Escuela
Politécnica
Superior

Artificial Intelligence for Videogames with Deep Learning

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:
Adrián Francés Lillo
Tutor/es:
José García Rodríguez
Alberto García García

Universitat d'Alacant
Universidad de Alicante

Junio 2019

Artificial Intelligence for Videogames with Deep Learning

A research on the practical applications of Deep Learning for Game Development

Author
Adrián Francés Lillo

Supervisors
José García Rodríguez
Department of Computer Technology (DTIC)
Alberto García García
Department of Computer Technology (DTIC)

Grado en Ingeniería Multimedia

Escuela Politécnica Superior

Universitat d'Alacant
Universidad de Alicante

Alicante, June 4, 2019

Abstract

In this project, we propose different techniques for developing Artificial Intelligences for videogames using Deep Learning models, with a special focus on the usage of neural networks for the development phase of a videogame.

To reach this point, we started the creation of a Deep Learning development pipeline, capable of integrating neural networks inside of the Unity game engine initially developed in Python. For the Python part of the project, we used mostly Keras and Tensorflow for the neural network training, with a special focus on compatibility and ease of use.

As background for our work, we carried out an extensive research about the current approaches inside and outside of the game development industry. We studied known cases where videogames are being used in the deep learning industry as a tool for research and experimentation, with well known examples such as AlphaStar, capable of defeating professional players on the game Starcraft II, or OpenAI Five, who has recently defeated the world champions on the game Dota 2.

We have also analyzed the status of the development of Artificial Intelligence inside the videogames industry, and the usage of Deep Learning as a part of the development process of videogames. We went through famous examples of usage of Artificial Intelligence as part of game development, especially pointing out games relationed with Deep Learning in some way.

Moreover, we have analyzed the current problems and flaws that a Deep Learning driven development of a game's artificial intelligence may have, and suggested guidelines for either application on a production environment or further experimentation.

Finally, we integrated the neural networks created on an Android videogame called HardBall, developed by *From The Bench Games, S.L.*. We used the Artificial Intelligence created to emulate the behavior of a Non-Playable Character inside of the game, with the intention to substitute the previous Artificial Intelligence with a neural network capable of learning and playing the game.

Acknowledgments

As one of the most important stages on the life of this student comes to an end, I would like to take a moment to show my gratitude to everyone who has helped me reach this point.

In first place, it is hard to find words capable of expressing my gratitude towards my supervisors, José and Albert, who have guided, helped and inspired me through the whole year. Thank you for giving me this opportunity, this work wouldn't be the same without you.

In addition to this, I would like to thank the *From The Bench Games, S.L.* team for their collaboration in this project. Especially to Pedro Pérez, for his support during our collaboration.

I would like to thank all the people from the 3D Perception Lab for sharing this experience with me and helping me when I needed. Thanks to Sergiu, Fran, Pablo, John, Andrés, Plácido, Jordi, Carlos, Álvaro and Groot. This journey was much better accompanied by great people. Additionally, I would like to thank my friends from Multimedia Engineering, specially to Nacho, for sharing all this year with me, and my companions from *Wasted Horchata*: Alexei, Pablo, Luis and Laura; the time with you was the best. Also I would like to thank all my friends, who have endured not seeming me while I was chasing this dream, specially to my roleplaying group, who have lost their dungeon master many times this year because of it. Thank you Arturo, Daniel, Manu and Álvaro.

I would also like to express my gratitude to all the good teachers who have inspired me through my student years, either in the University or before. I was not always the best student, but you always were the best teachers.

Finally, I wish to thank all the game developers who, with their hard work and dedication, have inspired me to develop this project. You made a small children dream with the impossible, and this is the result.

Last but not least, I would like to thank my family, for raising me, supporting me, teaching me and loving me; and my girlfriend Mónica for her inconditional love and affection. You encourage me to become better every day. To them I dedicate this thesis.

“The only way of discovering the limits of the possible
is to venture a little way past them
into the impossible.”

Arthur C. Clarke

Contents

List of Figures

List of Tables

Listings

1 Introduction

This first chapter introduces the main topic of this work. It is organized in six different sections: Section 1.1 sets up the framework for the thesis, Section 1.2 explains the motivation of this work, Section 1.3 exposes a state of the art of existing Deep Learning solutions in the Videogame industry, Section 1.4 sets down the proposal developed in this work, Section 1.5 explains the planned schedule of this work, and finally Section 1.6 details the structure of this document.

1.1 Overview

In this Bachelor's Thesis, we have explored the Artificial Intelligence field applied to videogames and how can fit the Deep Learning field there. For that reason, we have reviewed the different approaches of Deep Learning applied to the game development industry, and we have drawn conclusions from them. We have also implemented many different deep learning approaches inside the same videogame and compared its results, either from a performance and from a behaviour point of view, both looking at the perspective of the player and the game developer. Finally, several conclusions and ideas to improve the current development of deep learning inside the field have been proposed.

1.2 Motivation

This document presents the work that was carried out to prove the knowledge acquired during the Bachelor's degree in Multimedia Engineering taken at the University of Alicante between the years 2014 and 2019. The motivation of this work stems from diverse sources.

First, a collaboration with *From The Bench Games, S.L.* was pursued. Concretely, helping with the development of Deep Learning bots and Artificial Intelligence related tasks, especifically in the development process of a mobile game. This possibility, allowing us to collaborate with an industry project in a production environment was a big opportunity for this research, since it allowed us to check the obtained results with a real-world application being used globally. This also allowed us to see the difference of approaches between the academic world and the industry.

Secondly, on a personal level, the idea of this project was born by the motivation of the student to research about Artificial Intelligence in the videogame industry. Finally, the last reason behind the motivation of this work is the learning of a new field: Deep Learning in this case, and the improvement of the knowledge in the field of Artificial Intelligence (AI).

1.3 Related Works

Although in recent years a high number of advancements inside the Deep Learning field have been made using videogames as a platform for research, we have seen that almost none of them have approached the task of using those neural networks as part of the development process of a game's Artificial Intelligence. Instead, they have focused on developing models able to play and replicate human behaviour, but not agents capable to interact with humans during their play and enhace their experience.

Known examples of this are the emergence of Deep Reinforcement Learning intelligences capable of playing classic Atari 2600 games [5] in the year 2013, where in some games they were capable of defeating human experts by obtaining better scores than them in the games.

More known examples are the emergence of AlphaGo [55] first and AlphaZero [56]. AlphaGo was capable of defeating a world champion of Go, a classic board game, achievement never made before by a machine. On the other hand, AlphaZero, which was capable of playing different games, such as Go, Shogi and Chess, defeated the best AIs until then on each one of the games. All the learning of AlphaZero was done by playing against itself in those games.

Finally, since the start of this project, new and great examples of Deep Learning usage for videogames with research purposes have appeared. First, AlphaStar [60], a successor of both AlphaGo and AlphaZero developed by the DeepMind Team, was capable for the first time of defeating two professional Starcraft II players, a Real Time Strategy videogame. Later, OpenAI Five [15], an Artificial Intelligence capable of collaborating and cooperating with other AIs or humans, defeated three professional Dota 2 teams, being one of them the current Dota 2 world champion and showing never seen before advancements in the field of Artificial Intelligence in terms of cooperation.

Although we can cite many advancements developed in the recent years combining games, specially videogames, and the development of newer, better AIs with Deep Learning and Deep Reinforcement Learning techniques, the same advancements have not been seen on the development of Artificial Intelligence for game development. In this work we will explore the reasons behind that difference of development and try to explore solutions to overcome that gap.

1.4 Proposal and Goals

The main purpose of this project is to present improvements in the practical applications of Deep Learning techniques inside the game development industry, specially in the development of agents, and compare their functionality with tradicional agents in the game development field, such as those developed with Decision Trees or Behaviour Trees.

For that reason, the objectives of this work are the following:

- Review the current state of the art on the development of AI for videogames.
- Explore the most common Deep Learning techniques relevant to this project.
- Analyze the most recent Deep Learning solutions inside the videogame industry.
- Create a Deep Learning development pipeline capable of supporting the integration of neural networks inside the game engine of our choice.
- Train and compare Deep Learning models with different approaches, with the purpose of training an agent capable of playing a chosen videogame.

1.5 Schedule

The proposed goal will be carried out in a timespan of eight months. Starting at the beginning of the project in October, we will move through the different stages of the development, including an internship in *From The Bench Games, S.L.* in the period between November and January, and finishing the project by the date of 30th May. The different stages of the development can be seen in Figure 1.1.

We can observe that the first stage of the development is a learning stage, where we will research about Artificial Intelligence (AI) applied to the videogame industry first. Later, we will investigate about Deep Learning techniques relevant for our work as they come up, and in parallel about Deep Learning applications relationed with videogames.

Then, in parallel with the research process, we will start the development of the project, in the dates accorded with *From The Bench Games, S.L.* for the internship period. In that internship, we will first create a Deep Learning development pipeline to carry out our work, and in parallel to maintaining that pipeline we will create neural networks to solve the task proposed by the company.

Finally, the last months of this project will be focused on the redaction of the different parts of this document. The redaction not only includes the development of each chapter, but the creation of figures and the obtention of data from the work done previously to present it appropiatedly.

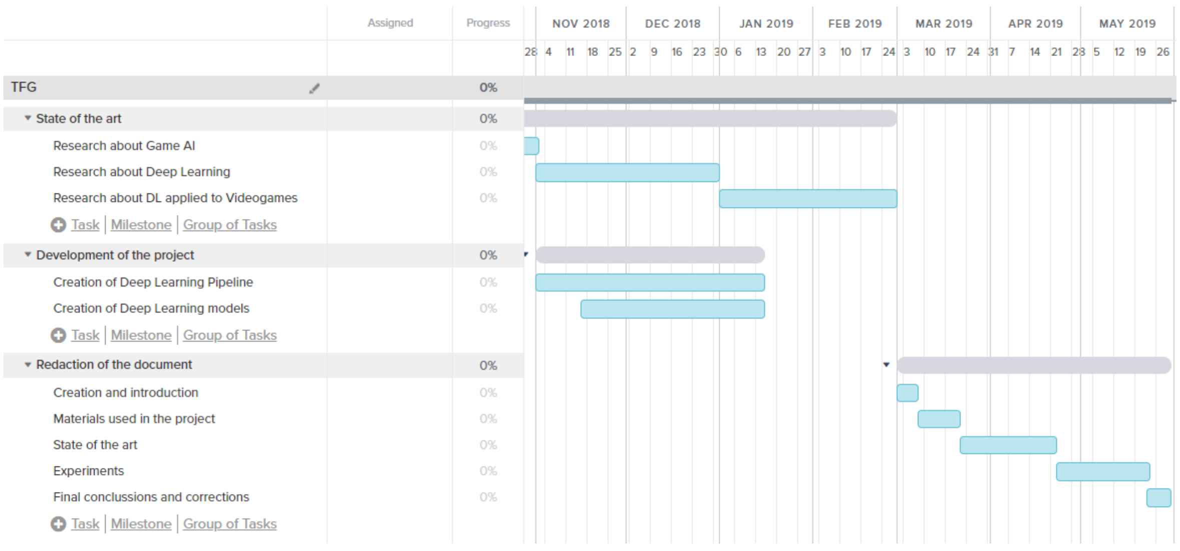


Figure 1.1: Gantt diagram of the project timeline.

1.6 Outline

The structure of this document is the following: Chapter 1 serves as the introduction of the project and its motivations, stating the current structure of this document and the objectives to be accomplished. Chapter 2 describes the hardware and software tools that are going to be used in the project. Chapter 3 reviews the most current approaches, further detailing the most relevant ones and highlighting the main lines of research in this field. Then, in Chapter 4 the implementation is presented and the main methods implemented are tested and compared. Finally, in Chapter 5 conclusions are extracted from this project and future lines of research are presented.

2 Materials

In this chapter we describe the different technologies we used in this work. The chapter is divided in: Section 2.1, where we summarize the context of this chapter, Section 2.2, where we describe in detail each of the software technologies used in this work, and Section 2.3, where the hardware used in this project is presented.

2.1 Introduction

To be able to approach this work, we need the support of many different technologies. First, since we want to approach the development of a videogame's artificial intelligence, we need a video game in a development stage, in which to research, and their related development tools to interact with.

Furthermore, we need the technologies to be able to develop different deep learning models and the libraries to migrate those models into the current game engine, in order to integrate the research process in the game development workflow. All the software technologies needed for this project will be addressed in Section 2.2.

Finally, to be able to develop our project and train different neural networks, and to be able to compare succesfully different implementations too, we will need a certain number of hardware tools. Mostly, we will need hardware platforms for the development of the project and the training of the neural networks on the one hand, and devices for testing purposes on the other hand. All the hardware used on this project will be presented in Section 2.3.

2.2 Software

In this section we will describe the different technologies which were used for the development of this project.

For the development of the different Deep Learning models we have implemented, we took both Python and C# as our programming language of choice, depending on the part of the development pipeline we worked on at the moment. As a tool for version control we used GIT, with a repository hosted on *BitBucket.org*, and Visual Studio Code as our IDE of choice.

As for the rest of the software used in this project, it can be divided in three categories: software for Deep Learning, software for game development and software for integration. For the development of Deep Learning models we will use two libraries: Tensorflow and Keras, since it will be done mostly on Python with the support of existent Deep Learning solutions and examples.

On the other hand, HardBall, as our game, and Unity, as our game engine, will be our choices for the game development software needed. This will make C# as the language for the game development stage, as it is one of the most common in the videogame industry.

Finally, for integration of these two parts, we will use two libraries: TensorflowSharp and ML-Agents Toolkit, that are specially designed for purposes closer to what we need. They will serve as a bridge between the Python part and the C# part of the code.

Tensorflow and Keras will be used to create our different Deep Learning models, and they both will be our Deep Learning part of the development process. We will mainly work on Keras for developing our models, in order to streamline the workflow. Since we are going to focus on the development of very different Deep Learning approaches, we are going to need the ease of development provided by Keras, that will allow us to focus on the definition of those models rather than the internal implementation. As Keras needs one low-level Deep Learning library as their background, we will choose Tensorflow, since it is the most compatible with our already established development process in C#. We will also use some of the Tensorflow's tools for saving and exporting our models in the different formats required, and to visualize the models we created.

In order to integrate those models in a production environment, we need a game in a development stage. At this point, we could chose to develop one game for our research purposes, or search for an existing solution. Thanks to the collaboration of FTB, we had access to HardBall. HardBall is a multiplayer football game for Android devices that confronts players in one-on-one or two versus two matches. It also features different player powers. The intention of FTB was to develop a bot using Deep Learning for the companion in the two versus two game mode, and one for the enemy in both game modes too. This served perfectly for the purpose of this research, since it had all the qualities we wanted.

HardBall was being developed in Unity, a game engine based on C#. For that reason, all the development of the AI for that game will be done in Unity too, including the Deep Learning models and their integration on the game. We will use most of Unity's tools for different aspects of our development, such as scripting, building and profiling our application's performance.

The C# integration of the Tensorflow models will be done in TensorflowSharp, since it provides C# bindings for the Tensorflow library and allows us to communicate Tensorflow with Unity, to ease our development process. We will also use TensorflowSharp as our inference engine, to obtain our model predictions from Unity when needed.

For further experimentation and research inside Unity, we are going to use their tool ML-Agents Toolkit, which allows to work with Machine and Deep Learning models inside Unity natively. Even though its focused on Deep Reinforcement Learning mainly, its structure will serve us as a basis for developing the implementation of our models inside Unity.

2.2.1 Software for Deep Learning

2.2.1.1 Tensorflow

Tensorflow [7] is an open source software library for numerical computation in the way of *data flows*, where data, organized in multidimensional arrays called tensors, is moved through a series of mathematical operations following a graph structure. It was developed by the Google Brain team as a tool for Machine Learning development, and released as an open software library (Apache 2.0 license) in 2015, rapidly growing into one of the most active repositories of Github. Nowadays, despite Tensorflow's initial purpose, it is mostly used for the research and development of Deep Learning models. According to Github's data [8] Tensorflow's official repository was ranked as

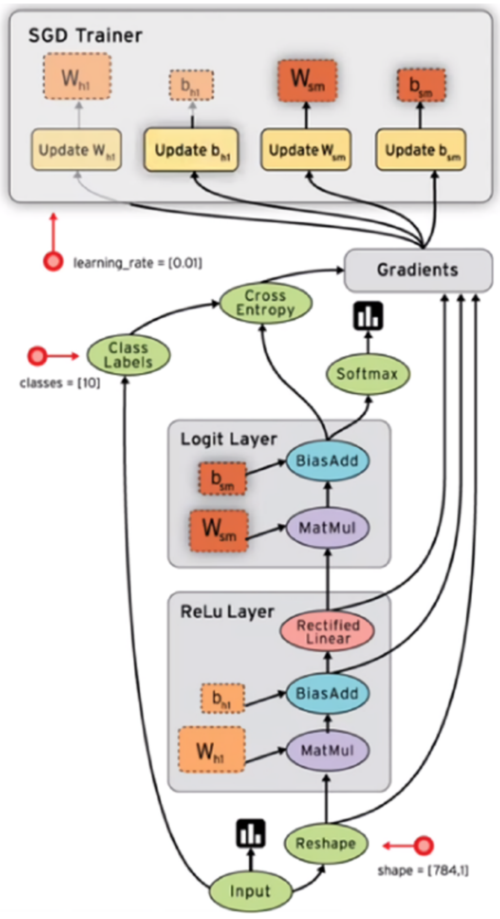


Figure 2.1: Tensorflow graph visualized using Tensorboard. Image obtained from Tensorflow.org.

the third bigger repository in terms of number of contributors, and also some of its derived repositories where among the fastest growing of all the platform.

Tensorflow's core is mainly developed in C++, and provides a higher-level interface in Python, although it supports more interfaces in other languages such as C++, Go and Java. It also supports, due to its architectural design, building and deploying graphs either on a Central Processing Unit (CPU) or a Graphic Processing Unit (GPU).

As mentioned before, Tensorflow works with the structure of a *data flow* graph. In this structure, the data is moved through cells containing different operations to be applied to the data. In the Figure 2.1 we can see a visual representation of one Tensorflow graph obtained with TensorBoard, an open source tool contained inside of Tensorflow's library for obtaining visual representations of graphs. In that figure, we can see how the data, organized into tensors, is moved through cells which contain mathematical operations, suchs as matrix multiplications and additions. Moreover, we can observe that some of those cells, called layers, are composed of smaller cells. This is because Tensorflow is based on the idea that any kind of operations, independently from its complexity, can be expressed as a group of basic functions and primitive operations.

This approach has many advantages for the development of Deep Learning models, because it allows to calculate the derivative of each operation individually and apply the chain rule, which is very useful for the loss functions based on gradient descent methods, a common approach in modern Deep Learning. It also helps to parallelize the operations between different GPUs/CPUs and to divide the model into different paths for different purposes.

Tensorflow is compatible with many different formats for saving and loading its models. The most used of them is the checkpoint, which lets us save a graph already trained or halfway through training, allowing to restore different training points of a model and compare its performance along the training period.

An aditional number of tools is provided by Tensorflow for further functionality. Some of those tools are the already mentioned Tensorboard, for graph visualization through a web interface; FreezeGraph, to convert our models into different supported formats, converting the operations variables into constants; and TransformGraph to apply transformations, such as removing operations no longer needed, to our already saved models.

2.2.1.2 Keras

Keras [9] is a high level Deep Learning open source framework developed by François Chollet in 2015. The aim of Keras is to serve as a middleware between the user and another Deep Learning library supported by Keras, which in this context is called backend, allowing for a faster prototyping and an easier development of Deep Learning models. Because of this, Keras allows the user to define a Deep Learning model interacting only with their own API, and then Keras is responsible for building that same model for the chosen backend.

The backends supported by Keras are Tensorflow, Theano [10] and Computational Network Toolkit (CNTK). It allows the user to interact directly with the chosen backend through Keras, without almost no operating differences between them.

Keras defines two different APIs to create Deep Learning models with: the Sequential API and the Functional API. The Sequential API consists of a linear stack of layers, and is created just by passing a list of layers and an input shape for the first one. The rest of the shapes can be obtained automatically with shape inference. The Functional API allows more complex models, since you can build multi-output models or models with shared layers. For building a model with the Functional API, you need not only to define a list of layers, but also to state which layer precedes each other, being responsible for matching the shapes between all layers. This system allows the user to have more control over each layer and the internal structure of the network, designing more complex models without much more effort.

Keras provides a number of the most common datasets for an easier access and experimentation in a varied number of topics, such as MNIST, CIFAR10 and the IMDB Movie Review Sentiment Classification. It also provides a number of the most common and well-known models for prediction and feature extraction. Those models are made available with pre-trained weights, and among them we can find the ResNet [11], InceptionV3 [12], VGG [4] and more aditional networks.

2.2.2 Software for game development

2.2.2.1 HardBall

HardBall is an indie football game for Android devices developed by FTB in Unity. HardBall is a fast-paced game with a cartoon aestethic which consists of a mix between football and air-hockey. It is inspired on the popular browser game HaxBall. In HardBall, users can compete in both one-on-one or two versus two matches, both with and against AI bots or other online players. HardBall is targeted to a wide range of mobile devices, both high-end and low-end cell phones.

To explain what a typical game of HardBall consists of, we are going to take Figure 2.3 as a reference, which depicts a screenshot of HardBall's two versus two game mode. In that game mode, the player is identified with a yellow marker, his companion with a blue marker and the two enemy players with a red marker. Also, we can see the ball in the centre of the field. At the top of the image, we can see the score of each team, and between them the time left until the game ends. Furthermore, on every top corner we can see the team's name (in this case, we are playing against an AI bot), and their division (the lower the number is, the more experience the player has in HardBall).

The range of actions each player can make is limited to a few. First, the player can move in every direction around them with a joystick. In addition, they can kick, which means that if the ball is in contact with the player, it will be shot in the player's opposite direction, bouncing in the edges of the field if the case occurs. Finally, the

Figure 2.2: HardBall cover image. Obtained with the permission of *From The Bench Games S.L.*

player can also use a special power, selected from a list of a few before the game starts, such as speeding up or pushing away enemy players from him. This special power is the only action other players, such as his companion or the enemy players, are not allowed to do.

At the moment of the start of this project, the system used to take decisions for the

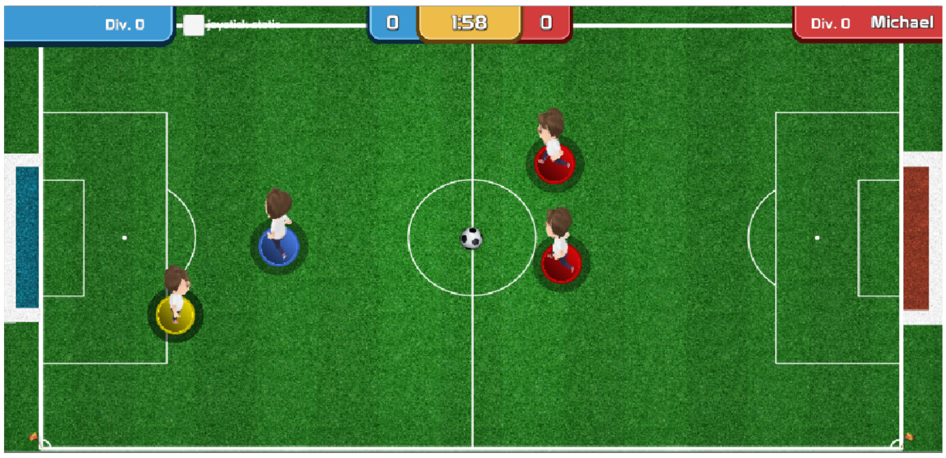


Figure 2.3: Screenshot of a game of HardBall's 2v2 mode. Obtained with the permission of *From The Bench Games S.L.*

AI bots was a Decision Tree, which we will see in more detail in chapter 4. Besides, it is important to notice that this version does not differentiate between friendly or enemy behaviour, because all the bots in the game, whichever the team they are on, use the same decision tree for all of its functionability.

Moreover, this same version was published in Google Play in a closed Beta state and collected different data from the players, with the intention of building a private dataset for training agents with Deep Learning methods. The data is collected using a script integrated inside of Unity, and stored in a private database. The information that is recorded is the game mode, the position of every one of the players and the ball in Unity's coordinate system, and boolean variables for different actions, such as if the player has shot the ball or if a goal was scored. This information is stored for every iteration of the game loop, for every game.

2.2.2.2 Unity

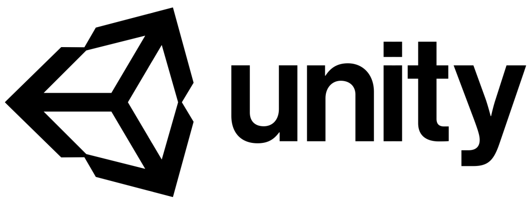


Figure 2.4: Unity logo. The Unity engine can be downloaded from `https://unity.com/`, from where this logo was obtained.

Unity is a cross-platform game engine developed by Unity Technologies. Even though the main purpose of the engine is the development of 3D videogames, it is also compatible with 2D games, as well as simulations, animations and applications that require 3D solutions in general. Unity first version was released in 2005, and it has been updated frequently since then. Currently, three big major versions are released every year, with the name of the version being the year of its release and the version number in that year (for example, 2018.2).

Unity development is mainly done with the C# programming language, despite having had other compatible languages in the past, like Javascript and the in-house language UnityScript. Currently, Unity supports two different runtimes: .NET 3.5 and .NET 4.6, although it is recommended that all new projects start to use the latter. Alongside with it, it supports C# 6 version of the programming language. Recently, Unity has released the new Burst compiler, capable of translating some portions of its

C# code into high-performing native code. An example of its usage was shown in the project Megacity [13], where a procedurally generated city with hundreds of meshes, AI agents and sounds is rendered in real time.

Unity's internal structure follows the component architectural pattern, where every object in the scene is defined by the number of components attached to it. In this structure, a component can be an independent piece of information or functionability, for example, a 3D mesh or an audio source. In Unity, all objects present in a project are contained inside a scene, where it is organized following a tree structure.

According with data [14], the engine is considered one of the most popular and used game engines in the market, specially in the independent game development scene, and in the development for mobile applications. It stands out in the development for Augmented Reality (AR) and Virtual Reality (VR) devices too.

Moreover, it is also one of the most used game engines in education and research, supported through the brand Unity Labs. Among its topics of research we can find VR, AR, computer visualization, AI and Deep Learning. Through Unity Labs, Unity supports multiple projects for research applied to the engine. For Deep Learning, Unity has specialized support with the library ML-Agents Toolkit. For more information about ML-Agents, check the Subsection 2.2.3.2.

Unity provides support for building projects for 29 different platforms. Among them we can find all the most common operating systems for PC architectures (Windows, Linux, Mac), and also most of the gaming consoles in the market (PlayStation 4, XBOX One) and mobile platforms (Android, iOS, Windows Phone), along with other platforms such as VR headsets and web-based applications.

2.2.3 Software for integration

2.2.3.1 TensorflowSharp

TensorflowSharp is an open source (MIT License) library created in 2017 by Miguel de Icaza, which serves as .Net binding for the Tensorflow library. This allows to interact directly with Tensorflow using a strongly-typed API both in C# and F#. This library surfaces all the low-level Tensorflow library, although it does not include a high-level API like the Python layer on the Tensorflow library, falling in a middle area and serving as a cumberstone for the high level operations usually defined on the Python part.

TensorflowSharp allows the user to import models developed in Keras and Tensorflow, trained or not, feed them with data and run inference on them. Because of this, it facilitates the process of deploying those models in production environments developed in .NET languages.

TensorflowSharp's main functionality comes from the TFGraph class, which serves to load and manipulate pre-trained models. The library also provides the TFTensor data class to natively work with multi-dimensional data arrays in C#, and to provide such data to the mentioned TFGraph class for inference.

However, it is important to remark that, even when TensorflowSharp is fully functional for loading and infering predictions from a pre-trained model, training a new model is not supported, since it does not provide optimizer classes or functions. For that reason, it is usually recommended to do that step natively in Tensorflow, and only use TensorflowSharp for the deployment stage.

2.2.3.2 ML-Agents Toolkit

ML-Agents Toolkit [1] is an open source project developed by Unity Technologies that serves as a tool for the creation and usage of Deep Learning models inside of Unity. The library works as a plugin for the Unity Engine, and allows to use games and simulations developed in Unity as environments for training agents. Although ML-Agents is mainly focused on Deep Reinforcement Learning, it can be used for imitation learning, neuroevolution and other machine learning methods too.

The library provides a number of tools to define learning environments inside the Unity Editor. ML-Agents main functionality is composed of two packages, the ML-Agents Software Development Kit (SDK) and a Python API. The core functionality, included in the SDK, allows to create a learning environment. In ML-Agents context, a learning environment is a Unity Scene where a number of agents, observations and a particular policy is defined, either for inference or for training new agents. On the other hand, the Python API serves to interface between the engine and Python, allowing to control built environments from there.

In the Figure 2.5 we can observe a visual representation of the components a Learning Environment defined by ML-Agents has. In that figure, we can differenciate three types of Unity components: Agent, Brain and Academy.

An Agent Component is a Unity GameObject placed in a scene, usually in a GameObject with a visualizable mesh associated with. Every Agent is responsible for collecting observations from the scene and for executing actions in that environment. Every Agent must be linked to a single Brain, which is responsible of taking the decisions for them.

A Brain Component is also a Unity GameObject placed in a scene, but unlike the Agent component, it is usually not attached to a rendered object nor one with additional functionability. A Brain provides the policy which the Agents asociated with follow. A Brain can be attached to any number of Agents, but only the Agents with a valid observation and action space can be attached to it. Although the types of Brains have changed in following versions, as of the version 0.5 of ML-Agents (the one mainly used in this project) there are four different types of Brains: Player, Heuristic, Internal and External.

A Player Brain is, as it name suggests, a Brain whose actions are mapped into keys and actions controlled by the player of the game. This is specially useful for testing the Agents actions when developing a learning environment. A Heuristic Brain is one with the logic hand-coded by extending the Decision class. It requires to define a script

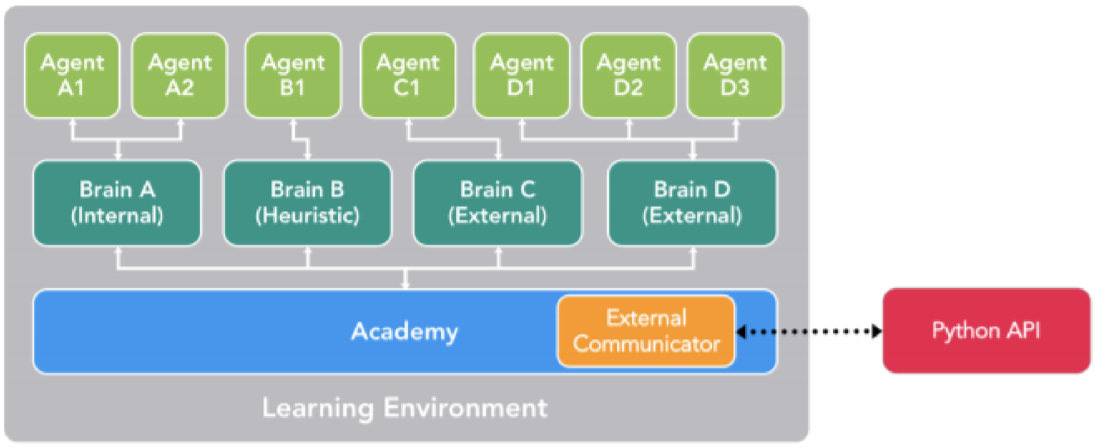


Figure 2.5: General structure of a learning environment in ml-agents toolkit v0.5. It features an Academy in communication with an external Python program through the Python API, four different Brains(one Internal, one Heuristic and two Externals) and seven different agents attached to those brains. Image obtained from the paper ”Unity: A General Platform For Intelligent Agents” [1].

where to delegate the decision process of the Brain.

Both External and Internal Brains are prepared to be used for usage with Deep Learning algorithms. On the one hand, an External Brain is one used for training using the environment, usually controlled through the Python API. On the other hand, an Internal Brain is used for creating Agents with an already trained Brain which we do not want it to keep learning. Internal Brains also allow to load a pre-trained model in *.bytes* format for inference, even when it has not been trained as an External Brain.

Finally, the last component of the library is the Academy Component. The Academy Component is responsible to orchestrate all the Agents and Brains in a Unity scene. A learning environment must contain one, and only one Academy in usage. The Academy will be responsible for initializing the environment and setting the values of the training, such as the vector of data the Players are going to collect from the environment or the number of frames accumulated by the Agents before a new inference is done by its associated Brain. An Academy is also responsible of communicating with the Python API, since it controlls all the learning environment.

In the following ML-Agents versions, the External and Internal Brains have been combined in the Learning Brain, which can be set to Training Mode, which works similarly to the External Brain, or to Inference Mode, instead of the Internal Brain. Moreover, the format a Brain expects to receive has changed in recent versions too, from the *.bytes* format to the *.nn* format.

On the other hand, until ML-Agents version 0.7, the library relied on Tensorflow-Sharp for running the models and obtaining predictions, which required to install it in the project too. Starting with the version 0.7 release, ML-Agents provides his own in-house Inference Engine, replacing the usage of TensorflowSharp in the project, and no longer requiring it as a dependency. Unity Inference Engine is focused on cross-platform compatibility and performance, with a small size (600KB), better for working on certain platforms such as mobile or some embedded systems.

Along with this functionalities, ML-Agents also provides a large number of example environments with pre-trained agents, both for learning about the library features and to serve as a starting point for further research on Reinforcement Learning tasks. These environments provide examples of ML-Agents multiple functionalities, such as single or multi-agent scenarios and discrete or continuous action spaces. A general view of some of the examples provided in the Toolkit can be seen in Figure 2.6. Although is not the purpose of this section to describe in detail each paper, its important to remark that one particular environment, called *Soccer Twos*, shows a little version of a two versus two football game with one player and one goalkeeper in each team.

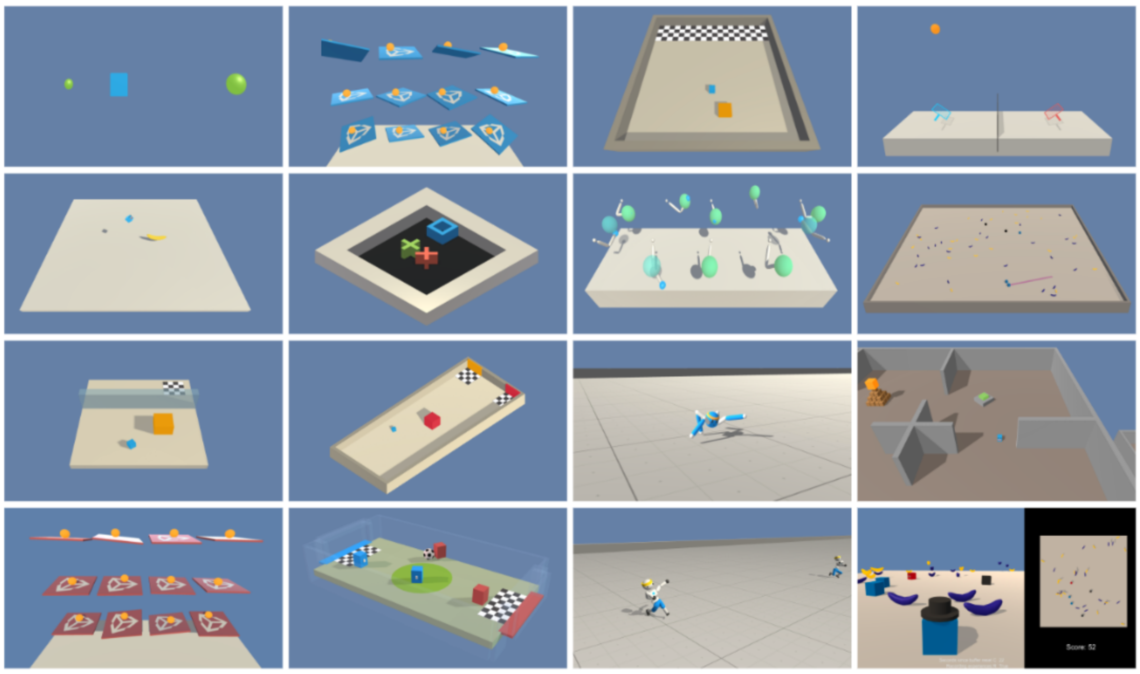


Figure 2.6: Pre-built environments provided by the ML-Agents Toolkit. Image obtained from ML-Agents Toolkit repository under the License Apache 2.0 (`https://github.com/Unity-Technologies/ml-agents`).

ML-Agents also provides a number of built Reinforcement Learning algorithms and modules, to serve as a base of experimentation for further research. Among the algorithms provided by Unity, we can find Proximal Policy Optimization (PPO), a state of the art Deep Reinforcement Learning algorithm used by OpenAI for the development of OpenAI Five [15], a Deep Learning Bot trained to play a videogame (Defense Of

The Ancients 2) competitively. This algorithm is provided with the option to extend it using a LSTM Cell and a Intrinsic Curiosity Module. Finally, an implementation of Behavioral Cloning, a simple Imitation Learning algorithm is provided too.

A number of projects have shown the usage of ML-Agents in production environments with success. A demo game was built by the Unity team to showcase the usage of the library with the name of Puppo, The Corgi [16]. There, a little dog called Puppo is trained to move and interact with the player using only a Reinforcement Learning algorithm and a set policy. The movement was developed using Unity's physics engine, therefore making the dog learn to walk in the process. Other projects have been built since then by external partners using ML-Agents. One remarkable example is Snoopy Pop [17], a mobile arcade game of the bubble shooter genre, which was trained to emulate a user playing the game.

2.3 Hardware

Apart from all the software tools needed to develop this project, it also needs hardware devices, both for training the neural networks developed and for testing the game, both in computers or mobile devices. For that reason, two different machines have been used for this project: a computer, that is going to be used for all the development process and the testing on Windows platforms, and a mobile phone compatible with Android, that will be our testing device for the final versions of the game.

The technical details of the Windows computer used on this project are the following:

- **CPU:** Intel® Core™ i5-7300HQ / 4 Cores 2.50 GHz
- **GPU:** NVIDIA® GeForce® GTX950M / 2GB GDDR5 VRAM
- **RAM:** 8GB DDR4 2133MHz
- **Storage:** 1TB 7200rpm SATA

The technical details presented here are the ones that result relevant for the training and testing of the project. For that purpose, the main priority when choosing our hardware is obtaining the fastest processing speed possible, because we expect the training process to be a large part of the development time and it could impact the schedule of the work the most. Since the technologies we planned to use already support the usage of GPU-based training, both CPU and GPU result important here, since both could be used for the training process. Moreover, for testing purposes the processing units result the most relevant parts of the hardware, since we will compare the number of frames per second produced by the game at different points of the development, and for that reason, the better the CPU and GPU are, the bigger the framerate is.

Finally, both rapid access memory and hard-drive storage can impact the speed of the workflow: RAM will allow us to load larger amounts of data on memory simultaneously at the time of training, and the storage device will let us to load and save faster new files from memory, and also in larger quantities. RAM will also impact the performance at the time of testing, in a same way as explained with the CPU and GPU.

For testing on mobile platforms, a *Xiaomi Redmi 4X* has been chosen, with the following technical details:

- **CPU:** Qualcomm MSM8940 Snapdragon 435 / Octa-core 1.4 GHz
- **GPU:** Qualcomm Adreno 505 / 450 MHz
- **RAM:** 4GB
- **Storage:** 64GB
- **Battery:** Li-Po 4100 mAh

Since this device is only planned to be used for testing purposes, the most relevant parts of it are the CPU, GPU and RAM, since we will look at the framerate obtained by playing HardBall on it at different points of the development. The battery becomes relevant too, because we will use that as a way of comparing battery consumption between different versions of the software.

3 Related Works

In this chapter, we review the current state of the art in the Deep Learning field applied to game development. The chapter is divided in four different sections. Section 3.1 sets the context of the chapter. Section 3.2 summarize and analyze the development of Artificial Intelligence for videogame development. Later, in Section 3.3 the most relevant Deep Learning techniques to this work are introduced and explained. Finally, in Section 3.4, the relevant Deep Learning work applied to the Videogame Industry is exposed.

3.1 Introduction

In the following sections, we review most modern deep learning works applied to the game development industry. This is mainly extracted from the most relevant Deep Learning and Game Development journals and conferences. Both projects that use games as a platform for research and projects that use Deep Learning techniques for its Artificial Intelligence are taken in consideration.

3.2 Artificial Intelligence in the Videogame Industry

Artificial Intelligence in the videogame industry has been one key aspect in the field, and its relation with it is almost as old as game development itself. Even before of the existence of videogames, AI was applied to game environments to research about the limits and possibilites of computers in solving tasks that seemed to require intelligence. A good example of this can be found on Alan Turing, the father of computer science, who reinvented and adapted the Minimax algorithm for playing Chess [18].

Since then, many researches carried in early AI have used game environments, mostly classic board games like Checkers and Chess, to push forward the limits of the field. Important milestones related to this are the defeat of the Checkers World Champion in 1994 [19], work that later ended solving Checkers on 2007 [20], and the defeat of the Chess World Champion Gary Kasparov in 1997 by IBM's Deep Blue [21], with a highly-adapted and modified version of Minimax.

AI in the videogame industry is not only used for research, using videogames as more complex game scenarios with a defined set of rules, instead they are mostly used as a tool of game developers themselves. In opposition to popular belief, the first videogames did not feature a single player mode, instead they were mostly two versus two games where one human played against other. Examples of this are popular games such as *Pong*, *Space Race* or *Gotcha*, mostly developed in the 1960s and early 1970s.

But soon after that, games with a single player mode started to appear, usually replacing the figure of the enemy played with a very simple AI composed of a predefined behavior that all enemies blindly followed. We can mention games like *Speed Race*, which also was the first vertical scrolling videogame, or *Qwak!*, as examples of this early use of Artificial Intelligence in videogames. Moreover, due to all enemies having, tipically, the same predefined behavior, a random element was added usually to achieve a false sense of intelligence in the enemies, and avoid the players to feel that they were facing always the same predefined rival.

The rise of video games as entertainment and the popularization of them among the population soon increased the demand of more engaging experiences, in pair with the development of better machines, capable of creating bigger games with better graphics. It was during this time, known as the golden age of arcade video games [22], that we can find examples of more complex AI, with remarkable examples like *Space Invaders*, which had distinct enemy patterns, an increasing difficulty level and in-game events based on the player's input, and *Galaxian*, which included enemy maneuvers who breaked out of formation to act independently, oppositedly to what was usual in games until then.

A very important example contemporary of that time is *Pac-Man*, which featured different personalities for the enemies of the game [23]. In Pac-Man, the player faced the task of collecting a number of points in a maze, while avoiding four ghosts, each

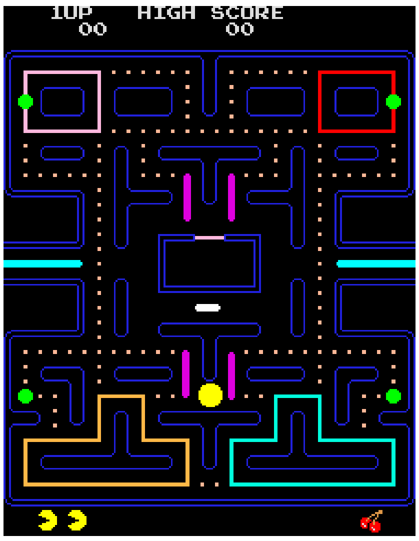


Figure 3.1: Scatter areas for the four different ghosts in Pac-Man. Each color defines the area where the ghost with the corresponding color moved when they were in scatter mode. Image obtained from `https://dev.to/code2bits/pac-man-patterns--ghost-movement-strategy-pattern-1k1a`

one of a different color, whose contact was typically lethal to the player under most circumstancies. Because of the structure of the game, most of the game engage and fun for the player relied on the AI developed for it.

To achieve the task of developing an Artificial Intelligence capable of showcasing different personalities for the four enemies of the game, while still being possible to work in a computer of that time at the frequency required by a videogame, a general scope intelligence was developed. This general purpose AI worked as a simple finite state machine, with three different states: Chase, Scatter and Frightened.

Chase was the state in which the ghosts behaved most of the time, trying to reach the player's position to make him lose a life and potentially finish the game. Scatter was a game state introduced to give the player a little break, making the enemies move to a different corner of the map, thus being less aggresive with the player and expanding over the map. Finally, Frightened was triggered when the player collected one of the four items, spread over the map, that allowed him to kill ghosts when in contact with them, for a short time after picking the item.

This finite state machine worked in the same way for all four ghosts; but two main differences were introduced to make them have different personalities, apparently. First, the time when each ghost entered scatter mode was predefined, and different, for each of them. Also, it moved them to a different corner of the map, making consequently some of the ghost seem more evasive because they used to move away from the player sooner or more frequently. In Figure 3.1 its possible to observe the areas where each enemy moved when they behaved in scatter mode.

Finally, the element which made a bigger impact on player's perception of different enemy personalities was in the Chase mode. When a ghost selected where to move in an intersection, it usually was done with a random choose, influenced by predefined probabilities. Those probabilities where influenced depending on Pac-Man's position, but those probabilities where different for each one of the ghosts. In the Figure 3.2 its possible to see those differences of behaviors due to the influence of Pac-Man in each ghost surroundings.

Its in the example of Pac-Man that we can identify an important difference between traditional Artificial Intelligence and game AI. While the main objective of traditional AI is to obtain better, more consistent intelligences, capables of coping with complex real-world problems, game AI is instead focused on the illusion of intelligence a player perceives. Because of this, big differences of application can be found between the academic world and the industry. Due to this, many experts have complained about the use of the term AI when talking about game AI, because of the difference between the academic and the industry approaches [24].

Since then, always at a slow pace, more and more traditional AI techniques started to be more used in the game industry. At the beginning of the 1990s, and coinciding with the emergence of new game genres, tools like finite state machines, decision trees, fuzzy logic [25] and, later, behavior trees, supposed important milestones in the development of game AI [26] [27].

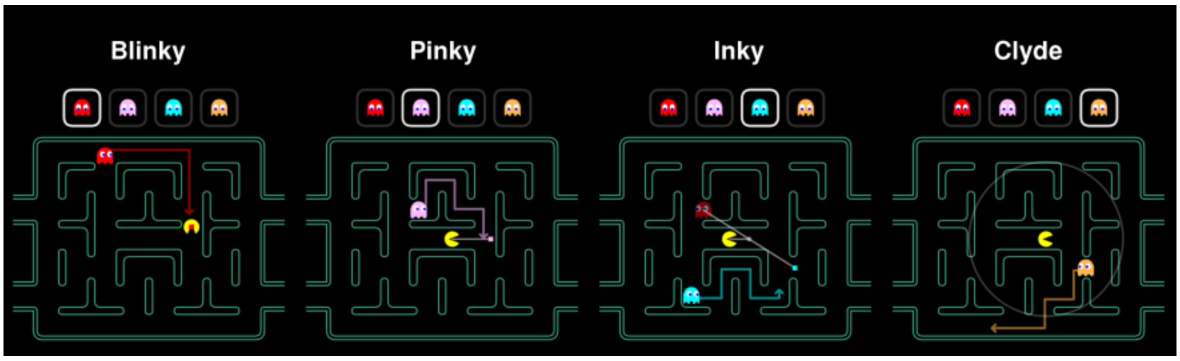


Figure 3.2: Example of different enemy movements when Pac-Man is near. It showcases a range of behaviors from more aggresive enemies, like Blinky, to more defensive or even evasive others, like Clyde. Image obtained from `https://dev.to/code2bits/pac-man-patterns--ghost-movement-strategy-pattern-1k1a`

Because of the own nature of video games and the players demands of bigger, more visually appealing games, game AI development was initially relegated to the background. The computational cost of creating bigger and better looking game worlds, with better physics simulations, forced game developers to choose between dedicating

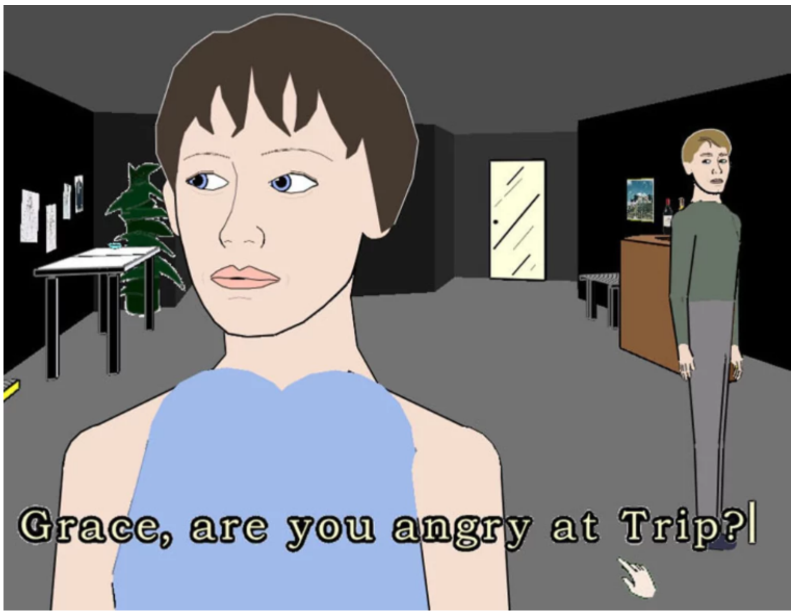


Figure 3.3: Screenshot of the game Façade, where the player, non-visible, is talking to an NPC called Grace, in the centre of the screen. The words that appear in the picture have been written by the player, and don't follow any guideline or restriction from the game. The NPCs of the game interact with the player words using a NLP system.

most of the CPU capabilities to graphics or AI, therefore making game AI development much more slower than the graphical part. Because of that, is possible to identify a clear evolution of videogames from year to year looking at the visuals, but when the AI is in the spotlight, the evolution is much more slower.

The emergence of modern GPUs, capable of rendering complex game-world scenarios with ease, left more room for the development of better game AIs. Also, the importance of the development of good AIs inside the industry has changed drastically, and now big game studios usually have dedicated teams to the development of Artificial Intelligence for their games [28].

Thanks to this, in recent years we have seen the presence of big milestones in the development of AI inside the industry. Good examples of that are games like *Façade* [29], a game fully developed around its AI, with special focus on Natural Language Processing (NLP), due to it being a conversational adventure where the player is capable of communicating with two different Non-Playable Characters (NPC) writing with the keyboard, like if he was talking to real people. A screenshot of this game can be seen in Figure 3.3.

Moreover, it is also possible to find working examples of early presence of learning intelligences, and specifically, Neural Networks, in the videogame industry. Two famous cases are the game series *Creatures* and the game series *Black & White*, which combined machine learning techniques with emergent behavior and behavior trees to create NPCs capable of learning from the players actions. In fact, the learning aspect of the game *Black & White* has been considered that it was very advanced in its time, integrating learning algorithms in the gameplay in a way that nowadays has not been replicated yet.

3.3 Introduction to Deep Learning

Deep Learning is an artificial intelligence field derived from Machine Learning that is concerned with algorithms inspired by the structure and function of the brain. Deep Learning can be explained as a subset of Machine Learning, composed by multiple (deep) layers of neurons with the intention of imitating the structure of the brain. Therefore, most of the Deep Learning methods use ANNs, which are general purpose functions with a graph structure capable of, given an expected input, learn its correspondent output and generalize that knowledge to input never seen before [30].

To explain in further detail the functionality of an ANN, we need to look at the structure of a single artificial neuron, known as the perceptron. The perceptron is a structure invented by Frank Rosenblatt to represent an artificial binary neuron [31] and its mainly aimed at the task of lineal classification through very simple learning algorithms.

In order to see in more detail the internal architecture of a perceptron, we can look at Figure 3.4, where a structure of a multi-input perceptron is depicted. There, we can identify four different elements or steps which, in total, compose a perceptron: Inputs, Weights, Summation and, optionally, Activation Function. The Inputs, which can be of any length as long as its always the same for the same perceptron, are the numerical values we are going to provide to the perceptron in order to make him predict an output. When the Inputs are provided to the perceptron, each of them is multiplied to its correspondent weight, which the perceptron must have initialized previously. In this step, a bias term can be added to gain more control over the output of the perceptron.

After that, we proceed to the Summation of all the result values we just obtained from the previous multiplications, obtaining a single value. This value may be considered the output of the perceptron, but it is possible to modify that obtained value with an activation function, which will be useful for mapping the obtained value to a certain range where the output is expected, in addition to allowing for a better training when several perceptrons are connected in different layers.

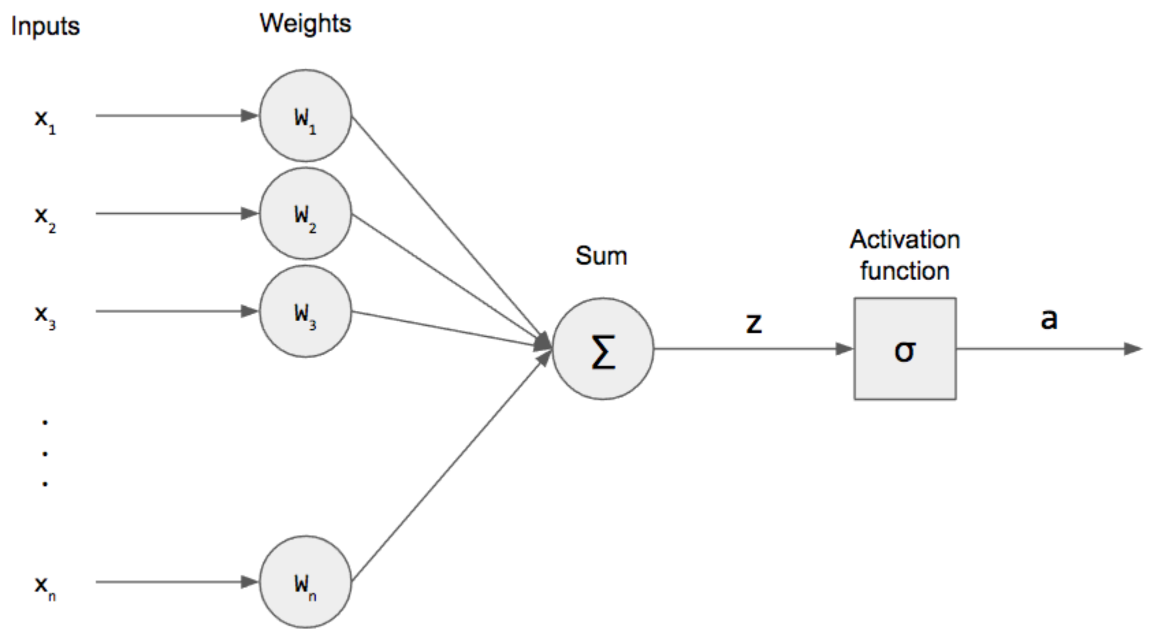


Figure 3.4: Structure of a perceptron. It contains the Inputs, defined from x_1 to x_n, the weights, defined from W_1 to W_n, a Summation, with the resulting value z, and an Activation Function, with the resulting value a. Source: `https://medium.com/@stanleydukor/neural-representation-of-and-or-not-xor-and-xnor-logic-gates-perceptron-algorithm-b0275375fea1`

Now that we understand the basic structure of a perceptron, its easier to explain the training process which happens in all these algorithms, and results in what we know as Machine Learning. As we saw before, the perceptron mainly provides a number of weights, usually initialized with a random value, equal to the number of inputs. When we do all the perceptron process with an input which we already know its expected output, its possible to compare the difference between that expected output and the output from the perceptron. That difference, which we call error, can be moved backwards through this structure, computing the partial derivatives of the error with regards to each weight, and therefore obtaining how we should update each weight to obtain the desired output. This process is known as gradient descent.

This algorithm has been proved useful in the past, and supposed the cornerstone for the development of Machine Learning first, and Deep Learning later, as the artificial neuron depicted here has been the basis for the development of ANNs. In Artificial Neural Networks, we can usually find many layers of neurons, tipically with hundreds or thousands of neurons each, capable of generalizing to very complex problems where humans have struggled in the past.

To further understand the structure of a neural network, we can look at Figure 3.5. This image, very similar to the perceptron figure, depicts the structure of an ANN with three layers, each composed of one or more perceptrons, identified here as a single node, with the exception of the Input Layer. As we can see in the figure, in modern neural networks the neurons are structured in layers, where each neuron receives as input all of the previous neurons in order (as depicted the Figure 3.4 in the first step) and produces a single output, which will serve as the input for the next layer. It should be noted that neurons do not interact with other neurons from the same layer

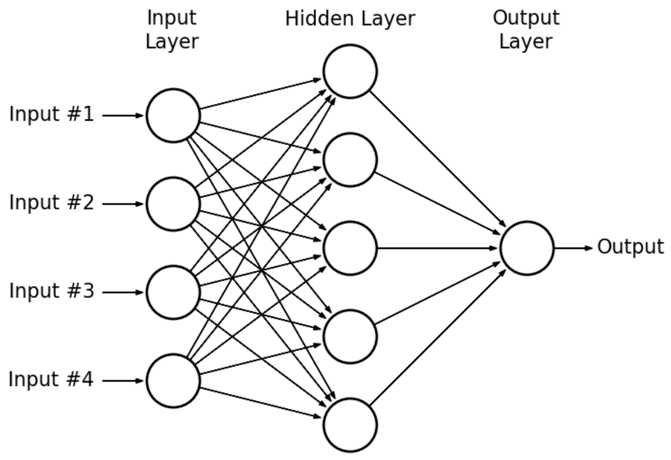


Figure 3.5: Architecture of a neural network composed of three layers: An Input layer with four elements, a fully connected hidden layer with five nodes and an output layer with one node. Image obtained from astroml.org.

As said before, the first layer, known in Deep Learning terminology as Input Layer, does not contain an initial array of weights and an activation function. Instead of that, each node in the Input Layer contains a single value ready to be provided to the network. Following the Deep Learning terminology commonly used in the field, each of the layers between the Input Layer and the last layer, known as the Output Layer, are known as Hidden Layers. Although the scheme shows a neural network with only one hidden layer, its common to have many more, as we can find in networks such as Inception V3 [12], with 48 layers, and ResNet-101 [11], with 101 layers in total.

Subsequently, when an input is carried through the network to obtain an output, the mathematical operations ocurring internally are dense matrix multiplications [32], where the data vector is multiplied by the weights of every neuron, and, therefore, obtaining a different matrix of values in each layer. For that reason, its important that the input and output shape of one layer matches the expected input and output shape of the next and previous layers, allowing to perform the matrix multiplications properly because the shape matches. Also, its important to remark that when an activation function is defined, it is done usually on a layer level, which means that every neuron in that layer will have the same activation function, with the possibility of it being different from other layers or not.

Finally, although the output layer usually has only one layer, specially for regression problems, with the predicted value as its output, it may have more nodes for classification tasks, where each node is used for one different category, and outputs the percentage of predicting that category.

As we saw before, ANNs present an structure where each input value is related in the same way to every node in the following layer. When the input contains spatial information in its structure, such as when the input is an image and every pixel has a position related to the rest of them, this presents a disadvantage because that spatial information is lost. To solve this issue, we can use CNN. CNNs are a specific type of Neural Networks that replace dense matrix multiplications with convolutional operations in atleast one layer [33].

In mathematical terms, a convolution is an operation on two functions producing a third function that expresses how the shape of one function is modified by the other. The mathematical expression of a convolution is the following:

$$y(t) = (x * w)(t) = \int x(\tau)w(t-\tau)d\tau\,,$$

where the first function, x, is tipically known as the *input*, and the second function, w, known as the *kernel*, are the functions being computed. The resulting function y, commonly named *feature map*, is defined as the integral of the product of both functions after one is reversed and shifted. When this operation is implemented in a computer, and specially in the deep learning field, the operation must be discretized, and therefore the inputs are multidimensional arrays of data with discrete sizes.

In the Figure 3.6 it is possible to see a visual example of a 2D convolution taking as input a 2D array of size 7×7 and as a kernel a 2D array of size 3×3. The feature map obtained is from size 5×5, less than the initial input value, since its not possible to compute the edges of the input because the kernel would come out of the input matrix,

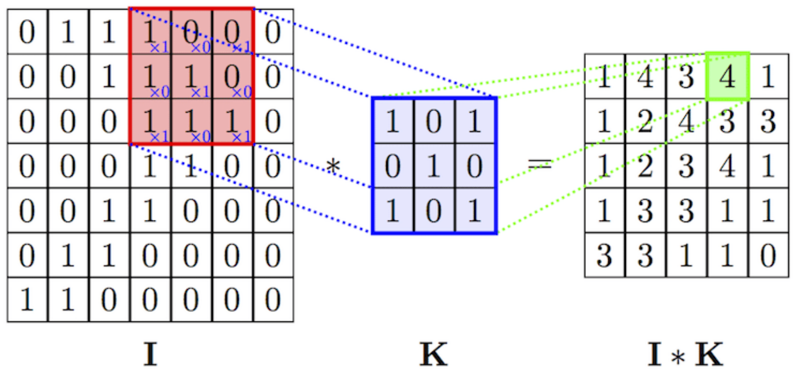


Figure 3.6: Example of a convolution operation in a 2D matrix. An Input I of size 7×7 is convoluted by a kernel K of size 3×3, producing the resulting matrix $I * K$, of size 5×5. Image obtained from the source: "Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques" [2].

and would not be able to compute anything. In case we want to obtain a feature map of the same size as the input, a stride can be added to the input to allow this operation.

In the figure, it is possible to understand how the operation is being computed. First, the kernel is placed on the input in the first available position where the full kernel fits the input and all those input values are multiplied by the corresponding value in the kernel, and then the result of all those products is added, obtaining a single value. That value is placed in the correspondent position, and then the kernel is slided one position each time, computing this operation in every step, until all the possible positions in the input have been convoluted by the kernel.

Convolutions, specially 2D convolutions, have been used widely on computer science for the detection of forms, shapes and features in images, and for adding effects, such as blur or depth, to them in postprocessing. In Figure 3.7 we can see a visual representation of this, where a grayscale image of a flower is convoluted by an emboss kernel (one particular kind of kernel to increase the perception of depth in an image) to produce a feature map, showing the result of that image transformed by this particular filter. In this case, the kernel highlights the presence of edges following a top-left to bottom-right direction, as it is possible to see.

Returning to CNNs, now that we have seen how a convolutional operation is done with regards to an input, it is possible to explain how a convolutional layer works. A convolutional layer is mainly composed of three elements: *local connectivity*, *weight sharing* and a convolutional operation, which we have just seen.

Opposed to the structure of a dense layer in classical Artificial Neural Networks, where each neuron is connected to every neuron in the previous layer, in a convolutional layer one particular neuron is only connected to a small number of neurons aligned in with and height between them. This concept, known as local connectivity, can be seen visually in Figure 3.8, where we can observe the connections of a complete layer, the

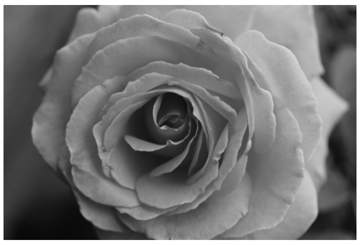
Input, to the first neuron of the following hidden layer (the other neurons are omitted for visual clarity). In the first case, we observe that the neuron treats every input in the same way, while on the second example only a small amount of values are taken into consideration. The size of those features, known as the kernel, can be determined as a hyperparameter of the network. Therefore, every neuron in the hidden layer will have the same number of inputs, but those will vary in function of the position of the resulting neuron in the hidden layer.

When a complete hidden layer is defined following this structure of local connectivity, we can observe that we are defining a convolution of an input, the previous layer, that obtains a feature map, where each value is the outcome of every individual neuron connected to only a small amount of input values.

CNNs have one more difference with ANNs that is important to remark. As observed in the previous example, it is possible to see that, typically, each neuron of the hidden layer would learn a different weight for every connection with an input element of the previous layer. This does not happen in reality due to weight sharing, which can be defined as forcing every neuron in the hidden layer to use the same weights for their relations with the previous inputs. Therefore, we can observe that the weights learned by the hidden layer act as a particular kernel, like those that could be defined to extract features from images, as seen in Figure 3.7, but that are learned inherently by the neural network depending on the features it needs to detect.

For that reason, a tipical convolutional layer does not contain a single filter, but a stack of them, with each of them being trained, usually, to detect a different feature on the image, depending on the needs of the task the neural network is performing.

Convolutional Neural Networks also feature one more type of layers not common in ANNs: Pooling Layers [34]. A Pooling Layer is a layer that reduces the spatial dimensions of a CNN following a predefined pattern. It does so by going through the image taking groups of data of a certain size, in a similar way of a convolution, but reducing all the values each node obtains as an input to a single output value. The most common type of Pooling is the Max Pooling, where the resulting value is the


(a) Original

$$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

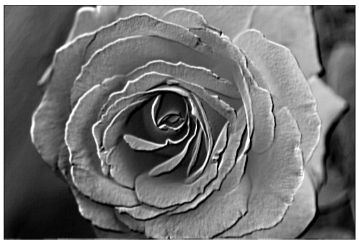
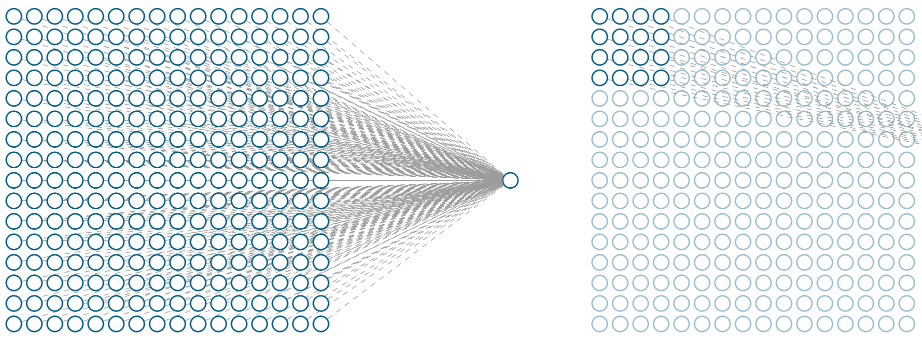

(b) Filtered

Figure 3.7: Effect of a 3×3 emboss kernel over a grayscale image (a). The feature map (b) gives the illusion of depth by emphasizing the differences of pixels in a given direction, in this case, the left-to-right and top-to-bottom diagonal.

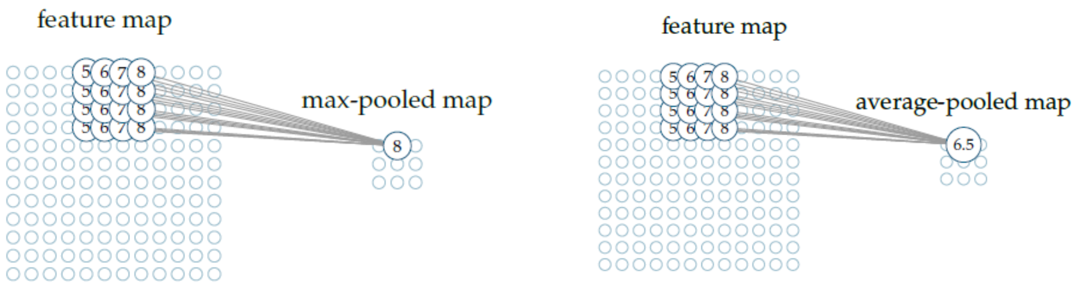


(a) Global connectivity (b) Local connectivity

Figure 3.8: Global (a) and local (b) connectivity for a hidden neuron over a 16×16 input. Each connection (in gray) learns a weight. Figure obtained from ”3D Object Recognition with Convolutional Neural Networks” [3] with the permission of the author.

maximum value present in the input. Another common type of Pooling is the Average Pooling, where the resulting value is the average value of all the inputs. Figure 3.9 features a visual representation of two different pooling layers, one with a Max Pooling policy, and one with an Average Pooling policy.

The concept behind the pooling layers is to prioritize the features considered more important, thus reducing the number of inputs and, therefore, the computational cost of the network. This collides with the main purpose of the CNNs, that was to take advantage of the spatial information in the input, because in fact we are reducing the spatial information obtained from it, but in the end this works properly due to the fact that is more important the relative position of one input with the others than the exact position of it.



(a) Max Pooling (b) Average Pooling

Figure 3.9: Max Pooling and Average Pooling operations over a 12×12 feature map using a filter size of 4×4 and a stride of 4×4. Figure obtained from ”3D Object Recognition with Convolutional Neural Networks” [3] with the permission of the author.

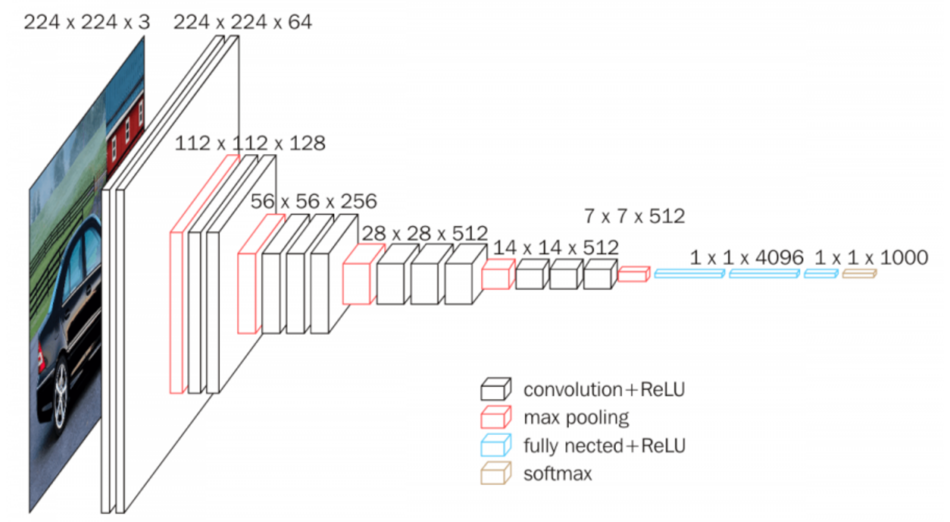


Figure 3.10: Structure of the VGG16 Network, a convolutional neural network trained for an image classification task, which was presented in 2014 with state-of-the-art results [4]. Image obtained from *Neurohive.io*

Knowing the types of layers a CNN usually presents, we can look with more detail into the typical structure of a CNN. In Figure 3.10 we can see the internal architecture of the VGG16 Network [4], a CNN developed for the purpose of image classification, which obtained state-of-the-art results by combining several layers of convolutional and pooling layers with dense layers at the end.

The input of the network is a squared image of 224 pixels each side and, since it is a color image, depth 3, one for each color in the RGB model. After that, we can observe how it is possible to find convolutional layers combined with pooling layers, increasing the number of convolutional layers between each pooling layer as we go deeper into the network. The depth of the convolutional layers is also increased exponentialy. This, as is explained in the relationed paper, is because in the first convolutional layers the features detected are low level features, such as basic forms, shapes and certain colors, and as we go deeper, the layers begin to use those low level shapes to detect more complex ones composed of combinations of previous features, such as the forms that make up a car.

Moreover, the network has a couple of fully connected layers for the classification of the features obtained, with a final layer being a softmax that classifys the image among all the possible categories. This lets us differentiate between the convolutional part of the network, used for obtaining information of the input, and the fully connected part, which purpose is only the classification of that input. Thanks to that, it would be possible to reutilize the network, replacing only the last part of it with another classifier for a different image classification task. This technique is known as Transfer Learning, and is very used in the Deep Learning field.

ANNs and CNNs are both known as Feedforward Networks because of its structure, but besides it there is another type of neural networks important to remark inside the Deep Learning field. Until now, it was possible to see that every neural network takes a single input and produces an output, withouth taking into consideration more information about it, such as previous information or predictions done by the network. This contrasts deeply with how humans think, because we do not reevaluate in every second all of our surroundings, because we have knowledge and memory of it which provides us more context. In Deep Learning, for example, when classifying images a CNN may be very good, but when those images are frames of a video, it may fail to identify properly the scene the image depicts because it does not obtain any input from the previous frames of the video. To solve this issue, we can use Recurrent Neural Networks (RNN).

RNNs are a type of Neural Networks usually applied to time series data, where the output can depend on several previous inputs instead of just the most recent one [35]. Therefore, during the feedforward process of the network, its hidden state is saved just to be fed back into the network at the following iteration. Thus, RNNs, instead of having a single input and output, as is common in Feedforward Networks, receive two different elements: the network input, as in a Feedforward Network, and the network's previous hidden state. It also produces one extra output, the new hidden state of the network, ready to be fed back.

This process allows the network to be context-aware, because the hidden state of the network acts as its memory, incorporating the previous inputs and outputs to the predictions of the network. Returning to the previous example, with a RNN it would be possible to obtain knowledge of a scene in a movie, and the network could predict and understand more accurately each frame by obtaining the context, like a human does. In Figure 3.11 it is possible to see the architecture of a simple RNN. At the left we can see a single RNN node in the wrapped visualization, which showcases the network architecture only. At the left, we can see that same network in an unwrapped visualization, where the inputs and outputs are depicted through time. We can also observe that, on the first iteration, the network does not have any previous hidden state to use. Tipically, to solve this issue, a random hidden state is generated.

Recurrent layers are not exclusive of Recurrent Neural Networks; they can be combined with other types of layers, such as convolutionals, to combine the advantages of different types of networks. This has been specially used in the videogame industry, where is common to have networks with convolutional layers first, that then are combined with recurrent layers, to predict accurately the images being depicted.

Traditional RNNs present a problem when they are trained for long periods of time, the Vanishing Gradients Problem [36]. The Vanishing Gradient Problem, which can happen in all kinds of Networks, but affects differently on RNNs, its the difficulty of training a Network due to the gradients being vanishingly small. This happens in Recurrent Neural Networks because, as we saw in the architecture of Figure 3.11, the hidden state keeps being acumulated each time, and therefore every instance since the very beginning is being recorded, even when its not influencing the output anymore, making it hard for the Network to understand what is affecting properly the current predictions of the network and what not.

To avoid these long term dependencies and solve the Vanishing Gradients Problem we can use Long Short-Term Memory (LSTM) [37] cells, a special type of recurrent node, developed to manage which inputs should the network forget and which should keep recording. To achieve this, LSTM cells divide the hidden state traditional RNNs receive into two different states, a long-term memory and a short-term memory. Following with the previous example of a movie, the short-term memory would be responsible of keepin track of the most immediate information, like the current scene, and the long-term memory would remember the overall plot and characters of the movie. Thanks to this structure, it is possible to maintain very long dependencies in the hidden state of the network, which is helpful for better inference, and still be able to learn from the most recent events, avoiding the Vanishing Gradient Problem.

The internal structure of an LSTM Cell can be seen in Figure 3.12. To be able to maintain both types of memory properly, and also make accurate predictions as is expected from a Neural Network, LSTM cells feature four different elements, known as Gates, which are combined inside every LSTM cell and interact with both memories and the current input. Those four Gates are: a Forget Gate, a Learn Gate, a Remember Gate and a Use Gate.

The Forget Gate is responsible for updating the long term memory, deciding which information keep for subsequently iterations and which not. It works by multiplying the current long term memory with a forget factor, which is obtained from applying a σ function to the combination of the current input and short term memory. In Figure 3.12, it is represented with the left σ and product.

The Learn Gate is responsible for joining the current input and the last short term memory, filtering them in the process. Joining both inputs is done by passing them through a tanh function, and the filtering them is done, in a same way as we saw in the Forget Gate, by multipliying the output of that tanh function with a forget factor, obtained from the sigmoid of the same input and short term memory. In Figure 3.12,

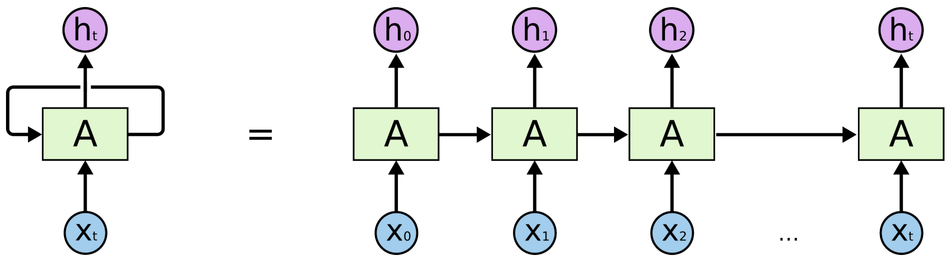


Figure 3.11: Architecture of a RNN. At the left, we can see the wrapped structure, where each outputs serves as the input of the next iteration. At the right, we can see the same structure unwrapped, over several activations. Image obtained from `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

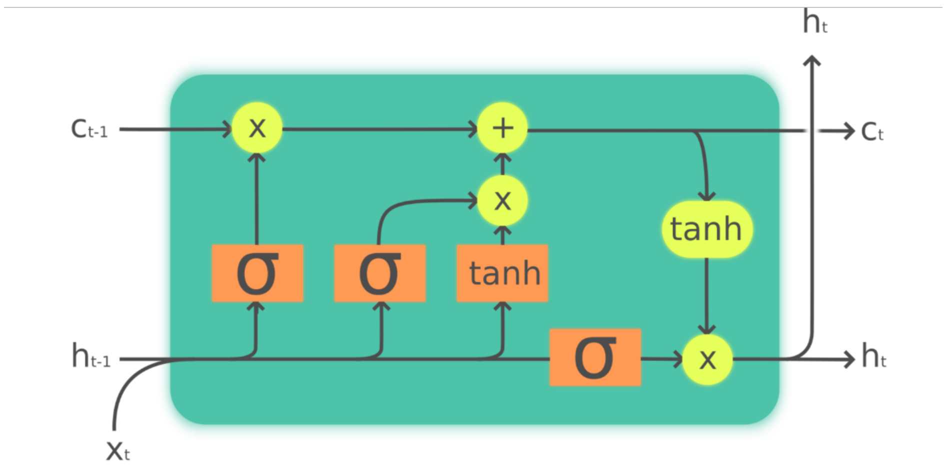


Figure 3.12: Architecture of a LSTM cell. It receives the input of the network, X_t, the previous short term memory, h_{t-1} and the previous long term memory, C_{t-1} and produces the cell output, h_t, the new short-term memory, h_t, and the new long-term memory, C_t. It contains a Forget Gate, composed of the first σ and the following product; a Learn Gate, composed of the central σ, a tanh and a product; a Remember Gate, the central addition symbol and finally the Use Gate, composed of the last σ, the last tanh and the final product. Image obtained from Wikipedia.org under the Creative Commons License.

it is represented with the central σ, tanh and products.

The Remember Gate is responsible for generating the new long term memory of the cell. It is done by combining both the Forget Gate and the Learn Gate, which is done with a summation of the outputs of those two Gates. In Figure 3.12, it is represented by the central summation on top of the Learn Gate.

Finally, the Use Gate is responsible for obtaining the output of that iteration, which is equal to the new short term memory of the network. It is done by combining the output of the Forget Gate, which is passed through a tanh function, and the output of the learn gate, which is passed through a sigmoid function, with a product. In Figure 3.12, it is represented by the right σ, tanh and product.

Long Short-Term Memory have been proved to be one of the best working techniques when dealing with time series data, and they have been used extensively both outside and inside the deep learning applied to videogames. Although its not strange to find solutions with networks that only have LSTM cells, they have been probed to work with utmost performance when combined with other elements, such as convolutional layers or even reinforcement learning techniques. A current state of the art example of this can be found on AlphaStar [38], the first AI developed with Deep Learning capable of consistely beating profesional players at the videogame Starcraft II.

3.4 Deep Learning approaches inside the videogame industry

In recent years, Deep Learning has suffered a massive increase in the number of researches and applications in many different fields. In the videogame industry, there has been a big increase in the number of publications [39]. Even when most of those have been using videogames as a research platform and it has not been integrated as part of the development process of a game, we consider the projects presented here are important enough to remark them. Also, we consider that they set the basis for the potential usage in the future of deep learning with development purposes.

One of the reasons of the growth of deep learning works related with videogames is the release in 2013 of the Arcade Learning Environment (ALE) [40]. The Arcade Learning Environment (ALE) is a public software framework designed for the development of AI agents for Atari 2600 games. At that time, this was considered specially useful because it allowed deep learning developers to create test environments for their agents with a well-defined and quantifiable measure for success.

The ALE, which is currently released and maintained by the community through a public repository hosted on GitHub, is powered by the Stella Atari Simulator [41]. It provides an interface for testing on hundreds of Atari 2600 games, with many games becoming study cases of development in the field. The library has been supported and heavily extended by the community [42], being very active in their development.

One of the works with more repercussion in the field, and that served as the cornerstone for the later development of more complex Deep Reinforcement Learning solutions was the paper published ”Playing Atari with Deep Reinforcement Learning” [5]. In this article, a deep reinforcement learning algorithm capable of playing different Atari games was proposed, beating expert human players in three of the seven games proposed, and outperforming DRL algorithms that then were considered the best.

Furthermore, a very important detail that diferentiates the algorithm proposed in that paper (DQN) to the others until then is the fact that the agent playing is only receiving the raw images (in pixels) of the games, and not any additional information, nor any prepared one. Also, the same algorithm without further modifications is the one that achieved those results in all the different games used for testing.

This project showed the reasons behind the great use of videogames as a research platform for Deep Learning. Apart from the reasons explained in previous sections that justify the development of AIs in game-like scenarios, the concrete advantages of videogames here are: In all of those cases, the chosen games represent a scenario with a limited set of actions for the agent, a limited set of rules to learn, and a clear score, which the algorithm can use as a scoring function for optimization. They also serve for benchmark and reference purposes, since it facilitates the process of comparison between different solutions, as it was seen in the original paper.

Moreover, although the videogames used in that particular work can be considered very simple, the wide range of games, specially modern videogames, allow for much

(a) Breakout

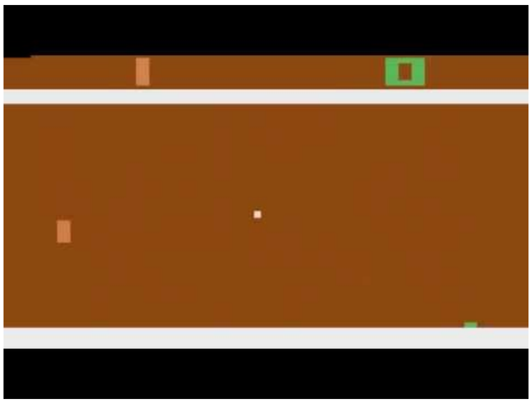

(b) Pong

Figure 3.13: Two of the games used in ”Playing Atari with Deep Reinforcement Learning” [5] as a testing environment for the algorithm presented. In those two games, along with the game Enduro, the deep learning agent obtained better results than an expert human with deep knowledge of the game. Images obtained from *Wikipedia.org* and *YouTube.com*.

richer environments in terms of complexity, surpassing in those terms traditional board games like chess or go. As a proof of the big impact the ALE and this original work supposed, we can find many important works in the same field that used Atari 2600 games as their benchmark.

In fact, in the following years, newer platforms that offer game scenarios, mostly based on old videogames, for experimentation with DRL have appeared. Featured examples of this are the OpenAI Gym [43] and OpenAI Gym Retro [44]. Specially this last example, developed by the OpenAI Team, extended the support of classic videogames to a wide range of emulators, allowing developers and researchers to not only use Atari 2600 games, but games from popular Nintendo’s consoles such as NES, SNES or Game boy, in addition to a huge number of other videogame consoles.

In parallel, other platforms for deep learning experimentation have appeared, focused on more modern games with huge complexity instead of traditional arcade games. Good examples of this are the Starcraft II Learning Environment [45], a general platform for the development of AI with Deep Learning techniques in Starcraft II, a modern Real Time Strategy (RTS) game published on 2010; and VizDoom [46], a general DRL platform for research based on Doom, a classic and very popular shooter published on 1993.

As the usage of the ALE became more and more generalized, it is possible to find either examples of new DRL algorithms [47] [48] [49] [50] or newer training methodologies [51] [52] [53] [54] that encompassed a big advantecement for the field of Deep Reinforcement Learning. This supposed that the number of games used for testing was extended, not limiting them to only the original seven videogames proposed, but to a much wider range, covering also ones with harder mechanics or different objectives, not as easy to understand for machines.

But without a doubt, the most important developments that followed, although they were in game applications, were not in the videogames field. We are talking about the creation of AlphaGo [55] first and AlphaZero [56] later.

AlphaGo is a Deep Learning intelligence, created combining Deep Reinforcement Learning techniques with traditional Deep Learning approaches. AlphaGo is the first computer program to defeat both a professional human Go player first, in October 2015, and a Go world champion later, in March 2016, therefore making it the strongest Go player in history. In the process of winning the matches, AlphaGo made highly inventive winning moves [57], that overturned hundreds of years of received wisdom.

AlphaGo was later improved with AlphaGo Zero [58]. AlphaGo Zero is a new version of AlphaGo trained only by playing games against himself, in opposition with AlphaGo, which was trained on thousands of human amateur and professional games. This new version surpassed the level of AlphaGo in only 3 days of training, and achieved a level never seen in the game before in just 40 days, showcasing the improvements and advantages of this type of training.

On the other hand, AlphaZero is a deep learning intelligence, powered by DRL methods, that is capable of beating computer programs of three different board games. AlphaZero, which was trained with the same method of playing against himself, defeated Stockfish in Chess, Elmo in Shogi and previous versions of AlphaGo in Go. All of the computer programs which it played against were considered at the moment of the matches the best AI for each one, therefore surpassing the best behaving references in each of the games. AlphaZero uses DRL methods to pick the best movement, in combination with a method known as Monte-Carlo Tree Search [59].

Returning to videogame applications, after the improvements in traditional board games we just presented, researchers looked for new challenges in modern videogames, that offered more complex aproaches in games with more space for decision space. For that reason, the Google DeepMind team, responsible of the development of both AlphaGo and AlphaZero, developed AlphaStar [60], a Deep Learning agent capable of defeating professional players in the game Starcraft II.

Starcraft II is a computer game encompassed in the genre of strategy, more concretely in the Real Time Strategy (RTS) strategy, which means that all the players in the game make decissions in a continuous time set, without turns, in opposition to what is seen in Chess or Go. Starcraft II is set in a fictionary world, where three different races fight between them: Terran, Protoss and Zerg. A tipical game of Starcraft II is usually a 1 versus 1 match, where each player controls a single race.

Starcraft II has a much wider range of possible actions and decissions in comparation with the previous board games we examined. In Starcraft II, a player can build a range of different buildings, usually between 10 or 15 different, depending of the race, and also recruit different units of a large number from which to choose. The game is not only focused on the military tactica of controlling those units, but the players also have to obtain resources from the environment and use them for building and recruiting.

Moreover, the units have the possibility of being merged on bigger groups, and usually most of them have spetial habilities that need to be activated on a certain moment. Additionally, the game has an exploratory component, since at the start of each game, none of the players know the position of the others and have to look for them in the map. Finally, each of the three races have additional mechanics that add more complexity to the game. For example, the zergs have a certain ”area of influence” where their units are better and they can build structures, that they need to manage and expand through the map.

For all of those factors, Starcraft II has become one of the most succesful competitive games in the world, with players from all around the globe contesting in big tournaments and living professionally from the game. It is considered one of the most complex and complete strategy games, and players have developed established strategies for it, sometimes with the purpose of countering a certain advantage of a different strategy or race. For that reason, the irruption of AlphaStar, which was announced to have defeated two professional Starcraft II players in a series of matches, was that relevant. To avoid complexities, AlphaStar was initially focused only in one of the races, Protoss, and only was faced against other Protoss players. AlphaStar initially faced and defeated TLO, a professional Zerg player who played Protoss for the match, therefore having a handicap. After the defeat of TLO, AlphaStar was matched against MaNa, a professional Protoss player, considered one of the best in the world with that race. In that match, AlphaStar won although MaNa was capable of winning a game because he finally adapted to AlphaStar's flaws, showcasing that it can still be improved.

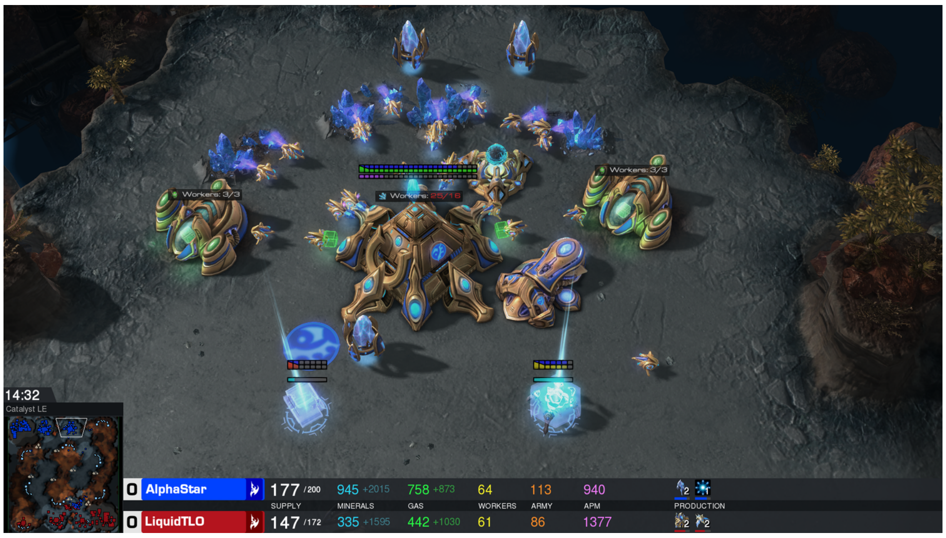

14:32
Catalyst LE
0 AlphaStar 177 /200 945 +2015 758 +873 64 113 940
SUPPLY MINERALS GAS WORKERS ARMY APM PRODUCTION
0 LiquidTLO 147 /172 335 +1595 442 +1030 61 86 1377

Figure 3.14: Screenshot of the game Starcraft II. It shows the match between AlphaStar and TLO, a professional Starcraft II player of Team Liquid. Image obtained from `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`

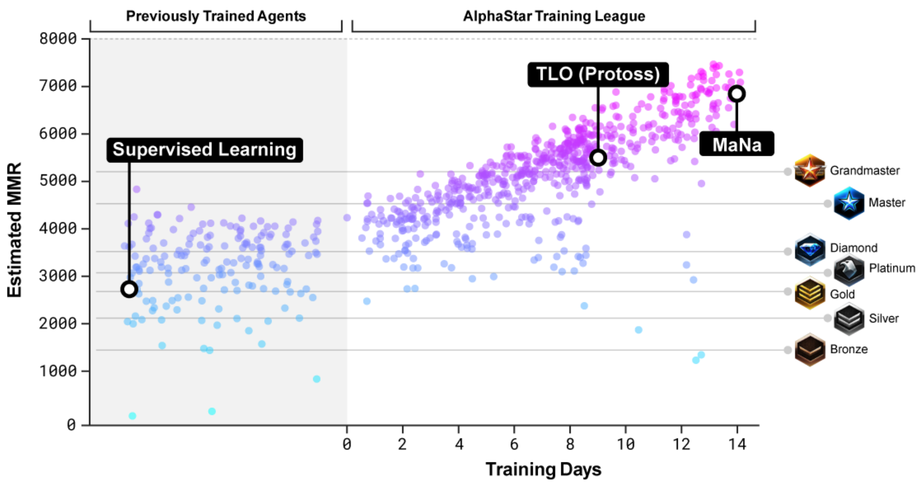


Figure 3.15: Estimate of the MMR of the different AlphaStar agents in the AlphaStar League. The levels at the right of the graph are the skill levels of the human online ranking of the original game. Image obtained from `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`

AlphaStar training process was carried by an internal league of AIs developed with Deep Learning in the DeepMind team. This league helped to push forward the development of better agents due to the competitive aspect of the league, and finally achieved the state-of-the-art results of AlphaStar. In Figure 3.15 it is possible to see the estimated Match Making Rating (MMR) of the different agents developed through the league, including those who faced both TLO and MaNa, compared with the human ranks of the Starcraft II online ladder.

The architecture of AlphaStar was mainly composed of a system similar to relational deep reinforcement learning [61], combined with a deep LSTM core, among other techniques. The first AlphaStar agents were trained by supervised learning from anonymised human games, allowing it to learn the basic macro and micro strategies used by players on the online game mode of Starcraft II. But, as it is possible to see in the last figure, the training process later shifted from supervised learning to unsupervised, learning by playing against himself.

In parallel with the development of AlphaStar, another big project relationed with gaming, specially with competitive gaming, appeared. We are talking about OpenAI Five [15], an Artificial Intelligence developed with DRL techniques to play Dota 2. Dota 2 is a an online videogame which primarily consists of a five versus five game mode, where each player picks a champion (of a map pool containing more than a hundred) with a different set of habilities, stats and strengths and they all try to defeat the enemy team by destroying a structure localed at the opposite side of the map.

The game showcases very deep complexity. Not only each champion has its own

Figure 3.16: Screenshot of the match between OpenAI and OG, current world champions of Dota 2. The image corresponds to the first game of the two played, and depicts OpenAI destroying OG's base.

special mechanics, that require several games to master it properly, but the game also has items to purchase, monsters from which obtain gold (to purchase items) and enemy team's structures to destroy in order to progress in the game and finally destroy enemy base. Furthermore, for the development of AI agents, games like Dota 2 are specially important because players need to collaborate, cooperate and communicate between them in order to success, since many of the current established tactics in the competitive scene require high coordination and teamplay among all team members.

OpenAI Five has made many advancements in this area since its first appearance in August 2017, where it defeated a professional player in a 1 versus 1 match, as a test of his individual skill with a champion. Later, in August 2018, it was matched against a team of five professional players, where he lost in a best of three maps match 2-0. However, since then, OpenAI Five has showed great improvements, defeating since then the teams Lithium, SG e-sports, Alliance and finally OG, the current world champions of Dota 2, therefore making it the first game AI to defeat world champions in a competitive videogame. All of the matches were played at best of three maps, where OpenAI Five won with a score of 2-0, although its important to remark that OpenAI was trained with some limited set of rules: for example, OpenAI was only trained to choose among 18 champions of the total champion pool to select.

Each one of the OpenAI Five networks, responsible of the control of one of the bots, contains a single-layer 1024-unit LSTM layer, connected to the game state through the Valve's (company responsible of the development of Dota 2) Bot API, although most of the decissions work with a DRL approach.

Since the outcome of that match, an online platform for playing againts OpenAI, and therefore participate in the training process of the network, was released with the name of OpenAI Five Arena [62]. OpenAI Arena is considered the largest deployed Deep Learning model, and allowed users to play five versus five matches against five controlled OpenAI bots, and also to play in teams in cooperation with OpenAI Five bots, mixing human and computer controlled characters in a team. In the three days OpenAI Arena was open for public access, many human teams were capable of defeating it, specially since certain flaws in their game were found, showcasing that there is still room for improvent in the game.

Although the current tendency in the applications of Deep Learning in videogames is the search for modern videogames, with high complexity and usually a competitive scene where to be faced against humans, there is a classic videogame, featured in the ALE, that is still being studied and new solutions are being propossed: Montezuma's Revenge.

Montezuma's Revenge is a classic videogame for the Atari 2600, where the player obtains rewards as he explores and completes a level, composed of a varied number of screens. Among the number of reasons behind the difficulty it supposes for a Deep Reinforcement Learning approach, we can point out specially the sparse rewards it gets, since they are usually obtained at the completion of the level, and the difficulty it supposes for a Deep Learning intelligence to explore by itself without a clear purpose or objective from where to obtain rewards. In fact, if during the exploring process an inmediate reward is obtained and it can be exploited subsequently, usually the exploring ends and the agent starts to only obtain those small rewards, instead of the bigger ones obtained from completing the level.

The first level of Montezuma's Revenge was finally completed by the OpenAI team [63] using Proximal Policy Optimization, the same Deep Reinforcement Learning algorithm behind OpenAI Five. The algorithm was trained with the help of a human demonstration, done by an expert human player. The training process consisted in the agent learning from certain states of the game, carefully chosen by the development team, until a certain behavior was learned, and the moved backwards from state to state until the beginning of the game. Therefore, the agent learned how to end the level first, and how to complete it stage a stage backwards until the beginning.

This approach has been criticized [64] as not being a real solution, since the agent has not learned how to explore the game succesfully, but instead how to exploit the deterministic approach of the game and just do a certain behavior in a particular game state. For that reason, in 2019 the team behind the development of the ML-Agents Toolkit for Unity (more information in Subsection 2.2.3.2) released an environment for the Toolkit called Obstacle Tower Challenge [65]. The Obstacle Tower Challenge is a videogame for the Unity Engine where the player has the objective of completing floors, each one with an increasing difficulty and complexity, inspired by the challenge Montezuma's Revenge supposes.

The Obstacle Tower environment [66] consists up to 100 floors, with the agent always starting at floor 0. Each floor contains atleast a starting room where the agent is started and a room with stairs to the next floor. Optionally, floors can contain puzzles to solve, enemies to defeat, obstacles to evade or locked doors to be opened with a key. The layout and contents of each floor becomes more complex as the game progresses. Also, floors are generated procedurally, following a certain number of guidelines, with both a mission and a layour graph being generated according to certain parameters. The visuals of the floor can change too from a number of pregenerated visual styles. Examples of three generated floors, corresponding with early, intermediate and later floors can be seen in Figure 3.17. The reasoning behind the floors being procedurally generated is to avoid the determinism present in Montezuma's Revenge, forcing agents to solve it by trully exploring the environments instead of learning a predefined way of reaching the end of the level.

The action space of the agent is partially limited. It mostly consists of the movement, which can be onwards, backwards, left, right or none; the rotation of the camera, in left or right directions, and the possibility of jumping. Because of this, agents trained in Obstacle Tower will not have to learn highly complex playing mechanics, but instead focus on the exploration and task-solving needed to progress.

To sum up, we have explored a large number of scenarios where videogames are used as a tool for training agents in complex scenarios, usually with the purpose of later migrating that learned behavior into another fields of the real world. But as it was mentioned in Section 3.2, some learning intelligences have been used for the

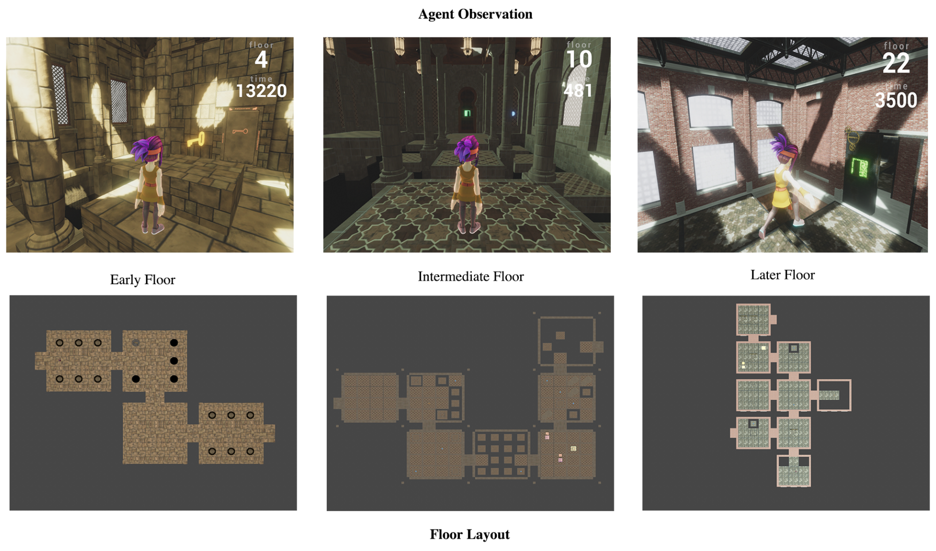


Figure 3.17: Example of three different levels, or floors, in the Obstacle Tower Challenge. The above images show the agent observation, used for the training, and also what an human agent would see if he played the game, while the images below show the floor layouts, of increasing complexity. Image obtained from `https://blogs.unity3d.com/es/2019/01/28/obstacle-tower-challenge-test-the-limits-of-intelligence-systems/`

Figure 3.18: Player's pet in the game *Black & White 2*. Image obtained from `https://www.gamefront.com/games/black-and-white-2`

development of game AI too.

The two most famous cases are *Creatures* and the *Black & White* series. While the game *Creatures* focused more on emergent behaviors of their agents, the *Black & White* games, specially the second installment of the series, used the possibility of teaching their agents as a game mechanic that was very appraised at the time of the game's release. In *Black & White*, which can be considered a strategy game, similar to an RTS but more focused on the management of a town, the user played as a god responsible for the welfare of the population, with the choice of being a good ruler or an evil one. Among the many possibilities the player had, he could have a pet, usually a giant mythological one. That pet could be teached to treat the town's population properly, defending them against external threats, or instead to terrorize the population.

In fact, the game also featured a non-deterministic learning feature for the pets, where the player could perform divine powers in front of them and the pets could learn those powers to be used later in combat scenarios, such as wars. As said before, the learning process was non-deterministic, and tried to emulate the different difficulties of teaching an animal, varying the number of uses needed depending on the pet chosen and the game state that was being played at the moment.

To conclude, in spite of seeing many numerous examples of Deep Learning usage relationed with videogames, the most recent examples of similar uses in game AI for game development applicatons are dated at the beginning of the century, without much development since then. In conclusion, we consider that although the Artificial Intelligence field and the videogames are deeply connected, the advancements carried out by those connections have not been taken by game developers for their industry.

4 Implementation and Experiments

In this chapter, we describe the different approaches carried for the development of a deep learning intelligence applied to a videogame, and we compare and discuss the performance and flaws of all of them. The work developed is introduced in Section 4.1. The initial analysis of the development status is presented in Section 4.2. The creation of a development pipeline for creating and integrating Deep Learning models into HardBall is explained in Section 4.3. Finally, in Section 4.4 the training process of three different Deep Learning approaches is explained.

4.1 Introduction

In the following sections, we describe the development of a deep learning pipeline for the training and integration of deep learning bots in HardBall. We also propose three different approaches for the development of artificial intelligent agents for the videogame in its development stage. We discuss the advantages and disadvantages of every method, and we propose guidelines for their usage.

Due to the collaborative nature of this work, which was carried out in cooperation with From The Bench Games (FTB), the initial state and objectives of this project are framed in the needs of the company and the situation of the development process of HardBall. For that reason, first an initial analysis of the current Artificial Intelligence being used in HardBall, and the Deep Learning solutions carried by the company is carried out.

Then, because of the needs extracted from that analysis and the objectives set by the company, a development pipeline is created, to serve as the cornerstone of the experimentation with different Deep Learning models. This involves various processes ranging from the adaptation of the current code on Python for exporting the models into the required format, to the creation of code for the use of those models in the HardBall project.

Finally, three different approaches are developed and compared, testing the performance of the deep learning models from different perspectives. Those approaches are: A CNN that receives a grid image created explicitly for the process, an ANN which receives a vector with the data collected by FTB during the game closed beta, and an RNN network that uses a LSTM core with that same data vector as input. The results of those three neural network approaches are compared and conclusions are drawn from this in Chapter 5.

4.2 Analysis of the current state of HardBall

When this project started, we were approached by From The Bench Games to collaborate in the creation of Deep Learning Bots for one of their games in the development stage, HardBall (more details about HardBall can be consulted in Section 2.2.2.1). The basis of this collaboration was first to integrate the models already developed by them, and then work on the development of new models based on the needs of the project.

It is important to remark that HardBall has an already implemented system for the AI of their NPCs, but due to the simplicity and performance (in terms of decision-making) of that approach, they looked for alternative solutions in the field of Deep Learning. The system used beforehand was a Decision Tree in common for all types of agents in the game, allied or enemies.

The Decision Tree, in general terms, consisted of checking in which half of the field it was the agent located. If he, and the ball, were both in the enemy half of the field, the agent just moved to the ball with the intention of shooting, therefore ignoring the presence of allies or enemies. If the bot was located in its half of the field, it checked which was closer: the ball or the goal line, and then just moved to it accordingly.

In order to properly integrate the work already developed by the company, first we proceeded with an analysis of the current approaches and the advantages and disadvantages they had. We observed that, initially, the purpose was to create a single Neural Network capable of working either for the enemy players or for the companion in HardBall's two versus two game mode.

The current approach was the development of different CNNs for each one of the different actions a player could make. For that reason, different models, but with the same structure, were being developed for the movement and the shooting of the player. The models were trained with a dataset, comprised of approximately 10.000 entries, which was a subset of all the games played during the closed beta, which all were recorded into a database for the purpose of training Deep Learning agents. The exact structure of a data entry in HardBall's closed beta database can be consulted in Table 4.1.

While the recorded information was the data obtained from the game, the models being developed were CNNs, due to the proven results [5] [67] in applying them for videogame applications. Therefore, a visual representation of the collected data was created, using the game soccer field size and position in Unity as a reference.

Those images, which were intended to serve to the development of all different models, were discrete mappings in a grayscale image of all the positions recorded in the data structure, but without any information about goals, shots or the forward vector of the player. The images were constructed using a Python library MatPlotLib, and they were generated each time a training session was launched. A visual example of the images used can be seen in Figure 4.1, which depicts a frame from the beginning of a two versus two game mode.

At the moment of the start of this project, three models were already trained by FTB. All models were trained using a technique similar to Imitation Learning, where the network received images as depicted above, and tried to predict a value, recorded in the data entry too, but not provided to the network.

Two models were trained to emulate the player's movement, being the only difference between them the target variable it had to predict. In one case, it had to predict two values, being those values the forward player direction in X and Y coordinates. In the other case, the network had to predict that same forward vector, but transformed to a degree system in a range from -1 to 1, where the range from 0 to 1 corresponded to a range from 0 to 360 degrees, and the -1 served as the value for expressing the absence of movement in that particular frame.

Finally, the last CNN trained beforehand was developed with the purpose of predicting if the player has shot in that time frame or not, using the boolean to compare with

Variable	Data type
Player Forward X	Float
Player Forward Y	Float
Shoot	Bool
Player Position X	Float
Player Position Y	Float
Ball Position X	Float
Ball Position Y	Float
Companion Position X	Float
Companion Position Y	Float
First Enemy Position X	Float
First Enemy Position Y	Float
Second Enemy Position X	Float
Second Enemy Position Y	Float
Goal	Bool

Table 4.1: Structure of a data entry, as recorded in HardBall's closed beta database. Players position's are recorded in Unity coordinate system, and the range of them vary in function of the field size and position. Player forward values are set in a range from -1 to 1, and ilustrate a coordinate system using the player as the reference point. It shows the current position where the player is moving, according to the data obtained from the player's joystick. Finally, both booleans only record the information of those events at the current frame, therefore if a player shoots the ball, only in that frame and not in the following ones the Shoot value is set to one. Obtained with the permission of *From The Bench Games S.L.*

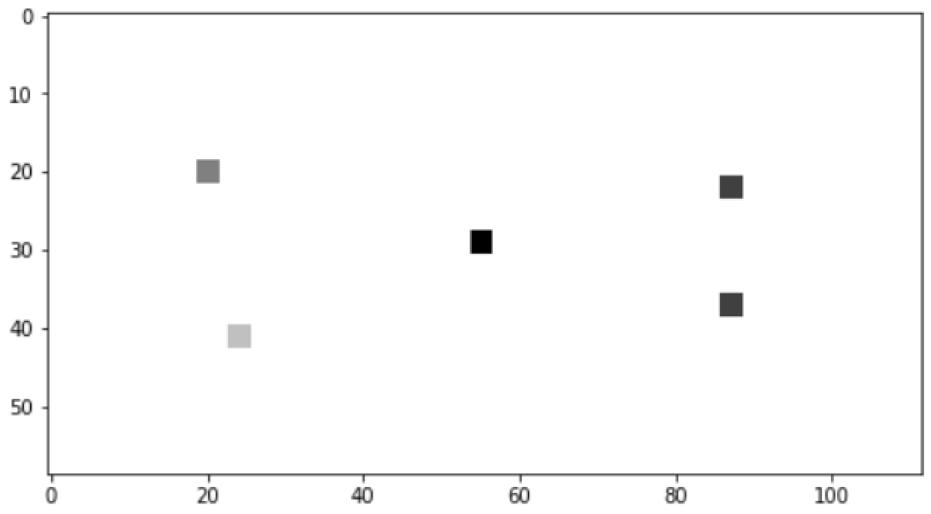


Figure 4.1: Grayscale image used for the training of CNNs in HardBall. It depicts the player in the bottom left of the image, and his companion above him. The ball is placed at the centre, and the two enemies, both with the same grayscale color, in the right of the image. The field has a size of 112 $\times$ 59. Obtained with the permission of *From The Bench Games S.L.*

the output of the network. To avoid adding more complexity to the created models, the possibility of using special powers was not taken into consideration when developing the models, and the frames which corresponded to the activation of powers were excluded from the dataset.

In the Table 4.2 we can observe a comparation between different data from the trained models. Its important to remark that the Shooting model is obtained through early-stopping [68], because the model's predictions when it was trained for a longer number of epochs tended to never predict a shoot, therefore obtaining better results on paper because of the low amount of shots in the dataset, but with a worse behavior for our purposes.

Model	Training Epochs	Size	Accuracy	Loss
Movement - Degrees	10	96	0.3548	0.0469
Movement - Forward Vector	40	96	0.8626	0.0827
Shooting	50	180	0.9725	-

Table 4.2: Comparison of the three models developed by FTB before the start of this project. The model size is expressed in MB, and uses the binary version of the models as the reference. Validation Accuracy and Validation Loss were recorded as obtained from Keras using MSE for validation. Validation Loss for the Shooting model was not recorded. Obtained with the permission of *From The Bench Games S.L.*

The models also present a number of flaws: since HardBall is a mobile phone game, more concretely for Android platforms, and which aims to work on all kind of devices, even low-end ones, the size of the models is unacceptable for deploying them on a production environment. In comparison with HardBall size, which was around 20 MB at the time of the start of this project, adding a model for the NPCs movement and other for their shooting behavior would increase their size until 300 MB of size aproximatedly.

But the biggest problem with those models was the lack of testing on the game itself. The models were developed in Python, using Keras [9] and Tensorflow [7] as their development tools, but the game was being developed in C#, using Unity as their development engine. Because of this, none of the models developed, either the versions presented in Table 4.2 or previous versions tested by FTB, were tested in the game, and therefore the performance of those models was unknown.

For that reason, and although the numerical data showed good results specially on the shooting model, it was decided that the development of a pipeline, capable of extracting Deep Learning models in the required formats and then adding them to the game for easily testing them was the priority.

4.3 Creation of a Deep Learning development pipeline

As we saw in the previous section, the most critical part at this point of the development was the creation of a Deep Learning development pipeline, capable of integrating in HardBall the models developed in Python. Initially, the pipeline required a tool for loading Deep Learning models natively in C#, and also a script to control that model from Unity natively.

Due to most of the work being necessary in C#, the first stage of this development was the research about existent solutions for applying Deep Learning to Unity. It was during this time that ML-Agents Tookit was found, whose most recent version at that time was the 0.5.0, which used TensorflowSharp as a background for loading and working with Deep Learning models. More information about TensorflowSharp and ML-Agents can be found on Sections 2.2.3.1 and 2.2.3.2 respectively.

At that time, research about the capabilites of ML-Agents for our needs was carried out, showing promising results. Although the library was mainly focused on the development and practical application of Deep Reinforcement Learning (DRL) algorithms in Unity, it also provided a working structure capable of integrating inside of Unity Deep Learning models, which with little work could be adapted to work with binary source files instead of the DRL trained with the library. It even had, among the working examples of the library with pretrained agents, a learning environment with a two versus two soccer game, with bots acting as either scorers or goalkeepers in a small playing field. In Figure 4.2 we can observe a screenshot of this game.

Due to the focus of the library on DRL and the desire to reutilize the already trained

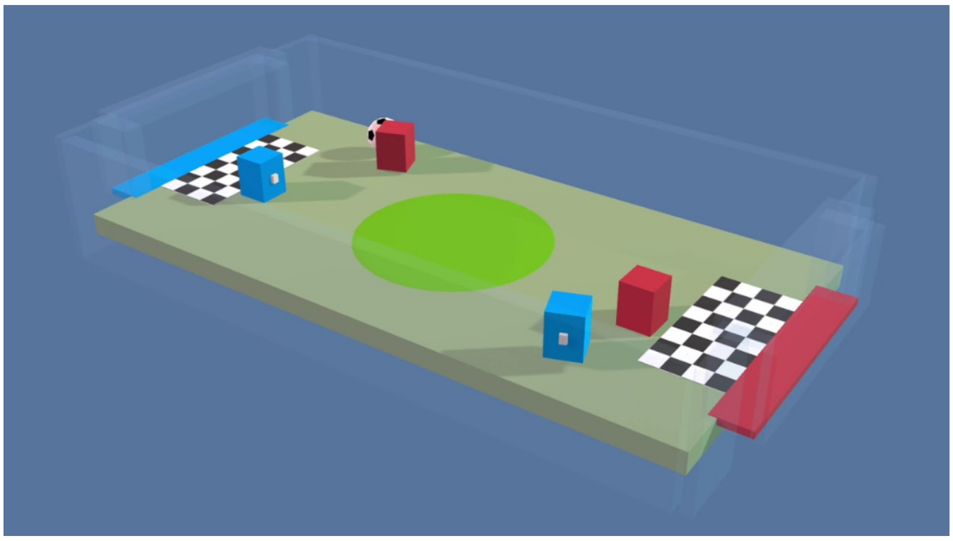


Figure 4.2: Screenshot of SoccerTwos, a learning environment provided by ML-Agents Toolkit. It features two teams of two players, from the red team and the blue team, trained each one to work with a different personality. One is trained to score goals, while the other is trained as a goalkeeper. The training of both agents has been done using DRL techniques.

models in Keras, the first step was to develop a model exporter in Python, because ML-Agents required a specific data format (a binary format with extension .bytes) for loading models that were not trained using the library beforehand.

A Python script called ModelExporter.py was developed, which used Tensorflow's freeze graph tool for exporting an already saved model in protobuf (.pb or .pbtxt, both are valid) format into a binary file, whose extension can be defined as bytes. The only restriction of the script was the need of defining previously the name of the node that acts as the output of the network. This is a common restriction, as the library itself in C# also needs to know this output node for working properly.

When the models were exported using this tool to the required format by ML-Agents, a problem appeared. Because of the experimental nature of the library, in open beta, it needed a newer version of the project to work than the version being used by FTB for the development of HardBall. The possibility of updating Unity's version was taken into consideration with the development team, and finally we decided to try to avoid updating the version, because it will add more risks to the integrity of the project.

For that reason, we decided to put aside the use of the library, and develop a script in C# capable of loading, integrating and managing our Deep Learning models itself. For the development of this script, we used TensorflowSharp as the basis of our development, and adapted the script behavior to the characteristics and needs of HardBall.

We created a script called CNN-Input.cs, since initially we were working with CNNs. His task was not only to load Deep Learning models from binary format, but also to manage and construct the input the Neural Network received from the game and saving the output of the network. For building an input, we recreated the process done previously in Python, were we took the width and height of the playing field, and the position of every agent and the ball, and discretized their positions. Then, with that information, we constructed a grayscale image, emulating the ones used as an input for the network, like the one presented in the Figure 4.1.

For loading and managing the neural network, two functions were defined. Since we only needed to load and save the neural network once, at the beginning of the execution, a function for that purpose was created, which we can consult in Listing 4.1. Although it is a very simple function, it should be noted that the code inside of that function is related to TensorflowSharp, since functionalities provided by the library, such as TFSession or TFGraph are being used.

Listing 4.1: Loading and saving of a Deep Learning model in C#

```
// Load the CNN model
void LoadModel()
{
    // Import the graph, given in .bytes format
    graph.Import(ModelGraph.bytes);

    // Create a session with the graph
    session = new TFSession(graph);
}
```

A function for giving input and obtaining output iteratively from the network was defined too. The code for this purpose is presented in the Listing 4.2. There, it is possible to observe that, as in the previous listing, most of the code is related to TensorflowSharp functions and data structures.

First, the runner of the session is obtained, in order to attach the correct input to it (which may vary on every iteration). Then, we define a TFTensor with an input we have already builded beforehand, because due to TensorflowSharp restrictions the input of the neural network must be in that format. Following this, the input and output nodes are defined, and the network runs, giving an output that finally is saved in a variable called TemporaryDirectionInGrades.

The name of that last variable is, as it may suggest, indicating that we save the output of the network in a temporal variable. Initialy, this was not the case, but due to performance reasons (which we will see in more detail in Section 4.4), support to multithreading was added to the script to enhance the performance of the model. This, due to Unity's own restrictions, required the use of an intermediate variable to avoid concurrency problems [69].

The integration with the rest of the code, more specifically the Decision Tree used in the project, was done with the objective of simplicity and ease of use in mind. The use of the Deep Learning script was managed from the same script that managed the use of the Decision Tree and, with the purpose of avoiding errors, was programmed to use the Decision Tree in every agent which did not had a Deep Learning model loaded.

Therefore, every NPC in the scene had the Deep Learning script, but only those who had a neural network loaded stopped using the Decision Tree for their decision-making.

Listing 4.2: Inference code for a loaded neural network in C#

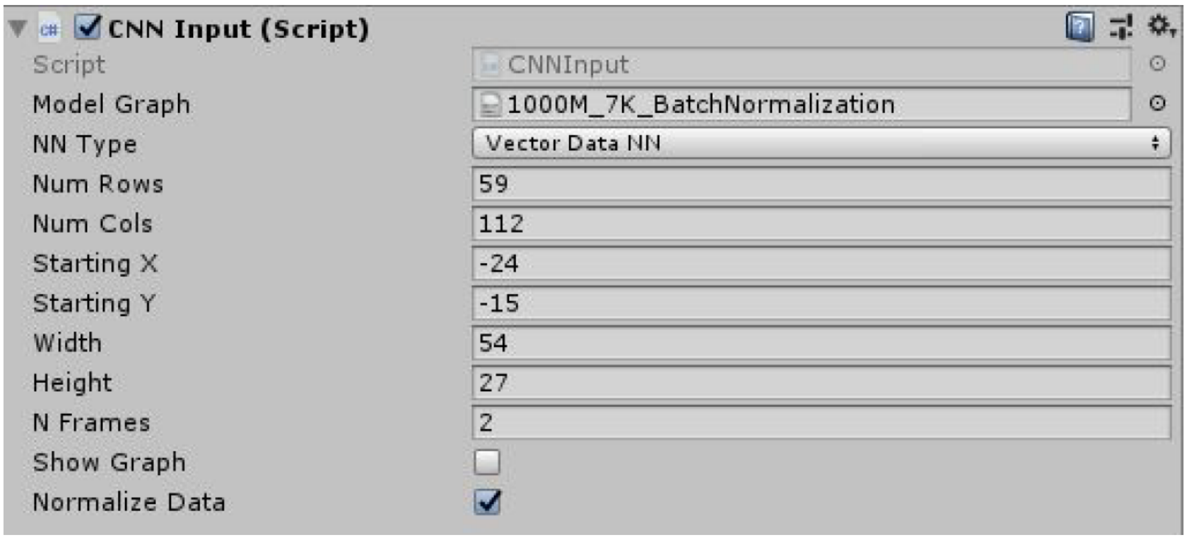
```
// Evaluate the model and infers a new input
public void Evaluate()
{
    // Output variable
    float[,] Output;

    // Obtain the runner of the session
    var runner = session.GetRunner();

    // Transform our Input array into a Tensorflow Tensor
    TFTensor Input = InputVector;

    // Add the Input tensor as the input of the NN
    runner.AddInput(graph["state"][0], InputVector);

    // Set up the output tensor
    runner.Fetch(graph["action/Relu"][0]);

    // Run the model
    Output = runner.Run()[0].GetValue() as float[,];

    // Save the output
    TemporaryDirectionInGrades = Output[0,0];
}
```

Thanks to this feature, it was possible to define different models for each one of the agents, because the model was loaded by the agent's script and not a manager in the scene. This choice was done in concordance with the FTB team, since we decided to focus on the development and testing of the player's companion behavior first, until a satisfying prototype was achieved.

As it may be noted, the code presented in Listings 4.1 and 4.2 works for a single neural network, but beforehand we had two different networks for the movement and one for the shooting. This was another decision taken in collaboration with the FTB team, due to the shooting model presenting big problems although the validation accuracy was high (more details about this model in Section 4.2). Also, due to the nature of the game, the movement is much more critical than the shooting, since it is possible to score by only moving the player. Because of this, we decided to focus on a single movement model, and after some initial analysis, we decided to use the model based on grades, because of the simplicity of use (one output against two) and the potential of improvement it was seen in it.

When the development of the script described here was finished, a basic pipeline for the development of Deep Learning models was implemented and tested. This allowed us to start using the models trained on Python in Unity and see how they behave in a production environment. It also allowed us to update in a matter of minutes a model in case we wanted to try a new one.

Even when the pipeline was working and considered finished, as new requirements



CNN Input (Script)

Script	CNNInput
Model Graph	1000M_7K_BatchNormalization
NN Type	Vector Data NN
Num Rows	59
Num Cols	112
Starting X	-24
Starting Y	-15
Width	54
Height	27
N Frames	2
Show Graph	☐
Normalize Data	✓

Figure 4.3: Screenshot of CNN-Input interface inside Unity. From top to bottom, it allows to configure the following aspects: Load a Neural Network model in binary format, select the type of neural network we are going go execute, setting the number of rows and columns of the input image's size in the case we are using an image as an input, set the starting position (X and Y coordinates), width and height of the field in case we are going to normalize the input, selecting the number of game iterations between each inference from the neural network, showing the graph loaded in Unity's terminal for debug purposes, and finally it allows to normalize the input data, in the case it is numerical and not an image.

arose, a number of changes and features where added to our system. First, as said before, support for multithreading was added to Unity's script for performance purposes (more details in Section 4.4).

Later, a new script for exporting the models was created, mainly for testing the exporting process and comparing the outcome with the one generated by the first script, but also for optimizing the neural network in the process. It was created using two different tensorflow tools: one for transforming the current graph, TransformGraph, which was used for removing the unused nodes and thus reducing the size of the neural networks; and finally a function for converting variables to constants, freezing the whole graph as a result, which then only needed to be saved in the format required.

Finally, the script CNN-Input.cs evolved widely, as more and more features and tests were required because of the development of different Deep Learning models. The reasons in detail for those changes are explained in Section 4.4. We can observe the final interface of the script in Figure 4.3, where a number of features added during the development can be seen. Although it is only a screenshot of the interface of the script, and therefore only the configurable options can be seen, it is a good summary of the features added during the development process.

During the development process, support for different neural network architectures was added. This mainly supposed the modification of the input given to the network, being able to pass either a grayscale image or a data vector with the values recorded from the game. Also, configuration options where added for those inputs. For the images, it was possible to configure the size of the field, and therefore the number of rows and columns that composed it, and for the data values it was possible to normalize all the data expressing the positions in a range from 0 to 1, taking as the reference point the top left corner of the field. Support for different neural network architectures also implied the modification of the input and output nodes, to adapt it to the network needs.

Aditional variables were added and supported for the data vector type of input, being them mostly relative positions between the agent and the other agents (the player, the enemies and the ball) in the field, but the support for that feature was later deprecated. Debug options where added to the script too, with the option to output the graph in the network's terminal for debug purposes, and also adding a testing phase between obtaining a new input and giving it to the network to avoid fatal errors. Finally, the possibility of choosing the number of frames between each inference of the neural network was added for performance reasons, looking to optimize the framerate in low-end mobile devices.

To sum up, a development pipeline for deep learning was elaborated almost from scratch, communicating two independent projects in different languages. Although most of the work was developed only for HardBall, the pipeline can be used for any other project with little to no modification needed, and it supposes a fast and lightweight solution for implementing deep learning solutions inside of Unity, which is a good alternative to ML-Agents Toolkit when the purpose of the work is not DRL, but instead more oriented to traditional Deep Learning techniques.

4.4 Training of Neural Networks

After the first version of the pipeline was integrated succesfully in the project, the focus shifted to the development of Deep Learning models capable of solving the needs raised by the From The Bench Games team. As explained before, it was decided that the focus of the development would shift to the movement model based on degrees. The shooting model presented problems in its predictions in spite of the good results showed by the validation process. Also the movement model based on degrees was much more easier to integrate in the game than its counterpart, due to the system already implemented.

4.4.1 Convolutional Neural Network

For that reason, the first model tested in the game was the CNN already developed by FTB before the start of this project. In Table 4.2 its possible to check the technical details of the network, in Figure 4.1 we can see the input the Network receives and in Listing 4.3 we can observe the structure of the network itself.

The structure of the neural network shows certain parallels with a typical structure of a neural network, such as the one seen in Figure 3.10, a couple of convolutional layers, followed by a pooling layer, a Flatten layer to prepare the input and a couple of Dense layers to finally make a prediction [4]. The network also has a number of Batch Normalization layers, with the purpose of obtaining better validation results.

Taking into consideration this structure and the data presented previously, we can observe two problems: The input of the network is, in a big part of the picture, redundant; but most importantly, the biggest problem this network presents is its size, far bigger than the size of the game itself. For those reasons, a test of the network was carried on in Unity, both from a behavior and a performance point of view.

Listing 4.3: Structure of the CNN developed by FTB prior to the start of the project. It is the model used for predicting the movement of the agent in degrees. It presents a typical CNN structure, with a couple of convolutional layers, followed by a Max Pooling layer and a couple of fully connected layers, with a Flatten layer for preparing the input previously. We can find Batch Normalizaton layers too between some of them [6]. It is programmed in Python, using Keras Sequential API. Code obtained with the permission of *From The Bench Games, S.L.*

```python
model = Sequential()
model.add(Conv2D(32, kernel_size=(2, 3),
activation='relu',
input_shape=input_shape,
name="state"))

model.add(Conv2D(64, (2, 3), activation='tanh'))
model.add(BatchNormalization())

model.add(MaxPooling2D(pool_size=(2, 3)))
model.add(BatchNormalization())

model.add(Flatten())

model.add(Dense(128, activation='tanh'))
model.add(BatchNormalization())

model.add(Dense(final_layers, activation='relu', name="action"))
```

From a behaviorally point of view, the agent, in this case the player's companion, begins the game at his starting position, and since then he moves to the right of the screen, running until the right end of the scoring zone and not stopping at the limit, therefore acting as if he was trying to escape from the playing field. The degree at which the player moves varies in a range from -5 to 5 degrees aproximatedly, depending on the input he is receiving, but he does not show any intention of moving to the ball or interacting with other players.

From a performance point of view, the result is insufficient and unacceptable. As the CNN's size may indicate, the neural network is too big and takes too many time, making it impossible to run on an environment where it should make several inferences per second. For that reason, two different solutions where developed. On one hand, support for multithreading on the game was added, and the use of the neural network was made independent in a secondary thread. On the other hand, a new CNN model was developed, highly simplifying the structure presented in the Listing 4.3. The resulting structure was composed of only a convolutional layer, a max pooling layer, a flatten layer and a dense layer, eliminating therefore the high number of parameters seen in certain steps, such as in the first Dense layer of the neural network.

In Table 4.3 we can see a comparison between the early solutions tried in the game. The first model presented, the Decision Tree, is the model the NPCs were already using in the game, and therefore, all the other models assume that the Decision Tree is still running for the enemies, and just changes for the player's companion. From the data obtained we can observe that the first CNN presented a performance insufficient for using it in the videogame.

That same version, but with multithreading support, had much more acceptable values, specially on the PCs, were the features were tested first. But when tested on mobile phones to meet that requirement, the framerrate dropped to values considered in the edge of what is acceptable for a game. Also, we should consider that the data in Table 4.3 shows the medium value. However, the version with Multithreading was much more unstable, with drops of framerrate to values around 10 frames per second. Finally, it also presented a big problem derived from the multithreading inclusion: a much bigger battery consumption. It consummed aproximatedly 1% of battery of a phone in 2 minutes or less, therefore dissabling this feature for production environments.

For these reasons, it was decided that the next step was the training of new CNNs following the simplified architecture, which solved the performance flaws, and focus then in solving the behaviourally problems it presented.

Varied CNN approximations were tested subsequently, with little to no improvement in the agents behaviors. New input configurations were tested, with varied sizes for the number of rows and columns on the board. Also at this time, support for the input's size configuration was added to the development pipeline in C#. The development of Convolutional Neural Networks presented a situation hard to balance: when the complexity of the neural network grew, the results (taking as a reference the validation loss and accuracy) improved, but the performance dropped.

Model	Frames per second
Decision Tree	60
Convolutional Neural Network	1
Convolutional Neural Network with Multithreading (PC)	25
Convolutional Neural Network with Multithreading (Mobile Phone)	18
Simplified Convolutional Neural Network	60

Table 4.3: Comparison in terms of performance between the different CNN models developed based on the initial FTB Neural Network. The values in the frames per second column represent the mean values, obtained through a period of time of 60 seconds. Obtained with the permission of *From The Bench Games S.L.*

Another reason for the bad behavior the model showed was the input the network received. Typically, CNNs use as an input the raw image of the game in pixels [5] [54], but in our case a grayscale image was constructed, which oversimplified and modified the information received by the network, by arbitrarily encoding the image with certain gray values, and making the sizes of the agents bigger than what they actually are.

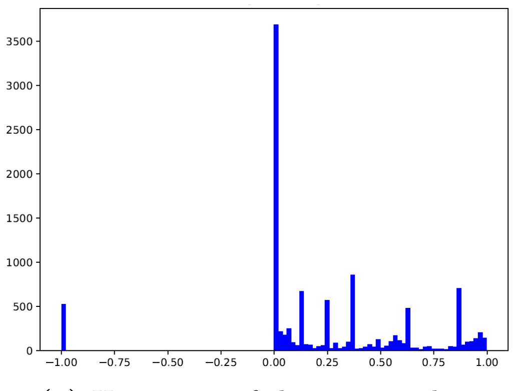
For those reasons, in collaboration with the development team we decided to stop using the CNN approach and move to a classic ANN, which uses a data vector constructed using the database entries (Table 4.1 for more information) as input data instead of images. The decision was made according to the data we had at the moment and the requirements presented by FTB, who preferred to avoid using Deep Reinforcement Learning before a solution with classic neural networks was achieved.

4.4.2 Artificial Neural Network

The development of the first ANN was fast and it was integrated in Unity without much effort, rehusing the structure created for the CNN without multithreading, and only needing to change the input to obtain the values of the game instead of constructing an image from those values.

However, the first Artificial Neural Networks approaches presented the same problem as the CNN in terms of behavior. The agent only moved forward, in a range between 5 and -5 degrees, even when the accuracy improved or worsened.

Looking at the data obtained from the training, we found that this was caused by all the frames in which the player is not moving on the dataset. As we said before, in those frames, the target variable is modified to be a -1, since the 0 represents a direction in a 0 to 360 degree system. This caused that when the input was forced to


(a) Histogram of the target value

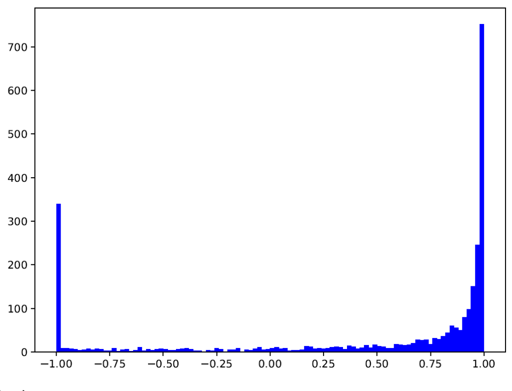

(b) Histogram of the predictions by the neural network

Figure 4.4: Target (a) and obtained (b) histograms of the training. It depicts the influence of the negative variable in the outcome of the whole network. Data obtained with the permission of *From The Bench Games, S.L.*

be in a range between -1 to 1 (using a tanh activation function) most of the results fell in the negative space of the prediction range. Moreover, when the input was forced to be in a range between 0 and 1 (using a sigmoid activation function) [70], the inputs were mostly grouped near the 0, because of the influence of the -1 as a target. More details can be seen in Figure 4.4.

Motivated by those results, we decided to stop using the value -1 to model the absence of movement in the agent and, after talking with the From The Bench Games team, we decided that the bot should never stop moving. This was decided in part for the problems presented by the frames where the player was not moving in the dataset, which was almost half of it and therefore influenced deeply the training process, and in part because the Decision Tree being used until the moment already had this peculiarity.

We proceeded to curate the dataset, eliminating the iterations without movement, but augmenting the number of samples obtained in the process and matching a similar amount of the training samples used previously. The dataset was curated by checking all the frames of the database and discarding those where the forward x and y values where 0. Then, the dataset was augmented by obtaining more recorded matches from the game's closed beta. This also served to clean all the frames recorded where the players where waiting at the beginning of the game, during the countdown before the start. In Figure 4.5 we present the histogram of the resulting dataset's target variable.

This resulted in an vast improvement on the validation loss and accuracy, mostly because it allowed us to modify the network to have an output in the range between 0 and 1, although some values went a little bit over the range. Also, instead of piling up closer to 0, therefore making the network to always predict movement to the right of the screen, were distributed a little more along the whole prediction range. Therefore, not only the accuracy of the model improved, but the values being predicted by the

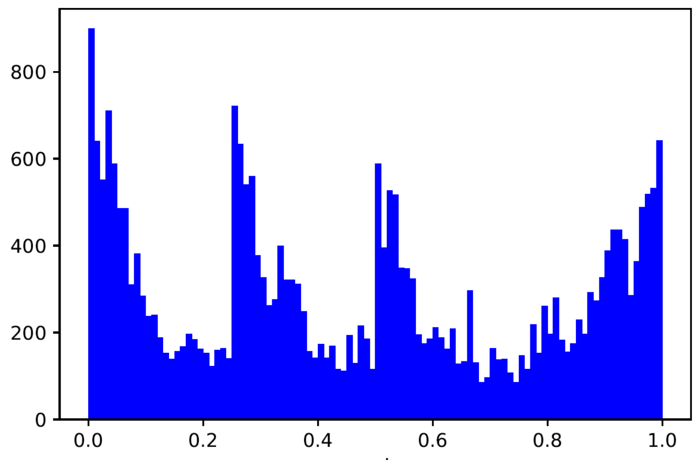


Figure 4.5: Histogram of the target value "angle" for the dataset used in the training process of the ANN. Data obtained with the permission of *From The Bench Games, S.L.*

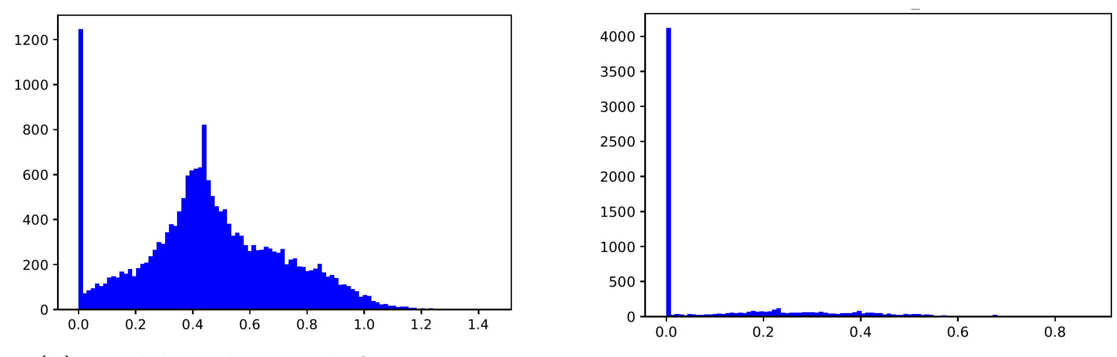
network did the same.

However, after several tests, whatever the accuracy or histograms obtained were, when the correspondent model was loaded in Unity for testing, the behavior of the bot never improved. After looking deeply at the possible causes of it along the pipeline developed previously (and building a new model exporter in the process, more information about that in Section 4.3) it was possible to isolate the problem to the training part of the code. In Figure 4.6 we can see histograms of the same neural network in the same training process, before and after being saved in a Protobuf and a Checkpoint format.

This leaded us to find that there was an error in the code responsible of saving the models. When the network finished training and the Python code proceeded to save it into a format which later the pipeline will use for exporting it, a new Tensorflow session was created for the process. This caused that although the graph was saved with the correct structure, therefore not raising any errors, the values inside of the network were random since it was not the same session that trained the network.

After solving this issue, the development of the deep learning bots progressed. Several models where trained in the following weeks, researching about different structures and hyperparameters for the neural network. During the development process, a scientific approach was folowed. We started with a very simple neural network of two layers and we increased the number of layers and neurons in each layer until the increase of it did not suppose a decrease in the validation loss.

The same procedure was done for activation functions, where following certain development guidelines we experimented with the combination of various of them in different layers; for the learning rate and momentum [71], where we found the combination of values which performed better together; for the activation functions, which we experimented with different combinations of them; and finally even with the number of



(a) Model predictions before saving **(b)** Model predictions after saving

Figure 4.6: Histograms of an ANN model before (a) and after (b) saving it. The data obtained in the Subfigure (b) should be the same as the one at Subfigure (a). It clearly highlighs the presence of a bug in the saving process. Data obtained with the permission of *From The Bench Games, S.L.*

epochs of training.

In Listing 4.4 the code of the final architecture of the ANN is presented. The code presented here is the combination of various functions in the original code, with the purpose of facilitating the reader the visualization of the information. The number of neurons in the initial layers is set to a closer value of the input received [72], and later the number of neurons descend gradually until the output single neuron is reached. Due to the number of layers, the usage of sigmoid activation function was replaced by ReLu and TanH functions to avoid the Vanishing Gradient Problem [36], which affected the network training. The rest of the choices were made because they obtained better resulting values than their counterparts.

Listing 4.4: Final model developed with an ANN architecture. The hyperparameters and structure was defined using the procedure explained previously. The neural network is defined by an input layer, seven hidden layers and an output layer, all defined with Keras Functional API. The optimizer chosen is the Stochastic Gradient Descent.

```

batch_size = 8192
epochs = 10000

visible = Input(shape=(n_cols,), name="state")
hidden1 = Dense(20, activation='relu')(visible)
hidden2 = Dense(20, activation='tanh')(hidden1)
hidden3 = Dense(20, activation='relu')(hidden2)
hidden4 = Dense(20, activation='tanh')(hidden3)
hidden5 = Dense(15, activation='relu')(hidden4)
hidden6 = Dense(10, activation='relu')(hidden5)
hidden7 = Dense(5, activation='tanh')(hidden6)
output = Dense(1, activation='relu', name='action')(hidden7)
model = Model(inputs=visible, outputs=output)

sgd = keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)

nn = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1,
     validation_data=(x_test, y_test))
```

While the loss values kept getting lower and lower during the training and validation, it did not translate to an improvement in the behavior of the bot: it moved randomly, tending to move to a certain edge of the field or other depending on the version trained. To solve this issue, we tried to look for errors in both the generalization capability of the neural network and the data itself.

First we looked at the generalization process. We analyzed the results obtained in the trained models, and concluded that maybe the improvement in the results was obtained because of overfitting [73], which meant that the networks learned how to obtain good results in the training process by adapting to the specific data of the training dataset, and was not able to generalize that to external data. For that reason, we created an alternate model using Dropout [74], based on the one presented before.

The development of this model using Dropout was done with the same methodology than the explained previously. The resulting model, as expected, trained much worse and needed more epochs to obtain a validation loss closer to the model without dropout. But unfortunately, the agent kept doing the same random behavior than before.

After this, a model using Batch Normalization [6] was developed, to compare and contrast with both the ANN and Dropout models. It obtained better results than both on the loss in the dataset, and also trained more easily than the two previous models, but in Unity it turned into random movement too.

Finally, we looked at the data and the causes of differences between the Python and the Unity's version. We observed that the training process was done taking as a reference the player's position and forward direction instead of the companion, while in the data frame constructed in Unity before each iteration, player and companion data was swapped, to make the agent understand that he was the companion and not the player. Taking into consideration that in the game the player always starts in the bottom half and the companion in the top half, this meant that there was no reference of the game's starting position in the training process.

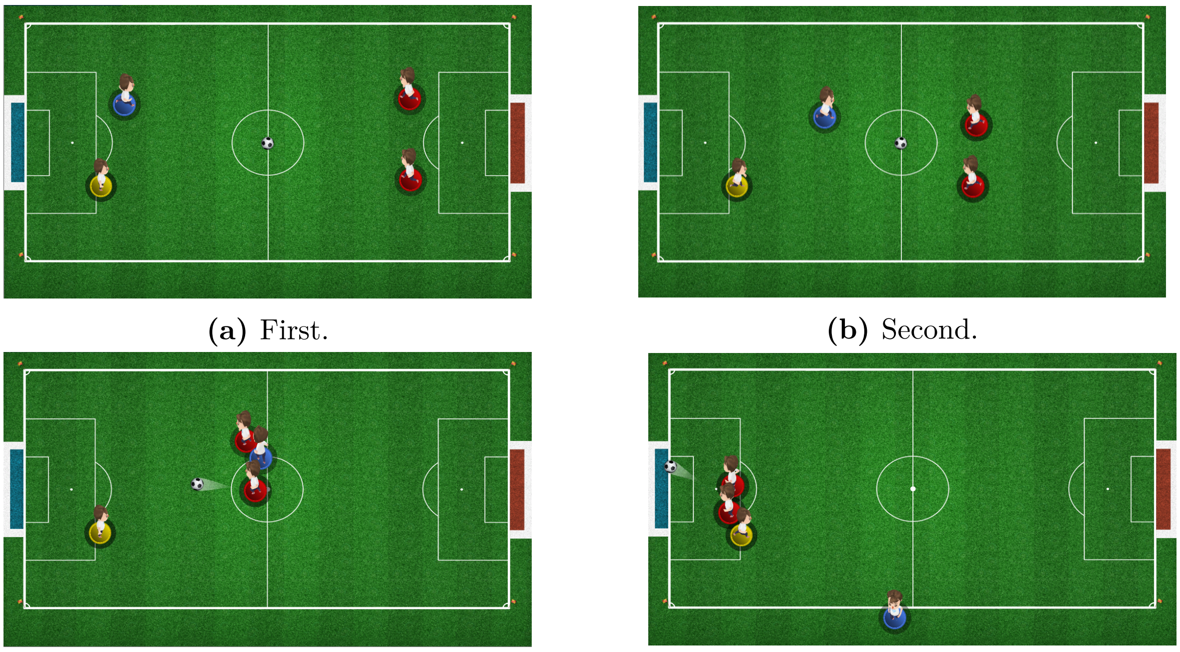
For that reason, we decided to vertically invert all the data of the dataset, obtaining a new mirrored dataset, which replicated the companion's starting position. When this was tested on the game, it caused the agent to start moving to the ball during the first stages of the game, confirming our theory that there was a lack of training examples that acted as good points of reference. However, after the first couple of seconds, or even less depending on the try, the bot started to move randomly and behave in a way similar to the previous versions.

Also at this time, all the data positions were normalized, both in the dataset and the game, to a range between 0 and 1. The normalization was done in order to clean a little bit the differences in sizes, and to have a reference point starting at 0 and not at an arbitrary value too.

We also tried adding new variables, constructed from the ones in the dataset. We tried adding relative distances between the agent and the ball, and the agent and his companion. Also we tried adding relative distances to the scoring lines, always with an effort to orientate the player. None of those additions became into an improvement, instead some of the additions worsened the results, and it was decided to finally remove

Model	Validation Loss
ANN	0.0465
ANN + Batch Normalization	0.0452
ANN + Dropout	0.0756
ANN + Inverted Data	0.0508
ANN + Inverted + Normalized Data	0.0537
ANN + Inverted + New Data	0.0542
ANN + Augmented Dataset	0.0533

Table 4.4: Comparison of model's validation loss between the different ANN approaches developed. The values correspond to the best performing neural network with each approach, but do not show the average performance of each approach. Obtained with the permission of *From The Bench Games S.L.*



(a) First. **(b)** Second. **(c)** Third. **(d)** Fourth.

Figure 4.7: Sequence of movements the player's companion, identified by a blue area, does at the beginning of the game. In the images (a) and (b) we can observe how the bot moves to the ball, identifying it and trying to fight over its control. In image (c), although the ball has not been taken the bot identifies that it must go back to defend. However, in the end it gets lost, as we can see in (d). Images obtained with the permission of *From The Bench Games, S.L.*

all the new variables created.

Starting from the previous conclussions that the agent needed more referential data we tried to expand the dataset of the game. We tested the capabilities of the training machine and augmented the number of training examples to about 200 full matches, always from the two versus two game mode. Unfortunately, this did not supposed an improvement on the bot behavior.

A summary of all the different Artificial Neural Networks developed in this process, accompanied by the Validation Loss of the best performing model can be found in Table 4.4. However, it is important to note that, as it was said before, the best score does not correlate with the best agent behavior of the model in Unity, being that factor what ended up guiding the development mostly.

An example of the behavior obtained by the Artificial Neural Networks models can be seen in Figure 4.7. The sequence presented corresponds to an ANN with inverted and normalized data, but withouth Batch Normalization, Dropout or new data entries. In the sequence we can observe the typical behavior of the neural network bots we developed and explained before, and it represents the overall performance of the results obtained.

In the sequence we can see that, initially, the agent, surrounded by a blue marking, seems capable to identify the ball and the other agents, and moves to the ball accordingly to what we expected and wanted. But we can see that after clashing with the enemy team, it falls on area less known for him, since there are much less examples of that exact same situation than from the beginning of the game, and starts to show an herratic behavior. Finally, it ends up running to an edge of the field, without being capable to go back and play. This final herratic behavior does not change even if we, controlling the player, try to make him interact with the ball or try to attack the enemy team.

The direction edge chosen by the bot to end up running seems to be random, since in subsequently executions it chooses different directions, but the rest of the behavior is more or less the same in all the games.

Although the version presented here is not the best version in terms of Validation Loss, as we can see in Table 4.4, we consider it to be the best in terms of behavior. The rest of the versions, even when they show an improvement in the training stage of the development, show a lack of it when they are tested on Unity. Therefore, the other versions developed obtain either a similar behavior or a worse one, usually because the agent gets lost at the beginning of the match and is not capable of even fighting the ball initially.

After looking at the reasons behind the lack of improvement, we concluded that it was because of a number of factors: First, most of the games had incomplete examples, since the powers usage was not recorded. Second, the frames with the player not moving were removed, therefore eliminating a big chunk of data in the process. Third, the game state was too complex to cover it with this approach, since each combination of five different agents supposed a distinct output, making it hard for the network to generalize from one behavior to other. Fourth, the dataset had contradictory entries, since each player behaved different in same situations. Finally, and most important, the dataset was very unbalanced, since each time a goal was scored the game restarted from the starting position. Therefore, most of the examples in the dataset were set in the starting position, or closer to those positions, and the neural network probably learned to ignore other data to minimize the error rate.

At this point, we found that it was not possible to keep improving the model without a change in the approach taken. We concluded that either a change in the neural network technique being used, or the way the data was recollected and used was needed. After looking at possible solutions, both from traditional Deep Learning and Deep Reinforcement Learning, and because of FTB team needs and requirements, we started developing a Long Short-Term Memory (LSTM) model [37]. This decision was motivated because we thought the agent needed to know the previous game states to better understand the movements of all the agents and the ball. It also was motivated because we found working examples of similar cases [75] with LSTM.

4.4.3 Long Short-Term Memory

The development of a LSTM model required several changes and adaptations in various points of the code. This was because Long Short-Term Memory, as others Recurrent Neural Networks, need to receive as an input a series of entries, instead of a single one. For that reason, we needed to modify and adapt the entire dataset, making it capable of, given a number of length, creating an entry with as many frames as length, and using the last frame's output as the output of the sequence. Similarly, the Unity code required further adaptations, making it capable of acumulating various time frames before a output is infered from the network.

The development of the LSTM model followed the same principles of the ANN. Various number of hyperparameters and architectures were tested, obtaining the best individual values for all of them. However, due to performance reasons and the little development time left, the final architecture being used was very simple, and only with the purpose of testing the approach more than the model itself. The model can be seen in more detail in Listing 4.5.

Listing 4.5: Final model developed with an LSTM architecture. Both the structure and hyperparameters were obtained from extensive experimentation. The network is composed of a single Input layer and a hidden LSTM layer composed of one node, all defined with Keras Functional API. The optimizer chosen is the Stochastic Gradient Descent.

```

batch_size = 8192
epochs = 100

visible = Input(shape=(n_cols,), name="state")
output = LSTM(1, name='action')(visible)
model = Model(inputs=visible, outputs=output)

sgd = keras.optimizers.SGD(lr=0.001, momentum=0.9, decay=0.0, nesterov=False)
nn = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1,
validation_data=(x_test, y_test))
```

Models in combination with fully connected layers were developed too, as is usual in RNN and LSTM architectures, but surprisingly the model which obtained the best performance in the training and validation was the one presented above. This contrasted with the references taken for the development of this approach [75], which proved working results when combined with fully connected layers.

The number of sequences acumulated also supposed no improvement in the results taken, in opposition to what we expected initially. In fact, as the sequences grew longer, the training process struggled to obtain good results, and finally we opted for sequences of length 3, as we tried to balance both the performance of the training process and the capabilities of the network of learning more long tern relations and dependencies.

Nonetheless, both the models with and without fully connected layers behaved in the same way inside of Unity. This indicated us that probably the problem we were experimenting with the development of our models had more relationship with the data and the imitation approach than in the models themselves.

Unfortunately, in parallel with the implementation of the LSTM model, the development of the game advanced. This meant the creation of a new 3 versus 3 game mode, thus changing the size and starting positions of all the game modes, which supposed the divergence between the working version and the version used for the development of the neural networks. This had not an easy solution, since the dataset used only contained entries from the version prior to the changes, and supposed that the final deep learning models being tested were only for academic reasons, since the usage of them in a production environment was put aside.

4.4.4 Summary

To sum up, even when not a single approach achieved a good working state for a production environment, we can extract some satistying points from the work presented here:

First, a fully functional system for integrating Deep Learning models was developed, adapting and enlarging the scope of it as the project needed it. A customizable platform that can be easily modified to the needs of our project was obtained. Also, we learned about the especific problems we can face when developing a platform of this type. The lack of standards and the earlier stages that some of the technologies on the Deep Learning field are found can provoke problems that delay the development process.

Later, the advantages and disadvantages of different deep learning approaches were tested, and specially from a performance point of view we were able to identify which ones were better, both in training and in production.

We could conclude that the usage of Convolutional Neural Networks when the input of the network is a constructed image from data is usually not a good idea, since the performance lost on providing the data in the form of an image is not compensated with better results.

We could also conclude that both Artificial Neural Networks and Recurrent Neural Networks can obtain similar results when the data is constructed in a way that the long and short time dependencies are taken advantage of.

Finally, the dataset used for the training was largely analyzed and severely curated, improving its efficiency and setting the starting point for obtaining better results in the future. We experimented with many transformations and new values and defined what provided better performance and what not.

5 Conclusion

This chapter summarizes the work presented in previous chapters and extracts conclusions from them, drawing guidelines and future lines of work. It is divided in three sections. In Section 5.1 we summarize and conclude the work done in this Thesis. In Section 5.2 we point out the most importat aspects carried out in this work. In Section 5.3 we propose possible improvements and lines of work that could be explored in the future.

5.1 Conclusions

In this Thesis, we have started with an extensive review of the current state of the art in the Artificial Intelligence applied to videogame development. Fist, we analyzed the current methods and concrete circumstances that surround the development of an AI for a videogame, in parallel with the evolution to the game AI techniques until nowadays. It showcased the difficulties game AI had through time, and the slower pace at which it was developed in comparison with other aspects of videogames such as graphics or mechanics.

Then, we explored the current Deep Learning techniques, with a focus on the internal workings of Neural Networks, specially in the types of Neural Networks used later on this Thesis. We went through traditional Neural Networks, better known as Artificial Neural Networks, and some special types of Neural Networks relevant for our work, such as Convolutional Neural Networks, Recurrent Neural Networks and Long Short-Term Memory Networks.

After that, we went through the most important works that used Deep Learning applied to Videogames, and analyzed the current trend in the usage of Videogames as a platform for Deep Learning experimentation, in opposition with the usage for the development of Game AI. As potential reasons for this development difference, we propose both the secondary aspect Game AI has been until recently, and specially the lack of defined Deep Learning solutions that are easy to integrate in game engines in addition to the lack of established solutions that can provide satisfying results. Also, we point out that most of the Deep Learning solutions applied to games have been done on games already developed, and that is another reason behind the lack of usage of Deep Learning for game development, since Deep Learning still has to be proven to work in a constantly changing environment such as a game in development.

As for the work carried out, we developed a fully-functional Deep Learning pipeline, capable of integrating Deep Learning models created in Python, either in Keras or in Tensorflow, inside of the Unity game engine withouth much effort.

Finally, we developed many neural networks with the purpose of serving as a Non-Playable Characters in the mobile game HardBall. We compared all the approaches carried through the development process, and showcased the advantages and disadvantages of each one. The final code of this project can be accessed at the repository: `https://github.com/AdrianFL/Bachelors_Thesis`.

As for the reasons behind of the functionament of the neural networks, we can conclude that:

- In order to develop Deep Learning approaches based on learning from game data or players, a deterministic AI needs to be implemented to allow players to play against it. Therefore, developing Artificial Intelligence in that scenario doubles the amount of work needed without much potential reward.
- A game in development is constantly evolving, usually changing important aspects of the game constantly. For that reason, a Deep Learning approach encompasses a big risk of getting outdated through the development process.
- The support that exists for Deep Learning solutions in established game engines, even when it is being developed lately, is still in an early stage and needs further improvements to be used extensively.
- In relation with the previous points, debug and test options and tools, especially when working with Deep Learning outside of Python, needs to be improved to allow game developers more control over the AIs implemented.
- Although being improved constantly, performance in low end devices, especially on mobile platforms, is still not good enough to use certain Deep Learning techniques on them.

5.2 Highlights

The highlights of the work developed in this document are:

- In-depth study of the development of Artificial Intelligence applied to game development since the beggining of videogames.
- Exploration of the most common Deep Learning techniques relevant to this work.
- Review and summary of the most relevant Deep Learning solutions and projects relationed with videogames, either for game development or using games as a research platform.

- Creation of a Deep Learning development pipeline, capable of exporting neural networks developed in Python to C#, integrating and configurating it for usage in Unity.
- Development of many Neural Networks models based on three different approaches: ANNs, CNNs and RNNs.
- Analysis of the aplication of Deep Learning models to games in their development process.

5.3 Future Work

Due to the time constraints imposed on this project, many potential improvements and lines of work were left out for the future. Here we summarize them to conclude this work:

- Regarding the Long Short-Term Memory approach, improvements in the structure and complexity can be made, exploring the functionality of the network with higher complexity models.
- Combine some or all of the approaches developed in this work in a single model, combining ANNs, CNNs and RNNs, as it it seen on some of the works presented in Chapter 3.
- As it has been probed out to work specially well on the field, explore Deep Reinforcement Learning solutions, specially with the usage of the ML-Agents Toolkit. This could help to overcome some of the problems presented in the conclusions, because DRL would allow us to obtain a certain policy that, even when the game field changes, it properly behaves. Analyzing and comparing the results of a Deep Reinforcement Learning approach with the ones presented on this work would be a great improvement to the results too.
- Explore other ways of obtaining data from the players to train Deep Learning Agents, and also explore which aditional data would be interesting to collect to improve the training process.
- Developing native support for the creation and training of Deep Learning models in Unity, without the need of integrating Python in the process, and allowing us to test and debug the models created directly on the engine.

Bibliography

[1] A. Juliani, V. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, “Unity: A General Platform for Intelligent Agents.,” *arXiv preprint arXiv:1809.02627*, 2018. https://github.com/Unity-Technologies/ml-agents.

[2] I. S. Mohamed, *Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques.* PhD thesis, 09 2017.

[3] A. García, “3d object recognition with convolutional neural networks,” Master’s thesis, Universidad de Alicante, 2016. `https://rua.ua.es/dspace/handle/10045/57438`.

[4] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2014.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[6] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *arXiv preprint arXiv:1502.03167*, 2015.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.

[8] Github, “The state of the octoverse, 2018.” `https://octoverse.github.com/`.

[9] F. Chollet *et al.*, “Keras.” `https://keras.io`, 2015.

[10] Theano Development Team, “Theano: A Python framework for fast computation of mathematical expressions,” *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[11] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. `arXiv:1512.03385` [cs.CV].

[12] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. `https://arxiv.org/abs/1512.00567` [cs.CV].

[13] Unity, “Unity megacity.” `https://unity.com/megacity`.

[14] Unity, “Unity growth facts.” `https://unity3d.com/public-relations`.

[15] OpenAI, “Openai five.” `https://openai.com/blog/openai-five/`.

[16] Unity, “Puppo, the corgi: Cuteness overload with the unity ml-agents toolkit.” `https://blogs.unity3d.com/es/2018/10/02/puppo-the-corgi-cuteness-overload-with-the-unity-ml-agents-toolkit/`.

[17] JamCity, “Snoopy pop.” `https://blogs.unity3d.com/es/2019/04/15/unity-ml-agents-toolkit-v0-8-faster-training-on-real-games/`.

[18] A. Turing, “Digital computers applied to games,” in *Faster than thought: a symposium on digital computing machines* (B. V. Bowden, ed.), ch. 25, 1953.

[19] J. Schaeffer, R. Lake, P. Lu, and M. Bryant, “Chinook the world man-machine checkers champion,” *AI Magazine*, vol. 17, no. 1, pp. 21–21, 1996.

[20] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, “Checkers is solved,” *science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[21] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, “Deep blue,” *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[22] V. Burnham, *Supercade: a visual history of the videogame age 1971-1984*. Mit Press, 2003.

[23] J. Pittman, “The pacman dossier.” `https://www.gamasutra.com/view/feature/132330/the_pacman_dossier.php`.

[24] G. N. Yannakakis, “Game ai revisited,” in *Proceedings of the 9th conference on Computing Frontiers*, pp. 285–292, ACM, 2012.

[25] M. Pirovano, “The use of fuzzy logic for artificial intelligence in games,” *University of Milano, Milano*, 2012.

[26] S. Rabin, *AI Game Programming Wisdom*, vol. 1. Charles River Media, Inc, 2002.

[27] S. Rabin, *Game AI Pro 3: Collected Wisdom of Game AI Professionals.* AK Peters/CRC Press, 2017.

[28] S. Woodcock, J. Laird, and D. Pottinger, "Game ai: The state of the industry," *Game Developer Magazine*, vol. 8, 2000.

[29] M. Mateas and A. Stern, "Façade: An experiment in building a fully-realized interactive drama," in *Game developers conference*, vol. 2, pp. 4–8, 2003.

[30] R. J. Schalkoff, *Artificial neural networks*, vol. 1. McGraw-Hill New York, 1997.

[31] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological Review, 65(6), 386-408.*, 1958. `http://dx.doi.org/10.1037/h0042519`.

[32] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[34] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *International conference on artificial neural networks*, pp. 92–101, Springer, 2010.

[35] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *CoRR*, vol. abs/1808.03314, 2018.

[36] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[37] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[38] K. Arulkumaran, A. Cully, and J. Togelius, "Alphastar: An evolutionary computation perspective," *CoRR*, vol. abs/1902.01724, 2019.

[39] N. Justesen, P. Bontrager, J. Togelius, and S. Risi, "Deep learning for video game playing," *IEEE Transactions on Games*, 2019.

[40] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[41] B. W. Mott, "Stella atari simulator." `https://stella-emu.github.io/`.

[42] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling, "Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents," *Journal of Artificial Intelligence Research*, vol. 61, pp. 523–562, 2018.

[43] O. Team, "Openai gym." `https://gym.openai.com/`.

[44] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, "Gotta learn fast: A new benchmark for generalization in rl," *arXiv preprint arXiv:1804.03720*, 2018.

[45] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, *et al.*, "Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[46] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski, "Vizdoom: A doom-based ai research platform for visual reinforcement learning," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, IEEE, 2016.

[47] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[48] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline monte-carlo tree search planning," in *Advances in neural information processing systems*, pp. 3338–3346, 2014.

[49] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[50] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *arXiv preprint arXiv:1611.01224*, 2016.

[51] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[52] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, pp. 1928–1937, 2016.

[53] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[54] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[55] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[56] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[57] L. Baker and F. Hui, "Innovations of alphago." `https://deepmind.com/blog/innovations-alphago/`.

[58] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[59] G. M. J.-B. C. Chaslot, *Monte-carlo tree search.* Maastricht University, 2010.

[60] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver, "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II." `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[61] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, *et al.*, "Deep reinforcement learning with relational inductive biases," 2018.

[62] OpenAI, "Openai five arena." `https://arena.openai.com/#/`, 2019.

[63] T. Salimans and R. Chen, "Learning montezuma's revenge from a single demonstration." `https://openai.com/blog/learning-montezumas-revenge-from-a-single-demonstration/`, 2018.

[64] A. Juliani, "On "solving" montezuma's revenge." `https://medium.com/@awjuliani/on-solving-montezumas-revenge-2146d83f0bc3`, 2018.

[65] D. Lange, "Obstacle tower challenge: Test the limits of intelligence systems." `https://blogs.unity3d.com/es/2019/01/28/obstacle-tower-challenge-test-the-limits-of-intelligence-systems/`, 2019.

[66] A. Juliani, A. Khalifa, V.-P. Berges, J. Harper, H. Henry, A. Crespi, J. Togelius, and D. Lange, "Obstacle tower: A generalization challenge in vision, control, and planning," *arXiv preprint arXiv:1902.01378*, 2019.

[67] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh, "Action-conditional video prediction using deep networks in atari games," in *Advances in neural information processing systems*, pp. 2863–2871, 2015.

[68] L. Prechelt, "Early stopping-but when?," in *Neural Networks: Tricks of the trade*, pp. 55–69, Springer, 1998.

[69] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[70] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems*, vol. 1, no. 4, pp. 111–122, 2011.

[71] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, pp. 1139–1147, 2013.

[72] S. Karsoliya, "Approximating number of hidden layer neurons in multiple hidden layer bpnn architecture," *International Journal of Engineering Trends and Technology*, vol. 3, no. 6, pp. 714–717, 2012.

[73] I. V. Tetko, D. J. Livingstone, and A. I. Luik, "Neural network studies. 1. comparison of overfitting and overtraining," *Journal of chemical information and computer sciences*, vol. 35, no. 5, pp. 826–833, 1995.

[74] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[75] C. Trivedi, "Building a deep neural network to play fifa 18." `https://towardsdatascience.com/building-a-deep-neural-network-to-play-fifa-18-dce54d45e675`.