TrackMania Reinforcement Learning

A computationally inexpensive reinforcement learning model for a complex real-time environment

by

Tilen Potočnik

Course:	AE4350: Bio-inspired Intelligence and learning for Aerospace Applications
Instructor:	Dr.ir. E. van Kampen
Project Duration:	May, 2023 - August, 2023
Faculty:	Faculty of Aerospace Engineering, Delft

TU Delft

Contents

1 Introduction

Reinforcement learning is often used in racing environments, the simplest and most common of environments being the Car Racing environment from OpenAI[3]. To increase the complexity the obvious answer is to look into racing games. These offer an increase in the complexity of its driving physics, as well as an increase in competitiveness as the games have player bases focused on achieving the fastest times possible. The most notorious example of a reinforcement learning model deployed on a racing game is *Sophy*, a model developed by *Sony* for the racing game *Gran Turismo*. The model is capable of achieving superhuman results, driving faster than professinoal players[2]. The game in question in this paper is the game series *TrackMania*, for which few models have also been developed, one achieving performance close to that of a semi-professional player[1]. What all the aforementioned environments and their models have in common is that they extract data of the game by processing screenshots of the game along with some telemetry data. Processing of images through a convolutional network is computationally intensive and poses a challenge for the wider community to access the capability of these models.

The goal of this report is to assess the feasibility of a computationally inexpensive reinforcement learning model on complex environments. This will be tested by developing a model for the game *TrackMania Stadium*[2], with model inputs being generated from data which can be accessed without any computer vision.

Firstly, the approach to the development of the model is presented in chapter 2. In the chapter the basics of the game and reinforcement learning are presented, followed by a detailed approach to the discretization of data. Furthermore, chapter 3 presents the results of the training, along with a sensitivity and stability analysis. Lastly, chapter 4 concludes on the findings of the report.

[1] See `https://youtu.be/wjHW3ai47Og`

2

Approach

2.1. Environment Description

The environment chosen for this research paper is the *TrackMania*[2]*: Stadium*[1] game. The game is a 2013 installment of the long standing racing game series *TrackMania* by *Nadeo*. It is set in the stadium environment, firstly introduced in *TrackMania Nations*, where the player drives a car similar to that of Formula 1.

On the surface, the game is very simple and beginner friendly, with only three controls available to the player: steer, accelerate, and brake. However, this simplicity means that to achieve the fastest possible time, the player must drive the optimal line, as well as take advantage of the physics engine, or find a shortcut to the finish.

The games' physics engine runs on a fully deterministic model, meaning identical inputs will always produce identical results. This has enabled the existence of scripts that can input commands at precision much higher than humanly possible, giving rise to perfected times on tracks, known as tool assisted speed-runs (TAS). Similarly to TAS, the model in this paper will script inputs to be played out by the game, but rather than being generated and tuned by hand it will be generated through reinforcement learning.

The game series was selected due to the authors personal affection to the series, while the specific version was selected through trial and error, proving to be the most optimal for our application.

Figure 2.1: TrackMania[2]: Stadium (sourced from `store.steampowered.com`)

2.2. Reinforcement Learning

Reinforcement Learning (RL) is a subset of Machine Learning (ML). Its learning process, reinforcement, comes from behavioral psychology, which states that a specimens behavior results from previous rewarding and punishing stimuli. Rephrased, this means that after learning on previous experiences,

[1]Trailer: `https://www.youtube.com/watch?v=P5Qnws7Njas`

when faced a situation, the specimen would try to perform the action which yields the most rewarding outcome.

When translated to a programming sense, the specimen is an actor agent, performing actions in an environment. Based on the decisions and actions made by the agent, the environment returns a corresponding reward. The agent only knows a limited set of variables from the environment, referred to as the observation. Based on the rewards and observation, the agent can learn which actions to perform, with the use of a background framework. The general RL framework is presented in Figure 2.2.

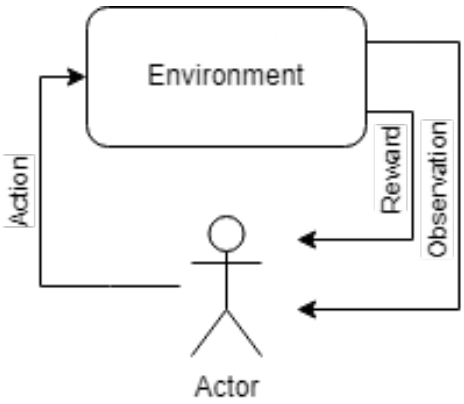


Figure 2.2: Reinforcement learning diagram

2.2.1. Soft Actor Critic

Soft Actor Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way[4]. It was chosen for this application as several existing models for driving already employ it and as it provided the highest performing output. The implementation of SAC used in our model is from the *Stable Baselines* library[2], which is a user friendly *Python* implementation of the algorithm from *OpenAI*. It is a successor to the Twin Delayed DDPG (TD3) algorithm with the main difference being the implementation of the clipped double-Q trick.

The central feature of SAC, which is a key element in the success of our model, is entropy regularization. The policy maximizes a trade-off between expected return and entropy, which is a measure of randomness. This trade-off prevents premature converging of the policy, which was experienced when training with other policies, including base Q-learning, Double Q-learning, as well as TD3.[4]

SAC can be implemented in different ways, where the action space can be continuous or discrete. The implementation which benefited our implementation is using a continuous action space, which means the steering of the car can be fully analogue (the car can steer at any value between fully left or right). However, the gas input was discretised after the action was decided to simplify the controls.

2.3. Implementation of the Model

The following section focuses on the implementation of the model. Firstly the approach to gathering data is described. Further paragraphs describe the discretization approach to simplify a 3D environment into a 2D dataset. Lastly, the observation space and the model rewards are explored.

2.3.1. Gathering Game Data

Commonly the environment is coded or directly accessible by the user, which unfortunately is not the case here. As data is not easily accessible to the user, an alternative approach is taken. Thankfully, the developers have provided a solution to gathering data, by storing the game-state in system memory, which can be read efficiently[5].

The code provided has been adapted from C++ to python, allowing us for direct access to several pieces of data, which can be used in the environment. These include, but not limited to, the following:

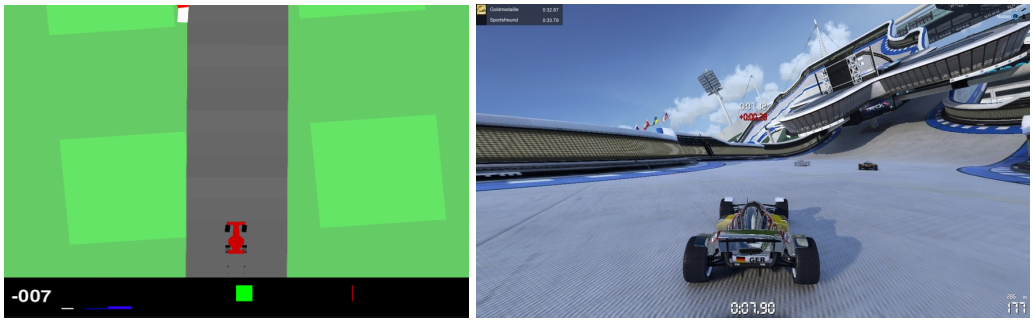
- Game state - State of the game (menu, racing, finished, etc.)
- Race time
- Car position and angle
- In-game velocity

The access to this data allows us to employ similar techniques to existing models. Several pieces of data are still missing for the algorithm to drive successfully. Among the required data is the information on the track that is being driven, along with a LIDAR like environment. As the latter requires the former, the track interpretation will be explored in the following section.

[2] `https://stable-baselines.readthedocs.io/en/master/`

2.3.2. 2D Track Discretization

Compared to a majority of other existing car driving/racing RL models, the *TrackMania* environment is considerably more complex, as can be seen in Figure 2.3. The most obvious change is the fact that most environments are in 2D while *TrackMania* is a 3D game. Some RL algorithms mitigate that by forming their decisions on game screenshots fed through a convolutional neural network (CNN). This however is considerably more resource intensive, making it unsuitable for our goal. Thus a discretization to two dimensions will be made for the game, to allow for low resource computing.



(a) Screenshot of the OpenAI Car Racing environment. **(b)** Screenshot from the *TrackMania* game.

Figure 2.3: Screenshots from competing RL environments.

The tracks in *TrackMania* normally include some sort of verticality, whether it be jumps or raising and lowering of the road. This would introduce several levels of complexity, which for the purpose of this report will be omitted. This led us to create a custom track with no elevation changes to be used for RL. The track features a combination of turns, varying in direction and sharpness, as well as straights. It was kept relatively short to aid with the training time. The track can be seen in Figure 2.4.

The tracks are saved into a *.Gbx* file format, which can be read by the *pygbx* library[3] developed specifically for reading *TrackMania* map files. From the map files one can read the blocks used to build the track, as well as their positions. The blocks used were coded into python as shapes made of a collection of x and y coordinate points. These can then be assembled together, like a jigsaw, to create the 2D outline of the map seen below, with the help of the *shapely* library.

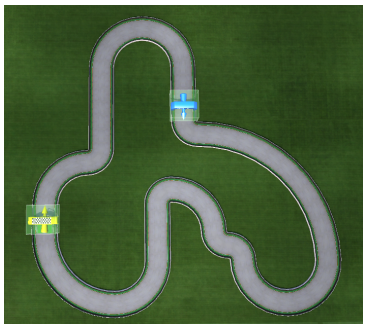


Figure 2.4: The racetrack created with the purpose of training a RL model.

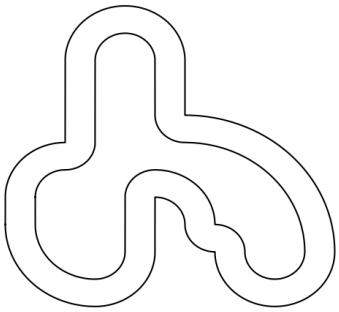


Figure 2.5: Generated 2D outline of the created racetrack.

The script streamlined the process of changing the map, as it is capable of generating the outline of any map. The current limitations are that it only supports a selection of possible blocks and that it can only generate circular tracks. These limitations did not prove to be a limiting factor in our research, however the script can be extended to support more blocks without much issues. The figures below represent the flexibility of the model to generate various tracks.

[3] `https://github.com/donadigo/pygbx`

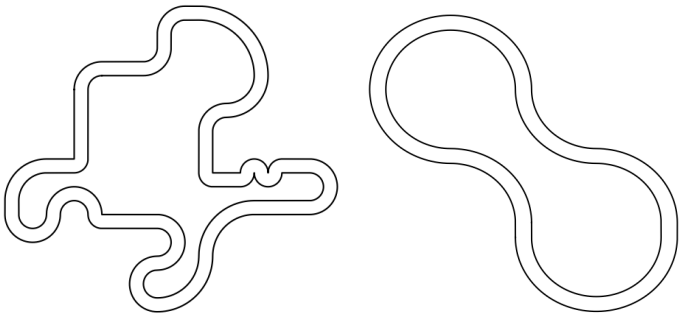


Figure 2.6: Generated 2D outlines of example tracks.

The track path is used in the reward function to reward progress along the track. In existing RL models for *TrackMania*, this path is generated by the user driving the track and storing the xy-coordinates of the car[1]. To generate the optimal track the user should drive as close to the center of the track as they can. An advantage of our approach is that by discretizing the track into a *python* variable, the track centerline can be automatically generated, and further discretized into checkpoints an arbitrary distance apart (further explained in subsection 2.3.4), eliminating the need for user interaction.

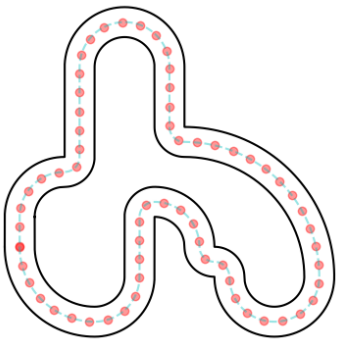


Figure 2.7: Track with generated centerline and checkpoints.

2.3.3. LIDAR and Track Curvature

The LIDAR and track curvature are key components in the observations fed into the RL model. The LIDAR works similarly to what a LIDAR in real life would, it generates beams originating from a point (in this case the car) and returns the distances to objects. This gives the actor a sense of position, as it becomes aware of its distance to track walls.

The LIDAR is generated by drawing lines originating from the car up to a maximum distance, arranged along 180° relative to the car orientation. The lines are split at the intersection point with the track, returning the distance of each of the lines into an array. This process is streamlined with the help of the *shapely* library and its *split* and *length* functions. A representation of the LIDAR can be found below.

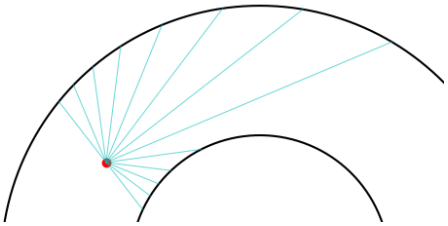


Figure 2.8: Graphical representation of the LIDAR.

The second piece of necessary data is the upcoming track curvature, adapted from a RL model for another game[4]. This gives the model some insight into the future inputs required. The function returns

[4] `https://gitlab.com/schw4rz/nfsmwai/-/tree/main/`

10 points ahead on the track, where the value is the radius of the circumscribed circle of the track at that point. The distance ahead that the points cover varies based on the velocity, the car is going at. This gives a temporally static output, as the point calculated will be reached within the same amount of time no matter the velocity.

This concludes in the final observation array, consisting of the velocity, acceleration, angle relative to the centerline, previous steering input, whether the vehicle is in contact with the wall, and the LIDAR and curvature. The observation fed into the model consists of the current and three previous states. The shape of the observation is presented below.

Table 2.1: Observation array structure.

Observation	Type	Shape
Velocity	int	1
Acceleration	float	1
Angle	float	1
Prev. steer input	float	1
Wall contact	bool	1
LIDAR	float array	13
Curvature	float array	10
Total (incl. previous)		4x28

2.3.4. Reward Function

The reward function is a crucial part of any RL model. Several iterations of a reward function have been tested for this model. The functions can be roughly separated into three components: the velocity, wall impact, and progress component.

The first component rewarded the agent for either going fast (velocity) or for the selected action being to press on gas. This function did prove successful in the early stages, as the car would go through the track fast, however in the later stages of training the reward proved to be a hindrance, as purely going fast is not enough to have a fast finishing time. This component was neglected in the final model, however an implementation which could benefit the model would be to dynamically change the weight of the component through episodes, such that the component lose importance with time, and gives way to the other components.

The second component of the reward function, is the wall impact component. This component imposes a negative reward, as we want to avoid contact with the wall as much as possible. The function reward is a function of the velocity with which the car impacts the wall. This component required some fine tuning with the reward factor. A factor too high would lead to the actor being afraid of even approaching the wall, while one too low led to the actor sometime slamming full speed into the wall flipping the car or escaping the track. The equation is presented below, with i representing the observation boolean, v the velocity, and k the reward scale.

$$R_{impact} = -i \cdot v^2 \cdot k \tag{2.1}$$

The final component, the progress, is the key component of the reward function. The component represents the progress along the track in each time-step. It is calculated by checking the number of previously mentioned checkpoints passed in the time-step. This component is responsible for the algorithm finding the optimal path through the track. The reason the component led to the optimal path is presented in Figure 2.9. Taking the fastest route between checkpoints resulted in the highest reward, which is gained by taking the path which is the optimal trade-off between speed and distance travelled.

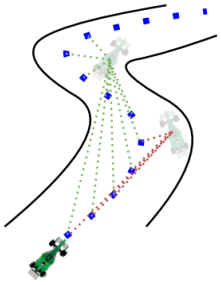


Figure 2.9: Better trajectories yield higher reward. (adapted from [1])

3 Results

This chapter will focus on the outcome of the training of the RL model. Firstly, the best results will be presented, followed by the analysis and the adjustments made to achieve the performance from the initial model, the sensitivity analysis.

3.1. Presentation of Results

TrackMania measures the success on a track with medals, from bronze to gold, as well as with the hidden author medal. The author medal is achieved by beating the time the creator of the map achieved. For the track used in this paper the medal times can be found in the table below. Times for medals gold through bronze can be set by the user, or set automatically by the game. The latter was chosen, therefore all medals but the author medal are automatically generated.

Table 3.1: Training track medal times.

Medal	Time [s]
Bronze	34.00
Silver	28.00
Gold	24.00
Author	22.50

The measure of performance is time to complete a lap. The times achieved by the model are presented below in Figure 3.1. The fastest time the model achieved is 24.22 seconds, which is close to achieving the golden medal and about two seconds or 10% slower than a human. Once stagnated, the model achieved an average time of 27.13 seconds, enough for a silver medal. These times do provide room for improvement, based on the performance that reinforcement learning is capable of achieving[2], but were deemed sufficient given the goal and the time-frame.

The model has been trained in real time on a computer equipped with 16 gigabytes of ram, a RTX 1060 3gb, and a Ryzen 1600 processor. The model was capable of 8 iterations per second resulting in a new input decision every 0.125 seconds. The results presented were achieved within 500000 timesteps, which took approximately 20 hours to compute. This could be sped up by running the game at an increased speed, however the iterations per second were limiting, as the game requires rapid inputs and small adjustments.

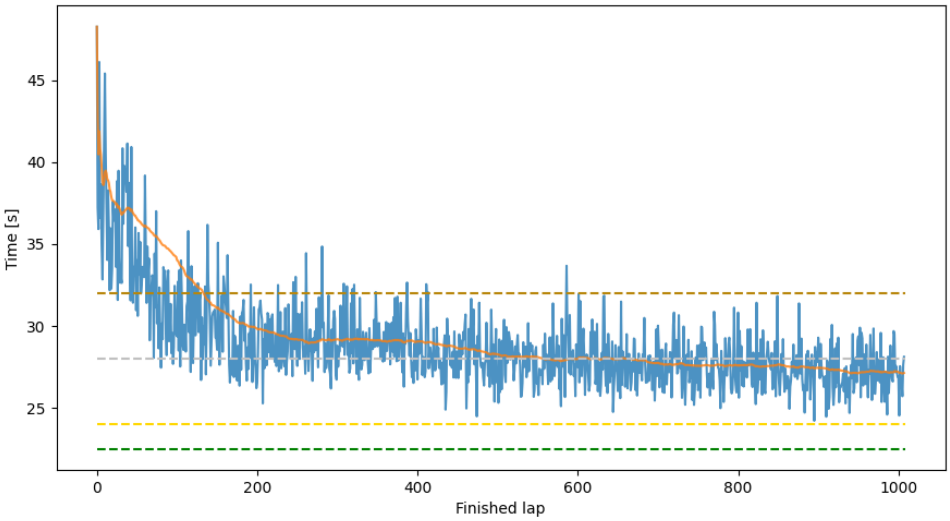


Figure 3.1: Progress of lap times throuhg training.

3.2. Analysis of Results

To compare the results the RL model achieved compared to a human driver, several metrics were logged. These metrics include the position, velocity, and steering input. The comparison is made to the actors fastest lap, as that is considered its peak performance. Firstly, the comparison of the path can be seen in the figure below.

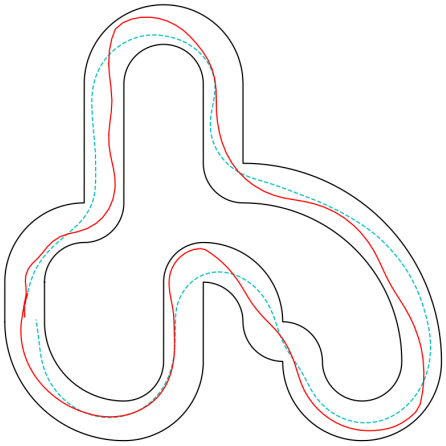


Figure 3.2: Comparison of the path driven by the human driver (cyan) compared to the RL model (red).

From path comparison, one can discern that while turns were driven successfully, the model lacks the precision, and anticipation of upcoming turns that a human does. This is most noticeable in the bottom middle of the track, where several small turns are stringed together, ending in a sharp turn. There the model tries to gather as many points as possible in the turns, but in tow fails to successfully adjust for the coming sharp turn and crashes into the wall. Another point which can be noticed from the comparison is the smoothness of driving. The human driver has turns that could be considered smooth or nearly perfect with little corrections, while the RL model tends to correct its path, which can be seen in the wiggling, when the track is relatively straight.

By comparing the logged velocity, it can be seen that the two are correlated, with the RL model only diverging slightly towards the end of the track, gaining some time loss. It does however manage to retain a higher velocity through the middle of the track. From the velocity, we can see that the time loss of the model is likely more correlated to the worse path taken rather than a lower velocity. The wiggles noticed before, are caused by the model steering fully to the left or right for most of the time, each period of steering also taking longer than that of a human. This could be the cause of the sample

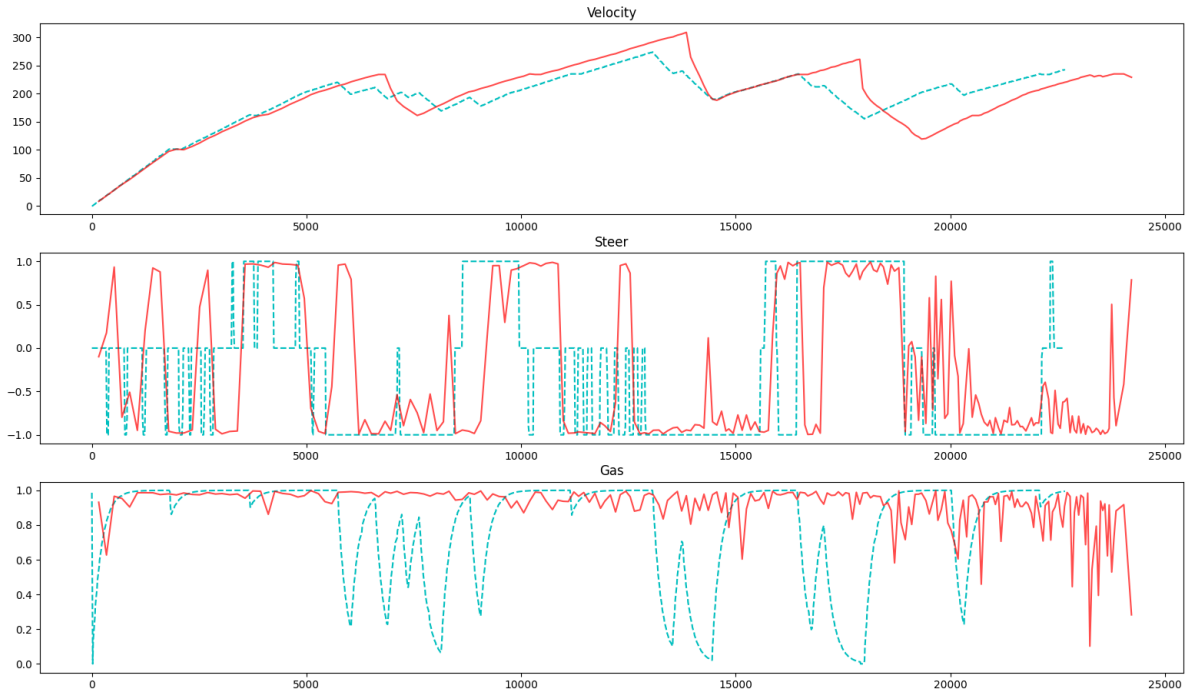


Figure 3.3: Comparison of the logged numbers between the human driver (cyan) ant the RL model (red).

rate of the model, which was on average 8 iterations per second or one input every 0.125 seconds. This is a limit of the hardware that was used for the computations, and is therefore out of our control. From the final plot it can be seen that the model prefers to constantly press on the gas. This results in several crashes into the wall, even with the negative reward for it. The reward scale was left as is even with crashes, as it did result in the fastest times, compared to those model iterations where no crashes were observed, further explored in following sections. This is likely also a quite large time loss, meaning the model could achieve faster times, if it were driving clean lines without crashing.

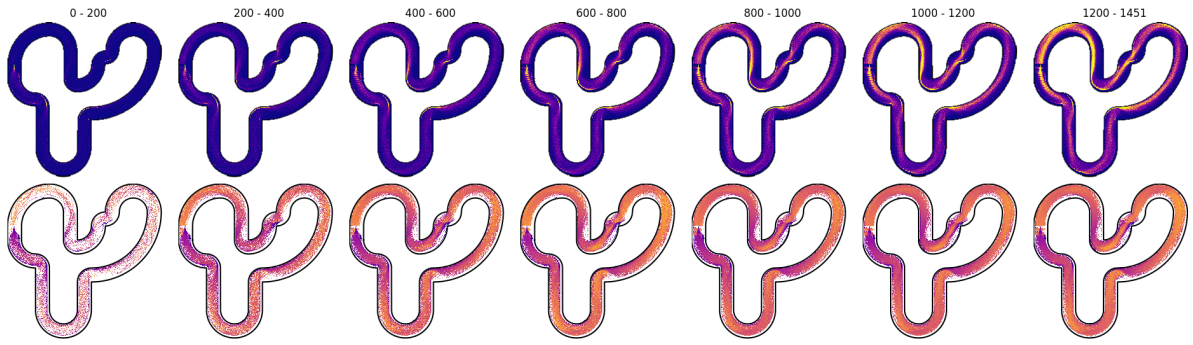


Figure 3.4: Heatmap of the RL evolution. (top: position heatmap, bottom: velocity heatmap)

Figure 3.4 shows how the model evolved through its episodes. Through the early episodes the common points between the episodes are mostly seen in the impact points on the walls. These points represent the difficult corners of the track, whether the corners are sharp or the car enters the turn with a high velocity, which can be seen in the bottom row, showing the velocities before impact are high, compared to the rest of the track. These impact points are common points even when continuing the training. The position heatmap does show that the runs between episodes tend to become more and more similar to each other towards the end of the track. This is likely a result of the reward discount factor γ. Similar behavior has been observed in various RL racing implementations and is tackled in different ways. Our result was achieved through tweaking the discount factor, to the final value of 0.85, which gradually decreases the importance of each future reward. Another approach which has proven success and should be explored in the future is by using a custom reward discount with a time-based future lookout. This would mean the rewards for x seconds in the future would have a reward weight

of 1 while the rest would have the weight of 0[1].

3.3. Sensitivity Analysis and Consistency

Several iterations of the algorithm were created, including variations on the environment, control scheme, rewards structure and policy structures, finally settling on one using SAC as the best performer. Only the latest versions will be explored in this section, where the final structure has been decided and several parameters were being tuned to achieve best performance. The initial tests were all performed with the same seed, to isolate random elements from the performance, while later on, to test the consistency of the algorithm, a random seed was used. The table below presents the various iterations, along with their parameters and fastest lap time.

Table 3.2: Parameters and results of various model iterations.

Iteration	RL Parameters					Reward Structure			Observation	Lap Time [s]
	Batch Size	Learning Rate	Discount Factor	Buffer size	SDE	Points Travelled	Collision Factor	Extra		
1	64	0.0003	0.99	50000	x	✓	x	x	Reduced	DNF
2	64	0.0003	0.85	50000	x	✓	x	x	Full	36.32
3	256	0.05-0.01[2]	0.85	50000	✓	✓	5e-3	Normalised Velocity Squared	Full	32.67
4	1024	0.05-0.01[2]	0.85	50000	✓	✓	1e-2	Normalised Velocity Squared	Full	DNF
5	256	0.0003	0.85	∞	x	✓(Squared)	x	x	Full, 4 Prev	DNF
6	256	0.0003	0.85	∞	x	✓	5e-3	x	Full, 4 Prev	25.51
7	256	0.0003	0.85	∞	x	✓	5e-3	Forward Action Squared	Full, 4 Prev	32.01
8[3]	256	0.0003	0.85	∞	x	✓	5e-3	x	Full, 4 Prev	25.39
9	256	0.0003	0.85	∞	x	✓	x	x	Full, 4 Prev	**24.22**
10	256	0.0003	0.7	∞	x	✓	x	x	Full, 4 Prev	32.98

The above table presents the iterations and hyper-parameter tuning performed on the model. The iteration numbering is purely for clarity and does not reflect the actual number of iterations tested, which is considerably higher, even when excluding other algorithm architectures tested. The table does not include some results as the performance was quickly deemed unfit and the training was stopped. The results which were not included in the table include ones with the discount factor of 0.99 and those with a learning rate higher than 0.0003. Furthermore, when the fastest model was found (iteration 9) it was tested for the performance stability. Several runs were repeated with a new random seed each time. The results converged to nearly the same point, thus the model is deemed stable and not a result of a lucky seed. The results of the stability analysis can be found in the figure below.

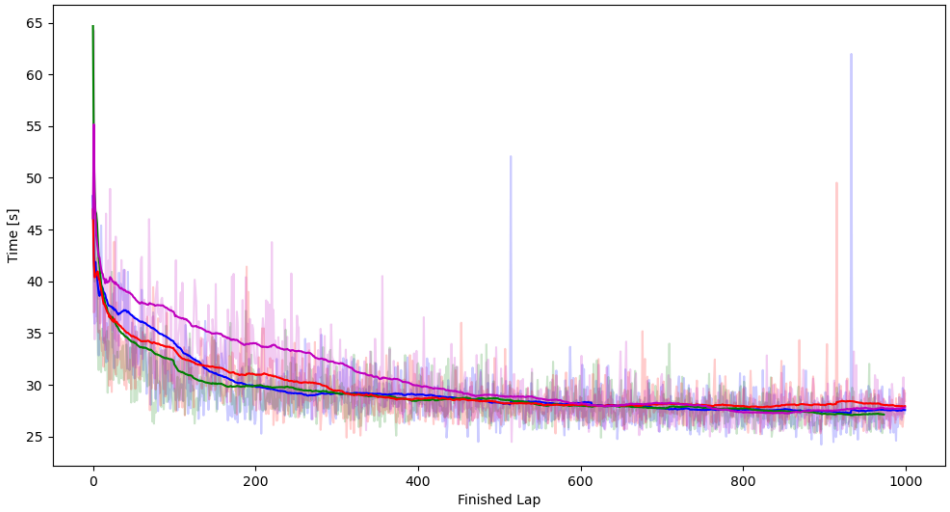


Figure 3.5: Performance of the model with several random seeds

[1]Through correspondence with the authors of `https://github.com/Agade09/trackmania_rl_public`
[2]Stepwise learning rate.
[3]Additional control input, model is capable of braking.

4

Conclusion

The goal of this report was to assess the feasibility of a computationally inexpensive reinforcement learning model on complex environments. This was achieved by developing a model for the game *TrackMania Stadium*[2], with model inputs being generated from data which can be accessed without any computer vision. The model discretized the data from the game into a simple 2D environment, for quick calculations. With the use of automatic 2D track generation, the model is capable of being adapted for any track, with no user effort required. Through learning, the model was nearly capable of achieving a gold medal with its best time and an average time fast enough for a silver medal. While this reports model does not achieve superhuman performance as some other models do, it has proven that a computationally inexpensive model is capable of achieving fast results. The comparison is as well unfair to this model as others were developed in a much longer time span of a year or more compared a limited timeframe allocated to this project.

We believe that with some improvements this approach is capable of achieving times faster than the author medal. The recommendations for improvements are as follows. Firstly, the implementation of a finite horizon reward discount would likely make the driving performance more consistent throughout the entire track. Furthermore, tuning the track curvature look ahead could give the model a better idea of which actions to take to achieve the highest rewards. Lastly, the model should avoid hitting the walls, this could be done by altering the reward functions, or enabling the model more complex control of the vehicle. The final point would be a major improvement to the model, as the fastest times in *TrackMania* rely on movements such as drifting, as well as exploiting the games' physics engine with actions found by the community, allowing for time cuts.

In conclusion, the model achieved its goal of proving the feasibility of a low complexity model for a complex environment. The performance it was capable of achieving is satisfactory, however several improvements can be made to it to increase its competitiveness with other models.

References

[1] Yann Bouteiller. “Deep Reinforcement Learning in Real-Time Environments”. PhD thesis. Polytechnique Montréal, 2021.

[2] Florian Fuchs et al. “Super-human performance in gran turismo sport using deep reinforcement learning”. In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 4257–4264.

[3] OpenAI. *Car Racing*. 2022. URL: `https://gymnasium.farama.org/environments/box2d/car_racing/`.

[4] OpenAI. *Soft Actor-Critic*. 2018. URL: `https://spinningup.openai.com/en/latest/algorithms/sac.html`.

[5] Mania Tech Wiki. *Telemetry interface*. 2022. URL: `https://wiki.xaseco.org/wiki/Telemetry_interface`.