POLITECNICO DI TORINO

Master's Degree in Software Engineering

Master's Degree Thesis

AI Server architecture for latency mitigation and cheating prevention: MPAI-SPG solution

Supervisors
Prof. MARCO MAZZAGLIA
Prof. FRANCESCO STRADA
Prof. EDOARDO BATTEGAZZORRE

Candidate
DANIELE SPINA

October 2023

Contents

1 Introduction

With their rising popularity, computer games have become one of the most sought-after forms of entertainment worldwide. A growing trend among these games is the online multiplayer functionality, allowing players from different locations to come together in a shared virtual world. However, network latency between players and the server can negatively impact game play by reducing responsiveness and introducing inconsistencies. Consequently, this can hamper player performance and diminish the overall quality of the gaming experience. In this thesis, we studied the solution proposed by the MPAI organization, called **Server Predictive Game**, in short, **SPG**. **SPG** aims to construct a robust server architecture capable of anticipating future game states. Whenever the server notices any missing data from clients or the evaluated game state is too different from the anticipated one, the server modifies the game state by applying the prediction. Since the system works even when the evaluated game state differentiates from the anticipated one, it also prevents cheating. This research focuses on a multiplayer online racing video game. The game was developed specifically for the project using an authoritative server architecture. To record a vast dataset for the training, we developed neural networks to automate player movement. Once the database was ready, we conducted extensive experiments to determine the optimal neural architecture and hyperparameters for the SPG's engines. After implementing the SPG components we integrated the trained networks inside the game. We studied the behaviour of the system, helping MPAI-SPG in its research.

1.1 MPAI

This thesis is done in collaboration with MPAI, a non-profit organization committed to maximizing data utilization efficiency across various domains.

MPAI, short for Moving Picture Audio and Data Coding, is a standard committee dedicated to pioneering advanced technological standards in the Computer Science field through the use of Artificial Intelligence. With the collaboration of multiple research groups, MPAI aims to create innovative technical specifications in various research areas, such as Video Compression (MPAI-EVC), Context-based Audio Enhancement (MPAI-CAE), and Human-Machine conversation (MPAI-MMC).

While MPAI takes inspiration from the previous work conducted by MPEG, it strives to bridge the gap between standardized specifications formulated by the committee and the practical implementation of these technologies in everyday applications. To ensure the effective utilization of these instruments, MPAI defines Intellectual Property Rights (IPR) Guidelines, including Framework licenses.

Different research groups, including SPG, work in various fields and adhere to a strict development process to establish a standard. This process comprises a multi-step workflow:

1. **Interest collection**: Gathering use cases.
2. **Use Cases**: Proposing use cases, describing them, and merging compatible and analogous ones.
3. **Functional Requirements**: Extracting functional requirements that the research group will implement to fulfil the standard and use cases.
4. **Commercial Requirements**: Developing and approving the Framework License.

5. **Call for technologies**: Preparing a document that introduces the standard and seeks companies capable of developing compatible technologies satisfying commercial and functional requirements.
6. **Standard development**: Establishing the standard through a dedicated Committee.
7. **Community comments**: Publishing the developed standard for community feedback.
8. **MPAI Standard**: Approving the commented and revised standard in the General Assembly and making it accessible to all.

The General Assembly, formed by Principal Members with the right to vote, approves the progress of each research project, discussing improvements during monthly sessions. At present, SPG is in the *Functional Requirements* stage, working on a prototype that elucidates the research aim and functionalities.

1.2 Network Architecture

Developers can implement networking in online gaming in many different ways. The article [9] describes the main techniques used, which are:

- Peer-To-Peer
- Client-Server
- Server Pool

1.2.1 Peer-To-Peer

In the **peer-to-peer** technique, the player's machine connects directly to the opponent's computer without any intermediaries. The advantage of this configuration is the lack of dependence on additional devices apart from those used by the players themselves. However, due to this setup, it becomes extremely difficult to prevent cheating as there is no external device not controlled by the players that can monitor the game state. Furthermore, players with less powerful computers may degrade the gaming experience for all other players connected to their devices.

1.2.2 Client-Server

In this architecture, the server acts as a bridge for all communications between all connected clients. This structure overcomes the shortcomings described earlier in the peer-to-peer architecture. Since the server is controlled by none of the players, it can objectively monitor the game state, making it easier to detect and prevent cheating. If the server is powerful enough, it can also mitigate the impact of weaker devices connected to the game. The only disadvantage of this architecture is the additional code required to govern the server.

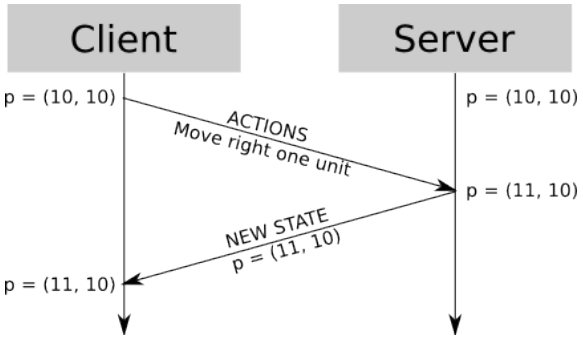

Figure 1: Authoritative Server

1.2.3 Server Pool

Lastly, the **server pool** technique involves multiple interconnected servers communicating with each other as equals, while clients connect to a local server. The advantage of this solution is a reduced demand on a single server as the number of players grows. Similar to the server-client technology, additional code is necessary to describe the server's behaviour, and it is also necessary to define communication between the various servers. This structure proves advantageous in games with an extremely high number of players at the cost of increased latency due to server-to-server communication and greater complexity.

1.2.4 Authoritative Server

The most common commercial solution is the Client-Server architecture. This choice stems from the need to prevent cheating by players, especially in competitive games where any form of cheating would degrade the gaming experience for all other players. The server pool architecture also achieves the same result, but increases the perceived latency for players. For these reasons, client-server architecture is the preferred option. **Authoritative server** is an architecture belonging to the client-Server typology. As Gambetta describes in his article [4], the authoritative server represents the real state of the game, while clients are merely privileged spectators. Each player's input is not immediately applied; instead, it is sent to the server. The server then updates the game state and sends the new game state to every client. This way, even if a player modifies the state on their machine, such as adding health to their character, the health value on the server will remain unaffected. If the cheater is hit, they will be defeated regardless of what their device shows.

In summary, the game state is controlled by the server alone. Players send their actions to the server, which updates the game state periodically and then the server sends the new game state to each client. The following behaviour is shown in figure number 1.

2 Client-Server Latency Mitigation Techniques

Latency is defined as the delay between a player's input and the game responding with audible or visual output. Different types of latency exists:

- Local Latency

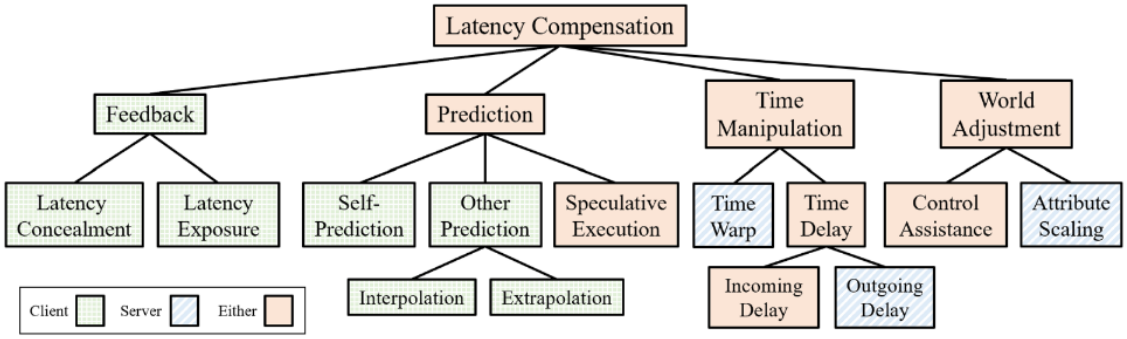

Figure 2: A taxonomy of latency compensation techniques for network games.

- Network Latency

Local Latency is caused by the communication delay between the user's peripherals, while **Network Latency** is due to the communication delay between client and server. This work only focuses on the latter. The objective of MPAI-SPG is to minimize latency and prevent potential cheating. With regards to the latter category, the only action that developers can take is to verify that the the player input received by the server are feasible. In case of highly competitive games, to be more secure, an external software is installed directly on the player's gaming device, which checks for any programs capable of manipulating the game state. Regarding latency mitigation, there are numerous possible solutions available. The following article [9] list all the research done on the subject and also some of the techniques used in published video games. Figure 2 depicts the taxonomy of latency compensation techniques. They are arranged based on common shared characteristics.

2.1 Feedback

Feedback provides audible or visual information to the player based on latency, without changing the state of the game.

2.1.1 Latency Concealment

Whenever an action is performed by a player, before rendering its result the client has to wait for the server response. With **Latency Concealment**, the client immediately run the animation and or the audible cue without actually changing the state of the game. The client still needs to wait for the server to update the game but the player may feel it more responsive. There are not many formal evaluation about this technique but it in the article [18] Latency Concealment improved virtual reality head tracking up to 380ms of latency.

2.1.2 Latency Exposure

A visual indicator is used to show the magnitude of the latency from the client to the server. Most games displays numeric value on the corner of the screen which report the client's **round-trip time**. In the article [20] shadows are used as visual indicators, they represent the estimated position of

the game avatar on the server based on latency. The result shown a slight increase of the player enjoyment.

2.2 World Adjustment

Instead of trying to solve the communication problems, **World Adjustment** changes the game state to decrease the difficulty based on latency.

2.2.1 Control Assistance

Player inputs is adjusted in order to accommodate for inaccuracies caused by latency. An example is **sticky target** where in an shooter game the gain of the mouse is reduced when near a target. The amount of assistance can be modified based on latency. This technique, is mostly used to mitigate local latency. For network latency, it slightly increased input quality when tested with low levels of correction, so it was not too obvious for the players, as reported in the article [10].

2.2.2 Attribute Scaling

The game difficulty is adjusted based on latency of each player in order to make player actions easier to complete even with high latency. An example could be decreasing the obstacle dimension to make it easier to avoid. This technique has been proven effective both for quality of experience by the player and reducing the perception of the delay [11].

2.3 Prediction

Prediction is the most researched of all solutions. The client predicts the future state and applies it while waiting for the server. The throwback of this solution is the possibility of a wrong prediction. In those cases the client must reconcile with the server, and this fix can be noticeable and may even be jarring to the player. The following techniques are explained in detail by [1] and [4] since they are the most used in commercial applications.

2.3.1 Client Prediction

Normally, the client sends a message to the Server whenever it plays an action. The Server evaluates the next game state based on the inputs received. Only when the response reaches the initial sender, the client renders the next frame. Thus the client has to wait for the server every time. With Client Prediction, instead of waiting, the client evaluates by itself the aftermath of the inputs and uses this computation to continue running the game. When the server packet arrives, a label specifies which client inputs it refers to. The player's machine updates the game state, then applies to the response received all the commands not yet analysed by the server. This step is called **Server Reconciliation**. Client Prediction achieves a smooth game play experience for the Client as if the entity controlled by the player was in a single-player game. For example, in the image 3, the client moves to the right by one unit two times, reaching the new position after 100 ms of animation. The communications between the Client and Server have a delay of 250 ms. Normally, the player would have reached the new position only after 500ms ($250 * 2$). It had to wait for both inputs to be handled by the server. With Client Prediction, the player's actions are processed immediately, thus the player moves without any delay. When the server response reaches the client, it reconciles with

the server. To the response of the first input the client applies, on top of it, the second command. Thus evaluating the result. While for the second response, there are no other commands to apply. In this way, the server still has an authoritative architecture, but all the waiting for the server response is hidden from the player's point of view. Client Prediction is only applied to the entity

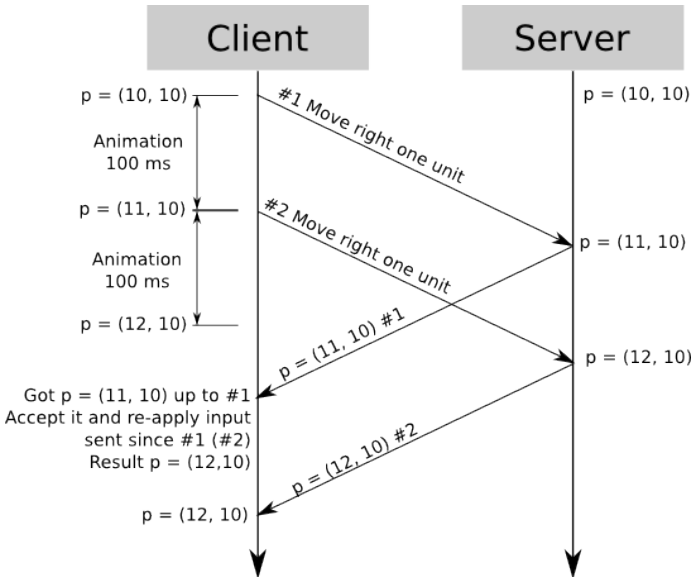

Figure 3: Client prediction

controlled by the client itself. Thus all other entities still may be experienced by the player with high latency. There are two ways to solve this problem:

1. **Entity prediction** While waiting for the server response, the client predicts the actions of all the other entities and uses the prediction to render the game. When the response arrives, it checks If the prediction is wrong. In this case, the client has to re-simulate everything from the erroneous prediction until the last frame rendered.

2. **Entity interpolation** While waiting for the server response, the client renders the action performed by the other entities described in the previous response of the server. Thus showing the other entities slightly delayed in time.

2.3.2 Entity Prediction

The effectiveness of Entity Prediction, also called Extrapolation, highly depends on the performance of the prediction algorithm. The most used algorithm for prediction is **Dead Reckoning**. The basic version of this algorithm uses the current player's position and his velocity to evaluate his future position. Multiple variations of the Dead Reckoning exist. Most of them are compared in the article [7] inside a 2D racing game. The comparison shows that the best overall algorithm is **Input Prediction** [13]. Input Prediction is a form of Dead Reckoning that includes user inputs as a factor when making predictions. More specifically it tries to predict not only the next position but also the

setting of the controlling input device. Then, based on the game's physics, the user input is used to evaluate the new object's position. Paper [2] tries to improve the Dead Reckoning using artificial intelligence. Their proposed solution is composed of two phases: in the first, a machine learning algorithm determines whether or not Dead Reckoning will correctly predict the new position of the player. If so, Dead Reckoning is used, otherwise, the second phase is executed. Dead Reckoning can easily predict the player's movement if he does not change its direction. Thus, the first algorithm bases its evaluation on whether or not the player will change its direction. In the second phase, another machine learning algorithm evaluates the new player's direction, depending on their game style. Multiple algorithms were tested. Bootstrap Aggregating algorithm was the best, achieving an accuracy of 78,76% against 44,54% of the original Dead Reckoning. It is worth mentioning that Bootstrap Aggregating performed better than AntReckoning [21], one of the best variations of the Dead Reckoning algorithm, which already improved Dead Reckoning's accuracy by 30%. Another advantage of the Smart Reckoning algorithm is its reusability. AntReckoning needs a specialist to manually configure multiple parameters to make it work, while Smart Reckoning automatically learns them.

2.3.3 Entity Interpolation

Commercially, developers implement Dead Reckoning in games where the player does not change direction frequently. In all the other cases Entity Interpolation is used. With Entity Interpolation, when a Client receives the latest information of other entities, it predicts their past states based on the current and previous states. Then it update the game state based on these predictions until it reaches the latest information received. This is done in order to achieve a seamless experience, since entities are not instantaneously moved to other positions. Due to the interpolation, all other entities are shown slightly delayed in time. Entity Interpolation is rarely used on its own, which is why there is not much research focusing on this technique. That being said, research [8] shows an improvement of 2% in accuracy in First Person Shooter (FPS) games, while article [15] demonstrates a decrease in game state inconsistencies by about 50% in arcade-style games. However, if nothing else is done, the illusion breaks down when an event needs high spatial and temporal accuracy, such as shooting at a moving target: the position where Client 2 renders Client 1 does not match the server nor Client 1's position. To solve this problem a technique called Time Warp, is used.

2.4 Time Warp

Time Warp, also called Lag Compensation, is a Time Manipulation technique. The objective of Lag Compensation is, to synchronise actions between all clients. When a packet from a client is received, the server uses the player's latency to rewind the game state to analyse the action in the correct time frame. If it causes any modification to the game, all other clients are notified, and they have to reconcile with the server. For example: In the image below we have two clients, Bob and Alice, with a delay of 50ms and 250ms respectively. When Alice receives the new Bob position she fires him. Using Lag Compensation the server rewinds the game state and analyses the packet received as if it was received 250ms earlier. The server confirms that Bob was hit. Bob is notified of the hit $250 + 50$ms later. The grey area shows that Bob could have moved into a new position at that time, therefore when Bob was notified it may be out of the line of fire of Alice.

From the attacker's perspective, everything went smoothly, as if there was no latency, while for the targeted player, it looks like their death is unreasonable. This problem is called **Shoot Behind Covers**, and it becomes a reality when one of the Clients has a very high delay. This

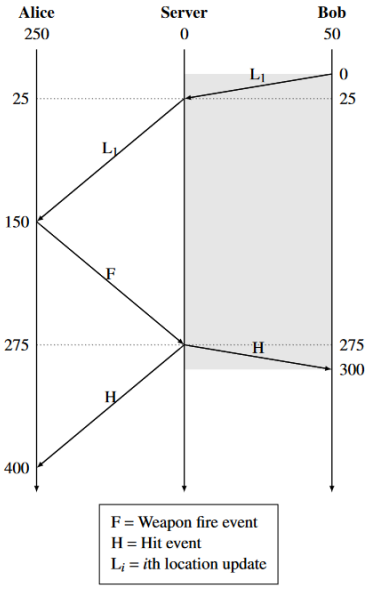

Figure 4: Lag Compensation

problem was analysed in the paper [12] in the context of a 2D shooter game. They tried to keep the game synchronised between all clients by putting the delay compensation on the client side instead of the server. Each Client predicts the behaviour of opponent players based on past game information. The algorithm that produced the best prediction is Single Deep Reinforcement Learning. Deep Reinforcement Learning refers to the model's architecture used for the prediction and how it is trained. The model uses three repetition of the block composed of a convolutional layer and activation function, to extract features from a snapshot of the game. The extracted features are then utilized to estimate the next command performed by the opposite player. A reward is given to the model based on the similarity between the estimated and actual commands. The evaluation criteria used are hit rate, which refers to the ratio between the number of hits to the total number of projectiles fired, and the Euclidean distance between the actual location and the predicted one. Single DLR achieves better synchronisation of the player position than the Lag Compensation technique, but the hit rate is still lower than classic Lag Compensation.

2.5 Commercial use

Now let's see an example of real-life application of the techniques described above. Rocket League, a car sports video game, uses several latency mitigation techniques, such as Client Prediction, Interpolation, and Extrapolation, as shown in the video [16]. Thanks to the deterministic physics system of the game and the knowledge of player inputs, it is possible to determine with almost absolute certainty the future state of the game. By having access to its player inputs, the client's device can represent the future game state without waiting for the server confirmation, as seen for the Client prediction technique. The prediction represent the game state reached after the time delay between the same client and the server. Therefore when the player actions are received on

the server they are correctly synced with the game state of the server. Using the same logic, the client tries to predict the state of every other entity in the game. However, unlike before, the client is not aware of the inputs of other players. The input prediction algorithm used is very simple: the system assumes that the input stays the same compared to the previous input. This assumption is correct most of the time, but the game allows extremely fast driving maneuvers with sudden changes in direction and position almost instantly. Because of this, when one of these maneuvers is played, the prediction algorithm fails. In these cases, the client becomes desynchronised with the server. When the corrected game state arrives from the server, the client tries to interpolate its state with the one received. Interpolation is used in order to avoid instantaneously changing the game state.

2.6 Advantage of our method

Latency mitigation techniques are evaluated based on responsiveness and consistency perceived by the players, as described in the article [9]. MPAI-SPG's solution focus on consistency between server and clients. Each client merely recreates the exact game state described by the server. In the event of latency, it will be the server's responsibility to address the issue, thereby maintaining the game state synchronised among all players. Differently from the Time Warp technique, which also focuses on consistency, MPAI-SPG also helps with shrinking the server-client delay. A client with high latency will still receive the results of their action with a delay, but thanks to the prediction system, the server will send the new game state before receiving the action from the client, effectively cutting the waiting time in half. Since MPAI-SPG is installed only on the server, is possible to implement additional techniques on clients to mitigate this problem. By adopting this approach, all clients can achieves an optimal user experience.

3 MPAI-SPG solution

The official site of MPAI-SPG describes the goal of their solution as follow:

"Server-based Predictive Multiplayer Gaming (MPAI-SPG) aims to mitigate the audio-visual and gameplay discontinuities caused by high latency or packet losses in online- and cloud-gaming and to detect game players who are getting an unfair advantage by manipulating the data generated by their game client."

All the terms used in this chapter are defined in the Table 1.

To better understand the MPAI solution, the typical functioning of an authoritative server in an online multiplayer game is presented, whose architecture is shown in the Figure 5:

1. The Clients send Controller Data to Server
2. The Game State Engine:
   (a) Computes the Game Messages using the Controller Data (CD) from the Clients.
   (b) Sends the Controller Data and/or the Game Messages to the Game Engines (GM).
3. The Game Engines:
   (a) Compute updated Game Messages (GM').
   (b) Send the updated Game Messages to the Game State Engine.

Term	Acr.	Definition
Controller Data	CD	Data generated by Game input/output devices
Game Message	GM	Messages sent to the Game Engines by the Game State Engine using its current Game State and the Controller Data. Game Engines send back new Game Messages GM
Game State	GS	A data structure containing all Game Messages at a given instant
Game State Engine	GSE	A process managing the Game State of the game machine playing a game.
Game Engines	GE	Engines receiving Game Messages from the Game State Engine and providing Game Messages back to the Game State Engine
Physic Engine	PE	The Engine that computes the behaviour of the world as different entities interact with each other, using the physics rules set by the developers
Rules Engine	RE	The Engine manages the rules within the game system triggered by changes in the system occurring during interactions among players or CPU-controlled characters or by in-game messages independently of system state changes determined by the players
Behaviour Engine	BE	The Engine that receives inputs from the different local or remote players, translates them into actions by the actors in the system and takes care of moving the actors managed by the CPU participating in the action at that moment

Table 1: Terminology

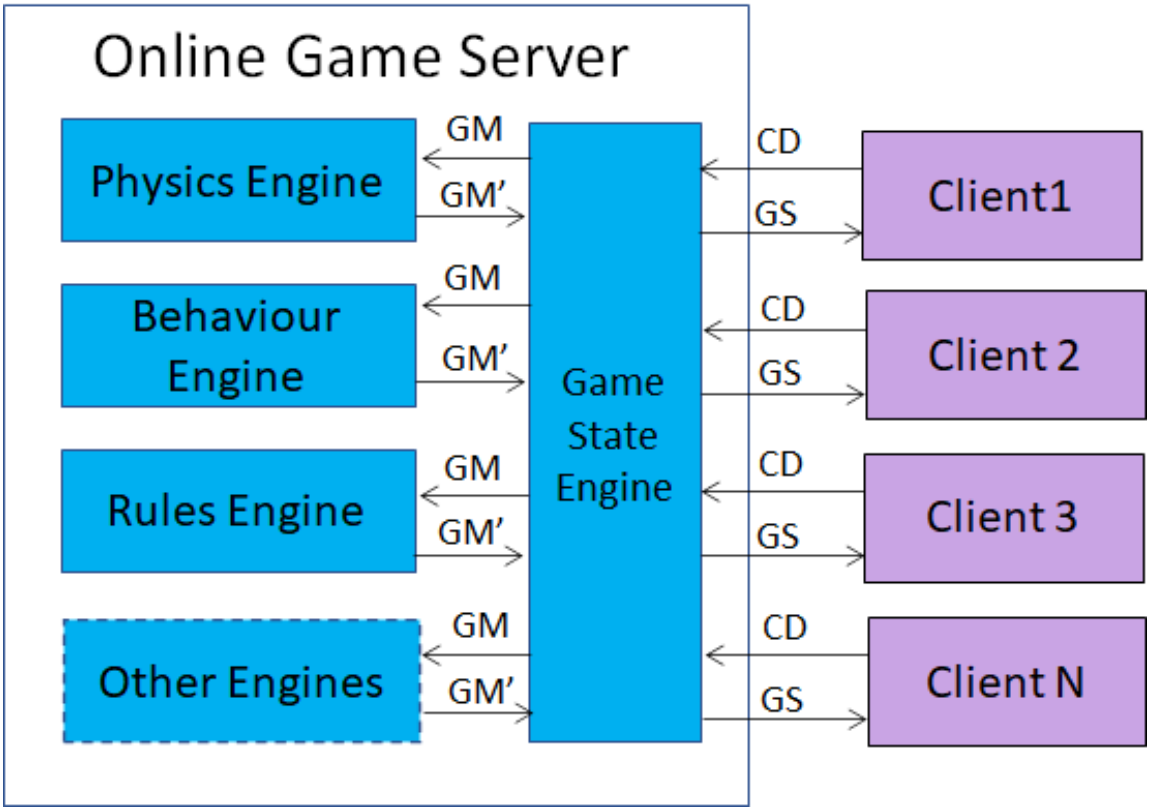

Figure 5: Multiplayer online gaming model

4. The Game State Engine:
   (a) Computes the updated Game State.
   (b) Sends the new Game State to all clients.
5. The Clients produce and display their video frames.

The MPAI-SPG solution, shown in Figure 6, supports the game server. It works in parallel, together with the authoritative server. The logic described above remains unchanged except for an additional final step. MPAI-SPG consists of three main blocks:

- Game State Engine AI
- Game Engines AI
- Game State Engine

Both AI and non-AI Game State Engines function in a similar manner to what was shown for the server. The first evaluates the GMs to send to each Game Engine AI, while the second evaluates

the predicted Game State, GSp, by combining the output of the Game Engines AI. MPAI-SPG uses three different Engines:

- Rule
- Behaviour
- Physics

Table 1 provides the definition of these three Engines.

Rule Engine is responsible for managing the game rules triggered by system changes due to player actions or the system itself. In our case, there are only two rules:

1. Number of laps performed by each car
2. Ranking position of each player

Behaviour Engine handles the input of each player. Taking into account the Controller Data (CD), this engine manages each car:

1. Velocity
2. Rotation
3. Position

Physics Engine evaluates the behaviour of the world based on the interactions between the various participants in the system, using the physical rules described by the developers. In this game, it involves calculating the velocity, position, and rotation of each car. However, this information is already evaluated by the previous engine.

The MPAI-SPG architecture operates as follows:

1. The Controller Data (CD) and the Game State (GS) enter the Game State Engine-AI.
2. The Game State Engine-AI produces a set of GM*. In normal conditions, the set of GM* should be close to the set of GM, the one evaluated by the server.
3. The GM* feeds the Game Engines AI's.
4. The Game Engines-AI's produce estimates of the set of Game Messages (GMp). In normal conditions, the set of GMp should be close to the set of GM'.
5. The green Game State Engine, functionally equivalent to the blue Game State Engine:
   (a) Computes the predicted Game State (GSp). In normal conditions, GSp should be close to GS.
   (b) Sends the predicted Game State (GSp) to the Server's Game State Engine.
6. The Server's Game State Engine:
   (a) Corrects its GS if:

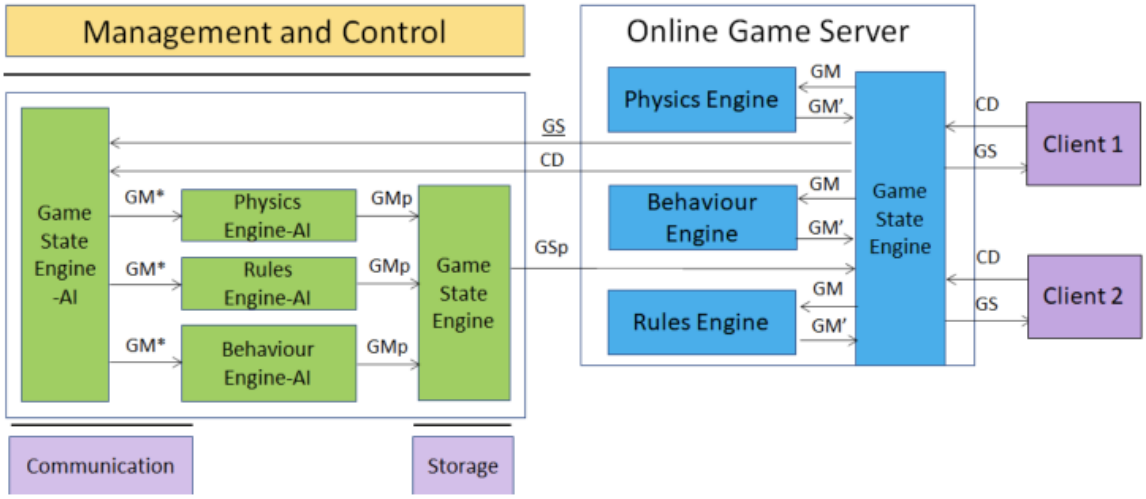

Figure 6: MPAI-SPG architecture

- GSp is too different than GS (anticheating).
- The Game State Engine was missing some Controller Data.

(b) Sends its current GS (GS) to the Game State Engine-AI. Repeating the process for the next frame

From the aforementioned logic, it can be observed that the effectiveness of the MPAI-SPG solution depends solely on the Game Engines AI. Therefore, it is necessary to define and develop appropriate AIs for the type of problem to be addressed.

3.1 Time Series Prediction

The proposed solution has to make a prediction of the next game state by extracting data from the current one. In order to achieve accurate predictions, the system must take into account each player's playing style. It can only be studied through the commands previously chosen by the player. As a result, the network's input should contain information not only from the present, but also from past game states. We can describe this input as a temporal series. In the field of mathematics, a time series refers to a collection of data points that are arranged or plotted in a chronological order. Typically, a time series is generated by capturing data at regular intervals, resulting in a sequence of discrete-time data. The quest for the optimal neural networks for time series analysis in machine learning is referred to as **Time Series Prediction**. The paper [6] shows a review of the state of the art techniques used for Time Series Prediction. In the mentioned article different architectures were tested with different data type, which were retrieved by a specific formula or from real-world events. The architectures that are the most relevant for this work are:

1. Multi Layer Perceptron
2. Recurrent Neural Network

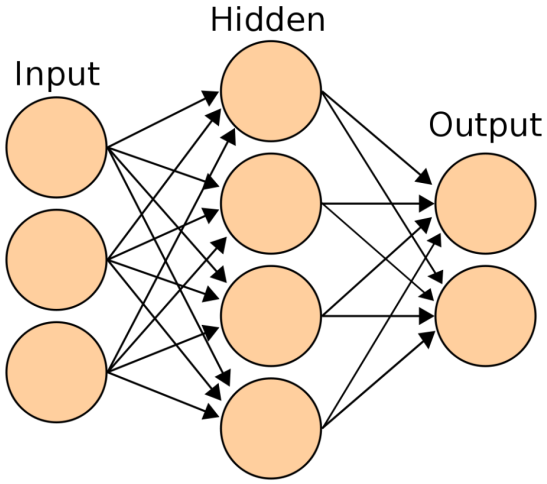

Figure 7: MLP

3. Long Short-Term Memory
4. Convolutional Neural Network
5. CNN & LSTM

In the following chapter, the functioning and benefits of the previously nominated architectures are expounded upon, as they have proven effective within the context of Time Series Prediction.

3.1.1 MLP

One of the first architectures ever used to solve this problem is Multi-Layer Perceptron, shown in Figure 7. A perceptron is one of the first neural models ever used. Given an input vector, different weights are applied to each element. The weighted inputs are then added to each other. The sum is used as input by a nonlinear function, called Activation Function. The perceptron's output is the result of the activation function. Mathematically speaking, its behaviour is defined as follows:

$$\theta = \sum_{i=1}^{N} w_i \times x_i$$

$$y = g(\theta + b) = g(z)$$

Where:

- x is the input vector
- w is the vector containing all weights

- N is the dimension of the input vector
- θ is the result of the linear combination
- y is the output of the perceptron
- b is the bias
- z is the sum between the bias b and the linear combination
- g is the activation function

A MLP si composed by three layers:

1. Input layer
2. Hidden layer
3. Output layer

The input layer is the vector containing all the input. The Hidden layer is composed of multiple perceptrons , which for simplicity are called activation units. The input is forwarded to the activation units of the Hidden layer. Their outputs are used by the units of the Output layer to evaluate the result. Each layer is fully-connected, meaning that each unit, also called neuron, has a weighted link to all the neurons of the layer behind. This type of neural network is called feed-forward, meaning that data flows from the input to the output layer.

3.1.2 RNN

Recurrent neural networks are a type of artificial neural network designed to handle sequential data. These networks share similarities with the Feedforward neural network, such as the Multilayer Perceptron, previously discussed. The primary distinction lies in the connections between nodes. Every neuron in the MLP is fully connected to the previous ones, while in an RNN, the connections form a cycle, meaning that a node's output affects subsequent input to the same node. The cycle formation in RNNs allows information to persist over time, making the architecture capable of analysing the temporal dynamics of the input. The RNN input takes the form:

$$Input = [Sequencelength, features]$$

Each input instance is composed of multiple elements equal to the Sequence length and, for each element, a set of data values, referred to as features. At time T the network receives inputs consisting of the corresponding features, and the nodes' output related to the (T-1) computation. We can simplify the network by considering it as a recurring structure that repeats over every timestep. Figure 8 displays the aforementioned architecture.

The RNN is associated with three significant problems, the Gradient Explosion, the Vanishing Gradient, and short memory limitation, where it struggles to recall previous steps in a long sequence. During training, backpropagation is the cause of both Gradient Explosion and Vanishing Gradient. If the gradients are too large, they can accumulate over time until the overall gradient explodes, producing an unstable network. Instead, if the gradients are too little, the system weights are not changed enough for the network to learn. This results in a stale network that fails to improve.

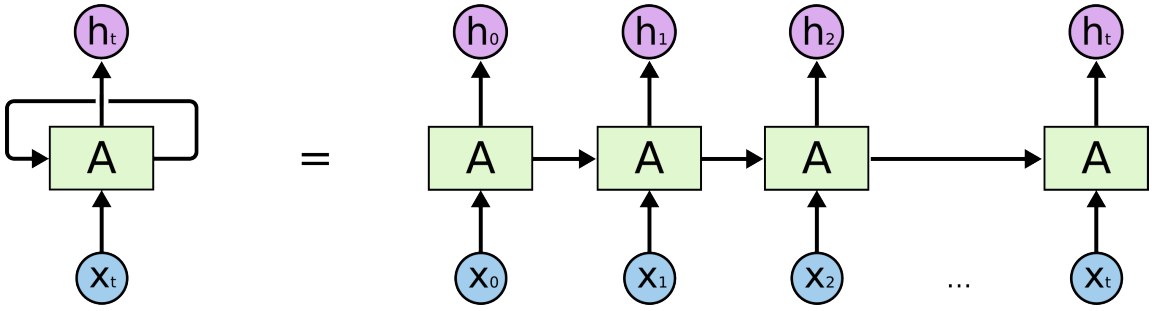

Figure 8: RNN structure

3.1.3 LSTM

Long Short-Term Memory (LSTM) neural networks solve all the previously described problems. LSTMs were first described in 1997 in the article [5]. LSTM is a type of Recurrent Neural Network. The difference lies in the structure repeated over time. A classic RNN is composed of a single neural network layer, while, in an LSTM, there are multiple. An LSTM unit is composed of a cell, which remembers values over arbitrary time intervals, the hidden state, which contains information about the input sequence that the LSTM has processed so far, and three gates that regulate the data that flows into and out of the cell. Figure 9 shows the structure of LSTM compared with RNN. Gates control information by assigning a value between 0 and 1. There are three different gates inside an LSTM unit:

- Input Gate
- Output Gate
- Forget Gate

The first one in the unit is Forget Gate. It decides which information to discard from the previous cell state. The Input Gate determines which part of the new input to store in the current state. The Output Gate controls what information of the current state to save in the hidden state. LSTM is one of the best architectures for Time Series Prediction. Its architecture keeps information of past instances and is not affected by the Vanishing Gradient or Explosion Gradient, which makes the RNN unsuitable for this work.

3.1.4 CNN

A Convolutional neural network (CNN) is a specialised class of artificial neural network commonly used for image-related tasks. Its architecture is typically composed of a combination of three layers:

- Convolutional layer
- Pooling layer
- Fully Connected layer

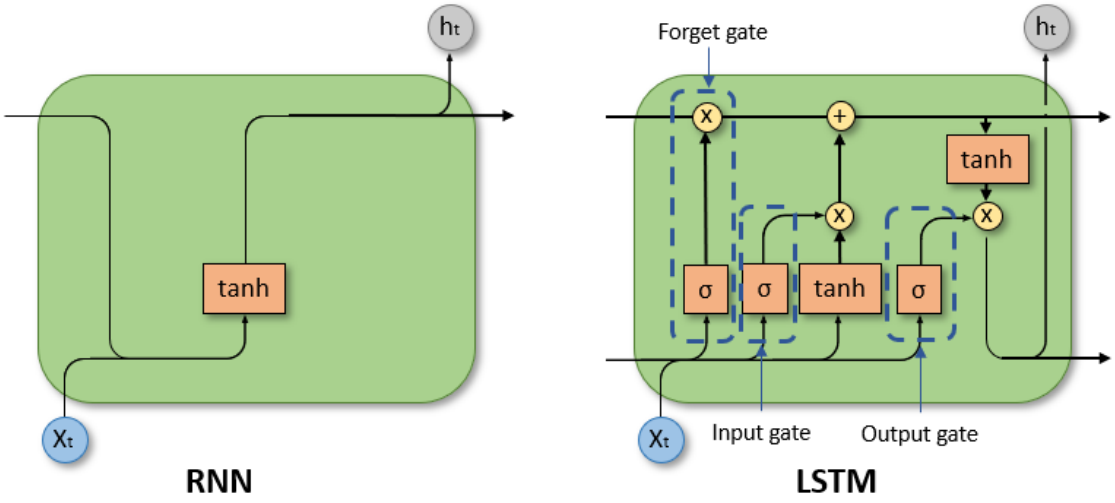

Figure 9: RNN vs LSTM

Convolutional Layer The Convolutional layer is the core block of the architecture. The layer's parameters consist of a set of learnable values called Kernels, which perform a dot production with the input, extracting the main features from it. Since the kernel is usually spatially lower than the input, it repeats its operation by sliding across it. The output is called Activation Map, which typically has smaller dimensions than the input. The following formula evaluates its size:

$$W_{out} = \frac{W - K + 2P}{S} + 1$$

Where:

- W is the input volume size
- K is the kernel field size
- S is the Stride, which is the sliding size of the kernel
- P is the padding, which indicates the padded size of the input with zeros or other values

Figure 10 displays an exampli of comvolution, with an input size $W = 5$ and kernel size $K = 3$.

Pooling Layer Pooling is a form of non-linear down-sampling. Given a matrix, it partitions the input into regions, usually of the same width and height, and for each one, applies a function which retrieves a single value. Each output value is a summary of the features inside the square of origin. The applied function depends on the type of pooling. The most frequently used is max pooling. It retrieves only the maximum value.

When declaring a Pooling Layer, it is necessary to define three elements:

- Type of pooling

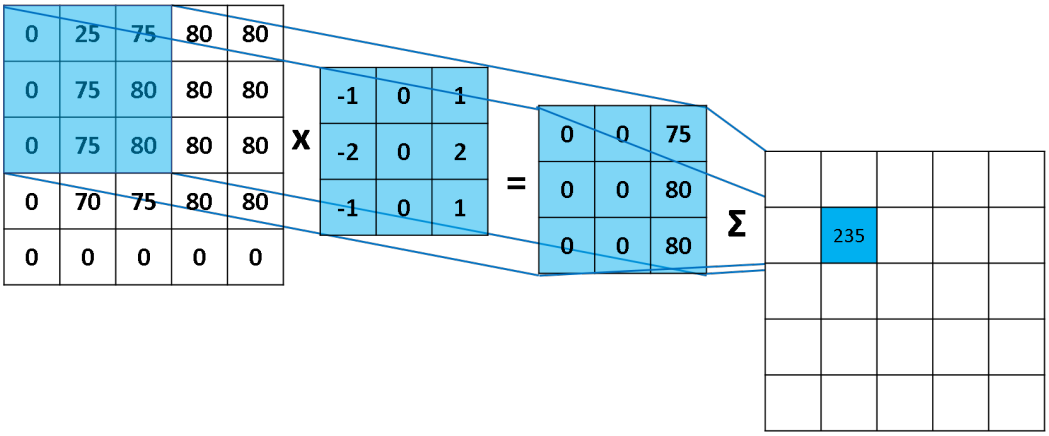

Figure 10: Example of convolution

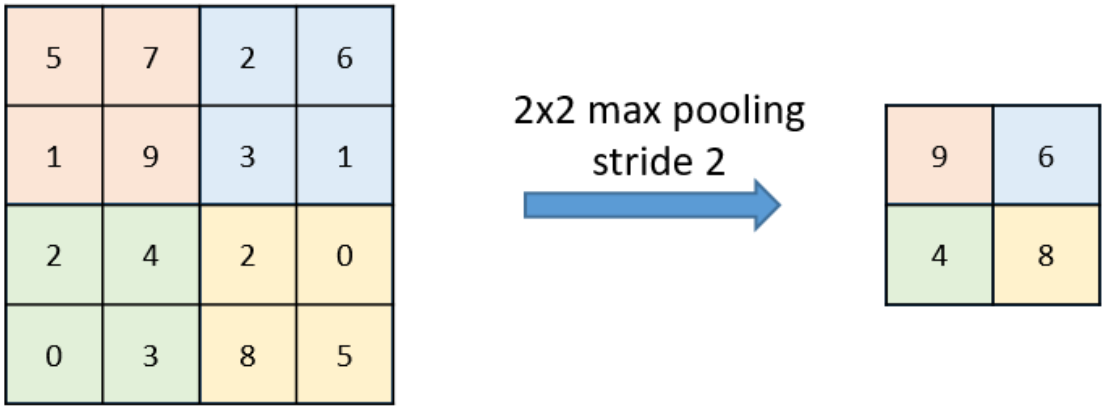

Figure 11: Example of max pooling

- The size of the rectangles
- The stride

Like the convolutional layer, the stride specifies the distance between each rectangle. When the stride is lower than the square width, squares will share some elements with each other. The idea behind pooling is to extract the most relevant feature of each input region, thus reducing the number of parameters and the amount of computation needed. Also, it helps control overfitting. Figure 11 depicts an example of max pooling. It is common to connect the pooling layer after a convolutional one, thus reducing, even more, the number of starting parameters.

Fully Connected Layer After many convolutional and pooling layers, a sequence of fully connected layers classifies the input. The MLP chapter explained how they work. The prototype of a CNN was described in 1980 by two neurobiologists [3]. Since then, numerous articles proved their usefulness, especially in image processing, where they are considered state-of-the-art. Other articles tested CNNs in other fields. The research [6] demonstrated the architecture's capability with time-series predictions.

3.1.5 CNN & LSTM

A **CNN-LSTM** neural network combines two deep learning architectures: Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM) neural network. The article [17] described for the first time the CNN-LSTM network. Its design is frequently applied to sequence-based tasks, including but not limited to image or video classification. It proves advantageous in scenarios where one must effectively account for both spatial and temporal information simultaneously. The behaviour of the architecture follows this structure:

- The CNN extracts relevant features from each input frame or region of interest and produces a sequence of feature maps.
- These feature maps are then fed into the LSTM network to model the temporal dependencies within the sequence.
- The LSTM takes the sequence of feature maps and processes them step by step, considering both the current frame's features and the information stored in its memory cells.
- The final hidden state of the LSTM passes through fully connected layers for classification or regression.

Overall, a CNN-LSTM neural network leverages the CNN's ability to extract spatial features and the LSTM's capability to model long-term dependencies to make predictions on sequence-based tasks.

3.2 The Game

For this research, a custom game was created. It is a racing game in which players compete to be the first to complete three laps. The key components of the game are as follows:

- Car
- Racing Track

3.2.1 Car

The car consists of two elements:

- Rigidbody3D
- Wheel Collider

The **Rigidbody3D** is a Unity component that controls the position of an object through physics simulation. The car's position cannot be directly manipulated; rather, force must be applied to the Rigidbody to move the car. The Rigidbody keeps track of parameters such as velocity, angular velocity, mass, and drag. A **Collider**, another Unity component, is attached to the Rigidbody to make the car susceptible to collisions. Additionally, four wheels are added to the car, with each wheel having its own Wheel Collider. The **Wheel Collider** is a special collider for grounded vehicles. It has built-in collision detection, wheel physics, and a slip-based tire friction model. The car is built using these components to create a more realistic and similar experience to commercially available racing video games. This allows for an evaluation of how the MPAI system interacts with complex systems. Detailed information on the functioning of these components can be found in the Unity Documentation [19]. The player input is divided into three variables:

- $moveZ \in [0, 1]$
- $moveX \in [-1, 0, 1]$
- $breaking \in [0, 1]$

moveZ variable determines whether an acceleration force is applied to the car. If **moveZ** is equal to one, a torques is applied to the back wheels of the car.

$$WheelCollider.motorTorque = moveZ \cdot MotorForce$$

MotorForce controls the amount of torque applied to the wheels. To aid the car achieving a better initial acceleration, we apply an additional force to the Rigidbody. It is calculated as follows:

$$|Force| = MotorForce \cdot e^{-Velocity \cdot BoostDuration}$$

BoostDuration is a parameter which controls the amplitude of the boost, while the **velocity** ensure that this force is greater at lower speeds and reduced at higher speeds. However, to prevent the car from accelerating indefinitely, a manual limit is imposed through a constant called **MaxSpeed**.

moveX variable determines the direction of the car. If it is less than zero, the front wheels are turned to the left; if they are grater then one we turn them to the right, otherwise they maintain their current direction.

$$WheelCollider.steerAngle = moveX \cdot maxSteerAngle$$

maxSteerAngle specifies the maximum angle that wheels can turn.

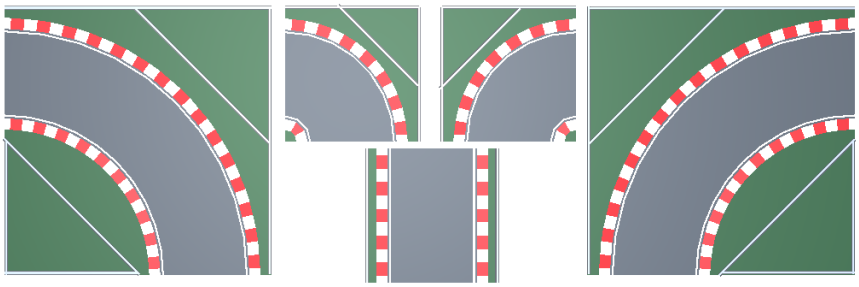

Figure 12: Race track tiles

breaking variable indicates whether the car is breaking or not. To apply brakes, a force is directly applied to the front Wheel Colliders:

$$WheelCollider.breakTorque = breaking \cdot breakForce$$

Here, **breakForce** is a parameter that controls the strength of the braking force. Finally, after applying the appropriate forces, the positions and rotations of each wheel are updated.

3.2.2 Racing Track

All the racing track available for the game are built by combining different tiles. There are four different tiles:

- Straight
- Turn 15° left
- Turn 15° right
- Turn 30° left
- Turn 30° right

The following tiles are depicted in Figure 12. A lap is considered complete only when a vehicle successfully passes through all designated **checkpoints**. These checkpoints are invisible colliders, uniquely identified, which trigger a verification process to determine if the object passing through is indeed a car. Upon successful identification, the **CarManager** is promptly notified. The primary function of this component is to keep a record of the latest valid checkpoint each car has traversed. A checkpoint is deemed valid if the car has successfully passed through the preceding gate. Each checkpoint is assigned an integer number and is organized in sequential order. For instance, gate 5 is followed by gate 6 and preceded by gate 4. If a player initiates the race by moving in reverse, the final checkpoint is encountered as the first one. In this scenario, the checkpoint communicates with the CarManager to verify if the gate is valid. Since the last valid checkpoint corresponds to the starting one, number 0, only checkpoint 1 is considered valid, therefore the CarManager abstains from updating the lap counter. To establish rankings, each car's performance is evaluated based on the number of completed laps, and if these numbers are equal, the algorithm compares the last valid checkpoint. Should there still be equality, the algorithm further assesses the distance between the car and the next gate.

4 Self Driving Cars

Before starting training the MPAI-SPG system, it is necessary to have a database. To build it, we need to play countless games and write all the needed information. Instead of playing numerous races, we trained different artificial intelligences to play games autonomously. To achieve this goal, we used a Unity library called ML-Agents. This library facilitates neural network development and training until they can effectively control objects in the Unity environment.

4.1 Theoretical Background

The agents undergo training using a method known as Reinforcement Learning. This approach is one of the three main divisions of Machine Learning techniques. With Reinforcement Learning, it is necessary to define three identities in our environment:

- Observations
- Actions
- Reward Signal

Observation The agent collects a set of numerical observations that represent attributes of the surrounding environment.

Actions A set of numerical values, which can be continuous or discrete, that define the actions to be taken by the agent.

Reward Signal A scalar value indicating how well or poorly our entity is performing.

The combination of these three identities corresponds to the basic cycle of our agent. Whenever it needs to make a decision, it collects observations, from which it determines its actions. Based on these actions, it receives a reward. The agent's objective is to identify the most suitable actions to take, considering the circumstances, to attain the highest possible reward. Training will therefore involve a series of trials and errors until it can define a logic that leads to the highest compensation. These three identities serve as the foundational pillars of the training process, especially the reward signals. Different Reinforcement Learning techniques are defined based on the previously described identities:

- Sparse Reward
- Dense Reward
- Imitation Learning
- Curriculum Learning

4.1.1 Sparse & Dense Reward

Both of these techniques refer to the frequency of the reward signals. During the training process, the Sparse Reward function is often equal to 0, changing only in a few states or actions. Taking a chess game as an example, a Sparse Reward function could be related to the end of the game. This reward is always zero except when the game is over. At that moment, the reward will be positive if the agent has won or negative otherwise. Dense Reward, on the other hand, is the opposite, with the reward being non-zero at almost every time step. This way, the agent receives feedback at each time step.

4.1.2 Imitation Learning

If defining or balancing the reward functions is too complex, Imitation Learning allows the agent to learn based on the behaviour of an expert agent. Instead of randomly trying different actions, the agent tries to define a policy based on the actions performed by a "teacher", always considering the context in which they are executed. The training process is divided into two phases. In the first phase, the agent analyzes the behaviour of the teacher and defines the policy. In the second phase, the agent is trained, and the previously defined policy determines the rewards the agent will receive during this phase. The negative side is that the agent can never be better than the teacher itself.

4.1.3 Curriculum Learning

At the start of its learning process, an agent decides which action to perform randomly since it has never received feedback. If the environment or the reward signals are puzzling, then it's arduous for the agent to understand the best line of action. Curriculum Learning divides the learning process into multiple phases called lessons. Each lesson is tougher than the other. In the first lesson, where the agent acts completely random, the environment and or the reward are modified to be simpler. Therefore it is easier for the agent to define a primitive policy. In the following lesson, the agent can start from what he has learned previously, thus making it easier for the agent to navigate through the more arduous environment.

4.2 Training Environment

To start training it is necessary to define the three identities:

- Observations
- Action
- Reward signals

4.2.1 Observation

The trained agent must effectively manoeuvre through any race track, requiring not only the avoidance of collisions but also the ability to traverse the course correctly. It is imperative to establish specific numerical values that the agent can utilize to comprehend its surroundings. Unity ML-Agents library offers a component known as the **Ray Perception Sensor 3D**, which enables an agent to perceive the surrounding world. It works by continuously projecting multiple beams in

all directions, starting from the attached object position. Developers can modify the total number of rays and their respective lengths. Each ray informs the agent if it intersects with an object, providing details such as the point of intersection, the distance from the ray's starting position, and the object's tag. Rays exclusively detect objects equipped with a Collider component, identified by tags specified in a predefined list established by the programmer. Objects possessing a Collider component with different tags are disregarded. To navigate a track accurately, an agent must perceive three crucial elements:

- Checkpoints
- Walls
- Cars

The agent's vehicle is equipped with three distinct Ray Perception Sensor 3D components, with each sensor dedicated to one of the aforementioned entities. This design choice is due to the ray behaviour: they stop once they hit a valid object. If a car is behind a checkpoint, and only one sensor is used, the agent will just see the checkpoint. When the agent passes the checkpoint it will not have enough time to avoid the other player.

The checkpoint's sensor only detects if there are any checkpoints and where they are. It does not help the machine learning algorithm understand the direction to follow. During a race the sensor usually hits two checkpoints, the next one and the one the car has just passed through. Using only the information received by the sensor it is not possible to understand which of the two checkpoint the agent has to go through. We use a different attribute for representing this information. Unity assigns the **Transform** component to every object in the scene, which is responsible for the position and rotation of the associated object. Among the Transform's public properties, there is the **Forward** property. Unity Scripting API defines Forward as follows:

"Returns a normalized vector representing the blue axis of the transform in world space."
The blue axis represents the Z axis. Each checkpoint's Z-axis in every track points to the direction to be followed. As the last parameter for the observations, the following formula is used:

$$Direction = Lerp(checkForward, nextCheckForward, percDist).normalized \tag{1}$$

Where:

- $checkForward$ is the Forward vector of the checkpoint to reach
- $nextCheckForward$ is the Forward property of the next checkpoint, the one after the one to reach
- $percDist$ is the distance, expressed as a percentage, between the agent and the checkpoint to pass through

The $Lerp$ function evaluates the linear interpolation between two points:

$$Lerp(a, b, t) = a + (b - a) \cdot t$$

Parameter t , called interpolant, is a value between $[0, 1]$. Formula 1 calculates the normalized interpolation between the two directions. The closer the agent is to the checkpoint to reach, the greater the influence of the direction of the next checkpoint. This way, the agent is aware both the the direction of the checkpoint to pass through and the next one.

Behaviours	
Positive	
Condition	Value
Reach valid checkpoint	+1
Completing a lap	+5
Negative	
Passing wrong checkpoint	-1
Start collision with a Wall	-0,005
Start collision with a Car	-0,005
Still in collision with a Wall	-1
Still in collision with a Car	-1

Table 2: Reward signals

Optional Behaviours	
Condition	Value
The agent is accelerating	$+0,005$
The agent is not breaking	$+0,005$
Direction of the car	$+0,005 \cdot dir$
The agent is turning	$-0,005$

Table 3: Optional reward signals

4.2.2 Actions

As described in chapter 3.2.1 Car, the agent has three values for actions:

- *moveZ*
- *moveX*
- *breaking*

All values are discrete. *moveZ* controls the acceleration; *moveX* governs the turns, and *breaking* has authority over the brakes.

4.2.3 Reward Signal

For achieving the best training possible is important to determine which behaviors are to be rewarded and, if so, to what extent, and which ones are to be punished. Table 2 provides a comprehensive list of all the reward signals employed.

In addition to the aforementioned signals, there are a series of optional additional rewards, listed in Table 3. Depending on which of these rewards is active during the training, the agent will exhibit different behaviour. In fact, three different agents have been trained, each with its own unique driving style. This was achieved precisely thanks to these optional rewards.

The reward of *direction of the car* changes based on the difference between the car's direction and the one evaluated by the formula 1. The parameter *dir* is a float value between -1 and 1. When the direction of the car perfectly aligns with interpolated one $dir = 1$, it is zero when it is perpendicular and minus one when it is in the opposite direction.
The final reward signal "**The agent is turning**" penalizes the agent every time it turns. Although it is normal for a car to turn on a track, after the first experiments, we noticed that the agent, despite constantly accelerating, would frequently rotate even when it was unnecessary, thus decreasing its speed. The penalty serves to teach the agent to turn only when necessary.

4.3 Training

The master thesis [14] explores the optimal processes for training ML-Agents on Unity within a racing video game. The research concludes that **Curriculum Learning** is the most efficient approach, offering faster average lap time and lower crashes, both in the track used during training and in tracks never seen by the agent. The thesis [14] structures the training methodology into a series of lessons:

1. Collisions with other cars and obstacles are ignored
2. Only collisions with other cars are disabled
3. All collisions are active

The agent's ultimate goal is to complete a given track in the shortest time while contending with the presence of other agents and obstacles. Following a preliminary analysis, Curriculum Learning emerges as the most effective technique, even within the specified context of this thesis. In contrast to [14], this research employs a different lesson framework. There are four lessons. Two factors, the track configuration and the number of cars, distinguish one lecture from another. To expedite the training process, rather than training a single agent, we train twelve agents simultaneously, all sharing the same network. We chose twelve agents based on the training device's computational capabilities. Each lesson exhibits distinct track layouts, depicted in Figure 13. The initial track teaches the agent how to accelerate and maintain trajectory. When a car reaches the track's end, the game moves it to the start. The second one incorporates simple turns, while still predominantly composed of straight sections. The third track introduces moderate complexity, featuring more challenging but sporadic curves. The fourth and final track primarily consists of turns. As previously stated, the other distinguishing characteristic between lessons is the number of agents that share the same track. Initially, only one car is allowed on the course. As the lessons progress, the number of players increases, with a maximum of twelve cars on the final one. In the initial circuit, the agent's sole goal is course completion. However, as the car's population increases in subsequent lessons, the agent must also navigate and avoid collisions with other participants.

4.3.1 Hyperparameters

The hyperparameters used were chosen empirically. The combination that produced the best results is reported in Table 4.

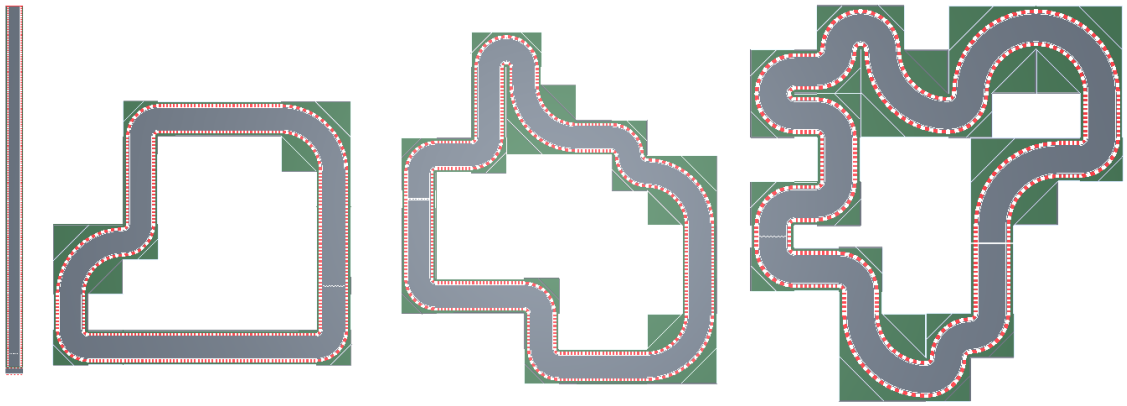

Figure 13: Training tracks

Hyperparameters	
Parameter	Value
Epochs	3
Steps	1.500.000
Steps per Lesson	300.000
Learning Rate	0,0003
Batch size	1024
Network Parameters	
Number of layers	2
Hidden units	128

Table 4: Hyperparameters

Track	Checkpoints	Total per lap
First track	41	$41 + 5 = 46$
Second track	40	$40 + 5 = 46$
Third track	40	$44 + 5 = 49$
Fourth track	47	$47 + 5 = 52$
Fifth track	41	$41 + 5 = 46$

Table 5: Maximum base reward

4.4 Result

Each neural network undergoes three rounds of training. In the first round, the weights of the network are initialized randomly. In the second round, the network starts from the results of the first round. Lastly, in the third round, the network weights are initialized using the ones from the second round.

In each graph, the **orange line** represents the results from the first part, the **blue line** represents the second part, and the **green line** represents the third part.

In all experiments, the lesson changes every 300 thousand steps.

The displayed graphs show the **average cumulative reward** value obtained by the agents in each **episode** relative to the training steps.

An episode concludes if:

- The agent completes a certain number of laps on the predetermined circuit.
- The agent collides with a wall or another player.
- The agent crosses an incorrect checkpoint.

Table 5 shows the maximum cumulative reward achievable in a lap without additional rewards. In the total calculation, the "+5" corresponds to the reward for completing a lap. For the checkpoint, since the reward for each correct checkpoint is equal to "1", the total reward value achievable in a lap corresponds to the total number of checkpoints. In each plotted graph, whenever there is a change in lesson, the trend of the cumulative reward tends to decrease and then rise again. This trend is evident in the first training sessions of every experiment where, during the transition from the second to the third lesson, around 600 thousand steps, the graph plummets. Whenever the lesson changes, a different track is encountered. The change prevents the car from learning to drive only on one track. When the agent drives on a new circuit, it takes time to readjust, causing the sudden drop in reward mentioned before. We trained a total of 4 different agent, each training differs on the optional rewards activated. Here we list all the setting used for each experiment.

4.4.1 First experiment

In the first experiment, only the acceleration reward signal among the various optional reward signals is activated. The number of laps to complete an episode is fixed at two. Figure 14 illustrates the trend of the Cumulative Reward for the three training runs.

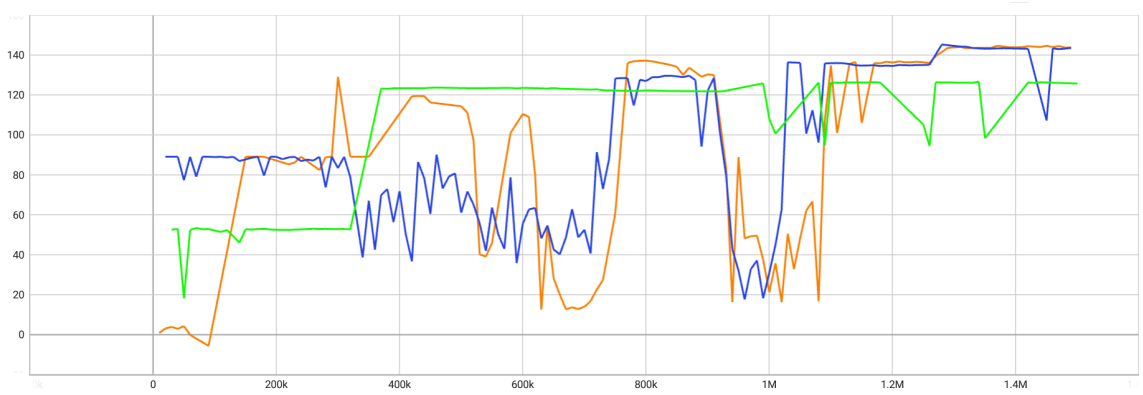

Figure 14: Cumulative Rewards first experiment

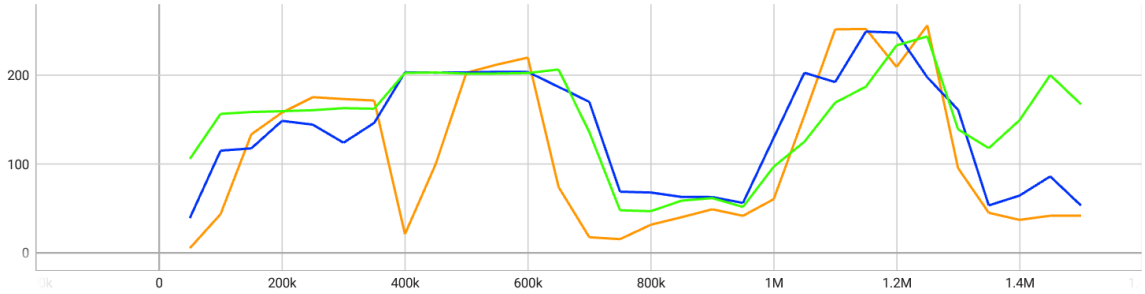

Figure 15: Cumulative Rewards second experiment

4.4.2 Second experiment

In the second experiment, the enabled additional rewards are *acceleration* and *breaking*. Figure 15 shows the trend of the cumulative reward of the described agent. The number of laps required to complete an episode is three.

4.4.3 Third experiment

In the third experiment, the enabled additional reward is *direction*. In the initial attempts, we observed that the architecture had difficulty accelerating. To solve the problem, we modified the code so that $moveZ = 1$. As described earlier, the attribute $moveZ$ controls the car acceleration. As for the second experiment, the laps number to complete an episode is three. Figure 16 reports the trend of the cumulative reward of this artificial intelligence.

4.4.4 Fourth experiment

In this experiment, we trained the agent solely on the last lesson circuit. Like in the third experiment, we set the acceleration to one. The additional reward enabled is *breaking*, and the number of laps per episode is three. This attempt's objective is to create the optimal environment for training

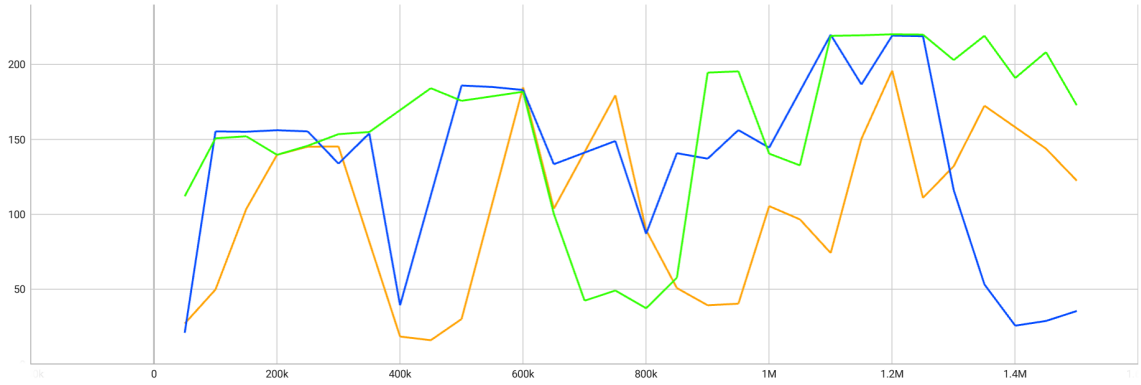

Figure 16: Cumulative Rewards third experiment

a driver to be as fast as possible. Additionally, it aims to determine if training an agent solely on one track allows it to compete on unfamiliar ones as well. The initial results of the experiment showed that the agent was slower than the one from the first experiment. After analyzing its driving style we noticed that the AI was constantly turning, causing the car to decelerate. To address this issue, we introduced an additional penalty, previously described as " The agent is turning", which decreased the reward of -0.005 each time the agent chooses to rotate. As a result, the agent now only turns when necessary. Figure 17 illustrates the training results.

4.4.5 Final Test

To evaluate the agents, we had them compete on two tracks. The first track corresponds to the one in the final training lesson, shown in Figure 19. The second track, displayed in Figure 18, is a new circuit unknown to the artificial intelligences. In each race, the agent competes alone. We consider a race completed if the car ends one lap. If the driver passes through an incorrect checkpoint or remains stuck against a wall for more than five consecutive seconds, the race ends with a negative outcome. Table 6 presents the results of this test. The displayed information, for each experiment, are the following:

- **Completed**: number of completed races compared to the total number of tries
- **Mean Time**: mean time of all completed races
- **Mean Velocity**: mean velocity of all completed races
- **Crashes**: number of crashes relative to the total number of tries

Table 6 does not show the number of failed attempts due to passing through an incorrect checkpoint since it has never occurred. Therefore, all failed attempts are due to prolonged contact with a wall. The number of crashes sometimes exceeds the number of failed attempts. In those cases, the car managed to move away from the obstacle in less than five seconds after the collision. Lastly, in the table, white rows indicate data related to the unknown track, while grey rows correspond to the known one.

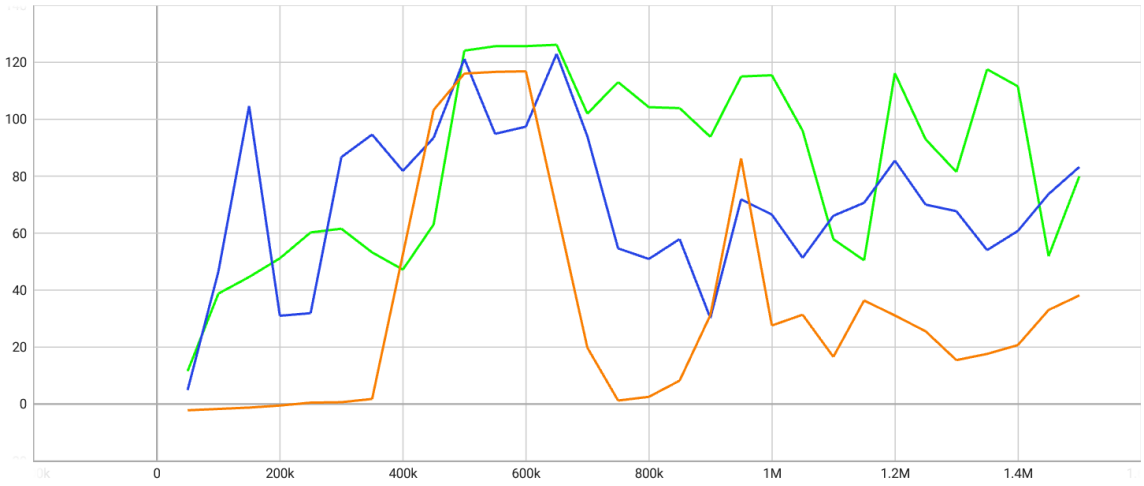

Figure 17: Cumulative Rewards fourth experiment

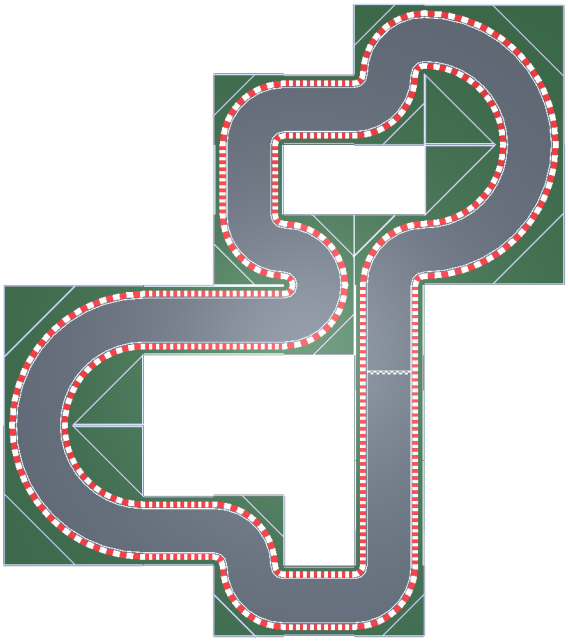

Figure 18: Unknown Track

Agent	Completed	Mean Time	Mean Velocity	Crashes
First	211/211=1.00	70.28	29.25	0/211=0.00
	243/243=1.00	61.37	31.70	0/243=0.00
Second	151/219=0.69	76.71	26.04	169/219=0.77
	122/383=0.32	64.35	26.90	300/383=0.78
Third	133/138=0.96	109.44	18.05	36/138=0.26
	155/204=0.76	89.31	18.68	88/204=0.43
Fourth	200/254=0.79	67.80	33.27	150/254=0.59
	59/444=0.13	58.48	33.46	493/444=1.11
Expert	5/5=1.00	53.25	34.42	0/5=0.00
	5/5=1.00	47.30	35.03	0/5=0.00
Rookie	5/5=1.00	68.08	29.64	0/5=0.00
	5/5=1.00	57.38	31.45	0/5=0.00

Table 6: Testing results

To accurately compare the results obtained, we have measured the performance of two human players. The first is an experienced gamer, while the second plays occasionally and rarely engages in racing games. We displayed their results in the last rows of the table under the names **Expert** and **Rookie**.

Based on the results presented, we can observe how the fourth experiment is the fastest agent overall on both the known and unknown tracks. On the unknown track, the same agent achieves the minimum completion rate. As initially hypothesized, training an agent only on one circuit is not the best choice. The second experiment has the highest number of crashes on the known track and the second-highest on the unknown. Considering the additional rewards activated, the second experiment should have achieved higher speeds and lower completion times among the top three, but it is only better than the third. The third one is the slowest among the experiments, but the number of collisions is lower than the third and fourth. The first experiment completed all the races it participated. It did not have any crashes in all attempts. The average speed and time are only lower than the fourth agent. Comparing the results with the performance of human players, no one can match the expert gamer, not so much in terms of average speed but rather in terms of time. The first agent's performance, on the other hand, is comparable to that of the rookie player.

4.5 Database Writing

With the newly trained agents, we can autonomously play various games. By storing the race information , the database for the training of the Game Engines AI is formed. Although the agents are capable of driving on unknown tracks, it was preferred to always let them play on the latest track of their training, shown in Figure 19. This choice stems from an experiment used to verify the effectiveness of MPAI-SPG. The experiment involves a video game player competing against the three agents. The agents are not as skilled as a common player, so to balance the game, the track chosen for the race is one they have already trained on. The script **RecordWriter** is responsible for writing on an external file, the race info. The stored elements are the following:

1. Car's position
2. Car's velocity

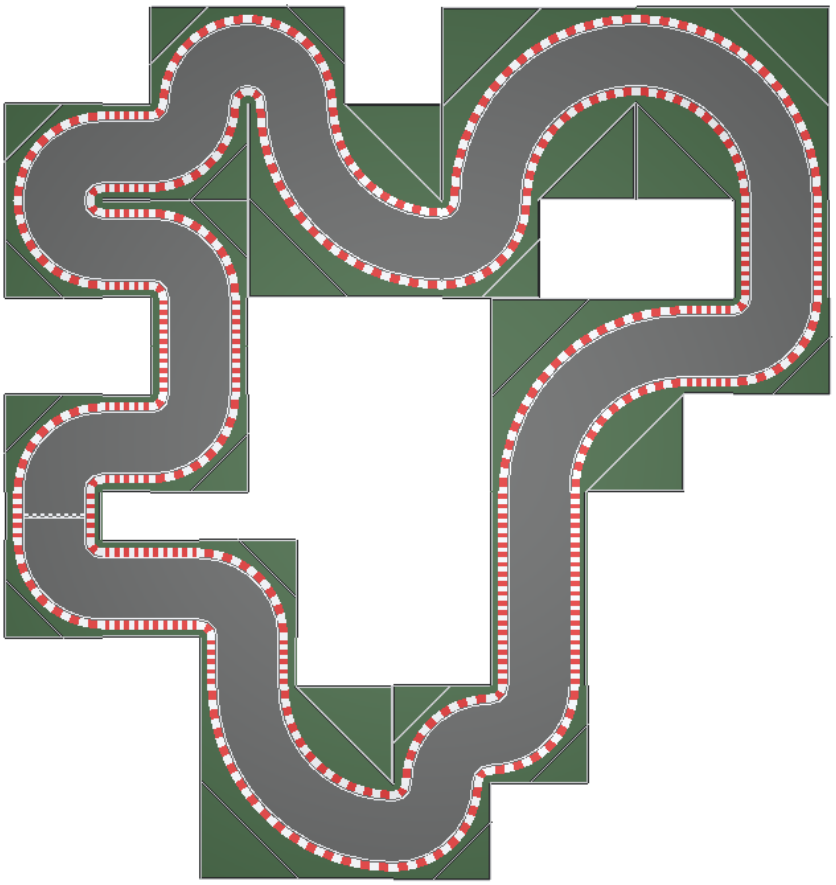

Figure 19: Race track

3. Car's rotation
4. Car's acceleration
5. Tile number
6. Tile tag
7. Relative position
8. Player last inputs
9. Car's ranking
10. Car's laps number

For all these elements, the information regarding the Y-axis is discarded, as the track does not have bumps or any general movements on that axis. With the exception of rotation. **Rotation** refers only to the rotation value on the Y-axis. The car never has to rotate along Z or X axis since the race track is flat. As described in Chapter 3.2.2, the circuit is composed of predefined square tiles. **Tile Tag** refers to the type of tile beneath the car. **Relative position** indicates the position of the car relative to the center of the tile. **Tile number** specifies the position of the tile it is on relative to the total number of tiles. The circuit in question consists of 33 tiles. If the car is on the last tile, then the tile number is equal to 32,since the first tile is equal to 0, and so on.

To speed up the writing process, we start multiple races in parallel. Each one has its own **RecordWriter**. To coordinate all the writers we defined the **Coordinator Writers** script. It creates multiple file, one for each race, for the RecordWriters to write on. Additionaly, at every Unity's FixedUpdate, which occurs every 0.02 seconds, the coordinator calls the writing function on each RecordWriter.

Each race is composed of three laps. When a car completes three laps a new race starts. The file used remains the same. The writer uses the special character '_' to differentiate the new race data from the previous one. Once the writing process finishes, a Python script reads all the files and merges them into a single file. The same special character is used in the merged file to separate each file.

5 Training of MPAI AIs

The MPAI-SPG architecture, shown in Figure 6, features three different Game Engines AI. In our case, both the **Physic** and **Behaviour Engine** predict the same information, as illustrated in Chapter 3. Therefore, instead of three engines, only two are trained. Based on what has been seen in previous chapters, the best architectures for Time Series Prediction are: MLP, LSTM, CNN, CNN & LSTM. We discarded solutions with CNN. The library used to convert Tensorflow models into ONNX files (tf2onnx), which is the extension needed to utilize the model in Unity, does not allow the use of non-square input.

To create the input dataset, we have to generate the time series. The length of the series is defined by the **Sequence length** attribute. Tensorflow, the library used to implement the AIs, provides the *timeseries_dataset_from_array* function. The Tensorflow API defines it as follows:

"Creates a dataset of sliding windows over a time series provided as an array. This function takes in a sequence of data points gathered at equal intervals, along with time series parameters

such as length of the sequences/windows, spacing between two sequences/windows, etc., to produce batches of time series inputs and targets."
We divide the initial database into subsections, where each section represents the information belonging to a single race. This operation prevents data mixing from different games. Each section is distributed into one of the three different sets: training, validation, or test set. The partition is done using a static proportion of 50% of the races for the training set and 25% for the validation and test sets. Using the *timeseries_dataset_from_array* function to generate each set, means loading the entire time series into the memory. Each input in the series is composed of **Sequence Length** elements. Therefore the set becomes heavier by a factor equal to the **Sequence Length**. Due to the physical limitations of the devices used for training, we adopt **generators**, called **DataGenerators**. In Python, a generator is a function that returns an iterator that produces a sequence of values when iterated over. Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once. DataGenerator returns the next batch with each invocation. Each batch is composed of numerous time-series equal to the value of the **Batch size** specified during the generator's creation.

5.1 Behaviour & Physic Engine

Behaviour/Physic engine predict physic information of each competitor. Those information correspond to:

- Position.
- Velocity.
- Rotation.

We divided the study of the training process into three steps:

1. Search of the best input.
2. Search of the best architecture.
3. Fine tuning of the hyperparameters.

5.1.1 Input

The engine receives as input **Game Message GM***, which is a subset of information derived from the **Game State GS** and **Controller Data CD**. From the conducted experiments, it was found that the input used is a crucial parameter for training.

In the **first experiment**, each race had four agents. The input consisted of the position, velocity, and rotation of each car in the race. Regardless of the trained architecture and combination of hyperparameters, the results were inadequate. The input data was too diverse and varied greatly from one another. Therefore, we modified the experiment by reducing its complexity.

In the **second experiment**, the number of participants per race was reduced to one, significantly decreasing the output and input of the network. The system no longer needs to consider how the presence of other agents influences each other. With this modification, the results improved but remained below expectations. Both position and velocity have a wide range of values, posing

Algorithm 1 Discard algorithm

```
pos = 0
for pos ≤ RaceLength do
    Group ← Race[pos]
    pos = pos + Discard
end for
```

difficulty for the last layer of the architecture. Attempts were made to normalize the data, but it did not help. The lower values become too small to have an influence, resulting in a high error.

In the **third experiment**, we worked with **differences** instead of absolute values. A new dataset was created by subtracting each row from the previous one. After the subtraction, each data represents how it has changed compared to the last snapshot. The same operation was also performed on the output, meaning that the Engine now predicts how the car's position, velocity and rotation changes concerning the prior game state. This calculation significantly reduces the order of magnitude of the values to predict. This technique led to better results compared to previous experiments, but has a major flaw. By implementing the model in the game, it becomes evident that the network always assumes that the car will continue along its trajectory. The sampling time is 0.02 seconds. This time interval is not large enough to cause a considerable change. Guessing that the object will continue on its own trajectory results in a negligible error during training.

To solve the problem we introduced a new parameter called *Discard*. It modifies the sampling interval without writing a new database. Given a single race, several datasets equal to *Discard* number are generated. We populate each subset by taking all rows *Discard* positions apart from each other. For example, if $Discard = 3$, the race is divided into three smaller datasets. We use the code illustrated in the Algorithm 1 for populating the first group.

To populate the additional sets, we modify the initial value of *pos*. Within a group, the time interval between each row is equal to

$$DeltaT = Discard \cdot T$$

where T indicates the sampling instant used during the writing of the initial database, which in our case is 0.02 seconds. During training, *DataGenerator* treats each subset as if it were a different race.

Another problem stems from the lack of information about the track. Previously, the car's position indicated where the agent was in the circuit. However, the relative position displays the distance vector travelled by the game object concerning the prior position. Therefore the network cannot extract useful information about the track's structure from this info. We decided to omit the position information from the input and instead introduce **Tile data**.

In Chapter 3.2, we describe how each circuit is composed of prefabricated tiles. The term *Tile data* refers to the information related to the track tiles. During the database writing process we save the following information:

- Tile number
- Tile tag
- Tile relative position

Tile tag refers to the type of tile beneath the car. **Tile relative position** indicates the car's position relative to the center of the tile. **Tile number** specifies the tile rank to the total number of tiles. For example if $TileNumber = 3$ then it is the fourth tile of the circuit. Tile data is not subtracted during the creation of the datasets.

In the **Fourth experiment**, we build the input using Tile data alongside the relative car's velocity and rotation.

The **fifth experiment** we added the controller data to the input. The theory specifies that the Physic Engine does not receive CD in the GM*, but the Behaviour Engine does.

5.1.2 Architecture

We tested two types of architectures, LSTM and MLP, for each experiment. According to the theory, LSTM should achieve better results compared to MLP.
We trained MLP to use it as a benchmark. Additionally, the training time required for MLP is much shorter than LSTM, which allows for a quick evaluation of the inputs and hyperparameters. Every architectures do not have a normalization layer, even during experiments with relative values. In the case of relative values, it is not possible to determine the absolute maximum and minimum change in speed and rotation. This impossibility arises from the native Unity components, such as the Wheel Collider, whose functioning is not visible to the programmer.

LSTM models are characterized by the **Network's depth**, which refers to how many LSTM layers are linked, and the **Units number** which controls the number of units in each LSTM layer. The activation function for the LSTMs layer is the **hyperbolic tangent** tanh. Additionally, we linked an **MLP** model at the end of the LSTM one. Initially we only attached a single Dense layer, with the same number of units as the output size, after the last LSTM layer. But after some testing we realized the attaching multiple Dense layers instead of just one helped achieving better results. A **MLP** model is characterized by the **number of Dense layers** connected and the **number of units**. The last layer is a Dense layer where the number of units is equal to the output size. The last layer is not considered in the depth of the network. The activation function of all Dense layer is the **linear** function. We attempted to insert a Batch Normalization layer between LSTM layers, but the results of such models were always worse. Additionally the LSTM layer are characterized by the **Dropout** attribute. This value represents the fraction of the units to drop for the linear transformation of the inputs. It is helpful to avoid over fitting when the model is too complex. In all the experiment involving LSTM we set $Dropout = 0.2$. We use the Keras library for implementing the LSTM layers. In the next section, where results are shown, we use the following nomenclature:

$$LSTM\ XnY\ MPL\ X1nY1$$

- X represents the depth of the LSTM network, i.e., the total number of LSTM layers
- Y represents the number of units present in each LSTM layers
- $X1$ represents the the depth of the attached MLP network, i.e., the total number of Dense layers
- $Y1$ represents the number of units present in each Dense layers except for the last one

Name	Utility
Common	
Learning Rate	It is the step size or magnitude at which the model parameters are updated during the optimization process
Epochs numbers	determines how many times the model iterates over the entire training dataset.
Sequence Length	refers to the number of elements in a sequence or time series data
Discard	modifies the time distance between two elements in a sequence
Patience	Number of epochs before decreasing the Learning Rate
MLP	
Number of Dense layers	Network's depth
Units number	Number of units per layer
LSTM	
Number of LSTMs	Network's depth
Units number	Number of units of each LSTM
Dropout rate	Dropout value

Table 7: Hyperparameters

5.1.3 Hyperparameters

Table 7 lists the most significant hyperparameters.

Learning Rate influences how quickly or slowly the model learns from the training data. A higher learning rate means larger parameter updates, potentially leading to faster convergence, but it may also result in overshooting the optimal solution. Conversely, a very small learning rate leads to slower convergence but may provide more precise updates to approach the optimal solution. Initially, the network learns quickly, while later on, the improvement becomes less significant. To aid the training, the Learning Rate changes during the process. In the beginning, it has a high value to expedite the training. Subsequently, the value decreases to assist the network in finding more precise data. The Keras callback *ReduceLROnPlateau* implements the logic. It reduces the learning rate when a metric has stopped improving. The metric used to evaluate improvement is the Mean Absolute Error MAE of the validation dataset. The function halves the Learning Rate whenever the validation MAE does not improve for several epochs equal to the parameter *Patience.* In our case $Patience = 5$. For all the experiment the starting value for the Learning Rate is 0.0001.

Epoch number indicates the number of epochs, i.e. the number of times the model iterates through the entire training dataset. Keras callback *EarlyStop* prevents the network from continuing training when when a monitored metric has stopped improving for a number of epoch equal to *PatienceStop.* In our case the observed metric is the same as the learning rate. During the training process the $EpochNumber = 100$ and $PatienceStop = 8$.

Sequence Length specifies the size of the time series. It affects the model's ability to capture temporal dependencies and make accurate predictions. Longer sequences may contain richer contextual information but can be computationally expensive to execute. On the other hand, shorter series may provide less context but are more computationally efficient. Empirically higher values of the following hyperparameters slow down the Unity game. If we want to increase the time analyzed by the model, we change *Discard.*

Discard determines the time interval between two entries in the sequence. Consequently, it also controls the activation frequency of the MPAI system. The Engine's prediction GMp pertains to the game state $\Delta T = Discard \cdot T$ away from the last data of the sequence. It means that the Engine performs a prediction every time interval ΔT.

5.2 Results

In the following chapter, we display the training and validation Loss and Mean Absolute Error (MAE) values of the most relevant models. For all of them, the input that achieved the best result is the one depicted in the fourth experiment. It consists of Tile data, relative velocity and relative rotation. The results are grouped based on the **Discard** and **Sequence length** values. The model LSTM 3\256 MLP 3\64 is present in all the groups. Testing the same model with different hyperparameters can provide valuable insights into how Discard and Sequence length affect the training process.

5.2.1 Discard = 9 & Sequence Length = 20

$Discard = 9$ entails that the temporal distance between one data and the next in the time series is equal to 0.2 seconds. Therefore, the total time window of the time series is $SequenceLength * 0.2 = 4$ seconds. Among all the architectures tried with this combination, the ones that achieved the best results are the following:

- LSTM 3\64 MLP 3\64
- LSTM 3\256 MLP 3\64

As depicted in Figure 20 and 21, the model with 256 units for the LSTM layer achieves lower values for both loss and MAE, but it stopped improving much earlier than the other model.

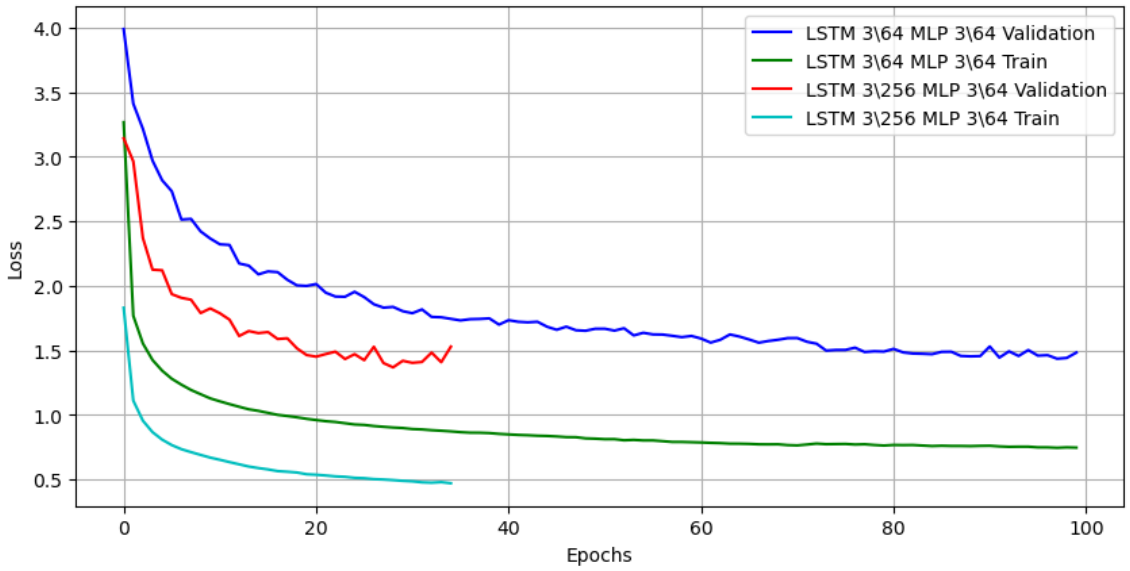

Figure 20: Loss comparison

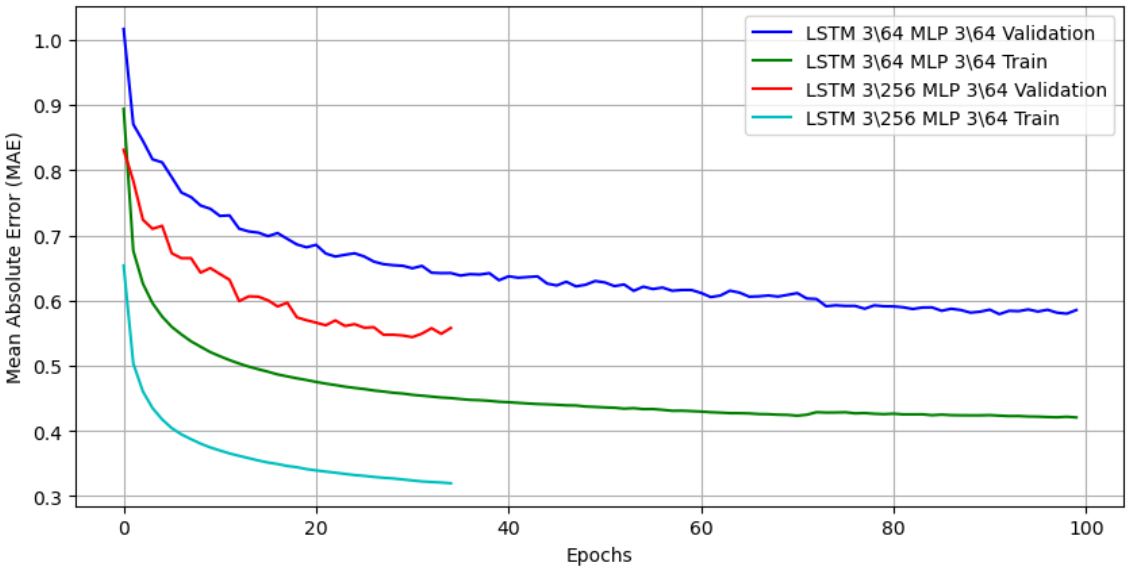

Figure 21: MAE comparison

5.2.2 Discard = 9 & Sequence Length = 40

Differently from the previous experiment, the **Sequence length** doubled, making the total time window of the time series equal to 8 seconds. The best architectures for these parameters are:

- LSTM 1\256 MLP 0
- LSTM 3\256 MLP 3\64

MLP 0 indicates that there are no additional Dense layers apart from the last one needed for the prediction. LSTM 3\256 MLP 3\64 achieved the best result for both Loss and MAE, for both train and validation set, as seen in Figure 22 and 23. By comparing the result with the previous experiment, we can see that the epoch number reached by the same model is higher, but the Loss and MAE values achieved, especially by the validation set, are much lower.

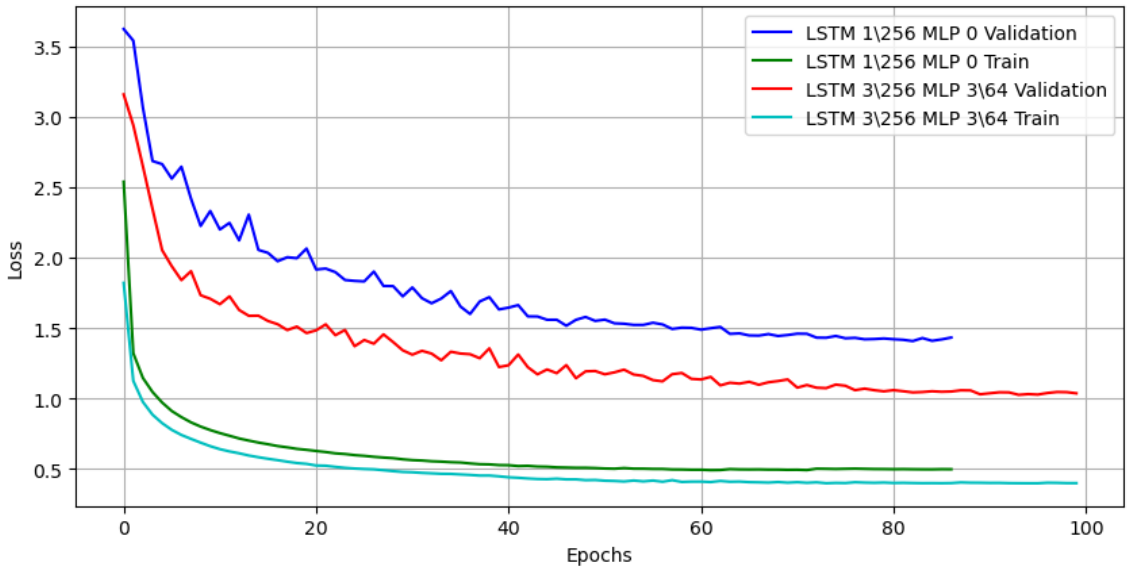

Figure 22: Loss comparison

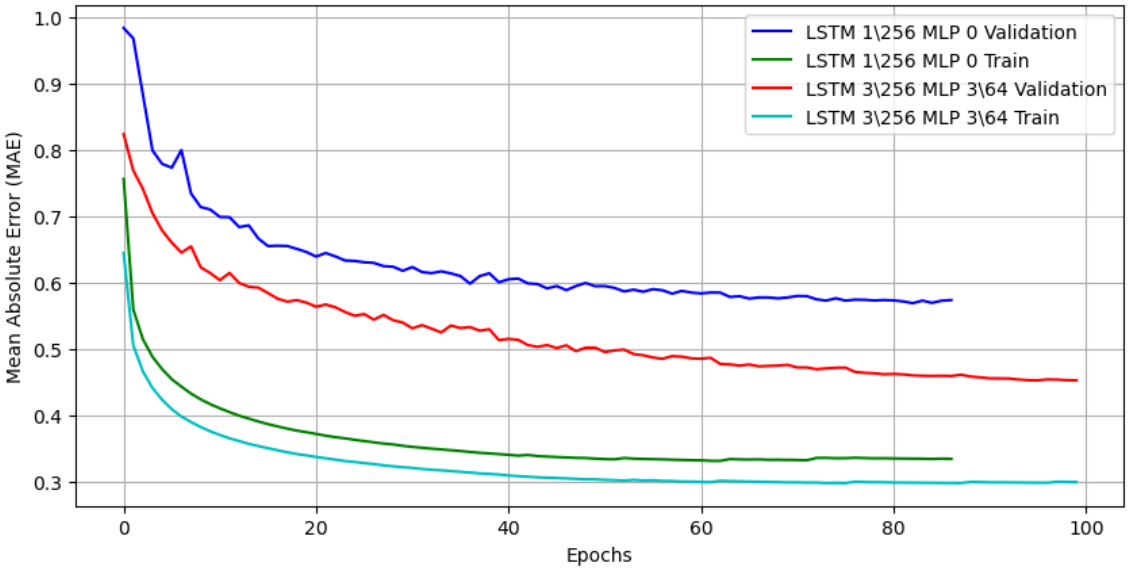

Figure 23: MAE comparison

5.2.3 Discard = 9 & Sequence Length = 50

With $SequenceLength = 50$ the time window of the time series reaches 10 seconds. We only tested one model with this setting. The reason is due to the heaviness of the model. With heaviness, we intend the computing time needed for both the training process and in the Unity context. When implementing this model in the game, the application frame rate dropped abruptly due to the numerous calculations. The results are shown in Figure 24 and 25. We can see that the validation and training results for MAE and Loss are worse than the previous trend when the $SequenceLength = 40$, but they are comparable to the ones obtained with Sequence Length of 20.

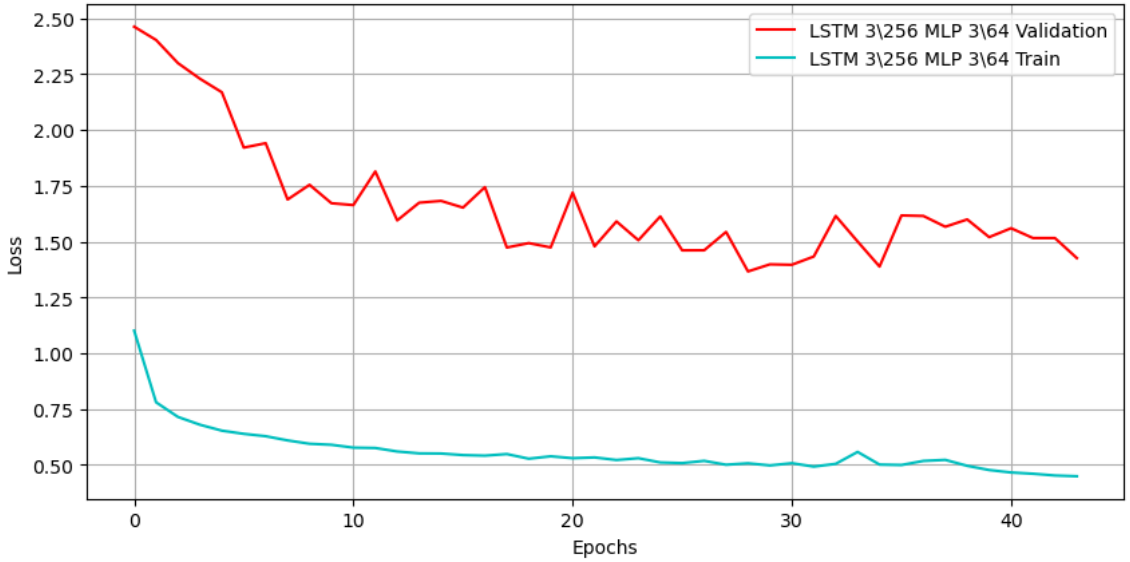

Figure 24: Loss

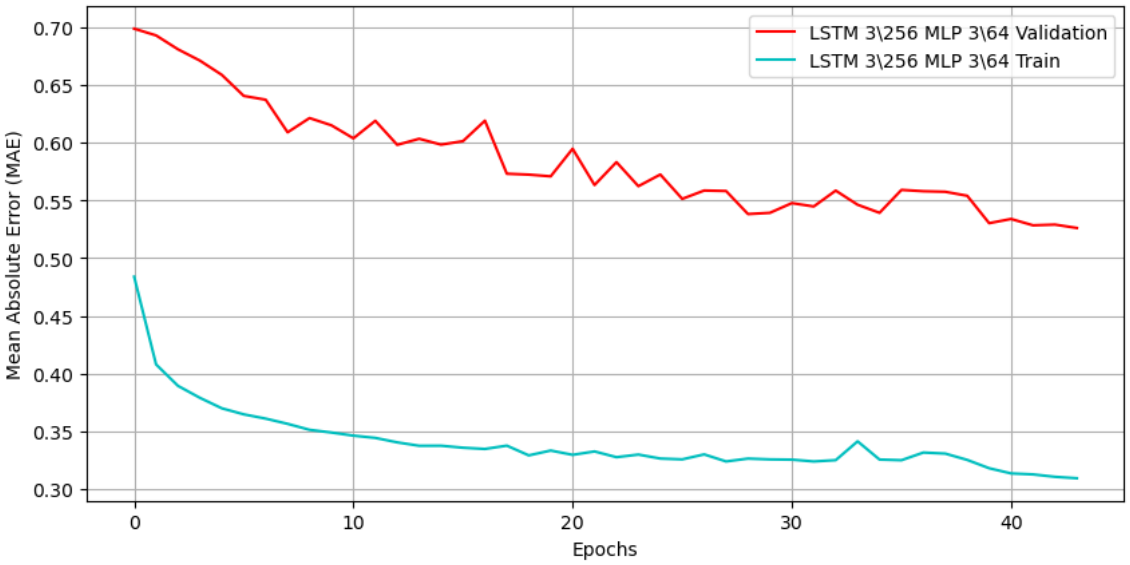

Figure 25: MAE

5.2.4 Discard = 4 & Sequence Length = 20

Changing the **Discard** means modifying the time distance between two elements in the time series. In this case the time distance is equal to 0.1 seconds, while the total time window of the series expands over 2 seconds. We can see from Figure 26 and 27 that the validation minimum values for both the Loss and MAE are the lowest among all experiments. The model predicts how much the position velocity and rotation changed concerning the previous snapshot. With this setting, we halved the time between snapshots, thus decreasing the output values. Lower values are easier for the model to predict.

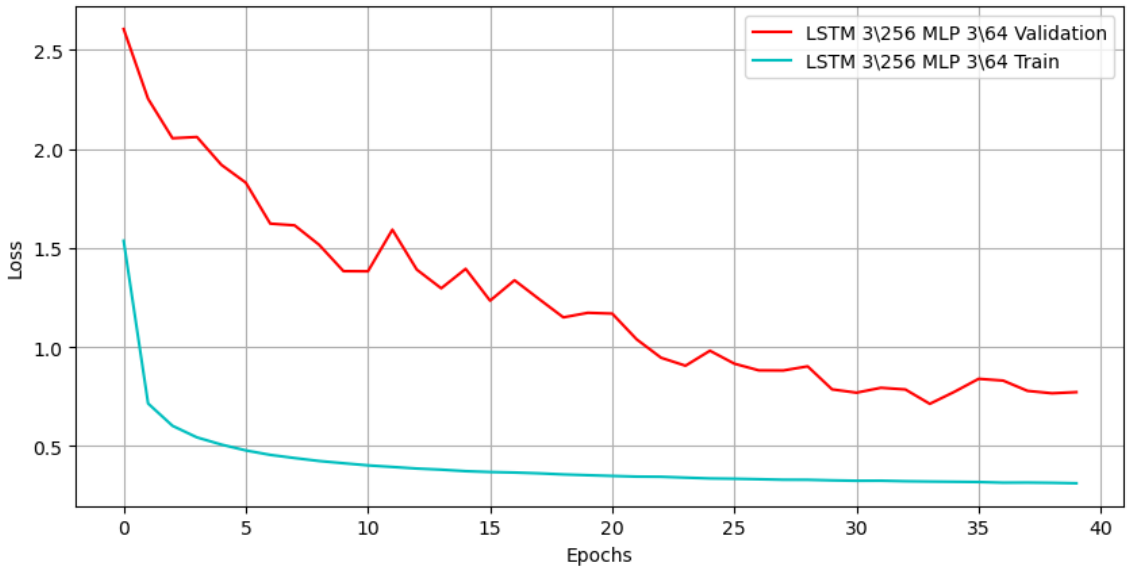

Figure 26: Loss

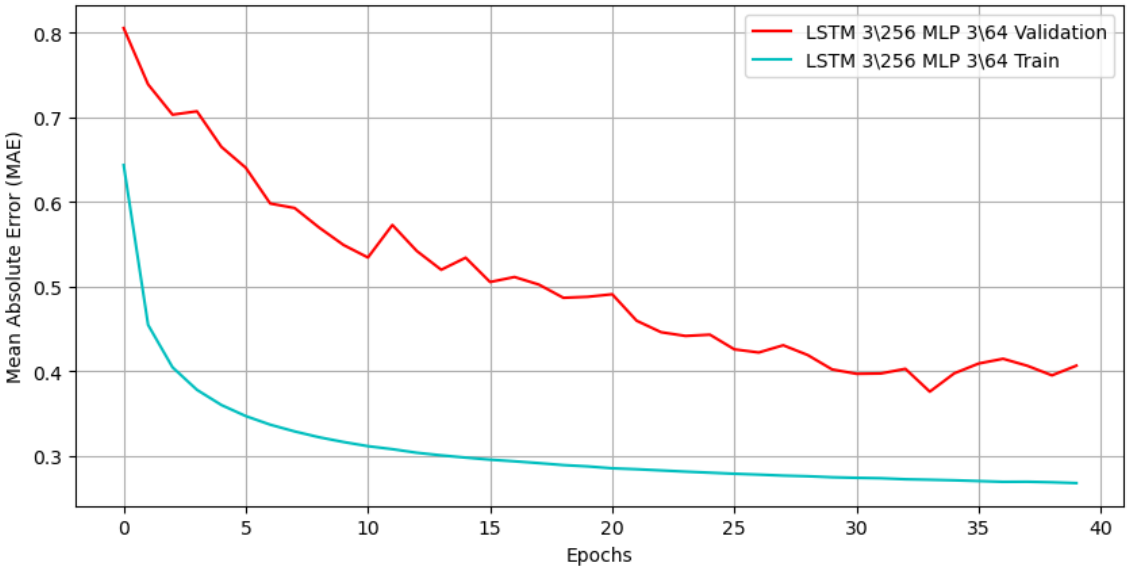

Figure 27: MAE

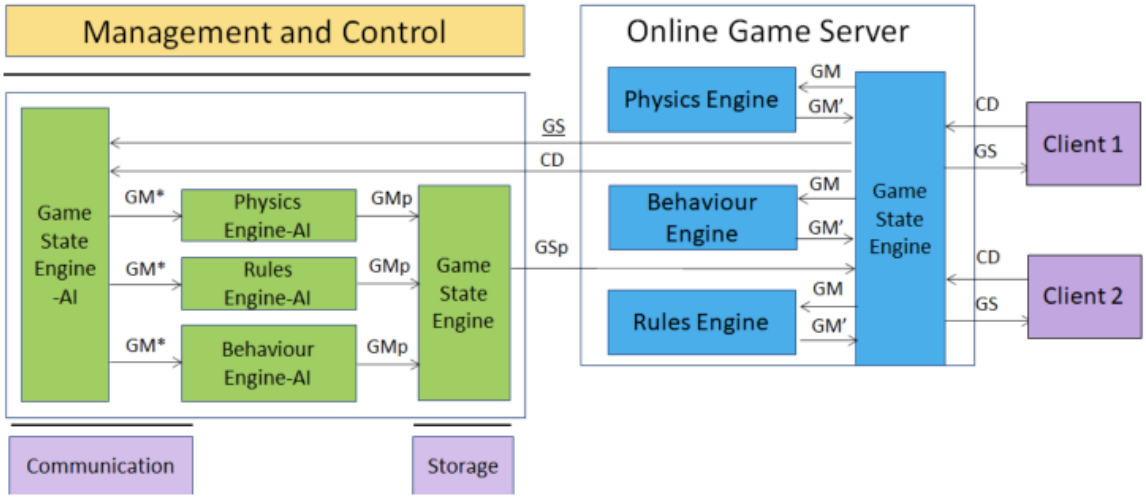

Figure 28: MPAI-SPG architecture

6 Implementation of MPAI architecture

In Chapter 3, we described the theoretical structure of MPAI-SPG, represented in Figure 28. Although the functionality remains the same, there are some differences in the Unity solution. The game had to be adapted to make it playable as a multiplayer online game. To achieve this, we used the Mirror library.

6.1 Mirror

To effectively communicate between servers and clients, Mirror provides **Commands** and **ClientRPCCalls**. A Command allows a client to execute a function on the server, while ClientRPCCalls lets the server run a method on a single or all connected clients. Another tool in the Mirror library is the so-called **SyncVar**. Mirror defines SyncVars as follows:

"SyncVars are properties of classes that inherit from NetworkBehaviour, which are synchronized from the server to clients. The server automatically sends SyncVar updates when the value of a SyncVar changes, so you do not need to track when they change or send information about the changes yourself."

Mirror implements SyncVar updates using ClientRPCCalls. For the online racing game, it is necessary to synchronize the cars ' position, velocity, and rotation. Mirror offers the **Network Rigidbody** and **Network Transform** components to complete this task. Network Transform handles the synchronization of position and rotation, while Network Rigidbody handles velocity. The functioning of these components is similar, and it works as follows. Position, rotation, and velocity are defined as SyncVars within these classes. At each frame, the server updates the values of the SyncVars by reading them directly from the server's game objects. By modifying the SyncVars on the server all clients update the associated values. When the SyncVars position, velocity, and rotation change, we modify the same attributes on the car game object.

For objects directly controlled by the player, the library advises leaving the authority to the

client. For a client to have authority means that it will be responsible for instructing the server on how to update the values of the associated SyncVars. Although this contradicts the structure of an authoritative server, where the client should not have any authority, this implementation results in a more responsive input system and smoother gameplay. Mirror suggests checking each client's commands to avoid trusting the client. When a command is deemed implausible, the server cancels its effects and notifies the client. For example, if acceleration can only vary between 0 and 1, only values within this range are considered plausible. Since **SPG** intervene every time the game state is too different from the prediction, even if we do not implement the check system, SPG covers any possibility of cheating.

In our game, the client has authority over his car: the client applies the player's inputs directly to the controlled object and then sends the new info to the server. However, if the client has authority over the position, rotation and velocity, the server cannot intervene, thus making it impossible for the SPG system to apply the prediction on the car. To solve this problem, when the server needs to correct objects controlled by the client, it temporarily takes authority over them. When the server needs to assume control, it removes the client Authority from the game object only on the server side. Taking the authority only on the server it means that the client discard any calls from the server regarding the car, while the server discard any command received by the client about the car. Therefore from the client's perspective, they are in control, but in reality, the server is driving the car and updating its position on all the other clients. While SPG is applying its prediction, the game state of the client is different from the server. After making the necessary changes to the game object, the server updates the authority giving it back to the client. Then the server sends a ClientRPCCall to the client notifying him of the car's position, rotation and velocity in the server. When the client receives the call, it changes the car attributes thus reuniting the client's game state with the server's.

6.2 PlayerGhost

As described in Chapter 6, the networks have been trained to predict the behaviour of a single machine, allowing scaling up the number of players without retraining the AIs. Therefore the MPAI-SPG logic must be implemented in a way that is independent of the number of players. The **PlayerGhost** class serves this purpose. This component stores all the necessary information for the SPG system. Specifically, the class contains the following information:

- Reference to the PlayerScript
- Reference to the GhostCar
- Input matrices for the Game Engines AI
- GM* prediction
- IWorker instance

6.3 PlayerScript

PlayerScript implements the logic regarding the game object **car**. This script is responsible for processing inputs. Additionally, when a new player connects to the server, PlayerScript starts the initialization process, such as updating the graphical interface or informing the **MPAI Manager** of the new car.

6.4 GhostCar

GhostCar script controls the game object **Ghost car**. This object is invisible to the player. Whenever the server needs to modify the game state by applying the GSp, the predicted state, the server changes the Ghost car instead of the player's. Ghost cars allow for the separation between the game logic and the MPAI-SPG one. GhostCar can operate in two modes: **testing** and **real-case**. In the case of **testing**, when the SPG system is in action, the ghost car is made visible on the server. The player's car continues to be controlled by the client, while the ghost car is driven by the server, based on the GSp predictions. When the system is inactive, the car remains invisible and follows the rotation, speed, and position of the client's car. In the **real-case** scenario the car is always invisible. When the game state needs to be changed, due to the discrepancy between GS and GSp, GhostCar takes the authority from the client and applies the prediction to the real car as explained before. Otherwise, it ensures that the authority remains with the client. The **testing** mode allows comparing the solution with the player commands.

6.5 IWorker

The **IWorker** instance derives from the **Barracuda** library, which is responsible for implementing artificial intelligence in Unity. Barracuda allows the usage of external ONNX model types in Unity, unlike ML-Agents, which trains the agent directly within the Unity environment. The IWorker interface is responsible for the execution of neural networks, which is asynchronous and non-blocking. **Coroutines** are employed to read the output without blocking the main thread. Unity's manual [19] describes coroutines as follows: "A coroutine allows you to spread tasks across several frames. In Unity, a coroutine is a method that can pause execution and return control to Unity but then continue where it left off on the following frame." In every frame, the completion of the execution is checked. When the output is ready, it is saved in the PlayerGhost GM* variable. The IWorker interface is associated with PlayerGhost, enabling the simultaneous execution of multiple neural networks.

6.6 Dispatcher

The Dispatcher corresponds to the Game State Engine AI shown in Figure 28. Its objective is to generate the GM*, or game messages, for every Game Engine AI. As explained in the previous chapter, instead of using absolute values for position, velocity, and rotation, we use the difference. Therefore, the Dispatcher retrieves the current data and information regarding the previous sampling moment and performs the subtraction. Like GhostCar, this component can also work in **testing** or **real-case** mode. The difference lies in how we retrieve the data. In the case of **testing**, the script takes the data from the actual machine. In the **real-case** scenario, the component reads the data from the ghost car. The first mode allows us to understand the neural network quality when we fed them using real data like it was during training. The second mode, on the other hand, demonstrates the behaviour of the network when it is fed with the predicted data. It allows us to observe how the error propagates. When the data is ready, the Dispatcher passes the values to the respective Game Engine AIs.

Algorithm 2 Confront algorithm testing mode

min = Random number between 0 and 8
$max = min + 2$
$time = GameTime\%10$
if $time \leq max$ & $time \geq min$ **then**
  return $true$
end if
return $false$

6.7 Game Engines AIs

Game Engines AI updates the input matrices. For each car, the Engine adds the new data as the first element of the input matrix. If the matrix size exceeds the Sequence length, it will remove the data present at the last position. Next, we check again the matrix size. If it is equal to the Sequence length, the Engine invokes the Prediction function of **PlayerGhost**. This function initiates the prediction process using the **IWorker** interface.

6.8 Collector

The **Collector** is the equivalent of the Game State Engine shown in Figure 28. When invoked, it reads the output of the Game Engines AI (GMp) and compares them with the game state evaluated by Unity's physics system. If the comparison returns positive, it applies the prediction on the ghost car; otherwise, it updates the ghost car by copying the real one. The component can work in two modes: testing or real-case. The mode changes the algorithm used for the comparison function. In the case of testing, the comparison function works as shown in Algorithm 2. The values of *min* and *max* are calculated during the setup of a new client and are different for each car. The operation % corresponds to the module operation, which divides the first element by the second and returns the remainder. In summary, the algorithm returns positive for three consecutive seconds every ten. For the real-case scenario, the method evaluates the mean absolute error of the real values with the predicted ones. When the error is higher than a defined threshold, it returns true.

6.9 Manager MPAI

The last component is **Manager_MPAI**. This component is not present in the MPAI-SPG architecture diagram. Manager_MPAI is responsible for coordinating the SPG components. In the previous chapter, we demonstrated how the **Discard** parameter modifies the time distance between one data and another within the Engine AI input matrix. Therefore, the SPG system does not function at every frame but rather at every $\Delta T = Discard \cdot T$, where T corresponds to the **Time.fixedDeltaTime**. Unity Scripting API defines Time.fixedDeltaTime as: "The interval in seconds at which physics and other fixed frame rate updates (like MonoBehaviour's FixedUpdate) are performed." The jobs of the Manager_MPAI are to:

1. Invoke the **Dispatcher** every ΔT to update the input matrices.
2. Call the **Collector** after $\Delta T - 1$ to evaluate the predicted game state.

Algorithm 3 Manager_MPAI

```
counter = counter + 1
if counter == (Discard − 1) then
    Collector()
else
    if counter == Discard then
        Dispatcher()
        counter = 0
    end if
end if
```

The reason for waiting for $\Delta T - 1$ stems from Unity's physics system. Unity updates the physics at the end of each Fixed Update, which we have seen occur every $Time.fixedDeltaTime = 0.02$ seconds. Any alterations made by the Collector become visible in the subsequent fixed frame. For the Dispatcher to see those changes, Manager_MPAI calls the Collector at the previous fixed frame. Algorithm 3 shows the functioning principle of Manager_MPAI.

7 Results

Now that the game is set we can test the quality of the SPG system during a match. For the evaluation, we prepared two different experiments. A car races for multiple laps. During the race, we set the **Collector** component in **testing** mode, which applies the prediction for 3 seconds every 10. When SPG applies the prediction another script called **EvaluateMPAI** evaluates the difference between the position, velocity and rotation of the **real car** and the **ghost car**. The two experiments differ based on the **Dispatcher**'s operating mode. In the first experiment, the Dispatcher works in **testing** mode, which means that the matrix input is populated using the agent's car info, while in the second experiment, the Dispatcher works in **real-case** scenario, where the matrix input is populated using the ghost's car info. The models tested are the same illustrated in the chapter 5.2. Table 8 displays the results of the tests. We evaluated the models based on the average error in position, velocity, and rotation. Additionally, we evaluate the Mean Absolute Error MAE by taking the mean between all the errors. For visual convenience, the value of MAE is highlighted in green when the Dispatcher's operating mode is **testing**, red otherwise. **Sequence Length** of 20 and a **Discard** value of 9 achieved the best outcome. During the training phase, models with $Discard = 4$ resulted in the lowest **MEA** and **Loss**. In these experiments, the **testing MEA** is the smallest of all models, but the textbfreal-case MEA is significantly higher. Upon analyzing the recorded game, we discovered that the model consistently predicted that the car velocity mostly remains unchanged. During the race, when the car was turning and the **SPG system** was applying its prediction, the **ghost car** always proceeded in a straight path. The model's performance suffers greatly in both **testing** and **real-life** scenarios when the **Sequence Length** is set to 50. The game's frame rate dropped to 5 frames per second during the match. Despite reducing the sequence length to 40, the outcomes remain scarce. During training, the inclusion of a longer time series helped the model gain a better understanding of the car's behaviour. However, in a real game, more data implies a higher discrepancy from the training scenario, making it challenging for the model to generate accurate predictions.

Table 8: Results

Begin of Table							
LSTM	MLP	SL	Discard	Val MAE	Type	Attribute	Value
3\256	3\64	20	9	0.544	Testing	Pos	1.07
						Rot	0.52
						Vel	1.43
						MEA	1.00
					Real Case	Pos	4.00
						Rot	0.51
						Vel	3.76
						MEA	2.76
3\64	3\64	20	9	0.579	Testing	Pos	1.02
						Rot	0.57
						Vel	1.35
						MEA	0.98
					Real Case	Pos	3.76
						Rot	0.57
						Vel	3.70
						MEA	2.68
3\64	12\64	20	4	0.376	Testing	Pos	0.83
						Rot	0.60
						Vel	0.96
						MEA	0.80
					Real Case	Pos	5.32
						Rot	0.64
						Vel	5.28
						MEA	3.75
3\256	3\64	40	9	0.453	Testing	Pos	1.93
						Rot	0.55
						Vel	1.87
						MEA	1.45
					Real Case	Pos	7.04
						Rot	0.57
						Vel	6.06
						MEA	4.55
3\256	3\64	50	9	0.526	Testing	Pos	8.09
						Rot	0.58
						Vel	6.33
						MEA	5.00
					Real Case	Pos	17.35
						Rot	0.61
						Vel	15.58
						MEA	11.18

Continuation of Table 8

LSTM	MLP	SL	Discard	Val MAE	Type	Attribute	Value
1\256	None	40	9	0.569	Testing	Pos	1.28
						Rot	0.46
						Vel	1.40
						MEA	1.04
					Real Case	Pos	4.55
						Rot	0.47
						Vel	4.18
						MEA	3.07

End of Table

8 Field test

MPAI-SPG aspires to enhance the gaming experience by mitigating latency and preventing cheating. To evaluate how our solution aids in achieving this objective, we have conducted tests with other people. Specifically, we aim to verify whether a person can perceive the effect of SPG and to what extent.

The experiment entails having a person play against three machines controlled by artificial intelligence. Each race consists of 2 laps. Each race is associated with a so-called **Latency level**. During the game, we activate the SPG system with a frequency and duration defined by the **Latency level**. The idea is to simulate how often and how much SPG predictions are used based on the client's latency. When MPAI-SPG controls a car, the driver's client discards any information about itself from the server, thus creating a different game state concerning the server. When SPG stops controlling the car, the driver's client is forced to realign with the server by copying the server's game state. Since we do not mask in any way the realigning, testing if the SPG system is noticeable when predicting the player is useless. Therefore the simulated latency is only applied to the AI-controlled cars. Table 9 lists the latency levels used.

The test's participant plays six races, two for each **Latency level**. After each race, the player must complete a questionnaire. The questions are as follows:

1. How would you rate your gaming experience in this race?
2. How would you rate the responsiveness of the inputs?
3. Did you experience any visual glitches during the race?
4. Did you experience sudden changes in the position of the other cars?
5. Did you experience sudden changes in the speed of the other cars?
6. Did you experience sudden changes in the direction of the other cars?

The participant answers each question by rating them from 1 to 5, where 1 is associated with a negative experience for the first two questions, while for the other questions, it indicates how often they see glitches or sudden changes. Before starting the experiment, the player plays a single race alone to become familiar with the controls and understand the game's dynamics. During this race, the MPAI-SPG system is active. We set SPG in the same manner described in the previous chapter for the **real case** experiment. Predictions are applied for 3 seconds every 10 seconds only to the ghost car, which is invisible to the player. During these activations, we calculate the differences in position, speed, and rotation between the two cars. In total, a dozen people participated in this test. Figure 29 reports the mean between all the answers received.
We can see that both responsivity and experience have the same response independently from the Latency level of the race. The first level has the lowest value in terms of glitches and sudden changes, which was expected since MPAI-SPG was never activated, but some players still have perceived problems due to the agent's driving capability. At times, the agents are not capable of driving correctly against the player, making unexpected driving choices. The same problem is also present for the other latency levels: predictions can move the agent in an unforeseen position, making it difficult for the AI to drive. Higher latency levels mean more predictions, which causes more driving errors, inducing a decrease in the total visibility time of the cars with higher latency levels. Figure 29 shows that Latency levels 2 and 3 have the same responses. Figure 32 shows the

Level	Frequency (s)	Duration (s)
L1	None	None
L2	10 ± 2	0.3
L3	8 ± 2	0.6

Table 9: Latency levels

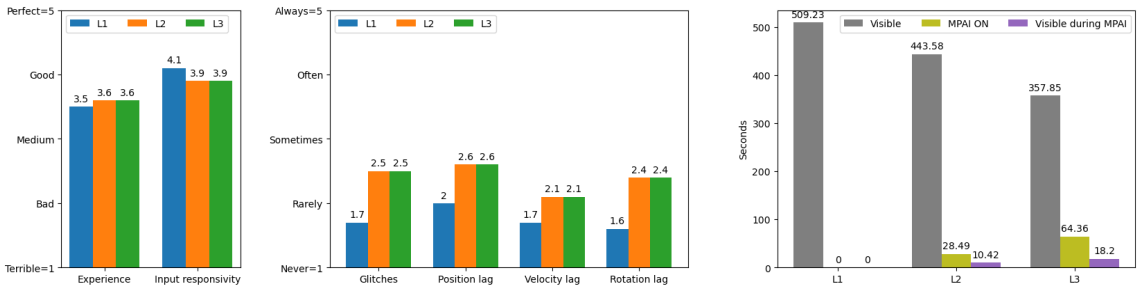

Figure 29: Questionnaire results

position error done by MPAI-SPG with Latency levels of L2 and L3. The difference is not very noticeable, making it difficult to perceive it during gameplay. While we anticipated more glitches with latency level 3 due to its higher frequency and duration, we found no difference compared to the prior latency level. This suggests that the number of glitches is not associated with the frequency, but rather solely dependent on the accuracy of the prediction.

Figure 30 shows the mean prediction error over 3 seconds achieved on the player and the agents. Position and velocity have a higher difference, while for rotation they are comparable. To evaluate the predictions' quality, we normalized the error based on the maximum error possible. For rotation, the highest error corresponds to ± 180 degrees. The maximum velocity reachable for a car is 40 units/seconds. For the position, since we do not change the car position directly, it depends on the velocity. The maximum error occurs when the prediction thinks that the car stopped while it is going at full speed. Over the course of three seconds, the possible max error increases. Figure 31 displays the normalized error for position, velocity and rotation. Velocity is the worst element to predict, which was expected. In a racing video game, this parameter is the fastest to change and, therefore, harder for the AI to predict.

Attribute	L2	L3
Position	1.02	2.63
	0.46	1.07
Velocity	4.51	7.07
	2.00	3.07
Rotation	0.58	0.59
	0.56	0.57

Table 10: Latency results

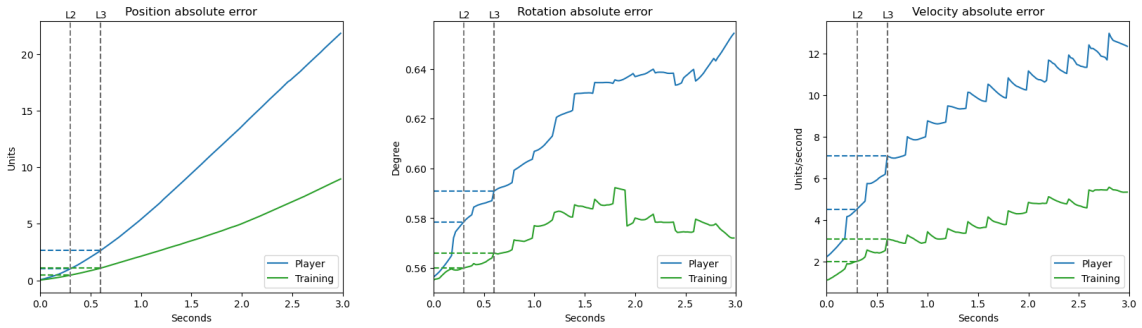

Figure 30: Absolute errors

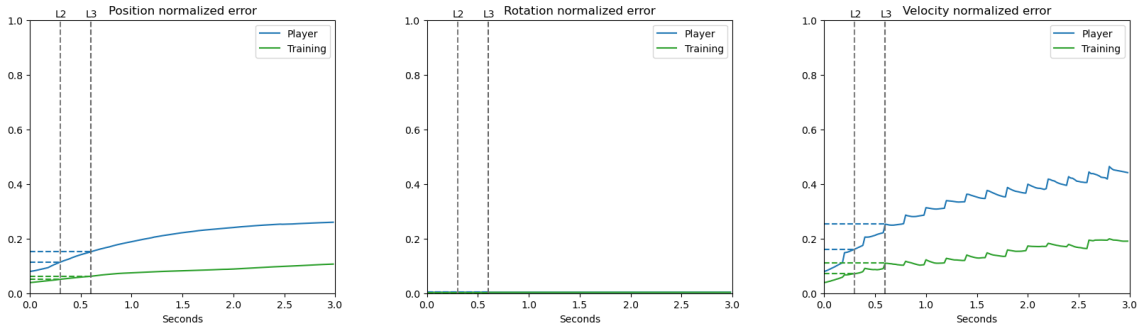

Figure 31: Normalized errors relative to the maximum possible error

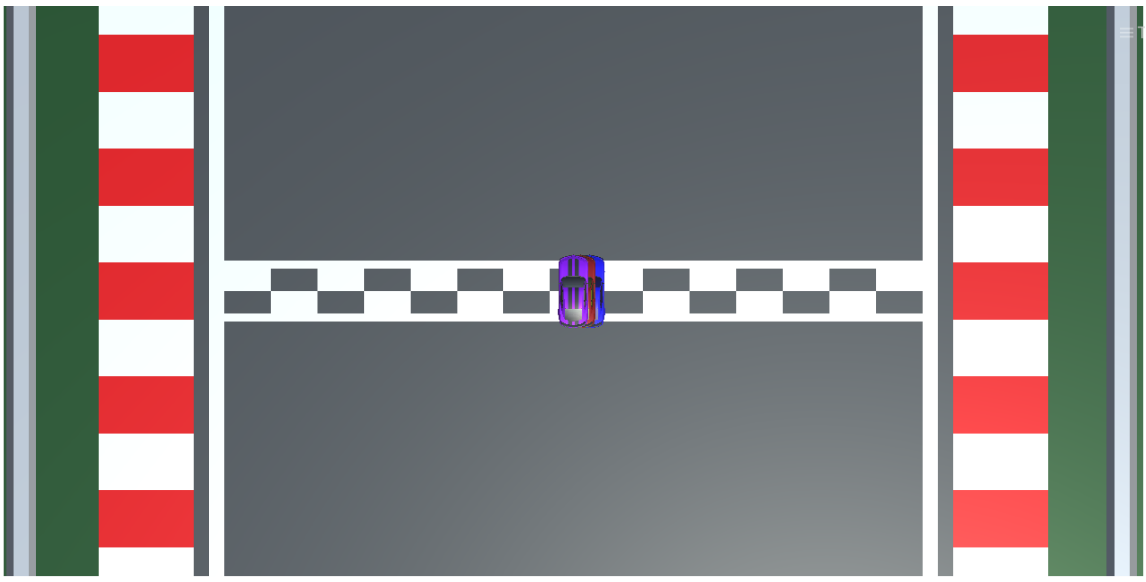

Figure 32: Position difference between Real car, L2 and L3

9 Conclusion and future works

In this Master's thesis, we investigated the implementation and practical application of MPAI-SPG in an online racing game. To accomplish this, we utilized the Unity physics system to develop a realistic car controller and a variety of race tracks. We created different autonomous drivers to write a vast dataset on which we trained an architecture capable of predicting game states. Subsequently, we integrated the MPAI-SPG system into our game, making necessary adjustments to align with the project requirements. To assess the quality of these predictions, we compared them with bot agents and player driving. Lastly, we allowed other players to engage with the project to evaluate the effectiveness of the proposed solution. The results show that the MPAI-SPG solution is a valid option for latency mitigation and cheating prevention. Players valued their experience with MPAI-SPG as positive as when it was not present, even if some glitches were visible during some matches. Moving forward, achieving more accurate predictions by identifying a more precise neural network model will be crucial. Having accurate predictions makes it possible to use this solution commercially, adding an interpolation technique to mask even further any discrepancies.

10 Ringraziamenti

Questa tesi è stata possibilie grazie al supporto di tutte le persone che mi sono state accanto.

Voglio ringraziare in primo luogo, la mia fidanzata, Alessandra, senza la quale non avrei mai avuto la forza di poter affrontare una sfida del genere. Mi è sempre rimasta accanto aiutandomi in questo percorso.

Ringrazio la mia famiglia: Mamma, Papà, Domenico, Fabrizio, Paolo, Miriam, Giacomo e Damiano, che mi hanno supportato e aiutato con i loro consigli.

Voglio ringraziare i miei amici di Torino: Giovanni, Giuseppe, Marco, Andrea Maria, Enrico, Giovanni, Alex, Rocco, Lauge, Fab e Doriana, con i quali ho condiviso momenti meravigliosi.

Ringrazio i miei amici di Pescara: Federico, Stefano, Alessia, Manuel, Caterina, Viz, Paolo, Alessandra, Mattia e Francesco che mi sono sempre rimasti accanto.

Voglio ringraziare il mio coinquilino preferito e non che amico Emiliano, con il quale ho vissuto insieme gli anni più belli nella mia vita a Torino.

Ringrazio Giancarlo, Cristina, Ludovica, Francesco, Giordano, Sara e Edith, con i quali ho condiviso esperienze e cene indimenticabili.

Infine voglio ringraziare la Comunità Capi del Pescara 9, che mi ha accompagnato sin da quando avevo 8 anni ad oggi.

Grazie a tutti.

References

[1] Yahn W. Bernier. "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization". In: *Game Developers Conference* 98033 (2001).

[2] Elias P. Duarte, Aurora T.R. Pozo, and Pamela Beltrani. "Smart Reckoning: Reducing the traffic of online multiplayer games using machine learning for movement prediction". In: *Entertainment Computing* 33 (2020), p. 100336. ISSN: 1875-9521. DOI: https://doi.org/10.1016/j.entcom.2019.100336. URL: https://www.sciencedirect.com/science/article/pii/S1875952119300552.

[3] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* (1980). DOI: https://doi.org/10.1007/BF00344251.

[4] Gabriele Gambetta. *Fast-Paced Multiplayer*. 2001. URL: https://gabrielgambetta.com/client-server-game-architecture.html.

[5] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[6] Stefan Hoermann, Martin Bach, and Klaus Dietmayer. "Dynamic Occupancy Grid Prediction for Urban Autonomous Driving: A Deep Learning Approach with Fully Automatic Labeling". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 2056–2063. DOI: 10.1109/ICRA.2018.8460874.

[7] Emil Larsson. "Movement Prediction Algorithms for High Latency Games : A Testing Framework for 2D Racing Games". In: Bachelor of Computer Science 2016-06 (2016), p. 37.

[8] Wai-Kiu Lee and Rocky K. C. Chang. "Evaluation of lag-related configurations in first-person shooter games". In: *2015 International Workshop on Network and Systems Support for Games (NetGames)*. 2015, pp. 1–3. DOI: 10.1109/NetGames.2015.7382997.

[9] Shengmei Liu, Xiaokun Xu, and Mark Claypool. "A Survey and Taxonomy of Latency Compensation Techniques for Network Computer Games". In: *ACM Comput. Surv.* 54.11s (Sept. 2022). ISSN: 0360-0300. DOI: 10.1145/3519023. URL: https://doi.org/10.1145/3519023.

[10] Regan L. Mandryk and Carl Gutwin. "Perceptibility and Utility of Sticky Targets". In: *Proceedings of Graphics Interface 2008*. GI '08. Windsor, Ontario, Canada: Canadian Information Processing Society, 2008, pp. 65–72. ISBN: 9781568814230.

[11] Regan L. Mandryk and Carl Gutwin. "Perceptibility and Utility of Sticky Targets". In: *Proceedings of Graphics Interface 2008*. GI '08. Windsor, Ontario, Canada: Canadian Information Processing Society, 2008, pp. 65–72. ISBN: 9781568814230.

[12] Takato Motoo et al. "Client-Side Network Delay Compensation for Online Shooting Games". In: *IEEE Access* 9 (2021), pp. 125678–125690. DOI: 10.1109/ACCESS.2021.3111180.

[13] Lothar Pantel and Lars C. Wolf. "On the Suitability of Dead Reckoning Schemes for Games". In: *Proceedings of the 1st Workshop on Network and System Support for Games*. NetGames '02. Braunschweig, Germany: Association for Computing Machinery, 2002, pp. 79–84. ISBN: 1581134932. DOI: 10.1145/566500.566512. URL: https://doi.org/10.1145/566500.566512.

[14] Alessandro Picardi. “A comparison of Different Machine Learning Techniques to Develop the AI of a Virtual Racing Game”. MA thesis. Politecnico di Torino, 2021.

[15] Cheryl Savery et al. “The effects of consistency maintenance methods on player experience and performance in networked games”. In: Feb. 2014, pp. 1344–1355. DOI: 10.1145/2531602.2531616.

[16] Rocket Science. *RL Netcode & Lag explained - Rocket Science #16*. Youtube. 2018. URL: https://www.youtube.com/watch?v=c373LsgiXBc&t=665s.

[17] Xingjian Shi et al. *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*. 2015. arXiv: 1506.04214 [cs.CV].

[18] Richard H. Y. So and Michael J. Griffin. “Compensating lags in head-coupled displays using head position prediction and image deflection”. In: *Journal of Aircraft* 29.6 (1992), pp. 1064–1068. DOI: 10.2514/3.46285. eprint: https://doi.org/10.2514/3.46285. URL: https://doi.org/10.2514/3.46285.

[19] *Unity Documentation*. URL: https://docs.unity3d.com/Manual/index.html.

[20] Greger Wikstrand. “High and Low Ping and the Game of Pong Effects of Delay and Feedback”. In: 2005. URL: https://api.semanticscholar.org/CorpusID:8664339.

[21] Amir Yahyavi, Kevin Huguenin, and Bettina Kemme. “AntReckoning: Dead reckoning using interest modeling by pheromones”. In: *2011 10th Annual Workshop on Network and Systems Support for Games*. 2011, pp. 1–6. DOI: 10.1109/NetGames.2011.6080977.