POLITECNICO DI TORINO

Master's Degree in
Computer Engineering

Master's Degree Thesis

AI Copycats: Imitation Learning for Driving Style Modeling with Unity ML-Agents

Supervisors
Prof. Francesco Strada
Prof. Andrea Bottino

Candidate
Matteo Colucci

July 2024

Contents

1 Introduction 3
1.1 Background . . . 3
1.1.1 Reinforcement Learning . . . 5
1.1.2 Neural Networks . . . 6
1.1.3 Latency in Multiplayer Videogames . . . 10
1.1.4 MPAI-SPG . . . 10
1.2 State of the Art . . . 11

2 Methods 13
2.1 Training . . . 13
2.1.1 Reinforcement Learning . . . 14
2.1.2 Imitation Learning . . . 16
2.1.3 Network Parameters . . . 16
2.1.4 Execution . . . 17
2.2 Evaluation . . . 19
2.2.1 Trajectory Similarity . . . 19
2.2.2 Execution . . . 20
2.3 Data Gathering Application . . . 22
2.3.1 Tracks . . . 22
2.3.2 Online Services . . . 30
2.4 Unity Editor . . . 31
2.4.1 Training and Evaluation . . . 31
2.4.2 Miscellaneous . . . 38
2.5 Experiment Setup . . . 39
2.5.1 Preliminary Lap Number Search . . . 39
2.5.2 User Data Analysis . . . 40

3 Results 41
3.1 Optimal Lap Number Search . . . 41
3.2 Final User Test . . . 44

4 Conclusion 47
4.1 Future Works . . . 47
4.2 Final Words . . . 48

Abstract

Latency mitigation (or compensation) is one of the main concerns when developing online applications that rely on real-time interaction between users, as latency constraints for highly dynamic applications (e.g., competitive First-Person Shooters or racing games) are very strict.

In a server-authoritative setting — i.e., a client-server configuration in which the server has the final say on any performed action — Artificial Intelligence (AI) enables a new solution for reducing the round-trip time of packets to and from users experiencing high latency.

This thesis' contribution, in collaboration with the MPAI (Moving Picture, audio and data coding by Artificial Intelligence) Community, is an application of an Imitation Learning approach to a custom-made kart racing videogame, with the purpose of showing a possible implementation of the SPG (Server-based Predictive Multiplayer Gaming) specification for the steps that concern a single user (namely data gathering, model training and evaluation).

Imitation Learning is widely used in conjunction with Reinforcement Learning to train robotic agents. In contrast with a fully Reinforcement-Learning-based approach, demonstrations from a human "expert" are provided to the agent to take example from, which kick-start the following autonomous learning phase typical of Reinforcement Learning.

For behavior modeling, however, Imitation Learning is used exclusively, training on users' in-game performances, with no following Reinforcement Learning.
The resulting models, according to the MPAI-SPG specification, are then employed by the server in order to temporarily take control of a user's vehicle if they were to incur in high latency spikes.
After a server intervention episode, a faithful model would allow for minimal reconciliation, and an ideally seamless experience for all other players.

In addition to the aforementioned karting videogame, an in-editor framework was developed to aid in automating training and testing, and to ease interaction with the data-hosting server.

Chapter 1

Introduction

Online videogames represent an ever growing share of the global videogame market, itself rapidly growing and expected to reach a total revenue of over $200 billion by the end of 2024 [1][2].

One of the main hurdles to overcome when developing a real-time multiplayer game is correctly synchronizing all players. Latency, or "lag", plays a role in any form of communication, but for highly dynamic videogames especially, great care is taken in implementing methods that mitigate discrepancies between player experiences and, at the same time, try not provide an advantage to any of the parties involved.

Many latency compensation methods were engineered throughout the decades [3], but MPAI, a standard-developing non-profit organization, proposes SPG: a specification for a novel approach to this task.
Server-based Predictive Multiplayer Gaming (SPG) mitigates latency by predicting the actions of a player, so as not to wait for their communication: this is obtained by means of intelligent agents, which are capable of controlling player avatars on the users' behalf. These agents were trained in order to behave as their respective player, and are issued by the server when connection responsiveness drops under a certain threshold.

This thesis showcases work done to gather user data and create personalized AI models by means of Imitation Learning: a Machine Learning approach that instructs an agent on how to perform a task by imitating a set of provided demonstrations.
Both an example karting videogame and a set of utilities were created thanks to the Unity game engine. Subsequently a small field test was conducted to assess the performance of this solution.

1.1 Background

Machine Learning (ML) is a broad term that refers to any computer algorithm that allows decisions to be taken based on previous experience, and to improve at a task by increasing the amount or quality of provided experience.

ML tasks are usually tasks for which a closed-form formula or algorithm does not exist: these can range from simple spam e-mail filters, to advanced recommendation systems or

real-time decision-taking strategies for autonomous cars.

Most ML tasks can be classified in two great categories:

- Unsupervised tasks;
- Supervised tasks;

Unsupervised tasks generally involve identifying hidden common patterns in data — most typical examples being *clustering* and *data generation*.
These tasks do not possess an inherently "correct" answer, frequently relying on relationships between elements to organize them.

Most tasks, however, fall under the *Supervised* category.
For these tasks, the algorithm needs some degree of external intervention, usually in the form of correct examples from which to learn.
For image classification problems, for example, these examples could be a set of images joined with a textual description of their contents (see Fig. 1.1).

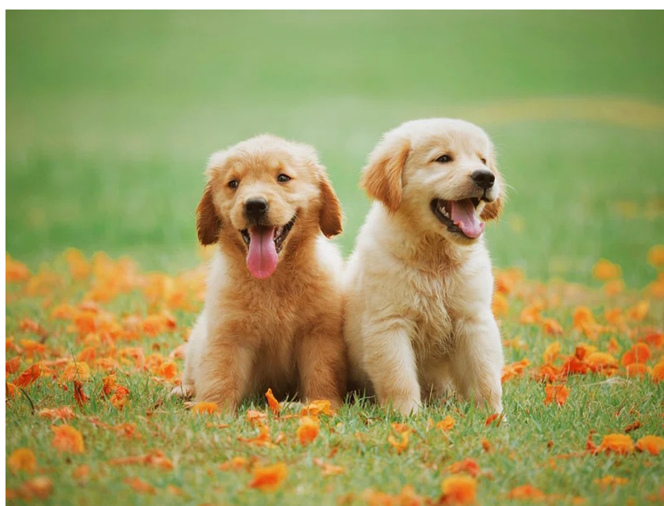


Figure 1.1: A possible description for this training image could be: "Two puppies sitting on a grassy field with orange flowers in the foreground and background".

The process of optimizing an algorithm's parameters for a certain task by processing examples is referred to as *training*.
It opposes to *testing*, i.e. the actual deployment of the algorithm to its target usage scenario, which — if training was successful — just consists in using the optimized parameters.

Yet, some supervised tasks are much too complex to be learned through examples alone, or perhaps they rely on styles or behaviors more than single answers.

1.1.1 Reinforcement Learning

Reinforcement Learning (RL) is a Machine Learning paradigm that builds on the concept of reinforcement. Due to this distinctive feature, it could be treated as a third macro-category other than Supervised and Unsupervised Learning.

Just like animals or children, a RL agent learns by doing: if an action leads to a positive or desirable outcome, the agent learns that it is good, and will tend to perform it again; otherwise the action is thought as bad, and the agent will be more reluctant to take that choice again in the future.

This approach is different than the usual "Question and Answer" style of Supervised Training, as the objective of the agent is to maximize its long-term reward rather than to provide an exact result.

In order to train for a task, a RL agent necessitates three elements:

- Observations;
- Actions;
- Rewards.

Observations allow the agent to sense the environment. They can be as simple as a few numbers or as complex as high-resolution images.

Observations are processed to form the *State*, the internal representation of the agent's present (and eventually past) condition.

Actions are the agent's decisions, based on previous observations, rewards and its current state.

Rewards are the environment's response to the agent's actions.
The agent's internal logic keeps track of the reward that was assigned for every action and every state.

Rewards are perhaps the most important aspect of Reinforcement Learning, as it is instrumental that the agent get rewarded for exactly what it should be doing and not for something it should not.
It seems obvious, but sometimes, correctly identifying for what actions and how much to reward the agent in order to obtain desirable behaviours is not a trivial task: e.g. rewarding a racing agent for driving fast is not a guarantee of good performance if it is not penalized for going backwards.
At the same time, if a task is very hard or emits rewards only upon specific series of actions, it can be beneficial to add some small "utility" rewards, in order to give the agent more feedback on its actions.

This process of devising optimal rewards is known as *reward shaping*, and varies greatly depending on the task.

Imitation Learning

Imitation Learning (IL) is a variation on the Reinforcement Learning paradigm, where the reward that the agent receives does not depend on the absolute quality of its actions, but rather on their similarity to some actions provided as reference.

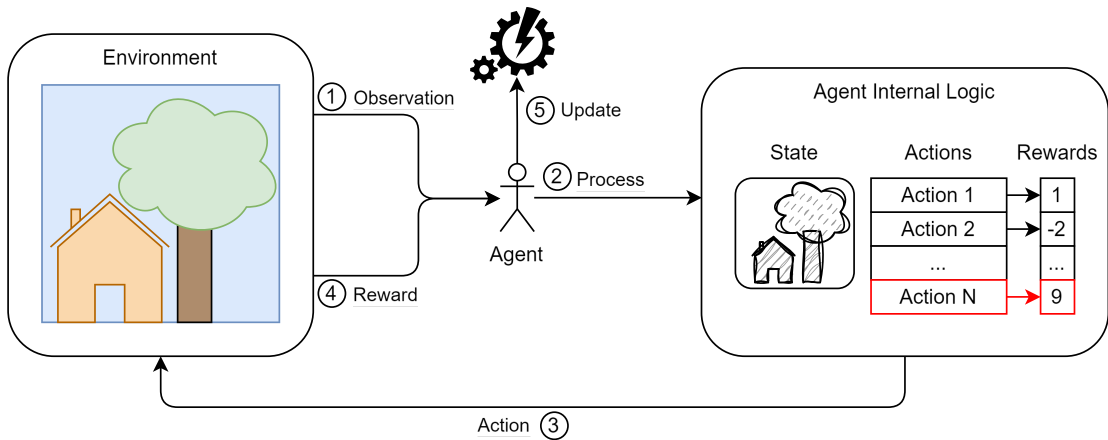


Figure 1.2: A schematic representation of RL. ① The environment provides the agent with an observation; ② The agent computes the current state; ③ The user performs the best action according to its internal logic; ④ The environment rewards the agent based on the action; ⑤ The agent updates its internal representation.

IL often co-exists with RL, as the two paradigms can be applied successively, or even at the same time — if the two rewards are appropriately balanced.
Specifically, IL can be used to kick-start RL: by providing successful examples performed by an *expert* (e.g., a human), by the time RL starts, the agent can already be capable of executing the task, leaving the RL step as a phase to further refine its actions.

1.1.2 Neural Networks

RL is a general paradigm that can be implemented in many ways: from simple state–action dictionaries to complex situation-aware models.

A Neural Network is a computational model inspired by biological neuron structures. As such, it consists of many simple computational units that can be arranged in complex formations to solve the most disparate tasks.
Each unit, called a *neuron*, receives inputs and can produce outputs depending on the value of its inputs and their *weights*, and its internal *activation function*.

Each task is characterized by a *loss function*: a function that codes a "penalty" for each mistake a network can make when emitting a result.

In order to find an optimal solution to a task, a neural network first produces a result, which gets fed to the loss function, producing a *loss*. Loss is minimized by traversing the network in reverse, changing the weights of its parameters based on their contribution to the loss: this is known as *back-propagation*[1].
This process gets repeated until the loss reaches some desired value, or an arbitrary amount of time has passed.

[1]Loss minimization is an example of *Empirical Risk Minimization*

Being neural networks a branch of Machine Learning, the ML definition of *training* applies to this process.

Not all parameters pertain to the network, however: for example, parameters regarding the training process itself, such as the overall duration or the intensity of the weight adjustment during back-propagation (known as *learning rate*), are fixed during a single training, but can nevertheless benefit from being optimized.

These parameters are referred to as *hyperparameters*, and their optimization is crucial for the performance of the resulting network — often called *model*.

Notable Layouts

Like biological neuronal structures, shapes and connections play a fundamental role in the function and performance of a neural network.

An ordered grouping of neurons is called a *layer*, and is usually treated as the smallest unit of a network.

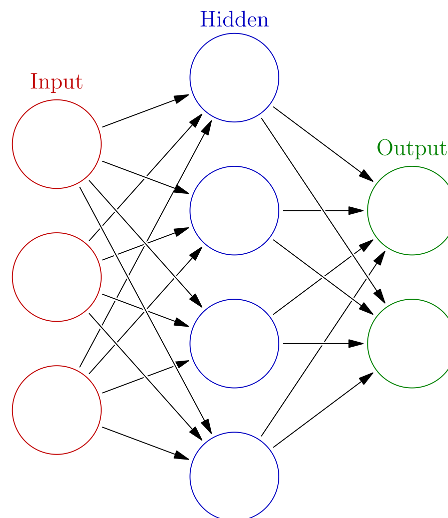


Figure 1.3: Diagram portraying a neural network with one input, one output and one intermediate (*hidden*) layer.

If each neuron of a layer has a connection to all neurons of the following layer, the layer is called *fully connected* (such as in Fig. 1.3).
Due to the presence of multiple layers, neural-network-based variants of most algorithms are often called *deep*.

Multi-Layer Perceptron Fully connected layers are the foundational elements of the *Multi-Layer Perceptron* (MLP), one of the simplest types of neural network. Fig. 1.3 shows the smallest possible MLP, with only one hidden layer.
MLPs are very flexible, and can prove a valid solution for many ML tasks — especially ones that feature limited possible answers, such as classification.
However, the high number of connections introduces some issues when increasing the size

and complexity of the network: for example, they are not optimized for large inputs, such as images.

Convolutional Neural Network Another extremely popular network layout is the *Convolutional Neural Network* (CNN).
In these networks, neurons are arranged to perform matrix-focused operations such as discrete convolution and sampling.
For this reason, this architecture is predominantly employed by tasks based on images and videos.

Convolutional layers are not fully connected, as the operation aggregates data, reducing its dimension, so that significant features can more easily be abstracted.
CNNs present fully connected layers only towards the output, where features are computed and formatted according to the network's specific use case (e.g assigned probability for classification tasks, coordinates for object detection tasks, etc.).

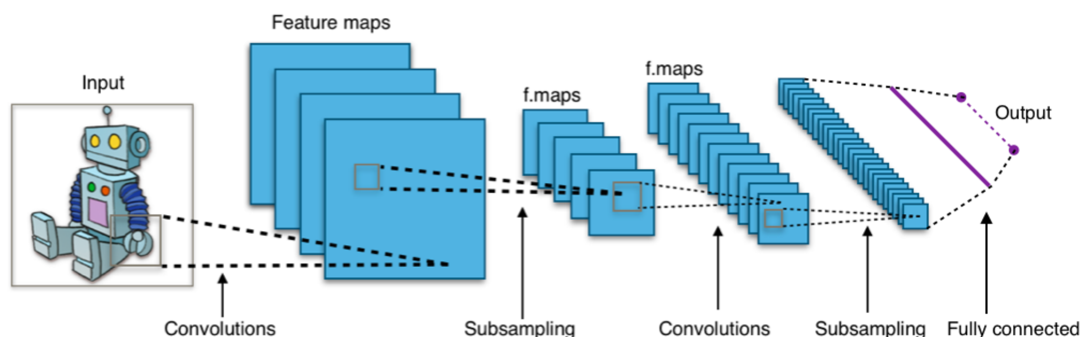


Figure 1.4: Diagram of a typical CNN architecture[4]: the input image is down-sampled repeatedly, corresponding to features of decreasing sizes being extracted progressively from the original data.

Recurrent Neural Network A *Recurrent Neural Network* (RNN) is a kind of neural network that features loops among its connections.
Unlike previously described architectures, which are typically *feed-forward* (i.e., data travels unidirectionally from the input to the output), some layer outputs are connected to previous layers' inputs.
This distinctive feature makes them inherently fit for tasks that are repeating in structure but varying in contents, like time series analysis or language modeling.

Especially noteworthy are Long Short Term Memory (LSTM) Networks, which feature a few different types of neurons that can be trained to discriminate what to remember.

Generative Adversarial Networks

Generative Adversarial Networks (GANs) extend the concepts of neural networks by using two networks at the same time and pitting them against each other as opponents of a 1-on-1 competition.

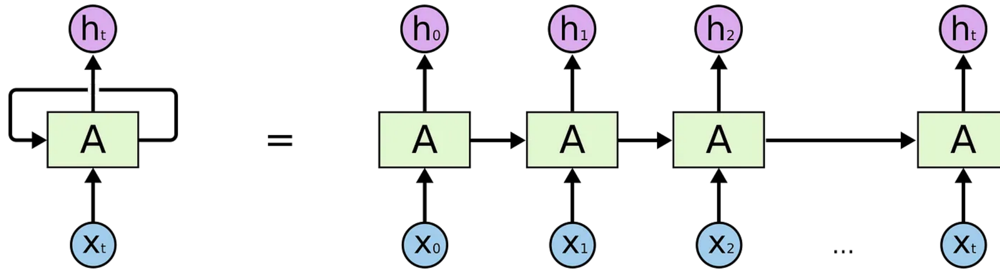


Figure 1.5: Diagram of a typical RNN architecture[5]. Both the input x_t and the output h_t are dependant on time, whereas the internals A keep a state that persists through iterations. The network on the right is a time-unrolled representation of the one on the left.

The *generator*'s objective is to produce content that is as similar as possible to a certain original set of data (e.g. portraits, cat pictures). This generated (also called *synthetic*) data is sent to the *discriminator*, which tries to ascertain whether it was produced by the generator or sampled from the original data.
This continuous battle between the generator, trying to fool the discriminator, and the discriminator, trying to see through the generator's deception, leads to an independent — albeit tied — improvement of both networks.
Once a certain level of proficiency has been reached by the generator, it can be detached and used on its own to create new content.

Training data is only used to train the discriminator, as the generator is trained based on its performance against the discriminator. Moreover — in contrast with usual training procedures — training data need not be labeled, since the objective of the discriminator is to learn defining data distributions, similarly to unsupervised tasks.
For this reason, GANs are usually categorized in yet another macro-category: *Self-Supervised* Learning.

Although they are primarily used for image generation, GANs can work with any kind of data with an identifiable distribution (provided the network architecture supports it). In this thesis, for example, a GAN will be employed to create a model that produces user actions for a videogame.

Generative Adversarial Imitation Learning

Generative Adversarial Imitation Learning (GAIL) applies the concepts of Imitation Learning to Generative Adversarial Networks [6].
Given that the aim of the generator in a GAN architecture is to mimic provided data, it is inherently a great fit for the concept of imitation.
In particular, the generator produces user actions, whereas the discriminator tries to determine whether the actions came from the generator or from the reference data.

With this approach, good performance and generalization can be obtained with relatively little reference, proving more flexible and data-efficient than other methods like Behavioral Cloning or Inverse Reinforcement Learning.

1.1.3 Latency in Multiplayer Videogames

Latency is an unavoidable aspect of all communications, online or not, due to the non-infinite transfer speed of any medium.

When speaking of videogames, the term latency (or *lag*) can refer to many, often unrelated, sources of delay: peripherals might operate at too low of a frequency, the CPU or GPU might not be performant enough for the task they have been assigned to, or communication with other users might be taking too long.
All these cases can have the effect of introducing a delay to the user's actions, which usually is what gets noticed [7]. However, from now on, the term "latency" will be referring exclusively to connection latency, often informally called *ping*.

Given the highly interactive nature of the medium, latency plays a crucial role in user experience and overall game feel [8][9].
For this reason, over the decades, numerous lag compensation techniques were developed [3], detailing a very fragmented landscape with great variety among solutions.

1.1.4 MPAI-SPG

MPAI[2] (Moving Picture, Audio and Data Coding by Artificial Intelligence) is an international, non-profit organization that aims to improve efficiency of data transfers through compression and AI.
MPAI-SPG[3] (Server-based Predictive Multiplayer Gaming) is a project with the objective of developing a standard for addressing client latency by means of server-side predictions.

Firstly, it is required that the communication be server-authoritative: i.e., all client actions must first get validated by the server before being registered and relayed to other users.
Secondly, the server has to possess an AI model for each connected user (for example, sent by the user when connecting to the server).

Given these prerequisites, the server is able to substitute any player experiencing connection issues and take actions on their behalf until the episode is over.
If the model was properly trained, other participating players would not notice any discontinuity in the actions of the substituted player, greatly benefiting their experience.

This framework can also double as an anti-cheating supervisor: by letting the prediction engine running continuously (instead of enabling it only situationally), every user action can be compared to its model's choice. If the two are found to be too different, it means that cheating is afoot and user input is discarded.

[2] `https://mpai.community/`

[3] `https://mpai.community/standards/mpai-spg/`

1.2 State of the Art

Like most of Machine Learning, Imitation Learning has seen significant advancements in the last three decades [10][11]. Nowadays, IL algorithms have been applied to very complex tasks, mainly in the field of robotics, to great success [12][13].

However, employing IL algorithms to control videogame characters is not a recent concept either: first applications date back to 2002 [14], although they were mostly Machine-Learning-based solutions.
Few years later, Neural Networks started gaining more traction, and with them Deep Reinforcement Learning. Using Neural Networks became the standard for training RL models, and similarly IL.

There are several studies investigating the usage of IL to move player characters in FPS (First Person Shooter) games [15][16], 2D side-scrolling games [17][18][19], and racing games [20][21]. However, given the great strides that were made in the field of autonomous driving in recent years, much of recent research related to Imitation Learning applied to driving concerns real life or realistic driving simulations [22].

Moreover, one crucial aspect that all mentioned examples share, is the usage of IL as a means to obtain optimal performance. All demonstrations are expected to be provided by "experts", i.e. real humans that are good at performing the task at hand, from which the agent can learn by imitation.
In fact, one remark that was frequently noted, is that the model's performance highly depends from the quality of reference demonstrations.
For some, the optimum is a generic, user-like behavior, for others the lowest possible completion time or the least number of mistakes. In any case, this approach is fundamentally different from the goal of this thesis, which is to model the driving style of a single user as faithfully as possible, no matter how optimal their demonstrations are. In this regard, literature is quite lacking.

Another notable difference in approach between this thesis and previously mentioned studies, is the presence of game engine data. Most of the mentioned models are image-based, which implies that the agent is being developed by a third party who has no access to internal game data. This is not the case for this work, as the development of a user modeling framework is expected to benefit any game developer who wishes to implement user predictions as part of their own game's multiplayer functionalities.

Ultimately, this thesis builds on the same concepts that were explored by Spina et al. [23], although it differs from their work in various ways: this project was started from scratch, and most features were implemented differently (namely, the prediction network and the training workflow).

Chapter 2

Methods

In order to train an Imitation Learning model, there need to be some reference demonstrations to imitate. They are collected via the Data Gathering Application: a browser game composed of various levels of increasing difficulty.
Agent observations, user actions and RL rewards are recorded for every lap and uploaded to a server for storage.

These files are subsequently retrieved by an import utility, then, training and evaluation runs are started and managed by the Training Session editor script, which allows automating an otherwise long and repetitive series of manual actions.

All aspects related to learning are controlled by the ML-Agents library's Python script, running in parallel with the Unity scene. The resulting model can then be evaluated against unseen trajectories from the same user it was trained upon.

After the user application and the training/testing framework were established, a field test was conducted to measure the performance of the final models when deployed in a realistic usage scenario.

2.1 Training

At the inception of the project — in order to gauge feasibility and performance — training was performed with a regular Reinforcement Learning setup.
Checkpoints were added to all tracks and reward conditions were put in place to allow for an optimal driving logic to emerge autonomously.

Given the good results obtained from the resulting model, building on this foundation, Reinforcement Learning was then substituted with Imitation Learning.
Many settings were retained, however — most notably — rewards went unused: since Generative Adversarial Networks come with their own Objective Function to maximize[1], task-specific rewards, as defined for Reinforcement Learning, cannot be used for model training and optimization.

[1]As will be explained in Section 2.2, this will complicate following evaluations.

2.1.1 Reinforcement Learning

As explained in Section 1.1.1, in order to learn, a Reinforcement Learning agent needs observations and rewards.

Observations

The agent's in-game vehicle is equipped with two sets of distance sensors called `3D Ray Perception Sensors`. Not unlike LIDAR sensors of real self-driving cars, these sensors radially shoot a multitude of rays that are able to measure whether they hit something, and if they did, at what distance from their starting point.

The two sets of sensors can identify, respectively, walls and other karts. It was necessary to implement them as separate sensors, since enabling sensitivity to both types of objects would have made it impossible to distinguish which of the two types was blocking the beams.

These sensors provide most of the agent's observations, as every sensor accounts for multiple rays (in the final configuration, 11 for walls and 9 for karts, as shown in Figure 2.1a and 2.1b respectively).

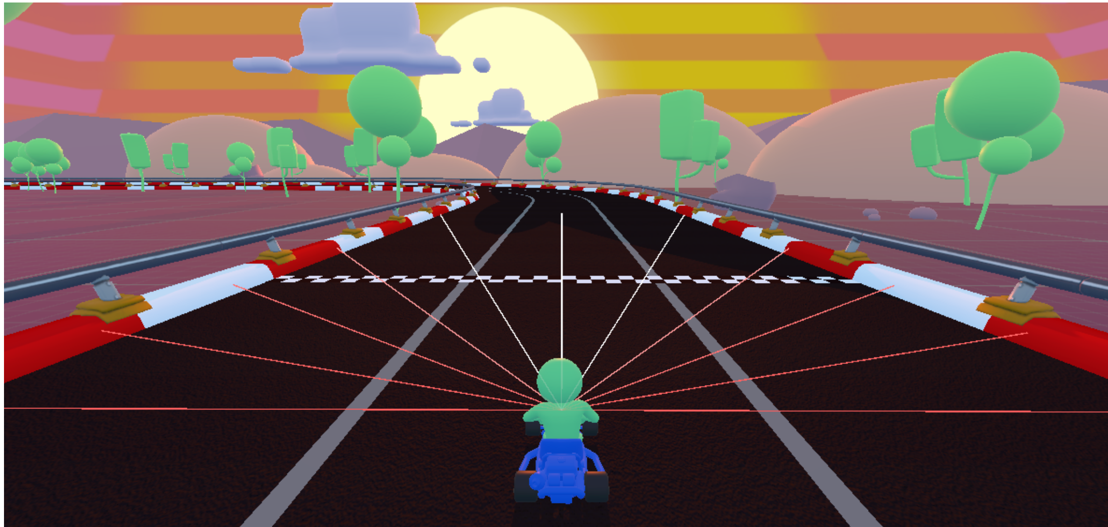
To ensure a better understanding of the task by the agent, in addition to the distance sensors, another four elements (of the equivalent size of nine floating point numbers) were selected to be added to the observations:

1. Normalized kart local speed (`float`);
2. Kart absolute rotation (`Quaternion`, 4 `floats`);
3. Normalized direction to next checkpoint (`Vector3`, 3 `floats`);
4. Dot product between kart's front and next checkpoint's front (`float`).

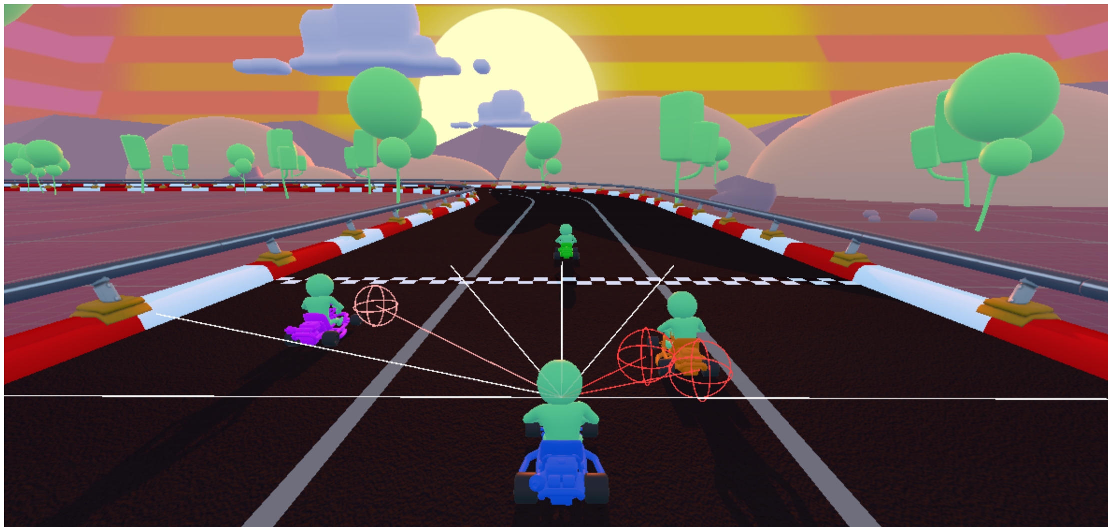
Rewards

Despite rewards being normally assigned synchronously (^{s}) after every agent decision — as with sensing, usually every `FixedUpdate` (20 ms) — most rewards are assigned asynchronously (^{a}), as their related events happen:

- Move towards correct checkpoint^s: $+0.03 \cdot \vec{v_{kart}} \cdot (\overrightarrow{pos_{checkpoint} - pos_{kart}})$;
- Pass correct checkpoint^a: +1;
- Pass incorrect checkpoint^a: -1;
- Hit walls^a: -1;
- Hit other karts^a: -0.5;
- Continued wall hit^a: End episode.



(a) Wall perception sensors. The color of a ray indicates the distance to the obstacle, red is closest, white is farthest.



(b) Kart perception sensors. These rays integrate a sphere cast for a denser sensing perimeter. In this image the green kart is too far, so it does not get picked up by the sensor.

Figure 2.1

Ending an episode is roughly equivalent to a high penalty, since it does not allow the agent to gain any more rewards.
Comparable results can be obtained by waiting for the end of an episode (10'000 steps), but in general prematurely resetting the agent upon a very bad choice can yield speedups.

Heuristics

The goal of the Heuristic function is to substitute the model's decision making: i.e., take actions bypassing normal decision logic.

This function is reserved for any logic that is external to the agent's experience, such as a custom decision function or a random policy, usually to be used if a trained model is not available.
However, for what concerns the ML-Agents library, the most common — and officially encouraged — way to use this function is to perform input processing in it.

Doing so fills the *Decision* Array with user input, but to all other aspects of the Reinforcement Learning environment (namely rewards), it is as if the decisions were taken by the agent itself.

When a `DemonstrationRecorder` component is present, Observations, Heuristics and Rewards are recorded into a demonstration file. Demonstrations are foundational elements for Imitation Learning.

2.1.2 Imitation Learning

When provided with demonstrations, a Reinforcement Learning agent can be enabled for Imitation Learning, either through Behavioral Cloning (BC) or Generative Adversarial Imitation Learning (GAIL).

Behavioral Cloning BC works similarly to regular supervised learning: the expert demonstrations provided by the user are used to learn possible situations and what actions to perform when they happen.

However, the greatest downside of this approach is that the resulting agent lacks generalization capabilities, being effectively limited to already encountered scenarios.
In other words, adequate performance can be reached only through a high amount of training data.

Generative Adversarial Imitation Learning GAIL harnesses the power of Generative Adversarial Networks (GANs) to generalize behaviors.
Thanks to the typical architecture of GANs, after the training phase it is possible to keep using the generator to take decisions — even for unseen situations.

This is particularly useful given the application scenario: since the purpose of the model is to substitute a player under any circumstance, but at the same time keep as consistent as possible of a behavior to their recordings.

2.1.3 Network Parameters

Both for RL and IL, the underlying Neural Network was mostly left untouched from its default settings.
The basic structure of the network is a MLP with two hidden layers of 128 neurons each, however, by specifying the *"Memory"* parameter, it is supported by a RNN.
Observations are first processed by the RNN, which selectively emits features that it

believes to be useful; this extra data is appended to the regular input, which flows through the MLP as normal.

General Hyperparameters	
Batch Size	1024
Buffer Size	10240
Learning Rate	$2 \cdot 10^{-3}$, Linear
PPO Hyperparameters	
Beta	$5 \cdot 10^{-3}$, Constant
Epsilon	0.2, Linear
Lambda	0.95
Network Settings	
Hidden Units	128
Hidden Layers	2
Memory Sequence Length	64
Memory Size	256

Table 2.1: An overview of the main hyperparameters and their values. These values were used in all settings.

GAIL Hyperparameters	
Strength	1
Gamma	0.99
Learning Rate	$2 \cdot 10^{-3}$
Use Actions	True

Table 2.2: GAIL-specific hyperparameters. "Use Actions" specifies that the agent should try to imitate actions, other than states.

2.1.4 Execution

Aside from model settings, hyperparameters and training options, almost all of the training logic is managed by the ML-Agents library, which communicates with its associated Unity package.

When training is started, all agents are spawned at random checkpoints along the track: this allows for a thorough exploration of the track and for a simple way to introduce collisions between agents.

Once instanced, the agents perform actions according to their policy, be it Reinforcement Learning or Imitation Learning.
All the agents share a single brain, so their experiences are compounded and processed jointly: each agent instance can thus access all the others' experience while retaining full decision-making abilities.

When the defined number of timesteps has expired, training is stopped and a file for the final model is created and moved to a separate folder.

2.2 Evaluation

Evaluating a model is as important as training it, as it allows for objective measurement of training progress (or lack thereof).
Moreover, suitable metrics allow for better supervision of the training task, and offer unbiased guidance when tuning hyperparameters.

In this context, the objective of a model evaluation task is to measure how similarly a model, trained on demonstrations from a certain user, behaves with respect to to the user it was trained on: an AI model and a user can be considered "similar" if they both take similar actions when subjected to the same stimuli.

Several metrics, including some rewards and losses, come included out-of-the-box with the training framework. However, being this a task that is highly integrated with Unity — for which a performance estimate is not easily obtainable during training — it is instrumental to integrate the available general metrics with an application-aware evaluation criterion.

In fact, the abundance of application-specific features that need to be taken into account for a model evaluation rendered necessary the development of a custom evaluation environment (2.4.1), which, unfortunately, could not be fully integrated with the Python training framework .

2.2.1 Trajectory Similarity

The result of a model evaluation is obtained by measuring similarity between a set of trajectories generated by the trained model and a set of ground truth trajectories that were generated by the user.
To ensure equal stimuli, the model has to be tested on the same track on which the user recorded the reference trajectories. This is not strictly enforced by the evaluation framework — which supports testing on a different track — however it was a prerequisite that was established for all evaluations.

The metric that was used to measure the similarity between any two trajectories is a simple variation of the Normalized Lockstep Euclidean Distance metric:

$$D_{T_1,T_2} = \frac{\sum_i (T_{1,i} - T_{2,i})}{Length(T)}, \tag{2.1}$$

where T_1 and T_2 are two trajectories (of the same length) and i iterates over the length of the trajectories.

Trajectory Similarity is thus defined as:

$$S_{T_1,T_2} = 1 - clamp_{[0,1]}(D_{T_1,T_2}), \tag{2.2}$$

where $clamp_{[0,1]}(f)$ is defined as:

$$\begin{cases} 0 & f < 0 \\ f & 0 \leq f \leq 1 \\ 1 & f > 1 \end{cases}$$

With respect to other well-known metrics [24], Lockstep Euclidean Distance does take time into account, meaning that identical trajectories at different times will be considered as different.
This is because of the strong connection that *should* exist between a real trajectory recorded by the player and a trajectory generated by their personalized AI model. In order for another player (or spectator) to have a suitable experience, the predicted position has to be in the same place *and* the same time of the ground truth.

Figure 2.2: The green trajectory follows the same path as the red one, but was obtained by driving slower. Metric-wise, it is equivalent to the blue trajectory.

Despite being able to process rotation data — being it both saved in the replay file and generated by the AI model — it was chosen not to take rotations into account for trajectory similarity calculations. Since position is the main focus of the project, and being this a heavily physically-constrained setting, rotations can be considered as consequences of positions.

2.2.2 Execution

Two types of evaluation were analyzed:

1. On unseen laps;
2. On unseen tracks;

Both methods extract training and test data from user demonstrations — being the only source of "labeled data" — however what differs between them is the criterion according to which demonstrations were selected to be training data or test data.

Unseen Laps

The model is trained on demonstrations from all tracks, however only some laps are used. The remaining laps are kept as test data.

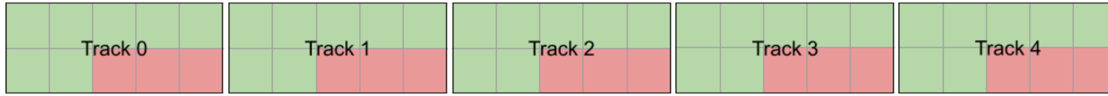


Figure 2.3: In this example, out of the ten laps, the first seven are used for training and the last three for testing.

This configuration allows gauging how much the model is able to understand of the driving style of a user, on a specific track, by processing demonstrations from that user, on that specific track.

Unseen Tracks

The model is trained on all laps, but not all tracks.

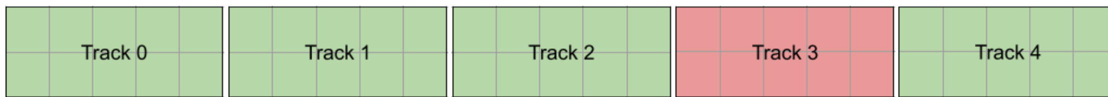


Figure 2.4: In this example, out of the five tracks, four are used for training and one for testing.

This configuration assesses how well the model infers an user's driving style on an unseen track from that user's driving data on different tracks.

This scenario is more akin to the theoretical use case if integrated in a server application, where — after the initial modeling phase — the model would have to generalize learned behaviors to new and unseen tracks.

2.3 Data Gathering Application

This is the application that was presented to the participants to the data collection phase. After an initial prototype made from scratch to test out the training systems, it was chosen to build the app upon the Karting Microgame[2] template: an example project provided by Unity that implements solid driving mechanics but is highly customizable in many aspects.
On top of these valuable features, it is also built for out-of-the-box compatibility with the ML-Agents library.

Time was spent investigating ways to create and export tracks made in Blender, however, due to the less than ideal workflow and unsatisfactory results, it was chosen to opt for a simpler, more scalable solution: building blocks.

2.3.1 Tracks

The recorded portion of the application consists of five tracks of increasing difficulty, which were entirely assembled using assets from the official "Racetrack" addon for the Karting Microgame[3].
The 16 modular pieces strike a good balance between interesting shapes (banked straights and turns, sloped hairpins) and ease of traversal (constant width, 90° corners).
Moreover, being the track pieces from an official source related to the base project, they share aesthetics to some extent, granting greater visual cohesion to the entire application.

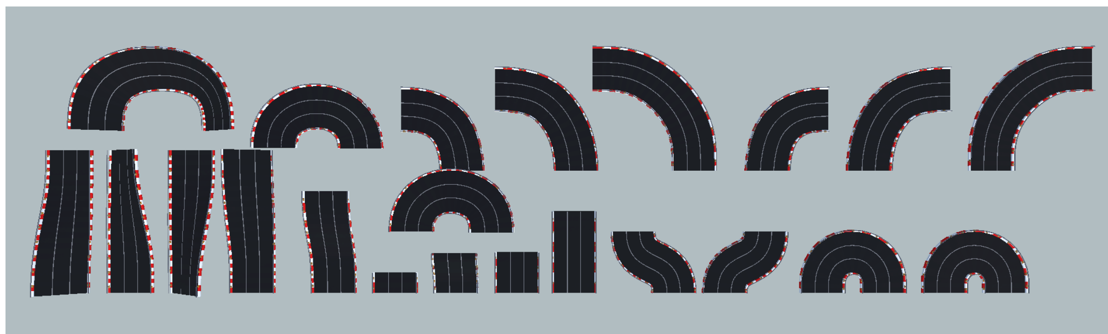


Figure 2.5: All the pieces that were used to build the tracks (including mirrored variants).

Considering that the scope of this project is to analyze a player's unique driving style, be it advanced or inexperienced, it is important that the tracks do not distract or take away focus from the driving itself, even at the cost of lower enjoyment from the participants. For this reason, all tracks are devoid of any power-ups or jumps (which would have been rather fitting for the context), only incentive being an inconsequential lap timer.

[2]https://learn.unity.com/project/karting-template

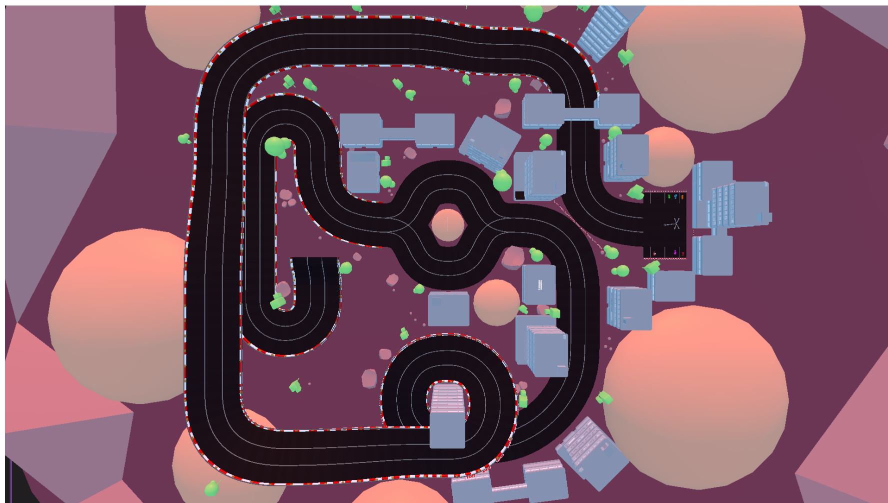
[3]https://assetstore.unity.com/packages/3d/racetrack-karting-microgame-add-ons-174459

By using a small set of fixed pieces, it was possible to manually add checkpoints for every block.
As explained in Section 2.1.1, checkpoint positions are part of the agent's observations, so it was necessary to add them to all track pieces.

Introduction

The application introduces controls and driving mechanics through an introductory, free-roam level, which is not timed nor recorded, and only serves as a warm up for the levels to come.

The track layout is simple, but at the same time offers a good variety of track blocks for the user to get acquainted with.
The scene is fully explorable and does not impose any time limit, so users can stay and experiment as long as they please.



(a) An aerial view of the play area.

(b) A view of the starting spot parking lot.

Square

The first recorded level is a simple square shape. This regular and lenient layout does not pose a real challenge, but it introduces the format of the following, more complex tracks.

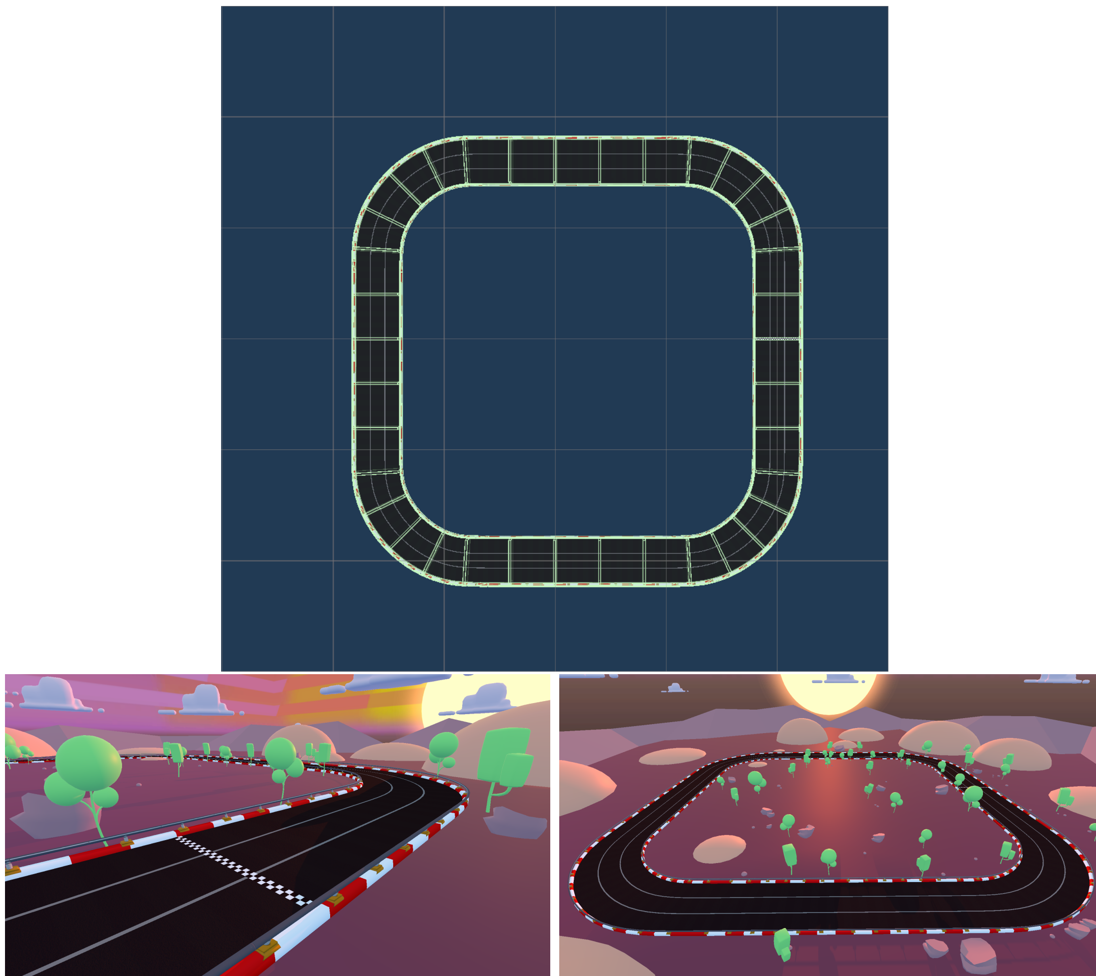


Figure 2.7: Track 0 – Square

8

The second recorded level is a variation on the first level in the shape of a figure eight: half of the track is elevated but the overall experience is very similar to the first track.

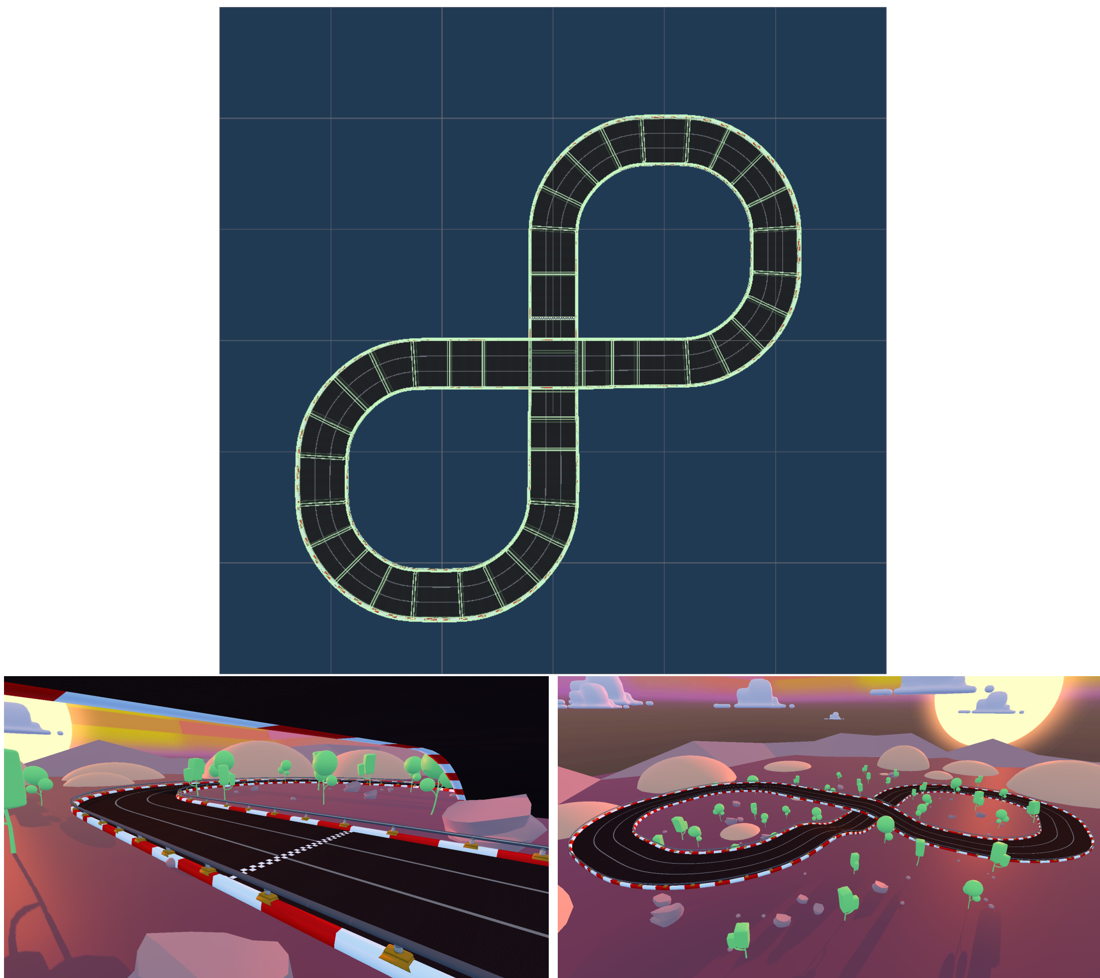


Figure 2.8: Track 1 – 8

Monza

The third recorded level was inspired by the famous Monza Formula 1 circuit.
It is still quite linear, but it features a couple of chicanes and a banked 180° corner.

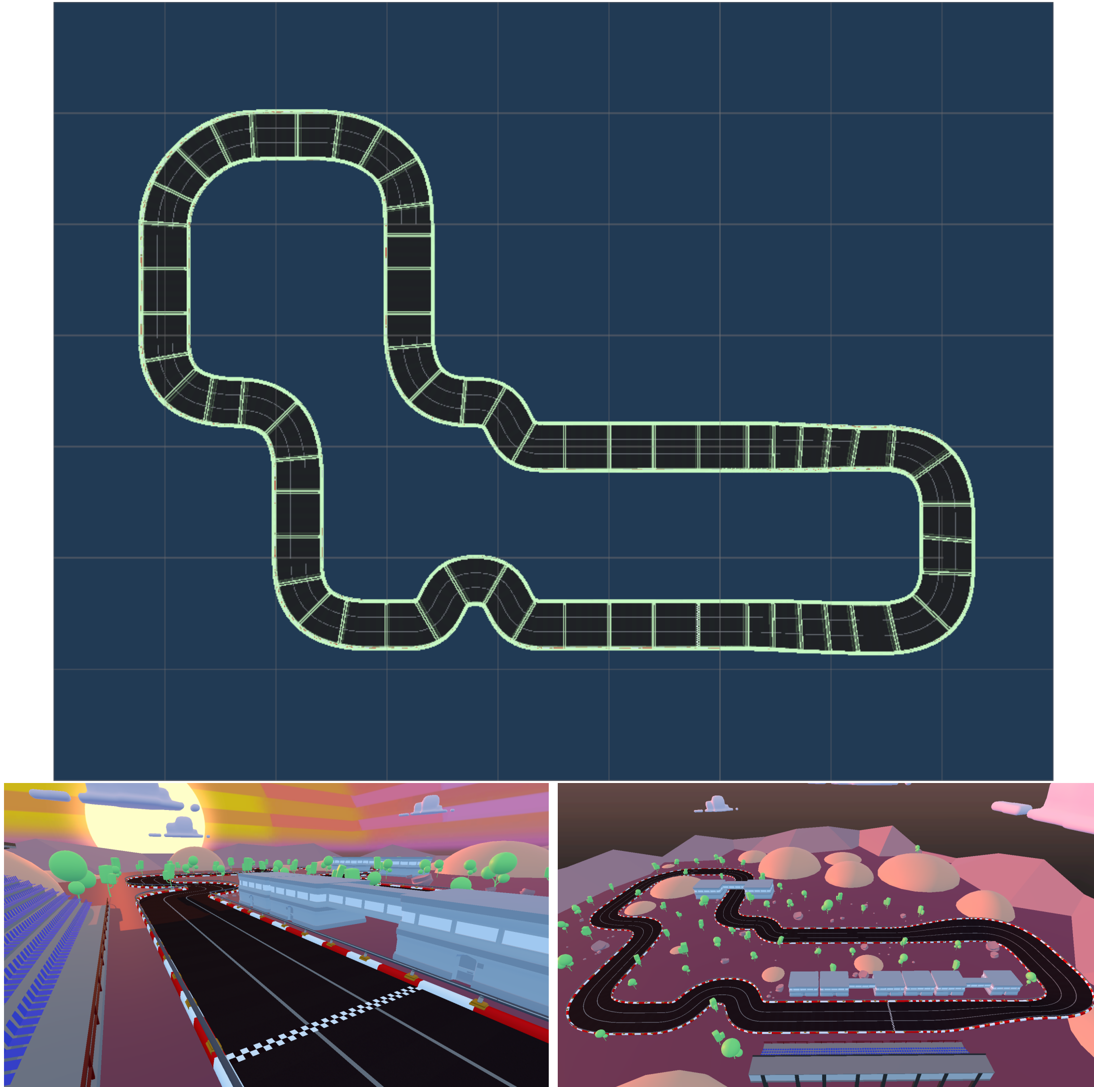


Figure 2.9: Track 2 – Monza

Spa

The third recorded level was inspired by the equally famous Spa-Francorchamps Formula 1 track.
This track is much more challenging than the previous ones, as it features many corners and sharp turns, especially towards the end.

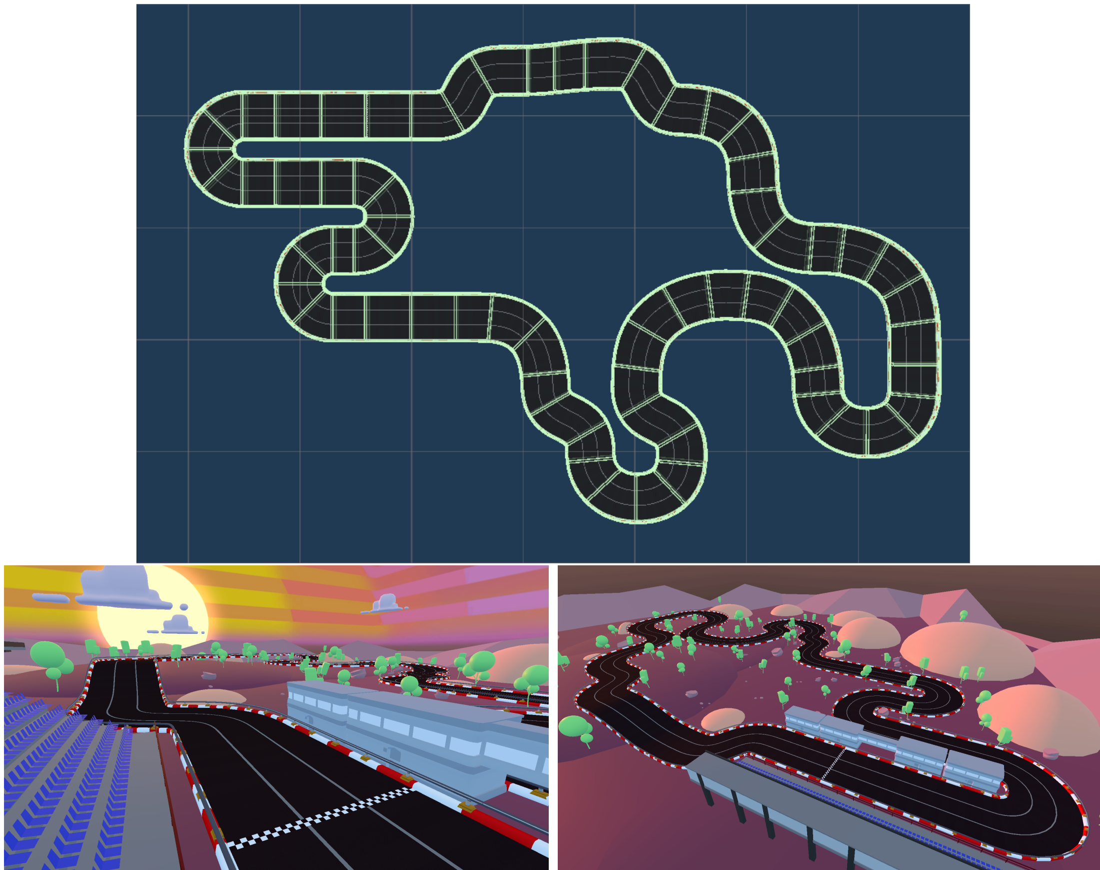


Figure 2.10: Track 3 – Spa

Twisty

The fifth and last recorded circuit is the hardest out of all the tracks.
It was made specifically to use all the types of track building blocks, and features many sharp turns, elevation changes and banked sections.

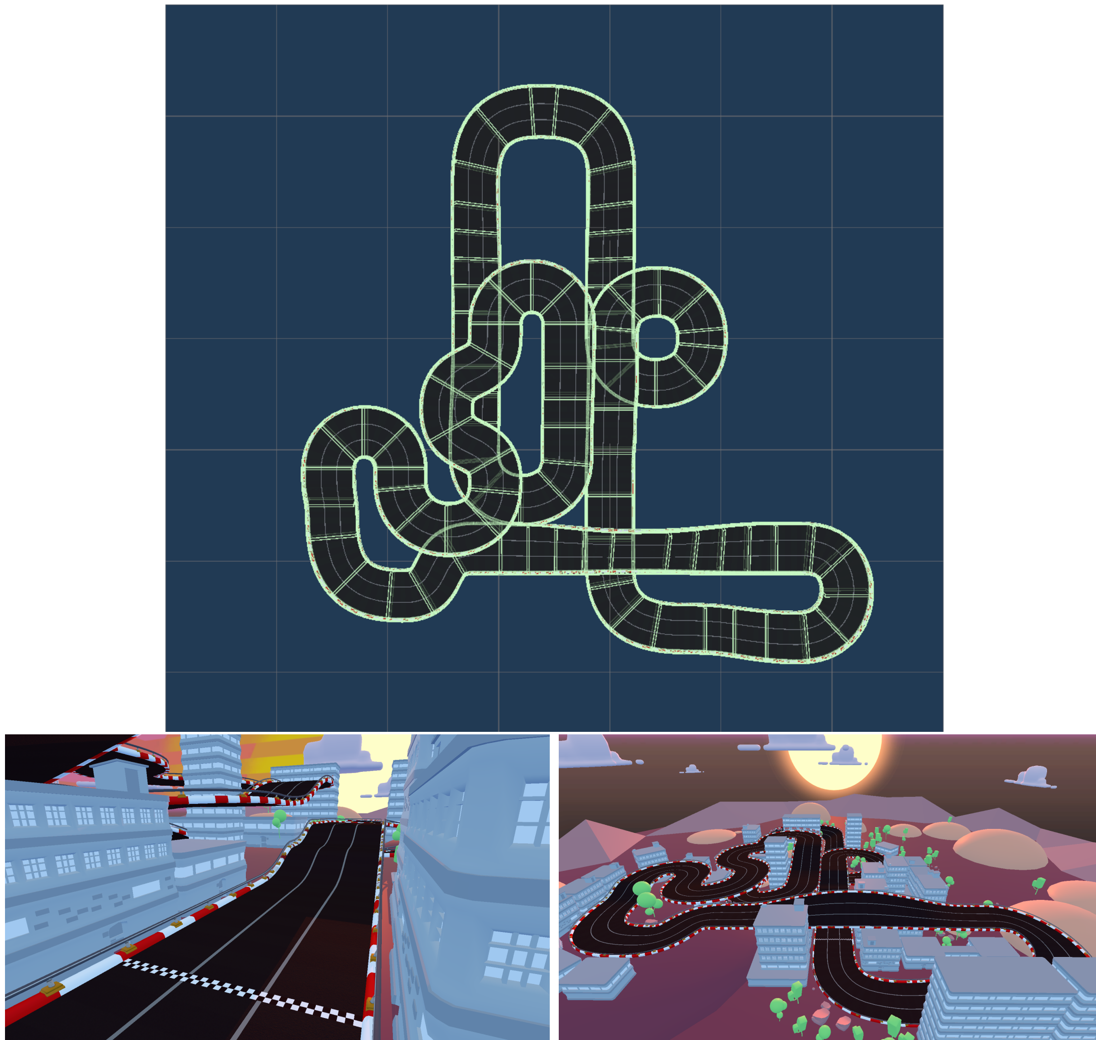


Figure 2.11: Track 4 – Twisty

2.3.2 Online Services

The final application was built for the WebGL platform and published via Unity Play[4], an online platform for sharing web-based games.

This approach was chosen over more popular download-based solutions mainly for ease of access by participants, but also to allow a more uniform experience among them.

When crossing the finish line, all user data produced during the lap gets collected and sent to the storage server provided by Firebase[5], a Google service for hosting web and mobile applications.
Despite Google providing an Unity implementation of the Firebase APIs, due to lack of support for WebGL, it was chosen to access the web server via simple HTTP requests.

[4] https://play.unity.com/

[5] https://firebase.google.com/

2.4 Unity Editor

A majority of development time was allotted to coding Editor Scripts to allow automating training and testing procedures.

During development it is fair to say that the training script was called at least a few thousand times: Editor Scripts might not be the flashiest out of all the elements of the project, but they were certainly the set of features that most (positively) impacted overall development time and iteration speed.

Editor Scripts Unlike regular Unity `MonoBehavior` Scripts, Editor Scripts exist independently from `GameObjects` or `Components`, since they live in a window of their own in the Editor.
Like `MonoBehaviors`, however, they possess a life cycle with many of the same functions (namely `Awake`, `OnEnable` and `Update`) and can access Editor variants of most of `Play Mode`-only classes and methods.

These features make Editor Scripts ideal for supervising tasks that require entering and exiting `Play Mode` multiple times, such as training and evaluating, or in general for tasks that interact with the OS at a deeper level than a regular Unity Application, for example reading and writing files in arbitrary locations, or launching a detached executable that runs in parallel with the application.

Conda Virtual Environments are an invaluable tool for managing different Python projects, since every one requires different libraries, different versions of the same library, or different versions of Python itself.

Creating a Virtual Environment for each project greatly aids in troubleshooting package version issues (which are very insidious by themselves) and ensures nothing gets altered on projects that are not being worked on.

The Virtual Environment tool that was employed is Anaconda[6] (more specifically a lighter distribution called "Miniconda", *Conda* for short).

Every time the `Trainer` Python script is called, it has to be called from the Anaconda Terminal. This is very easy to do as a human, but significantly harder to automate.
This will be a recurring theme for the next section.

2.4.1 Training and Evaluation

`TrainingSessionEditor` is the most important Editor Script, both in size and relevance, as it manages saving, loading and execution of Training and Evaluation procedures.

It establishes the concept of a *Training Session*, which is comprised of an arbitrary number of steps, each of which can be either a Training Step or an Evaluation Step (Fig. 2.12-5.1).

[6] `https://www.anaconda.com/`

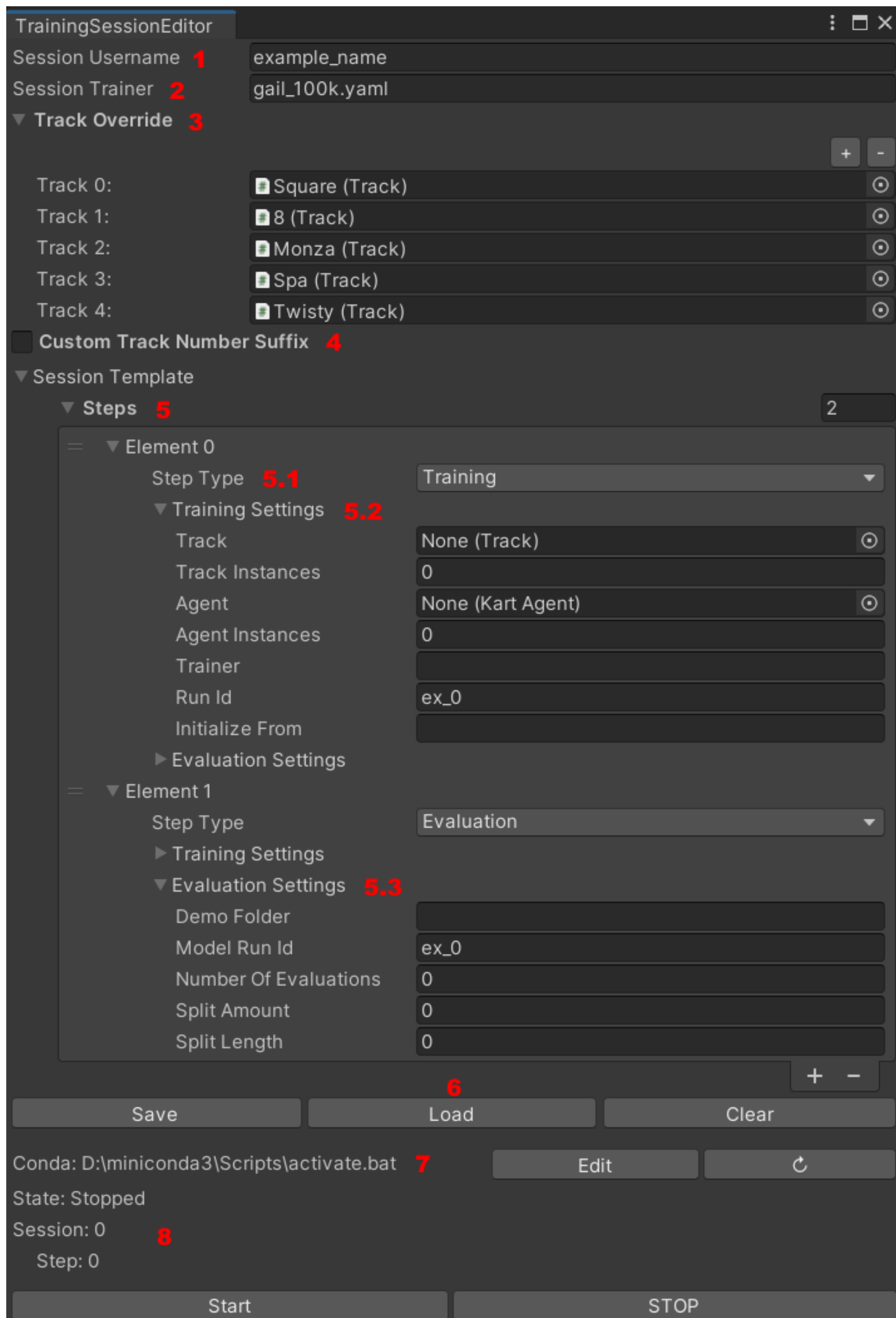


Figure 2.12: A screenshot of the `TrainingSessionEditor` script window.

Each step possesses the information needed for its execution (e.g. Track `GameObject` reference, number of agent instances, `trainer` file, number of evaluations to perform...)(Fig. 2.12-5.2, 5.3).

The Training Session can be saved as a `json` file for ease of retrieval (Fig. 2.12-6). Moreover, it is possible to specify some common parameters, such as *username* (Fig. 2.12-1) and ***trainer*** *name* (Fig. 2.12-2), so as not to repeat them for each step.

Each session can be repeated for different tracks (Fig. 2.12-3, 4). In this case all the fields and suffixes related to the tracks are automatically compiled at the start of each session.

Lastly, in the lowermost area of the window (Fig. 2.12-8) lie some diagnostics and progress status (Session State is better explained in section 2.4.1).

Training

Abstracting momentarily from Editor Scripts, the training process — be it *Imitation* or *Reinforcement Learning* — consists of the following steps:

1. Download the demonstrations in the correct folders (if needed);
2. Load the training scene in Unity;
3. Start the Virtual Environment (if present);
4. Launch the command `mlagents-learn`, passing all relevant Command Line Arguments such as `trainer` file and `run-id`;
5. After the script has started (but before it times out), enter `Play Mode` in the Unity Editor;
6. Instantiate agent(s) and track(s) and perform the training;
7. When training is complete, both the Scene will exit `Play Mode` and the script will stop automatically.

Automation As mentioned earlier, these steps are straightforward enough for a human, but automating them requires solving a few criticalities:

- The Virtual Environment startup file is machine-dependant;
- The demonstration folder changes for each training run, but it cannot be specified as an argument;
- `Play Mode` has to be started in a (relatively) precise time window;

The first two issues were solved without too much trouble by respectively setting up a dedicated, per-machine configuration file (Fig. 2.12-7), and by having a series of fixed, identical `trainer` files, only changing the demonstration directory between them.

The last constraint was the hardest to solve, requiring plenty of trial and error.

There needs to be some form of communication between the Python script (the launched party) and the Editor script (the launching party), so that the Editor knows when to start `Play Mode`. During training, communication between Unity and Python is handled by the respective libraries. This link is established via a `loopback` connection that is created when the Python script is launched.
This connection, however, sees the Python script as the listener, so no information can be gathered by listening on the connection prior to the launch of `Play Mode`.

As an alternative to the previous method, when launching a process, the `C#` language (which Unity uses as a scripting language) allows the caller to keep open streams to the `stdin`, `stdout` and `stderr` of the launched process, in order to read its output or send further commands to it. This is particularly useful for this setting, since the Python script signals that it is ready by printing a fixed string, that can be analyzed accordingly and acted upon.

However, in order to enter `Play Mode`, there must be no running processes bound to the Editor. Solving the timing issue as just described causes a deadlock situation, where the script is waiting for Unity to enter `Play Mode` and Unity is waiting for the script to finish before doing so.

The only solution to this obnoxious issue is launching the Python script as a detached process: this means that no information can be exchanged between the two, since the streams will not be established, but to the advantage of allowing Unity to freely enter `Play Mode`.

Now, neither the `loopback` connection, nor the process's `stdout` can be used to bind the Editor script and the Python script, so how does Unity know that the script is ready in order to start `Play Mode`?
It does not.
This solution is rather disappointing, as it simply operates on a timer. Since — at least in my configurations — performances are quite consistent between all machines, a fixed 10 second delay sufficiently accounts for all the initial loading.

Execution Training takes place in a dedicated empty scene, where the track(s) and agent(s) are instantiated procedurally. The Training configuration settings allow specifying how many copies of the track and agent to spawn, other than various parameters to be passed to the Python Script.

Training multiple instances of the agent at the same time is beneficial to the overall results: other than allowing *Reinforcement Learning* agents to learn about collisions between them, it speeds up training time significantly, effectively spreading training timesteps among all the instances.

Another crucial *quality of life* feature of the training script is the ability to pass an entire folder of demonstration as argument, instead of single files. In conjunction with the aforementioned ad-hoc `trainer` files, which are addressable via command line argument, this feature enables specifying a different demonstration folder for each training run.
This is especially useful when frequently switching demonstration files, for example when changing the number of laps to train on (as needed in Section 2.5.1).

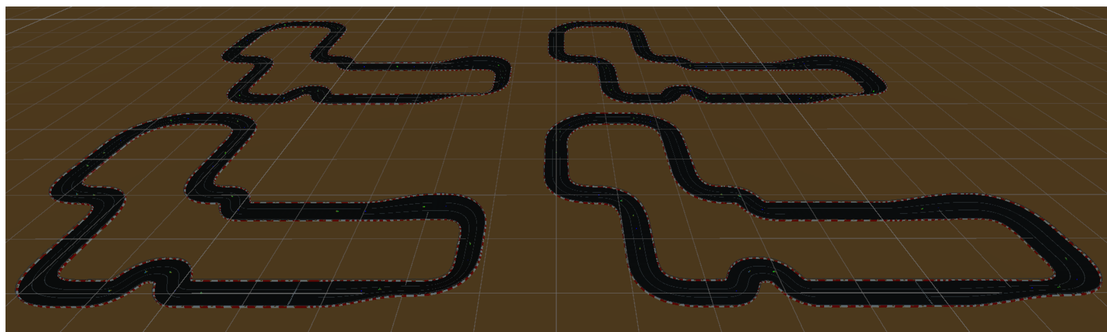


Figure 2.13: The training scene populated with the default training configuration of four tracks, each with 16 agents.

Evaluation

Evaluating is less involved than training from a conceptual standpoint, but no less complex from an implementation standpoint.

Execution Given a trained model and a replay `state` file, an evaluation is performed as follows:

1. The scene containing the reference's track (or an override if specified) is loaded, and the necessary setups are performed;
2. The reference trajectory is extracted from the provided `state` file;
3. The trajectory is split into a certain number of sub-trajectories of fixed length (both amount and length are specified via the evaluation settings);
4. Starting at each sub-trajectory, the agent is instantiated and allowed to drive for the defined amount of timesteps;
5. The resulting agent-generated trajectory is compared to the original;
6. All evaluations are averaged and saved in the `evaluations.jsonl` file of the model.

Due to an unplanned degree of variability in the agent's actions — despite enforcing *deterministic decisions* — all evaluations are averaged over a settings-specified amount of repetitions.

Moreover, when provided with a folder instead of a single file, the evaluation is performed (multiple times, if specified) for each file in the folder, each getting its line in the `evaluations.jsonl` file.

`jsonl` files

`jsonl`[7] is a variant of the extremely popular and widely used `json` (JavaScript Object Notation) file format.
The "*l*" in `jsonl` stands for *lines*, and it refers to the ability of storing one `json` object per line, instead of encasing them into an overarching array.

This distinguishing feature of `jsonl` makes it non-compliant with regular `json` syntax, which normally ignores newlines and requires separators between objects. However, it avoids reading and parsing the entire file, and allows adding new entries to the file simply by seeking its end and writing to it.

Given the fact that a model usually gets evaluated multiple times, this format is really convenient for saving many, independent measurements in a single file of arbitrary length.

State Management

Given the complexity of the interactions between the editor, the application and the external scripts, a Finite State Machine (FSM) had to be developed to supervise and coordinate these interactions. Figure 2.14 shows a diagram of the main workings of the FSM.

As shown in the diagram with rounded boxes, the states of the FSM are:

- Stopped: rest state;
- Started: checks input data and triggers execution;
- Waiting: during scene setup and timed waits;
- Training/Evaluating: during training/evaluation;

All steps in the box marked as "Session" can be repeated multiple times if more than one track is specified (Fig. 2.12-3).

The *State Change* conditional towards the bottom of Figure 2.14 refers to another state machine, managed by the Unity Editor: `EditorApplication.playModeStateChanged`, which tracks entering and exiting `Play Mode` and `Edit Mode` and allows reacting to these changes via callbacks.

[7] `https://jsonlines.org/`

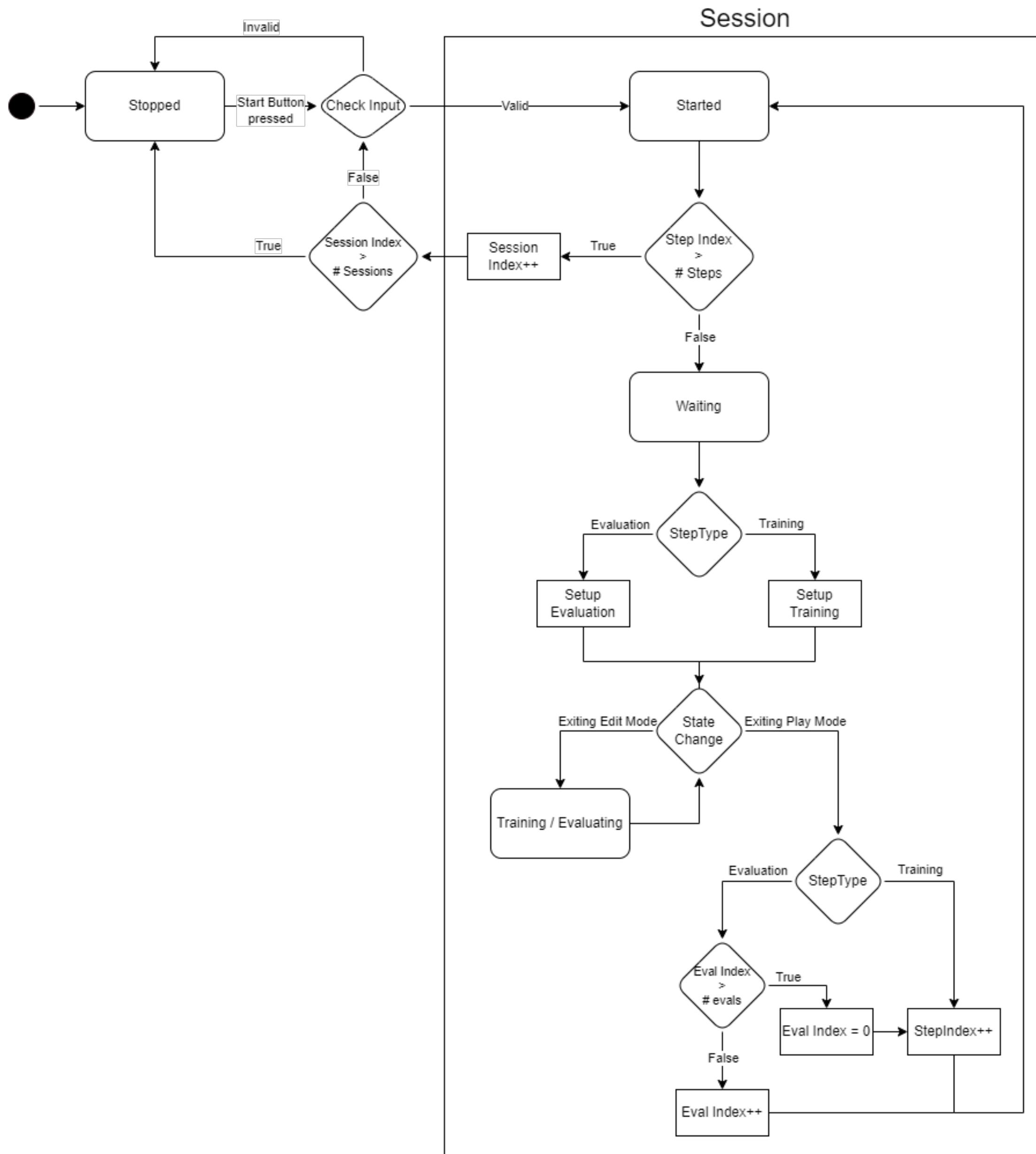


Figure 2.14: A diagram of the `TrainingSessionEditor` Finite State Machine.

2.4.2 Miscellaneous

As mentioned, training and evaluation were undoubtedly the most frequently executed tasks, however, a handful of editor scripts were created to aid in other, less important or more uncommon actions.

Demonstration Importer

This utility allows downloading `.demo` and `.state` file from an user's Firebase folder, and sorting them to the correct local folders.
Given the great number of separate files to download — reaching up to twenty per user — it was an invaluable tool for automating this frequent task.

It offers support for both types of training/evaluation configurations (as explained in Section 2.2.2), variable number of laps and arbitrary download folder.

Standalone Model Evaluation

It possesses all the features that are available to evaluations during a training session, however it integrates them with a few extra functionalities that were thought for standalone usage: namely a trajectory visualization utility — to see the trajectories that were generated by the agent right where they were generated — and an in-scene trajectory score viewer, to see the score of each sub-trajectory and the overall score without needing to save to a file.

Replay File Player

It allows playback of `.state` files.
It was not used much during development, but can prove useful, as replays are saved as plain text, and would otherwise be quite difficult to analyze or troubleshoot.

One completely unused feature of the entire file saving/loading system is support for `.input` files: instead of saving physical positions, velocities, etc., `.input` files only save user inputs.
Likewise, there exists an Input File Player, although it did not find any practical usage.

2.5 Experiment Setup

Before executing training and evaluations as previously described, a preliminary parameter search was performed to identify an optimal number of laps to ask of the users.
This analysis was inconclusive, and a reasonable number of laps was chosen based on the total runtime of the performance.

Thus the application was built, published on Unity Play and publicized to friends and colleagues. The final sample size amounted to twenty unique users, each getting a model trained and evaluated on the laps they ran.

2.5.1 Preliminary Lap Number Search

When creating the User Application, development had to face the simple and practical question of how many laps to set for every track.
This, however, opened another question, of greater significance, as to whether it made any significant difference to the agent's learning capabilities.
Seeking a definitive answer to this query, a lengthy parameter search was conducted on data obtained by means of my own demonstrations.

Firstly, all tracks were ran for ten laps. Considering that it required approximately fourty minutes of constant focus, it was obviously not feasible for a large-scale user test, but it served as an upper boundary to the agent's performance.

Secondly, models were trained and evaluated for each number of laps, ranging from two to ten, for both *unseen laps* and *unseen tracks* configurations (as explained in Section 2.2.2).

All evaluations were repeated around 50 times, spread amongst the various laps, the exact number depending the closest multiple of the number of laps (e.g 48 for 3 laps, 49 for 7 laps, etc.).

Unseen Laps Since some laps were needed to perform evaluations, only six models were trained, from three to eight training laps.
Training was started from scratch for each track, each being divided in ten steps of 100'000 timesteps each, totalling one million steps per track.
Evaluations were performed after every step on the laps that were not used for training, so from seven to two.

Unseen Tracks Nine models were trained, from two to ten training laps.
Training was performed on *Square*, *8*, *Monza* and *Twisty* (Tracks 0, 1, 2, 4), one million steps at a time, each step continuing from the resulting model of the previous one.
Evaluations were performed after every step (after each track) on all ten laps of *Spa* (Track 3).

As will be further analyzed in the next chapter, results were inconclusive, so it was arbitrarily opted to set four laps for each track: enough to grant sufficient familiarity with the track by the last lap, but not too much to strain the user, especially for longer tracks. Moreover, with these settings, the overall runtime of the app amounts to 10-15 minutes, which is adequate.

2.5.2 User Data Analysis

Twenty people took part in the data collection, each running four laps on each of the five tracks[8].
One model was trained and evaluated for each user, in the same *unseen tracks* configuration that was explained in the last section, with splits of both 20 and 40 timesteps in length.

In addition to the regular GAIL-trained models, one purely RL-based model was trained as a control. This model was trained exactly with the same settings (same tracks, same network parameters, same number of training timesteps), only difference being that it is completely unbiased: it receives no demonstrations and only tries to drive as optimally as possible (as described in Section 2.1.1).

[8] A couple of users experienced connection issues, which lead to the loss of a small portion of their data. However, impact was minimal, and their results ended up being fully (statistically) compatible with the others'.

Chapter 3

Results

3.1 Optimal Lap Number Search

Evaluations were performed by splitting every lap into 20 sub-trajectories, each being 20 timesteps long (400 ms).
Despite, as explained in the previous chapter, being evaluated at multiple times during training, all following plots show evaluations performed on the final model.

Results (Fig. 3.1a, 3.1b) show lower performance for later tracks, which is understandable, since they are ordered by complexity.
However, they also show negligible difference for any number of laps, and no clear trend. This can be attributed to multiple possible causes:

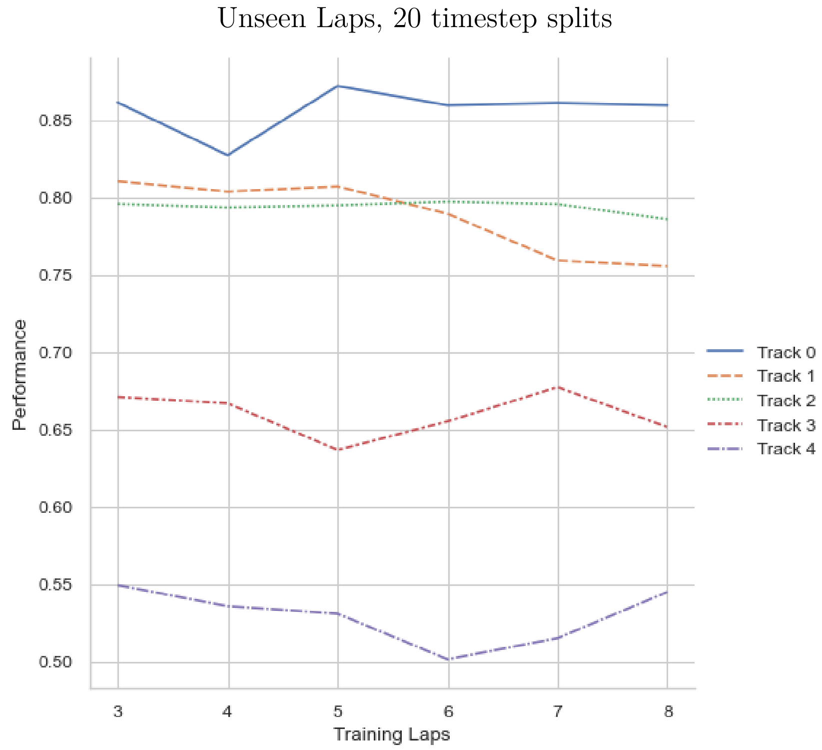
1. Demonstrations might be very data-rich: given the high recording frequency, the high generalizing potential of the simple observation format and the high data-efficiency of GAIL, a few laps might be enough reference for the model;

2. There might be some issue with the training library: even when providing more demonstrations, internally only the same few demonstrations might get processed;

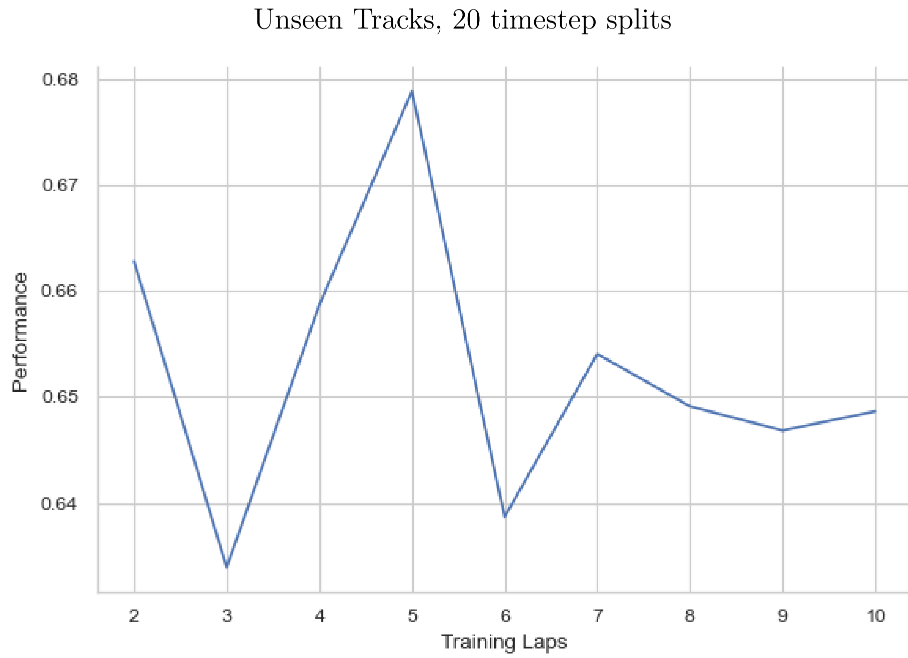
3. Evaluation sub-trajectories might be too short to evince differences between good and bad models: not much can go wrong in 400 milliseconds, starting from the same spot and speed. Longer splits might better exacerbate differences in behavior between the player and its respective model.

Disproving the first hypothesis would have required a restructuring of the entire project, other than an additional parameter search, so it was opted not to explore it and leave it as a possible explanation.
In the same vein, it was chosen not to pursue a proof to the second hypothesis either, as it would have lead the analysis too far off the intended scope of the thesis.

The third hypothesis, however, had a feasible solution readily available. Given the already established training and evaluation framework, evaluation splits could easily be adjusted in length, so it was decided to perform a second run of evaluations on all models, with a doubled length (40 timesteps, 800 ms).

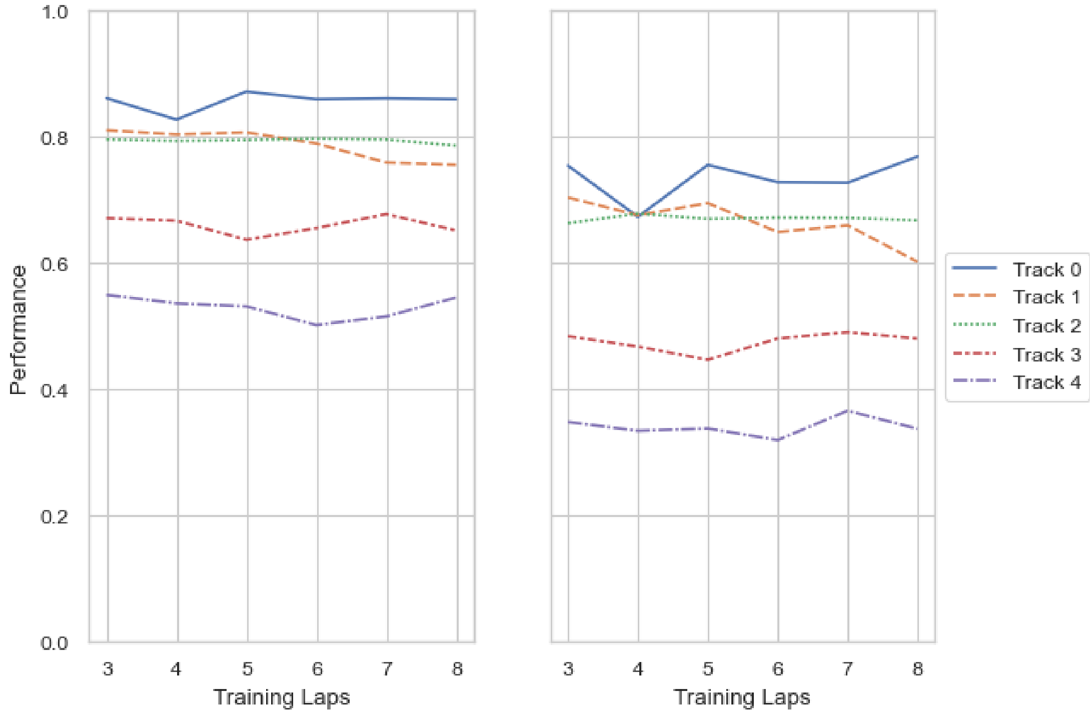


(a) Performance of models trained on each track as the number of training laps increases.



(b) Performance of the model resulting from training on Tracks 0, 1, 2 and 4 as the number of training laps increases. Evaluations performed on Track 3.

Unseen Laps, 20 vs 40 timestep splits



Unseen Tracks, 20 vs 40 timestep splits

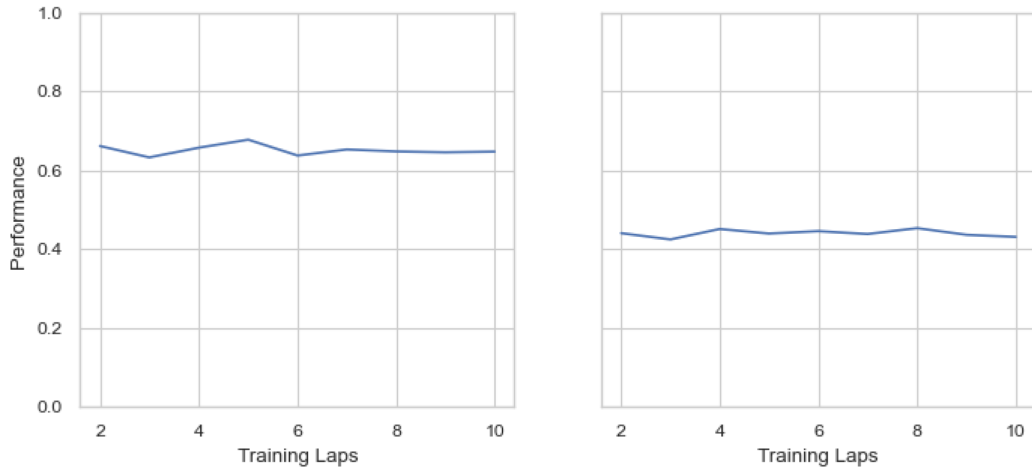


Figure 3.2: Comparison of performance for 20 timestep splits (left) against 40 timestep splits (right) for both Unseen Laps and Unseen Tracks configurations.

Increased Split Length

Results from the second run (Fig. 3.2, right) yielded results comparable to the previous run, only significant distinction being a lower average score — which was to be expected, given the higher potential for error that a longer trajectory entails.

It would seem that, in this case, changing split length did not add any discerning quality to the evaluation procedure.

3.2 Final User Test

Results with actual user data do not deviate significantly from what emerged from the preliminary analysis[1] (Fig. 3.2b).
As expected, lengthening the evaluation splits caused a drop in performance, again compatibly with what was obtained previously.

Interestingly, the user-independent RL model that was trained as a control ended up performing slightly better than the average of the user-based GAIL models (Fig. 3.4). While IL models were trained and evaluated on a single player each, the RL model was evaluated on all players' Track 3 replays, and the results were then averaged.

[1] Due to an oversight in the model configuration, results from previous analyses benefit from slightly higher values. However, it was ensured that none of the fixes changed any of the conclusions that were drawn from them.

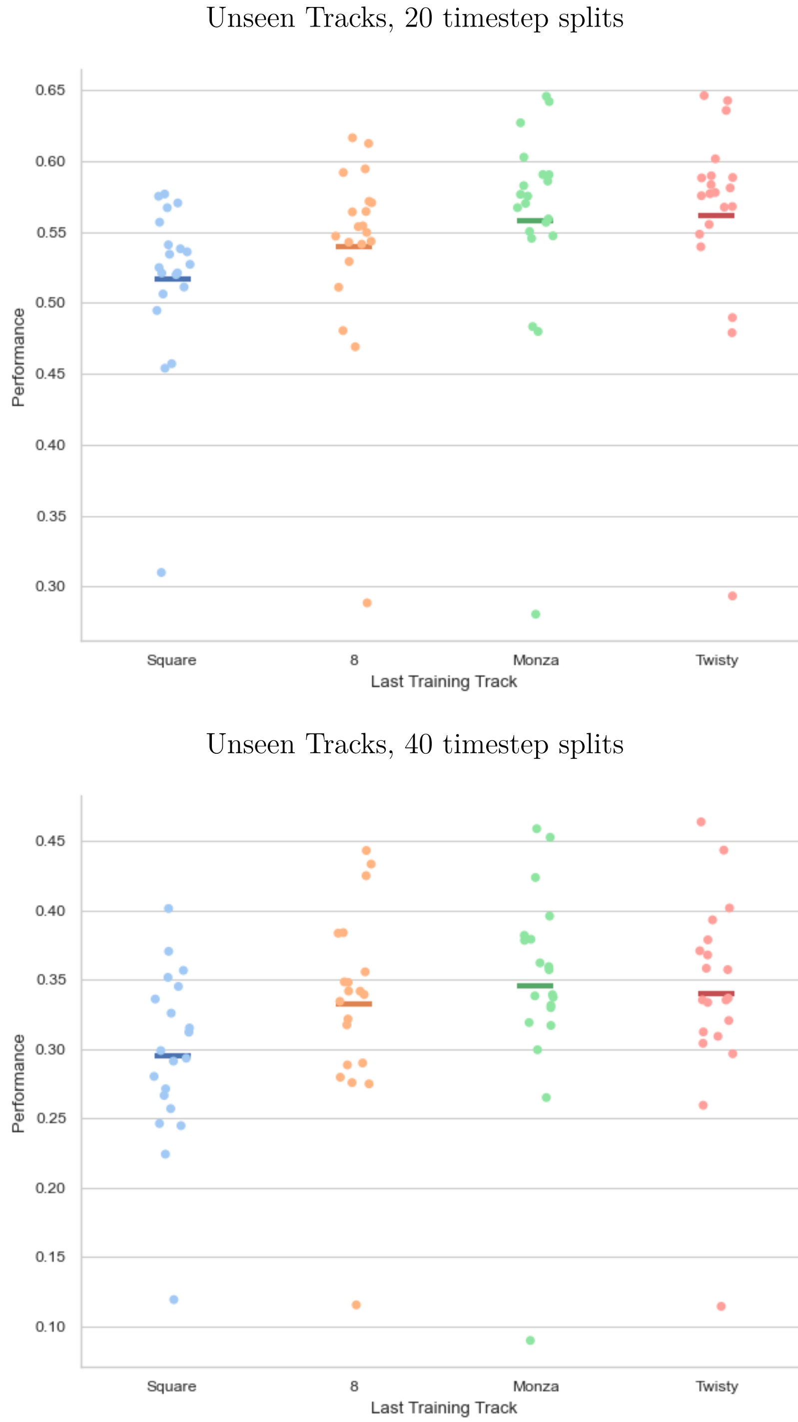


Figure 3.3: Scatter plots displaying the performance of models trained on user demonstrations and tested with 20 (a) and 40 (b) timestep splits.

Unseen Tracks, 20 vs 40 timestep splits

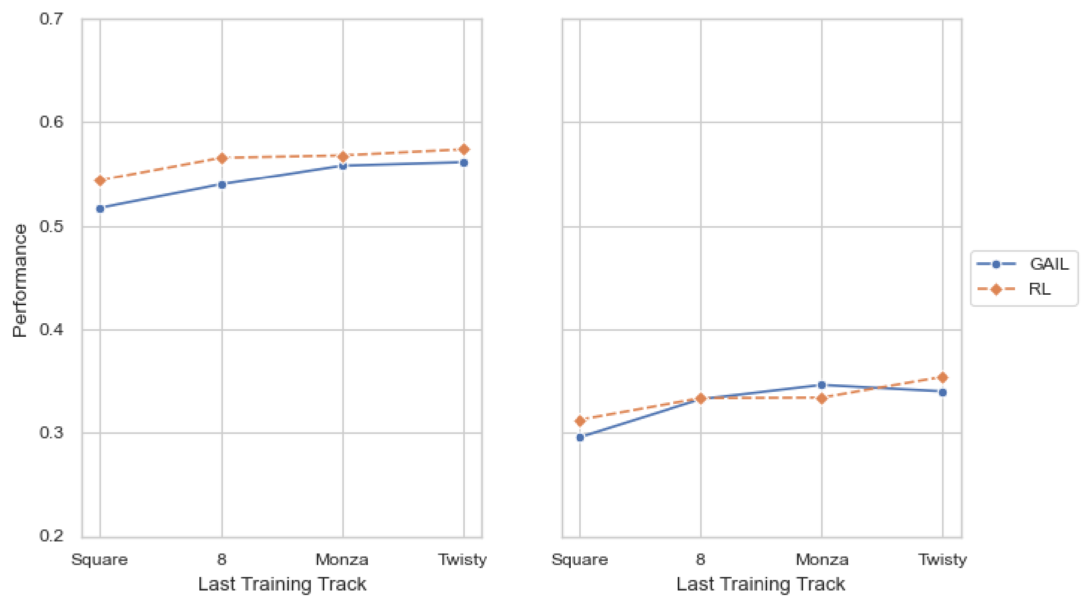


Figure 3.4: Plots comparing the average performance of all GAIL models with the score obtained by the RL model, on both 20 and 40 timestep splits.

Chapter 4

Conclusion

Despite experiencing a satisfactory development overall, most of the development time and effort ended up being taken by the user app (2.3) and the in-editor training framework (2.4.1) — unlike initially planned.
Due to this fact, some aspects of the project were neglected for the final product, opting for a minimal functioning implementation. There is much room for improvement for these features, especially given the final results.

4.1 Future Works

The most important absentee from the overall process is *hyperparameter optimization*. As mentioned in Section 1.1.2, often hyperparameter optimization is what makes a solid neural-network-based project a groundbreaking neural-network-based project.
However, it is to be taken into consideration that there are numerous hyperparameters to analyze, so a proper optimization requires adequate time and methodology.
The same case could be made for exploring different network architectures — which is not a hyperparameter per se, but likewise needs a dedicated search — especially Recurrent Neural Networks (RNNs) such as Long Short Term Memory (LSTM) Networks.

The second missing key aspect is a full-stack implementation of the SPG paradigm. This thesis focuses on the aspects that concern a single user, i.e. data gathering and model training, so no server infrastructure, nor multi-user interactions were considered. Once the standard is integrated in every aspect, it would be insightful to perform at least a qualitative test on user perception of the prediction engine at work, and possibly compare results to those of Spina et al. ([23]), who conducted a similar enquiry.

Another aspect that was overlooked in this thesis is the performance of different algorithms for IL, mainly Behavioral Cloning — as it is already included in the ML-Agents library — but possibly other popular algorithms such as Inverse Reinforcement Learning or simpler Machine Learning solutions [12].

In this regard, looking into ML-Agent's own support for custom trainers[1] might be a worthwhile investment.

One development that will likely be investigated once this project is further along the roadmap for the MPAI-SPG standard, is an assessment of the flexibility of the framework to different genres, such as FPS, racing simulations or strategy games.
Since the standard aims to be application-agnostic, it will be interesting to see how easily the process can be adapted, or if it will ever be a plug-and-play implementation for the developer.

4.2 Final Words

In this thesis, a framework was developed to aid in the collection and processing of user demonstrations for the purpose of training an autonomous driving agent based on the user's driving style.
This was obtained by means of Generative Adversarial Imitation Learning, a training technique that produces a behavior that resembles the reference material, but at the same time is able to adapt to unseen scenarios.

User data was gathered from multiple tracks, the number of laps being determined by a previous investigation, stored on a server, then downloaded and processed.

While personalized learning produced subpar results when applied to real users, it was found that the performance of an agent trained by Reinforcement Learning only was, on average, surprisingly high.

As stated, many improvements can be made to the training process, but the hope is that any future work can build on top the framework that was developed for this thesis.

[1] `https://unity-technologies.github.io/ml-agents/Tutorial-Custom-Trainer-Plugin/`

Ringraziamenti

Ringrazio i miei genitori per il supporto incondizionato che mi hanno sempre fornito e per la fortuna immensa che è stata averli avuti entrambi accanto per tutta la mia vita.

Ringrazio Sara, queen dei commit e spalla reciproca. Grazie per le confidenze e per la sincerità, per quest'amicizia bellissima nata per caso e per la fiducia che hai posto in me.

Ringrazio gli amici della Tavernetta estesa, compresi anche quelli di Torino. Grazie di essermi sempre stati vicini, per le innumerevoli serate passate in compagnia e per i bei ricordi che abbiamo condiviso insieme.
In particolare un grazie ai miei coinquilini, per aver coinquilinato pacificamente, un grazie a Tommaso, per avermi dato l'esempio su come non mollare mai, e un grazie a Luca, in cui in questi tre anni ho trovato un amico sincero e, soprattutto, freschissimo.

Ringrazio i miei "colleghi" del Lab 1 per avermi accolto da subito come uno di loro, per avermi supportato attivamente, e in generale per aver ravvivato la maggior parte delle giornate di questi 9 mesi di tesi. In ordine geografico: Alessandro, Stefano, Constantin, Francesco, Fabrizio, Daniele, Leonardo, Tommaso, Lorenzo, Marco, Giacomo, Antonio, Nicola e Pietro, e anche i miei colleghi (senza virgolette) tesisti Amedeo e Andrea.
È stato un piacere avervi conosciuti, a tutti voi auguro i più grandi successi nel vostro percorso dottorale e lavorativo. Spero di avervi lasciato un ricordo positivo quanto quello che avete lasciato voi a me.

Infine, vorrei ringraziare quel paio di persone che hanno voluto dedicare un quarto d'ora della loro vita alla raccolta dati e che non sono stati già compresi negli altri ringraziamenti.

Questa tesi è stata scritta al 100% da me, nessuna forma di AI generativa è stata impiegata in nessuna parte della stesura del testo, se non come contenuto del testo stesso. A tal proposito vorrei ringraziare WordReference, Reverso e Thesaurus.

Bibliografia

[1] Statista, 2023. https://www.statista.com/outlook/amo/media/games/worldwide.

[2] Reuters, 2023. https://www.reuters.com/technology/video-game-market-recovery-pick-up-steam-2024-strong-console-sales-report-2024-01-23/.

[3] Shengmei Liu, Xiaokun Xu, and Mark Claypool. A survey and taxonomy of latency compensation techniques for network computer games. *ACM Comput. Surv.*, 54(11s), sep 2022. ISSN 0360-0300. doi: 10.1145/3519023. URL https://doi.org/10.1145/3519023.

[4] Aphex34, 2015. CC BY-SA 4.0, https://commons.wikimedia.org/wiki/File:Typical_cnn.png.

[5] Pranoy Radhakrishnan, 2017. https://towardsdatascience.com/introduction-to-recurrent-neural-network-27202c3945f3.

[6] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning, 2016. URL https://arxiv.org/abs/1606.03476.

[7] Shengmei Liu, Mark Claypool, Atsuo Kuwahara, Jamie Sherman, and James J Scovell. Lower is better? the effects of local latencies on competitive first-person shooter game players. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380966. doi: 10.1145/3411764.3445245. URL https://doi.org/10.1145/3411764.3445245.

[8] David Halbhuber, Philipp Schauhuber, Valentin Schwind, and Niels Henze. The effects of latency and in-game perspective on player performance and game experience. *Proc. ACM Hum.-Comput. Interact.*, 7(CHI PLAY), oct 2023. doi: 10.1145/3611070. URL https://doi.org/10.1145/3611070.

[9] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003®. In *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '04, page 144–151, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113942X. doi: 10.1145/1016540.1016556. URL https://doi.org/10.1145/1016540.1016556.

[10] Nishanth Kumar. The past and present of imitation learning: A citation chain study. *CoRR*, abs/2001.02328, 2020. URL `http://arxiv.org/abs/2001.02328`.

[11] Boyuan Zheng, Sunny Verma, Jianlong Zhou, Ivor W. Tsang, and Fang Chen. Imitation learning: Progress, taxonomies and opportunities. *CoRR*, abs/2106.12177, 2021. URL `https://arxiv.org/abs/2106.12177`.

[12] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Comput. Surv.*, 50(2), apr 2017. ISSN 0360-0300. doi: 10.1145/3054912. URL `https://doi.org/10.1145/3054912`.

[13] Tianhao Zhang, Zoe McCarthy, Owen Jow, Dennis Lee, Ken Goldberg, and Pieter Abbeel. Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. *CoRR*, abs/1710.04615, 2017. URL `http://arxiv.org/abs/1710.04615`.

[14] Ben J. Geisler. An empirical study of machine learning algorithms applied to modeling player behavior in a "first person shooter" video game. 2002. URL `https://api.semanticscholar.org/CorpusID:7579864`.

[15] Jack Harmer, Linus Gisslén, Jorge del Val, Henrik Holst, Joakim Bergdahl, Tom Olsson, Kristoffer Sjöö, and Magnus Nordin. Imitation learning with concurrent actions in 3d games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018. doi: 10.1109/CIG.2018.8490398.

[16] Bernard Gorman and Mark Humphrys. Imitative learning of combat behaviours in first-person computer games. *Proceedings of CGAMES*, pages 85–90, 2007.

[17] Juan Ortega, Noor Shaker, Julian Togelius, and Georgios N. Yannakakis. Imitating human playing styles in super mario bros. *Entertainment Computing*, 4(2):93–104, 2013. ISSN 1875-9521. doi: https://doi.org/10.1016/j.entcom.2012.10.001. URL `https://www.sciencedirect.com/science/article/pii/S1875952112000183`.

[18] Geoffrey Lee, Min Luo, Fabio Zambetta, and Xiaodong Li. Learning a super mario controller from examples of human play. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2014. doi: 10.1109/CEC.2014.6900246.

[19] Siddharth Reddy, Anca D. Dragan, and Sergey Levine. SQIL: imitation learning via regularized behavioral cloning. *CoRR*, abs/1905.11108, 2019. URL `http://arxiv.org/abs/1905.11108`.

[20] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Learning drivers for torcs through imitation using supervised methods. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 148–155, 2009. doi: 10.1109/CIG.2009.5286480.

[21] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. No-regret reductions for imitation learning and structured prediction. *CoRR*, abs/1011.0686, 2010. URL `http://arxiv.org/abs/1011.0686`.

[22] Luc Le Mero, Dewei Yi, Mehrdad Dianati, and Alexandros Mouzakitis. A survey on imitation learning techniques for end-to-end autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(9):14128–14147, 2022. doi: 10.1109/TITS.2022.3144867.

[23] Daniele Spina, Andrea Bottino, Davide Cavagnino, Leonardo Chiariglione, Maurizio Lucenteforte, Marco Mazzaglia, Francesco Strada, et al. Ai server-side prediction for latency mitigation and cheating detection: the mpai-spg approach. In *IEEE CTSoc Gaming, Entertainment and Media*. IEEE, 2024.

[24] Jannik Quehl, Haohao Hu, Ömer Şahin Taş, Eike Rehder, and Martin Lauer. How good is my prediction? finding a similarity measure for trajectory prediction evaluation. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6, 2017. doi: 10.1109/ITSC.2017.8317825.