TÉCNICO LISBOA

Imitation-based Artificial Player for Pic-A-Boo

Elio Samuel Granados Freitas

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Carlos António Roque Martinho

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Carlos António Roque Martinho
Member of the Committee: Prof. João Miguel de Sousa de Assis Dias

June 2020

Acknowledgments

This small text might fall short in expressing my gratitude to all the people that collaborated to create this project. However, I'll do my best, first I want to thank IST for accepting my candidature which was the first step in the creation of this project. I would specially like to thank Professor Carlos Martinho who helped me with this project from the very beginning and for his labor in teaching about game development which was my favorite course. A Big thanks to Nuno Monteiro Pic-a-boo's main developer for providing access to the code and for collaborating in the creation of this project. I would also like to thank Hitachi Vantara company where I had been working the past 4 years for allowing me the flexibility to be able to study and work at the same time. Thanks to the UI-kit team, Boba Fett team and my mentor because their support was a big part in the creation of this work. As for my family thanks to both my mother Maria Isabel and my Grandmother Laurinda for advising and supporting me on my life. My father Elio and my sister Dangelys thanks to you even though you are far away from me. Finally, thanks to my girlfriend Ivone for supporting and allowing me to focus on this project.

Abstract

Pic-a-boo is multiplayer game about taking pictures of other players in a dark room. Not all information is available at all time to the players, so some guessing is required to win. In this work, we developed artificial agents that would be able to imitate human players and serve as their substitute when needed. This was achieved through an artificial neural network iteratively trained by playing against a human player, in the attempt of capturing and replicating what makes his or her gameplay distinct and identifiable as a player. Although the agents were able to navigate the map believably, other actions such as taking photos were produced randomly without any discernible human pattern.

Keywords

Artificial Intelligence; Artificial Neural Networks; Human Behavior; Imitation Learning

Resumo

Pic-a-boo é um jogo onde múltiplos jogadores tentam tirar fotografias uns dos outros numa sala às escuras. Nem toda a informação está disponível a todo o momento aos jogadores, obrigando a alguma dedução para vencer. Neste trabalho, desenvolvemos jogadores artificiais que são capazes de imitar jogadores humanos para poder servir de substitutos quando necessário. Para resolver este problema, foram usadas redes neurais artificiais treinadas iterativamente com jogos contra um jogador humano, com o objetivo de capturar e replicar a forma específica de jogar deste jogador. Apesar dos agentes desenvolvidos serem capazes de navegar no mapa do jogo de uma forma credível, outras ações como tirar fotografias foram reproduzidas de uma forma aleatória sem apresentar qualquer padrão humano identificável.

Palavras Chave

Inteligência Artificial, Redes Neurais Artificiais, Comportamento Humano, Aprendizagem por Imitação

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
API	Application Program Interface
ALE	Arcade Learning Environment
FP	False Positive
FN	False Negative
ICM	Intrinsic Curiosity Module
LSTM	Long Short Term Memory
MCTS	Monte Carlo Search tree
CNN	Convolutional Neural Network
ANN	Artificial Neural Network
PPO	Proximal Policy Optimization
PAG	Pic-a-boo Agent Gym
SDK	Software development kit
TP	True Positive
TN	True Negative
UMLAT	Unity Machine Learning Agents Toolkit
UI	User Interface

1

Introduction

Contents

1.1 Motivation

It is challenging for an Artificial Intelligence (AI) to imitate human behavior. The challenge lies in the difficulty of predicting human behavior. There are many humans with different personalities. This make it hard to express human behavior as a simple mathematical model or algorithm. However, there have been several attempts at achieving this. Some of the most successful ones used recordings of players to train algorithms. Others analyzed common strategies used in games and reproduced them.

Creating a **human-like** AI is the current challenge that the game Pic-a-boo is facing. A **human-like** AI is an AI capable of imitating the actions patterns of a human. Pic-a-boo is about taking pictures of your enemies in a dark room. The winner is the player able to take pictures of all the other players. Figure 1.1 showcases a pic-a-boo match. It uses a top-down view map with several dark zones, the players are only able to only see their enemies and themselves in light sources. Pic-a-boo is a game that can be classified as an action game, this type of games require fast response times. This limits the type of solutions to be implemented because the decisions must be in real time. Otherwise, the credibility of the agents is compromised.

Apart from this pic-a-boo presents an interesting challenge for an AI because it is a game with hidden information. An AI by virtue of being a program may have access to information that the player does not. However, to be believable and fair the AI should behave as if it did not possess this data.

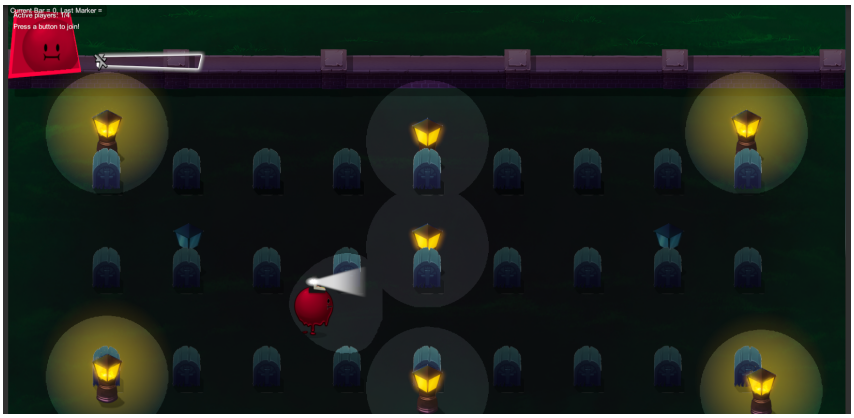

Figure 1.1: Pic-a-boo match.

But why should an AI imitate a human player?. Several reasons exist, One reason is to fill vacant players for multiplayer games, because sometimes finding 4 human players is not feasible so an AI that plays the part helps in obtaining the complete game experience. Another one is to present a fair challenge to a new player. A match between humans with the same game playing condition its perceived as more fair, compared to a match between an AI that only seeks to win the game. This type of AI will be designated as perfect-play AI for this project. Perfect-play AI is an AI that seeks to solve the game. Perfect-play AI seeks the best answer to the problem presented in the game. For instance, in real time strategy games

like Starcraft [1], players are sometimes required to manage each unit individually. Figure 1.2 showcases Starcraft game-play of unit management in a top-down map. A perfect play AI could manage each unit in an almost instantaneous time. For an average player this would be impossible. A perfect-play AI is not limited by fairness and may be frustrating for newer players. New players haven't learned the game yet. A novice confronting a perfect player is an unfair challenge. Even for an expert in the game, whom may like their challenge, it is still unfair. It is unfair because it is rare for a human to perform in a perfect and instantaneous manner. It is unlikely that humans will be able to manage units at the same speed as a computer. Instead, a **human-like** AI purpose is to imitate the way a human will play, this includes imitating mistakes.

Figure 1.2: Starcraft match.

But what requirements exist for creating a **human-like** AI for Pic-a-boo? First, it is necessary to find an appropriate algorithm. Given that a key factor in selecting an algorithm is the response times using an Artificial Neural Network (ANN) might be appropriate. Training a neural network can take from a few minutes to several hours. However, once trained a ANN does not take long times to calculate, so it won't compromise the response time of the game.

Apart from this an ANN is capable of reproducing patterns and to create a **human-like** AI another important aspect of the development is to capture the player actions. This is because the data is going to be the basis for the player model that the AI is going to imitate. Finally, the resulting agent similarity to a human requires measuring. The measuring will help to determine if the resulting agent is behaving like a human.

[1] StarCraft II: Wings of Liberty real-time strategy video game developed and published by Blizzard Entertainment.

1.2 Objective

Implement an artificial intelligence agent into the Pic-a-boo game that imitates human behavior.

1.2.1 Sub-objectives

- Analyze matches of the game Pic-a-boo to create a data model for player inputs.
- Gather data using the data model.
- Find and apply a measuring technique to the resulting algorithm to check the quality of the imitation.

1.3 Outline

- Chapter 2 Related Work: In this section previous works in the area of human like AI are reviewed. The aim is to extract information to help guide this project.
- Chapter 3 Environment Description: In this section Pic-a-boo is described. To understand the rules, limitations and requirements for the agents
- Chapter 4 Implementation: The proposed solution is described alongside its implementation.
- Chapter 5 Results: The measurements produced by the agents are shown and analyzed in this section.
- Chapter 6 Conclusion: The conclusions of this work are presented as well as some remarks on possible future work.

2

Related Work

Contents

2.1 Chapter description

In this chapter various related topics will be explored to gather the required knowledge needed to proceed with the project. The first topic is going to be Player modelling to understand the different approaches to imitate a player. Following player modelling, the next topic will be neural networks. In this section previous works will be explored to find a good configuration for the ANN that will be used. Pic-a-boo is developed in Unity, Unity provides several libraries for different purpose one of them being machine learning. So the fourth section will be dedicated to exploring its capabilities. The final section will be about how to measure AI with the aim of exploring the methodologies that had been used previously to determine the quality of the resulting AI.

2.2 Player Modelling

Player modelling is the study of computational models of players in games. It includes the detection, modelling, prediction, and expression of human player characteristics. These characteristics manifest themselves through cognitive, affective and behavioral patterns. The main goal of player modelling research is to understand how players interact with a game. Following this, a model is produced based upon that interaction [1].

While playing video games humans produce dynamic and complex behaviors. There are 2 approaches to model these complex behaviors when creating player models. The first approach is a theory based approach which relies on making a rough approximation of the human's behavior. The agents produced from this method will behave in the predetermined behavior or strategy, for example fleeing attackers. According to [2] we can call this model-based approach a **persona** or personality. The second approach is a data-driven approach which rely on gathering data from players to analyze the patterns inside the data. The agents produced from this method will then behave in a way that agrees with the patterns extracted from the play traces. We can call this data-driven approach a **clone**.

Both cases have problems, for instance, clones are prone to over-fitting, where the agent only learns to cope with situations appearing in the play traces. In the case of new situations, the agent might have irregular behavior. Personalities to a large extent solve the irregular behavior problem. The solution provided is through learning a robust general strategy. This solution is also called a decision-making style. But a decision-making style can be general and predictable.

This project requires choosing between clones or personas. Trying to formulate a general strategy to imitate a player is difficult and will lead to a predictable agent. Guesswork is one of the main components of Pic-a-boo's matches so being predictable would eliminate a big part of the challenge. The cloning method might bring irregular behavior but humans are not a regular creature so it isn't a big issue. So between the two the cloning methodology is the chosen.

2.3 Artificial Neural Networks

Artificial Neural networks (ANN) are a family of algorithms used to simulate human capabilities among other uses, for instance image recognition, face identification and self driving cars [3]. These tasks are easy for humans to perform but not for a computer. This is because it's not easy to describe these task in a couple of reproducible steps. It is not easy because the numbers of steps might be too big or have too many caveats, thus making it impractical to code. ANN overcome this problem by imitating a human brain.

There are various examples of using an ANN for creating human behavior, for instance the company [1] behind candy crush saga[2]. The game is shown in Figure 2.1. The company sought to improve their game testing by using a Convolutional Neural Network (CNN). CNN is a specific type of ANN. CNN original purpose was for image-like data feature extraction.

Figure 2.1: Candy Crush Saga puzzle.

The need came from the speed at which they wanted to release new content. The speed of release was not aligned with the velocity of the testing efforts. This because humans were required to test the product. Human testing comes with the disadvantage of introducing latency and costs [4]. They saw an advantage in creating a human behaving algorithm to automate testing. The idea was that an agent could run tests at any time after the new levels were completed. These automated tests would help find problems in the new levels increasing the speed of development and reducing the costs of human testers.

According to [4] there have been many attempts at creating a human agent, for example simulating a player playing a level using an agent with a simple heuristic. But these approaches lack one important aspect. The approaches were trying to imitate humans without human input. For this fact the researchers

[1] King Digital Entertainment
[2] Candy crush sagademo

sought to include human data because they thought it could be beneficial.

The general approach they used was an agent that would play the game with the help of a CNN to guide decisions. The agent would then generate the metric of interest needed to test the quality of the levels. Then the researchers would compare the values from the agent with the values from humans. The closer the strategy of the agent to that of human players the better.

Researchers compared the approach against another approach using Monte Carlo Search tree (MCTS). The decision to benchmarks with an MCTS agent came out of pragmatism. It was practical to use because the company already had a working MCTS agent. So it was useful to compare the two to check if the new approach was better than the previous one. The conclusion was that the new approach outperformed the MCTS agent. The CNN agent had more predictive power, execution efficiency and also outperformed humans. Human play testers took around 7 days to complete a level. Instead, the new CNN agent could perform the task in less than a minute which is a huge optimization of time.

Another example of using an ANN for creating a human agent came from the researchers from the University of Essex. Their goal was to create a general game playing AI. They used a CNN paired with a Q-learning algorithm. This solution is called Deep-Q-Network which is a combination between a CNN and Q-learning. The CNN would analyze the image translate it into inputs and make actions. While the Q-learning functioned as a training supervisor for the CNN. Q-learning would analyze the state-action relationship to select what data to feed into the CNN to train, based on previous states.

The researchers decided to use screen capture to receive the inputs of the games, which is a **sub-symbolic** input source. Meaning it is made by constituent entities that are not direct representations of the game state, for example pixels. In contrast to using **symbolic** inputs, which are a representation system in which the atomic constituents are a direct representation of the game state. For example the system recording the player position [5].

The decision came because humans receive most of their information through visual sensors. The reception of the inputs from a visual source is beneficial for a general game playing AI. Because this allows decoupling the algorithm from the game software, in turn increasing its adaptability to other problems.

The method used contains two steps. The first step was to process the image, they shrank each defined block down into one pixel. Then normalized the RGB value and expanded into the smallest size allowed. The modification of the image normalized the inputs. The normalization need came because the CNN structure is the same for all games. Also, the algorithms could get overloaded with the whole information contained in the image.

For the second step the Q-learning algorithm consumes the image as a current state analyzing the reward. The Q-learning will choose the next action to train the CNN based on the Q values. The CNN then receives the input image and generates an action.

Results were positive, the agent was able to learn how to solve both static [3] and stochastic [4] games. The accumulative winning percentage in static games increased the more the agent played. The same applied to the cumulative average score in stochastic games. Suggesting the agent was applying knowledge acquired during the previous plays.

In general after checking these works it seems feasible to use an algorithm from the ANN family. Candy crush is a game played in real time proving that ANN based solutions is capable of fast response times. Regarding the work done on the university of Essex, they demonstrate the ability of a neural network to play not only one game but several. Even more in a learning style close to a human because it is based on vision. While the goal was different, because they wanted to achieve a general perfect-play agent and this work seeks to create a human-like agent.

One of the main decision lies in how the neural network will receive data. As previously mentioned there are 2 approaches to the inputs **symbolic** or **sub-symbolic**.

The main problem of using a neural network is the training phase because this process requires an adequate quantity of data. The quantity of data is difficult to assess but is related to how many scenarios the ANN requires learning. The problem is there is no available data already created like for the candy crush saga example. Data collection requires recording several play-through of the game. These recording are time-consuming.

A **sub-symbolic** approach like screen capturing brings the need of analyzing, compressing and simplifying the video. Otherwise, the amount of data to input into the ANN would be too much hampering the training times. Also, Pic-a-boo is a game played with fog of war. Which does not let you see your character or the enemy character. This means it is an imperfect information game. So a big chunk of information represented in the video will be useless to train requiring more refining efforts.

This fact shift the balance to favor using **symbolic** inputs. A symbolic data model has as much information as required. This means more focused data to train ANN and less refining efforts which translates into faster training times.

2.4 Unity Machine Learning Agents Toolkit

As the previous examples have shown, ANN has seen some advances and usages in recent years. Simulation platforms are one of the many tools that had help in the advancement of ANN and AIs in general. Examples of these platforms are Arcade Learning Environment (ALE) [6], VizDoom [7] and Mujoco [8]. The existence of these environments have been essential because they provide the means to benchmark and test algorithms. Also, these simulation platforms serve not only to enable algorithmic

[3] Game in which a single decision is made by each player, and each player has no knowledge of the decision made by the other players

[4] Game where each player selects an action and receives a payoff, the games moves to a new random state based on the previous state and actions

improvements, but also as a starting point for training models which may be deployed in the real world. A prime example of this is the work being done to train autonomous robots within a simulator, and transfer that model to a real-world robot [9]. In these cases, the simulator provides a safe, controlled, and accelerated training environment.

Pic-a-boo is a game that was developed using the Unity game engine and conveniently Unity provides a good platform for developing AIs. This platform is known as Unity Machine Learning Agents Toolkit (UMLAT). UMLAT is an open source project that enables researchers and developers to create simulated environments using the Unity Editor and interact with them using a Python API. The toolkit is built to take full advantage of the properties of the Unity Engine which makes it a strong research platform [10].

UMLAT provides many resources necessary for creating simulated environments in Unity. It contains two components. First a Software development kit (SDK), which contains all functionality required to create environments within the Unity Editor and associated C# scripts. Second a Python package which enables interfacing from outside of Unity with environments built using the SDK.

UMLAT presents a high-level Application Program Interface (API) for creating AIs using ANNs. This means that designing and implementing an ANN is done through configuring a series of parameters exposed by the API. For instance parameters like: number of layers, numbers of units in layers, input format, etc. UMLAT will read these parameters and create the desired modification in the provided basic ANN architectures. To create these ANN UMLAT uses the TensorFlow python package. TensorFlow is an open-source software from Google used for data-flow and differentiable programming. It is mostly a math library, and one of its main uses is machine learning applications such as neural networks [11].

UMLAT provides a set of baseline algorithms implemented with TensorFlow, which can be used for starting the development of novel algorithms. The toolkit currently provides an implementation of Proximal Policy Optimization (PPO) [12], a state-of-the-art Deep Reinforcement Learning algorithm, with the option to extend it using an Intrinsic Curiosity Module (ICM) [13] and Long Short Term Memory (LSTM) [14]. However, the most relevant of the algorithms for this project is the Behavioral Cloning, a simple Imitation Learning algorithm [15] that was implemented to showcase the Unity Editor as a tool for recording expert demonstrations.

Any scene object within Unity can be made into a learning environment. To create a learning environment is only necessary to include the UMLAT-SDK into the Unity project Once included the SDK provides three entities **agent**, **brain**, and **academy**.

An **agent** object represents an actor in the scene that collects observations and carries out actions. The **academy** object orchestrates agents and their decision-making processes. The **brain** asset is responsible for providing a policy for decision-making for each of its associated agents, a general overview of the toolkit is shown in Figure 2.2.

Given the fact that Pic-a-boo is developed in Unity, this project will use UMLAT due to several reasons:

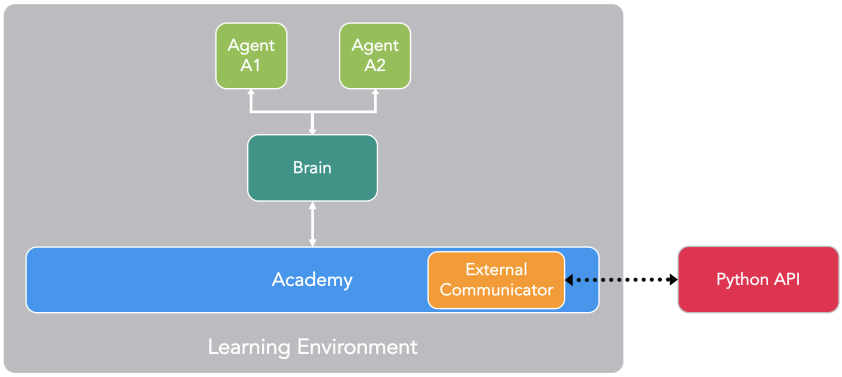

Figure 2.2: Block diagram of ML-Agents toolkit example game.

- Having a Behavioral cloning algorithm already implemented speed-ups the development process.
- Using UMLAT will ensure that the agents can be coupled with the game seamlessly.
- The project is maintained by Unity technologies and has an active community which provides long term support.

2.5 Measuring Artificial Intelligence

2.5.1 Human Tests

One important aspect of this work will be how to test the final results. There is no standard way of measuring human behavior. One good way that has been used before [16] is to ask a human because humans have the innate ability to recognize other humans. In the previous mentioned work [16] the researchers used this method. The researchers presented a video of the resulting agent to various people. The video contained several play-traces of the agents mixed with players. Each participant had to identify between humans and agents.

This test has a close relationship with the Turing test [17] which consist of an imitation game. The idea of the test is that a machine has to try to pretend to be a human. The machine should answer questions put to it. It will only pass if it manages to convince a human that the machine is a human. This human should not have a considerable knowledge of how machine works [17]. By using this kind of tests we can get an idea of the agent quality. However, the test will always be a little biased, because it will depend on the previous knowledge of the participants. Some participants may have knowledge of the game. This knowledge helps to identify quirks in the agent that a normal player will not perform. However, a less expert person may not identify the agents. So previous knowledge is an important factor for these tests.

2.5.2 Confusion Matrix

A more unbiased way of measuring an AI will be using a mathematical model. In [18] the main tool used for evaluating agents is the **confusion matrix** as seen in Figure 2.3. For [18] evaluating a model is the process of understanding how good are the predictions the model makes. A **confusion matrix** is a table of rows and columns that presents predictions versus actual outcomes. It is used to understand how it is performing based on giving the correct answer at the right moment.

	P' (Predicted)	N' (Predicted)
P (Actual)	True Positive	False Negative
N (Actual)	False Positive	True Negative

Figure 2.3: A confusion matrix diagram

.

A binary confusion matrix measures 4 values. To better understand this, imagine having an AI capable of classifying between humans or an AI acting as human the values will be:

- True Positive (TP): a true prediction of a positive outcome, a human is classified as a human.
- False Positive (FP): a false prediction of a positive outcome, an AI is classified as a human.
- True Negative (TN): a true prediction of a negative outcome, an AI is classified as AI.
- False Negative (FN): a false prediction of a negative outcome, a human is classified as AI.

With these values in hand a more detailed analysis on the performance of the model can be made by calculating the following four relationships:

- Precision: The degree to which repeated measurements under the same conditions give us the same results.

$$P = TP/(TP + FP)$$

- Recall: how often we classify an input record as the positive class and it is the correct classification.

$$R = TP/(TP + FN)$$

- Specificity: how often we classify an input record as the negative class and it is the correct classification.

$$R = TN/(TN + FN)$$

- Accuracy: Accuracy is the degree of closeness of measurements of a quantity to that quantity's true value.

$$A = (TP + TN)/(TOTAL)$$

- F1 Score: is the harmonic mean of both the precision and recall measures can be used as an overall score on how well the model is performing.

$$F1S = (2 * (P * R))/(P + R) = 2TP/(2TP + FP + FN)$$

2.5.3 Multi Class Confusion Matrix

The goal of a great number of ANNs models is to learn to map a given input to a corresponding known output. For **prediction** tasks, the output comes in the form of a single label. For **regression** tasks, it is a single value. ANNs are able to solve these simple single-output problems like binary classification with ease. For instance ANNs are able to filter spam in an email system or label images.

However, these single output models are not coping well with the increasing needs of today's complex decision-making. As a result, there is a pressing need for new paradigms. Here, multi-output learning has emerged as a solution. The aim is to simultaneously predict multiple outputs given an input, which means it is possible to solve far more complex decision-making problems.

Pic-a-boo player actions can't be solved with a binary output alone. For instance moving in the game requires 5 different outputs up, down, left, right and neutral. The problem lies in how these outputs are to be measured. A binary confusion matrix is not enough to solve the problem. To solve the measuring problem this work is going to use a multi-class confusion matrix [19].

Binary classification problems usually focus on a positive and a negative class which must be detected. Instead, in a multi-class classification problem, we need to categorize outputs into 1 of N different classes.

For example, we can use an AI trained to predict player movements between left, right or remain neutral. A multi-class confusion matrix will then measure n by n values for this case n=3 that would mean 9 measures which can be observed in Figure 2.4.

With these values an analysis similar to the binary confusion matrix can be made by calculating 3 of the previous relationships as can be seen in Figure 2.5.

After calculating each one of these we can them calculate the mean recall precision and mean f1-score for the model as:

		True/Actual		
		Left	Right	Neutral
Predicted	Left	4	6	3
	Right	1	2	0
	Neutral	1	2	6

Figure 2.4: A multi confusion matrix example using dogs, cats and hens

.

	RECALL	PRECISION	F1SCORE
A	AA/(AA+BA+CA)	AA/(AA+AB+AC)	(2*(AP * AR))/(AP + AR)
B	BB/(AB+BB+CB)	BB/(BA+BB+BC)	(2*(BP * BR))/(BP + BR)
C	CC/(AC+BC+CC)	CC/(CA+CB+CC)	(2*(CP * CR))/(CP + CR)

Figure 2.5: Multi confusion matrix metrics

.

- Mean Precision:

$$MP = (AP + BP + CP)/3$$

- Mean Recall:

$$MR = (AR + BR + CR)/3$$

- Mean F1-Score:

$$MF1S = (AF1S + BF2S + CBF3S)/3$$

The these values are also known as macro-precision, macro-recall and macro-F1-Score, in this work they will be known as the macro-metrics of a confusion matrix. Nonetheless, there is a second set of metrics that can be calculated. These metrics are what we are going to call the micro-metrics, the micro-metrics do not analyze the matrix by class but instead as a whole. However, there is a particularity, in a multi-class confusion matrix when analyzing the whole matrix a TP and TN have the same meaning both are correct predictions. FP and FN also have the same meaning both are incorrect predictions so that means:

- Micro-Precision:

$$MP = (TP)/(TP+FN) = (AA+BB+CC)/(AA+AB+AC+BA+BB+BC+CA+CB+CC)$$

- Micro-Recall:

$$MP = (TP)/(TN+FP) = (AA+BB+CC)/(AA+AB+AC+BA+BB+BC+CA+CB+CC)$$

- Accuracy: Accuracy is the relationship between good predictions and the total amount of predic-

tions

$$A = (MATCHES)/(TOTAL) = (AA+BB+CC)/(AA+AB+AC+BA+BB+BC+CA+CB+CC)$$

- Micro-F1 Score: since micro-precision=micro-recall, they are also equal to their harmonic mean

$$(AA + BB + CC)/(AA + AB + AC + BA + BB + BC + CA + CB + CC)$$

In addition to theses four parameters calculated from the confusion matrix, other metrics exist for instance the hamming loss [20], commonly used with strings. The hamming loss computes the average difference between the predicted and actual output. These differences are calculated by counting the number of different outputs in a set. Now going back to the previous example of an AI trained to predict player movements between left, right or remain neutral, imagine we are comparing two sequences of actions:

- Player: "left, right, left, left, right, neutral, neutral"
- AI prediction: "left, right, right, right, right, neutral, neutral"

Between these sequences there are 2 differences. D will be the sum of all the differences found and N the number of comparisons.

$$HL = D/N$$

So for this example, the hamming loss will be:

$$HL = 2/7 = 0.28$$

Normally hamming loss is used for string however the output of the model can be treated as a string, so hamming loss can be applied.

Another related metric that can be calculated is the percentage of perfect predictions of the model. Meaning that the difference between both actions is 0, For this project this metric will be called **Perfect match**. Z will be the number of zero matches and N will be the total number of sequences.

$$PM = Z/N$$

Going back to the movement predicting AI suppose we have the following set:

- First Player movement: "left, right, left, left, right, neutral"
- First AI prediction: "left, right, right, right, right, neutral"

- Second Player movement: “left, left, neutral, right, right, left”
- Second AI prediction: “left, left, neutral, right, right, left”
- Third Player movement: “left, right, neutral, right, right, left”
- Third AI prediction: “left, right, neutral, right, left, left”

For this case, only the second prediction has a difference of 0 so the **Perfect match** will be:

$$PM = 1/3 = 0.33..$$

With these metrics the models now can be correctly evaluated to check the quality and differences of the generated clones.

2.6 Closing Remarks

To close this chapter a recapitulation can be made with the elements necessary to progress. This project will use the **clone** model that encompasses the creation of an AI by extracting the patterns from human data. To extract these patterns, an ANN will be used which is an algorithm that matches a set of inputs to a set of outputs. The ANN will be using **symbolic** inputs which are a representation system that has a direct meaning without requiring previous interpretation. The project will be implemented using UMLAT to speed-up development and to ease the coupling with Pic-a-boo. Finally, the project will be measured using both a **Confusion Matrix** and its various associated metrics and recognition test with human subjects.

3

Game Description

Contents

3.1 Chapter description

In this chapter Pic-a-boo's rules and components will be detailed to extract the parameters that the agents will follow while playing. The first section gives a general overview of how the game is played. The next section details the different actions that can be performed. Finally, the third section details the map.

3.2 Game concept

Pic-a-boo is a game whose main objective is taking pictures of enemy players. The game is played in a top-down view map which can be seen in Figure 3.1 with a two-dimensional movement. The screen is shared between all players so each player has the same information as their adversaries. Players may move in horizontal, vertical and diagonal directions [1].

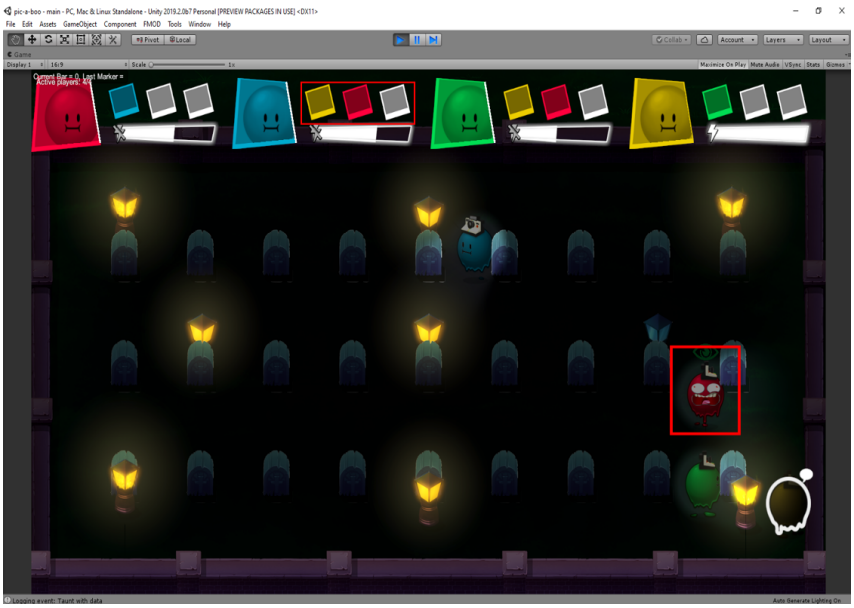

Figure 3.1: Blue player current photograph.

Taking pictures of the player is done by using the flash action near an enemy. Being photographed does not mean you are out of the game. All enemies must be captured to win the game, repeated photos don't count. Taking a photo inside the game means first navigating near the position of your target. Next, with the avatar near the target the player must press the flash button action. The flashlight direction cannot be controlled independently it will point at the last direction that the player moved. The flashlight is represented by a cone of light being emitted from the player avatar which can be seen in Figure 3.2. A target must be inside the cone of light when it is initially pressed. In case the target is

[1] Gameplay videos can be found in here or here

illuminated by the cone but it wasn't present in the initial flash it won't count. Capturing a target with the initial flash will cause the target to enter a flashed state which can be exemplified by the red player in Figure 3.1. In this flashed state the enemy can only move but may not perform any other action.

Given the fact that the map is dark the players can only see themselves near the light emitted by the lamps or after using the flash action. In addition, the player may also use the taunt action to see their player avatar without a light source and to lure the other players.

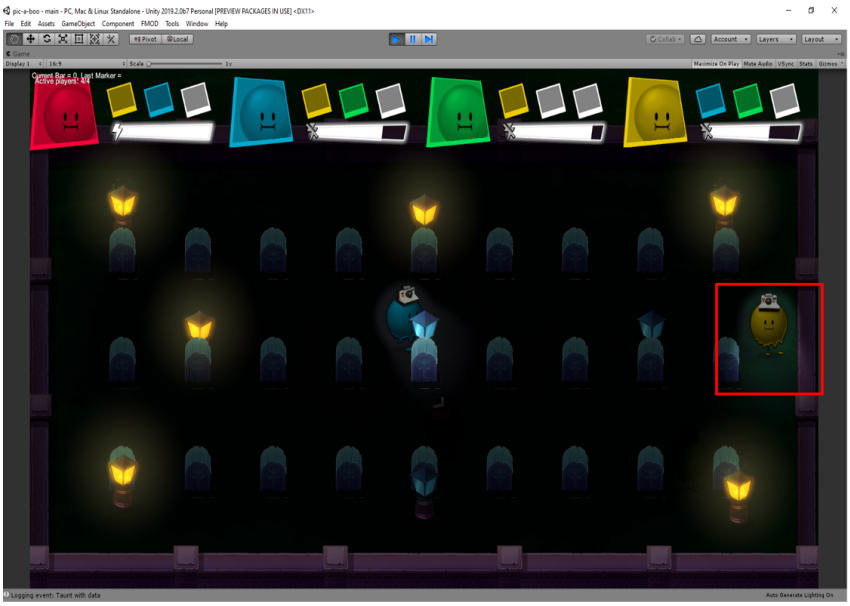

Figure 3.2: Yellow player using its flash.

3.3 Game Actions

The game may be played with either a keyboard or a controller. In game, players can perform 6 actions which are:

- LEFT(A): Starts moving the player's avatar horizontally to the left.
- RIGHT(D): Starts moving the player's avatar horizontally to the right.
- UP(W): Starts moving the player's avatar vertically to the top.
- DOWN(S): Starts moving the player's avatar vertically to the bottom.
- FLASH(G): Turns on the player flashlight.
- TAUNT(T): Highlights the player showing its current position.

The controller button scheme may vary depending on the controller. The vertical and horizontal movement actions can be combined to move in a diagonal. For instance if the player is moving UP and combines it with LEFT the avatar starts moving in a 45 degree diagonal to the north-west. The avatar's velocity is constant so it can go from 0 to maximum speed with just one action. Colliding with a tomb stops the player's avatar, colliding with an enemy causes the avatar to starts pushing the enemy in the same direction the player's avatar is going. The enemy may stop the push by walking in a direction opposite to where he is being pushed.

Flashing allows capturing enemies in a photo, however flashing has a cool-down period of 3 seconds. Following the use of the flash action the bar near the player's portrait will begin to refill itself representing the cool-down period. The player should be able to use flash again once the bar is refilled, these two states can be seen in Figure 3.3.

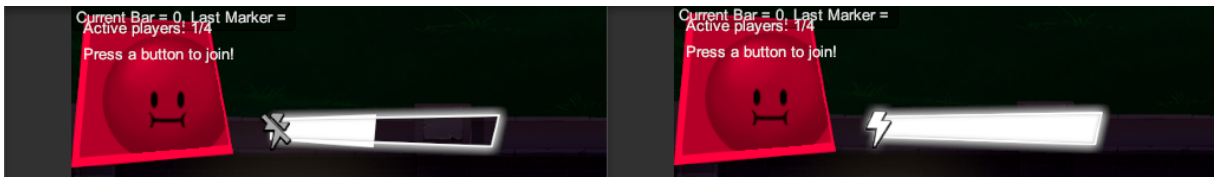

Figure 3.3: Refilling flash bar.

Taunting is an action that serves two functions. The first one is to help the players find themselves the map without resorting to their flashlight. This is because the flashlight has a cool-down. Should the player use the flash to find themselves it will mean losing the chance to take a picture, remaining in a vulnerable state during the cool-down. The second use is to lure enemy players. The idea is to use the taunt action to highlight themselves tempting enemies to go after them. Taunting has no cool-down period and may be used constantly. The appearance of the taunt action can be seen at Figure 3.4.

Figure 3.4: A player using the taunt action.

3.4 Map

Pic-a-boo currently offers 3 maps that are chosen at random before starting a match. The levels are played in a top-down view which can be appreciated in Figure 3.5. A top-down view is achieved by positioning a camera above the map and looking down in a 90 to 45 degrees. Given the type of view, the player navigation is limited to a two-dimensional movement scheme, to be specific vertical, horizontal and diagonal movement.

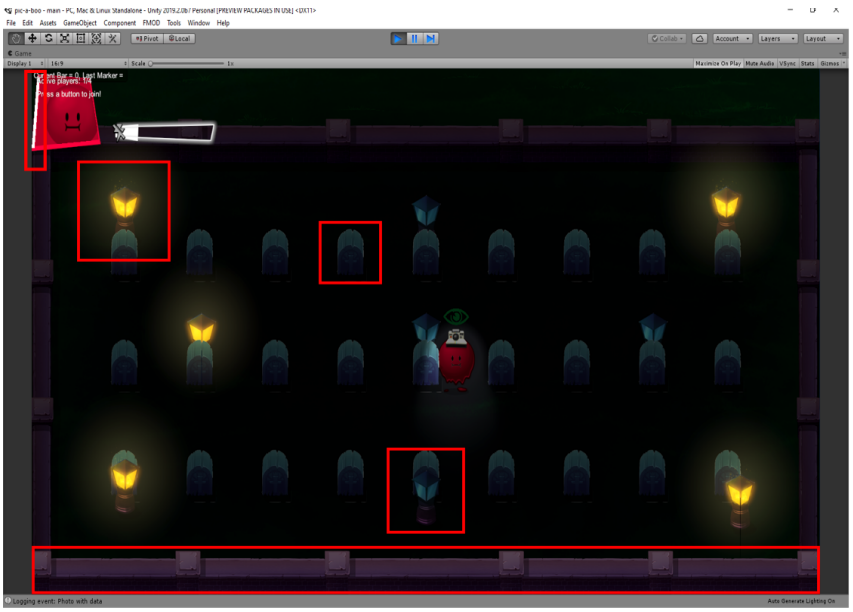

Figure 3.5: Various map elements.

All the players share the same view, so every player can see his own avatar and the avatar of their enemies. The only thing stopping the players from seeing everything on the map is the fog of war presented by the lack of light in the room. Light sources in the map come from 2 sources the first one comes from the lamps that are all around the map. The light emitted by the various lamps is turned on randomly and remains lit a limited amount of time illuminating a small radius around it which can be seen in Figure 3.5. The second light source comes from the player flash which as said before is a cone coming from the player avatar. Regarding collisions the player can collide with the small tombs present around the map (see Figure 3.5). A wall surrounds the playing area preventing players from leaving the game area. At this stage of development there are no moving obstacles in the game. Players can collide between themselves and be pushed around by other player. When a player collide with an obstacle or between each other the UI gets highlighted from the side it collided. Obstacles don't block light so currently players can be photographed over an obstacle.

3.5 Closing remarks

Now to recapitulate the knowledge we need to continue, to interact with the game the agent that is going to be created, needs to be able to perform the 6 actions available in Pic-a-boo namely up, left, right, down, flash and taunt. To be able to do this, these actions are going to represent the output of the ANN, The ANN is going to output an array that should represent these actions which then will be mapped into the game to be executed by the avatar. As for the inputs, there are several things to take into account. As previously said this project is going to use a **symbolic** input so that means data must be extracted directly from the game state. The most basic values needed to make any kind of prediction by the ANN are the position and velocity of each avatar, other values that might be useful are the cool-down of the flash, the visibility of each agent the current state of each lamp and the photos required to win theses will be detailed in the next chapter.

4

Implementation

Contents

4.1 Chapter description

In this chapter, the project implementation will be explained. To better understand the extent of the implementation, it first begins by explaining the two main limitations present during the development phase. Afterwards the training environment used to construct and train the agents will be detailed alongside the reasoning for the choices made. The next section explains the architecture of the ANN that each agent possesses. The fifth section details how the UMLAT handles the training of an ANN. Next how the trained model was introduced into Pic-a-boo. The final two sections are about the procedures that were performed to test the generated agents.

4.2 Limitations

In the development of this project there were certain factors that limited the scope of the study.

4.2.1 Agent training time

Previously it was referred that the most time-consuming part of using an ANN is the training phase. An ANN requires many training iterations to be able to make good predictions. This project used UMLAT to train the agents by reinforcement learning with human players as reference. To train one agent, one player had to play against a set of agent for 10000 iterations which took around 20 minutes in a simplified environment. Some improvements were found passing the 10000 mark, but it is difficult to find subjects willing to take more than 20 minutes training an agent. Supposing that the training time of a clone increases linearly using for instance 100000 steps it would take around 200 minutes to train a clone. So much time is unreasonable for a one sit training session, several sessions would have to be made which is cumbersome. For this reason the iterations were limited to 10000 to be easier to manage.

The training time also limits the quantity of tested architectures, finding a good architecture for an ANN entails a process of trial and error. Testing an architecture implies training agents and checking the results which may take around 20 minutes per architecture. ANN had many possible configurations, for instance number of layers, number of neurons, number of inputs. Each small change required a 20 minutes to test and for that few architectures were tested and when one promising architecture was found the exploration was stopped.

4.2.2 Unity machine learning agents toolkit

Given the fact that this work was using UMLAT version **0.9.3**, the scope of the project solutions was limited to the tools present in UMLAT. UMLAT was also in beta phase so the presence of bugs or outright

broken functionalities was expected. Features not present in UMLAT at the start of the development of the project were not explored. Because fixing or adding a feature implied customizing a very volatile code-base it made the modification difficult to maintain.

4.3 Training environment architecture

The architecture used in this project is based on UMLAT. An overview of the architecture can be found in Figure 4.1.

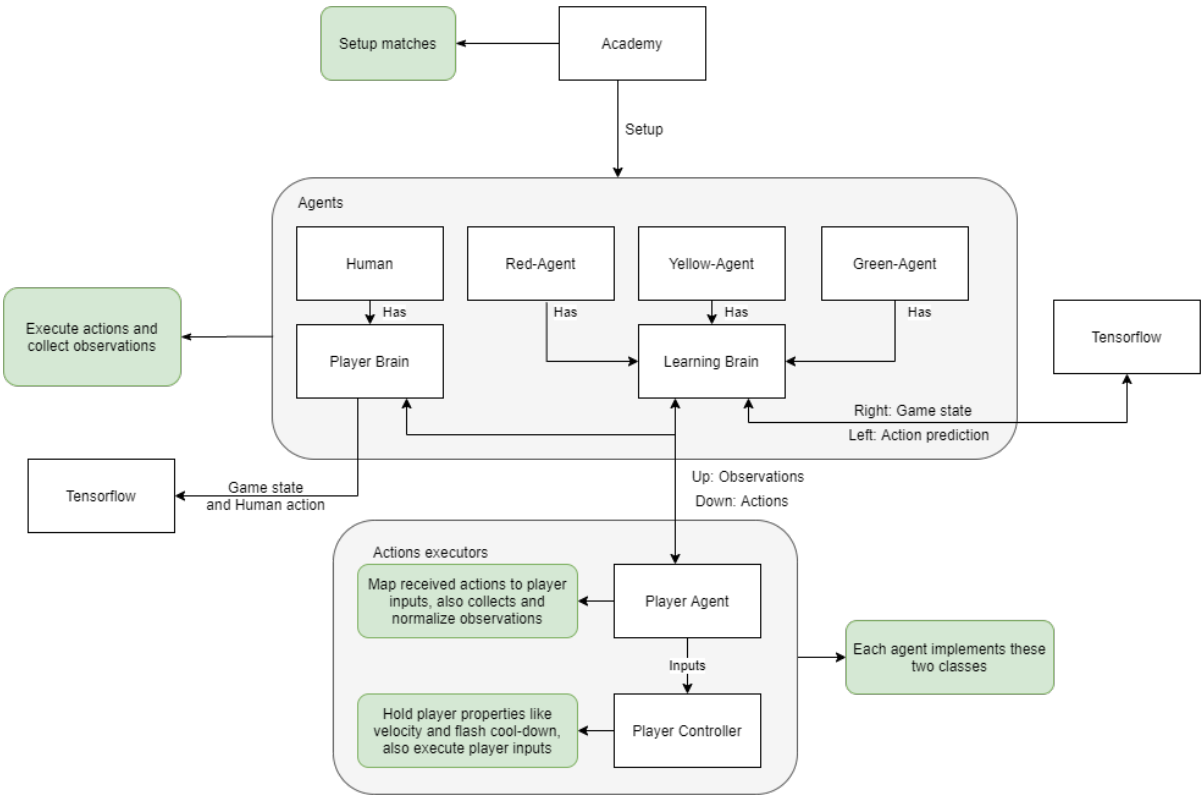

Figure 4.1: Training environment architecture.

This project architecture is centered around three entities namely: **agent**, **brain**, and **academy**. By examining the diagram in Figure 4.1 the first entity encountered from top to bottom is the **academy**. The **academy** is the entity in charge of orchestrating the environment. This means that it should control the flow of each training session alongside with resetting each agent to its original state which mean configuring the agents positions, cool-downs and velocities to their initial values.

Being the entity that orchestrates the whole environment the **academy** was the first to be implemented. However, Pic-a-boo had several steps before a match could begin for example: using the starting menu, mode selection, selecting player color. These steps represented an obstacle for the development. Trying

to implement the academy directly into Pic-a-boo's project structure created unnecessary complexities. This fact led to the creation of a different Unity project structure called Pic-a-boo Agent Gym (PAG). PAG sole objective was to train agents so it didn't need to support the menus and assets present in Pic-a-boo. However, with the creation of PAG came another problem, the trained agents still needed to be used in Pic-a-boo. To ensure this, PAG was created mirroring Pic-a-boo's architecture but eliminating the unnecessary parts and simplifying certain other aspects of the game. Maps size, number of obstacle, position of the several entities and physics properties were copied from Pic-a-boo. However, there were also several other simplifications for this project as to test if a basic solution was viable. The specific simplifications present in PAG were:

- Only one of the three available stages of the game was present.
- Does not include the lamps light.
- No vibration on collision.

To compensate for this when the player collides it is visible for a few seconds. With the creation of PAG the **academy** implementation remained simple because it didn't need to interact with Pic-a-boo setup steps. So the final task for the **academy** were:

- Reposition players to the corner.
- Reset the velocities of the player to zero.
- Reset flash cool-down to zero.
- Set photos counts to zero.

The initial state of PAG can be seen in Figure 4.2.

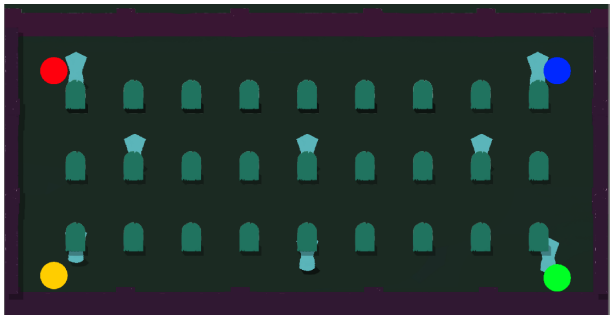

Figure 4.2: Training environment initial setup.

The next entity to be found in the diagram Figure 4.1 is the **agent**. The **agent** entity is in charge of controlling the different playable avatars while collecting observations and receiving actions from the

brain. **Agents** collect information about the current game state in the form of observations and send them to the **brain**. Following this the **brain** sends actions to the **agent** which must be executed in the environment, Given the fact that the **agent** entity was in charge of executing actions, it was attached to the player avatar. The player avatar object is the object that the player uses to interact with the game, the player avatar was copied from Pic-a-boo preserving the same physics properties. There were 4 instances of this object running one for each player, 1 human and 3 agents.
To maintain the separation of concerns the **agent** script was separated in two parts. One was the "PlayerController" script, its objective was to receive a series of inputs and execute them in the game environment. For instance if it received a UP input the "PlayerController" would have pushed the avatar up using Unity physics system. The second script was the "PlayerAgent" script, its objective was to collect the game state and send it to the **brain**. "PlayerAgent" then received an action array that would be mapped to a controller input and be sent to the "PlayerController" for execution. The values that "PlayerAgent" extracted from the game as observations represents the input of the ANN and will be explored in the next section. The meaning of each value in the action array is given by the **brain** entity and will be explained alongside the **brain**.
Finally, the last entity to be implemented was the **brain** which can be seen in the diagram Figure 4.1 below each agent. This entity was the one in charge of receiving the current state of the game in the form of observations and pass these values through the ANN to obtain a prediction. The **brain** UMLAT provides 3 types of brains in the form of an asset:

- Learning brain: used to train a new ANN model or to make predictions once the model is trained.
- Player brain: used to control an agent by receiving humans inputs from a mapped control scheme.
- Heuristic brain: used to implement custom algorithm using the UMLAT interface.

For this project only the learning and player brain were used. A learning **brain** represents a bridge to the ANN model provided by UMLAT in TensorFlow. To construct a **brain** entity the only requirements are to define the inputs and outputs of the ANN model. Inputs are defined through the "Vector Observation Size" and will be detailed alongside the ANN architecture. Outputs are defined through the "Vector Action Space" and represents the actions that the **brain** can take in the form of an N-dimensional float array received by the **agents**. The final parameters for the **brain** were set in UMLAT as:

- Vector Observation Size: 19, it represents the number of collected observations, this will be explained in the next section.
- Space Type: discrete, which means the values outputted did not use decimals.
- Branch size: 4, which means that the float array had a dimension of 4.

- Branch 0 size: 3, the branch 0 controls the horizontal movement, 0.0f means do nothing, 1.0f means right, 2.0f means left
- Branch 1 size: 3, the branch 1 controls the vertical movement, 0.0f means do nothing, 1.0f means up, 2.0f means down.
- Branch 2 size: 2, the branch 2 controls the Flash, 0.0f means do nothing, 1.0f means Flash.
- Branch 3 size: 2, the branch 3 controls the Taunt, 0.0f means do nothing, 1.0f means Taunt.

The final configuration can be observed in Figure 4.4. Following the configuration of the **brain** it produced outputs in a format like this Figure 4.3. For the previous example an **agent** would go left because the first index was 2.0f and up because the second index was 1.0f, When these actions are combined the avatar moves diagonally going to the top-left side of the map. Finally, it will use taunt because the fourth index was one.

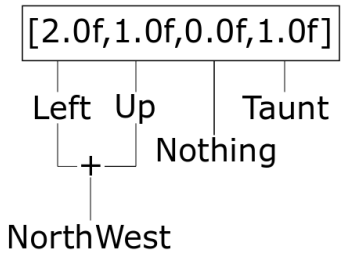

Figure 4.3: This actions means go Northwest and use taunt.

A Player **brain** is the one that a human trying to teach the agent uses, this brain doesn't make predictions the only actions that it emits are the ones provided by a human teaching the agents, this brains only interacts with TensorFlow to serve as reference for comparisons. It Sends observations and actions and receiving nothing from TensorFlow.

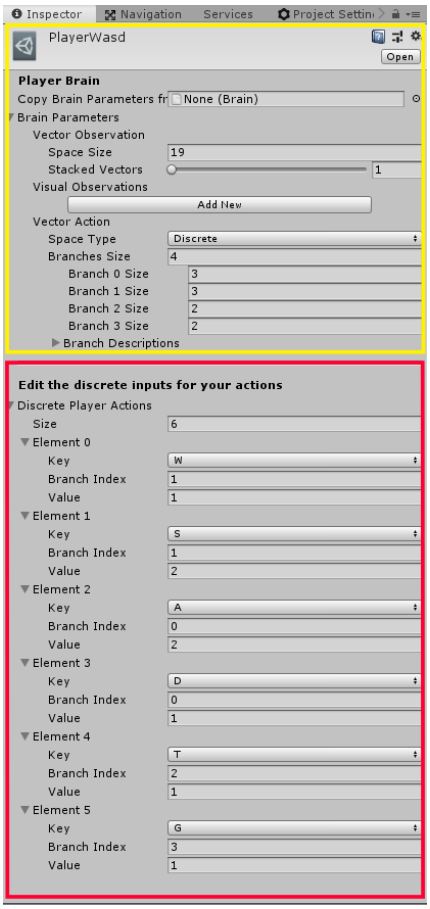

Figure 4.4: Configuration for a player brain.

To resume the general flow of the architecture is as follows, the **academy** sets the environment, each **agent** collects observations and sends them to its respective **brain**. The **brain** uses the observations as inputs for the ANN implemented in Tensorflow and then receives an output in the form of an action array. This action array is then sent back to its respective **agent** which maps the array to an input for the "PlayerController" to execute. This process is repeated each frame until the game is reset either because one of the players won or the time is over. Once the game is finished the **academy** resets the environment and it starts over again. This loop continues until the maximum iterations are reached.

4.4 Neural network architecture

Let's detail the final architecture used for the agents in this project. Several architectures were tested, ones for instance that had the exact same inputs as the players. Others had many more values like flash cool-down and remaining targets, however when tested these alternatives did not produce good results. For more details about the **alternatives** please check Appendix B.1.

As said before, UMLAT provides a high-level API for creating ANNs with Tensorflow. This means that to use and train an ANN from the toolkit it wasn't necessary to define nor implement any of the common steps like activation or optimization functions. Instead, the only requirements were to define the inputs and the topology. Let's begin by exploring the inputs of the implemented ANN. The inputs are represented by the previously mentioned observations that were collected by the **agents** which are also related to the vector observation size defined in the **brain**. In the **brain** this value was defined as 19 and it can be divided into 3 categories which are:

- Position and velocity of each agent meaning two 2-Dimensional vectors for four agents, which adds to sixteen values, the first set of velocity and position correspond to the player's own.
- Position of the last collided object, again a 2-Dimensional vector which adds to two values.
- Visibility of the agent, a boolean, which adds to one value.

UMLAT is capable of training with the values as extracted without any extra treatment, but it is convenient to normalize the values to be between 0 and 1. The normalization helps speed-up the learning process.

Following this, the next step was to define the network topology. This is done through the training configuration files provided by UMLAT in [21]. Parameters like hidden units and number of layers can be set in these files. For this implementation the following parameters were used.

- Trainer: online_bc, use the online training as to play at the same time as the training is done.
- Brain_to_imitate: PlayerWasd, which brain will be performing demonstrations.
- Batch_size: 64, number of experiences in each iteration.
- Time_horizon: 64, how many steps of experience to collect per-agent
- Summary_freq: 1000, how often, in steps, to save training statistics.
- Max_steps: 1.0e4, the maximum number of simulation steps to run during a training session.
- Batches_per_epoch: 10 in imitation learning, the number of batches of training examples to collect before training the model.
- Use_recurrent: false, train using a recurrent neural network.
- Hidden_units: 128, the number of units in the hidden layers of the neural network.
- Learning_rate: 3.0e-4, the initial learning rate for gradient descent.
- Num_layers: 2, the number of hidden layers in the neural network.

Most of the parameters shown above are the default parameters provided by UMLAT except for **Max_Steps**. The **Max_Steps** value was lowered to 1.0e4 from 1.0e5 because as shown in the graph available in the next chapter Figure 5.3 after 10000 steps there were no big improvements in the agent quality. Finally, with these values in place, UMLAT creates a Feed-forward ANN that can be seen at Figure 4.5.

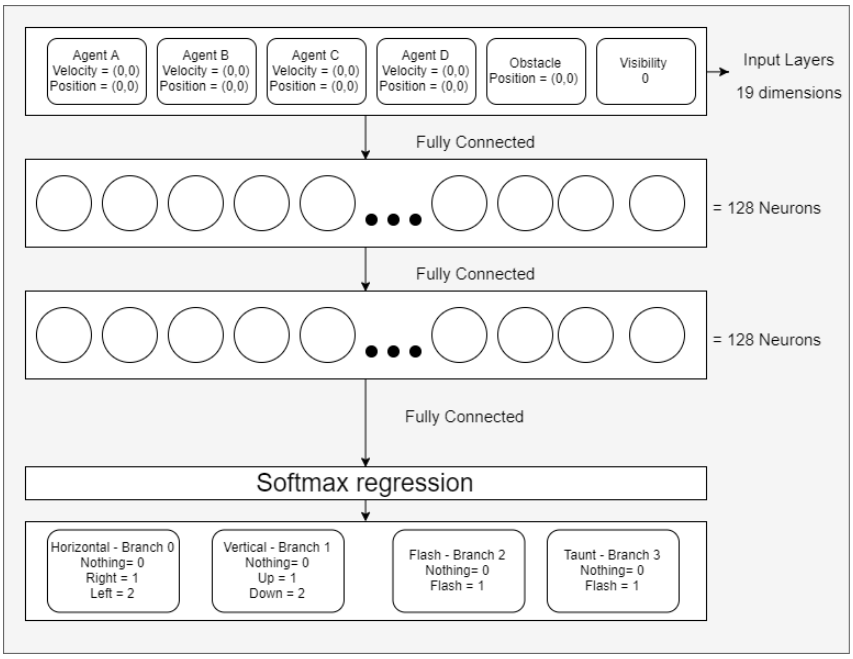

Figure 4.5: Feed-forward Neural Network Topology.

4.5 Neural network training

The training process of the ANN is handled entirely by UMLAT. UMLAT provided two type online training and offline training. To train the ANN this project used the online training mode. In this mode a human must actively play the game against the learning agents. This mode contrast with the offline training mode in the sense that training offline meant recording the game and feeding the results to the ANN. Online training[1] was chosen because it allowed the continuous monitoring of the agents. If left to train alone following the recordings the training iterations took more time because at the beginning the agents aren't able to finish a match.

After setting the training type, how UMLAT handles the training is not documented. However, the code is open-source so it was reverse engineered. A general overview of the training architecture can be seen at Figure 4.6.

The training is done using a python environment with the TensorFlow library. First the parameters defined in the configuration files are received and the topology of the ANN is created. Second the observations stream composed of the previously mentioned positions and velocities reaches the ANN where the stream is multiplied by the weights-matrix and summed by the bias.

```
outputs = linearActivation(inputs * weights-matrix + bias)
```

[1] During development of this project UMLAT decided to deprecate this functionality [22] forcing the project to remain in UMLAT **version 0.9.3**

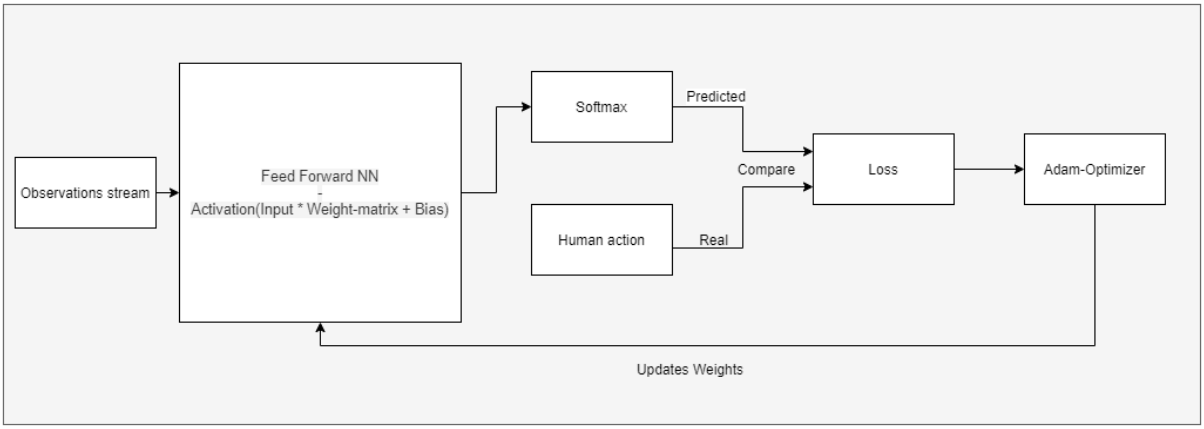

Figure 4.6: Neural Network training architecture.

The values then pass through a linear activation function which controls when the neurons are active. This process is repeated as many times as there are layers. For this project the process was repeated two times because there were only two layers. The final step to obtain the predictions is to pass through a soft max function. The soft max function is going to normalize the result and perform a logistic regression. Logistic regression are used to predict categorical results like the ones in Pic-a-boo, for instance flashing and not flashing [23].

With the prediction finally obtained, the value is then compared to the human **brain** action. From this comparison the loss value is obtained and then the value is passed through an Adam optimizer.

The Adam optimizer is a variant of a stochastic gradient descent function, these functions are used to calculate a local minimum by approximating the gradient descent instead of calculating it with the whole data. This estimation is done by estimating it with part of the data. These functions are commonly used as an optimization algorithm to reduces loss and updates the network weights based on the training data [24].

The process described above is repeated continuously in a training session, so each time the weights are updated, the prediction of the ANN changes to be more similar to the real action of the human because the Adam optimizer reduces the difference between both values.

4.6 Exporting agents to Pic-a-boo

To use the agents inside Pic-a-boo there were two things required. One was the model from PAG produced by UMLAT which is a binary that represents the trained brains. The second was to add UMLAT to Pic-a-boo main project, the process was similar. It was required to include the **Brain**, **Agent** an **Academy** entities. This was an easy process because PAG architecture is a copy of Pic-a-boo, so it was a matter of copying the code. More details of the implementation process can be found in

Appendix B.3. A video of the final result can be seen in here

4.7 Agent quality tests

To test the agents, it was necessary to enter an input to the ANN and get a prediction. The inputs of the ANN are the observations extracted from the game-state as mentioned before. So it was required to save these observations to later input them into the ANN. A small modification was done in the **agent** entity for it to record a player match into a JSON file. The JSON file contained the various observations of the game space through each frame alongside with what actions the human performed based on that observation. The values that the JSON file contained were:

- player: contains an object describing the player position and velocity.
- targets: contains an array objects describing the targets position and velocity.
- isVisible: a flag describing if the player avatar was visible.
- lastObstacleX: describe the horizontal position of the last collision with an object.
- lastObstacleY: describe the horizontal position of the last collision with an object
- horizontalMovement: The output horizontal action made by the player based on the above observation. Zero neutral, one right, two left.
- verticalMovement: The output vertical action made by the player based on the above observation. Zero neutral, one up, two down.
- flash: The output flash action made by the player based on the above observation. Zero neutral, one flash.
- taunt: The output taunt action made by the player based on the above observation. Zero neutral, one taunt.

These values correspond with the inputs and outputs of the ANN. An example file can be seen in Figure 4.7.

Following the recording of various matches, this data was inputted into the ANN to obtain a prediction. This prediction was then compared to the real action that the human did. To compare the predicted actions and real actions, 4 confusion matrices with their respective parameters [2] were calculated. One for each index, horizontal, vertical, flash and taunt. For flash and taunt a binary matrix was used, which can be seen in Figure 4.8 and in Figure 4.9.

[2] sensitivity, specificity, accuracy, precision and f1Score

```
{
  "gameObs": [
    {
      "player": {
        "px": 0.9512168765068054,
        "py": 0.885932207107544,
        "vx": 0.5,
        "vy": 0.5
      },
      "targets": [
        {
          "px": 0.05057943984866142,
          "py": 0.037100136280059814,
          "vx": 0.5,
          "vy": 0.5
        },
        {
          "px": 0.05057943984866142,
          "py": 0.885932207107544,
          "vx": 0.5,
          "vy": 0.5
        },
        {
          "px": 0.9512168765068054,
          "py": 0.027258556336164474,
          "vx": 0.5,
          "vy": 0.5
        }
      ],
      "isVisible": "true",
      "lastObstaclex": 0.5032892227172852,
      "lastObstacley": 0.4455236792564392,
      "horizontalMovement": 0,
      "verticalMovement": 0,
      "flash": 0,
      "taunt": 0
    }
  ]
}
```


Figure 4.7: Saved JSON.

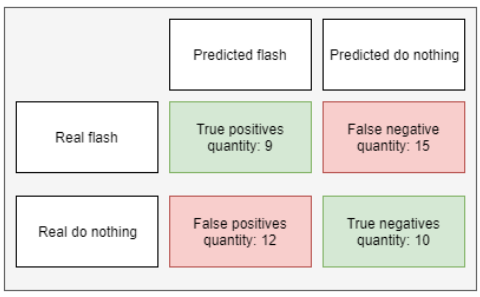

Figure 4.8: Flash matrix example.

In contrast, for the horizontal and vertical movement a multi-class matrix was used. An example of these matrices can be seen in Figure 4.10 and in Figure 4.11.

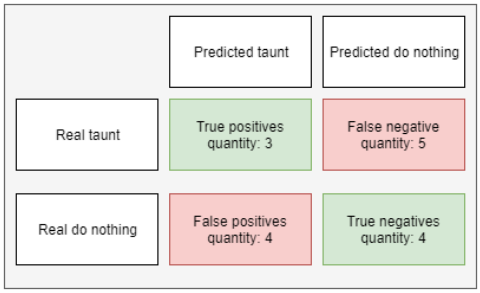

Figure 4.9: Taunt matrix example.

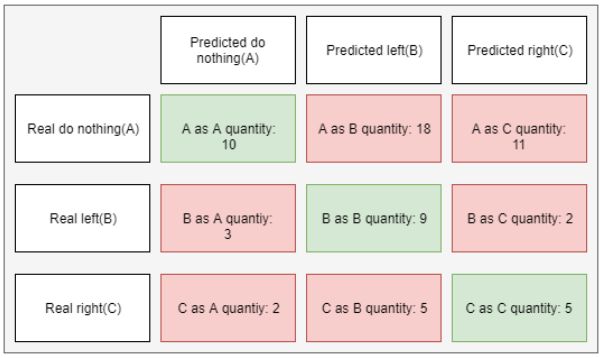

Figure 4.10: Horizontal matrix example.

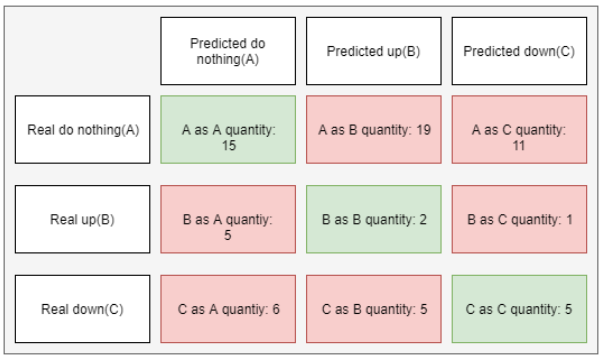

Figure 4.11: Vertical matrix example.

Alongside these matrices two more metrics were calculated the previously mentioned hamming loss and the perfect match. The training and test process was repeated with 6 different people plus the researcher. The data produced from these tests will be explored in the next chapter.

4.8 Human assessment tests

Previously, it was explained that a human-test was required to assess the quality of the agent. For this test, a query was created for the cloned subjects. The query consisted of 4 match videos. In the

matches there were four clones. One clone was the subject clone, another one was a control agent that trained just 10 iterations so it can be considered a random clone, the remaining players were other clones selected randomly. The first video was a training video, all agents inside the video were identified by the color of their avatar in the title of the video. For the next three videos the agents starting position and colors were switched. The query presented then 3 questions.

- What color is your clone?
  - Red
  - Blue
  - Green
  - Yellow
- How confident is your choice?
  - Strongly Agree
  - Agree
  - Partially Agree
  - Disagree
- Why did you choose that clone?

The first two questions were multi-selection and the last one was an open-ended question. This query was made to understand if the clone's owner was capable of differentiating between the others thus implying there are subjective differences between the clones. Being four clones the possibility of choosing randomly and matching the correct clones is feasible. For that reason the confidence and reasoning behind the selection was asked. The results of this query are presented in the next chapter.

4.9 Closing remarks

By using UMLAT as a base, this project managed to rapidly produce a working AI for Pic-a-boo, we could also see that UMLAT's three entity architecture was versatile enough to be applicable for Pic-a-boo. The **brain**, **agent**, and **academy** architecture works very well for AI development because it defines clear objectives for each entity and can serve as a guide in environments outside of unity and UMLAT. However, for all the good that UMLAT brought it also limited the available options of ANN types to what was present in the beta.

5

Results

Contents

5.1 Chapter description

This chapter showcases the results of the several measures done through the development of the project. The results are separated into two sections. The first section showcases the values that lead to the decision of stopping the learning process at 10000 iterations with 2 layers. The next section showcases the results of the cloning capability of the agents.

Throughout the chapter the confusion matrices results are grouped into three related sets. First the taunt and flash results values that correspond to actions that affected visibility and whose occurrence in the data set were rare. Next the Confusion matrices for navigation composed of the horizontal and vertical actions, actions that affected the player position and whose occurrence was common. Finally, the global analysis parameters hamming loss, perfect match and its complement imperfect match, values that were calculated by counting all the predictions made in a complete game session.

All the following graph were generated from the results that can be found in Appendix A, additionally the legends for them can be seen in Table 5.1.

Abbreviate	Name
Rec-A	Recall average
Spec-A	Specificity average
Prec-A	Precision average
Acc-A	Accuracy average
F1S-A	F1Score average
PM-A	Perfect match average
IM-A	Imperfect match average
HL-A	Hamming loss average

Table 5.1: Graph legends

5.2 Learning optimization results

5.2.1 Iteration stopping point

For these measures, three games were recorded and then inputted into the ANNs model manually across various training stages, to be specific 10, 3000, 5000, 10000, 20000. The values present in the following graph are the average of these games. These tests were done with the objective of understanding at what point the agent achieved its best performance to stop the training process. The model used was the 2 layers model presented in Figure 4.5.

5.2.1.A Confusion matrices for flash and taunt

The metrics progression for the flash action can be seen in Figure 5.1 and for the taunt action in Figure 5.2 Specificity starts high and maintains itself high. Both the flash and taunt are actions which rarely occurs in the game, players are normally moving and only flash or taunt when is necessary. Specificity measures the rate of negatives recognition, so the high values are caused by the low occurrence of the flash or taunt. However, the fact that the lines remain around the same values means that there is no large difference between a new agent and an untrained agent regarding when not to flash or taunt.

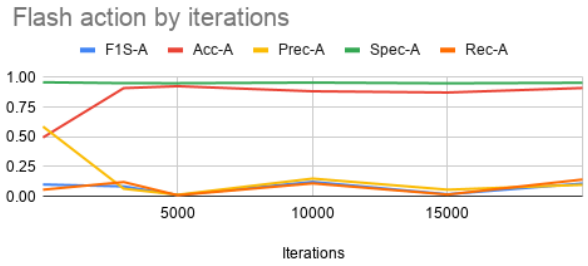

Figure 5.1: Flash action progress.

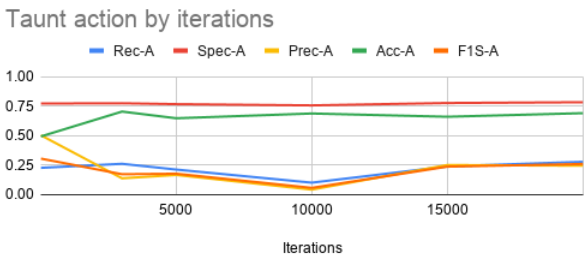

Figure 5.2: Taunt action progress.

Accuracy starts low and rapidly increases showing that the agent is improving the recognition of events. One important aspect to notice is that there are more TN events than TP as you can see in Table A.2, Table A.3, Table A.4 and Table A.7, Table A.8 or Table A.9. This means that the accuracy is going higher not because the agent is improving in using correctly the flash or taunt but instead the agents avoid using it. Meaning the agent doesn't understand how to use the flash correctly to win the game or the taunt to bait the enemies. It just predicts that it shouldn't use it because it's the most common occurrence, getting some correct predictions because it's common not to use a flash or a taunt. As for precision, it starts high and then decreases, mirroring the accuracy. This is because precision measures the prediction of positive values, the flashing and taunting actions are used more frequently in untrained agents. Trained agents begin to understand that the both actions aren't supposed to be used all the time because it's the most common event. This lowers the precision because there are

few flashing or taunting events and for those few events the prediction is incorrect. In other words the untrained agents flashes or taunts frequently guessing the prediction by just trying.
Recall for the flash follows a similar path as the precision differing at the beginning, because at the beginning the agent doesn't known when to hold the actions increasing the FP.
Finally, the F1Score which is a harmonic mean between precision and recall. The F1Score remains lows through the iterations. It shows some progress however it goes up and down almost like a sine function for the flash and a cosine for the taunt. F1Score shows the overall quality of the agent and its results express that the agent does not understand when is the right opportunity to use neither action. The only thing that it is learning is to avoid using taunts or flashes.
The taunt presents lower values in general compared to the flash but each parameter follows a similar trend. This could be because the taunt action is a difficult action to predict because it depends on the ability of a player to find themselves in the map. Human players use taunt normally when they feel lost which makes it difficult to determine a good opportunity for when to taunt. In contrast, the best opportunity for a flash is when there are players nearby.
Nonetheless, the agents are unable to learn how to taunt or flash because both events are rare compared to the navigation events. The erratic behavior for both taunt and flash can be seen clearly while playing, there aren't any patterns in the execution of these actions, and if an agent catches a player with the flash, it was most likely by accident. These predictions are so chaotic that we could substitute them for a simple random algorithm and the result would be similar to the agent constructed in this project. In fact a heuristic approach could be a more robust solution for the flash and taunt action.
It is important to notice that the effect of these two actions in the cloned subjects perception of their clone is not apparent as can be seen in Table A.53, Table A.54, Table A.55. No one mentioned flashes or taunts as the reasoning behind a selection, but various subjects mentioned positioning. So given the fact that the importance of the taunt and flash in clone perception is low and that the predictions are chaotic, these actions weren't a factor in choosing the 10000 mark. However, it is worth noticing that the flash has a peak at 10000,but the same cannot be said for the taunt.

5.2.1.B Confusion matrices for navigation

Now continuing with the navigation. For horizontal action the progress is shown in Figure 5.3 and for the vertical progress in Figure 5.4.
In this case all the metrics are near or beyond the 0.70 mark. This means that the agents have a good understanding on how to navigate both horizontally and vertically compared to the taunt and flash actions. Horizontal and vertical navigation are the most common events in Pic-a-boo. However, horizontal navigation is more common because the map has more width than height forcing the players to navigate more horizontally than vertically. The abundance of events compared to the flash and taunt events

reflects itself in the overall performance of the agent while navigating. Having more events helped the agent to learn how to navigate with consistency.

Any point between 5000 or 10000 could have served as a stopping point for simulation The agent navigation is at its best in this range and decreases if the agent goes beyond that. The 10000 point was chosen because with 5000 the cloned subjects were still getting comfortable with the training experience. Referring back to the human perception, the most mentioned aspect for identifying a clone is also the navigation as seen in Table A.53, Table A.54, Table A.55. This enforces even more the reasoning behind the selection of the 10000 mark as the stopping point favoring navigation over anything else.

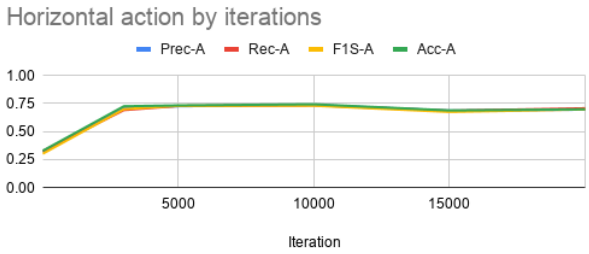

Figure 5.3: Horizontal action progress.

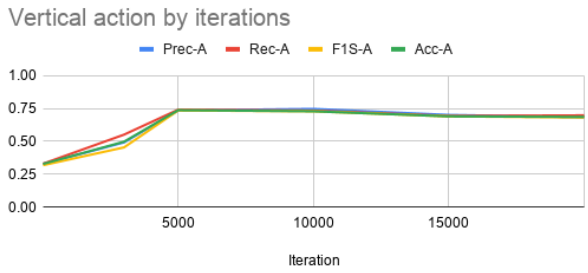

Figure 5.4: Vertical action progress.

5.2.1.C Global analysis parameters

For the global metrics the progress can be seen at Figure 5.5. Beginning by the hamming loss, it dictates how many errors there are in the prediction, it begins high beyond the 2.0 mark and begins to decrease. So initially the agents have in average more than 2 errors, there are only 4 actions so that means the agent if failing predictions constantly. The hamming loss stabilizes near 1.0 around the iteration 5000. Following training the agent commonly fails only 1 predicted action in average. By taking into account the previous graphs it can be assumed that this failed prediction is caused by failing to predict a flash or a taunt.

For the perfect and imperfect match cases the graph mirror each other because these values are complementary and as the agent trains, it is able to make more perfect predictions. However, the agent

is unable to make more perfect predictions than imperfects. Again it can be assumed that the predictions are failing because of the difficulty in predicting the flash or taunt.

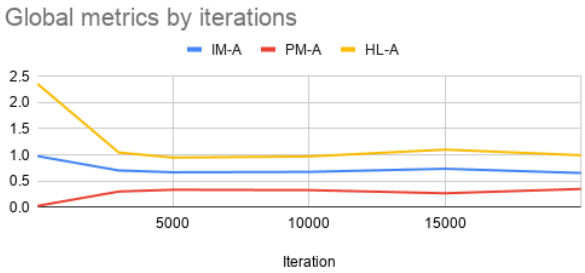

Figure 5.5: Global metrics progress.

5.2.1.D Closing remarks

To conclude, the main idea to extract from here is that for navigation the best performance is achieved in the range between 5000 and 10000 iterations. Having more than 10000 iterations does not improve the agent performance instead it may decrease it. As for the taunt and flash, no matter how many iterations are done the performance is never good. This is because of the scarcity of the events and the difficulty of determining when to use these actions.

5.2.2 Optimal Layer count

For these measures the same three games were inputted into three different ANNs that were similar to the one in Figure 4.5, The main difference being that the hidden layers value was increased from 2 to 4 and 6. All the models were trained for the exact same amount of iterations 10000. The following graphs are the average of the recorded games. The idea was to understand if the performance could be improved by adding more layers.

5.2.2.A Confusion matrices for flash and taunt

In Figure 5.6, Figure 5.7, Figure 5.8 and Figure 5.9 the comparison between the performance of the agents when increasing the number of layers in the model can be seen. Both specificity and accuracy have close values when compared between layers but the 2 layer models has the overall higher values. The biggest difference is present in the f1Score, precision, and recall metrics. For the flash the 2 layer model is clearly superior when compared to the other ones just by checking these three metrics. The same cannot be said for the taunt, for that case the 4 layer model is the superior one. Taunt is the only action where the 2 layer model is not the superior one. However, the impact of the taunt action in the game is minimal, there is no clear moment when it should be used. As said before taunt is used normally when human players are feeling lost in the game. However, in general the agents are still bad at understanding when to flash or taunt, increasing layers did not improve the performance in any significant way. All these models were trained for 10000 iterations.

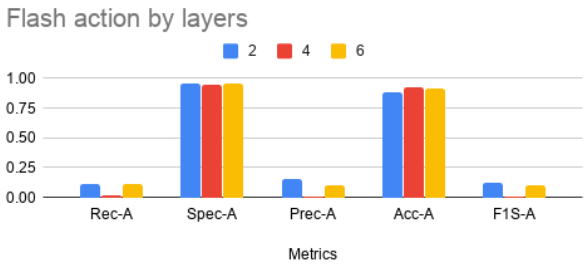

Figure 5.6: Flash action by layers average.

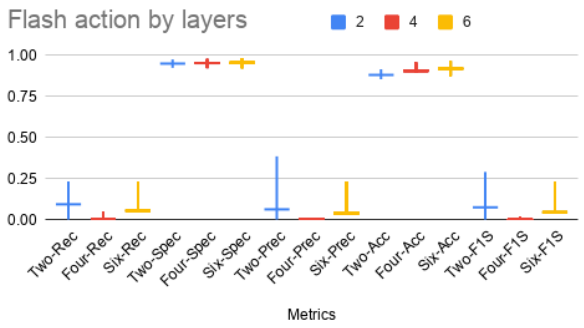

Figure 5.7: Flash action by layers box plot.

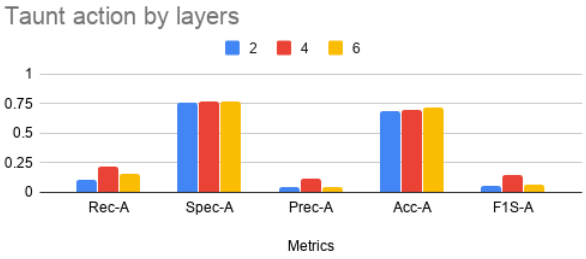

Figure 5.8: Taunt action by layers average.

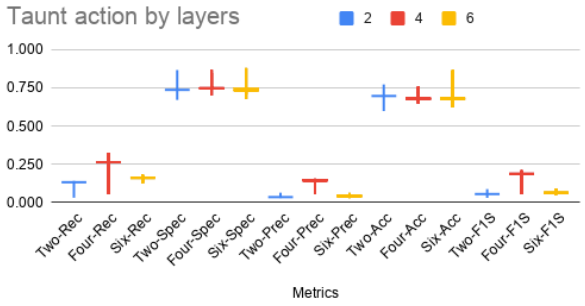

Figure 5.9: Taunt action by layers box plot.

5.2.2.B Confusion matrices for navigation

For the navigation, the comparison is shown in Figure 5.10, Figure 5.11, Figure 5.12 and Figure 5.13. In this comparison all the metrics are very close to each other, near or beyond the 0.70 mark. So no matter how many layers were used the agents still navigated more or less the same. However, the 2 layer model also has the biggest performance peaks compared to the other ones. Nonetheless, the difference is not enough to be significant especially when compared to the 6 layer models. Being marginally the highest performing model and also the fastest to train the 2 layer was selected to create the agents.

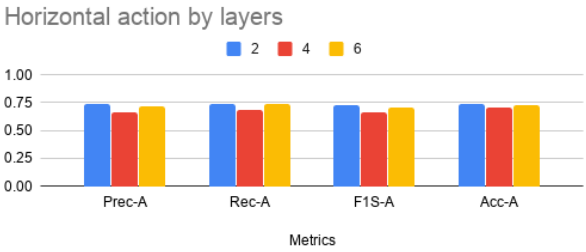

Figure 5.10: Horizontal action by layers average.

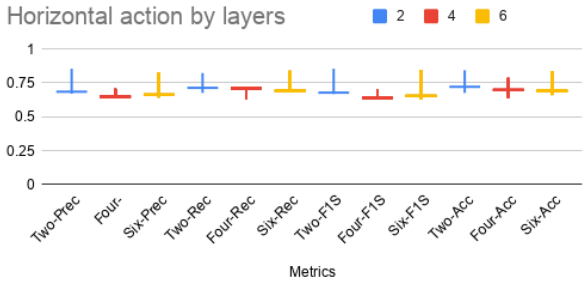

Figure 5.11: Horizontal action by layers box plot.

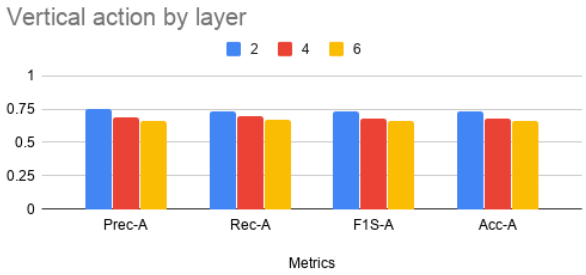

Figure 5.12: Vertical action by layers average.

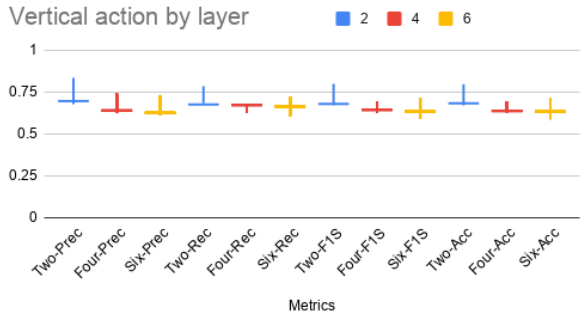

Figure 5.13: Vertical action by layers box plot.

5.2.2.C Global analysis parameters

The global metrics comparisons can be seen at Figure 5.14 and Figure 5.15. In this case the values are again very close. The 2 layer model wins in the hamming loss because is the lowest. However, for the perfect match metric the 4 layer models is a bit better. Still, this difference did not justify increasing the quantity of layers.

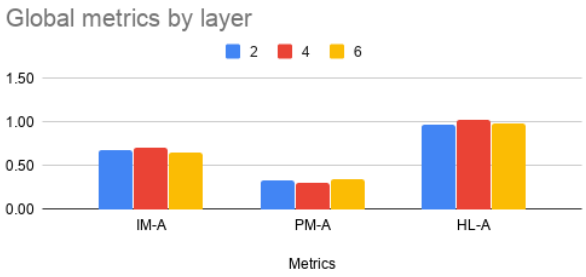

Figure 5.14: Global metrics by layers average.

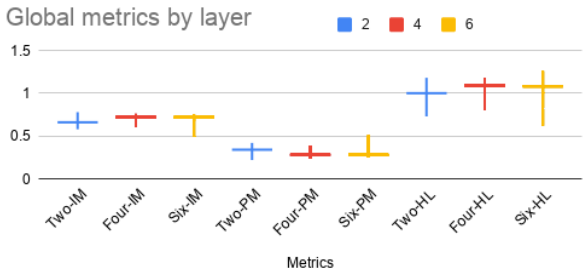

Figure 5.15: Global metrics by layers box plot.

5.2.2.D Closing remarks

The idea that can be extracted from these measures is simple, increasing layers does not modify significantly the performance of the agent. Given this fact the layer quantity was set to 2 because the fewer layers the model had, the faster it trained.

5.3 Imitation quality results

For this test, 6 extra agents were trained by different human subjects, the agents were code named **AR, BH, MF, TS, VC and VG**. the seventh agent code named **EF** is the agent trained by this project main researcher. Each subject trained an ANN by playing against 3 agents continuously for 10000 iterations which took around 20 minutes. In the first part all the agents try to predict the main researcher game.

For the second part the cloned subjects were asked to identify their clones inside a match while three other different clones were present.

5.3.1 Cross agent comparisons

For this subsection, previously recorded games were inputted into each of the 7 agents. These games were all played and recorded by this project main researcher as reference, and were different from the games used for training the agents. The objective of this test was to check whether a clone was capable of predicting another person game. If clones are truly different they should have different predictions, because of the different play-style of each person. Furthermore, someone's clone should be the best predictor of that person's clone.

In this case the new agents namely **AR, BH, MF, TS, VC and VG** were trying to predict the main researcher's games. The **EF** agent which is the researcher's clone served as a reference for the measures. To be able to support the previous statements, **AR, BH, MF, TS, VC and VG** predictions should be different from each other, and **EF** should be the overall best predictor.

5.3.1.A Confusion matrices for flash and taunt

We will begin with the flash and taunt action found in Figure 5.16, Figure 5.17, Figure 5.18 and Figure 5.19. The graphs show differences between each clone showcasing that some clones are better predictors. Nonetheless, the graph for these actions are very spread and chaotic as shown in Figure 5.17 and Figure 5.19. In the flash graph the **EF** clone is one of the best predictors having the highest average and peaks but with medians close to **AR**. Both the flash and taunt graph validate the idea that clones are indeed different, However the idea that the clone of one person is the best at predicting that person game may be challenged if we check the taunt graph.

For the taunt case **EF** predicting **EF** is one of the worst predictions. It is important to remember that the taunt action is not the most consistent action as shown in Figure 5.19 the taunt values are very spread. For example the previous iteration graph presented in Figure 5.2 suggested that an agent with no training was capable of predicting the taunt better than a trained agent. So the explanation for this result where **MF** is predicting **EF** is the same as before, the **MF** is trying to taunt much more than the **EF** brain. This explanation can be further supported by the **MF**'s accuracy which is lower than **EF**'s, the best values for accuracy in **MF** are the worst for **EF** as seen in Figure 5.19.

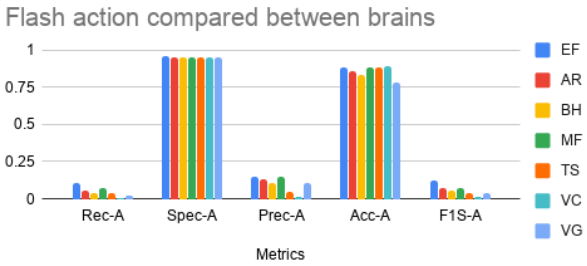

Figure 5.16: Flash action by brains average.

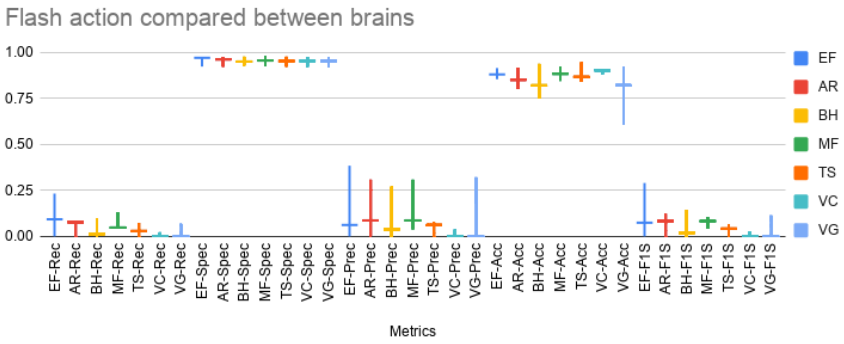

Figure 5.17: Flash action by brains box plot.

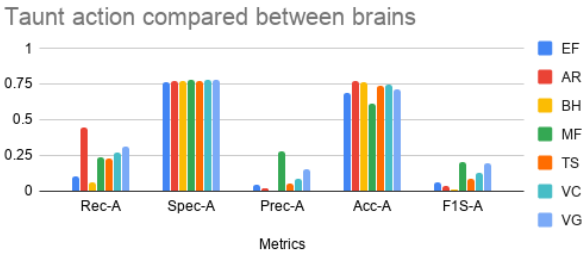

Figure 5.18: Taunt action by brains average.

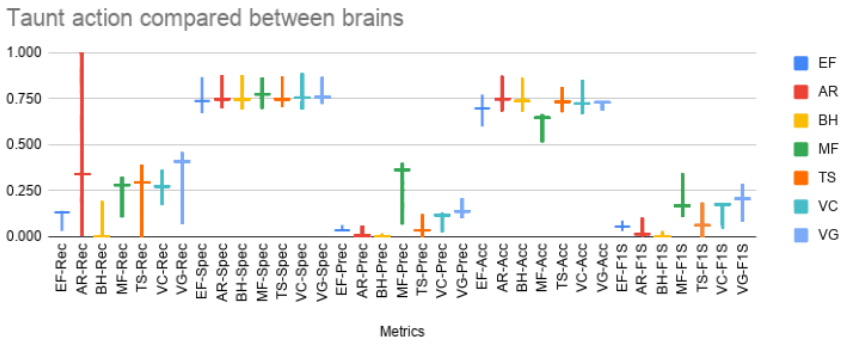

Figure 5.19: Taunt action by brains box plot.

5.3.1.B Confusion matrices for navigation

For the navigation, the comparisons are shown in Figure 5.20, Figure 5.21, Figure 5.22 and Figure 5.23. The values are close between each agent, near or beyond the 0.70 mark. But there are clear differences between each other. This supports the idea that each clone is different. For the idea that the clone of the person is the best at predicting that person action. **EF** predicting **EF** is one of the best performing brain as shown by its average being among the highest, However is surpassed by **AR**, **BH**, **MF** in some peaks as shown in Figure 5.21 and Figure 5.23. Nonetheless, **EF**'s values are more consistent than **AR**, **BH** and **MF**. Meaning that while other brains are able to predict **EF** their values are more spread supporting the idea that **EF** is the best predictor of **EF**. However, It is important to remark that overall differences are not big enough and **EF** is barely ahead of others. Suggesting that while some brains might be better predictors than others it is difficult to differentiate between each clone navigation pattern. The navigation patterns similarities could be explained by the influences coming from the map layout, the map is static and has more width than height, forcing the players to move horizontally most of the time to reach their objectives.

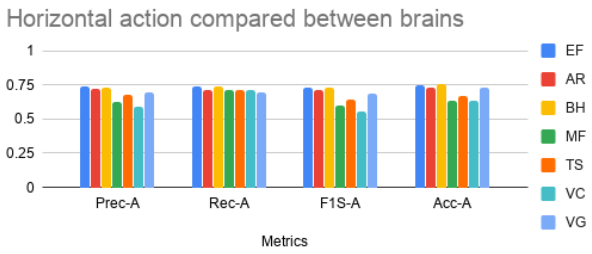

Figure 5.20: Horizontal action by brains average.

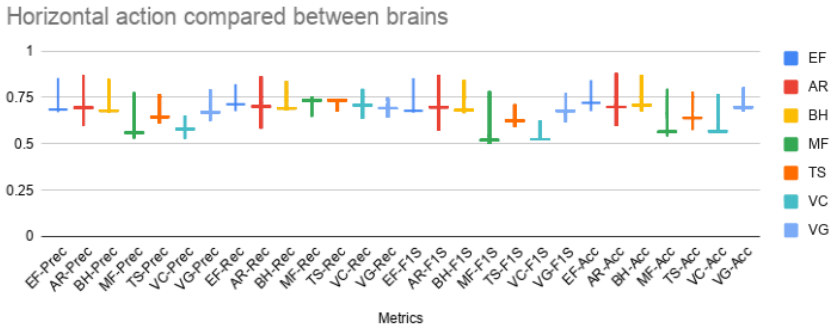

Figure 5.21: Horizontal action by brains box plot.

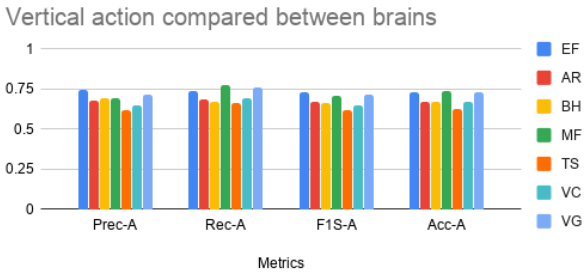

Figure 5.22: Vertical action by brains average.

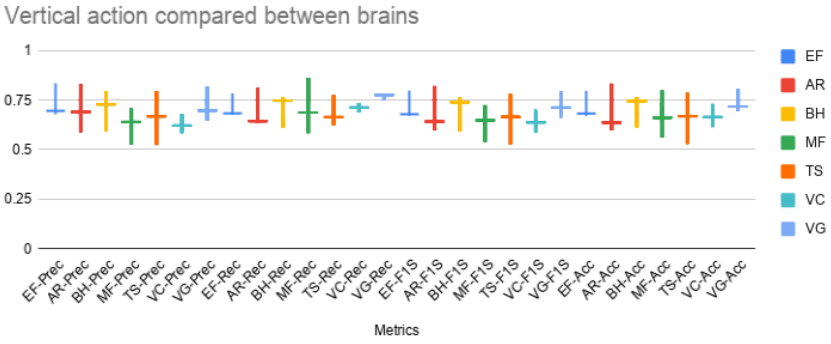

Figure 5.23: Vertical action by brains box plot.

5.3.1.C Global analysis parameters

The global metrics comparisons can be seen in Figure 5.24 and Figure 5.25. For this case, **BH** is the best brain, compared to **EF** in all metrics. Both hamming loss and imperfect matches are lower than **EF**. The **BH** brain was one of the closest to the **EF** brain in the previous actions. So it is no surprise that it is able to make good predictions. Nonetheless, the difference between **EF** and **BH** prediction is not big.

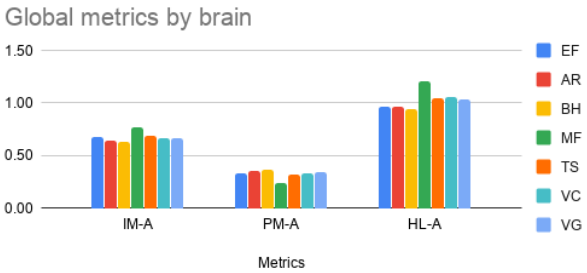

Figure 5.24: Global metrics by brain average.

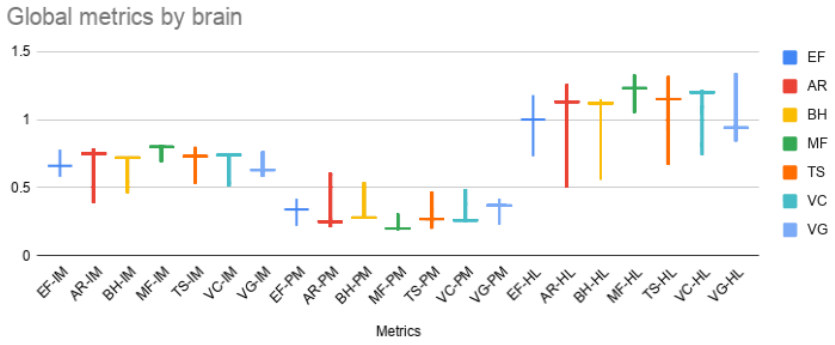

Figure 5.25: Global metrics by brain box plot.

5.3.1.D Closing remarks

The main idea to extract from these tests is that the clones are very similar. To the point that a brain not trained by the user is able to predict the user's games, suggesting that there is no different-play style. However, when taking into account all the tests as a whole, the **EF** brain is one of the best performing and most consistent brains. For the cases where it is not the best, it is one of the best. In comparison, the brains that were able to surpass **EF** in one type of actions is not able to surpass it in another. For instance **MF** surpass **EF** metrics in some values of taunt and vertical actions but falls in the horizontal and global metrics. So the overall best and most consistent predictor of a person's game is its own clone.

5.3.2 Survey

In this section, the results from the questionnaires are reviewed. The idea was to support the evidence found through the quantitative tests done previously. To that goal the 6 previously cloned subjects were asked to recognize themselves in three different games. A fourth game was also presented as an example. The fourth game has all the clones identified to help the subjects understand the idea of the survey and the patterns.

5.3.2.A Survey recognition results

In Figure 5.26, the total results of the queries can be seen. There were a total of 18 recognition, 3 games each person multiplied by 6 subjects. Of those 18, 10 were correct and 8 were incorrect. Meaning that barely more than half of the recognition attempts were correct. However, this value is better than if answer were chosen randomly which would give a 25% chance of answering correctly or 33% if the control agent is discounted. This may seem to validate the idea than the agents are indeed playing differently or at least there is something that can be recognized. However, being so close also means

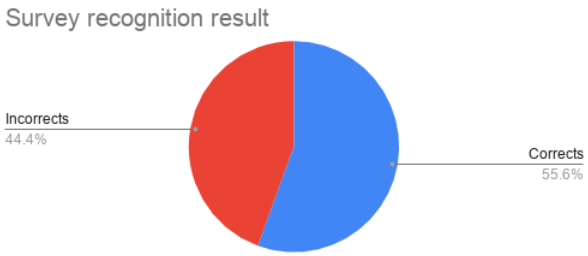

Figure 5.26: Survey recognition result.

that the difference is not enough to be distinguished consistently.

5.3.2.B Recognition confidence

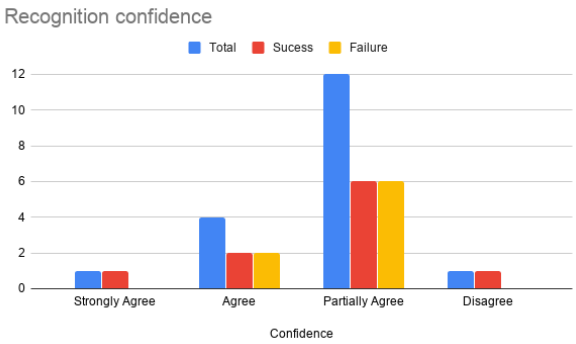

Figure 5.27: Survey confidence result.

In the survey, the following affirmation was present: “I am confident in my choice”, the subjects could answer either, “Strongly Agree”, “Agree”, “Partially Agree”, “Disagree”. The idea was to understand the confidence of the subjects while answering. This is because they only have 4 choices, meaning that they had a certain probability of answering correctly by chance. The graph shown in Figure 5.27 shows that the subjects were in doubt. And that doubt was reflected in the recognition attempt because both “Agree” and “Partially Agree” successful and failed recognition are equal. However, only one of the subjects completely disagreed and that subject successfully identified his clone. Which shows that even in doubt the subjects perceived some differences or quirks in the clones that helped them differentiate between the clones, but not enough to for them to be sure. It is important to remark that the subjects were not experienced with the game in general nor with the clones resulting behaviors apart from watching the clones in videos, which may impact in the confidence of answers. As said before previous knowledge is an important factor in recognizing an AI.

5.3.2.C Recognition reasoning

To better understand the reasoning behind the choices of the subjects the survey asked to explain why the selection was made. The results can be seen in Appendix A.4. One common explanation found was that people compared the clones presented in the games to the demo one. This suggests that rather than identifying themselves, the subjects were identifying the pattern the clone had. However, there are some answers that reveal that part of the subjects were identifying their own quirks for instance “Seems a bit erratic at first.. just like me!” and “I was often on the corners”. This again enforces the idea that clones are indeed different, however the differences are not easily perceived.

5.4 Closing remarks

The results shown throughout this chapter lead to the conclusion that this project successfully created agents capable of playing Pic-a-boo. The agents were created by imitating the playing patterns of human subjects. However, agents are not able to reproduce all the actions successfully. The agents are only able to navigate the map and can’t use the flash or taunt actions correctly. Alongside this, the navigational patterns produced by different agents are quite similar, something players could notice because even the cloned subject had difficulty identifying their own clones.

6

Conclusion

Contents

Pic-a-boo is a game currently in development that, before this project, was only playable with other humans. This was cumbersome because to obtain the complete game experience of 4 players playing against each other, a player was required to find 3 available players. For a quick game session this might not be feasible in some cases. The developers of Pic-a-boo had the challenge of creating an artificial player for the game.

A match of Pic-a-boo centers around taking pictures between players in a dark room. There are four players in any given game and each one must take pictures of the other. The game is played in a top-down map and all the players share the same screen thus all player possess the same information. This game involve guessing because of the darkness in the room, players are unable to see their avatar while playing, only near light sources. However, if a player is able to see its avatar the other players are also able to see it giving away his position with the repercussion that the player might be photographed. Being photographed does not mean that the player is out of the game, but it means that an adversary is closer to winning. Given the fact that so much guesswork is involved creating a fair artificial player for this game is difficult.

This project sought to solve the problem of creating an artificial player for Pic-a-boo through the use of an ANN. The idea was that an ANN could be trained to produce actions similar to a human. The ANN would be trained using **symbolic** observations of the game environment for it to produce actions the same way a human would do.

This project manages to do that thanks to the three entity architecture of **Academy, Brain and Agent** alongside a feed-forward ANN, that received observations of the state of the games and sent the actions to perform to each agent. A simplified environment was created to speed-up the development time avoiding heavy modifications into the already implemented game. This environment called PAG allowed the training of several agents that were created to imitate six different subjects. The AI algorithm used to create these clone was a feed-forward ANN created by using the API provided by UMLAT.

To assess the quality of the resulting agent, this project used one of the most common tools to evaluate ANN classifiers, the confusion matrix. The confusion matrix allowed the measurement of the agents after training to check their progress, to compare agents between subjects and to choose the best configurations for the clones. In general, the resulting agents are acceptable in their play style, the strong point of the agents is navigation which is a big part of what Pic-a-boo requires. However, the capacity for flashing and taunting is quite limited because these events are not as common in the game and so the agents are not able to learn these actions. Another weak point of the agent is to understand the sequence of flashing which means it doesn't try to flash the targets in the sequence required to win, instead it favors the nearest target. This could be solved by using a recurrent ANN, which was one of the desired options to explore but UMLAT it is still in beta and the functionality at the time of writing was bugged. Nonetheless, the resulting clones are still tolerable and provide a good challenge for players.

In addition to the previously mentioned weak points, cloning different subjects lead to similar results. Even the cloned subjects had a hard time recognizing their own clones when asked to identify them. This fact was confirmed when the clone of subject A was used to predict subject B game trace. The clone of subject A was able to predict a great part of subject B game, nonetheless, the subject A clone is normally the best predictor of subject A game.

6.1 Future Works

- **Recurrent neural network**: The recurrent neural network architecture seemed like a natural fit for this case scenario. This project was using UMLAT for its implementation and the toolkit already possess the capacity of using this kind of ANN. However, the recurrent functionality wasn't working at the time of development. The recurrent architecture allows ANNs to work with sequence of inputs. This could benefit the clones because they would be able to understand the sequence for taking pictures. Understanding sequences may improve the agent flashing ability, potentially allowing it to win in a more consistent manner.

- **Different map training**: This project was limited to one of the simplest map of Pic-a-boo, as of the time of writing there are 3 maps. The agents should be tested to check whether they work properly inside the other maps, and if not train the same agent to be able to behave appropriately. Alternatively, new agents could be trained to be specialized in each map. One interesting scenario worth exploring is if Pic-a-boo implements dynamic maps, which means that a map possess moving obstacles, this would require changing the input layers of the agents to understand the changes in the map.

- **Offline training**: The training time was limited because our subjects did not have too much time to spare, it could be beneficial to increase the iterations. In this project increasing the iterations did not produce a big enough difference to warrant increasing it. However increasing the iterations 10 times may produce an improvement in the flashing action, because there would be more examples of the flashing event. Nonetheless, it is not practical to ask a person to train an agent for hours, so a good solution would be to use UMLAT offline training mode. This mode requires already recorded games to train, if Pic-a-boo is eventually released, this data could be available allowing a more extended recorded period.

- **Separating navigation from flash and taunt**: The agent produced by this project was unable to imitate neither the flash nor the taunt actions. These two actions are too scarce compared to the navigational actions, so the ANN wasn't able to understand a good pattern for reproducing these events with all the noise produced by the navigation. A future approach for this problem might be

the creation of a ANN specialized model that only handles the flash and taunt. By Being more specific the ANN might be able to learn how to imitate these actions correctly. In alternative if this doesn't work an algorithm that makes an heuristic prediction might be able to supplant these behaviors.

Bibliography

[1] D. L. Georgios N. Yannakakis, Pieter Spronck. (2013) Player modeling. [Online]. Available: http://yannakakis.net/wp-content/uploads/2013/08/pm_submitted_final.pdf

[2] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis, "Personas versus clones for player decision modeling," in *ICEC*, 2014.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[4] S. F. Gudmundsson, P. Eisen, E. Poromaa, A. Nodet, S. Purmonen, B. Kozakowski, R. Meurling, and L. Cao, "Human-like playtesting with deep learning," *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, 2018.

[5] K. Kunanusont, S. M. Lucas, and D. P. Liebana, "General video game ai: Learning from screen capture," *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2078–2085, 2017.

[6] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents (extended abstract)," *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, 2013.

[7] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaskowski, "Vizdoom: A doom-based ai research platform for visual reinforcement learning," *Journal of Artificial Intelligence Research*, 2016.

[8] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," *2012 IEEE/RSJ International Conference*, 2012.

[9] M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning dexterous in-hand manipulation," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1808.00177

[10] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *ArXiv*, vol. abs/1809.02627, 2018.

[11] Tensorflow. (2019) Tensorflow: Open source machine learning. [Online]. Available: https://www.youtube.com/watch?reload=9&v=oZikw5k_2FM

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint*, 2017. [Online]. Available: https://arxiv.org/pdf/1707.06347.pdf

[13] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," *arXiv preprint*, 2017. [Online]. Available: https://arxiv.org/pdf/1705.05363.pdf

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[15] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 21:1–21:35, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3054912

[16] A. Khalifa, A. Isaksen, J. Togelius, and A. Nealen, "Modifying mcts for human-like general video game playing," in *IJCAI*, 2016.

[17] D. Proudfoot, "Anthropomorphism and ai: Turing s much misunderstood imitation game," *ScienceDirect*, 2011.

[18] J. Patterson and A. Gibson, *Deep Learning A Practitioner's Approach*, 1st ed. USA, 1005 Gravenstein Highway North,Sebastopol, CA 95472: O'Reilly, 2017.

[19] B. Shmueli. (2019) Multi-class metrics made simple, part i: Precision and recall. [Online]. Available: https://towardsdatascience.com/multi-class-metrics-made-simple-part-i-precision-and-recall-9250280bddc2

[20] D. Xu, Y. Shi, I. W. Tsang, Y.-S. Ong, C. Gong, and X. Shen, "A survey on multi-output learning," *arXiv preprint*, 2019. [Online]. Available: https://arxiv.org/pdf/1901.00248.pdf

[21] U. Technologies. (2019) Training config files. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/164d1ab98efc620b2e8c18e680e5fc99c19d69f1/docs/Training-ML-Agents.md#training-config-file

[22] ——. (2019) In game traning with 0.11. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/issues/2901

[23] C. Bishop, *Pattern Recognition and Machine Learning*. Springer Science+Business Media, LLC, 2006.

[24] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint*, 2015. [Online]. Available: https://arxiv.org/pdf/1412.6980.pdf

[25] U. Technologies. (2019) Environment unable to start when setting use recurrent to true. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/issues/3002

A

Measurements

A.1 Confusion Matrices

A.1.1 Binary matrix Legend

Abbreviate	Name
TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative
Rec	Recall
Spec	Specificity
Prec	Precision
Acc	Accuracy
F1S	F1Score
SD	Standard Deviation
A	Average

Table A.1: Binary matrices abbreviate legend

A.1.2 Flash Confusion Matrices

Layers	Iteration	Brain	TP	FP	TN	FN	Rec	Spec	Prec	Acc	F1S
2	10	EF	49	32	466	491	0.09	0.94	0.60	0.50	0.16
2	3000	EF	3	78	919	38	0.07	0.92	0.04	0.89	0.05
2	5000	EF	0	81	949	8	0.00	0.92	0.00	0.91	0.00
2	10000	EF	5	76	908	49	0.09	0.92	0.06	0.88	0.07
2	15000	EF	1	80	916	41	0.02	0.92	0.01	0.88	0.02
2	20000	EF	11	70	923	34	0.24	0.93	0.14	0.90	0.17
4	10000	EF	1	80	938	19	0.05	0.92	0.01	0.90	0.02
6	10000	EF	3	78	904	53	0.05	0.92	0.04	0.87	0.04
2	10000	AR	7	74	876	81	0.08	0.92	0.09	0.85	0.08
2	10000	BH	22	59	757	200	0.10	0.93	0.27	0.75	0.15
2	10000	MF	7	74	910	47	0.13	0.92	0.09	0.88	0.10
2	10000	TS	5	76	894	63	0.07	0.92	0.06	0.87	0.07
2	10000	VC	0	81	936	21	0.00	0.92	0.00	0.90	0.00
2	10000	VG	26	55	607	350	0.07	0.92	0.32	0.61	0.11

Table A.2: Flash confusion matrix game 1

Layers	Iteration	Brain	TP	FP	TN	FN	Rec	Spec	Prec	Acc	F1S
2	10	EF	14	12	269	278	0.05	0.96	0.54	0.49	0.09
2	3000	EF	0	26	497	50	0.00	0.95	0.00	0.87	0.00
2	5000	EF	1	25	510	37	0.03	0.95	0.04	0.89	0.03
2	10000	EF	10	16	514	33	0.23	0.97	0.38	0.91	0.29
2	15000	EF	0	26	530	17	0.00	0.95	0.00	0.92	0.00
2	20000	EF	2	24	492	55	0.04	0.95	0.08	0.86	0.05
4	10000	EF	0	26	514	33	0.00	0.95	0.00	0.90	0.00
6	10000	EF	1	25	525	22	0.04	0.95	0.04	0.92	0.04
2	10000	AR	8	18	451	96	0.08	0.96	0.31	0.80	0.12
2	10000	BH	1	25	469	78	0.01	0.95	0.04	0.82	0.02
2	10000	MF	1	25	536	21	0.05	0.96	0.04	0.92	0.04
2	10000	TS	2	24	479	68	0.03	0.95	0.08	0.84	0.04
2	10000	VC	1	25	503	44	0.02	0.95	0.04	0.88	0.03
2	10000	VG	0	26	527	20	0.00	0.95	0.00	0.92	0.00

Table A.3: Flash confusion matrix game 2

Layers	Iteration	Brain	TP	FP	TN	FN	Rec	Spec	Prec	Acc	F1S
2	10	EF	8	5	275	287	0.03	0.98	0.62	0.49	0.05
2	3000	EF	2	11	557	5	0.29	0.98	0.15	0.97	0.20
2	5000	EF	0	13	558	4	0.00	0.98	0.00	0.97	0.00
2	10000	EF	0	13	491	71	0.00	0.97	0.00	0.85	0.00
2	15000	EF	2	11	463	99	0.02	0.98	0.15	0.81	0.04
2	20000	EF	1	12	556	6	0.14	0.98	0.08	0.97	0.10
4	10000	EF	0	13	551	11	0.00	0.98	0.00	0.96	0.00
6	10000	EF	3	10	552	10	0.23	0.98	0.23	0.97	0.23
2	10000	AR	0	13	527	35	0.00	0.98	0.00	0.92	0.00
2	10000	BH	0	13	539	23	0.00	0.98	0.00	0.94	0.00
2	10000	MF	4	9	482	80	0.05	0.98	0.31	0.85	0.08
2	10000	TS	0	13	544	18	0.00	0.98	0.00	0.95	0.00
2	10000	VC	0	13	520	42	0.00	0.98	0.00	0.90	0.00
2	10000	VG	0	13	472	90	0.00	0.97	0.00	0.82	0.00

Table A.4: Flash confusion matrix game 3

Layers	Iteration	Brain	Rec-A	Spec-A	Prec-A	Acc-A	F1S-A
2	10	EF	0.06	0.96	0.59	0.49	0.10
2	3000	EF	0.12	0.95	0.06	0.91	0.08
2	5000	EF	0.01	0.95	0.01	0.93	0.01
2	10000	EF	0.11	0.96	0.15	0.88	0.12
2	15000	EF	0.01	0.95	0.06	0.87	0.02
2	20000	EF	0.14	0.95	0.10	0.91	0.11
4	10000	EF	0.02	0.95	0.00	0.92	0.01
6	10000	EF	0.11	0.95	0.10	0.92	0.11
2	10000	AR	0.05	0.95	0.13	0.86	0.07
2	10000	BH	0.04	0.95	0.10	0.84	0.05
2	10000	MF	0.07	0.95	0.14	0.88	0.08
2	10000	TS	0.03	0.95	0.05	0.88	0.04
2	10000	VC	0.01	0.95	0.01	0.90	0.01
2	10000	VG	0.02	0.95	0.11	0.78	0.04

Table A.5: Flash confusion matrix average

Layers	Iteration	Brain	Rec-SD	Spec-SD	Prec-SD	Acc-SD	F1S-SD
2	10	EF	0.03	0.02	0.04	0.00	0.05
2	3000	EF	0.15	0.03	0.08	0.06	0.10
2	5000	EF	0.02	0.03	0.02	0.04	0.02
2	10000	EF	0.12	0.03	0.21	0.03	0.15
2	15000	EF	0.01	0.03	0.09	0.06	0.02
2	20000	EF	0.10	0.02	0.03	0.05	0.06
4	10000	EF	0.03	0.03	0.01	0.03	0.01
6	10000	EF	0.11	0.03	0.11	0.05	0.11
2	10000	AR	0.05	0.03	0.16	0.06	0.06
2	10000	BH	0.05	0.02	0.15	0.09	0.08
2	10000	MF	0.05	0.03	0.14	0.04	0.03
2	10000	TS	0.04	0.03	0.04	0.06	0.03
2	10000	VC	0.01	0.03	0.02	0.01	0.02
2	10000	VG	0.04	0.03	0.19	0.16	0.07

Table A.6: Flash confusion matrix standard deviation

A.1.3 Taunt confusion matrices

Layers	Iteration	Brain	TP	FP	TN	FN	Rec	Spec	Prec	Acc	F1S
2	10	EF	153	162	348	375	0.29	0.68	0.49	0.48	0.36
2	3000	EF	58	257	594	129	0.31	0.70	0.18	0.63	0.23
2	5000	EF	76	239	578	145	0.34	0.71	0.24	0.63	0.28
2	10000	EF	20	295	601	122	0.14	0.67	0.06	0.60	0.09
2	15000	EF	46	269	556	167	0.22	0.67	0.15	0.58	0.17
2	20000	EF	81	234	560	163	0.33	0.71	0.26	0.62	0.29
4	10000	EF	49	266	621	102	0.32	0.70	0.16	0.65	0.21
6	10000	EF	13	302	632	91	0.13	0.68	0.04	0.62	0.06
2	10000	AR	19	296	686	37	0.34	0.70	0.06	0.68	0.10
2	10000	BH	5	310	702	21	0.19	0.69	0.02	0.68	0.03
2	10000	MF	21	294	669	54	0.28	0.69	0.07	0.66	0.11
2	10000	TS	38	277	664	59	0.39	0.71	0.12	0.68	0.18
2	10000	VC	8	307	685	38	0.17	0.69	0.03	0.67	0.04
2	10000	VG	66	249	645	78	0.46	0.72	0.21	0.68	0.29

Table A.7: Taunt confusion matrix game 1

Layers	Iteration	Brain	TP	FP	TN	FN	Rec	Spec	Prec	Acc	F1S
2	10	EF	83	63	217	210	0.28	0.78	0.57	0.52	0.38
2	3000	EF	19	127	397	30	0.39	0.76	0.13	0.73	0.19
2	5000	EF	19	127	365	62	0.23	0.74	0.13	0.67	0.17
2	10000	EF	5	141	394	33	0.13	0.74	0.03	0.70	0.05
2	15000	EF	45	101	336	91	0.33	0.77	0.31	0.66	0.32
2	20000	EF	45	101	322	105	0.30	0.76	0.31	0.64	0.30
4	10000	EF	21	125	368	59	0.26	0.75	0.14	0.68	0.19
6	10000	EF	9	137	380	47	0.16	0.74	0.06	0.68	0.09
2	10000	AR	1	145	427	0	1	0.75	0.01	0.75	0.01
2	10000	BH	0	146	422	5	0	0.74	0	0.74	0
2	10000	MF	53	93	317	110	0.33	0.77	0.36	0.65	0.34
2	10000	TS	5	141	415	12	0.29	0.75	0.03	0.73	0.06
2	10000	VC	17	129	397	30	0.36	0.75	0.12	0.72	0.18
2	10000	VG	20	126	398	29	0.41	0.76	0.14	0.73	0.21

Table A.8: Taunt confusion matrix game 2

Layers	Iteration	Brain	TP	FP	TN	FN	Rec	Spec	Prec	Acc	F1S
2	10	EF	32	38	242	263	0.11	0.86	0.46	0.48	0.18
2	3000	EF	7	63	431	74	0.09	0.87	0.10	0.76	0.09
2	5000	EF	9	61	362	143	0.06	0.86	0.13	0.65	0.08
2	10000	EF	2	68	442	63	0.03	0.87	0.03	0.77	0.03
2	15000	EF	21	49	404	101	0.17	0.89	0.30	0.74	0.22
2	20000	EF	12	58	458	47	0.20	0.89	0.17	0.82	0.19
4	10000	EF	4	66	433	72	0.05	0.87	0.06	0.76	0.05
6	10000	EF	2	68	496	9	0.18	0.88	0.03	0.87	0.05
2	10000	AR	0	70	502	3	0	0.88	0	0.87	0
2	10000	BH	0	70	495	10	0	0.88	0	0.86	0
2	10000	MF	28	42	267	238	0.11	0.86	0.40	0.51	0.17
2	10000	TS	0	70	467	38	0	0.87	0	0.81	0
2	10000	VC	9	61	481	24	0.27	0.89	0.13	0.85	0.17
2	10000	VG	7	63	409	96	0.07	0.87	0.10	0.72	0.08

Table A.9: Taunt confusion matrix game 3

Layers	Iteration	Brain	Rec-A	Spec-A	Prec-A	Acc-A	F1S
2	10	EF	0.23	0.77	0.50	0.49	0.31
2	3000	EF	0.26	0.78	0.14	0.71	0.17
2	5000	EF	0.21	0.77	0.17	0.65	0.18
2	10000	EF	0.10	0.76	0.04	0.69	0.06
2	15000	EF	0.24	0.78	0.25	0.66	0.24
2	20000	EF	0.28	0.78	0.25	0.69	0.26
4	10000	EF	0.21	0.77	0.12	0.69	0.15
6	10000	EF	0.16	0.76	0.04	0.72	0.07
2	10000	AR	0.45	0.77	0.02	0.77	0.04
2	10000	BH	0.06	0.77	0.01	0.76	0.01
2	10000	MF	0.24	0.78	0.28	0.61	0.21
2	10000	TS	0.23	0.77	0.05	0.74	0.08
2	10000	VC	0.27	0.78	0.09	0.75	0.13
2	10000	VG	0.31	0.78	0.15	0.71	0.19

Table A.10: Taunt confusion matrix average

Layers	Iteration	Brain	Rec-SD	Spec-SD	Prec-SD	Acc-SD	F1S-SD
2	10	EF	0.10	0.09	0.06	0.03	0.11
2	3000	EF	0.16	0.09	0.04	0.07	0.07
2	5000	EF	0.14	0.08	0.06	0.02	0.10
2	10000	EF	0.06	0.10	0.02	0.09	0.03
2	15000	EF	0.08	0.11	0.09	0.08	0.07
2	20000	EF	0.07	0.09	0.07	0.11	0.06
4	10000	EF	0.14	0.09	0.05	0.06	0.08
6	10000	EF	0.03	0.10	0.02	0.13	0.02
2	10000	AR	0.51	0.09	0.03	0.10	0.06
2	10000	BH	0.11	0.09	0.01	0.09	0.02
2	10000	MF	0.12	0.08	0.18	0.08	0.12
2	10000	TS	0.20	0.09	0.06	0.07	0.09
2	10000	VC	0.09	0.10	0.06	0.09	0.08
2	10000	VG	0.21	0.08	0.06	0.02	0.10

Table A.11: Taunt confusion matrix standard deviation

A.1.4 Horizontal matrices Legend

Abbreviate	Name
A	Neutral
B	Right
C	Left
AA	Neutral as Neutral
BB	Right as Right
CC	Left as Left
AC	Neutral as Left
AB	Neutral as Right
BC	Right as Left
BA	Right as Neutral
CB	Left as Right
CA	Left as Neutral
Rec	Recall
Spec	Specificity
Prec	Precision
Acc	Accuracy
F1S	F1Score
SD	Standard Deviation
A	Average

Table A.12: Horizontal matrices abbreviate legend

A.1.5 Horizontal Confusion Matrices

Layers	Iteration	Brain	AA	BB	CC	AC	AB	BC	BA	CB	CA
2	10	EF	137	108	92	137	147	102	123	98	94
2	3000	EF	300	196	210	39	82	12	125	28	46
2	5000	EF	249	256	197	120	52	20	57	28	59
2	10000	EF	203	262	225	85	133	16	55	11	48
2	15000	EF	186	256	216	109	126	23	54	24	44
2	20000	EF	150	242	212	139	132	38	53	7	65
4	10000	EF	221	299	144	7	193	1	33	35	105
6	10000	EF	234	307	145	34	153	2	24	44	95
2	10000	AR	301	238	186	47	73	22	73	41	57
2	10000	BH	275	236	191	46	100	4	93	56	37
2	10000	MF	319	212	55	14	88	2	119	14	215
2	10000	TS	376	156	132	6	39	7	170	11	141
2	10000	VC	332	185	68	11	78	10	138	27	189
2	10000	VG	295	225	180	44	82	10	98	19	85

Table A.13: Horizontal confusion matrix game 1

Layers	Iteration	Brain	A-Prec	B-Prec	C-Prec	A-Rec	B-Rec	C-Rec	A-F1S	B-F1S	C-F1S
2	10	EF	0.33	0.32	0.32	0.39	0.31	0.28	0.35	0.31	0.30
2	3000	EF	0.71	0.59	0.74	0.64	0.64	0.80	0.67	0.61	0.77
2	5000	EF	0.59	0.77	0.69	0.68	0.76	0.58	0.63	0.77	0.63
2	10000	EF	0.48	0.79	0.79	0.66	0.65	0.69	0.56	0.71	0.74
2	15000	EF	0.44	0.77	0.76	0.65	0.63	0.62	0.53	0.69	0.68
2	20000	EF	0.36	0.73	0.75	0.56	0.64	0.54	0.44	0.68	0.63
4	10000	EF	0.52	0.90	0.51	0.62	0.57	0.95	0.57	0.70	0.66
6	10000	EF	0.56	0.92	0.51	0.66	0.61	0.80	0.60	0.73	0.62
2	10000	AR	0.71	0.71	0.65	0.70	0.68	0.73	0.71	0.69	0.69
2	10000	BH	0.65	0.71	0.67	0.68	0.60	0.79	0.67	0.65	0.73
2	10000	MF	0.76	0.64	0.19	0.49	0.68	0.77	0.59	0.66	0.31
2	10000	TS	0.89	0.47	0.46	0.55	0.76	0.91	0.68	0.58	0.62
2	10000	VC	0.79	0.56	0.24	0.50	0.64	0.76	0.61	0.59	0.36
2	10000	VG	0.70	0.68	0.63	0.62	0.69	0.77	0.66	0.68	0.69

Table A.14: Horizontal confusion matrix game 1 metrics by parameters

Layers	Iteration	Brain	Prec	Rec	F1S	Acc
2	10	EF	0.32	0.32	0.32	0.32
2	3000	EF	0.68	0.69	0.69	0.68
2	5000	EF	0.68	0.68	0.68	0.68
2	10000	EF	0.69	0.67	0.67	0.66
2	15000	EF	0.66	0.64	0.63	0.63
2	20000	EF	0.61	0.58	0.58	0.58
4	10000	EF	0.64	0.71	0.64	0.64
6	10000	EF	0.66	0.69	0.65	0.66
2	10000	AR	0.69	0.70	0.70	0.70
2	10000	BH	0.68	0.69	0.68	0.68
2	10000	MF	0.53	0.65	0.52	0.56
2	10000	TS	0.61	0.74	0.62	0.64
2	10000	VC	0.53	0.64	0.52	0.56
2	10000	VG	0.67	0.69	0.68	0.67

Table A.15: Horizontal confusion matrix game 1 metrics

Layers	Iteration	Brain	AA	BB	CC	AC	AB	BC	BA	CB	CA
2	10	EF	75	70	42	76	77	66	57	59	51
2	3000	EF	66	72	270	19	37	24	42	8	35
2	5000	EF	163	118	116	37	28	1	74	19	17
2	10000	EF	89	61	263	20	13	46	31	1	49
2	15000	EF	78	48	254	36	8	38	52	3	56
2	20000	EF	78	63	236	34	10	34	41	3	74
4	10000	EF	53	99	248	18	51	12	27	25	40
6	10000	EF	90	52	254	24	8	28	58	6	53
2	10000	AR	79	75	188	23	20	14	49	47	78
2	10000	BH	75	81	250	29	18	22	35	9	54
2	10000	MF	112	31	168	10	0	12	95	0	145
2	10000	TS	100	94	136	3	19	2	42	16	161
2	10000	VC	115	31	179	2	5	6	101	1	133
2	10000	VG	82	46	271	24	16	48	44	15	27

Table A.16: Horizontal confusion matrix game 2

Layers	Iteration	Brain	A-Prec	B-Prec	C-Prec	A-Rec	B-Rec	C-Rec	A-F1S	B-F1S	C-F1S
2	10	EF	0.33	0.36	0.28	0.41	0.34	0.23	0.36	0.35	0.25
2	3000	EF	0.54	0.52	0.86	0.46	0.62	0.86	0.50	0.56	0.86
2	5000	EF	0.71	0.61	0.76	0.64	0.72	0.75	0.68	0.66	0.76
2	10000	EF	0.73	0.44	0.84	0.53	0.81	0.80	0.61	0.57	0.82
2	15000	EF	0.64	0.35	0.81	0.42	0.81	0.77	0.51	0.49	0.79
2	20000	EF	0.64	0.46	0.75	0.40	0.83	0.78	0.50	0.59	0.76
4	10000	EF	0.43	0.72	0.79	0.44	0.57	0.89	0.44	0.63	0.84
6	10000	EF	0.74	0.38	0.81	0.45	0.79	0.83	0.56	0.51	0.82
2	10000	AR	0.65	0.54	0.60	0.38	0.53	0.84	0.48	0.54	0.70
2	10000	BH	0.61	0.59	0.80	0.46	0.75	0.83	0.52	0.66	0.81
2	10000	MF	0.92	0.22	0.54	0.32	1.00	0.88	0.47	0.37	0.67
2	10000	TS	0.82	0.68	0.43	0.33	0.73	0.96	0.47	0.70	0.60
2	10000	VC	0.94	0.22	0.57	0.33	0.84	0.96	0.49	0.35	0.72
2	10000	VG	0.67	0.33	0.87	0.54	0.60	0.79	0.60	0.43	0.83

Table A.17: Horizontal confusion matrix game 2 metrics by parameters

Layers	Iteration	Brain	Prec	Rec	F1S	Acc
2	10	EF	0.32	0.33	0.32	0.33
2	3000	EF	0.64	0.65	0.64	0.71
2	5000	EF	0.70	0.70	0.70	0.69
2	10000	EF	0.67	0.71	0.67	0.72
2	15000	EF	0.60	0.67	0.60	0.66
2	20000	EF	0.62	0.67	0.62	0.66
4	10000	EF	0.65	0.63	0.64	0.70
6	10000	EF	0.64	0.69	0.63	0.69
2	10000	AR	0.60	0.58	0.57	0.60
2	10000	BH	0.67	0.68	0.67	0.71
2	10000	MF	0.56	0.73	0.50	0.54
2	10000	TS	0.65	0.67	0.59	0.58
2	10000	VC	0.58	0.71	0.52	0.57
2	10000	VG	0.62	0.64	0.62	0.70

Table A.18: Horizontal confusion matrix game 2 metrics

Layers	Iteration	Brain	AA	BB	CC	AC	AB	BC	BA	CB	CA
2	10	EF	80	97	15	66	69	96	100	26	26
2	3000	EF	127	262	62	56	32	5	26	0	5
2	5000	EF	167	262	48	15	33	1	30	1	18
2	10000	EF	176	249	60	18	21	0	44	2	5
2	15000	EF	163	218	60	23	29	4	71	0	7
2	20000	EF	173	258	60	3	39	1	34	1	6
4	10000	EF	159	261	33	27	29	4	28	1	33
6	10000	EF	185	244	53	6	24	3	46	1	13
2	10000	AR	181	269	57	10	24	3	21	6	4
2	10000	BH	183	266	53	18	14	0	27	6	8
2	10000	MF	147	260	51	26	42	2	31	3	13
2	10000	TS	190	212	47	15	10	27	54	0	20
2	10000	VC	197	227	18	3	15	1	65	1	48
2	10000	VG	145	265	54	36	34	8	20	2	11

Table A.19: Horizontal confusion matrix game 3

Layers	Iteration	Brain	A-Prec	B-Prec	C-Prec	A-Rec	B-Rec	C-Rec	A-F1S	B-F1S	C-F1S
2	10	EF	0.37	0.33	0.22	0.39	0.51	0.08	0.38	0.40	0.12
2	3000	EF	0.59	0.89	0.93	0.80	0.89	0.50	0.68	0.89	0.65
2	5000	EF	0.78	0.89	0.72	0.78	0.89	0.75	0.78	0.89	0.73
2	10000	EF	0.82	0.85	0.90	0.78	0.92	0.77	0.80	0.88	0.83
2	15000	EF	0.76	0.74	0.90	0.68	0.88	0.69	0.71	0.81	0.78
2	20000	EF	0.80	0.88	0.90	0.81	0.87	0.94	0.81	0.87	0.92
4	10000	EF	0.74	0.89	0.49	0.72	0.90	0.52	0.73	0.89	0.50
6	10000	EF	0.86	0.83	0.79	0.76	0.91	0.85	0.81	0.87	0.82
2	10000	AR	0.84	0.92	0.85	0.88	0.90	0.81	0.86	0.91	0.83
2	10000	BH	0.85	0.91	0.79	0.84	0.93	0.75	0.85	0.92	0.77
2	10000	MF	0.68	0.89	0.76	0.77	0.85	0.65	0.72	0.87	0.70
2	10000	TS	0.88	0.72	0.70	0.72	0.95	0.53	0.79	0.82	0.60
2	10000	VC	0.92	0.77	0.27	0.64	0.93	0.82	0.75	0.85	0.40
2	10000	VG	0.67	0.90	0.81	0.82	0.88	0.55	0.74	0.89	0.65

Table A.20: Horizontal confusion matrix game 3 metrics by parameters

Layers	Iteration	Brain	Prec	Rec	F1S	Acc
2	10	EF	0.31	0.33	0.26	0.33
2	3000	EF	0.80	0.73	0.77	0.78
2	5000	EF	0.80	0.80	0.81	0.83
2	10000	EF	0.85	0.82	0.85	0.84
2	15000	EF	0.80	0.75	0.79	0.77
2	20000	EF	0.86	0.87	0.89	0.85
4	10000	EF	0.71	0.71	0.70	0.79
6	10000	EF	0.83	0.84	0.85	0.84
2	10000	AR	0.87	0.86	0.87	0.88
2	10000	BH	0.85	0.84	0.84	0.87
2	10000	MF	0.78	0.76	0.78	0.80
2	10000	TS	0.77	0.73	0.71	0.78
2	10000	VC	0.65	0.80	0.63	0.77
2	10000	VG	0.79	0.75	0.77	0.81

Table A.21: Horizontal confusion matrix game 3 metrics

Layers	Iteration	Brain	Prec-A	Rec-A	F1S-A	Acc-M
2	10	EF	0.32	0.33	0.30	0.33
2	3000	EF	0.71	0.69	0.70	0.73
2	5000	EF	0.73	0.73	0.73	0.73
2	10000	EF	0.74	0.73	0.73	0.74
2	15000	EF	0.69	0.68	0.67	0.69
2	20000	EF	0.70	0.71	0.70	0.70
4	10000	EF	0.67	0.69	0.66	0.71
6	10000	EF	0.71	0.74	0.71	0.73
2	10000	AR	0.72	0.72	0.71	0.73
2	10000	BH	0.73	0.74	0.73	0.75
2	10000	MF	0.62	0.71	0.60	0.63
2	10000	TS	0.67	0.72	0.64	0.67
2	10000	VC	0.59	0.71	0.56	0.63
2	10000	VG	0.70	0.70	0.69	0.73

Table A.22: Horizontal confusion matrix game average

Layers	Iteration	Brain	Prec-SD	Rec-SD	F1S-SD	Acc-SD
2	10	EF	0.01	0.00	0.04	0.00
2	3000	EF	0.08	0.04	0.07	0.05
2	5000	EF	0.06	0.07	0.07	0.08
2	10000	EF	0.10	0.08	0.11	0.09
2	15000	EF	0.10	0.06	0.10	0.07
2	20000	EF	0.14	0.15	0.17	0.14
4	10000	EF	0.04	0.04	0.03	0.07
6	10000	EF	0.10	0.09	0.12	0.09
2	10000	AR	0.14	0.14	0.15	0.14
2	10000	BH	0.10	0.09	0.10	0.11
2	10000	MF	0.14	0.06	0.16	0.14
2	10000	TS	0.08	0.04	0.06	0.10
2	10000	VC	0.06	0.08	0.06	0.12
2	10000	VG	0.09	0.06	0.08	0.07

Table A.23: Horizontal confusion matrix game standard deviation

A.1.6 Vertical matrices Legend

Abbreviate	Name
A	Neutral
B	Up
C	Down
AA	Neutral as Neutral
BB	Up as Up
CC	Down as Down
AC	Neutral as Down
AB	Neutral as Up
BC	Up as Down
BA	Up as Neutral
CB	Down as Up
CA	Down as Neutral
Rec	Recall
Spec	Specificity
Prec	Precision
Acc	Accuracy
F1S	F1Score
SD	Standard Deviation
A	Average

Table A.24: Vertical matrices abbreviate legend

A.1.7 Vertical Confusion Matrices

Layers	Iteration	Brain	AA	BB	CC	AC	AB	BC	BA	CB	CA
2	10	EF	159	90	105	155	162	75	75	103	114
2	3000	EF	364	155	206	45	67	0	85	11	105
2	5000	EF	366	173	206	17	93	0	67	14	102
2	10000	EF	319	146	243	105	52	16	78	13	66
2	15000	EF	293	123	257	131	52	22	95	0	65
2	20000	EF	402	113	229	41	33	0	127	5	88
4	10000	EF	296	138	219	67	113	13	89	22	81
6	10000	EF	256	187	167	53	167	8	45	23	132
2	10000	AR	205	210	246	109	162	5	25	17	59
2	10000	BH	391	176	203	27	58	7	57	1	118
2	10000	MF	343	104	135	81	52	34	102	0	187
2	10000	TS	329	169	196	77	70	16	55	31	95
2	10000	VC	411	129	149	16	49	10	101	14	159
2	10000	VG	390	87	245	74	12	2	151	3	74

Table A.25: Vertical confusion matrix game 1

Layers	Iteration	Brain	A-Prec	B-Prec	C-Prec	A-Rec	B-Rec	C-Rec	A-F1S	B-F1S	C-F1S
2	10	EF	0.33	0.38	0.33	0.46	0.25	0.31	0.39	0.30	0.32
2	3000	EF	0.76	0.65	0.64	0.66	0.67	0.82	0.71	0.66	0.72
2	5000	EF	0.77	0.72	0.64	0.68	0.62	0.92	0.72	0.67	0.76
2	10000	EF	0.67	0.61	0.75	0.69	0.69	0.67	0.68	0.65	0.71
2	15000	EF	0.62	0.51	0.80	0.65	0.70	0.63	0.63	0.59	0.70
2	20000	EF	0.84	0.47	0.71	0.65	0.75	0.85	0.74	0.58	0.77
4	10000	EF	0.62	0.58	0.68	0.64	0.51	0.73	0.63	0.54	0.71
6	10000	EF	0.54	0.78	0.52	0.59	0.50	0.73	0.56	0.61	0.61
2	10000	AR	0.43	0.88	0.76	0.71	0.54	0.68	0.54	0.67	0.72
2	10000	BH	0.82	0.73	0.63	0.69	0.75	0.86	0.75	0.74	0.73
2	10000	MF	0.72	0.43	0.42	0.54	0.67	0.54	0.62	0.53	0.47
2	10000	TS	0.69	0.70	0.61	0.69	0.63	0.68	0.69	0.66	0.64
2	10000	VC	0.86	0.54	0.46	0.61	0.67	0.85	0.72	0.60	0.60
2	10000	VG	0.82	0.36	0.76	0.63	0.85	0.76	0.71	0.51	0.76

Table A.26: Vertical confusion matrix game 1 metrics by parameters

Layers	Iteration	Brain	Prec	Rec	F1S	Acc
2	10	EF	0.35	0.34	0.34	0.34
2	3000	EF	0.68	0.71	0.69	0.70
2	5000	EF	0.71	0.74	0.72	0.72
2	10000	EF	0.68	0.68	0.68	0.68
2	15000	EF	0.64	0.66	0.64	0.65
2	20000	EF	0.68	0.75	0.70	0.72
4	10000	EF	0.63	0.62	0.62	0.63
6	10000	EF	0.61	0.61	0.59	0.59
2	10000	AR	0.69	0.64	0.64	0.64
2	10000	BH	0.73	0.77	0.74	0.74
2	10000	MF	0.52	0.58	0.54	0.56
2	10000	TS	0.67	0.66	0.66	0.67
2	10000	VC	0.62	0.71	0.64	0.66
2	10000	VG	0.65	0.75	0.66	0.70

Table A.27: Vertical confusion matrix game 1 metrics

Layers	Iteration	Brain	AA	BB	CC	AC	AB	BC	BA	CB	CA
2	10	EF	75	70	42	76	77	66	57	59	51
2	3000	EF	106	42	145	120	2	27	124	0	7
2	5000	EF	163	118	116	37	28	1	74	19	17
2	10000	EF	107	147	130	86	35	7	39	6	16
2	15000	EF	133	115	115	78	17	22	56	4	33
2	20000	EF	134	141	80	10	84	3	49	22	50
4	10000	EF	148	115	102	67	13	8	70	2	48
6	10000	EF	163	111	89	44	21	5	77	8	55
2	10000	AR	164	95	84	36	28	10	88	5	63
2	10000	BH	139	160	52	36	53	2	31	5	95
2	10000	MF	186	120	73	19	23	11	62	16	63
2	10000	TS	148	70	85	74	6	15	108	0	67
2	10000	VC	169	139	44	7	52	2	52	3	105
2	10000	VG	182	151	78	3	43	1	41	8	66

Table A.28: Vertical confusion matrix game 2

Layers	Iteration	Brain	A-Prec	B-Prec	C-Prec	A-Rec	B-Rec	C-Rec	A-F1S	B-F1S	C-F1S
2	10	EF	0.33	0.36	0.28	0.41	0.34	0.23	0.36	0.35	0.25
2	3000	EF	0.46	0.22	0.95	0.45	0.95	0.50	0.46	0.35	0.65
2	5000	EF	0.71	0.61	0.76	0.64	0.72	0.75	0.68	0.66	0.76
2	10000	EF	0.47	0.76	0.86	0.66	0.78	0.58	0.55	0.77	0.69
2	15000	EF	0.58	0.60	0.76	0.60	0.85	0.53	0.59	0.70	0.63
2	20000	EF	0.59	0.73	0.53	0.58	0.57	0.86	0.58	0.64	0.65
4	10000	EF	0.65	0.60	0.67	0.56	0.88	0.58	0.60	0.71	0.62
6	10000	EF	0.71	0.58	0.59	0.55	0.79	0.64	0.62	0.67	0.61
2	10000	AR	0.72	0.49	0.55	0.52	0.74	0.65	0.60	0.59	0.60
2	10000	BH	0.61	0.83	0.34	0.52	0.73	0.58	0.56	0.78	0.43
2	10000	MF	0.82	0.62	0.48	0.60	0.75	0.71	0.69	0.68	0.57
2	10000	TS	0.65	0.36	0.56	0.46	0.92	0.49	0.54	0.52	0.52
2	10000	VC	0.74	0.72	0.29	0.52	0.72	0.83	0.61	0.72	0.43
2	10000	VG	0.80	0.78	0.51	0.63	0.75	0.95	0.70	0.76	0.67

Table A.29: Vertical confusion matrix game 2 metrics by parameters

Layers	Iteration	Brain	Prec	Rec	F1S	Acc
2	10	EF	0.32	0.33	0.32	0.33
2	3000	EF	0.55	0.63	0.49	0.51
2	5000	EF	0.70	0.70	0.70	0.69
2	10000	EF	0.70	0.68	0.67	0.67
2	15000	EF	0.65	0.66	0.64	0.63
2	20000	EF	0.61	0.67	0.63	0.62
4	10000	EF	0.64	0.67	0.64	0.64
6	10000	EF	0.63	0.66	0.63	0.63
2	10000	AR	0.59	0.64	0.60	0.60
2	10000	BH	0.59	0.61	0.59	0.61
2	10000	MF	0.64	0.69	0.65	0.66
2	10000	TS	0.52	0.62	0.53	0.53
2	10000	VC	0.58	0.69	0.59	0.61
2	10000	VG	0.70	0.78	0.71	0.72

Table A.30: Vertical confusion matrix game 2 metrics

Layers	Iteration	Brain	AA	BB	CC	AC	AB	BC	BA	CB	CA
2	10	EF	103	47	39	105	116	41	41	43	40
2	3000	EF	245	15	107	75	4	0	114	0	15
2	5000	EF	262	112	97	49	13	1	16	2	23
2	10000	EF	234	123	101	72	18	0	6	0	21
2	15000	EF	227	117	89	71	26	1	11	0	33
2	20000	EF	251	118	84	37	36	0	11	1	37
4	10000	EF	195	100	103	68	61	1	28	1	18
6	10000	EF	218	96	95	95	11	0	33	2	25
2	10000	AR	270	112	97	26	28	6	11	0	25
2	10000	BH	230	106	104	61	33	0	23	0	18
2	10000	MF	312	112	36	2	10	0	17	6	80
2	10000	TS	250	109	94	58	16	0	20	0	28
2	10000	VC	267	79	74	18	39	0	50	0	48
2	10000	VG	252	120	91	37	35	0	9	6	25

Table A.31: Vertical confusion matrix game 3

Layers	Iteration	Brain	A-Prec	B-Prec	C-Prec	A-Rec	B-Rec	C-Rec	A-F1S	B-F1S	C-F1S
2	10	EF	0.32	0.36	0.32	0.56	0.23	0.21	0.41	0.28	0.25
2	3000	EF	0.76	0.12	0.88	0.66	0.79	0.59	0.70	0.20	0.70
2	5000	EF	0.81	0.87	0.80	0.87	0.88	0.66	0.84	0.88	0.72
2	10000	EF	0.72	0.95	0.83	0.90	0.87	0.58	0.80	0.91	0.68
2	15000	EF	0.70	0.91	0.73	0.84	0.82	0.55	0.76	0.86	0.63
2	20000	EF	0.77	0.91	0.69	0.84	0.76	0.69	0.81	0.83	0.69
4	10000	EF	0.60	0.78	0.84	0.81	0.62	0.60	0.69	0.69	0.70
6	10000	EF	0.67	0.74	0.78	0.79	0.88	0.50	0.73	0.81	0.61
2	10000	AR	0.83	0.87	0.80	0.88	0.80	0.75	0.86	0.83	0.77
2	10000	BH	0.71	0.82	0.85	0.85	0.76	0.63	0.77	0.79	0.72
2	10000	MF	0.96	0.87	0.30	0.76	0.88	0.95	0.85	0.87	0.45
2	10000	TS	0.77	0.84	0.77	0.84	0.87	0.62	0.80	0.86	0.69
2	10000	VC	0.82	0.61	0.61	0.73	0.67	0.80	0.78	0.64	0.69
2	10000	VG	0.78	0.93	0.75	0.88	0.75	0.71	0.83	0.83	0.73

Table A.32: Vertical confusion matrix game 3 metrics by parameters

Layers	Iteration	Brain	Prec	Rec	F1S	Acc
2	10	EF	0.33	0.33	0.31	0.33
2	3000	EF	0.58	0.68	0.54	0.64
2	5000	EF	0.82	0.80	0.81	0.82
2	10000	EF	0.83	0.78	0.80	0.80
2	15000	EF	0.78	0.74	0.75	0.75
2	20000	EF	0.79	0.76	0.78	0.79
4	10000	EF	0.74	0.68	0.69	0.69
6	10000	EF	0.73	0.72	0.71	0.71
2	10000	AR	0.83	0.81	0.82	0.83
2	10000	BH	0.79	0.75	0.76	0.77
2	10000	MF	0.71	0.86	0.72	0.80
2	10000	TS	0.80	0.78	0.78	0.79
2	10000	VC	0.68	0.74	0.70	0.73
2	10000	VG	0.82	0.78	0.79	0.81

Table A.33: Vertical confusion matrix game 3 metrics

Layers	Iteration	Brain	Prec-A	Rec-A	F1S	Acc-M
2	10	EF	0.33	0.33	0.32	0.33
2	3000	EF	0.49	0.55	0.45	0.50
2	5000	EF	0.73	0.74	0.73	0.74
2	10000	EF	0.75	0.73	0.73	0.73
2	15000	EF	0.70	0.69	0.69	0.69
2	20000	EF	0.68	0.70	0.68	0.69
4	10000	EF	0.68	0.70	0.68	0.68
6	10000	EF	0.66	0.67	0.66	0.66
2	10000	AR	0.68	0.68	0.67	0.67
2	10000	BH	0.69	0.67	0.67	0.67
2	10000	MF	0.69	0.77	0.70	0.73
2	10000	TS	0.61	0.66	0.62	0.63
2	10000	VC	0.64	0.70	0.65	0.67
2	10000	VG	0.71	0.76	0.71	0.73

Table A.34: Vertical confusion matrix game average

Layers	Iteration	Brain	Prec-SD	Rec-SD	Acc-SD	F1S-SD
2	10	EF	0.01	0.01	0.01	0.01
2	3000	EF	0.07	0.04	0.10	0.11
2	5000	EF	0.07	0.05	0.07	0.06
2	10000	EF	0.09	0.06	0.07	0.07
2	15000	EF	0.08	0.04	0.07	0.06
2	20000	EF	0.09	0.05	0.08	0.08
4	10000	EF	0.06	0.03	0.03	0.04
6	10000	EF	0.07	0.06	0.06	0.06
2	10000	AR	0.12	0.10	0.13	0.12
2	10000	BH	0.10	0.08	0.08	0.09
2	10000	MF	0.09	0.14	0.12	0.09
2	10000	TS	0.14	0.08	0.13	0.13
2	10000	VC	0.05	0.02	0.06	0.06
2	10000	VG	0.09	0.02	0.06	0.07

Table A.35: Vertical confusion matrix game standard deviation

A.2 Global metrics

A.2.1 Global metrics Legend

Abbreviate	Name
PMC	Perfect Match Count
IMC	Imperfect Match Count
PM	Perfect Match percentage
IM	Imperfect Match percentage
HL	Hamming Loss
SD	Standard Deviation
A	Average

Table A.36: Global metrics abbreviate legend

A.2.2 Global metrics Matrices

Layers	Iteration	Brain	PMC	IMC	Total	IM	PM	HL
2	10	EF	35	1296	1331	97	3	2.36
2	3000	EF	380	951	1331	71	29	1.11
2	5000	EF	389	942	1331	71	29	1.06
2	10000	EF	292	1039	1331	78	22	1.18
2	15000	EF	281	1050	1331	79	21	1.25
2	20000	EF	322	1009	1331	76	24	1.18
4	10000	EF	321	1010	1331	76	24	1.18
6	10000	EF	330	1001	1331	75	25	1.26
2	10000	AR	335	996	1331	75	25	1.13
2	10000	BH	372	959	1331	72	28	1.15
2	10000	MF	250	1081	1331	81	19	1.33
2	10000	TS	359	972	1331	73	27	1.15
2	10000	VC	335	996	1331	75	25	1.20
2	10000	VG	310	1021	1331	77	23	1.34

Table A.37: Global metrics matrix game 1

Layers	Iteration	Brain	PMC	IMC	Total	IM	PM	HL
2	10	EF	16	559	575	97	3	2.35
2	3000	EF	127	448	575	78	22	1.18
2	5000	EF	168	407	575	71	29	1.05
2	10000	EF	196	379	575	66	34	1.00
2	15000	EF	135	440	575	76	24	1.11
2	20000	EF	148	428	575	75	26	1.22
4	10000	EF	159	416	575	72	28	1.09
6	10000	EF	164	411	575	72	28	1.08
2	10000	AR	123	452	575	79	21	1.26
2	10000	BH	159	416	575	72	28	1.12
2	10000	MF	115	460	575	80	20	1.23
2	10000	TS	116	459	575	80	20	1.32
2	10000	VC	149	426	575	74	26	1.22
2	10000	VG	212	363	575	63	37	0.94

Table A.38: Global metrics matrix game 2

Layers	Iteration	Brain	PMC	IMC	Total	IM	PM	HL
2	10	EF	11	562	573	98	2	2.37
2	3000	EF	223	350	573	61	39	0.84
2	5000	EF	238	335	573	58	42	0.74
2	10000	EF	239	334	573	58	42	0.73
2	15000	EF	201	372	573	65	35	0.93
2	20000	EF	312	261	573	46	54	0.57
4	10000	EF	224	349	573	61	39	0.80
6	10000	EF	293	280	573	49	51	0.62
2	10000	AR	348	225	573	39	61	0.50
2	10000	BH	312	261	573	46	54	0.56
2	10000	MF	178	395	573	69	31	1.05
2	10000	TS	269	304	573	53	47	0.67
2	10000	VC	279	294	573	51	49	0.74
2	10000	VG	239	334	573	58	42	0.84

Table A.39: Global metrics matrix game 3

Layers	Iteration	Brain	IM-A	PM-A	HL-M
2	10	EF	98	2	2.36
2	3000	EF	70	30	1.04
2	5000	EF	67	33	0.95
2	10000	EF	67	33	0.97
2	15000	EF	73	27	1.10
2	20000	EF	65	35	0.99
4	10000	EF	70	30	1.02
6	10000	EF	65	35	0.98
2	10000	AR	64	36	0.96
2	10000	BH	63	37	0.95
2	10000	MF	77	23	1.20
2	10000	TS	69	31	1.05
2	10000	VC	67	33	1.05
2	10000	VG	66	34	1.04

Table A.40: Global metrics average

Layers	Iteration	Brain	IM-SD	PM-SD	HL-SD
2	10	EF	0	0	0.01
2	3000	EF	8	8	0.18
2	5000	EF	7	7	0.18
2	10000	EF	10	10	0.22
2	15000	EF	7	7	0.16
2	20000	EF	17	17	0.36
4	10000	EF	8	8	0.20
6	10000	EF	14	14	0.33
2	10000	AR	22	22	0.41
2	10000	BH	15	15	0.33
2	10000	MF	7	7	0.14
2	10000	TS	14	14	0.34
2	10000	VC	13	13	0.27
2	10000	VG	10	10	0.26

Table A.41: Global standard deviation

A.3 Confusion Matrices for complex architecture

A.3.1 Flash Confusion Matrices

Layers	Iteration	Brain	TP	FP	TN	FN	Rec	Spec	Prec	Acc	F1S
2	10	EF	10	11	124	143	0.07	0.92	0.48	0.47	0.11
2	5000	EF	12	9	148	119	0.09	0.94	0.57	0.56	0.16
2	10000	EF	3	18	246	21	0.13	0.93	0.14	0.86	0.13
2	15000	EF	0	21	258	9	0.00	0.92	0.00	0.90	0.00
2	20000	EF	0	3	263	22	0.00	0.99	0.00	0.91	0.00

Table A.42: Flash confusion matrix

A.3.2 Taunt confusion matrices

Layers	Iteration	Brain	TP	FP	TN	FN	Rec	Spec	Prec	Acc	F1S
2	10	EF	0	3	173	112	0.00	0.98	0.00	0.60	0.00
2	5000	EF	0	3	253	32	0.00	0.99	0.00	0.88	0.00
2	10000	EF	0	3	221	64	0.00	0.99	0.00	0.77	0.00
2	15000	EF	0	3	263	22	0.00	0.99	0.00	0.91	0.00
2	20000	EF	0	3	227	58	0.00	0.99	0.00	0.79	0.00

Table A.43: Taunt confusion matrix

A.3.3 Horizontal Confusion Matrices

Layers	Iteration	Brain	AA	BB	CC	AC	AB	BC	BA	CB	CA
2	10	EF	41	12	48	29	30	23	19	45	41
2	5000	EF	30	34	38	11	59	6	14	62	34
2	10000	EF	40	32	75	24	36	9	13	34	25
2	15000	EF	49	25	61	35	16	6	23	7	66
2	20000	EF	45	22	77	39	16	11	21	14	43

Table A.44: Horizontal confusion matrix

Layers	Iteration	Brain	A-Prec	B-Prec	C-Prec	A-Rec	B-Rec	C-Rec	A-F1S	B-F1S	C-F1S
2	10	EF	0.41	0.22	0.36	0.41	0.14	0.48	0.41	0.17	0.41
2	5000	EF	0.30	0.63	0.28	0.38	0.22	0.69	0.34	0.33	0.40
2	10000	EF	0.40	0.59	0.56	0.51	0.31	0.69	0.45	0.41	0.62
2	15000	EF	0.49	0.46	0.46	0.36	0.52	0.60	0.41	0.49	0.52
2	20000	EF	0.45	0.41	0.57	0.41	0.42	0.61	0.43	0.42	0.59

Table A.45: Horizontal confusion matrix metrics by parameters

Layers	Iteration	Brain	Prec	Rec	F1S	Acc
2	10	EF	0.33	0.34	0.33	0.35
2	5000	EF	0.40	0.43	0.35	0.35
2	10000	EF	0.52	0.51	0.49	0.51
2	15000	EF	0.47	0.49	0.47	0.47
2	20000	EF	0.48	0.48	0.48	0.50

Table A.46: Horizontal confusion matrix metrics

A.3.4 Vertical Confusion Matrices

Layers	Iteration	Brain	AA	BB	CC	AC	AB	BC	BA	CB	CA
2	10	EF	46	23	34	40	52	18	21	30	24
2	5000	EF	97	29	25	6	35	4	29	17	46
2	10000	EF	75	33	61	45	18	5	24	9	18
2	15000	EF	109	20	48	9	20	1	41	9	31
2	20000	EF	96	22	51	12	30	2	38	4	33

Table A.47: Vertical confusion matrix game

Layers	Iteration	Brain	A-Prec	B-Prec	C-Prec	A-Rec	B-Rec	C-Rec	A-F1S	B-F1S	C-F1S
2	10	EF	0.33	0.37	0.39	0.51	0.22	0.37	0.40	0.28	0.38
2	5000	EF	0.70	0.47	0.28	0.56	0.36	0.71	0.63	0.41	0.41
2	10000	EF	0.54	0.53	0.69	0.64	0.55	0.55	0.59	0.54	0.61
2	15000	EF	0.79	0.32	0.55	0.60	0.41	0.83	0.68	0.36	0.66
2	20000	EF	0.70	0.35	0.58	0.57	0.39	0.78	0.63	0.37	0.67

Table A.48: Vertical confusion matrix metrics by parameters

Layers	Iteration	Brain	Prec	Rec	F1S	Acc
2	10	EF	0.36	0.36	0.35	0.36
2	5000	EF	0.48	0.55	0.48	0.52
2	10000	EF	0.59	0.58	0.58	0.59
2	15000	EF	0.55	0.61	0.57	0.61
2	20000	EF	0.54	0.58	0.56	0.59

Table A.49: Vertical confusion matrix game metrics

A.4 Survey

Subject	Predicted	Confidence	Real
BH	Green	Partially agree	Yellow
VG	Blue	Partially agree	Blue
VC	Green	Strongly agree	Green
MF	Red	Agree	Red
TS	Red	Agree	Red
AR	Blue	Partially agree	Yellow

Table A.50: Survey recognition game 1

Subject	Predicted	Confidence	Real
BH	Blue	Disagree	Blue
VG	Yellow	Partially agree	Yellow
VC	Red	Partially agree	Red
MF	Yellow	Partially agree	Green
TS	Blue	Partially agree	Yellow
AR	Yellow	Partially agree	Blue

Table A.51: Survey recognition game 2

Subject	Predicted	Confidence	Real
BH	Red	Partially agree	Green
VG	Red	Partially agree	Red
VC	Green	Agree	Red
MF	Blue	Partially agree	Blue
TS	Green	Agree	Blue
AR	Green	Partially agree	Green

Table A.52: Survey recognition game 3

Subject	Reason
BH	Always on the move but often got stuck in the corners
VG	Because in the demo my clone was visible more time and moving in the center.
VC	Seems a bit erratic at first.. just like me!
MF	Because its behavior was similar to the demo
TS	It's the more active.
AR	It has a similar movement to demo however i don't perceive it as my clone

Table A.53: Survey reasoning game 1

Subject	Reason
BH	The least dumb of all!
VG	Similar reasons to question 1.
VC	seemed the most consistent
MF	Because it was colliding too much
TS	It is the only one that explores a little big of the map.
AR	Same reason

Table A.54: Survey reasoning game 2

Subject	Reason
BH	I was often on the corners
VG	Similar reasons to question 1.
VC	It seemed to be the one that was more confident!
MF	Flashing without a clear objetive
TS	It explores the map.
AR	Same reason

Table A.55: Survey reasoning game 3

B

Extra details

B.1 Alternative architectures

B.1.1 Extra inputs

The architecture present in Figure B.1 is one of the many alternatives that were tested, This architecture in contrast with the one used with the project includes more types of inputs, for instance, the visibility state of the other players, the current cool-down of the flash, how many targets are left. The fact is that architectures that include so many inputs were not good because the agents presented two noticeable problems. Most movement done were horizontal and the agents have the tendency of forming conglomerates and not separating. An architecture with these inputs may seem like a natural answer to the problem presented by Pic-a-boo. However, given the initial results this option was not explored further.

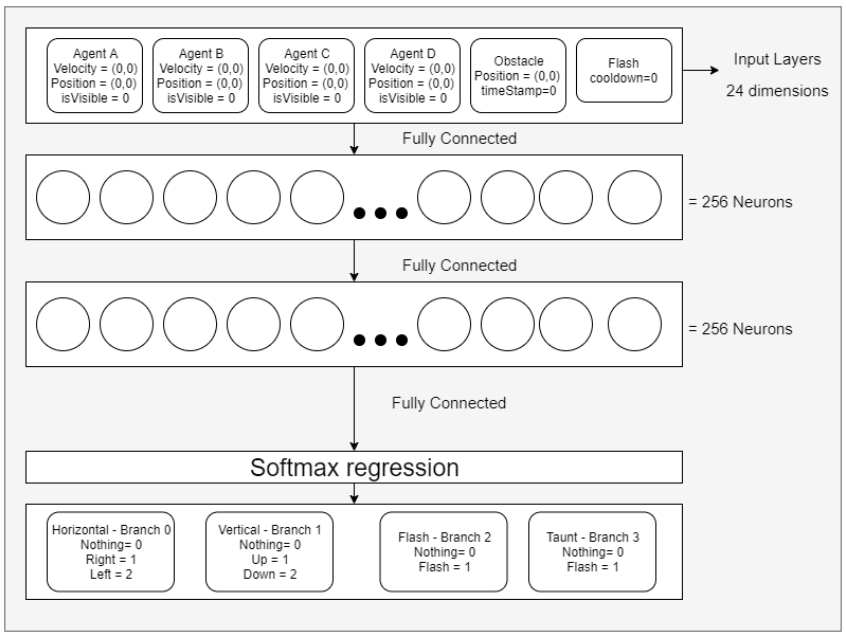

Figure B.1: Alternative feed-forward Neural Network Topology.

B.1.2 Alternative architecture results

In this section the initial measures obtained from the alternative architecture mentioned previously are analyzed. However, the test present are fewer compared to the previous architecture. This is because the agent produced by this architecture had a weird behavior moving in only one direction without exploring the map, in addition the clones have a tendency to form clusters [1]. Numerous modifications were done to try to make them work. For instance increasing the layers adding or eliminating inputs, however the efforts were futile. The following graphs are the final iteration of this architecture before abandoning it. The agents in this architecture were trained from 10, 5000, 10000, 15000 and 20000 iterations with 2 layers.

B.1.2.A Confusion matrices for flash and taunt

[1] alternative architecture game video

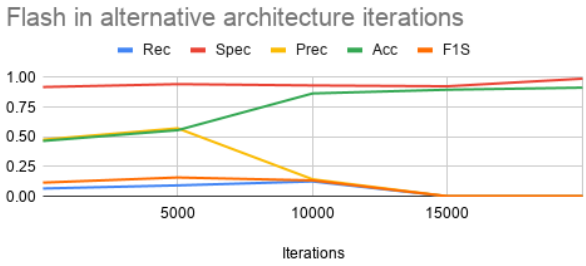

Figure B.2: Flash action progress for alternative architecture.

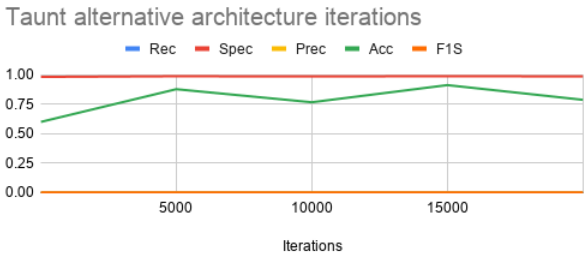

Figure B.3: Taunt action progress for alternative architecture.

Starting by the flash and taunt action which can be seen in Figure B.2 and in Figure B.3 respectively. The graph presents almost the same behavior as the more simple architecture. In here Specificity starts high and maintains itself high. Accuracy starts low and increases but slower than in the previous architecture. Precision starts high and then decreases but for this case it plummets down to zero. For the recall and F1Score both measures start in low number but again plummets to zero. In general the performance of the flashing and taunt for this architecture is quite bad taking into account that 3 of the 5 metrics are 0

B.1.2.B Confusion matrices for navigation

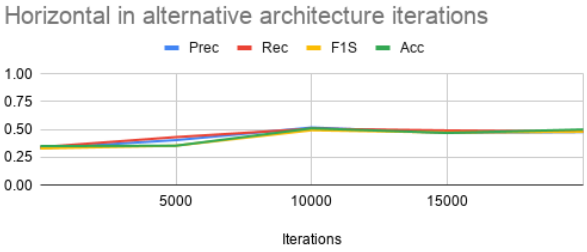

Figure B.4: Horizontal action progress for alternative architecture.

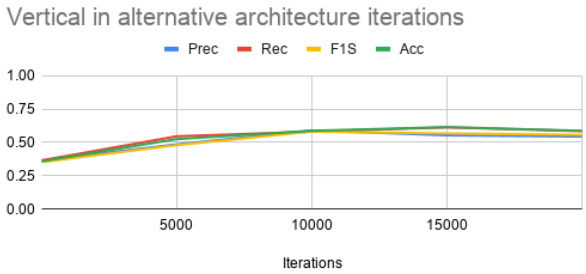

Figure B.5: Vertical action progress for alternative architecture.

Following with the navigation actions which are shown in Figure B.4 and Figure B.5. This case is more promising showing that is able to understand the navigation. All the metrics are near or beyond the 0.50 mark. The graph behavior is similar increasing until 10000 at which point it stabilizes, however it stabilizes at a lower number than the simple architecture.

B.1.2.C Closing remarks

This architecture at first glance seems to be more promising than the simple architecture because it includes more inputs. Which on first though would appear to help the performance because the agent has more inputs to understand the human patterns. In reality the performance was worse that the to the more simple architecture. A fair comparison between this architecture and the simple is difficult because the test were not as extended. However, none of the games tested on the simple architecture fall to such a low quality plus none of the agents produced the weird behavior. So, no further exploration was made and the architecture was discarded.

B.1.3 Recurrent Neural Network

Another alternative was a recurrent ANN topology. A recurrent ANN seemed like another natural answer for Pic-a-boo because the agent would be able to remember taking certain photos or remember the position of the targets. Short test were done using this option. However, an issue with UMLAT prevented further development [25]. UMLAT was unable to start the model after training so it could not be used.

B.2 Training procedures

To initiate the training of agents a couple of extra dependencies were required. This is because the training is not done in Unity but in TensorFlow. The dependencies were **Python 3.6** and the **mlagents 0.9.3** environment. To install python is only necessary to download the installer from here and following the steps. Python comes with a package manager called “pip”. “Pip” helps to install packages available in

their servers, which is the case for **mlagents 0.9.3**. To install the correct environment is only necessary to open the console of your respective system and write the following command:

```
pip install mlagents==0.9.3
```

Then to start the external training service the following command is inputted:

```
mlagents-learn config/online_bc_config.yaml --train --slow
```

Before starting the external service the academy control flag was checked as you can see in Figure B.6.

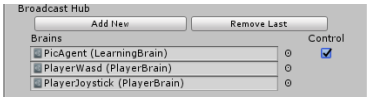

Figure B.6: Academy control flag.

Finally, the play button was pressed starting the simulation. The simulation is a simplified version of Pic-a-boo with simple textures and no User Interface (UI). Given the fact that there is no UI part of the information presented to the player in a normal game is not available. For example: Flash cool-down and remaining targets. There were two details that were modified from Pic-a-boo to this environment which is the collision side information. In the UI Pic-a-boo showed from which side the player avatar collided, alongside this a vibration was sent to the control if available. Given the fact that this environment has no UI the avatar was briefly revealed as to give similar information to the player. Second given that the ANN topology did not use the visibility information from the other players the lamps were not implemented. Apart from this the game-play remains the same take 3 different pictures of the other players. Once all the player were captured or 5000 steps has passed, the game ends and reset itself beginning another game. This is repeated until 10000 steps were achieved which stopped the execution.

With the execution stopped the generated model could be found at the root folder of the project. This models folder then was dragged unto the asset folder of the project making it available in the Unity editor. With this model inside the editor it could now be tested.

B.3 Pic-a-boo implementation details

Commencing by the **Academy**, Pic-a-boo does not require any reset logic so it only necessary to create a Unity empty object and attach a script that inherits from the **Academy** parent class. For the agent entity Pic-a-boo already possessed agents so a new script was attached to this object, this script is a copy of the script used in PAG with the difference that it does not contain any reset logic it only collects observations and receives actions from the academy. To finish the implementation a new **Brain** asset

was created and assigned to the agents, The **Brain** was configured exactly like in Figure 4.4. Given that the **Brain** has the exact same configuration as the one from PAG the **Brain** can reference the model trained in PAG. Now that all the test are done Pic-a-boo is ready to receive models to use, however Pic-a-boo is unable to train models only the PAG environment its capable of creating new agents. In addition, Pic-a-boo doesn't have any in-game form of selecting a model, meaning that until that options its implemented Pic-a-boo can only have one model playing at a time.

B.4 Repository location

The code for this project is located in here. The code was developed using Unity version 2019.2.0b9, to run a quick test, the agents red yellow and blue must be enabled along with the teacher, the test agent must be disabled given the fact that this agent purpose is only to gather testing data. In the "playerAgent" script variable named Brain the desired brain asset must be assigned to be able to see it in the match.