available to All GT7 Players – Sony AI* [Online]. Available from: https://ai.sony/blog/blog-027/ [Accessed 4 June 2023].

Squaresoft74, 2022. *Conversation with Squaresoft74*.

Stable-baslines3, 2023. *RL Algorithms — Stable Baselines3 2.1.0a4 documentation* [Online]. Available from: https://stable-baselines3.readthedocs.io/en/master/guide/algos.html [Accessed 13 August 2023].

Subramanian, K., Fuchs, F. and Seno, T., 2022. *Training the World's Fastest Gran Turismo Racer – Sony AI* [Online]. Available from: https://ai.sony/blog/blog-019/ [Accessed 30 August 2023].

Sutton, R.S. and Barto, A.G., 2018. *Reinforcement Learning: An Introduction Adaptive Computation and Machine Learning*. Second. The MIT Press.

Syed, N., 2020. *psxBreakout a PSX PS1 Demo for CS50x - YouTube* [Online]. Available from: https://www.youtube.com/watch?v=oKt2dAhkS7E [Accessed 9 September 2023].

Syed, N.Z., 2021. *BuzzyBee: A homebrew PSX game as an entry to the 32 bit Jam. Thanks to all the members who supported with the audio and the art.* [Online]. Available from: https://github.com/NDR008/BuzzyBee [Accessed 22 January 2023].

Szeto, G.P.Y., Straker, L.M. and O'Sullivan, P.B., 2005. The effects of typing speed and force on motor control in symptomatic and asymptomatic office workers. *International Journal of Industrial Ergonomics*, 35(9), pp.779–795. Available from: https://doi.org/10.1016/J.ERGON.2005.02.008 [Accessed 30 December 2023].

TASEmulators, 2023. *GitHub - TASEmulators/BizHawk: BizHawk is a multi-system emulator written in C#. BizHawk provides nice features for casual gamers such as full screen, and joypad support in addition to full rerecording and debugging tools for all system cores.* [Online]. Available from: https://github.com/TASEmulators/BizHawk [Accessed 26 August 2023].

The Ray Team, 2023a. *Overview — Ray 2.6.3* [Online]. Available from: https://docs.ray.io/en/latest/ray-overview/index.html [Accessed 16 September 2023].

The Ray Team, 2023b. *RLlib: Industry-Grade Reinforcement Learning — Ray 2.6.3* [Online]. Available from: https://docs.ray.io/en/latest/rllib/index.html [Accessed 16 September 2023].

Thubron, R., 2021. *Ripping and Tearing: 27 Years of Doom | TechSpot*. *TECHSPOT*. Available from: https://www.techspot.com/article/2245-doom/ [Accessed 30 August 2023].

Ubisoft, 2023. *Trackmania | Ubisoft (EU / UK)* [Online]. Available from: https://www.ubisoft.com/en-gb/game/trackmania/trackmania [Accessed 9 September 2023].

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A.S., Yeo, M., Makhzani, A., Kuettler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J. and Tsing, R., 2017. *StarCraft II: A New Challenge for Reinforcement Learning* [Online]. Available from: https://en.wikipedia.org/wiki/Professional_StarCraft_competition [Accessed 2 January 2024].

Wang, X., 2023. *starwing/lua-protobuf: A Lua module to work with Google protobuf* [Online]. Available from: https://github.com/starwing/lua-protobuf [Accessed 12 September 2023].

Wurman, P.R., Barrett, S., Kawamoto, K., MacGlashan, J., Subramanian, K., Walsh, T.J., Capobianco, R., Devlic, A., Eckert, F., Fuchs, F., Gilpin, L., Khandelwal, P., Kompella, V., Lin, H., MacAlpine, P., Oller, D., Seno, T., Sherstan, C., Thomure, M.D., Aghabozorgi, H., Barrett, L., Douglas, R., Whitehead, D., Dürr, P., Stone, P., Spranger, M. and Kitano, H., 2022. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*, 602(7896), pp.223–228. Available from: https://doi.org/10.1038/s41586-021-04357-7 [Accessed 5 March 2022].

Yosh, 2023. *Training an unbeatable AI in Trackmania - YouTube* [Online]. Available from: https://www.youtube.com/watch?v=Dw3BZ6O_8LY [Accessed 27 December 2023].

Appendix A – Example of Reverse Engineering Code

Function responsible for tracking lap-time records and the maximum speed achieved in a race.

Raw Ghidra output:

```
void FUN_8002e21c(void){
  int iVar1;
  int iVar2;

  iVar1 = (int)DAT_800b6700;
  iVar2 = (int)DAT_800b619c;
  if (iVar1 != 0) {
    iVar1 = iVar1 + -1;
  }
  if (iVar2 == 0) {
    iVar2 = 100;
  }
  if (iVar1 < iVar2) {
    FUN_8002d8c4(DAT_80093bc8);
    FUN_8002da04(DAT_80093bc8 - DAT_800b66f4);
  }
  if ((DAT_800b6226 == 0) && (DAT_800bd990 < DAT_800b66ec)) {
    DAT_800bd990 = DAT_800b66ec;
  }
  if (DAT_8009056a != 0) {
    DAT_8009056a = DAT_8009056a + -1;
  }
  if (DAT_8009056c != 0) {
    DAT_8009056c = DAT_8009056c + -1;
  }
  return;
}
```

Partially reversed engineering code:

```
void FUN_8002e21c(void){
  int local_hiddeCurrentLap;
  int local_MaxLaps;

  local_hiddeCurrentLap = (int)(short)GlobalCar.currentLapCountsFrom0;
  local_MaxLaps = (int)MaxLaps;
  if (local_hiddeCurrentLap != 0) {
    local_hiddeCurrentLap = local_hiddeCurrentLap + -1;
  }
  if (local_MaxLaps == 0) {
    local_MaxLaps = 100;
  }
  if (local_hiddeCurrentLap < local_MaxLaps) {    /* total time? */
    totalLapTime(currentTotalLapTime);                /* lap_time? */
    FUN_8002da04(currentTotalLapTime - GlobalCar.lastLapTime);
  }
  if ((raceMode == racing) && (hiddenMaxSeenSpeed < GlobalCar.currentSpeed)) {
    hiddenMaxSeenSpeed = GlobalCar.currentSpeed;
  }
  if (DAT_8009056a != 0) {
    DAT_8009056a = DAT_8009056a + -1;
  }
  if (DAT_8009056c != 0) {
    DAT_8009056c = DAT_8009056c + -1;
  }
  return;
}
```

Appendix B – Screenshot Benchmark

```
# just to find screen limits

screenXY = []

import pyautogui as py
import time
for i in range(0, 2):
    print("position", i+1)
    time.sleep(3)
    screenXY.append(py.position())
x1,y1 = screenXY[0]
x2,y2 = screenXY[1]
print(x1,y1,x2,y2)

Frames = 120

import numpy as np
from PIL import ImageGrab
import cv2
from time import process_time_ns, time
def screen_record_pil():
    start_time = time()
    for i in range(0, Frames):
        printscreen = np.array(ImageGrab.grab(bbox=(x1, y1, x2, y2)))
        cv2.imshow('window', cv2.cvtColor(printscreen, cv2.COLOR_BGR2RGB))
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.destroyAllWindows()
            break
    end_time = time()
    cv2.destroyAllWindows()
    return (end_time - start_time)

import pyautogui
def screen_record_py():
    start_time = time()
    for i in range(0, Frames):
        printscreen = np.array(pyautogui.screenshot(region=(x1, y1, x2, y2)))
        cv2.imshow('window', cv2.cvtColor(printscreen, cv2.COLOR_BGR2RGB))
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.destroyAllWindows()
            break
    end_time = time()
    cv2.destroyAllWindows()
    return (end_time - start_time)




import mss
def screen_record_mss():
    start_time = time()
    bbox = (x1, y1, x2, y2)
    for i in range(0, Frames):
        with mss.mss() as sct:
            printscreen = np.array(sct.grab(bbox))
            cv2.imshow('window', printscreen)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.destroyAllWindows()
            break
    end_time = time()
    cv2.destroyAllWindows()
    return (end_time - start_time)
```

```
import d3dshot
def screen_record_d3dshot_sc():
    screen_buffer = d3dshot.create(
        capture_output="numpy")
    screen_buffer.display = screen_buffer.displays[0]
    start_time = time()
    for i in range(0, Frames):
        printscreen = screen_buffer.screenshot(region=(x1, y1, x2, y2))
        cv2.imshow('window', cv2.cvtColor(printscreen, cv2.COLOR_BGR2RGB))
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.destroyAllWindows()
            break
    end_time = time()
    cv2.destroyAllWindows()
    screen_buffer.stop()
    return (end_time - start_time)


def screen_record_d3dshot_buff():
    screen_buffer = d3dshot.create(
        capture_output="numpy", frame_buffer_size=20)
    screen_buffer.display = screen_buffer.displays[0]
    screen_buffer.capture(region=(x1, y1, x2, y2))
    start_time = time()
    for i in range(0, Frames):
        printscreen = screen buffer.get latest frame()
        cv2.imshow('window', cv2.cvtColor(printscreen, cv2.COLOR_BGR2RGB))
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.destroyAllWindows()
            break
    end_time = time()
    cv2.destroyAllWindows()
    screen_buffer.stop()
    return (end_time - start_time)


print("PIL: ", 1/(screen_record_pil()/Frames))
print("pyautogui: ", 1/(screen record py()/Frames))
print("mss: ", 1/(screen_record_mss()/Frames))
print("d3d_sc: ", 1/(screen_record_d3dshot_sc()/Frames))
print("d3d_buff: ", 1/(screen_record_d3dshot_buff()/Frames))
```

Appendix C – Protobuf Definition

```
syntax = "proto3";

// Not necessary for Python but should still be declared to avoid name collisions in the
Protocol Buffers namespace and non-Python languages
package GT;

enum BPP {
    BPP_16 = 0;
    BPP_24 = 1;
}

message PosVect {
    int32 x = 1;
    int32 y = 2;
    int32 z = 3;
}

message GameState {
    int32 raceState = 1;
}

message Vehicle {
    int32 engSpeed = 1;
    int32 engBoost = 2;
    int32 engGear = 3;
    int32 speed = 4;
    sint32 steer = 5;
    int32 pos = 6;
    int32 fLeftSlip = 7;
    int32 fRightSlip = 8;
    int32 rLeftSlip = 9;
    int32 rRightSlip = 10;
    int32 eClutch = 11;
    int32 fLWheel = 12;
    int32 fRWheel = 13;
    int32 rLWheel = 14;
    int32 rRWheel = 15;
}

message Screen {
    bytes data = 1;
    int32 width = 2;
    int32 height = 3;
    BPP bpp = 4;
}

message Observation {
    Screen SS = 1;
    GameState GS = 2;
    Vehicle VS = 3;
    int32 frame = 4;
    PosVect posVect = 5;
    int32 trackID = 6;
    int32 drivingDir = 7;
}
```

Appendix D – Server & Client Connection

Server Side

```
import socket
import game_pb2 as Game
import numpy as np
from PIL import Image
import cv2
from time import sleep, time
from enum import Enum


class messageState(Enum):
    mPing = 1  # expect a simple "P"
    mRecvHeader = 2  # header (which contains the size)
    mRecvSize = 3  # size of the data
    mRecvData = 4  # actual raw data


class server():
    def __init__(self, ip='localhost', port=9999, debug=False):
        """Returns a GT AI server object
        Parameters:
        ip is a hostname as string (default localhost)
        port as int (default 9999)
        benchmark true / false (default false) * checks fps
        """
        self.debug = debug
        self.ip = ip
        self.port = port
        self.mState = messageState.mPing.name
        self.header = bytearray()
        self.mSize = 0
        self.message = bytearray()
        self.pic = None
        self.myData = Game.Observation()
        self.fullData = False
        self.connection = None
        self.clientAddress = None
        self.excpt = False
        self.lostPing = 0
        self.buffer = None
        self.lastFrame = 0
        self.sock = None
        print("GT AI Server instantiated for rtgym")

    def connect(self):
        """Starts up the server ready for a single connection
        """
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        serverAddress = (self.ip, self.port)
        self.sock.setblocking(True)
        self.sock.bind(serverAddress)
        self.sock.listen(1)
        print('starting up on {} port {}'.format(*serverAddress))
        self.receiveClient()
        # self.lostPing = False

    def recvall(self, expectedSize):
        """Returns an expected number of bytes from the socket connection
        This function will not return until all the data has been received
        """
        data = bytearray()
        while len(data) < expectedSize:
            packet = self.connection.recv(expectedSize - len(data))
            if not packet:
                return None
            data.extend(packet)
        return data
```

```
    def toNumpy(self, im):
        """This function converts a PIL image into a a numpy array
        """
        im.load()
        # unpack data
        tmpImage = Image._getencoder(im.mode, 'raw', im.mode)
        tmpImage.setimage(im.im)

        # NumPy buffer for the result
        shape, typestr = Image._conv_type_shape(im)
        #data = np.empty(shape, dtype=np.dtype(typestr))
        data = np.empty(shape, dtype='uint8')
        mem = data.data.cast('B', (data.data.nbytes,))
        buffSize, s, offset = 65536, 0, 0
        while not s:
            l, s, d = tmpImage.encode(buffSize)
            mem[offset:offset + len(d)] = d
            offset += len(d)
        if s < 0:
            raise RuntimeError("encoder error %d in tobytes" % s)
        return data

    def decodeImg(self):
        """Protobuf decoding of the screenshot and converts into an numpy array
        """
        size = (self.myData.SS.width, self.myData.SS.height)
        if self.myData.SS.bpp == 0:
            # not actually 16bpp... BGR555
            self.pic = self.toNumpy(Image.frombuffer("RGB", size, self.myData.SS.data, 'raw',
'BGR;15', 0, 1))
        elif self.myData.SS.bpp == 1:
            self.pic = self.toNumpy(Image.frombuffer("RGB", size, self.myData.SS.data, 'raw',
'BGR', 0, 1))
        else:
            print("no clue what to do with this image")

    def receive(self):
        """this function will receive as much data as possible
        and if possible decode it
        """
        self.part = bytearray()
        try:
            if self.mState == messageState.mPing.name:
                self.buffer = self.recvall(1)  # can be removed later
                if self.buffer is not None and self.buffer.decode() == 'P':
                    self.lostPing = False
                    self.mState = messageState.mRecvHeader.name
                else:
                    self.lostPing = True

            if self.mState == messageState.mRecvHeader.name:
                self.part.clear()
                if len(self.header) < 4:
                    self.part = self.connection.recv(4 - len(self.header))
                    self.header.extend(self.part)
                else:
                    self.mSize = int.from_bytes(self.header, 'little')
                    self.mState = messageState.mRecvData.name

            if self.mState == messageState.mRecvData.name:
                self.part.clear()
                if len(self.message) < self.mSize:
                    self.part = self.connection.recv(self.mSize - len(self.message))
                    self.message.extend(self.part)
                else:
                    self.buffer = self.recvall(1)  # can be removed later
                    if self.buffer is not None and self.buffer.decode() == 'D':
                        self.fullData = True
                        self.mState = messageState.mPing.name
                        self.myData.ParseFromString(self.message)
                        self.header.clear()
                        self.message.clear()
        except socket.error as e:
            self.excpt = True

    def sendPong(self, pong):
```

```
        self.connection.send(pong.to_bytes(4, 'little'))

    def receiveClient(self):
        print("Waiting for a connection")
        self.connection, self.clientAddress = self.sock.accept()
        self.connection.setblocking(False)
        print('Connection from', self.clientAddress)

    # rename startReceiving to run for threading
    def receiveAllAlways(self):
        while True:
            try:
                self.sendPong(1)
                self.receive()
                if self.excpt and self.fullData:
                    self.excpt = False
                    self.fullData = False
                    self.decodeImg()
                    if self.debug:
                        #size = self.pic.shape
                        #self.pic = cv2.resize(self.pic, (size[1] * 2, size[0] * 2),
interpolation=cv2.INTER_NEAREST)
                        cv2.imshow('Preview Display', self.pic)
                        self.lastFrame = self.myData.frame
                        if cv2.waitKey(1) & 0xFF == ord('q'):
                            cv2.destroyAllWindows()
                            print('Forced Exit')
                            return
            except:
                print("lost")
                return

    def receiveOneFrame(self):
        self.sendPong(1)
        while True:
            try:
                self.receive()
                if self.excpt and self.fullData:
                    self.excpt = False
                    self.fullData = False
                    self.decodeImg()

                    self.lastFrame = self.myData.frame
                    #size = self.pic.shape
                    #self.pic = cv2.resize(self.pic, (size[1] * 2, size[0] * 2),
interpolation=cv2.INTER_NEAREST)
                    if self.debug:
                        cv2.imshow('Preview Display', self.pic)
                        if cv2.waitKey(0) & 0xFF == ord('q'):
                            cv2.destroyAllWindows()
                            print('Forced Exit')
                            return
                    else:
                        return
            except:
                print("Exception on single frame")
                return

    def reloadSave(self, trackChoice):
        print("reload save for track :", trackChoice)
        self.sendPong(trackChoice) # loads the save state
        sleep(0.05) # Redux takes a few milliseconds to load the savestate
```

Client Side (v1)

```
-- Protobuf related
local pb = require('pb')
local protoc = require('protoc')
local frames = 0
local frames_needed = 1
local obs = {}
local client = nil
local reconnectTry = false
local CurrentPos = 0
local maxLostPings = 500
local currentMissedPings = 0

local function read_file_as_string(filename)
    local file = Support.File.open(filename)
    local buffer = file:read(tonumber(file:size()))
    file:close()
    return tostring(buffer)
end

local function check_load(chunk)
    local pbdata = protoc.new():compile(chunk)
    local ret, offset = pb.load(pbdata)
    if not ret then
        error("load error at " .. offset ..
            "\nproto: " .. chunk ..
            "\ndata: " .. buffer(pbdata):tohex())
    end
end

local proto_file = read_file_as_string('game.proto')
check_load(proto_file)

-- REDUX related
local mem = PCSX.getMemPtr()

local function readGameState()
    local gameState = {}
    local raceStart = readValue(mem, 0x800b6d60, 'uint8_t*')
    local raceMode = readValue(mem, 0x800b6226, 'uint8_t*')
    local racing = readValue(mem, 0x8008df72, "int8_t*")
    if racing ~= 58 then
        gameState['raceState'] = 5 -- not in race
    elseif raceStart == 1 then
        gameState['raceState'] = 1 -- race start
    elseif raceMode == 0 then
        gameState['raceState'] = 2 -- racing
    else
        gameState['raceState'] = 3 -- race finished
    end
    return gameState
end

local function readVehicleState()
    local vehicleState = {}
    vehicleState['engSpeed'] = readValue(mem, 0x800b66ee, 'uint16_t*')
    vehicleState['engBoost'] = readValue(mem, 0x800b66f8, "uint16_t*")
    vehicleState['engGear'] = readValue(mem, 0x800b66e8, "uint8_t*")
    vehicleState['speed'] = readValue(mem, 0x800b66ec, 'uint8_t*')
    vehicleState['steer'] = readValue(mem, 0x800b66d6, "int16_t*")
    vehicleState['pos'] = readValue(mem, 0x800b6d69, "int16_t*")
    vehicleState['eClutch'] = readValue(mem, 0x800b6d63, "uint16_t*")
    vehicleState['fLeftSlip '] =readValue(mem, 0x800b674e,'uint8_t*')
    vehicleState['fRightSlip'] =readValue(mem, 0x800b6792,'uint8_t*')
    vehicleState['rLeftSlip'] = readValue(mem, 0x800b67d6,'uint8_t*')
    vehicleState['rRightSlip']= readValue(mem, 0x800b681a,'uint8_t*')
    vehicleState['fLWheel'] =readValue(mem, 0x800b6778, 'int8_t*')
    vehicleState['fRWheel'] =readValue(mem, 0x800b67bc, 'int8_t*')
    vehicleState['rLWheel'] =readValue(mem, 0x800b6800, 'int8_t*')
    vehicleState['rRWheel'] =readValue(mem, 0x800b6844, 'int8_t*')
    return vehicleState
end

local function readVehiclePositon()
    local posVect = {}
```

```
    posVect['x'] = readValue(mem, 0x800b6704, 'int32_t*')
    posVect['y'] = readValue(mem, 0x800b6708, 'int32_t*')
    -- posVect['z'] = readValue(mem, 0x800b670c, 'int32_t*')
    return posVect
end
-- TCP related

function grabGameData()
    local screen = PCSX.GPU.takeScreenShot()
    screen.data = tostring(screen.data)
    screen.bpp = tonumber(screen.bpp)
    local gameState = readGameState()
    local vehicleState
    local posVect = readVehiclePositon()
    if gameState['raceState'] < 6 then
        vehicleState = readVehicleState()
        lap = readValue(mem, 0x800b6700, 'int8_t*')
        if lap == 0 and gameState['raceState'] == 1 then
            CurrentPos = 0
            obs['trackID'] = CurrentPos
        else
            local x = readValue(mem, 0x800b6704, 'int32_t*')
            local y = readValue(mem, 0x800b6708, 'int32_t*')
            CurrentPos = closestPoints(Xc, Yc, x, y) + (lap - 1) * TrackMaxID
            --if CurrentPos < TrackMaxID
            obs['trackID'] = CurrentPos
        end
    else
        obs['trackID'] = 0
    end
    -- print(obs['tackID'], lap, gameState['raceState'])
    obs['SS'] = screen
    obs['GS'] = gameState
    obs['VS'] = vehicleState
    obs['frame'] = frames
    obs['posVect'] = posVect
    obs['drivingDir'] = readValue(mem, 0x800b6e74, 'int16_t*')
    GlobalData = assert(pb.encode("GT.Observation", obs))
end

function netTCP(netChanged, netStatus)
    -- this seciton is messy
    local turnOn = (reconnectTry or netChanged)
    if turnOn then
        if netStatus then
            client = Support.File.uvFifo("127.0.0.1", 9999)
            frames = 0
            reconnectTry = false
        else
            client:close()
        end
    -- main loop
    elseif netStatus then
        local readVal = client:readU16() -- receive a 1 or 2
        local ready = false

        -- 1 is the main loop for frame capture
        if readVal == 1 then
            ready = true
            currentMissedPings = 0
        -- 2 is for loading a savestate
        elseif readVal == 2 then
            local file = Support.File.open("arc5.slice", "READ")
            PCSX.loadSaveState(file)
            file:close()
        elseif readVal == 3 then
            local file = Support.File.open("mr2_400.slice", "READ")
            PCSX.loadSaveState(file)
            file:close()
        elseif readVal == 4 then
            local file = Support.File.open("sim5.slice", "READ")
            PCSX.loadSaveState(file)
            file:close()
        elseif readVal == 5 then
            local file = Support.File.open("sim6.slice", "READ")
            PCSX.loadSaveState(file)
```

```
            file:close()
        else
            currentMissedPings = currentMissedPings + 1
        end

        if currentMissedPings > maxLostPings then
            currentMissedPings = 0
            reconnectTry = true
            print("Retry to connect to server")
            ready = false
        end
        -- keep track of the number of frames rendered
        frames = frames + 1

        -- if this is the nth frame and we have previously received a 1
        if (frames % frames_needed) == 0 and ready then
            grabGameData() -- take screenshot, encode it with protobuf and get ready to send
it
            client:write("P") -- send "P" for the python server to know we are ready
            client:writeU32(#GlobalData) -- send the size of the chunk of data
            client:write(GlobalData) -- send the actual chunk of data
            client:write("D")            -- send "P" for the python server to know we are
ready
        end
    end
end
```

Client Side (v2) (Skipping the count-down phase from the episode)

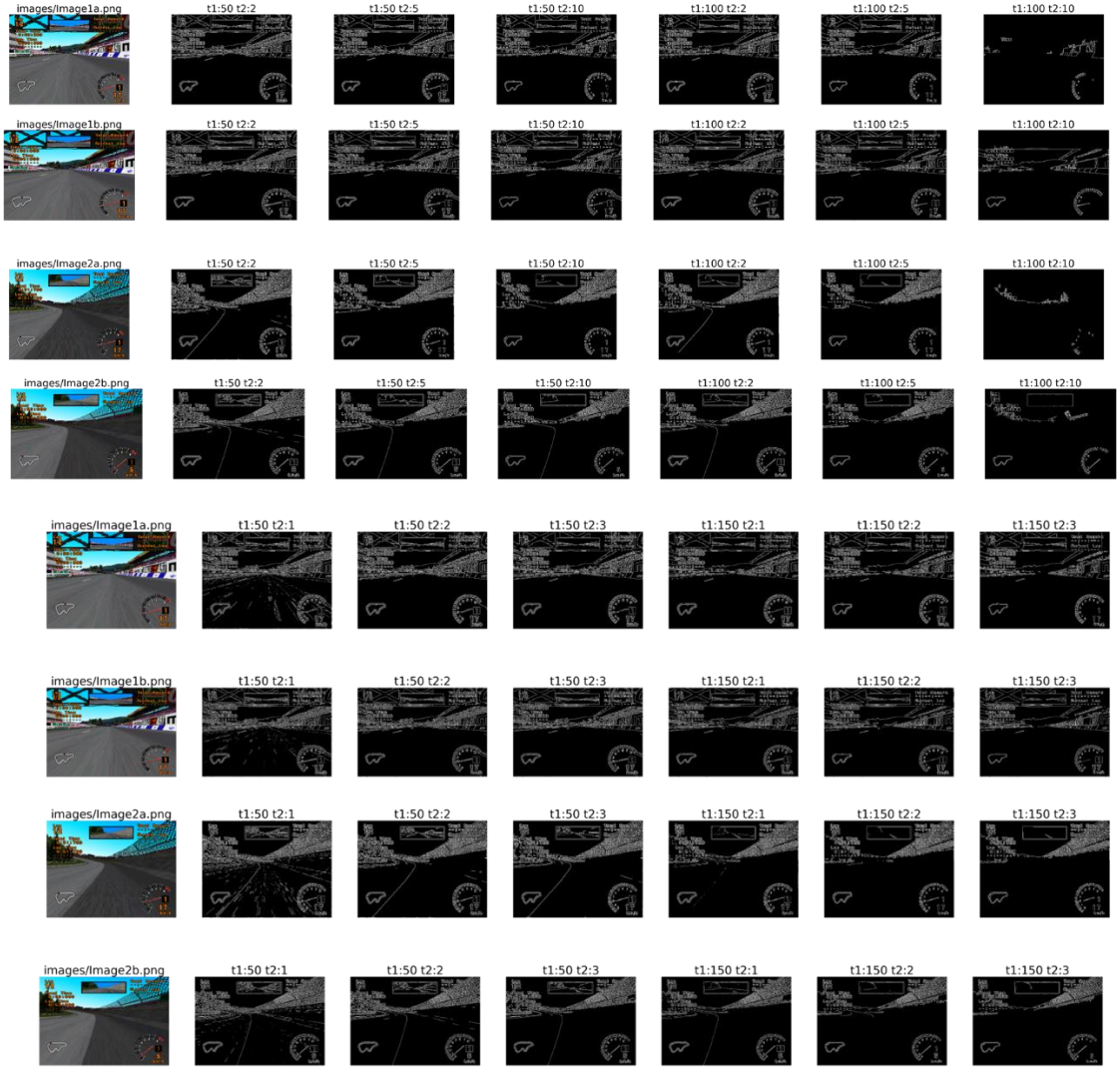
```
-- Protobuf related
local pb = require('pb')
local protoc = require('protoc')
local frames = 0
local frames_needed = 1
local obs = {}
local client = nil
local reconnectTry = false
local ready = false
local takeControl = false

local function read_file_as_string(filename)
    local file = Support.File.open(filename)
    local buffer = file:read(tonumber(file:size()))
    file:close()
    return tostring(buffer)
end

local function check_load(chunk)
    local pbdata = protoc.new():compile(chunk)
    local ret, offset = pb.load(pbdata)
    if not ret then
        error("load error at " .. offset ..
            "\nproto: " .. chunk ..
            "\ndata: " .. buffer(pbdata):tohex())
    end
end

local proto_file = read_file_as_string('game.proto')
check_load(proto_file)

-- REDUX related
local mem = PCSX.getMemPtr()

local function readGameState()
    local gameState = {}
    local raceStart = readValue(mem, 0x800b6d60, 'uint8_t*')
    local raceMode = readValue(mem, 0x800b6226, 'uint8_t*')
    local racing = readValue(mem, 0x8008df72, "int8_t*")
    if racing ~= 58 then
        gameState['raceState'] = 5 -- not in race
    elseif raceStart == 1 then
        gameState['raceState'] = 1 -- race start
    elseif raceMode == 0 then
        gameState['raceState'] = 2 -- racing
    else
        gameState['raceState'] = 3 -- race finished
    end
    return gameState
end

local function readCollission()
    local collisionState = readValue(mem, 0x800b66e9, 'int8_t*')
    local collisionValue = readValue(mem, 0x800b66ea, 'int8_t*')

    -- print("pre", HeldCollState, collisionState, collisionValue)
    if collisionValue > 0 and collisionState > 0 then
        HeldCollState = collisionState
    elseif collisionValue == 0 then
        HeldCollState = 0
    end
    -- print("pos", HeldCollState, collisionState, collisionValue)
    return HeldCollState
end

local function readVehicleState()
    local vehicleState = {}
    vehicleState['engSpeed'] = readValue(mem, 0x800b66ee, 'uint16_t*')
    vehicleState['engBoost'] = readValue(mem, 0x800b66f8, "uint16_t*")
    vehicleState['engGear'] = readValue(mem, 0x800b66e8, "uint8_t*")
    vehicleState['speed'] = readValue(mem, 0x800b66ec, 'uint8_t*')
    vehicleState['steer'] = readValue(mem, 0x800b66d6, "int16_t*")
    vehicleState['pos'] = readValue(mem, 0x800b6d69, "int16_t*")
```

```
    vehicleState['eClutch'] = readValue(mem, 0x800b6d63, "uint16_t*")
    vehicleState['fLeftSlip '] = readValue(mem, 0x800b674e, 'uint8_t*')
    vehicleState['fRightSlip'] = readValue(mem, 0x800b6792, 'uint8_t*')
    vehicleState['rLeftSlip'] = readValue(mem, 0x800b67d6, 'uint8_t*')
    vehicleState['rRightSlip'] = readValue(mem, 0x800b681a, 'uint8_t*')
    vehicleState['fLWheel'] = readValue(mem, 0x800b6778, 'int8_t*')
    vehicleState['fRWheel'] = readValue(mem, 0x800b67bc, 'int8_t*')
    vehicleState['rLWheel'] = readValue(mem, 0x800b6800, 'int8_t*')
    vehicleState['rRWheel'] = readValue(mem, 0x800b6844, 'int8_t*')
    vehicleState['vColl'] = readCollission()
    return vehicleState
end

local function readVehiclePositon()
    local posVect = {}
    posVect['x'] = readValue(mem, 0x800b6704, 'int32_t*')
    posVect['y'] = readValue(mem, 0x800b6708, 'int32_t*')
    -- posVect['z'] = readValue(mem, 0x800b670c, 'int32_t*')
    return posVect
end
-- TCP related



function grabGameData()
    local screen = PCSX.GPU.takeScreenShot()
    screen.data = tostring(screen.data)
    screen.bpp = tonumber(screen.bpp)
    local gameState = readGameState()
    local vehicleState = readVehicleState()
    local posVect = readVehiclePositon()

    obs['SS'] = screen
    obs['GS'] = gameState
    obs['VS'] = vehicleState
    obs['frame'] = frames
    obs['posVect'] = posVect
    local vDir = readValue(mem, 0x800b6e74, 'int16_t*')
    if vDir >= 1 then
        vDir = 1
    end
    obs['drivingDir'] = vDir
    GlobalData = assert(pb.encode("GT.Observation", obs))
end

function netTCP(netChanged, netStatus, port)
    -- this seciton is messy
    local turnOn = (reconnectTry or netChanged)
    if turnOn then
        if netStatus then
            client = Support.File.uvFifo("127.0.0.1", port)
            frames = 0
            reconnectTry = false
        else
            client:close()
        end
        -- main loop
    elseif netStatus then
        local readVal = client:readU32() -- receive a 1 or 2 had a bug till U32 not U16 14.10!
        -- 1 is the main loop for frame capture

        local tmp = readGameState()
        if tmp['raceState'] == 1 then
                setValue(mem, 0x800b6d61, 2, 'int16_t*')
                takeControl = false
        elseif tmp['raceState'] == 2 and not takeControl then
            print("agent has control")
            setValue(mem, 0x800b6d61, 0, 'int16_t*')
            takeControl = true
        end
        if readVal == 1 then
            ready = true
            -- 2 is for loading a savestate
        elseif readVal == 8 + 64 then -- MR2 at Drag
            lapTime = readValue(mem, 0x80093bc8, 'uint32_t*')
            --PCSX.SIO0.slots[1].pads[1].setAnalogMode(false)
```

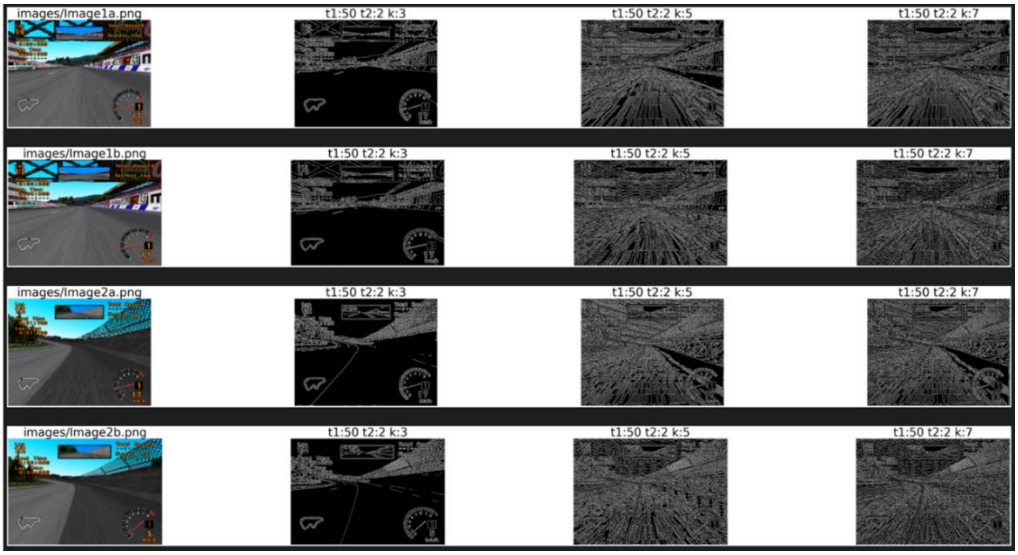
```
            print("lapt_time ", lapTime)
            --local file = Support.File.open("mr2_1_0_0_0.slice", "READ")
            PCSX.loadSaveState(file)
            file:close()
        elseif readVal == 24 + 64 then -- Supra at Drag
            lapTime = readValue(mem, 0x80093bc8, 'uint32_t*')
            --PCSX.SIO0.slots[1].pads[1].setAnalogMode(false)
            print("lapt_time ", lapTime)
            local file = Support.File.open("Sup_1_0_0_0.slice", "READ")
            PCSX.loadSaveState(file)
            file:close()
        elseif readVal == 0 + 64 then -- MR2 at HS
            lapTime = readValue(mem, 0x80093bc8, 'uint32_t*')
            --PCSX.SIO0.slots[1].pads[1].setAnalogMode(false)
            print("lapt_time ", lapTime)
            local file = Support.File.open("mr2_0_0_0_0.slice", "READ")
            PCSX.loadSaveState(file)
            file:close()
        elseif readVal == 0 + 64 + 10 then -- MR2 at HS
            lapTime = readValue(mem, 0x80093bc8, 'uint32_t*')
            PCSX.SIO0.slots[1].pads[1].setAnalogMode(true)
            print("lapt_time ", lapTime)
            local file = Support.File.open("mr2_0_0_0_0_cont.slice", "READ")
            PCSX.loadSaveState(file)
            file:close()
        elseif readVal == 16 + 64 then -- Supra at HS
            lapTime = readValue(mem, 0x80093bc8, 'uint32_t*')
            --PCSX.SIO0.slots[1].pads[1].setAnalogMode(false)
            print("lapt_time ", lapTime)
            local file = Support.File.open("sup_0_0_0_0.slice", "READ")
            PCSX.loadSaveState(file)
            file:close()
        end
        -- keep track of the number of frames rendered
        frames = frames + 1

        -- print("read", ready, "      race state is...", tmp['raceState'], "      take
control...", takeControl)
        -- if this is the nth frame and we have previously received a 1
        if (frames % frames_needed) == 0 and ready and takeControl then
            grabGameData()                -- take screenshot, encode it with protobuf and get
ready to send it
            client:write("P")             -- send "P" for the python server to know we are
ready
            client:writeU32(#GlobalData) -- send the size of the chunk of data
            client:write(GlobalData)      -- send the actual chunk of data
            client:write("D")             -- send "P" for the python server to know we are
ready
        end
    end
end
```

Appendix E – Canny Edge Tests

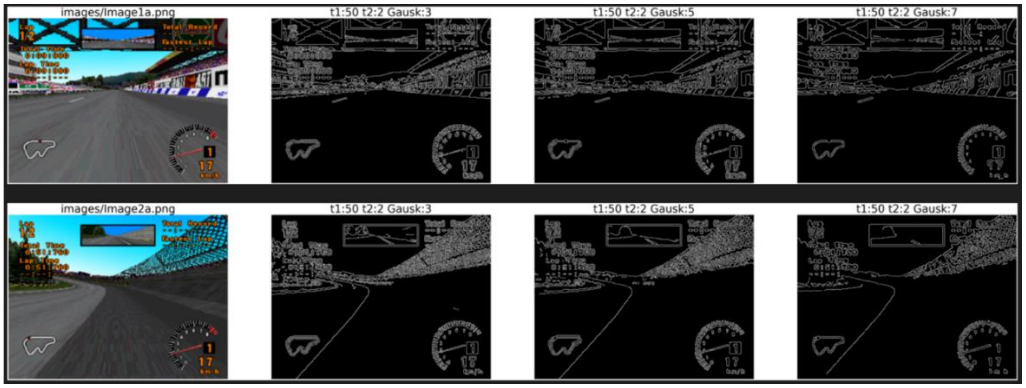
Canny Edge Threshold Sweep



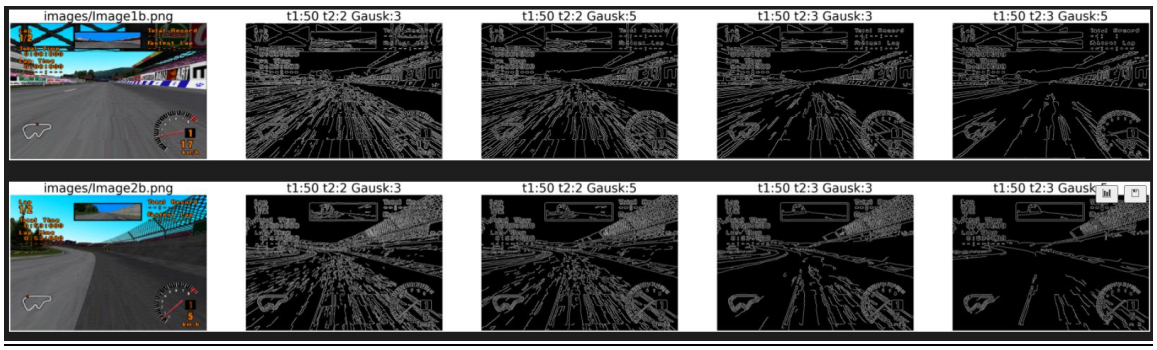
Varying the parameters of the Sobel Operator



Applying Gaussian Blur (Redux only)



Equalisation and Gaussian Blur (Redux only)



Appendix F – Texture Manipulation

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Thanks to the help of spicyjpeg

import numpy, requests
from PIL import Image

image = Image.open("texture.png")
image.load()

# NDR008" Easy swap to palette mode and resize :)
image = image.convert('RGBA').convert('P', colors=3)
image = image.resize((256,256))

if image.mode != "P":
    raise RuntimeError("texture is not in indexed color format")

## Palette conversion
# Analyze the palette's raw size to determine how many entries it contains. (If
# there's a better/more efficient way to get the number of colors in a palette,
# I don't know about it.)
colorSize = { "RGB": 3, "RGBA": 4 }[image.palette.mode]
palette   = image.palette.tobytes()
numColors = len(palette) // colorSize

# Extract the palette into a 2D array; Pillow makes this harder than it should
# be. The array is then converted to 16bit to make sure there's enough room for
# the large values temporarily produced by the 15bpp conversion (see below).
paletteData = numpy \
    .frombuffer(palette, numpy.uint8) \
    .reshape(( numColors, colorSize )) \
    .astype(numpy.uint16)

# For each RGB channel, convert it from 24bpp (0-255) to 15bpp (0-31) by
# adjusting its levels and right-shifting it, then recombine the channels into
# a single 16bpp 2D array which is going to be uploaded to the GPU as a 16xN
# "image".
# https://github.com/stenzek/duckstation/blob/master/src/core/gpu_types.h#L135
red   = ((paletteData[:, 0] * 249) + 1014) >> 11
green = ((paletteData[:, 1] * 249) + 1014) >> 11
blue  = ((paletteData[:, 2] * 249) + 1014) >> 11

paletteData = red | (green << 5) | (blue << 10)
paletteData = paletteData.reshape(( 1, numColors ))

## Texture conversion
# Get the image data as a 2D array of indices into the palette and ensure the
# width of this array is even, as the GPU requires VRAM uploads to be done in
# 16bit units (see upload()).
imageData = numpy.asarray(image, numpy.uint8)

if imageData.shape[1] % 2:
    padding   = numpy.zeros(imageData.shape[0], numpy.uint8)
    imageData = numpy.c_[ imageData, padding ]

# If the texture is 4bpp, pack two pixels into each byte. This is done by
# splitting the array into vertically interlaced odd/even columns and binary
# OR-ing them after relocating the odd columns' values to the upper nibble. As
# this operation halves the width of the image, another alignment check must be
# performed afterwards.
# https://numpy.org/doc/stable/user/basics.indexing.html#other-indexing-options
if numColors <= 16:
    imageData = imageData[:, 0::2] | (imageData[:, 1::2] << 4)

    if imageData.shape[1] % 2:
        padding   = numpy.zeros(imageData.shape[0], numpy.uint8)
        imageData = numpy.c_[ imageData, padding ]
```

```
## Uploading
def upload(x, y, numpyArray):
    requests.post(
        "http://localhost:8080/api/v1/gpu/vram/raw",
        data   = numpyArray.tobytes(),
        params = {
            "x":      str(x),
            "y":      str(y),
            # The width is always in 16bit units.
            "width":  str(numpyArray.shape[1] // 2 * numpyArray.itemsize),
            "height": str(numpyArray.shape[0])
        }
    )

AllFlag = True
if AllFlag:

    imageData2 = imageData
    for i in range(448, 800, 128):
        upload(i, 0, imageData2)
        upload(i, 256, imageData2)
    for i in range(640, 1024-127, 128):
        upload(i, 0, imageData)
        upload(i, 256, imageData)
else:

    for i in range(640, 1024-127, 128):
        upload(i, 0, imageData)
        upload(i, 256, imageData
```

Appendix G – Ethics@Bath Application