Università degli Studi di Torino
Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica
Indirizzo "Realtà Virtuale e Multimedialità"
Anno Accademico 2021/2022

Tesi di Laurea

MPAI-SPG: an architecture for game server support in network communication issues and cheating detection

Relatore: Maurizio Lucenteforte
Co-Relatore Interno: Davide Cavagnino
Co-Relatore Esterno: Marco Mazzaglia
Co-Relatore Esterno: Leonardo Chiariglione

Candidato:
Giorgio Gamba
Matricola 833313

A mio padre.

Contents

Chapter 1

Abstract

In this Master's thesis are collected the experiences I made during my participation to the "MPAI-SPG" project, a research work done for the MPAI Standard Commitee with the collaboration of the Turin University's Computer Science Department (represented by professors Davide Cavagnino and Maurizio Lucenteforte) and Synesthesia (represented by professor Marco Mazzaglia).

Project's name stands for "Server Predictive Game", and has the objective to build an authoritative server architecture able to predict client's commands, and in general the game behavior, in order to avoid network latency problems and players cheating. This research project adopts as Use Case the infamous videogame "Pong", rebuilt in an online manner implementing a classic Authoritative Server architecture. Given this application, neural networks for automatic players movement have been developed, enabling me to collect a huge dataset of game states and create a training set. This collection has been used to train and evaluate a set of neural networks able to make predictions. A huge experimentation on which kind of neural architecture best suits the problem together with hyperparameters tuning has been done and, once trained, these networks have been inserted inside the Pong Online project. In order to use them, I implemented the components introduced by SPG and I created a real application of the Predictive architecture. Once this system has been developed, I studied its behavior in order to proceed with the MPAI-SPG research.

Chapter 2

MPAI Commitee

MPAI is a standard committee based in Geneva with the aim to create new technological specifications in Computer Science using Artificial Intelligence in many fields of research, like Video Compression (MPAI-EVC), Context-based Audio Enhancement (MPAI-CAE), Human-Machine conversation (MPAI-MMC) to cite some of them.

As explained in [8], MPAI ("Moving Picture, Audio and Data Coding") is a non-profit organization formed by many collaborating research groups with the objective to use data (generic, or from automotive, media, health) in an efficient way, in order to develop technical specifications in the fields of Audio, Video and Data Coding. It is inspired by the MPEG work performed during the past years, but, this time, MPAI wants to fill the gap between the standard specifications made by the commitee and the practical use of this instruments inside everyday technology, definining IPR Guidelines like, for example, Framework licenses.

The MPAI Organization is formed by many different entities, where the most important is the "General Assembly" (GA), formed by Principal Members (with the right to vote) and the Associate Members (not able to vote, but able to partecipate to the development of the Technological Specifications). Another body of the MPAI Organization, the "Boards of Directors" (BD), whose president is Leonardo Chiariglione, is the one calling the monthly General Meeting of the General Assembly. Other bodies inside the Commitee are the "Advisory Committees", the "Standing Committees", the "Development Committees" (DC) and the "Secretariat".

2.1 The MPAI Standard workflow

Given the set of research groups working on different fields (SPG belongs to them), each one of them has to follow a rigid development process in order to define a standard. This process is structured in a (6+1)-steps workflow as follows:

1. "Interest collection": collection of use cases
2. "Use Cases": Proposal of use cases, together with their description and merge with compatible and analogous cases
3. "Functional Requirements": functional requirements are extracted by the research group. These will have to be satisfied by the standard and the use case implemented
4. "Commercial Requirements": the Framework License will have to be developed and approved
5. "Call for technologies": a document in prepared introducing to the standard, looking for companies able to develop compatible technologies that satisfy commercial and functional requirements
6. "Standard development": the standard is developed by a specific Committee
7. "Community comments": the standard developed at the previous stage is published to the community in order to receive comments
8. "MPAI Standard": the commented and revised standard is approved by the General Assembly and made accessible to everyone

A single research project goes to the next stage by General assembly approval, following the monthly Assembly where each research group talks about its improvements. At the moment of the writing of this thesis, SPG is at the "Functional Requirements" step, defining a prototype able to explain the aim and the funcionalities researched.

Chapter 3

Libraries

In the following sections, the libraries and frameworks adopted during my research are introduced. Except the Unity IDE, all of them concern Artificial Intelligence, applied in different contexts.

3.1 ML-Agents

ML-Agents ("Unity Machine Learning Agents Toolkit") is an open-source framework developed in order to create agents able to execute actions in a game environment, trained using different kinds of available Machine Learning techniques. It is mainly used for bots creation, employed inside gameplay and game testing.

The main development workflow of this framework, as cited in [4], is based on a first step, called "Training" step, and the following "Inference" step. In the former one, the agent is trained inside the Unity game scene using some kind of learning algorithm, chosen by the programmer, based on state-of-the-art PyTorch implementations. This algorithm is implemented through a series of C-Sharp methods written by the programmer and invoked during the training through the "Play" mode offered by Unity. At the end of this first step, a neural network representing the agent's “brain” is returned, that can be used to automatically control the same agent. Some of the training algorithms offered by ML-Agents Framework are Reinforcement Learning, Imitation Learning and NeuroEvolution. In this research project, MLAgents is used in the first part of the development in order to create paddles able to automatically play Pong. This way, I was able to produce a huge game states dataset for neural networks training.

Figure 3.1: Official ML-Agents Framework logo

3.2 Barracuda

Barracuda is a Unity official framework for inference execution, developed for cross-platform compatibility. In other words, Barracuda can be integrated inside a project in order to use a neural network. Barracuda also offers the opportunity to create neural networks inside Unity, but in SPG it is only adopted for game states prediction through inference. Its way of working is very simple: the programmer has to build an input data tensor which will be passed as input to the inference engine, which returns an output tensor. In this project, Barracuda is used inside the server implementation in order to execute predictions through the "MPAI-SPG Predictor" Game Object. In this way, we can take input data from game state and return prediction results inside the same scene.

3.3 TensorFlow

This open source framework was developed by Google for Artificial Intelligence and Machine Learning applications. It is used in my research for deep neural network trainings and evaluations. This library is available in many of the most popular programming languages, but mostly important enables programmers to use its Python API called "Keras", which permits users to create very complex and sophisticated neural networks in a very simple way. In TensorFlow are available quite the totality of architectures developed during the years, together with a huge variety of support tools.

TensorFlow works defining dataflow graphs in order to show how data move through a graph containing a series of operations, taking as input and returning as output a multimodal matrix called "tensor". For this reason, framework's

Figure 3.2: Official TensorFlow logo

name comes from the combination of “tensor”, and “flow”, the movement of the matrix performing operations inside the graph. Before using a neural network, it is necessary to define the input tensor, with a fixed set of dimensions. This framework makes easy the use of GPU in order to speed up operations. Because of their architectures, "General Purpose Units" or "Graphics Processing Units" (GPUs) are suitable for repetitive operations, like the one performed during NN algorithms or graphical rendering, and graphs implemented in TensorFlow are very suitable for accelerators because of their parallelizable structure. As cited in [11], a graph inside TensorFlow is a data structure containing a set of operation objects, representing computational units, and tensor objects, representing sets of data flowing inside the graph.

There is a huge advantage using graphs, over the parallelizable structure, because using their data representation they can be stored in order to make partial trainings in different moments, also allowing to use objects in environments not containing a Python engine. Graphs are also very suitable for classical neural networks operations like gradient descent during the backward propagation, a tecnique used during training in order to update weights.

3.4 Unity

Unity is one of the most popular game engines currently used for 3D and 2D videogames development. It offers an IDE (Integrated Development Environment) able to execute many of operations necessary for a videogame execution, like graphic rendering and user’s input management, enabling the creation of interactive applications. It can also be used for cinematic, not necessarly involving user’s actions. This IDE gives a lot of support tools for game objects

management. It is possible to import 3D models (Unity is not designed for Computer graphics modeling) from "Blender" or "Maya" and easily associate colliders to enable the model to interact with the Physics Engine. Unity is developed in C#, and all the scripts used inside a Unity project have to be written using this language. In my research project, this game engine has been used for Pong implementation (both Online and Offline versions) and the subsequent MPAI-SPG Digital Twin development.

Figure 3.3: Official Unity logo

Chapter 4

Problem and architecture explanation

All the games developed use some kind of Internet connection in order to work. This kind of functionality is used by default by consoles and gaming computers. Also if the Web is now very spread inside the game industry and a huge variety of competitive multiplayer games are now very popular (like “Fortnite” by Epic Games, “Apex Legends” by Respawn Enterteinment or “Call of Duty: Warzone” by Activision), some kind of problems still happen inside player’s experience, like network lag problems that spoils competitive matches all over the world, or cheaters able to go over the game rules and take an advantage over the other players. The aim of MPAI-SPG is to overpass these problems using Artificial Intelligence applications. Given the growing popularity of GPUs (Graphic Processing Units), already used for video rendering inside game computers and consoles, these AI applications can run very smoothly and be integrated with videogames, because GPUs are, in addition, used for neural networks training and inference.

My contribution to the MPAI-SPG research starts from the Pong Online version. It has been developed by Antonio Guarino (Turin University’s Computer Science Department). This application uses the "Photon Online Service" framework developed for Unity [10] and Gambetta’s studies about "Lag Compensation" (see section 4.3).

MPAI-SPG was born in order to correct problems caused by network packet loss and cheatings that can happen inside a videogame based on an Authoritative Server architecture. Examples of games using this kind of structure are “Rocket League” by Epic Games, “Overwatch” by Blizzard Games or “League of Legends” by Riot Games, to cite some of them.

Figure 4.1: In order: "League of Legends", "Overwatch" and "Rocket League" official logos

4.1 Authoritative Server

This kind of structure is used inside videogames industry in order to develop online multiplayer applications, especially competitive games, like the ones cited before, with the aim to avoid the player to have some kind of advantage over the others. Additionally, an authoritative server can correct the **game state** (the set of transforms of every game object inside the scene, together with all the important informations), in order to guarantee a consistent state also if a client is having network latency problems. This structure is based on the fact that each client joining the game sends user's commands to the server, that validates received data and updates the game state through this new knowledge. Then, the new game state is sent to the clients, that will update their rendered scene.

As cited in [1], this setup avoids many types of hacks, also very sophisticated, like *upgraded velocity*. For example, if a client moves faster, for some time it will be in a "hacked" position, but the server, knowing game rules, will take it back to the correct position. This way, it is the server to determine and validate players' position inside the field, so that all the connected clients will be subject to the same game rules. This is the reason why this architecture takes this name: because the server application has got an **authority** over clients, imposing game rules and determining the correct game state.

As we can see from the image 4.2, each client gets user's inputs and send them to the server through the network. The Server, who owns the current correct game state, updates it and returns the new state to the clients, that will render the next frame.

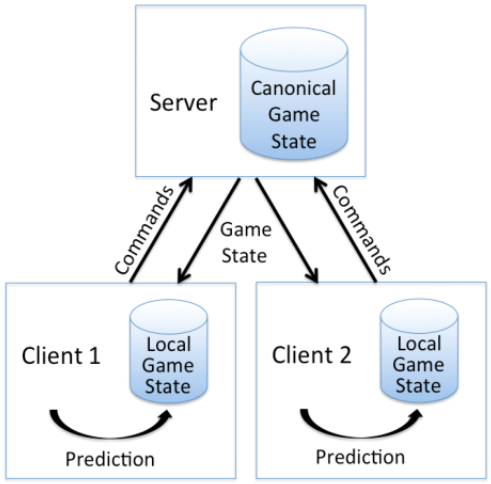

Figure 4.2: Classical Authoritative Server Architecture

Like every implementation, this setup has got pros but also cons.
Pros:

1. The server controls the game experience for all the connected clients
2. Hacking is much harder to reach, making a game suitable for a competitive experience, because the server can check game rules and make them respected
3. All informations are saved inside the server, so the uninstallation of the game from a user's machine doesn't lose data.

Cons:

1. The server will be more complex because it will have to execute much more computations
2. In order to reach a correct game experience, the client needs a prediction system, because network latency is also possibile from server to client

Different kinds of authoritative server are available, like "Fully-Authoritative", "Semi-Authoritative" and "P2P".

4.2 MPAI-SPG Authoritative Server architecture

In picture 4.3, we can see the MPAI-SPG Authoritative Server Architecture defined by Marco Mazzaglia and Leonardo Chiariglione. This architecture shows

the server structure composed by two different parts, excluding Client 1 and Client 2 (violet boxes)

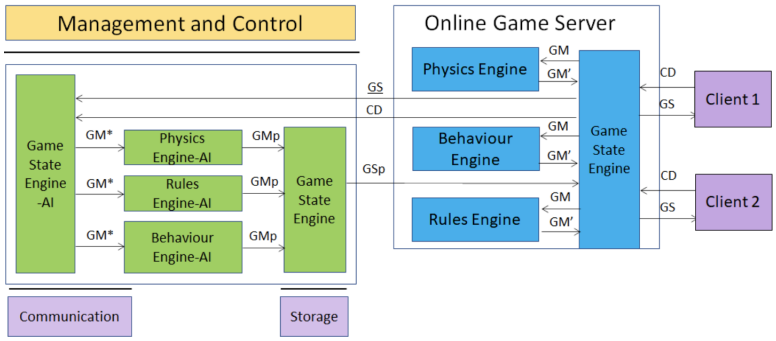

Figure 4.3: MPAI-SPG Architecture

The first part is the “Online Game Server” (set of blue boxes). This structure represents the classical "client-server" way of working, where the server gets data from clients (command data, CD) going inside the Game State Engine. At this point, inside the "Online Game Server", there is a data exchange between the "Game State Engine" and the 3 associated sub-engines: "Physics Engine", "Behaviour Engine" and "Rules Engine". (Pay attention, this transfer depends on the IDE used for game creation, like, for example, "Unity" or "Unreal Engine"). Each one of these sub-systems has the objective to execute some computation given GM received from the "Game State Engine". They return GM’ in order to create the new game State (GS), that is sent back to the clients.

The Predictive Server (the Digital Twin) is on the left, composed by the green boxes connected to the blue ones with the data flow visibile in figure 4.3. These components work together in order to compute the state-that-should-be, receiving the informations about the game states calculated before and getting the next one with a method different from the one used by the "Online Game Server". One of the objectives of this research is to find this new method.

4.3 Lag compensation

As we learned from section 4.1, the way of working of an Authoritative Server is based on a specific routing (client commands to the server, game state update and then sent back to the client) that can make a game very slow, not just because of the routing made, but especially because the connection between the server and the clients can be subject to packet loss or network congestion. In

order to avoid this kind of problems, Gabriel Gambetta in its articles [2] explains a series of concepts that bring to its tecnique called **"Lag compensation"**.

This tecnique is based on different concepts, where one of them is the **"Client Prediction"**. Assuming to observe a non-cheating player, we can imagine the player sending its new commands to the server, and then instantly render its new scene without waiting for server's response. This way, the game inside the server (that in an authoritative architecture is the most important part) will still be consistent and kept authoritative, because client's render, also if wrong, won't affect the one returned from the server, calculated on its own. Instantly render the new frame inside the client can also avoid waiting for the updated render from the client, making a better flow inside the game experience.

Assuming the delay taken by the information to route from client to server and return, the server sends back the updated and correct state also if the player has done another action in between. This brings game experience issues, because the user would receive informations about past actions, making the client ungovernable. In order to avoid this problem, another concept from Gambetta's studies comes in help, the **"Server Reconciliation"**. In order to understand the problem, we have to think that the client is rendering the game in the present, while the server is knowing the past, because it renders the state based on old commands, while the client sends new ones.

Server Reconciliation labels commands sent by client, and the server, when returning its new render, specifies that it is based on that old listed input controls. This way, the client can render its game state through **Client Prediction**, using commands starting from the new server render. For example, suppose the client send 3 commands, named *cdm1*, *cdm2* and *cdm3*. The server receives them, calculates the new game state and then returns the new graphic render, names *rdr1*. While it is receiving new commands from the player, the client receives *rdr1*, updates its state, because we are still in a authoritative server, and updates it with the new commands still not analyzed from the server. The result of this series of operations is a fluid set of consecutive renders inside player's applications, like a single-player game.

The scenario expressed so far can work for a little number of players connected to the same server, but it is too expensive in terms of CPU load and bandwidth for a huge set of clients, because of the informations sent through the network. In other words, this approach is not scalable, especially in a real-time rendering (at least 30 times per second). A first simple solution would be, inside the server, to render at low frequency (like 10 times per second) and create a player's commands queue. Supposing to call the time between two consecutive frames *"timestep"*, at each timestep the server updates the state

using informations inside the queue, containing all the commands from all the clients, and then cleans it.

In extreme situations, where there are a lot of players making very fast movements, like 3D First Person Shooters, the way of working expressed so far can be very inadeguate, because the player could see the others teleporting every *timestep* seconds from one position to another (Note that this kind of concepts can be applied to every kind of variables inside the game). The solution to this problem is to *show the positions of the other players in the past relative to the current one*. This shows the current player in the correct position, while the other ones in the position at the render before. Others' position from the past timestep to the current one is interpolated using **Entity Interpolation**. Note that this way every player will see a slightly different render of the same game, but with little timesteps this doesn't affect the game experience.

Finally, we're arriving to the concept of the **"Lag Compensation"**. Given the concepts expressed before, we could think that playing with users at past positions could affect the experience, making, for example, the First Player Shooter unable to shoot correctly to the others. Suppose we are still working on this kind of game, and we are shooting to an adversary. The client, when enabling the trigger of the gun, sends complete informations about the exact time the action was performed. The server, knowing everything happened before because of the informations received by all the clients in the past, can reconstruct the old states and then see if the adversary has been correctly shot, recreating the condition of the shoot. Then, the server updates the game state at that point in time and updates the clients.

Gambetta's concepts expressed in [2] are an important part inside the Pong Online implementation made by Antonio Guarino, together with the Photon Online service. These two elements together permit to create a fully authoritative server architecture, which will be the study field of my future researches inside this thesis.

Chapter 5

Pong Offline creation

The first step of my research project was the creation of my own Pong implementation in order to study the use case and create automatic movement for paddles. Pong is a 2D, 1 vs 1, Ping Pong simulation game (where it takes its name) released by Atari Game Company in 1972. It is very important inside videogame history because it is one of the first commercialized video games ever and it was very popular in those years. It was first released as an Arcade game in 1972 and then rebuilt as a stand-alone console in 1975. The game is basically composed by a field with a ball and two paddles, each one controlled by one player, or one of them controlled by the computer in case of a single player match. Each paddle can only vertically move (on the Y axis) with the aim to hit the ball and make it pass behind the other player. Each time a player can make the ball go beyond the other, it scores one point. The total score situation is displayed in the center of the rendered field. The first player reaching 10 points wins the game. During the years, many extensions of this game were developed with new rules, like, for example, ball acceleration at each paddle hit. All these new rules have the objective to make the game more competitive and less masterable by the players, making it more intriguing.

I recreated, inside the Unity Editor, this game defining a fixed dimension camera space and putting, first of all, 4 2D BoxColliders covering field sides. This way, the ball, provided itself with a CircleCollider, isn't able to pass over the field, creating this way the collision and bouncing behavior. Paddles and ball are represented in Unity as distinct Game Objects. Paddles are defined using a simple rectangular sprite, while the ball has been defined using a circular sprite.

Ball game object has various components:

- Rigidbody2D: this component says to the Unity Game Engine that the associated game object is considered by the Physics engine. This means

that the object will be subject to classic physics rules that control the scene. Ball has got a mass of 0.1 Kg and a discrete collision detection. Using the Rigidbody2D also enables the programmer to easly get informations about the object's velocity accessing to its "velocity" private field and avoid to manually calculate it

- SpriteRenderer: used to graphically represent the object. In the Pong case, where all the game objects have got only 2 dimensions, it is necessary because it contains the pointer to the sprite.
- Circle Collider: this kind of component is necessary to enable the collision detection with the other game objects present in the scene. Thanks to this component, the ball is able to collide with field walls and paddles
- Ball Controller: custom C-Sharp script implemented by me in order to create the Pong ball movement
- LineRenderer: Unity built-in component used to have a graphical feedback about how the Raycasting Algorithm is working

Inside the classic Pong game, there are two players on the field, each one represented by a paddle. Paddle game object is composed by the subsequent elements:

1. Sprite Renderer: it has got the same aim of the ball's one
2. Box Collider: it creates a rectangular collider around the game object, following the sprite's shape, in order to enable collision detection for this object
3. Rigidbody2D: enables the Physics Engine's action on the game object, applying a discrete collision detection together with "kinematic" body type

5.1 Game Manager

This script has the simple aim to keep track of players' score inside the game and display it on the GUI. It uses two static integer variables representing scores, updated depending on the side wall that has been hit by the ball. The method "OnGUI" manages the Graphical interface components, and reacts to the "Restart" button, resetting players' score.

5.2 Ball movement

5.2.1 Classic Pong Ball movement

Ball movement inside Pong follows a simple rule: the ball hits a collider (a wall or a paddle) in a specific direction, and it is reflected on the X or Y axis using Fresnel rule. If the ball hits a vertical collider, it is reflected respect to the X axis, while if it hits an horizontal collider, the reflection axis is the Y. For example, if the ball hits the right paddle, coming from the left, it will go in the specular direction to the X axis. Game field is a rectangular surface whose sides correspond to camera limits. It is a 16:9 view in a 2D point of view, so we only need the X and Y axis

Figure 5.1: Pong Offline example screen

5.2.2 Implemented Ball movement

Ball is automatically controlled using a script called "BallControl.cs" that extends the Unity "MonoBehaviour" class. At game start, ball can randomly choose between 4 starting directions, defined as 2D vectors like, for example, Vector2D(16, 11). When a new direction is chosen, a force is applied to the ball, which starts moving on that direction with fixed speed. Every time the ball catches a collision, its speed is calculated from the velocity vector and its reflected direction is calculated using the classic Pong rule. This reflected vector is computed using Unity built-in method "Reflect", which takes as arguments the velocity vector and the reflection axis (that, in our case, is the normal to the collision point). Then, the new velocity vector is calculated multiplying

the reflected direction with the speed and used to overwrite ball's Rigidbody2D velocity component.

```
1  void OnCollisionEnter2D(Collision2D coll) {
2    var speed = rb2d.velocity.magnitude;
3    // First argument is the direction, second one is the normal
4    var direction = Vector2.Reflect(lastVelocity.normalized,coll.
         contacts[0].normal);
5
6    rb2d.velocity = direction * speed;
7  }
```

Listing 5.1: "OnCollisionEnter2D" method code

"BallControl.cs" also contains informations about scoring. Ball has got a collider component with an enabled trigger, which invokes the "OnTriggerEnter2D" method every time it is activated. If the colliding object is a side wall, the opposite paddle invokes its "Goal" method, that makes a score upgrade through the GameManager's static integer variable. Also, informations about the hit target are updated for MPAI-SPG logs construction.

```
1  void OnTriggerEnter2D(Collider2D collision) {
2    // if the ball collides with the right wall
3    if (collision.gameObject.name.Equals("RightWall")) {
4      paddle1.Goal(hits);
5      hitTarget = "RightWall";
6    // if the ball collides with the left wall
7    } else if (collision.gameObject.name.Equals("LeftWall")) {
8      paddle2.Goal(hits);
9      hitTarget = "LeftWall";
10   }
11 }
```

Listing 5.2: "OnTriggerEnter2D" method code

As we can see from image 5.2, given the blue dot representing the ball, the red vectors representing ball's direction before and after wall collision, and the green line representing the top wall's normal, we can see the reflective behavior of the ball when hitting the top wall. The sphere goes on until it hits the wall, then the collision's normal axis is taken and, using Fresnel's Rule, the reflected direction is calculated: the angle between the entering vector and the normal is equal to the angle from the normal axis to the reflected vector.

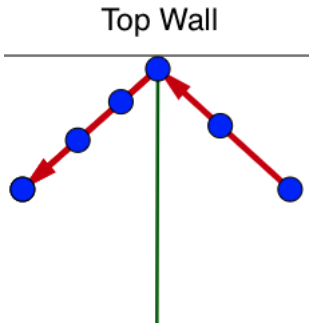

Figure 5.2: Example of reflection movement inside Pong

5.3 Player's movement

Player's paddle movements are defined inside the "PlayerControls.cs" script. It is a custom script (that implements Unity "Monobehaviour" class) and so it defines the "Update" method. This function is invoked at each frame, and into it player's movement is controlled and constrained by the field limits (the maximum Y value is 2.25f). Defined player's speed ("speed") as a constant, user sends to the game an input through its controller. It sends only a value at a time because just the Y axis movement is required. If the received input is positive, then player's Y velocity value is set to *speed*, otherwise it is set to *-speed*, so player's movement is just a velocity update. In addition, informations about player's movement are recorded in order to perform the log creation following MPAI-SPG formal rules.

```
1  void Update () {
2    //Constraining player's position
3    var pos = transform.localPosition; //getting current player's
         position
4    //If position exceedes field bounds, it is overwritten
5    if (pos.y > boundY) {
6      pos.y = boundY;
7    } else if (pos.y < -boundY) {
8      pos.y = -boundY;
9    }
10   transform.localPosition = pos; //updating player game object
         position
11
12   // Translating player
13   float moveY = Input.GetAxis("Vertical"); //getting player's
         input info
14   Rigidbody2D rb = GetComponent<Rigidbody2D>();
```

```
15    var vel = rb.velocity; // getting velocity vector
16    if (moveY > 0) { //go up
17      vel.y = speed;
18      action = "Up";
19    } else if (moveY < 0) { //go down
20      vel.y = -speed;
21      action = "Down";
22    } else { //Idle
23      vel.y = 0;
24      action = "Idle";
25    }
26    rb.velocity = vel; //updating player's velocity
27  }
```

Listing 5.3: PlayerControls.cs "Update" method code

As we can notice, Pong is a very simple videogame, very suitable as SPG use case because it can permit us to think about complex problems through simple examples, also keeping a low number of factors to manage inside the game.

Chapter 6

Neural Networks for automatic movement

In order to create a huge dataset to train AIs that will form the SPG ecosystem, I thought about a method to automatically move players inside my Pong Offline application and record their actions while playing. Each Pong match will be observed by a game object called "Record Writer", which will keep track of all the physical quantities listed in the SPG document that will have to be predicted.

To create a paddle able to play without user's support, I used the "ML-Agents" framework (see 3.1), toolkit created to define neural networks able to control an Agent inside the game.

6.1 Reinforcement Learning

ML-Agents offers multiple training methods, listed in [6], and the one I've chosen is **Reinforcement Learning**, one of the three classical Machine Learning Paradigms together with "Unsupervised Learning" and "Supervised Learning". This kind of training is based on the concept of **"reward maximization"**, achieved through a **Reward Cycle** (figure 6.1 based on 4 principal steps, generally defined as:

1. Environment Update: the scene the agent is involved in is updated at each frame consequently to agent's actions

2. Environment Analysis: the agent analyzes through a sensor some of the elements present inside the scene in order to determine its next action

3. Agent's action: the agent, given observed data got at the previous step, determines an action to perform and executes it

4. Reward: the agent receives a positive or negative feedback depending on the fact the action he has just performed is correct or not. This kind of parameters are determined by the programmer and the agent tends to maximize this positive value in order to act always in a correct way

Generally speaking, the agent is constantly making decisions about the actions to perform, given its objective to achieve (the function maximization) and its view of the world (the set of observed physical quantities). As explained in [7], through the Reinforcement Learning algorithm we want to learn a **policy**, a mapping between observations and actions.

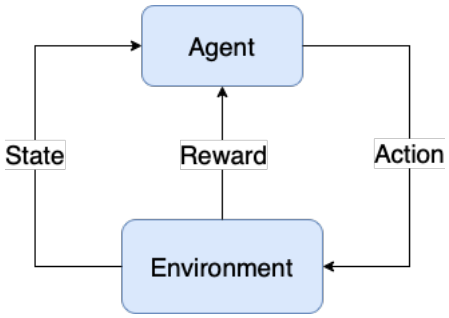

Figure 6.1: Reinforcement Learning Cycle

During learning, ML-Agents trains a neural network representing the "engine" or "brain" of the agent. This takes in input the observed world data and returns a specific action to execute. Actions are defined as a vector of values returned by the NN, with each vector's dimension referring to a modification to perform inside the agent parameter. For example, given the Pong use case, the only action the agent can perform is the movement along the Y axis, so the developed NN will return a one-position vector (so, just one value) containing the offset to apply to the y component of the transform's position.

ML-Agents uses a time subdivision defined through an "Episode". During this interval, the agent performs the Reinforcement Learning cycle until the *max_step* value is reached, where a **step** is defined as "an atomic change of the engine that happens between Agent decisions", as determined in [5]. Alternatively, an episode can be ended by the programmer invoking the "EndEpisode" method.

6.2 ML-Agents Neural Network

ML-Agents lets the programmer define neural network's hyperparameters and train it through the Unity environment. This NN is determined inside a **yaml** file, which writes all the parameters, partially displayed below. Referring to the Pong training, an example of network is defined as follows:

```
default_settings: null
behaviors:
  PlayPong: # name of the trained Neural Network
    trainer_type: ppo
    hyperparameters: # fundamental hyperparameters of the NN
      batch_size: 1024
      buffer_size: 2048
      learning_rate: 0.0003
      beta: 0.0005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 8
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 64
      num_layers: 1
      vis_encode_type: simple
      memory: null
      goal_conditioning_type: hyper
    max_steps: 5000000
    time_horizon: 64
    summary_freq: 10000
```

Listing 6.1: ML-Agents Neural Network YAML file

This yaml file is passed as argument to the learning command "mlagents-learn". As we can see, the network uses the "Proximal Policy Optimization" (PPO), a Policy Grandient algorithm developed by OpenAI used to create a connection between observations and actions. PPO is a method which trains a stochastic policy. In few words, the agent, collecting observations from the environment, returns an output, while the critic method gives the expected rewards for the input data given. A Policy Gradient method is used when an agent, given input data, doesn't know exactly which would be the best move, so it chooses through the policy gradients. Generally talking, Policy Gradient methods are not very strong because they are hypersensible to hyperparameters, meaning that changing a bit some parameters, the obtained results are way different. PPO wants to remove this correlation between parameters and algorithm performance.

Referring to the yaml file 6.1, the training set is divided in batches of 1024 elements each and the learning rate for weight update is equal to 0.0003. The training is performed in 8 epochs.

6.3 ML-Agents methods

For the purposes of being correctly integrated inside a Unity project, ML-Agents requires the implementation of abstract methods belonging to the "Agent" class through a custom script extending it. Inside programmer's script, the following methods will be overridden:

1. "CollectObservations": gets all necessary input informations for the Agent's neural network execution, passing them as argument to the "AddObservation" method of the "VectorSensor" class. For example, suppose we want to observe a object's position, then the code will be

```
1  // Class definition extending "Agent" class
2  public class PlayerControls : Agent {
3    public override void CollectObservations(VectorSensor
         sensor) {
4      sensor.AddObservation(object.transform.position);
5    }
6  }
```

Listing 6.2: "CollectObservations" method example

2. "OnEpisodeBegin": determines the actions to perform at the beginning of an ML-Agents episode

3. "OnActionReceived": on this method, game state is updated using informations returned by the NN and applied to the agent, which executes some action. For example, supposing we want to move the agent on the X axis, then we have to access to the "ContinuousActions" array component of the "ActionBuffers" class. The resulting code will be like this:

```
1  public override void OnActionReceived(ActionBuffers actions
       ) {
2    float xOffset = actions.ContinuousActions[0];
3    object.transform.position += new Vector3(xOffset, 0, 0) *
         Time.deltaTime * playerSpeed;
4  }
```

Listing 6.3: "OnActionReceived" method example

4. "Heuristic": method employed when we want to “manually” use the agent, controlling it with some kind of input. It bypasses the neural network computation, so the object cannot move on its own. It consists of overwriting the "ContinuousActions" array cited at the previous point with the information got from the input manager. Here’s an example of implementation

```
1  public override void Heuristic(in ActionBuffers actionsOut)
       {
2    ActionSegment<float> continuousActions = actionsOut.
       ContinuousActions;
3    continuousActions[0] = Input.GetAxis("Vertical");
4  }
```

Listing 6.4: "Heuristic" method implementation example

6.4 Implementation tests

A huge variety of tests has been performed in order to find suitable methods implementations, understand which rewards assign and which data observe. Here are presented two different implementations, one executed at the beginning of the research and the definitive one. With the aim to correctly understand necessary data for a useful learning, I’ve executed different tests, every time providing distinct inputs and studying agent’s behavior. This way I’ve created different implementations, trying to achieve the most useful behavior. For all the experiments made during my research, the playground was always the same: the Pong field with two paddles working on it, where each paddle contributes to the building of the same neural network.

6.4.1 First implementation

On my first implementation, I’ve assigned both agents the following items to observe:

- Opponent’s Y-axis position (1 parameter)
- Opponent’s velocity (2 parameters)
- Ball’s position (2 parameters)
- Ball’s velocity (2 parameters)

reaching a total of 7 observed parameters.

The set of rewards assigned to agents’ actions is presented in table 6.1 below:

Reward	Condition
+1	player hits the ball
+5	player scores
+2	player comes closer to ball's Y position
-2	player goes further from ball's Y position

Table 6.1: Set of rewards defined in the first implementation

An extract of the training log is presented here on table 6.2:

Num	STEP	TIME ELAPSED	MEAN REWARD	REWARD STD
1	50000	62.702 s	-243.506	376.626
2	100000	119.254 s	-396.149	866.985
3	150000	178.011 s	-221.295	828.040
4	200000	237.410 s	-240.160	1132.579
5	250000	297.554 s	11.087	1142.961
6	300000	356.357 s	-46.425	881.249
7	350000	424.000 s	140.331	844.170
8	400000	483.611 s	184.036	1169.185
9	450000	551.613 s	286.787	838.384
10	500000	613.806 s	312.625	1088.825

Table 6.2: Extract of training log resulting from the first implementation

As we can imagine, this reward system definition is very poor, and the behavior obtained shows the agents not able to understand ball's movements, moving randomly. Obtained this results, it was necessary to make the agents able to "understand" ball's movement inside the field, in the same way a human would do. For example, a real person playing Pong would try to anticipate ball's position on his side of the field. This way, the player will already be in position for ball reception when it will arrive on his side of the field. To emulate this kind of behavior, first of all we have to obtain the informations about the ball arrival using a "Raycasting" system. Then, we have to correct the reward system to teach the agent to go closer to that position. The aim is to create a player able to anticipate ball's position and reach it with no difficulty.

6.4.2 Second implementation

Pong Offline, with the new "trajectory-prediction" system, makes the player observe the following values:

- Player's Y position: Agent's position on the Y axis (1 parameter)
- Ball's position: (x, y) coordinates (2 parameters)
- Ball's velocity: in form of a 2-dimensional vector, represents ball's movement along X and Y axis using some speed (2 parameters)
- Opponent's position: Agent gets knowledge about other player's Y axis position in form of float value (1 parameter)
- Destination: value returned by the Raycasting system. Ideally it is the position along the Agent's Y axis where the ball will arrive (1 parameter)

In the end, the total number of observed parameters is equal to 7.

Given all this observations, the reward system is quite different from the one shown on table 6.1. Instead of reaching the same opponent's Y position, the paddle wants to go on the predicted ball's position. In addition, actions get new rewards. For example, if a player scores a point, gets a +10 reward instead of +5. The "OnActionReceived" method calculates at each invocation the distance, in absolute value, between agent's position and Raycasting algorithm output two times: at the method start, and after applying the NN output to the transform's position. The resulting implementation is shown in listing 6.5:

```
1  public override void OnActionReceived(ActionBuffers actions) {
2    float yPosition = transform.localPosition.y;
3    float yTargetPosition = target.transform.localPosition.y;
4
5    // Difference between player's y coord and ball's y coord before
         player's translation
6    float yOldDiff = Math.Abs(yPosition - yTargetPosition);
7
8    // Translating player
9    float moveY = actions.ContinuousActions[0];
10   transform.localPosition += new Vector3(0, moveY, 0) * Time.
         deltaTime * playerSpeed;
11
12   // Difference between player's y coord and ball's y coord after
         player's translation
13   yPosition = transform.localPosition.y;
14   yTargetPosition = target.transform.localPosition.y;
15   float yNewDiff = Math.Abs(yPosition - yTargetPosition);
16
17   // If player's y coord comes closer to to ball's y coord, it
         gets a reward
18   if (yNewDiff <= yOldDiff) { // player is closer to ball's y
         position
19     AddReward(+4); // Reward
```

```
20      } else { // player is further from ball's y position
21        AddReward(-4); // Penality
22      }
23  }
```

Listing 6.5: "OnActionReceived" second implementation code

Given the Reinforcement Learning Policy, the agent will know that, at every rendered frame, it will have to go closer to the computed destination in order to obtain a positive reward. Thus, the paddle will learn to intercept the ball and also to wait for it. The scores defined in this second implementation are collected inside the table 6.3 below.

Score	Condition
+4	Closer to the raycasting destination
-4	Further from the raycasting destination
+6	Exactly on the raycasting destination
+10	Made score
-5	Suffers a point
+3	Hits the ball
+3	Never scored a point

Table 6.3: Scores defined for second implementation

An extract of the log obtained runnig the *"mlagents-learn"* command is shown here on table 6.4

Num	STEP	TIME ELAPSED	MEAN REWARD	REWARD STD
1	10000	92.994 s	371.400	1468.132
2	100000	566.819 s	354.988	1503.835
3	200000	1078.304 s	310.587	1342.535
4	300000	1615.779 s	540.529	1916.601
5	400000	2127.366 s	550.549	2026.855
6	500000	2638.983 s	222.172	1055.794
7	600000	3152.937 s	410.307	1645.523
8	700000	3695.544 s	755.875	2387.369
9	800000	4210.249 s	249.855	1084.000
10	900000	4711.712 s	639.587	2148.836
11	1000000	5223.487 s	1099.250	2966.666
12	1100000	5734.626 s	471.046	1700.886
13	1200000	6245.336 s	1000.152	2780.156
14	1300000	6754.257 s	521.220	1853.611
15	1400000	7276.392 s	480.516	1759.401

Table 6.4: Extract of training log resulting from the second implementation

Qualitatively talking, the player obtained by this new definition is able to play both long and short rallies, in a way a human being could play: good or bad matches. If I created a "perfect player", the collected log would represent an "ideal" situation, not reachable in reality.

6.5 Raycasting algorithm

The need to create this technique comes because the trained players weren't able to reach a satisfying ability to hit the ball, so the aim was to make them capable to predict ball's position through a system that indicates where it will go on their field side. The raycasting system is based on a ray drawn from the sphere that reflects on the field until it reaches the other player's side. Raycast is triggered when the ball hits a paddle. From that, it calculates its new direction using the classic Pong Reflection rule: it repeatedly reflects (always following the same rule) on field's sides, and then intersects a vertical line placed at the opponent's X value. This intersection represents the place where the ball will arrive on the opponent's side, and that will have to be reached to hit the ball back. The raytracing script is attached to the "Ball" game object. It keeps track of right and left paddle destinations values and it upgrades one of them every time the ball hits a paddle. Players get information about this destination every

"OnActionReceived" invocation, and ML-Agents reward algorithm gives them a score if the paddle comes closer to that destination or not.

In order to see the raycasting working, a Unity "LineRender" component is used. It is a built-in script that draws a interpolated line between two or more points, as we can see in image 6.2:

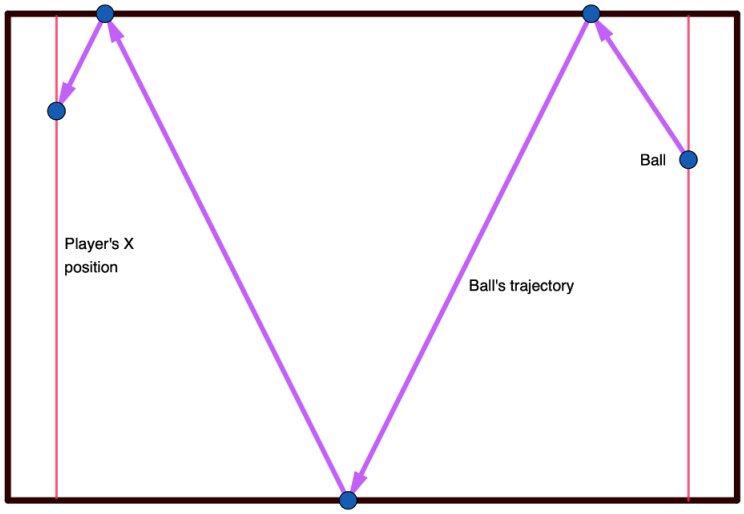

Figure 6.2: Raycasting example

6.5.1 Initial Raycasting

The raycasting algorithm is composed by multiple methods used in different situations. Supposing the ball in position (0, 0) (the center of the field) at the game start, a first ray is automatically fired in the initial direction, randomly chosen by the Game Manager. The position that will be reached by the ball is computed by the "initialRaycast" method. This way, the paddle firstly receiving the ball will be able to hit it correctly. The “initialRaycast” method implementation is presented below in the code listing 6.6:

```
1 // Launches a ray following the given direction
2 // Returns ray intersection Y value with lateral plane
3 public float initialRaycast(Vector2 direction, Vector2 startingPos
      ) {
4
5   // Enabling line renderer
6   lineRenderer.enabled = true;
7   lineRenderer.sortingOrder = 0;
```

```
8   lineRenderer.positionCount = 2;
9
10  // Getting parent's (field) position
11  Vector2 parpos = transform.parent.gameObject.transform.
        localPosition;
12  Ray ray = new Ray(parpos, direction); // creating a new ray
13  Plane plane;
14
15  if (direction.x < 0) { // Ball is going to the left
16    // Creating plane with normal to the right
17    Vector3 planeNormal = Vector3.right;
18    Vector3 planePoint = new Vector3(parpos.x - 4f, parpos.y, 0);
19    plane = new Plane(planeNormal, planePoint);
20  } else { // ball is going to the right
21  // Creating plane with normal to the left
22    Vector3 planeNormal = Vector3.left;
23    Vector3 planePoint = new Vector3(parpos.x + 4f, parpos.y, 0);
24    plane = new Plane(planeNormal, planePoint);
25  }
26
27  float distance = 0f;
28  Vector3 point = Vector3.zero;
29  // Computing ray intersection with the plane
30  if (plane.Raycast(ray, out distance)) {
31    point = ray.GetPoint(distance); // getting intersection point
32  }
33
34  // Setting up line renderer interpolation points
35  lineRenderer.SetPosition(0, parpos);
36  lineRenderer.SetPosition(1, point);
37
38  return point.y - parpos.y; // returning the destination position
        , relative to field's position
39 }
```

Listing 6.6: "initialRaycast" method

First of all, the script enables the line renderer and sets it up. Then, given parent's position (the field), a Ray object starting from that point and going to ball's direction is created. Note that the script often refers to the parent's position because ML-Agents lets the user create multiple *n* field instances inside the Unity game scene, in order to reduce the training time *n* times. So, in order to relativize returned results, all calculations are made referring to field's position.

Given ball's direction, we understand if it is going to the left or to the right player in order to create the correct intersection line, which is represented by a Plane object with relative position (-4, 0, 0) for the left player and (4, 0, 0) for

the right player. Values 4 and -4 are default Pong paddles' positions on the X axis. Plane is then directed pointing to (0, 0) (to the left for the right one, and viceversa) through a normal axis. Then, ray-plane intersection is calculated and relative destination value is returned (not considering parent's position).

6.5.2 Recursive method

Supposing that the paddle reaches the ball coming from the center of the field and then hits it back, the "createRayRecursive" method is invoked. Specifically, it is invoked every time a collider hits the ball. The method implementation is presented in the following code listing:

```
1  public float createRayRecursive(Vector2 start, Vector3 normal,
       float xPos) {
2    // Setting up line renderer
3    lineRenderer.enabled = true;
4    lineRenderer.sortingOrder = 0;
5    lineRenderer.positionCount = 2;
6
7    // "hit" represents the collision point
8    hit = launchRay(start, normal); // Launching the first ray from
         collision point with a paddle, following ball's direction
9
10   float res = 0f;
11   if (hit) { // if something has been hit
12
13     // Line Renderer management
14     int posCounter = 0; // index where to put line positions
15     lineRenderer.SetPosition(posCounter, start); // setting line
           starting point
16     posCounter ++;
17     lineRenderer.SetPosition(posCounter, hit.point); // setting
           line new point
18
19     // if one of the players has been hit
20     if (hit.collider.gameObject.name == "Paddle1" || hit.collider.
           gameObject.name == "Paddle2") {
21       //First ray already hit opposite plane, calculating
             intersection
22       Vector2 parpos = transform.parent.gameObject.transform.
             position;
23       float temp = xPos - parpos.x;
24       if (temp > 0) { // Ball has been hit by right player
25         Ray ray = new Ray(start, rigidbody2D.velocity.normalized);
26         res = straightPlaneIntersection(ray, true);
27       } else { // Ball has been hit by left player
28         Ray ray = new Ray(start, rigidbody2D.velocity.normalized);
```

```
29          res = straightPlaneIntersection(ray, false);
30        }
31      } else { // ball hit another collider
32        bool end = false;
33        int i = 0; // counter for maximum number of reflections
34        int maxIter = 50; // maximum number of allowed reflections
35        Vector2 rayStart = hit.point;
36        Vector2 hitNormal = hit.normal;
37
38        Vector2 oldDirection = Vector2.zero; // initializing
              variable
39        if (rigidbody2D != null) {
40          oldDirection = rigidbody2D.velocity.normalized;
41        }
42
43        while (end == false) { // until the ray keeps intersecting a
              lateral wall
44
45          // Getting old layer info, so an object cannot collide
                with itself
46          int oldLayer = hit.collider.gameObject.layer;
47
48          // Setting up a new layer
49          hit.collider.gameObject.layer = LayerMask.NameToLayer("
                Ignore Raycast");
50
51          //Creating a new reflected ray and launching it
52          Ray2D ray2 = createReflectedRay(rayStart, hitNormal,
                oldDirection);
53          RaycastHit2D hit2 = Physics2D.Raycast(ray2.origin, ray2.
                direction);
54          hit.collider.gameObject.layer = oldLayer; // setting up
                the old layer back
55
56          if (hit2) { // if something has been hit by the second ray
57            if (hit2.collider.tag == "ScoreWallRight" || hit2.
                    collider.tag == "ScoreWallLeft"||
58                hit2.collider.tag == "Player1" || hit2.collider.tag
                      == "Player2") {
59
60              end = true; // Here hit2 represents a collision of a
                    ray to a lateral wall or player -> exiting from
                    the cycle
61
62              // Calculating intersection with plane at player x
                    heigth
63              Vector2 parpos = transform.parent.gameObject.transform
                    .position;
64              float temp = xPos - parpos.x;
```

```
65              if (temp > 0) { // Ball has been hit by right player
66                res = planeIntersection(rayStart, hitNormal,
                      oldDirection, true);
67              } else { // Ball has been hit by left player
68                res = planeIntersection(rayStart, hitNormal,
                      oldDirection, false);
69              }
70            }
71
72            // Preparing the cycle to execute another interaction
73            lineRenderer.positionCount++;
74            posCounter++;
75            lineRenderer.SetPosition(posCounter, hit2.point);
76            oldDirection = ray2.direction;
77            rayStart = hit2.point;
78            hitNormal = hit2.normal;
79            hit = hit2;
80          }
81
82          i++; // upgrading iteration
83          if (i == maxIter) {
84            end = true;
85          }
86        } // end while
87      }
88    }
89    return res;
90  }
```

Listing 6.7: "createRayRecursive" method

Many of the operations are the same of the "initialRaycast" algorithm. First of all, a ray is fired following ball's direction using the auxiliar method "launchRay" shown below:

```
1  // Creates and launches a new ray, returning hit point
2  // Start: ray starting point
3  private RaycastHit2D launchRay(Vector2 start) {
4    Vector2 direction = rigidbody2D.velocity.normalized;
5    Ray2D newRay = new Ray2D(start, direction);
6    return Physics2D.Raycast(newRay.origin, newRay.direction);
7  }
```

Listing 6.8: "launchRay" method

The objective of this function is to draw the direction that the ball will follow after the collision with a paddle. This method takes as argument the collision point's position and fires a ray in the ball' direction. Note that the direction used for this specific ray tracing is the ball's one, and not the reflected one as it could

be expected, because, at the time of method invocation, the ball is already reflected along the new direction. The method "launchRay" returns informations about what the ray has hit and if there's some intersection or not. In case an hit happens, the algorithm continues, following two different execution branches. In the first situation, the ray has already hit a paddle, so the destination computation is already possible using the intersection method explained next. In the other case, one or more other rays have to be fired, so a "while" cycle starts indicating that the algorithm will fire new rays until a lateral wall or paddle is hit. This iteration practically defines a **broken line** that represents ball's trajectory inside the field (analogous to the one presented in image 6.2), where this line is defined by a series of rays where the top of the previous is connected to the tail of the following. The cycle is structured as follows:

1. Some game object gets a “Ignore Raycast” layer. This way it will not be considered during ray intersection and the ray won't stop.

2. A reflected ray is created starting from the previous ray's intersection point through the "createReflectedRay", receiving as argument the starting point, the normal vector of the hit object and the ball's direction before collision

3. This new reflected ray is thrown and its intersection informations are obtained

4. If the new ray has hit one of the lateral walls or one of the paddles, then the line intersection with this new ray is calculated and the while cycle can stop. Otherwise, a new interpolation point is added to the line renderer and the cycle repeats

Plane intersection is basically calculated the same way as in the "initialRaycast" method: a plane on the left or on the right is created, and the intersection point is returned.

Chapter 7

Log files production

As I already explained in chapter 6, players able to play Pong on their own were created with the objective to record physical quantities values and create a training dataset. This set of files is created through a specific game object able to watch all the environment and write values into external files.

7.1 Datasets

A predictive Neural Network composing the MPAI-SPG Digital Twin has to be able to make a forecast about the **Game State**, the data set defining all the values inside the scene. This set of values refers to game objects' transforms and quantities associated to the game environment, like, for example, the score. It can be seen as a game **snapshot** defined only by numbers. In Pong, the state is defined by the following features:

1. Ball position: 2D vector defining x and y ball's position inside the field
2. Ball velocity: 2D vector containing x and y movement of the ball in some direction. From velocity, when can get the direction removing the speed information, by normalizing the velocity vector
3. Left paddle position
4. Left paddle velocity
5. Right paddle position
6. Right paddle velocity

7. Match state: string defining the condition of the game
8. Left player score: integer containing information about how many points a player has made, following game's rules
9. Right player score
10. Hit target: the last side wall that has been hit by the ball (the side of the last player who lost a point)

The total number of recorded values is equal to 16. Inside the MPAI-SPG standard, other datasets are defined, some of them containing a subset of the Game State, like Physics, Behavior and Rules. Specifically, these log files keep track of the following physical quantities:

Physics Engine:

Physical Quantity	**Parameters Number**
Ball position	2
Ball velocity	2
Left paddle position	2
Left paddle velocity	2
Right paddle position	2
Right paddle velocity	2

Table 7.1: Physics Engine Log structure

Behavior Engine:

Physical Quantity	**Parameters Number**
Left paddle position	2
Left paddle velocity	2
Right paddle position	2
Right paddle velocity	2

Table 7.2: Behaviour Engine Log structure

Rules Engine:

Physical Quantity	Parameters Number
Match State	1
Left player score	1
Right player score	1
Hit Target	1

Table 7.3: Rules Engine Log structure

Commands:

Physical Quantity	Parameters Number
Controller 1 Motion	1
Controller 2 Motion	1
Controller 1 Fire	1
Controller 1 Fire	1

Table 7.4: Commands Log structure

7.2 CSV files

All these collections of data introduced in section 7.1 are written inside external CSV files.

CSV is a specific file format that separates values using semicolons, commas or tabs and puts them in a tabular format. Each row is formed by a series of separated values ended with a "newline" escape character. This way it is easy to organize a huge amount of data in a simple way, because they can be automatically created through simple scripts. For example, a string defined in a code as:

```
1    string row = "1; 2; 3; 4; 5; 6; 7\n"
```

Listing 7.1: String example

and written on a file with ".csv" extension, will be displayed inside a program like Microsoft Excel as:

1	2	3	4	5	6	7

Table 7.5: Example of CSV file display

The Log file is defined as follows: given the first row showing columns *attributes* (the name of the columns), each file row is a record of informations, a vector, where each position refers to a specific label, given the index. Example of a complete Pong Game State with an associated record is the one displayed on table 7.6.

7.3 Record Writer

In order to record data listed in section 7.1, I've created a script that gets informations from all the game objects, composes a string containing and separating them with semicolons, and writes them down on different CSV files, following the SPG definition. So, at each invocation, a new row is added to the files "GameState.csv", "PhysicsEngine.csv", "BehaviorEngine.csv", "RulesEngine.csv", "Commands.csv". This custom script is called "RecordWriter.cs" and extends the "MonoBehaviour" class offered by the Unity framework in order to attach it to a game object as a component. To keep running multiple instances of the game at the same time, decreasing the necessary time for log creation, the script creates at its start a new filename containing a incrementing index, which depends on the files already present in the log folder. For example, supposed the file "GameState.csv" already exists, a new file "GameState_1.csv" will be created. This trick enables me to avoid file access conflicts between different Pong Offline instances running at the same time.

As we can see from code 7.2, the method is implemented as a routine (IEnumerator), which is invoked at script start. This implementation is caused by the necessity to record data at a given rate, because of an architectural issue discovered during the Pong Online inference execution (see section 10.2). Initially, the method was executed as a "FixedUpdate", working at the constant rate of 50 invocations/second, so the number of collected records was very high, but testing the neural networks inside the Pong Online implementation, it was necessary to collect data at a lower rate, requiring a specific delay time between two following invocations. In order to achieve this behavior, first of all I've recorded the execution time needed by my "writeLogs" method, noticing that the time from function start to end is equals to few milliseconds, so it is negligible. After that, I've switched the method from "FixedUpdate" to a routine, because it offers special commands like "WaitForSeconds", (line 38) that en-

B Pos		B Vel		LP Pos		LP Vel		RP Pos		RP Vel					
X	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**MS**	**LP Score**	**RP Score**	**HitTarget**
-1.2	-0.3	-3.2	2.2	-4.0	0.0	0.0	3.0	4.0	0.0	0.0	-3.0	MS	2	0	RWall
-1.2	-0.3	-3.2	2.2	-4.0	0.1	0.0	3.0	4.0	-0.1	0.0	-3.0	MS	2	0	RWall
-1.1	-0.2	-3.2	2.2	-4.0	0.1	0.0	3.0	4.0	-0.2	0.0	-3.0	MS	2	0	RWall
-1.0	-0.2	-3.2	2.2	-4.0	0.2	0.0	3.0	4.0	-0.1	0.0	3.0	MS	2	0	RWall
-1.0	-0.1	-3.2	2.2	-4.0	0.3	0.0	3.0	4.0	0.0	0.0	3.0	MS	2	0	RWall
-0.9	-0.1	-3.2	2.2	-4.0	0.3	0.0	3.0	4.0	0.0	0.0	3.0	MS	2	0	RWall
-0.8	0.0	-3.2	2.2	-4.0	0.4	0.0	3.0	4.0	0.1	0.0	3.0	MS	2	0	RWall
-0.8	0.0	-3.2	2.2	-4.0	0.4	0.0	3.0	4.0	0.1	0.0	3.0	MS	2	0	RWall
-0.7	0.0	-3.2	2.2	-4.0	0.5	0.0	3.0	4.0	0.2	0.0	3.0	MS	2	0	RWall
-0.6	0.1	-3.2	2.2	-4.0	0.6	0.0	3.0	4.0	0.3	0.0	3.0	MS	2	0	RWall
-0.6	0.1	-3.2	2.2	-4.0	0.6	0.0	3.0	4.0	0.3	0.0	3.0	MS	2	0	RWall
-0.5	0.2	-3.2	2.2	-4.0	0.7	0.0	3.0	4.0	0.4	0.0	3.0	MS	2	0	RWall
-0.4	0.2	-3.2	2.2	-4.0	0.7	0.0	3.0	4.0	0.4	0.0	3.0	MS	2	0	RWall
-0.4	0.3	-3.2	2.2	-4.0	0.8	0.0	3.0	4.0	0.5	0.0	3.0	MS	2	0	RWall
-0.3	0.3	-3.2	2.2	-4.0	0.9	0.0	3.0	4.0	0.6	0.0	3.0	MS	2	0	RWall
-0.3	0.4	-3.2	2.2	-4.0	0.9	0.0	3.0	4.0	0.6	0.0	3.0	MS	2	0	RWall
-0.2	0.4	-3.2	2.2	-4.0	1.0	0.0	3.0	4.0	0.7	0.0	3.0	MS	2	0	RWall
-0.1	0.4	-3.2	2.2	-4.0	1.0	0.0	3.0	4.0	0.7	0.0	3.0	MS	2	0	RWall
-0.1	0.5	-3.2	2.2	-4.0	1.1	0.0	3.0	4.0	0.8	0.0	3.0	MS	2	0	RWall
0.0	0.5	-3.2	2.2	-4.0	1.2	0.0	3.0	4.0	0.9	0.0	3.0	MS	2	0	RWall

Table 7.6: Pong Game State extract

ables us to wait for a specific time before executing another invocation. This way, it is possibile to record game state values at a specific rate, and create a dataset suitable for the Pong Online execution.

```
1  IEnumerator writeLogs() {
2    while(true) {
3      // Getting physical quantities informations
4      Vector2 ballPos = ballControl.GetPosition();
5      Vector2 ballVel = ballControl.GetVelocity();
6      Vector2 leftPlayerPos = leftPlayerControl.GetPosition();
7      Vector2 leftPlayerVel = leftPlayerControl.GetVelocity();
8      Vector2 rightPlayerPos = rightPlayerControl.GetPosition();
9      Vector2 rightPlayerVel = rightPlayerControl.GetVelocity();
10     string matchState = "MatchState";
11     string leftPlayerScore = GameManager.PlayerScore1.ToString();
12     string rightPlayerScore = GameManager.PlayerScore2.ToString();
13     string hitTarget = ballControl.getHitTarget();
14     string rightPlayerAction = rightPlayerControl.getAction();
15     string leftPlayerAction = leftPlayerControl.getAction();
16
17     // Composing strings
18     string text = ballPos.x + ";" + ballPos.y + ";" + ballVel.x +
           ";" + ballVel.y + ";" + leftPlayerPos.x + ";" +
           leftPlayerPos.y + ";";
19     text = text + leftPlayerVel.x + ";" + leftPlayerVel.y + ";" +
           rightPlayerPos.x + ";" + rightPlayerPos.y + ";" +
           rightPlayerVel.x + ";" + rightPlayerVel.y + ";";
20     text = text + matchState + ";" + leftPlayerScore + ";" +
           rightPlayerScore + ";" + hitTarget + "\n";
21     File.AppendAllText(gameStateFilename, text);
22
23     string behaviourtext = leftPlayerPos.x + ";" + leftPlayerPos.y
           + ";" + leftPlayerVel.x + ";" + leftPlayerVel.y + ";";
24     behaviourtext = behaviourtext + rightPlayerPos.x + ";" +
           rightPlayerPos.y + ";" + rightPlayerVel.x + ";" +
           rightPlayerVel.y + "\n";
25     File.AppendAllText(behaviourfilename, behaviourtext);
26
27     string physicsText = ballPos.x + ";" + ballPos.y + ";" +
           ballVel.x + ";" + ballVel.y + ";";
28     physicsText = physicsText + leftPlayerPos.x + ";" +
           leftPlayerPos.y + ";" + leftPlayerVel.x + ";" +
           leftPlayerVel.y + ";";
29     physicsText = physicsText + rightPlayerPos.x + ";" +
           rightPlayerPos.y + ";" + rightPlayerVel.x + ";" +
           rightPlayerVel.y + "\n";
30     File.AppendAllText(physicsFilename, physicsText);
31
32     string rulesText = matchState+";"+ leftPlayerScore + "; " +
```

```
            rightPlayerScore + ";" + hitTarget + "\n";
33      File.AppendAllText(rulesFilename, rulesText);
34
35      string commandText = leftPlayerAction + ";" +
            rightPlayerAction + ";" + "Null; Null\n";
36      File.AppendAllText(commandsFilename, commandText);
37
38      yield return new WaitForSecondsRealtime(waitTime);
39    }
40  }
```

Listing 7.2: "writeLogs" method code

As we can see, at each routine invocation, all informations from the various game objects are collected. After that, they are casted to string format and then composed in a unique string divided by semicolons. Once the composition is completed, each string is written to the specific CSV file using the "AppendAllText" method from the "File" class (see line 21 from code 7.2 in example). When all the operations are executed, the routine waits for *waitTime* seconds before starting another cycle, thanks to the infinite while loop managing the routine. Supposing to set *waitTime* to 0.05 seconds, the execution rate is equals to 20 invocations/seconds, so this way the record rate is much lower than the FixedUpdate's one.

Logs record all the games made during the simulation. Referring to the Pong Offline way of working, we know that every time a game ends, ball's transform position is resetted to (0, 0). This can be seen as a resetted state, so, inside the log files, it won't be necessary to introduce a "special state" about a new game because it is implied inside the records themselves.

Here's an extract from a Physics Engine Log (all values have been truncated for better visualization):

Ball Pos		**Ball Vel**		**LP Pos**		**LP Vel**		**RP Pos**		**RP Vel**	
X	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**
-0.6	0.4	-3.2	2.2	-4.0	0.0	0.0	0.0	4.0	0.0	0.0	0.0
-0.6	0.4	-3.2	2.2	-4.0	-0.1	0.0	-3.0	4.0	0.1	0.0	3.0
-0.7	0.5	-3.2	2.2	-4.0	-0.1	0.0	-3.0	4.0	0.1	0.0	3.0
-0.8	0.5	-3.2	2.2	-4.0	-0.2	0.0	-3.0	4.0	0.2	0.0	3.0
0.8	0.6	-3.2	2.2	-4.0	-0.2	0.0	-3.0	4.0	0.2	0.0	3.0
-0.9	0.6	-3.2	2.2	-4.0	-0.3	0.0	-3.0	4.0	0.3	0.0	3.0

Table 7.7: Physics Engine values example

As we can see from this table, some values never change during game

simulation: the right player has always X position equals to +4, while left's one is equals to -4, because these two paddles cannot move on the X axis. For the same reason, players' X velocity component is always 0. In reality, the values recorded are way more precise, reaching 5 or 6 decimal numbers, but they are displayed here with only 1 in order to reach more readability.

In addition, some tests were executed with values rounding or truncation, but I came back to the original ones in order to avoid problems with spurious values returned by support rounding methods. The neural networks behavior also wasn't improved by this preprocessing operations, so they're useless.

Chapter 8

Refined MPAI-SPG Architecture

Starting from MPAI-SPG initial architecture, shown in the diagram below (image 8.1), we thought about a good Server AI implementation.

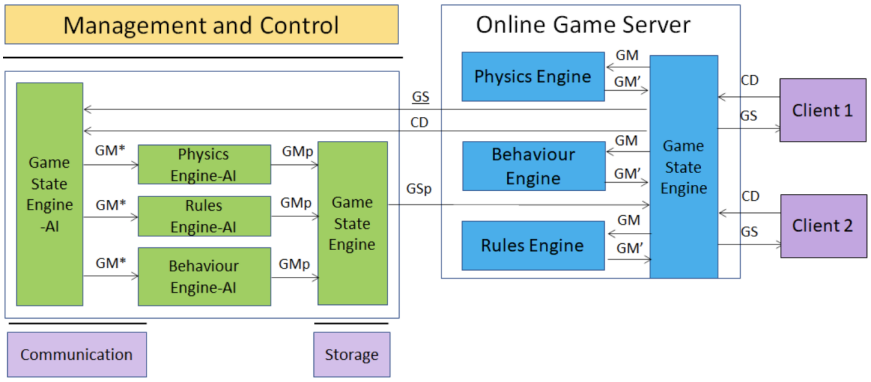

Figure 8.1: MPAI-SPG Original Schema

The aim is to train our Digital Twin's neural networks inside the Server without giving them the exact knowledge about Game Engine's way of working. **This way the standard would be usable also with new technologies.** Making references to image 8.1, we can see that the three Online Game Server components called Physics, Rules and Behaviour Engine (blue boxes) get a set of input data called **GM** and return their results as a new set **GM'**. These sets of data are different depending on the IDE software used to build the game (Unity, Unreal Engine, etc.), so we cannot base our standard on those formats to create a interoperable and long-term usable standard. Instead, we can see that, independently from the platform used, one thing that will be always visible is the **Game State**, so we will base our AI training on that.

Supposing the Game Engine owns the current game state **GS**, at moment **t** the blue "Game State Engine" creates from it a new state **GS** using user's controls **CD** received from clients. At the same time, the current state **GS** is sent to the AI together with **CD**. The Digital Twin has to create and send back the next game state **GSp** at time **t+1** without using the same technique applied by Blue Boxes, but through Prediction instead. **This way the AI should work like a human brain, which creates physics approximations about the world and doesn't make calculations about objects' movements**. This could look like a person who knows the game and, watching it from the outside, knows how the things should go in order to make the game state consistent. This forecasted state **GSp** will be compared with **GS** in order to understand if there is some kind of problem with the network or the players.

Referring to the Pong use case and knowing what has to be predicted about it, we can see that all these informations can be managed by the three AIs: Physics Engine-AI, Behavior Engine-AI and Rules Engine-AI (green boxes from image 8.1). After this consideration, we can conclude that the two boxes on the side ("GameStateEngine-AI" and "GameStateEngine") have not to necessarily be Predictive AIs.

We studied how we could implement this architecture in real life, and we arrived at the conclusion that, also in order to keep a different approach from the one applied by the Online Game Server, the SPG Digital Twin will have to forecast the next game state knowing the ones generated in the past. Thus, we'll have to train our predictive neural networks using this kind of informations: the logs collected on chapter 7.

Starting from this reasoning, we created a new *refined* architecture definition, shown in diagram 8.2 below:

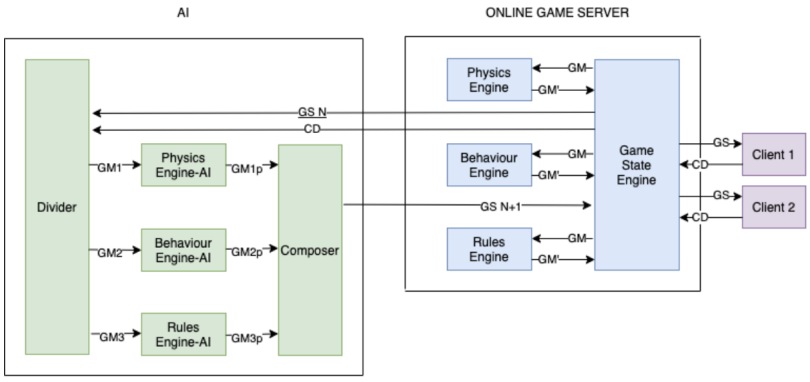

Figure 8.2: MPAI-SPG Refined Architecture

As we can see from diagram 8.2, the Online Game Server sends to its Predictive part the entire game state **GS N** got from the last render, together with user's commands **CD**. Knowning what PhysicsEngine-AI, BehaviourEngine-AI and RulesEngine-AI have to predict, we can state that each one of them doesn't need the entire game state. What has to be performed it's a subdivision of the game state between the three AIs, so the green box known as "Game State Engine-AI" in figure 8.1 becomes the **"Dispatcher"** or **"Divider"**, whose role is to feed each NN with input data, named respectively **GM1**, **GM2** and **GM3**. The three AIs return each a forecasted subset of the game state data, called **GM1p**, **GM2p** and **GM3p**. The last thing to do is to join them into a unique set in order to create the predicted game state **GS N+1**. This operation is performed by the "GameStateEngine", that in the refined architecture is renamed as **"Composer"** or **"Collector"**. Summarizing, each NN will take a subset of the Game State **N** and will have to return a subset of the Game State **N+1**.

The idea is to create an architecture which operates not knowing exactly what kind of data are shared between the "GameStateEngine" and the three sub-engines "Physics Engine", "Behaviour Engine" and "Rules Engine" (blue boxes). Keeping the objective to build a standard, we have to bypass all this technological constraints, so we thought about all these components together as a unique "black box" (the "Online Game Server"), just knowing what are the inputs and the outputs. The inputs are user's control data, while the output is the game state.

8.1 Predictive Neural Networks attributes examples

As I explained before and in section 7.1, each of the three sub neural networks predicts a subset of the rendered game state. Here below are shown the three AIs' inputs and an example of the values contained into them (the values are truncated for readability).

PhysicsEngine:

B Pos		**B Vel**		**L Pos**		**L Vel**		**R Pos**		**R Vel**	
X	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**
-1.2	-0.3	-3.2	2.2	-4.0	0.0	0.0	3.0	-4.0	0.0	0.0	-3.0

Table 8.1: Physics Engine's set of attributes with example

BehaviorEngine:

L Pos		L Vel		R Pos		R Vel	
X	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**
-4.0	0.0	0.0	3.0	-4.0	0.0	0.0	-3.0

Table 8.2: Behavior Engine's set of attributes with example

RulesEngine:

MatchState	L Score	R Score	HitTarget
MatchState	2	0	RWall

Table 8.3: Rules Engine's set of attributes with example

Note: as we can see from these 3 tables, some physical quantities are repeated betwen the Behavior and Physics Engine. This because I'm referring to the general MPAI-SPG definition and not the Pong use case. The next part of this thesis will try to find a suitable architecture for each one of these 3 NNs, in order to implement the "refined" architeture presented earlier. Each one of these nets will predict its subset of attributes and the inference outputs will be combined to create a game state.

Chapter 9

Predictive Neural Networks

In order to develop the refined MPAI-SPG architecture explained in chapter 8, it is necessary to find some suitable network architecture to be used for the predictions.

9.1 Problem explanation

Given a game state, we can see that it is composed by different kinds of data, divided in 3 subsets, each one managed by a different neural network. After receiving a complete state from the Online Game Server (referring to the MPAI-SPG schema in figure 4.3) at time **N**, each neural network has to make a prediction about values assumed at time **N+1** by its data subset.

This tecnique defines a different approach from the one used inside the Unity Game Engine. As explained in chapter 8, between the Online Game Server Engine and the 3 subengines there is a data exchange that we don't precisely know. In addition, we don't want to create a standard defined on a specific technology, but suitable for future implementations and softwares. The new approach I defined takes a series of game states in input and tries to give a future approximation about its values without knowing the engines' way of working, like human's brain approximated measures about physics in real life.

Making a concrete example about the forecasting process, let's analyze table 9.1, which refers to the Behavior Engine subset:

	Left Pos		Left Vel		Right Pos		Right Vel	
Index	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**
1	-4.0	0.3	0.0	3.0	4.0	0.0	0.0	3.0
2	-4.0	0.3	0.0	3.0	4.0	0.0	0.0	3.0
3	-4.0	0.4	0.0	3.0	4.0	0.1	0.0	3.0
4	-4.0	0.4	0.0	3.0	4.0	0.1	0.0	3.0
5	-4.0	0.5	0.0	3.0	4.0	0.2	0.0	3.0
6	-4.0	0.6	0.0	3.0	4.0	0.3	0.0	3.0

Table 9.1: Behavior Engine data subset example

Supposing the engine has already received as input the last 4 rendered game states in the last 4 updates (lines 1-4 on table 9.1), once received its fifth state (line 5), its objective is to predict the 6th state, which is displayed in line 6. Practically, the engine wants to infer which should be the next set of values given the ones from the past, overtaking the latency problems that can occur during game execution. Analyzing table's columns, each one represents a **monovaried time series**, because each one depends only on **one variable**, the **time**. Considering all the columns varying together, we can say that it is a **multivariate time series**. Summarizing, the overall objective is to build three neural networks able to predict the next step of a multivariate time series.

9.2 Neural Networks types

Given all the kinds of Neural Networks developed during the years from AI researchers, the most suitable ones for data forecasting are the Recurrent Neural Networks (RNN). It is possible also to make predictions using the classical Multi-Layer Perceptron (MLP), but with worse results. During my research, I've tried to implement a forecasting MLP, but the obtained results are not so satisfying and significant, so I avoided to talk about them in this thesis.

Generally talking, there are three main NN typologies: "Supervised Learning", "Unsupervised Learning" and "Recurrent Neural Networks". Referring to the first two of them, they differ for the fact that the "Supervised Learning" stands for a network that learns through **labelled examples**, meaning that each input value has got an assigned resulting **label defined a priori** used to make output comparison, while the second type "Unsupervised Learning" means that there's no assigned known label. Referring to the networks created through MLAgents, they are an example of Supervised Learning, because through Reinforcement Learning we define different labels: the rewards assigned to each

action. We'll talk about the Recurrent Neural Networks in section 9.2.2.

A neural network takes inspiration for its definition from the human brain, meaning that it is formed by **neurons**. Thus, considering a NN as a **set of neurons**, it is a *parallel distributed processor* that stores its knowledge inside weights contained in it. These weights are set up through the "Learning" Phase first, while they are used in the "Inference" phase next. Given this particular definition, this kind of software is suitable for parallelization through **accelerators** like GPUs (see 14.2).

Formally, a neural network is composed by three principal elements:

1. Neuron model: the architecture that defines each network component. There are different kinds of it, like the "Perceptron" and the "LSTM Cell"
2. NN Architecture: set of neurons connected through weighted links.
3. Learning Algorithm: the procedure that updates weights based on training data

The different typologies of networks differ on these 3 components: which kind of neurons compose them, how they are connected and how weights are updated.

9.2.1 Multilayer Perceptron

In order to talk about the predictive neural networks I created for MPAI-SPG and all that comes with them, it is convenient to talk about one particular network, the Multilayer Perceptron (MLP). It is one of the most classical nets created ever, that brought many features now common in more complex architectures, so it is useful for contextualization. MLP takes its name by the fact that it is formed by two distinct elements: a set of **neurons** and a set of **layers**.

Perceptron

This is the first neural model ever. It is formally defined by a set of different elements:

1. inputs x: a input vector containing values. Each vector's position is connected, through a single weighted link, to an adder function
2. adder function $\sum$: executes a linear combination of vector's positions, each one of them weighted by the link

3. Activation Function f: gets in input the linear combination returned by the adder function and chooses if the neuron has to be activated or not

The Perceptron structure is defined in image 9.1 shown here below:

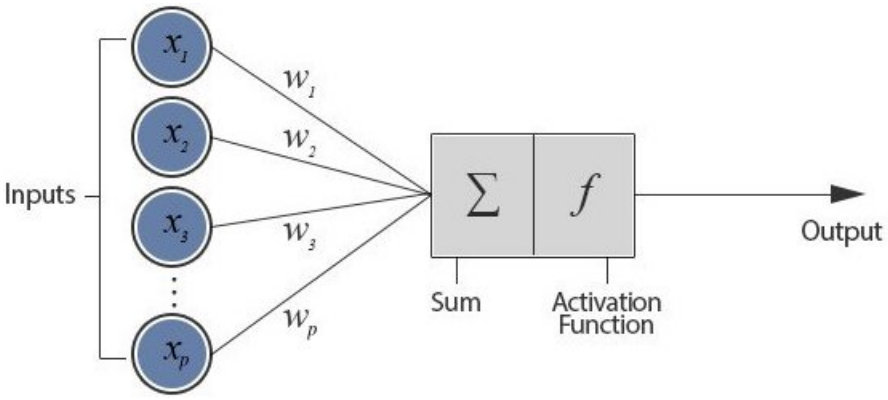

Figure 9.1: Perceptron Schema

Mathematically speaking, the action of the single j-th neuron is defined as

$$y_j = \varphi(u_j - b_j) = \varphi(v_j) \tag{9.1}$$

where:

- y_j is the output,
- u_j is the result of the linear combination (adder)
- b_j is the *bias* of the j-th neuron, whose objective is to apply an affine transformation to the weighted sum u_j.
- v_j is defined as the subtraction of the linear combination u_j with its bias b_j
- φ is the activation function, that takes in input v_j

Specifically, the linear combination of the input vector x with the j-th perceptron is defined as

$$u_j = \sum_{i=1}^{p} w_{ji} \times x_i \tag{9.2}$$

where p is the dimension of the vector, w_{ji} is the weight that connects the i-th input vector's position to the j-th neuron, and x_i is the value contained inside the i-th dimension of the vector.

Multilayer

Second fundamental feature of a Multilayer Perceptron is its composition through a series of 3 different layers, as we can see in image 9.2:

1. Input layer: collects data received in input
2. Hidden Layer: useful for large input layers because it can help extract higher-order dynamics. Given this layer, it takes us the advantage to resolve **non-linearly separable** tasks.
3. Output layer: returns the network output

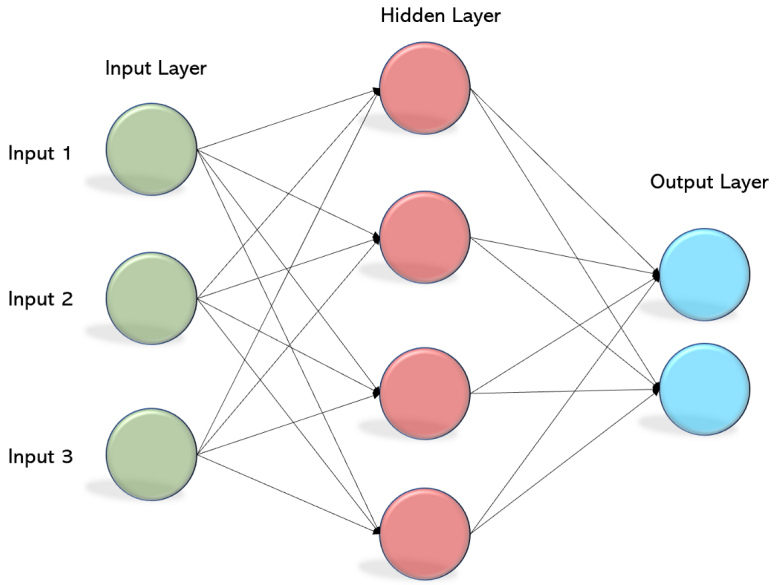

Figure 9.2: Multilayer Perceptron Architecture

Each layer is **fully-connected**, meaning that each neuron has got a weighted link to the all the neurons of the layer behind. This kind of neural network is called **feed-forward**, meaning that data flows from the input layer to the output layer

The typical MLP learning algorithm is the "Backpropagation Algorithm", whose objective is to look for weight values, inside the neural network, that minimize the total error over the examples contained in the training set. The error is represented by the **loss function**, that we'll recall also in the next chapters. This algorithm consists of 2 repeated steps:

1. "Forward pass": passing as input to the network an example taken from the dataset, the total error of each neuron is calculated

2. "Backward pass": the network's weights are updated using the error calculated at the previous step, recursively computing the local gradient of each weight

9.2.2 Recurrent Neural Networks

Once explained how generically a NN is built, we can talk about another kind of network typology, which is fundamental in this research. The Recurrent Neural Network structure is formed by nodes connected in order to form a graph (directed or undirected) in a temporal sequence. That's why they are suitable for **forecasting**. During the years, a huge number of recurrent architectures have been developed, but the most generic one is the "Fully Recurrent Neural Network". This setup is based on the fact that **network outputs are connected to network inputs**, meaning that, during training and inference, the network results will be influenced by the output of the previous calculation, showing some kind of temporal dependence in the network.

As we can see in image 9.3, a Recurrent Neural Network takes some input x_t (some input *x* at time *t*) and returns the output h_t. At the same time, the network contains a loop that takes information from the network and passes it to the next step.

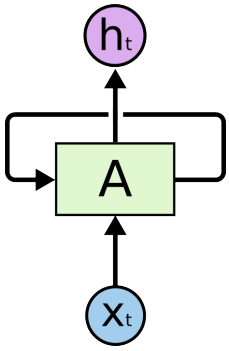

Figure 9.3: Recurrent Neural Network structure

Repeating the loop, we can think about the network as a repetition of the same structure in next time instants, always receiving some information from the previous step, as we can see in image 9.4:

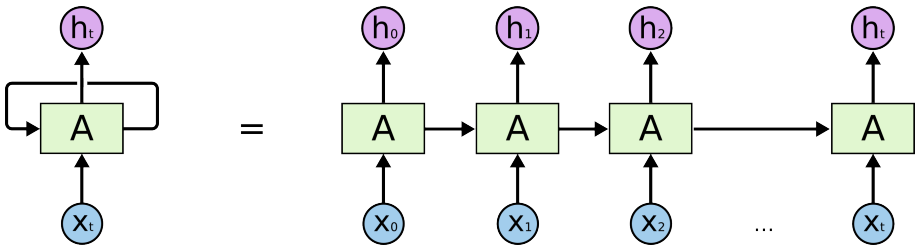

Figure 9.4: Recurrent Neural Network unrolled structure

The classical recurrent network suffers of two problems: the Gradient Explosion and the Vanishing Gradient, which happen during the Backpropagation Algorithm execution. In addition, a classic RNN suffers of the **short memory problem**, meaning that it has difficulty to remember earlier steps in a long sequence.

Vanishing Gradient problem

It is a typical problem for networks that use backpropagation and gradient-based learning algorithms. It consists in the update value too small to effectively change the weights and that, in the worst case, won't change the network at all. As explained in [12], this kind of problem is caused, for example, by some activation functions like the **Hyperbolic Tangent** (see 9.7), which spans in the (0, 1] interval. Knowing that the backpropagation algorithm multiplies these values through the "Chain Rule", supposed to have a network with *n* layers defining it, the gradient signal will exponentially decrease making the training very slow and inefficient.

Gradient Explosion problem

On the inverse, the Gradient Explosion problem happens when large gradients accumulate during training, resulting in big weight updates and creating an **unstable** network, meaning that its values will change dramatically from one iteration to the next. An *unstable* network is basically useless, because it is unable to make predictions, classifications and all the other kind of tasks. Generally talking, the gradient based algorithm works well when the updates are small and controlled.

9.2.3 Long Short-Term Neural Networks

Long Short-Term Memory Neural Networks (LSTM NN) are a kind of Recurrent Neural Network developed by Sepp Hochreiter and Jurgen Schmidhuber and presented in their article [3] "Long Short-term Memory" published in 1997. They were defined in order to overpass the Gradient Explosion, the Vanishing Gradient and the Short Memory problems. They are mainly used as **deep neural networks** (nets with more than 2 hidden layers) and they are provided with feedback connections, thus they can process data sequences and perform predictions based on time series. Thus, they are suitable for forecasting problems and Natural Language Processing, because **they are able to learn long temporal sequences**.

Unlike other neural network architectures, provided with the classical "Perceptron", LSTMs are composed by a set of neurons based on a model called "cell", whose schema is presented in figure 9.5.

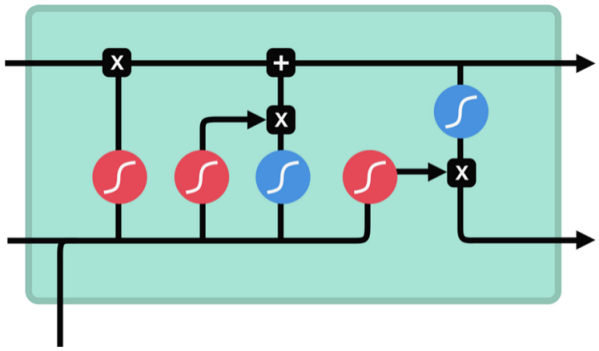

Figure 9.5: LSTM Cell schema. Red Symbol: sigmoid function. Blue symbol: tanh function

A cell contains a **state** and a group of **gates** that controls the information flow inside the unit. The cell state has the aim to collect informations taken from training in a long time, while gates are used to select the relevant informations. Gates are formed by two different kinds of function: a "Sigmoid" function, also called "Logistic Function", and a "Hyperbolic Tanget" function, also called "Tanh".

Sigmoid function is mathematically defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{9.3}$$

and its plot is displayed in figure 9.6

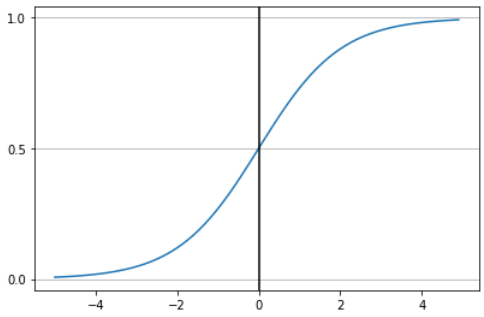

Figure 9.6: Sigmoid function

The sigmoid function is the most classical activation function used in NNs, and its objective is to normalize values inside the range [0, 1]. Given this characteristic, it is useful for "filtering" operations, deciding what informations delete multiplying them by 0 and what to keep, multiplying them by 1.

Tanh function, instead, is used to regulate the values flow inside the network, executing a normalization in the range [-1, 1]. It is defined as:

$$tanh(x) = \frac{sinh(x)}{cosh(x)} \tag{9.4}$$

where $sinh(x)$ is the "hyperbolic sine" function and $cosh(x)$ is the "hyperbolic cosine" function. The $tanh(x)$ plot is diplayed in figure 9.7.

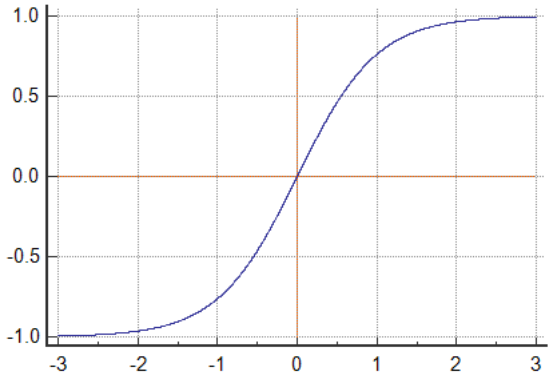

Figure 9.7: Hyperbolic Tanget function

Basically, a LSTM Network works like a classic Recurrent Neural Network, where some data is transmitted forward for next iterations, but using different operations inside the neurons.

9.2.4 LSTM Cells

Every Long Short-Term Memory Network cell owns a **state** that propagates relevant informations and creates the first improvement brought by LSTMs: **they can maintain information about the oldest data received in input also after many steps**. This fact reduces the "short-memory" problem that affects the classic RNN.

The information management inside the cell is performed by gates. Any LSTM cell has got 3 gates: the "input" gate, the "forget" gate and the "output" gate. Each one of them has a different objective:

1. Forget Gate: decides what is important to keep
2. Input Gate: states what information is relevant to add from the current step
3. Output Gate: states what the next hidden state should be

The cell keeps track of two different states: an explicit state and an hidden state. The latter contains informations about previous inputs and it's used to make predictions.

Forget Gate

Aim of this gate is to decide what information to keep and what to throw away. This component works as follows: taking data from the previous hidden state *prevhiddenstate* and from the current input, these are summed together and then passed as argument to the sigmoid function. Knowing the sigmoid function's way of working, we know we'll receive the output data *forgetout* in the interval [0, 1]: **if *forgetout* is near to 0, we want to forget, while if it is near to 1, we want to keep**. Summarizing, the operation performed is the following:

$$forgetout = sigmoid(prevhiddenstate + input) \tag{9.5}$$

Input Gate

Also this gate takes the same elements in input: the previous hidden state and the current input. Two operations are performed inside the gate:

1. Update decision: through sigmoid function, it is decided what values will be updated passing the two previously cited elements in input. If we receive back 0, the value won't be updated, while if we receive 1 it will. We call the output *sigmoidout*.

2. Regularitzation: this operations has the objective to regularize the network. We take the same input and we pass it to the tanh function in order to normalize it in [-1, 1]. We call the output *tanhout*.

Once collected these results, the cell multiplies *sigmoidout* and *tanhout*, with the consequence that basically *sigmoidout* will "filter" the important informations from *tanhout*. Supposing to call the Input Gate's output *inputout*, the operations described can be mathematically written as follows:

$$sigmoidout = sigmoid(prevhiddenstate + input) \tag{9.6}$$

$$tanhout = tanh(prevhiddenstate + input) \tag{9.7}$$

$$inputout = sigmoidout \times tanhout \tag{9.8}$$

Cell state

Given all the informations calculated so far, we multiply *forgetout* with the previous explicit cell state *prevstate* executing a filtering operation through a pointwise multiplication that deletes state's values multiplied near 0. This computation returns a filtered state which will be added in a pointwise manner with *inputout* returning the updated explicit cell state, called *updatedstate*.

Summarizing all the informations given so far, we can say that the updated cell state *updatedstate* is calculated as follows:

$$updatedstate = forgetout \times prevstate + inputout \tag{9.9}$$

The cell structure, as we can see from figure 9.5, is built in a way that the cell state's output *updatedstate* takes two different roads: the first one will keep track of it, becoming the new explicit cell state, while the second one will bring it to the last step, called "Output Gate".

Output Gate

Objective of this gate is to decide what which be the next **hidden state**, which is a set of informations used for predictions. Inside this gate are performed different operations:

1. execution of the sigmoid function, taking as argument the previous hidden state and the input of the cell

$$sigmoidout = sigmoid(prevhiddenstate + input) \tag{9.10}$$

2. execution of the tanh function, passing *updatedstate*

$$tanhout = tanh(updatedstate) \tag{9.11}$$

3. the output of the two previous operations are multiplied together in order to decide what informations to keep inside the hidden state, returning the updated hidden state

$$hiddenstate = sigmoidout \times tanhout \tag{9.12}$$

This new hidden state *hiddenstate*, together with the new explicit cell state *updatedstate*, will go forward to the next cell execution in the following timestep.

9.3 Reasons why to choose LSTM NNs

Summing up all the informations given before, the reasons why I've chosen the LSTM Neural Networks instead of other architectures are:

1. Vanishing Gradient and Gradient Explosion resolution: this kind of network resolves this two tipical problems
2. Longer Memory: LSTMs overpass the "short memory problem", common in classic RNNs, that's why they are called "Long Short-Term": they can remember past input values, allowing us to make better predictions using a bigger number of past records.
3. Unity execution: even if this kind of network is not guaranteed by Barracuda developers to be executed in a constant and little time, it can be used through this framework to perform inference inside a Unity project
4. Better results: relating to other architectures, this kind of networks return better predictions than others, like Multilayer Perceptron for example

Chapter 10

Refined architecture components introduction inside Pong

After the new Refined MPAI-SPG Architecture was defined, as explained in chapter 8, I implemented it in Unity in order to execute predictions inside the game. To do this, I created 5 different scripts, each one representing one component of the new architecture:

1. Dispatcher (or Divider)
2. Collector (or Composer)
3. Physics Engine Predictor
4. Behaviour Engine Predictor
5. Rules Engine Predictor

All these scripts extend the C# "MonoBehavior" class defined in the Unity Framework. This class extension enables the scripts to be attached to a game object inside a Unity scene. This way, I created an object called "MPAI-SPG Predictor", which takes all these components and whose task is to get game state informations, make predictions and return the output.

In the following sections, I'm describing all the implemented scripts.

10.1 Dispatcher

When enabled, this script activates all the other SPG components, in order to avoid synchronization problems. At each frame render, the Dispatcher executes the "updateData" method. As we can see in code listing 10.1, this function gets status informations, like objects' position and velocity, and creates 3 different vectors, that will be used as input by the 3 prediction engines. These vectors have been formally defined by the MPAI-SPG standard, as already explained in section 7.1.

```
1  void OnEnable() {
2    //Activating other SPG Components
3    GetComponent<PhysicsEngine>().enabled = true;
4    GetComponent<BehaviourEngine>().enabled = true;
5    GetComponent<RulesEngine>().enabled = true;
6    GetComponent<Collector>().enabled = true;
7  }
8
9  void Update() {
10   if (ball != null && leftPlayer != null && rightPlayer != null) {
11     // Getting references to game objects' components
12     ballControl = ball.GetComponent<BallControl>();
13     leftPlayerControl = leftPlayer.GetComponent<PlayerControls>();
14     rightPlayerControl = rightPlayer.GetComponent<PlayerControls
           >();
15
16     // Updating physical quantities values
17     updateData();
18
19     // Printing collected data on external files if flag enabled
20     if (writeData) {
21       printDataVectors();
22     }
23   }
24 }
25
26 // Returns input data for the Physics Engine
27 public float[] getPEInputData() {
28   return peDataVector;
29 }
30
31 // Returns input data for the Behaviour Engine
32 public float[] getBEInputData() {
33   return beDataVector;
34 }
35
36 // Returns input data for the Rules Engine
```

```
37 public float[] getREInputData() {
38   return reDataVector;
39 }
40
41 // Updates input data for the 3 engines
42 void updateData() {
43   // Getting updated informations
44   Vector2 ballPos = ballControl.GetPosition();
45   Vector2 ballVel = ballControl.GetVelocity();
46   Vector2 leftPlayerPos = leftPlayerControl.GetPosition();
47   Vector2 leftPlayerVel = leftPlayerControl.GetVelocity();
48   Vector2 rightPlayerPos = rightPlayerControl.GetPosition();
49   Vector2 rightPlayerVel = rightPlayerControl.GetVelocity();
50
51   // Physics Engine's input data vector creation
52   peDataVector = new float[12];
53   peDataVector[0] = ballPos.x;
54   peDataVector[1] = ballPos.y;
55   peDataVector[2] = ballVel.x;
56   peDataVector[3] = ballVel.y;
57   peDataVector[4] = leftPlayerPos.x;
58   peDataVector[5] = leftPlayerPos.y;
59   peDataVector[6] = leftPlayerVel.x;
60   peDataVector[7] = leftPlayerVel.y;
61   peDataVector[8] = rightPlayerPos.x;
62   peDataVector[9] = rightPlayerPos.y;
63   peDataVector[10] = rightPlayerVel.x;
64   peDataVector[11] = rightPlayerVel.y;
65
66   // Behaviour Engine's input data vector creation
67   beDataVector = new float[8];
68   beDataVector[0] = leftPlayerPos.x;
69   beDataVector[1] = leftPlayerPos.y;
70   beDataVector[2] = leftPlayerVel.x;
71   beDataVector[3] = leftPlayerVel.y;
72   beDataVector[4] = rightPlayerPos.x;
73   beDataVector[5] = rightPlayerPos.y;
74   beDataVector[6] = rightPlayerVel.x;
75   beDataVector[7] = rightPlayerVel.y;
76
77   // Rules Engine's Input Data Vector creation
78   reDataVector = new float[4];
79   reDataVector[0] = 0;
80   reDataVector[1] = GameManager.PlayerScore1;
81   reDataVector[2] = GameManager.PlayerScore2;
82   reDataVector[3] = 0;
83 }
```

Listing 10.1: "Dispatcher" class methods

This class also permits each prediction engine to get its new input vector through the corresponding getter methods defined at lines 27, 32, 37 in code listing 10.1. In addition, in order to keep track of the data movement, this script optionally writes down collected values on an external CSV file.

10.2 Prediction Engines

Main part of the Refined MPAI-SPG Architecture implementation, this script has the objective to make a prediction through data taken from the Dispatcher. This operation is possible thanks to the Barracuda framework, that enables us to use a inference engine inside a Unity project.

In this section are explained the different methods composing the "PhysicsEngine.cs" script. This code also contains support components and methods adopted to make an analysis of the inference execution, like the "lineRenderer" tool which is used to draw lines inside the field referring to ball's trajectory. The "BehaviorEngine.cs" and "RulesEngine.cs" follow an analogous implementation.

10.2.1 Start method

```
1  void Start() {
2    // Inference Engine Setup
3    runtimeModel = ModelLoader.Load(model);
4    engine = WorkerFactory.CreateWorker(runtimeModel, WorkerFactory.
         Device.CPU);
5
6    dp = GetComponent<Dispatcher>();
7    cl = GetComponent<Collector>();
8    inputRows = 0; //number of rows currently filling the input
         matrix
9
10   // matrix of dimensions [timesteps x featuresNumber] creation
11   // This will be converted into a input tensor
12   this.data = new float[timesteps][];
13   // Input data matrix initialization
14   for (int i = 0; i < timesteps; i++) {
15     this.data[i] = new float[featuresNumber];
16   }
17
18   // Matrix containing the last [timesteps] inference outputs
         creation
19   predictionResults = new float[timesteps][];
20   for (int i = 0; i< timesteps; i ++) {
```

```
21      predictionResults[i] = new float[featuresNumber];
22    }
23
24    // Cleaning data collection file
25    File.WriteAllText(filename, physicsheader);
26    predictionNumber = 0; // index of the last prediction made
27
28    // Cleaning output data collection file
29    File.WriteAllText(predictionFilename, physicsheader);
30
31    // Inference engine output vector creation
32    predictionOutputs = new float[featuresNumber];
33    for (int i = 0; i < featuresNumber; i++) {
34      predictionOutputs[i] = 0;
35    }
36
37    // Line Renderers initialization
38    lr = GetComponent<LineRenderer>(); // draws ball's predicted
          trajectory
39    childLr = transform.GetChild(0).GetComponent<LineRenderer>(); //
           draws ball's correct trajectory
40
41    // Starting prediction routine
42    StartCoroutine(predictionFunctionEnumerator());
43
44    previousTime = 0f; // past inference execution time needed
45    timeList = new List<float>(); // list of previousTime values,
          updated after each inference execution
46
47    // Matrix containing ball's correct positions creation
48    this.correctPositions = new float[timesteps][];
49    for (int i = 0; i < timesteps; i ++) {
50      this.correctPositions[i] = new float[2];
51    }
52  }
```

Listing 10.2: PhysicsEngine class "Start" method code

The "Start" method is invoked at the script instantiation, and it is used to initialize all the variables and components. First of all, using the built-in Barracuda methods, the neural network is loaded, creating the *runtimeModel*. From this model, the prediction engine is built through the "CreateWorker" method. After that, the input matrix is initialized. Given the number of time steps "*timesteps*" necessary to the network in order to make predictions, and given the number of features "*featuresNumber*" the network has to forecast, a $timesteps \times featuresNumber$ matrix is created and initialized with zeros. At each step, this matrix will be updated with the new values taken from the Dis-

patcher. Other matrices are initiliazied in the same way, like *correctPositions*, *predictionOutputs* and *predictionResults*.

Inside this script are also maintained data structures for time measures collections, in order to study the general behavior of the script and the neural networks used.

10.2.2 Prediction routine

```
1  IEnumerator predictionFunctionEnumerator() {
2    while (true) { // infinite cycle
3      if (inputRows == timesteps) { // input matrix is totally
           filled with data
4        // Getting the prediction routine starting time
5        previousTime = Time.realtimeSinceStartup;
6
7        updateDataArray(); // updating matrix with new data from
           Dispatcher
8        if (writeLog) {
9          printDataArray(); // printing data on external file
10       }
11
12       // Prediction execution
13       using(var input = createTensor()) { // Creation of the input
             tensor
14         // Inference execution
15         Tensor output = engine.Execute(input).PeekOutput();
16
17         // Waiting for prediction completion (Necessary for
               IEnumerator execution)
18         yield return new WaitForCompletion(output);
19
20         input.Dispose(); // deleting input matrix
21
22         // Saving prediction results as floats vector
23         predictionOutputs = output.AsFloats();
24
25         output.Dispose(); // deleting output matrix
26
27         // Saving the new output vector inside the output matrix
28         updatePredictionsArray(predictionOutputs);
29       }
30
31       // Calculates and records passed time
32       if (recordDelayTime) {
33         currentTime = Time.realtimeSinceStartup;
34         float passedTime = currentTime - previousTime;
35         timeList.Add(passedTime);
```

```
36        }
37
38        // Writing prediction outputs on external file
39        if (writeLog) {
40          printResultsRow(predictionOutputs);
41        }
42
43      } else { // Not reached enough input data
44        addDataRow(); // adding new data inside the input matrix
45      }
46
47      // (Necessary for IEnumerator execution)
48      // We don't need to wait, so it takes 0 as argument
49      yield return new WaitForSeconds(0);
50    }
51  }
```

Listing 10.3: Prediction Routine code

The Prediction Routine has many tasks to perform:

1. Data matrix management: the integer variable *inputRows* is maintained in order to know if the matrix is filled with enough data rows (we need *timesteps* of them) and so ready for being used as inference input. *inputRows* keeps tracks of how many rows are currently available inside the input matrix. From the script start, a number of "predictionFunctionEnumerator" invocations equals to *timesteps* has to be performed in order to collect required data before making a prediction. If *inputRows* is not equal to *timesteps* , then the "addDataRow" method is invoked (see line 44). This method fills the first empty matrix row using data taken from the Dispatcher and, once completed, *inputRows* is increased.

   On the other side, if the *data* matrix is already full, the "updateMatrix" method is invoked. It has the objective to take a new vector from the Dispatcher and add it to the last matrix row, overwriting it. Before executing this operation, matrix rows are translated one position up, so that, for example, row at position 2 becomes the row at position 1. In other words, this matrix is a **FIFO queue**. This way, **we can express the time flow moving up the rows inside the matrix**. Given a matrix with rows indexes from 0 to *timesteps* excluded, row at position 0 represents the oldest dataset, while the one at position *timesteps-1* contains the newest vector, with the current game state values. In other words, data rows contained in the upper part of the matrix are older than the ones contained in the lower part. The presented example graphically shows the update phenomenon. Suppose the Physics Engine works with a neural network

with *timesteps* = 5. This means that, when the input matrix will contain 5 concrete rows, it will start making predictions. As predictionFunctionEnumerator routine starts, the engine takes from the Dispatcher a new values row and updates the matrix, meaning that each row will be translated one position up: the result is that, referring to table 10.1, the row at index 0 **will be lost**, while the new vector will be added in position 4, returning a situation like the one presented on table 10.2.

	Ball Pos		**Ball Vel**		**L Pos**		**L Vel**		**R Pos**		**R Vel**	
Index	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**
0	0.0	-1.1	-3.2	2.2	-4.0	0.4	0.0	-3.0	4.0	-0.4	0.0	3.0
1	-0.1	-1.0	-3.2	2.2	-4.0	0.3	0.0	-3.0	4.0	-0.3	0.0	3.0
2	-0.2	-0.9	-3.2	2.2	-4.0	0.2	0.0	-3.0	4.0	-0.2	0.0	3.0
3	-0.3	-0.9	-3.2	2.2	-4.0	0.1	0.0	-3.0	4.0	-0.2	0.0	3.0
4	-0.3	-0.8	-3.2	2.2	-4.0	0.2	0.0	3.0	4.0	-0.1	0.0	3.0

Table 10.1: Physics Engine's data matrix before update

	Ball Pos		**Ball Vel**		**L Pos**		**L Vel**		**R Pos**		**R Vel**	
Index	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**X**	**Y**
0	-0.1	-1.0	-3.2	2.2	-4.0	0.3	0.0	-3.0	4.0	-0.3	0.0	3.0
1	-0.2	-0.9	-3.2	2.2	-4.0	0.2	0.0	-3.0	4.0	-0.2	0.0	3.0
2	-0.3	-0.9	-3.2	2.2	-4.0	0.1	0.0	-3.0	4.0	-0.2	0.0	3.0
3	-0.3	-0.8	-3.2	2.2	-4.0	0.2	0.0	3.0	4.0	-0.1	0.0	3.0
4	-0.4	-0.8	-3.2	2.2	-4.0	0.3	0.0	3.0	4.0	0.0	0.0	3.0

Table 10.2: Physics Engine's data matrix after update

Returning to the script 10.3, we note that it also contains a boolean flag, called *networkLatency*, which says to the script, if set to *true*, that SPG detected some network latency or packet loss, and so it is necessary to fill the input matrix with the engine's previous inference outputs. This is basically the aim of the MPAI-SPG Architecture: **being able to determine the next game state without knowing data received from clients**. Otherwise, if *networkLatency* is set to *false*, the script keeps taking new data from the Dispatcher.

2. Tensor creation: A Tensor is a **multimodal** (or **multidimensional**) data structure used by Barracuda to manage input and output data, and in general very common inside Machine Learning applications. On each di-

mension, this object represents a **mode**, and we can think about a tensor as the most general form of matrix. For example, a vector or an array is a 1D tensor, while a matrix, an array of arrays, is a 2D tensor. At the same time, a 3D matrix, which pratically is an array of matrices, is formally defined as a 3D tensor. We can apply this reasoning with an increasing number of dimensions, also referring to image 10.1, arriving at the conclusion that a tensor is a multidimensional array.

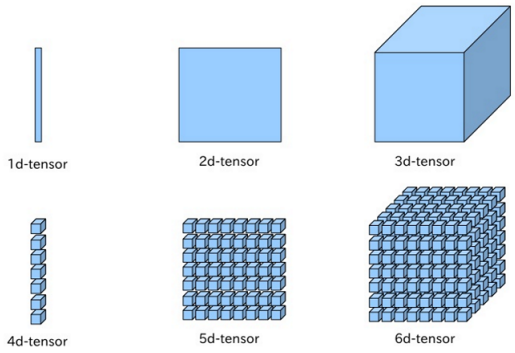

Figure 10.1: Tensors examples

In order to make Barracuda work, the "createTensor" method is called. Its task is to create a Tensor with dimensions (1, 1, featuresNumber, time steps) and fill it up with values contained inside the "data" matrix

3. Prediction: given the new tensor, a prediction is performed using Barracuda "Execute" method and returned as a vector of floating point numbers.

4. Data track: at each routine execution, all input and output data are written inside a CSV file to keep track of the experiments made.

5. Drawings: in order to get a graphical response about the predictions, the "drawLine" method is invoked with the aim to create an interpolation line between the ball's positions collected inside the matrices. On the field are basically drawn two kinds of information: ball's correct trajectory, and ball's predicted trajectory.

6. Time tracking: with the aim to calculate the time needed by Barracuda to execute the prediction method using a certain NN, a time difference is performed between the instant time measured at the method start (see

line 5) and the one recorded at method end (see line 34). The difference is then added to a list of float numbers, which will be written on an external CSV file at application quit.

The prediction computation through a routine was necessary in order to make the code asynchronous compared to the main Unity execution thread. I needed to make it asynchronous because the inference through a LSTM Network is computational expensive and time consuming, making the game freeze and thus unplayable. With this solution, it is possible to perform predictions without making the game wait for their completion at each invocation, so SPG can work indipendently from the Pong main thread.

10.2.3 New Data

```
1  // Gets a new data vector from Dispatcher
2  float[] getNewData() {
3    float[] res = null;
4    if (networkLatency) {
5      res = new float[featuresNumber];
6      for (int i = 0; i < featuresNumber; i++) {
7        res[i] = predictionOutputs[i];
8      }
9
10     if (inputRows == timesteps) {
11       float[] dispatcherRes = dp.getPEInputData();
12       if (dispatcherRes != null) {
13         // Adding info to correct positions
14         updateCorrectPositionsArray(dispatcherRes);
15       }
16     }
17   } else {
18     res = dp.getPEInputData();
19     if (inputRows == timesteps && res != null) {
20       // Adding info to correct positions
21       updateCorrectPositionsArray(res);
22     }
23   }
24   return res;
25 }
```

Listing 10.4: "getNewData" method code

The "getNewData" method's task is to manage the new data row. It can choose the source where to take data depending on the *networkLatency* value:

- *false*: Instantiate a new float array and fills it with data taken from the Dispatcher through the "getPEInputData" method.

- *true*: the new data row will be the last prediction made by the engine

In both the situations, there is the same condition evaluation: if the data matrix is full ($inputRows == timesteps$), then the *correctPositions* array is updated in the same way the *data* matrix is updated. This matrix will be used to draw ball's trajectories.

When a score is made, the "resetData" method is invoked in order to reset the input data matrix and make coherent predictions, because the ball will be in the "resetted" state (0, 0).

Note that in this thesis only the PhysicsEngine script is presented. Referring to the Rules and Behaviour Engine, they work in an analogous way.

10.3 Collector

Last script of the SPG Architecture, this class has the objective to get outputs from the 3 engines and collect them inside a new game state, defined as a 16-dimensional vector representing the **predicted state**. Once created, the Collector writes the new set of values inside an external CSV file.

```
1  void Update() {
2    // Getting inference results
3    float[] reResults = re.GetOutput();
4    float[] beResults = be.GetOutput();
5    float[] peResults = pe.GetOutput();
6
7    // If the results are correct and the writeLog flag is enabled
8    if (reResults != null && peResults != null && beResults != null
         && writeLog) {
9      predictionsResults = new float[16];
10     predictionsResults[0] = peResults[0]; // ball x position
11     predictionsResults[1] = peResults[1]; // ball y position
12     predictionsResults[2] = peResults[2]; // ball x velocity
13     predictionsResults[3] = peResults[3]; // ball y velocity
14     predictionsResults[4] = peResults[4]; // leftplayer x position
15     predictionsResults[5] = peResults[5]; // leftplayer y position
16     predictionsResults[6] = peResults[6]; // leftplayer x velocity
17     predictionsResults[7] = peResults[7]; // leftplayer y velocity
18     predictionsResults[8] = peResults[8]; // rightplayer x
           position
19     predictionsResults[9] = peResults[9]; // rightplayer y
           position
20     predictionsResults[10] = peResults[10]; // rightplayer x
           velocity
21     predictionsResults[11] = peResults[11]; // rightplayer y
           velocity
```

```
22      predictionsResults[12] = 0; // game state
23      predictionsResults[13] = reResults[0]; // left player Score
24      predictionsResults[14] = reResults[1]; // right player score
25      predictionsResults[15] = 0; // hit target
26
27      // Writing predicted game state inside a CSV file
28      File.AppendAllText(filename, "PREDICTION;NUMBER;" +
            predictionNumber.ToString() + "\n");
29      File.AppendAllText(filename, header);
30      predictionNumber++;
31
32      // Casting prediction results to string and then writing them
            down into the file
33      string row = "";
34      for (int i = 0; i < 16; i++) {
35        row = row + predictionsResults[i].ToString() + ";";
36      }
37      row = row + "\n";
38      File.AppendAllText(filename, row);
39    }
40  }
```

Listing 10.5: Collector's "FixedUpdate" method code

As we can see from the code listing 10.5 above, these operations are performed at each frame render. Also, a boolean flag named "writeLog" is present, used to control the Collector's actions from the Unity Inspector. The game state is parsed into a string and then written into a CSV file.

After introducing these implemented components, we can correctly infer game states inside the Pong Online and Offline applications, receiving visual and written feedbacks about the engines' behaviour. In addition, with simple code alterations, it could be possible to modify game objects' transform in order to enable SPG to modify the game state itself.

Chapter 11

Neural Networks Study

In this chapter I will explain the training tests I've made in order to find a good neural network architecture and the tools used. As I introduced in section 3.3, I adopted the TensorFlow framework to create scripts for networks creation and test.

TensorFlow, through the Keras Python API, offers three different methods for NN management:

- "fit": training function executed on a network model previously defined and that takes as argument: epochs number (number of times to use a certain dataset during training), batch size (size of the training subsets), validation set (dataset used for validation)
- "validate": it executes a network test (with data never seen before by the network) at the end of each epoch. It is automatically invoked if a validation set is passed as argument to the "fit" method
- "evaluate": method for neural network test, executed at the end of the training algorithm. It performs a test analogous to validation, using data never seen before

These three methods define as many different stages of network creation. Supposing to have a unique set of records generated from the PongOffline application, it is divided in 3 subsets, each one associated to a stage. Thus, each phase will use its own dataset to perform its operations. Generally, a complete dataset is divided in: 60% for training, 20% for validation, 20% for evaluation.

In general, criterions used to evaluate a NN can be: loss function, accuracy, and custom metrics. TensorFlow offers a wide variety of evaluation metrics, and

each one of them is used during the 3 stages defined before. During training, evaluation and validation, the metrics I adopted are:

- Mean-Squared Error (MSE): it is mathematically defined as

$$MSE = \frac{\sum_{i=1}^{n}(x_i - \hat{x}_i)^2}{n} \tag{11.1}$$

  where x_i is the correct value (in our case, the label), while $\hat{x}_i$ is the output returned by the network

- Mean-Absolute Error (MAE): it is defined as

$$MAE = \frac{\sum_{i=1}^{n}(x_i - \hat{x}_i)}{n} \tag{11.2}$$

  It is simply an average of the differences between the correct values (x_i) and the outputs ($\hat{x}_i$) returned by the network

- Accuracy: Metric offered by TensorFlow, defined as

$$accuracy = \frac{frequency}{total} \tag{11.3}$$

  where $frequency$ is the number of times the output of the network is equal to the label, while $total$ is the total number of comparisons made.

11.1 Training script

Once formally defined a Long Short-Term Neural Network in chapter 9 and the TensorFlow main methods in the previous section, it is time to present the Python script I created for network training, validation and evalutation and executed inside the NNTA environment (see 14.2). This script is composed by distinct sections:

1. Input preprocessing (see section 11.1.1)
2. Neural Network definition (see section 11.1.2)
3. Training and Validation (see section 11.1.3)
4. Evaluation (see section 11.1.4)

11.1.1 Input preprocessing

The dataset is composed by a series of CSV files, each one containing a series of vectors (records), where each position refers to a different feature or physical quantity. As explained in chapter 9.1, **our LSTM network has to predict a new vector given an array (a set of ordered vectors) in input**, so it is necessary to create these arrays and associate to each one of them the subsequent vector to be predicted, the **label**.

To generate the records sets, I executed at the same time multiple PongOffline instances inside the LPA machine (see section 14.3). Each one of them generates its own files set. At the end of the generation, given these very big files, I divided them in files with dimensions around 28MB, each one containing about 500000 records. The complete dataset is thus composed by multiple files, not three big ones associated to a SPG network each. One of the reasons this operation is necessary is that it is practically impossible to allocate all these informations inside the RAM, because it will generate a **memory oveload**, so it is useless to collect all the data inside a single file. In order to avoid this space problem, TensorFlow offers a useful feature called "Generator", a sort of iterator that takes a list of files in input and reads them one at a time, thus reducing the memory space required. This kind of feature is necessary also on computers like "Neural Network Training Asset" (NNTA), with 32 GB of RAM available. Once the generator has loaded a file, it executes the preprocessing operations shown in code listing 11.1, that return a set of 2D arrays, each one labelled with a vector. Formally, this is a "Supervised Training Algorithm" (see 9.2), where a 2D matrix represents the input object, while the following vector taken from the CSV file represents the label, the set of values that the network has to predict.

Given a certain input file, generator's objective is to create a set of **batches**, each one containing **batch_size** couples of the type (array, label). In order to execute this operation, it takes a **lambda** function as argument, the **generate_batches** method, together with the set of filenames forming the dataset. Batch generation is defined in the following code listing:

```
1 # Given a CSV file line, parses it returning a list of elements
2 def manageLine(line):
3   parsed = []
4
5   # Given a row (string), splits it where ";" is present
6   splitted = line.split(";")
7   i = 0
8   while i < len(splitted): # for each splitted element
9     if i % 2 == 0: # left column
```

```
10        newleft = splitted[i].replace("(", "") # deletes left round
              bracket
11        newleft = newleft.replace(",", "") # deletes commas
12        parsed.append(float(newleft)) # float cast
13      else: # right column
14        newRight = splitted[i].replace(")", "") # deletes right
              round brackets
15        if "\n" in newRight:
16          newRight = newRight.replace("\n", "") # deletes newline
                character
17        parsed.append(float(newRight))
18
19      i = i + 1
20
21      return parsed
22
23  # Returns a list of matrices and a list of associated labels
24  # In the two returned lists, matrix in position i is associated to
        label in position i
25  def manageInput(lists, timesteps):
26    resX = list()
27    resY = list()
28    i = 0 # current start position
29    while i < (len(lists) - timesteps): # while it is possible to
        build a couple (input, output)
30      currList = []
31      j = 0 # timestep counter
32      while j < time_steps:
33        currList.append(lists[i + j])
34        j = j + 1
35
36        resX.append(numpy.array(currList)) # Note that numpy
              automatically deletes 0s from float decimal part
37        resY.append(numpy.array(lists[i+j]))
38
39        i = i + 1 # shift
40
41    return numpy.array(resX), numpy.array(resY)
42
43  # Given a list of filenames, executes input preprocessing one file
        at a time
44  def generate_batches(files, batch_size):
45    counter = 0 # file index
46
47    while True: # loops infinitely
48      filename = files[counter] # getting filename
49
50      counter = (counter+1)%len(files)
51      filepointer = open(data_directory + filename, "r") # opens the
```

```
        CSV file in "read" mode
52
53      parsedFile = [] # csv file content after first preprocessing
54      header = 0
55      for line in filepointer:
56        if header == 0: # deletes the first line (labels)
57          header = 1
58        else:
59          parsedFile.append(manageLine(line))
60
61      input, output = manageInput(parsedFile, time_steps) # creates
          inputs and labels lists
62
63      for local_index in range(0, input.shape[0], batch_size):
64        input_local = input[local_index:(local_index+batch_size)]
65        output_local = output[local_index:(local_index+batch_size)]
66
67        yield input_local, output_local
68
69
70  # Getting dataset filenames
71  files = [] # list of files contained inside "data_directory" path
72  for (dirpath, dirnames, filenames) in os.walk(data_directory):
73    files.extend(filenames)
74    break
75
76  # Divides the files between training set, validation set and
      evaluation set
77  # This is necessary because most of the files are used for
      training
78  train_files = []
79  validation_files = []
80  test_files = []
81  for i in range(0, len(files)):
82    if (i < training_files_number):
83      train_files.append(files[i])
84    elif (i >= training_files_number and i < (training_files_number
        + validation_files_number)):
85      validation_files.append(files[i])
86    elif (i >= (training_files_number + validation_files_number)):
87      test_files.append(files[i])
88
89  # Calculating steps per epoch for each set:
90  steps_per_epoch_train = 0
91  total_train_records = 0
92  for filename in train_files:
93    lines_number = sum(1 for line in open(data_directory+filename))
94    total_train_records = total_train_records + lines_number
95    steps_per_epoch_train = steps_per_epoch_train + (lines_number/
```

```
      batch_size)

steps_per_epoch_val = 0
total_validation_records = 0
for filename in validation_files:
  lines_number = sum(1 for line in open(data_directory+filename))
  total_validation_records = total_validation_records +
      lines_number
  steps_per_epoch_val = steps_per_epoch_val + (lines_number/
      batch_size)

steps_per_epoch_test = 0
total_test_records = 0
for filename in test_files:
  lines_number = sum(1 for line in open(data_directory+filename))
  total_test_records = total_test_records + lines_number
  steps_per_epoch_test = steps_per_epoch_test + (lines_number/
      batch_size)


# Creating the three different datasets
# "output_shapes" are the input and output shapes
train_dataset = tf.data.Dataset.from_generator(
  generator = lambda: generate_batches(files=train_files,
      batch_size = batch_size),
  output_types = (tf.float32, tf.float32),
  output_shapes = ([None, time_steps, features_number], [None,
      features_number])
)

validation_dataset = tf.data.Dataset.from_generator(
  generator = lambda: generate_batches(files=validation_files,
      batch_size = batch_size),
  output_types = (tf.float32, tf.float32),
  output_shapes = ([None, time_steps, features_number], [None,
      features_number])
)

test_dataset = tf.data.Dataset.from_generator(
  generator = lambda: generate_batches(files=test_files,
      batch_size = batch_size),
  output_types = (tf.float32, tf.float32),
  output_shapes = ([None, time_steps, features_number], [None,
      features_number])
)
```

Listing 11.1: Input preprocessing

Analyzing the "generate_batches" method, we note that it loops infinitely.

At each cycle, it gets a filename from the list and opens the corresponding file, allocating it inside the RAM. Getting a line at a time, executes the "manageLine" method passing it as argument. Referring to the "RecordWriter.cs" script defined in section 7.3, we can see that the vectors collected from the game state are parsed in string format through the "toString" method, that inserts round brackets around physical quantities' subvectors. For example, supposing to have a C-Sharp 2D vector Vector2(3f, 4f).ToString(), it is translated into "(3, 4)". In addition, knowing that these values are written on a CSV file, some escape characters are added because necessary. The "manageLine" method removes all these symbols, returning a list of floats.

Once a file has been parsed, obtaining a **list of float lists** called "parsedFile", input matrices and labels are created. The "manageInput" method takes as argument the "parsedFile" list and the *timesteps* integer, where the latter is the number of rows of a input matrix for the neural network, the past vectors necessary to make a prediction. Iterating over "parsedFile", this function takes *timesteps* following lists and creates an input array through the NumPy library command "array". This conversion is necessary for Keras execution. In addition, the method takes the timesteps+1th vector and set it as the output label. **The meaning of this operation is that the network has to be able to predict the *timesteps+1th* vector given the past *timesteps* vectors**. The "manageInput" method will return two lists, with the former containing arrays and the latter containing vectors, structured in a way that the i-th array is associated with the i-th vector, the expected inference result.

The script, receiving in input the dataset folder and the number of training, validation and test files, creates three lists named "train_files", "validation_files" and "test_files" respectively. This is necessary in order to count the total number of records (rows) available for each set and how many steps must be done in each epoch in order to analyze all the dataset. Keras doesn't allow me to define the epoch dimension a priori because the script doesn't load all the dataset, thus it doesn't have knowledge about the total size. So, given the total dataset and batches dimensions, I define the epochs size for each set in the integer variables "steps_per_epoch_train", "steps_per_epoch_val", "steps_per_epoch_test". Each of them is defined through the following computation:

$$StepsPerEpoch = \frac{TotalNumberOfLines}{BatchSize} \tag{11.4}$$

where $TotalNumberOfLines$ is the sum of the rows of the files composing each subdataset. Finally, the three preprocessed datasets are generated through the "from_generator" method beloning to the "tf.data.Dataset" Python class. This function takes as argument the lambda function "generate_batches"

explained before, the output types and the input and output shapes.

11.1.2 Neural Network definition

Here, the Keras network definition code listing is presented.

```
1  log_dir = "logs/fit/" + model_name # directory path contaning
       training fit logs for Tensorboard execution
2  tensorboard_callback = keras.callbacks.TensorBoard(log_dir=log_dir
       , histogram_freq=1) # training logs callback definition
3
4  optimizer = keras.optimizers.Adam(learning_rate=learning_rate) #
       optimizer definition
5
6  # Network definition
7  model = Sequential()
8  model.add(LSTM(nodes_number, activation="tanh", return_sequences =
        True, input_shape=(time_steps, features_number))
9  model.add(Dropout(dropout_value))
10 model.add(LSTM(nodes_number, activation="tanh", return_sequences =
        True))
11 model.add(Dropout(dropout_value))
12 model.add(LSTM(nodes_number, activation="tanh"))
13 model.add(Dropout(dropout_value))
14 model.add(Dense(features_number))
15 model.compile(optimizer=optimizer, loss="mse", metrics=[tf.keras.
       metrics.MeanAbsoluteError(), "accuracy"])
16
17 model. summary() # network summary printing
18
19 # early stopping setup
20 es = tf.keras.callbacks.EarlyStopping(monitor="loss", mode="min",
       verbose = 1, patience = 10)
```

Listing 11.2: Model definition and compilation

In this phase, the neural network model is defined and compiled through the Keras methods. It is a "Sequential" model (the most classical one, meaning that all the layers are put in a serial manner, where each one takes as input the output of the past layer). Once created this Python object, a series of alternating "LSTM" and "Dropout" layers are added. Analyzing a LSTM layer, it takes as argument the number of nodes, the activation function, the input shape and the "return_sequence" flag, which is set to *true* to make the layer return its state at each step. Analyzing the first LSTM layer (see line 8), we can see it takes in input an array of dimension $timesteps \times featuresNumber$, where *featuresNumber* is the number of dimensions belonging to a input or

output vector. The other type of layer, "Dropout", has the objective to execute a **dropout** operation, meaning that some nodes will be periodically turned off to avoid network overfitting. The dropout value as argument means the turn off probability.

In the end, the built network is compiled through the "compile" method, which takes as argument the optimizer, the loss function and the support metrics. The optimizer used is ADAM, the loss is the MSE and the support metrics are MAE and "Accuracy".

In addition, a special Keras functionality is allocated, called "Early Stopping". It is a function that analyzes the "monitor" target (in this case the loss function) and stops the training if this doesn't change enough from one epoch to the following.

11.1.3 Neural Network training

The listing 11.3 presented below shows the "fit" method with its associated arguments.

```
model.fit(
  train_dataset,
  steps_per_epoch = steps_per_epoch_train,
  batch_size=batch_size,
  epochs=epochs_number,
  verbose=1,
  validation_data = validation_dataset,
  validation_steps = steps_per_epoch_val,
  callbacks=[tensorboard_callback],
)

# Saving model in default format
model.save(models_path + "_" + model_name)

# Saving model in a ONNX conversion compatible format
tf.saved_model.save(model, onnx_folder_path+
    current_saved_models_path+"/"+model_name+"/"+"saved_tf")
```

Listing 11.3: Model training

Here we can see the real Keras advantage: in addition to the fast network construction, the training algorithm is all included and represented by the "fit" method, which takes the following arguments:

1. training dataset: set of couples (input array, output label)
2. steps_per_epoch: number of steps to be executed in each epoch

3. batch_size: dimension of a single batch, a set of data
4. epoch: number of epochs the training algorithm must execute
5. verbose: the type of strings output we want to get on terminal during training
6. validation_data: the validation dataset built before. Note that, passing this argument, the "validate" method is automatically invoked at the end of each training epoch
7. validation_steps: number of steps to be performed in a validation epoch
8. callbacks: set of auxiliar functions to be performed during training. The only one I use writes the training fit log for Tensorboard execution

Once invoked this method, it starts training the *model* network and prints strings on terminal showing progress. At the end of each training epoch, the validation set is passed in input to the model and the same metrics are computed, enabling us to evaluate the network with unknown data.

11.1.4 Neural Network evaluation

Finally, the last stage is perfomed through the "evaluate" method presented in the listing below. This function is invoked when the network is completely trained, and tests it passing in input to the network a set of completely unknown data. This kind of operation has the objective to understand how the net behaves "in real life". At the end of evaluation, metrics' results are printed.

```
1 ev_results = model.evaluate(test_dataset, batch_size=batch_size)
2 print("Mean Squared error (loss), Mean Absolute Error, Mean
      Squared Error, Accuracy: "+str(ev_results))
```

Listing 11.4: Model evaluation

11.2 Neural Networks tests

Different kinds of architectures were tested during the SPG development. I've executed many trainings with the objective to find the most suitable setup for game state prediction. In the following sub-sections, I will talk about networks all based on the most functional structure I created. As we will note, in fact, the described neural networks are very similar, differing only on the dataset

size and the timesteps in input. All the improvements I made are based on accuracy maximization, instead of loss minimization: in fact, a network with minimum loss and 50% of accuracy will be worse than another one with higher loss but with 90% of accuracy. This point of view is also caused by the fact that, in order to achieve loss minimization, an increasing number of nodes is required. This brings to a way higher computational cost (prohibitive also for a GPU like the NVIDIA Tesla T4 (see 14.2)), and space cost (for example, a network with 2 layers with 1000 nodes each is 50MB large).

Note that all the explanation I will make refers only to the Physics Engine, which is the most complex system (given 12 features to predict) among the three and so the most valuable.

11.2.1 NN4

NN4 is the first important network with the new architecture. Its Keras summary is the following:

```
1  _________________________________________________________________
2  Layer (type)                 Output Shape              Param #
3  =================================================================
4  lstm (LSTM)                  (None, 20, 50)            12600
5
6  dropout (Dropout)            (None, 20, 50)            0
7
8  lstm_1 (LSTM)                (None, 20, 50)            20200
9
10 dropout_1 (Dropout)          (None, 20, 50)            0
11
12 lstm_2 (LSTM)                (None, 50)                20200
13
14 dropout_2 (Dropout)          (None, 50)                0
15
16 dense (Dense)                (None, 12)                612
17
18 =================================================================
19 Total params: 53,612
20 Trainable params: 53,612
21 Non-trainable params: 0
22 _________________________________________________________________
```

Listing 11.5: NN4 Keras summary

We can see that it is formed by 3 LSTM layers, each one of them containing 50 nodes, alternated with 3 Dropout layers. The last component of the network is a Dense layer that collects results and returns a vector of *featuresNumber* size (in case of the Physics Engine, it has got size 12). NN4 takes in input a

3D tensor of dimensions $1 \times timesteps \times featuresNumber$. This network has been trained with 2,5 millions of records, equivalent to about 2400 Pong rallies and it takes 20 timesteps in input.

NN4's training was executed passing the following values as argument:

Parameter	Value
Input timesteps	20
Features	12
Batch Size	128
Number of epochs	20
Dropout value	0.2
Nodes per layer	50

Table 11.1: NN4 training parameters

Through the command *tensorboard* we are able to see inside graphs the movement of MSE, MAE and Accuracy in the three phases of the Python script I described in listing 11.3.

In picture 11.1 below, we can analyze the loss function variation in NN4 during its training

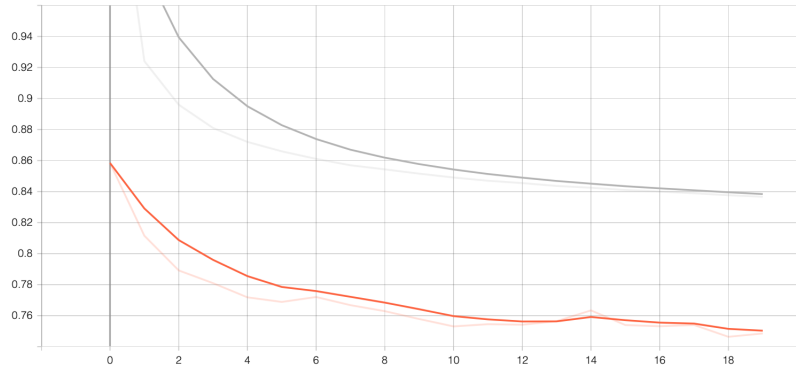

Figure 11.1: NN4 loss plot. Orange Line: validation loss. Grey Line: training loss

where the orange line represents the **validation** loss, while the grey one represents the **training** one. On the X axis we have the epoch number, while on the Y axis the loss value. As we can see from picture 11.1, the grey line starts from a higher point than the orange one, because in the first epochs the NN doesn't know anything about the problem. Already from the first epoch we can see the validation loss assuming lower values, meaning that the network

has got a good behaviour with data it has never seen and it has already learned something in the first training iteration.

In picture 11.2, we can observe the network behaviour through the accuracy metric. Given the same color coding as in the previous plot (11.1), we can observe that, also in this case, the network has got a good behaviour with unknown data since from the start, reaching the 99,8% of accuracy in the end, better than during training.

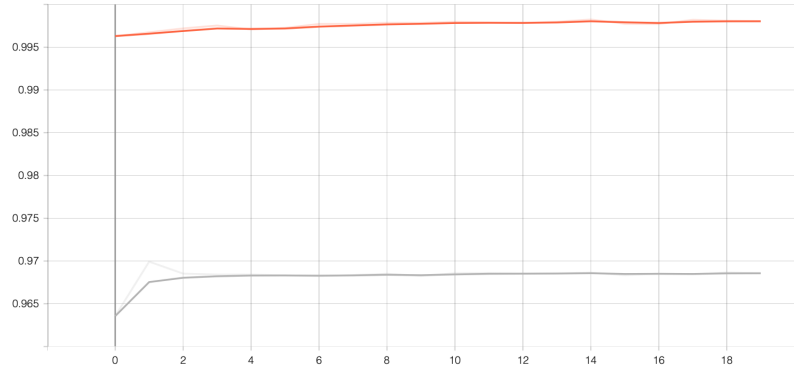

Figure 11.2: NN4 accuracy plot. Orange Line: validation accuracy. Grey Line: training accuracy

Concluding the NN4 analysis, we observe the Mean Absolute Value metric in figure 11.3. There is a huge difference between the training movement (grey line), reaching 0.41 as minimum value, and the validation one (orange line), reaching 0.31. This means that the network can infer new states from unknown data with this margin of error.

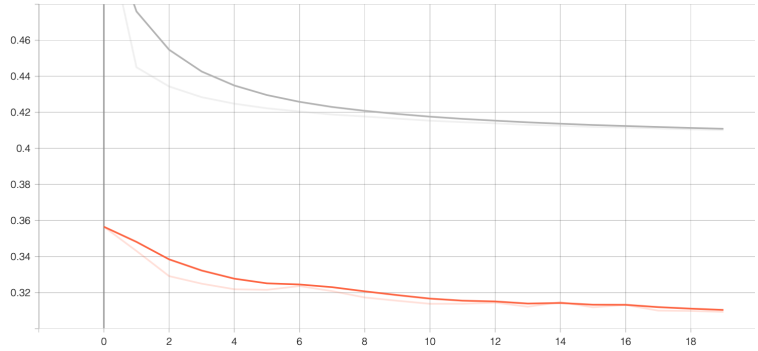

Figure 11.3: NN4 Mean Absolute Error plot. Orange Line: validation mean absolute error. Grey Line: training mean absolute error

11.2.2 NN6

Another important neural network is NN6, a net that is equal to NN4, but that needs to receive 40 timesteps in input, thus the input tensor will have size $1 \times 40 \times featuresNumber$. I tested this network in order to understand if NN4's behaviour could be overpassed, and so to get better predictions, by changing the number of states got from the past. NN6 summary is displayed in the listing below:

```
1  _________________________________________________________________
2  Layer (type)                 Output Shape              Param #
3  =================================================================
4  lstm (LSTM)                  (None, 40, 50)            12600
5
6  dropout (Dropout)            (None, 40, 50)            0
7
8  lstm_1 (LSTM)                (None, 40, 50)            20200
9
10 dropout_1 (Dropout)          (None, 40, 50)            0
11
12 lstm_2 (LSTM)                (None, 50)                20200
13
14 dropout_2 (Dropout)          (None, 50)                0
15
16 dense (Dense)                (None, 12)                612
17
18 =================================================================
19 Total params: 53,612
20 Trainable params: 53,612
21 Non-trainable params: 0
22 _________________________________________________________________
```

Listing 11.6: NN6 Keras summary

As we can note, the layers arrangement is not changed. The only difference is input timesteps number.

Its tensorboard results are displayed in the figures below, where the light blue line represents training, while the violet one represents validation. We can still see the differences between validation and training results, with the former one reaching better values than the latter.

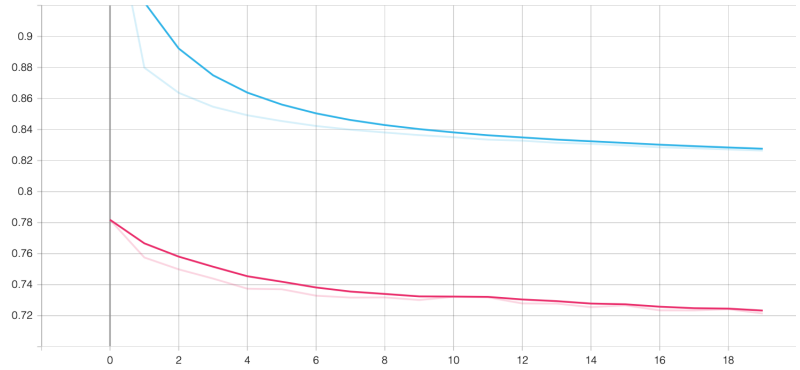

Figure 11.4: NN6 loss plot. Violet Line: validation loss. Lightblue Line: training loss

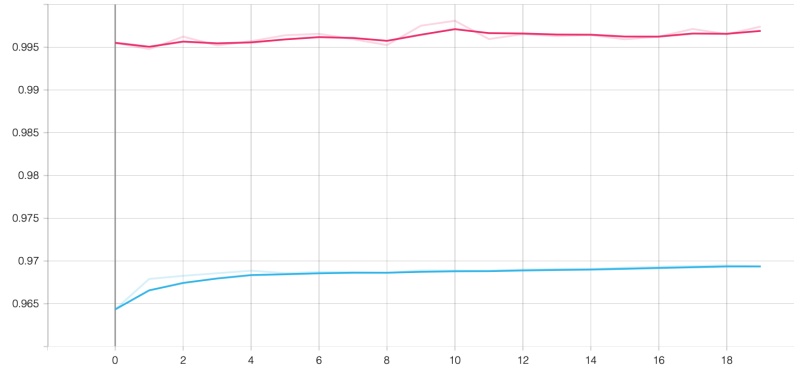

Figure 11.5: NN6 Accuracy plot. Violet Line: validation accuracy. Lightblue Line: training accuracy

As we can note, its loss is slightly better than the NN4's one. In NN6 we reach 0.72 as minimum MSE value, while in NN4 it is about 0.75. Referring to the accuracy, instead, we can see that it is better in NN4 than in NN6. In the former, looking at plot 11.2, we note that it gets about the 99,8% of correctness, while in diagram 11.5, it is equal to 99,6%.

Analyzing Inference Engine's time of execution, we can notice that NN6 takes more time to be executed (around 0.09 seconds) respect to NN4 (around 0.086 second). This leads us to an important result: **Barracuda, using LSTM, takes more time to execute inference with a network which uses more timesteps than a network that requires less of them**. Summing up all the considerations, we can achieve a smaller loss receiving more input states, but at the same time reaching worst accuracy. In addition, a network with a bigger

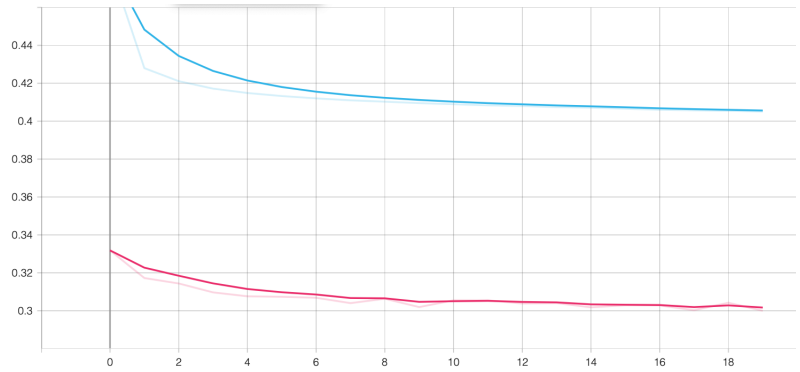

Figure 11.6: NN6 Mean Absolute Error plot. Violet Line: validation mean absolute error. Lightblue Line: training mean absolute error

input tensor is more computationally expensive, so not suitable for SPG implementation. Given this consideration, the different time needed for execution will lead to a different set of input data, because sampled at lower rate, and this will take us to a wrong behavior by the network. If we want to correctly execute NN6, we should generate from PongOffline a new dataset containing records with a different time distance, and then retrain the network.

11.2.3 NN9

This network has the same structure as NN4 (so we can refer to summary listing 11.5 to understand its architecture), but it is trained with a way higher number of records. NN9 was created through 20 millions of records, which corresponds to about 16500 rallies. As we can see from the following graphs, this network reaches, during validation, loss values around 0.735 and accuracy values around 99.6%. We note that this values *in accuracy* are worse than the ones collected with NN4, but as we will see in chapter 12, regarding *ball position* NN9 behaves slightly better than NN4 inside the Pong Online game: it can predict very well the ball's correct trajectory, and also basing its predictions on its past outputs it can create complex and correct trajectories for a few seconds.

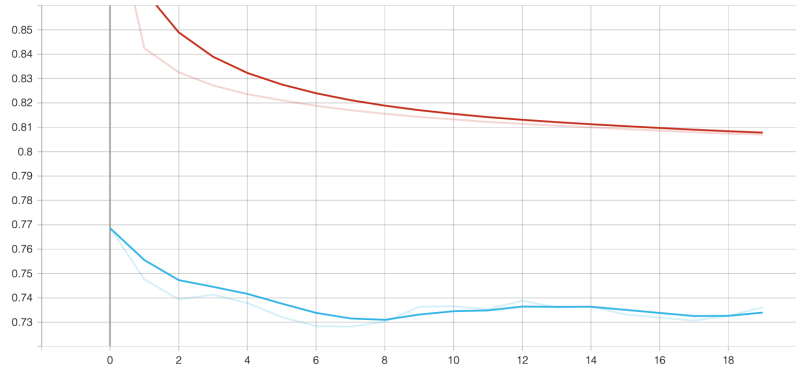

Figure 11.7: NN9 Loss function plot. Lightblue Line: validation loss. Red Line: training loss

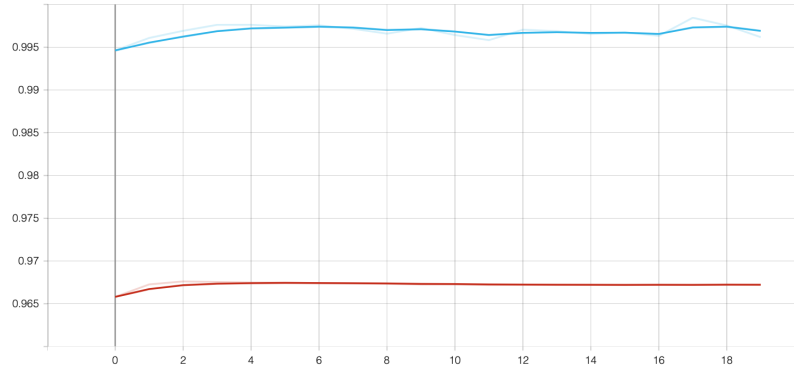

Figure 11.8: NN9 Accuracy plot. Lightblue Line: validation accuracy. Red Line: training accuracy

11.2.4 NN10

Also this network has the same structure as NN4, so again we can refer to Keras summary 11.5. This is the last net I created and it's the one trained with the biggest dataset. In fact, NN10 was trained through 51 millions of records, corresponding to 31000 rallies. We could imagine that, given the considerations made with NN9, we could achieve better performances than every other network, but it is not like this. NN10 creates bad predictions and doesn't behave very well inside Pong Online.

Looking at figures 11.9 and 11.10, we note that the reached validation values are worse than the ones collected with the other networks, both in loss and accuracy. The performances observed are not at the expected level, especially

given the computation effort in order to train this massive net. This brings us to the result that a bigger training dataset not necessarily brings to better neural performances, but maybe it's more about hyperparameters tuning.

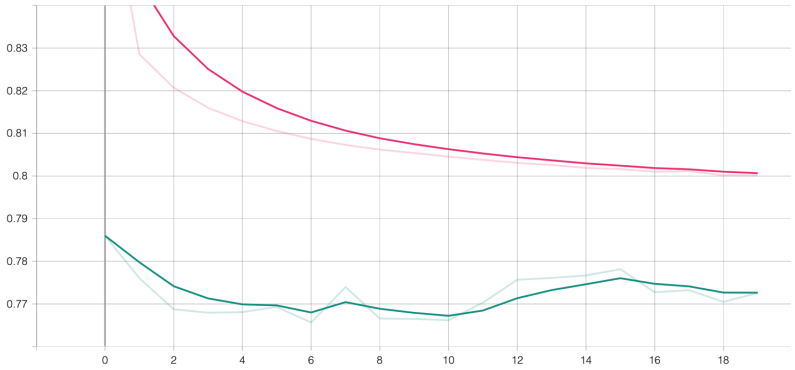

Figure 11.9: NN10 Loss function. Green Line: validation loss. Red Line: training loss

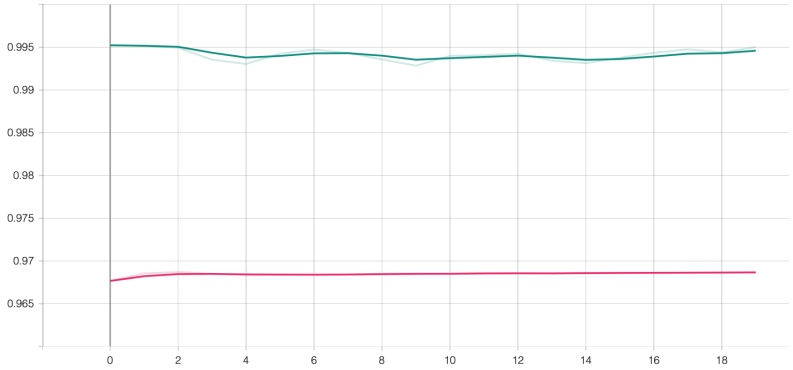

Figure 11.10: NN10 Accuracy. Green Line: validation accuracy. Red Line: training accuracy

11.2.5 Final results

Comparing together all the 4 neural networks presented in the previous sections, the resulting plots are the following:

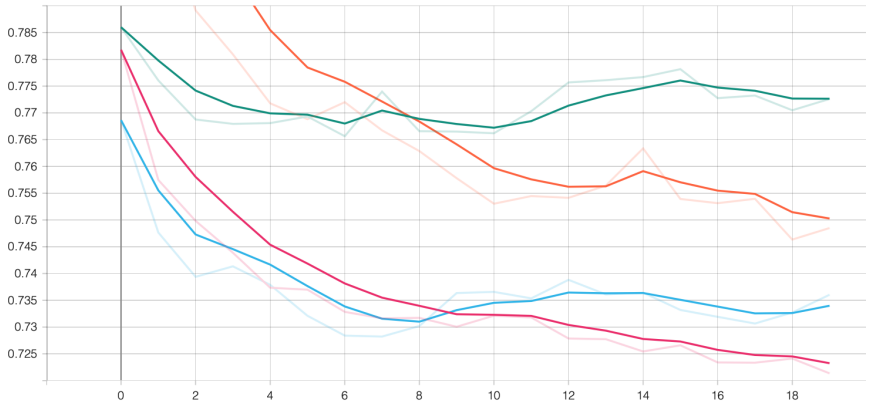

Figure 11.11: Four Neural Networks losses. Green line: NN10 validation loss. Orange line: NN4 validation loss. Lightblue line: NN9 validation loss. Violet line: NN6 validation loss

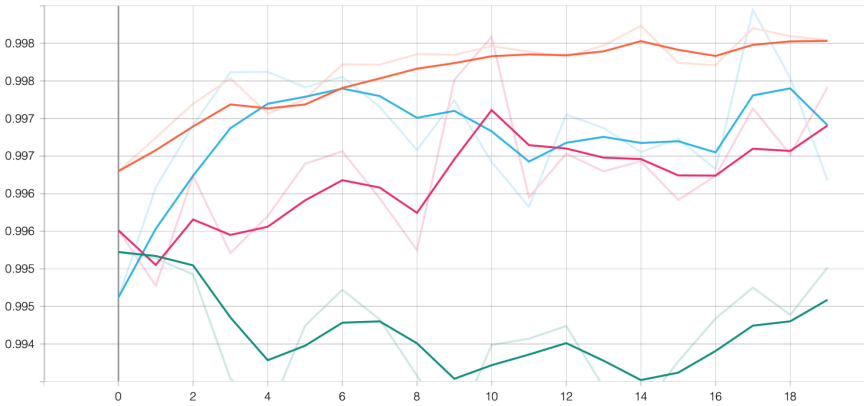

Figure 11.12: Four Neural Networks accuracies. Green line: NN10 validation accuracy. Orange line: NN4 validation accuracy. Lightblue line: NN9 validation accuracy. Violet line: NN6 validation accuracy

As we can see from image 11.12, the accuracy reached by NN4 (orange line) during the validation process is the best ever. NN6 (violet line) and NN9 (lightblue line) reach, at the 20th epoch, comparable results, also if they draw two different curves: NN6 approximately draws a growing line, while NN9 draws a more descending one. This means that a bigger number of training records brings to greater uncertainty inside the predictor, maybe because it knows more different possible rallies and situations. NN10, instead, reaches the worst accuracy.

Analyzing image 11.11, we note that the network that reached the minimum loss is NN6, but it isn't the best also in terms of accuracy. This result states that a network that uses more timesteps to make predictions, thus looking further in the past, can achieve better results in term of MSE, but it doesn't necessary bring to improved performances in real life applications. So, **a network with minimum loss is not necessary a good network**.

The network that reached again the worst result is NN10, which is also the one trained with the bigger dataset. This bad result could be caused by the fact that the possible situations it has to learn are a lot, and so the variability is too high.

Referring to NN9 and NN4, they both obtain loss values in between, while NN4 has got a better accuracy also if trained with a 10 times smaller dataset. As we'll also understand in chapter 12, the neural network creation is based on hyperparameters and dataset tuning, trying to find the best implementation, which hopefully works well also in real life applications, and NN4 and NN9 are the ones that have the best behaviour in Pong Online.

Chapter 12

Network latency simulation

The last part of my Master's Degree project was the introduction of a lag simulator inside Pong Online, with the aim to see MPAI-SPG in action and study its behavior. This is possible because, once defined the real SPG implementation inside the Unity project (see chapter 10), the game became a playground for tests and experiments.

12.1 Client lag simulation

The neural networks study in Pong Online started with the introduction of the lag simulator. Fortunately, the Photon Library contains some useful tools that enable programmers to test the network support in their applications. This tool is called "Photon Lag Simulation GUI" and it's a component that can be attached to a game object present in the game scene. The object we have to choose has to transmit its information from one application to another, so I decided to attach the simulator to the "Paddle" prefab inside the Pong Online **Client**. Referring to the Photon documentation [9], we can see that this component offers various functionalities:

1. Lag: Slider that defines a measure (in milliseconds) of the delay that is added to all the outgoing and incoming messages of the application
2. Jit: Slider that has got the same functionality of "Lag", but the delay value is chosen randomly up to the maximum value defined by the slider
3. Loss: Slider that sets the drop percentage of the incoming and outgoing packages.

As we can see, with the Photon Lag Simulator GUI it is possible to simulate both packet latency and loss, so it is perfectly suitable for our use case. This component is displayed inside the scene as a GUI box on the left side of the scene containing the three sliders and a button "Simulate" that enables the "malfunctions". In addition, it shows the current Roundtrip Time (RTT), which is a measure of the time in milliseconds until a message is aknowledged by the server.

12.2 Server Lag computation

Once given to the client the tools to execute network disruptions, it was necessary to permit the server to recognize a network lag. This operation is perfomed inside the "PlayerControl.cs" script, specifically inside the "onPhotonSerializeView" method, whose signature is the following:

```
1 public void OnPhotonSerializeView(PhotonStream stream,
      PhotonMessageInfo info) { }
```

Listing 12.1: "onPhontonSerializeView" method signature

As we can see, the method takes two arguments:

1. stream: the data stream received from the client
2. info: the collection of support informations contained inside the messages sent from the client. Photon, in fact, doesn't send only data about the game objects, but also support informations about the packets transmission

Formally, the **lag** is defined as **the time occured between the message shipping and reception**. So, known this definition, we can calculate it through the following formula:

$$lag = receptionTime - shippingTime \tag{12.1}$$

that can be translated in C-Sharp as:

```
1 float lag = Mathf.Abs((float)(PhotonNetwork.Time-info.timestamp));
```

Listing 12.2: Lag computation C# code

where *PhotonNetwork.Time* represents the Photon instant time when the "Time" field is accessed, while *info.timestamp* represents the instant when the message was shipped from the client. The difference is casted to float type and

then the absolute value is returned. Photon activates for each room a counter that keeps track about a relative time flowing inside the game. Note that there's no possibility to get informations about a "universal clock" executing inside the Photon Framework, common for client and server applications, because it is not supported inside this free license version of the library. Other paid Photon libraries offer this tool.

Setting up the maximum lag to 0,2 seconds, which is the universally accepted value inside the game industry, when the calculated lag is bigger than this maximum value, MPAI-SPG is **activated**, meaning that **it will start forecasting using its past output predictions as input**.

In order to avoid conflicts, an SPG activation mechanism based on the player's name was created. When the server detects a lag, while activating the predictor it keeps track of the client's name causing the problem, bringing us to the result that SPG can be disabled only by the same player who enabled it. Through a method called "activatePrediction", which belongs to the "GameManager" class and that takes as argument the player's name, firstly the nickname is compared with the one saved inside a variable and, if they're equals, SPG is enabled. The "deactivatePrediction" method works in reverse. This mechanism is necessary because the Game Manager, which, as I explained, is the component that activates SPG, constantly receives lag informations from both clients. Supposing just one of them is having network problems, in a first time SPG would be activated, but immediately after it would be deactivated because the other client would send information about good network conditions. Using this system, SPG remains enabled until the player having problems reports that the latency or packet loss ended.

At this point, the lag prototype is completed: we have two applications (client and server) able to communicate and share informations about Pong. In addition, the client can simulate network problems while the server observes them, consequently enabling the MPAI-SPG system and displaying predicted informations about the game state inside the server scene.

12.3 Qualitative tests

Once completed the lag simulator integration, it is time to perform some tests about data collected from Server and Client applications. This kind of experiments explores in a qualitative manner the predictive behaviour of the Digital Twin. The tests I'm gonna perform firstly collect information about Server's and Client's game states, and then compare them calculating errors and plotting diagrams. In particular, I wanna study the ball's behavior and see what are

the differences between its transform's position and velocity taken respectively from the predictor's output and the Pong correct execution.

These tests plot different diagrams that show the transform's components evolution during some time interval. In images 12.3 and 12.6 I plotted 4 different cartesian diagrams. Each one of them refers, on the X axis, to the *PhotonNetwork.Time* value, while the Y axis is different for each plot, referring respectively to: position's x component, position's y component, velocity's x component, velocity's y component.

Position diagrams:

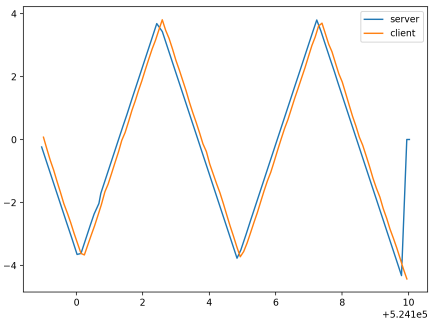

Figure 12.1: X position diagram

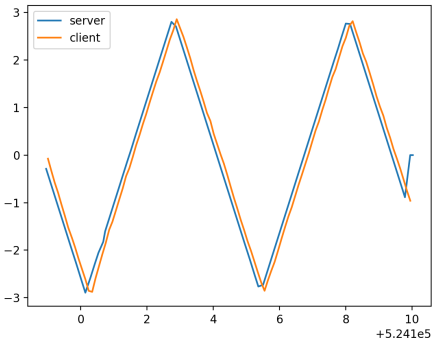

Figure 12.2: Y position diagram

Figure 12.3: Ideal Client-Server situation, position

Velocity diagrams:

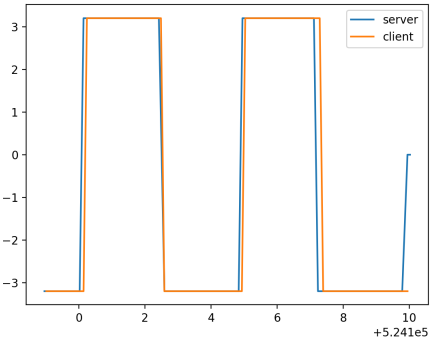

Figure 12.4: X velocity diagram

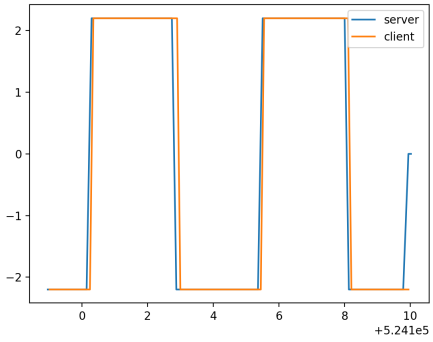

Figure 12.5: Y velocity diagram

Figure 12.6: Ideal Client-Server situation, velocity

These diagrams have been plotted with the MPAI-SPG not enabled, and thus we can note that the orange and blue curves, in each plot, are practically identical. They were generated from two CSV files created from client and server applications respectively. An extract of these two logs is shown in tables 12.1 and 12.2.

PhotonNetwork.Time	Ball XPos	Ball YPos	Ball XVel	Ball YVel
524099.011	0.07300802	-0.07502481	-3.2	-2.2
524099.111	-0.292412	-0.3262511	-3.2	-2.2
524099.214	-0.6558896	-0.5761419	-3.2	-2.2
524099.318	-0.9730166	-0.7941667	-3.2	-2.2
524099.417	-1.32386	-1.035372	-3.2	-2.2

Table 12.1: Client Test Log extract

PhotonNetwork.Time	Ball XPos	Ball YPos	Ball XVel	Ball YVel
524099.078	-0.626489	-0.5559291	-3.2	-2.2
524099.194	-0.9977667	-0.8111826	-3.2	-2.2
524099.328	-1.426083	-1.10565	-3.2	-2.2
524099.456	-1.834453	-1.386405	-3.2	-2.2
524099.557	-2.157968	-1.608821	-3.2	-2.2

Table 12.2: Server Test Log extract

Looking at the two tables, we can note that the *PhotonNetwork.Time* values contained into them are different: even if the integer part is identical, the decimal part is not the same and increases in dissimilar manners. This means that the two collections are taken at different time instants. As I already explained in section 12.2, *PhotonNetwork.Time* is a property feeded by the Photon framework that refers to room's time. For these plots, this value is referring to the instant when the transform's values were recorded inside the CSV files. It is important to note that these values aren't the same between client and server for two main reasons:

1. it is a computation made in milliseconds, so the precision required is too high to obtain a perfect corrispondence between the time recorded from the two applications

2. the logs are generated through two different scripts performed into an asychronous way: inside the client application, I use the "BallControls.cs"

script, while in the server I use a Inference Engine, in order to be able to get SPG inference results. This means that the two sets of values are recorded in different time instants

This time difference is visible in plots 12.3 and 12.6, where the two curves are not perfectly aligned on the X axis because interpolated from different time instants.

Once created the files, the 4 physical quantities are plotted through the *MatPlotLib.pyplot* library imported into a Python script. This library enables us to draw different curves inside the same graph. So, having a series of (time, value) couples, this framework can automatically interpolate a line passing between these points in a 2D cartesian diagram. For example, supposing to plot the position's x component, I parsed the two log files obtaining a list of tuples of the type (*timestamp*, *position.x value*) and then I inserted them inside the pyplot figure obtaining a diagram like 12.1.

Ideally, collecting SPG results, we would like to achieve situations like the ones presented in figures 12.3 and 12.6, where the orange and the blue lines draw identical curves, shifted on the X axis. This could be possible if SPG was able to make incredibly correct predictions, emulating the Pong behavior.

12.4 Error computation

Once obtained the log files, I used them also to calculate three different metrics, in order to understand what's the mathematical difference between the curves contained in the plots that will be shown in section 12.5. I calculated the already presented Mean-Squared-Error, Mean-Absolute-Error and the Root Mean-Squared Error.

Knowing how the log files are composed, in order to calculate this metrics I interpolated the client's values on the server's *PhotonNetwork.Time* timestamps through the NumPy library method "interp". This way, I obtained a floats list of the same length as the server's one containing values referring to the same time instants. Once executed the interpolations, I calculated the metrics. The results are presented in the following section.

12.5 Neural Networks results

In the following figures are displayed the results obtained making inference inside Pong Online through the neural networks presented in chapter 11.

The diagrams have been generated through logs with the same structure shown in tables 12.1 and 12.2, but, concerning server's files, they contain SPG forecast results. The orange lines (client's position and velocity x and y components) show the "correct" Pong position, the ideal situation we would like to achieve with the SPG predictor.

The tests made have been performed executing both server and client applications on NNTA machine (14.2), thus not considering more complex network situations.

12.5.1 NN4

Position diagrams:

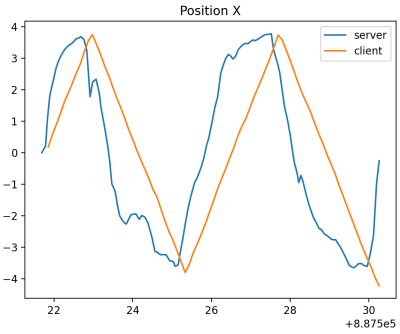

Figure 12.7: X position diagram

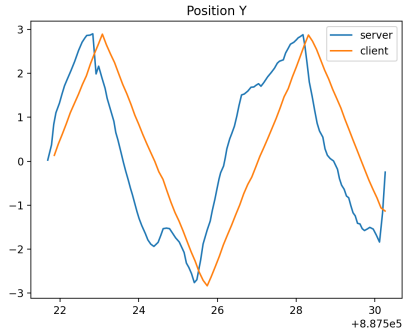

Figure 12.8: Y position diagram

Figure 12.9: NN4 resulting diagrams, ball position

Velocity diagrams:

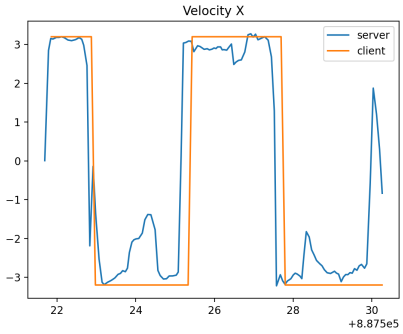

Figure 12.10: X velocity diagram

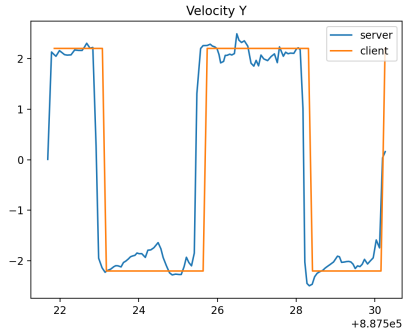

Figure 12.11: Y velocity diagram

Figure 12.12: NN4 resulting diagrams, ball velocity

As we can see from figures 12.9 and 12.12, the server line (blue one) tries to achieve the ideal situation (orange line), even if with a high degree of error.

The resulting metrics are the following:

Physical Quantity	**MSE**	**RMSE**	**MAE**
Position X	4.6729	2.1617	1.9551
Position Y	8.6911	2.9480	2.6927
Velocity X	3.7908	1.9470	1.3350
Velocity Y	1.6866	1.2987	1.0570

Table 12.3: NN4 simulation resulting metrics

12.5.2 NN6

Position diagrams:

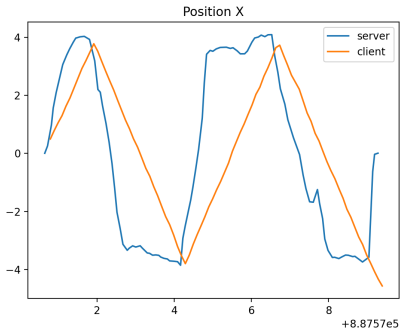

Figure 12.13: X position diagram

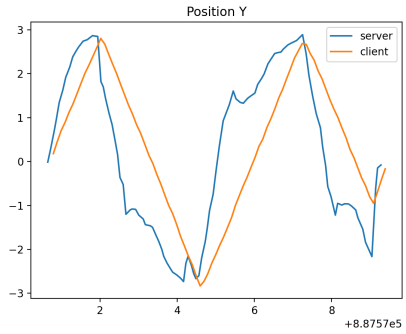

Figure 12.14: Y position diagram

Figure 12.15: NN6 resulting diagrams, ball position

Velocity diagrams:

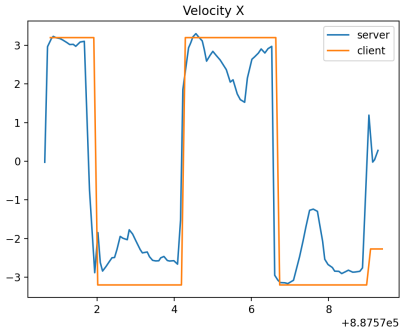

Figure 12.16: X velocity diagram

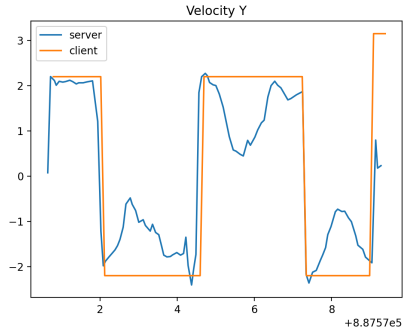

Figure 12.17: Y velocity diagram

Figure 12.18: NN6 resulting diagrams, ball velocity

Analyzing figures 12.15 and 12.18, we note that they show worse results than the ones displayed in NN4's diagrams. This means that the NN4's prediction is better than the NN6's one. We arrived at this conclusion also in chapter 11 comparing loss and accuracy plots. We can conclude that a network that reaches a better accuracy instead of a better loss during training and evaluation will get better results also in real life applications.

This concept can be inferred also looking at the following metrics:

Physical Quantity	MSE	RMSE	MAE
Position X	6.5696	2.5631	2.2273
Position Y	10.4737	3.2363	2.8833
Velocity X	4.3364	2.0824	1.4037
Velocity Y	2.4380	1.5614	1.3552

Table 12.4: NN6 simulation resulting metrics

where we can note higher errors than the ones collected in table 12.3.

12.5.3 NN9

Position diagrams:

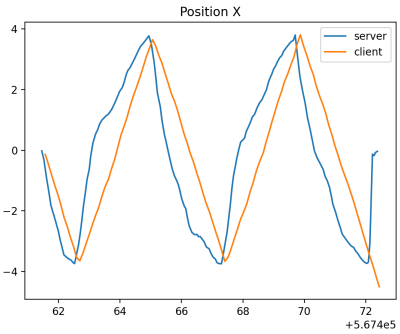

Figure 12.19: X position diagram

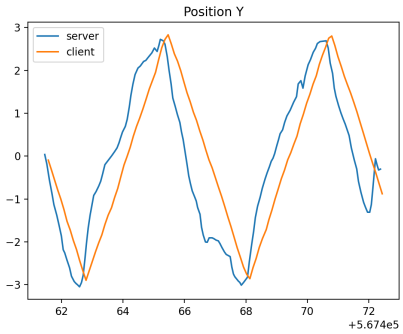

Figure 12.20: Y position diagram

Figure 12.21: NN9 resulting diagrams, ball position

Velocity diagrams:

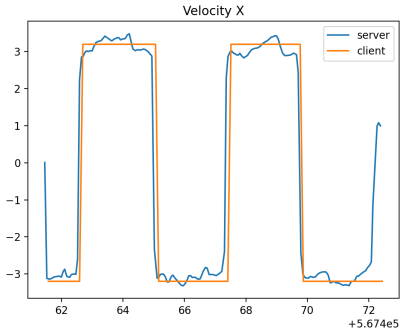

Figure 12.22: X velocity diagram

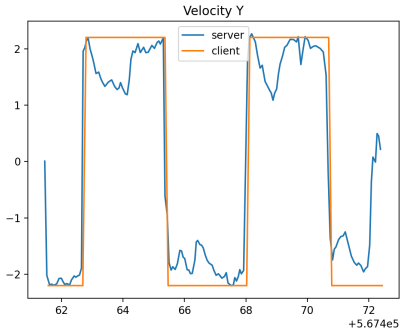

Figure 12.23: Y velocity diagram

Figure 12.24: NN9 resulting diagrams, ball velocity

NN9 is able to create a X position curve that is closer to the ideal one than the analogous one displayed inside the NN4 diagram 12.7. Also the Y position diagram is, in some intervals, better than NN4 (see plot 12.8), but it reaches deeper points than the orange valleys.

The metrics resulting form NN9 execution are the following:

Physical Quantity	**MSE**	**RMSE**	**MAE**
Position X	3.2654	1.8070	1.6356
Position Y	7.6057	2.7578	2.5060
Velocity X	5.5167	2.3487	1.7562
Velocity Y	1.8532	1.3613	1.1371

Table 12.5: NN9 simulation resulting metrics

Comparing tables 12.5 and 12.3, we note that, concerning transform's position, NN9 reaches better results tha NN4, while referring to velocity, the latter returns better results than the former, also if, graphically talking, NN9's velocity plots seem to be closer to the ideal line. Thus, we could say that these two neural networks behave in a similar way in real life, and they are both better then NN6.

12.5.4 NN10

Position diagrams:

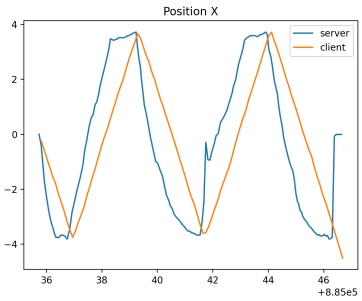

Figure 12.25: X position diagram

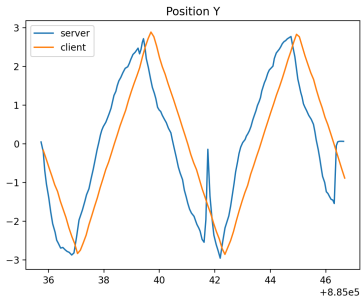

Figure 12.26: Y position diagram

Figure 12.27: NN10 resulting diagrams, ball position

Velocity diagrams:

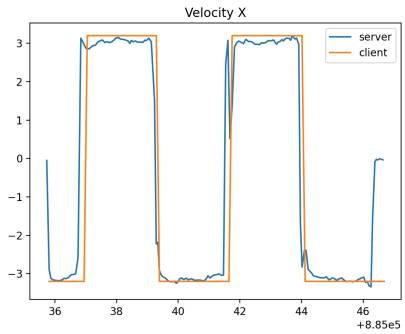

Figure 12.28: X velocity diagram

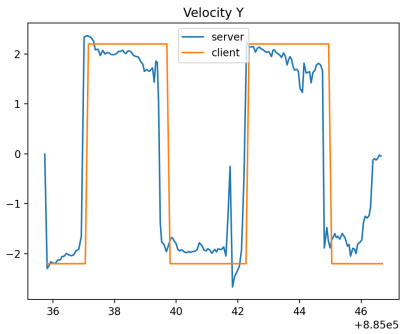

Figure 12.29: Y velocity diagram

Figure 12.30: NN10 resulting diagrams, ball velocity

Finally, here are displayed the results obtained for NN10, a neural network architecturally identical to NN9, but trained with more simulation data. Referring to figure 12.28, the x velocity, the server line is very close to the orange one, also reaching better results than NN9 if we analyze metrics contained inside table 12.6. Looking at the other 3 physical quantities, instead, the collected results are, on the contrary, worse in NN10 than in NN9, reaffirming the resulting concepts presented in chapter 11.

The resulting metrics are the following:

Physical Quantity	**MSE**	**RMSE**	**MAE**
Position X	4.7701	2.1840	2.0072
Position Y	9.8411	3.1370	2.8373
Velocity X	4.4681	2.1138	1.4941
Velocity Y	2.3392	1.5294	1.3078

Table 12.6: NN10 simulation resulting metrics

In conclusion, introducing the 4 neural networks inside a real life environment, we confirmed the results obtained from their study during training and evaluation: NN4 and NN9 are the ones that behave in the best way, each one of them making better predictions regarding to different physical quantities. In addition, we can also say that a greater number of timesteps in input or a larger training dataset doesn't necessary lead us to finer inference results.

Chapter 13

Conclusion and future works

Inside this Master's Degree project we talked about the MPAI-SPG theorical and practical definition, applying it to the Pong Online use case. We talked about what is the objective of this research, the tools needed and the initial Digital Twin Architecture. We explained how the classical Pong game works and how automatic players can be created in order to easily create a dataset for neural networks training. We introduced my adjustments to the architecture and its real implementation inside the Online version of the Pong project in Unity. In the end, we introduced the neural networks definition, studying their behavior both with training data and inside the "real life" application. We found out a good architecture able to predict the game states for a little time interval and now the MPAI-SPG prototype is able to infer how the game should behave when a network problem occurs and eventually correct the game state (details not explained in this thesis). In the future, this work for the MPAI standard can be improved finding out new and more suitable predictive architectures and an anti-cheating system can be introduced in order to make the Pong rules respected. In the future, after the General Assembly approval, the SPG project will advance to the "Call for technologies" step, asking to the videogame industries to implement their own SPG prototype and study it in a commercial way.

Chapter 14

Machines

In this chapter are listed the computers used during this Master thesis research. Thanks to the Universtity of Turin's Computer Science Department it was possible to access to the computers inside their Computational Center, instantiating two Virtual Machines named "NNTA" and "LPA" respectively.

14.1 Macbook Pro

Macbook Pro 13" 2017 Edition with Processor Intel i7 dual-core, RAM 16 GB DDR3 and Intel Iris Plus Graphics 640 1536 MB.

14.2 NNTA

"Neural Network Training Asset" Virtual Machine with i7 processor, 32GB RAM and NVIDIA Tesla T4 GPU, 50Gb of primary memory and 1TB of storage memory, running Windows 10 2016 Server Edition. This machine was used to execute Neural Network trainings, through the Tesla T4, and Pong Online test runs.

14.3 LPA

"Logs Production Asset" Virtual Machine with i7 processor, 32GB RAM, 50Gb of primary memory and 1TB of storage memory, running Windows 10 2016 Server Edition. This machine was used to produce logs forming the training dataset, running 4 Pong Offline instances at the same time.

Chapter 15

Ringraziamenti

Ringrazio i professori Maurizio Lucenteforte, Davide Cavagnino, Marco Mazzaglia e Leonardo Chiariglione per avermi accompagnato prima durante il percorso magistrale e poi in questo lungo progetto, trasmettendomi la passione per l'Informatica che spero potrò conservare e coltivare nel futuro.

Grazie a papà per averci creduto dal giorno 0.
Grazie a mamma per essere la mia spalla.
Grazie a Carlo e Maria per essere un grande esempio da seguire.
Grazie al Bunkereeno Kollektiv e la Di Gran Carriera per esserci nei momenti felici e tristi.
Grazie alla Musica per essere una sempre fedele compagna di viaggio su cui poter contare.

E' nei momenti di incertezza che si vedono le verità di ognuno: le pareti si stringono, le menti si allargano.
Le nostre preoccupazioni diventano schermo di ciò che avremmo potuto e vorremmo essere.
Sta a noi decidere che cosa guardare.

"Detesta il vuoto dei rumori della realtà, ma col volume a stecca può sopravvivere"
(Subsonica, "Aurora Sogna", 1999)

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Bibliography

[1] *Authoritative Server definition*. URL: https://doc.photonengine.com/zh-CN/bolt/current/troubleshooting/authoritative-server-faq (visited on 01/22/2022).

[2] *Gambetta Lag Compensation*. URL: https://www.gabrielgambetta.com/client-server-game-architecture.html (visited on 02/08/2022).

[3] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[4] *ML-Agents Framework Official documentation*. URL: https://github.com/Unity-Technologies/ml-agents (visited on 02/16/2022).

[5] *ML-Agents Glossary*. URL: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Glossary.md.

[6] *ML-Agents Learning Algorithms*. URL: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md#training-methods-environment-agnostic.

[7] *ML-Agents Reinforcement Learning official documentation*. URL: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Background-Machine-Learning.md.

[8] *MPAI Official Website*. URL: https://mpai.community/ (visited on 01/10/2022).

[9] *Photon Lag Simulation GUI Official documentation*. URL: https://doc.photonengine.com/en-us/pun/v1/troubleshooting/photon-lag-simulation-gui (visited on 02/25/2022).

[10] *Photon Unity Framework official site*. URL: https://www.photonengine.com/pun.

[11] *tf2*. URL: https://www.guru99.com/what-is-tensorflow.html#:~:text=TensorFlow%20is%20an%20open%2Dsource,inference%20of%20deep%20neural%20networks. (visited on 01/22/2022).

[12] *Vanishing Gradient problem explanation*. URL: https://en.wikipedia.org/wiki/Vanishing_gradient_problem.