MASTER THESIS

Deep Learning for Pattern Recognition in Movements of Game Entities

Author:

Karel Lommaert

Supervisor:

David Hörchner & Robbie Grigg

This thesis is submitted in fulfillment of the requirements
for the degree of Master Game Technology

In the

International Games Architecture and Design
Academy for Digital Entertainment

21/06/2019

Declaration of Authorship

I, Karel Lommaert, declare that this thesis titled, “Deep Learning for Pattern Recognition in Movements of Game Entities” and the work presented are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 21/06/2019

Breda University of Applied Sciences

Abstract

International Games Architecture and Design
Academy for Digital Entertainment

Master Game Technology

Deep Learning for Pattern Recognition in Movements of Game Entities

By Karel Lommaert

Current generation game artificial intelligence lacks the capability to understand and adapt to player behaviour. The aim of this research is to investigate if a neural network can recognize patterns in player movements and if these patterns indicate their playstyle. That way, the developer can use this information to change the behaviour of artificial intelligence agents accordingly. Following the success of recent studies applying spatio-temporal pattern recognition for play classification in sports, a deep convolutional recurrent neural network is built to analyze player movements in games. Data to train the network is collected from a simple tank shooter game, in which two game modes were introduced, each enforcing either a defensive or offensive playstyle. The network is trained to recognize which player is playing, which action they are performing and which game mode they are playing. For each classification task, accuracies of 53% (nine classes), 52% (six classes), and 70% (2 classes) respectively are achieved. From these results, it is deduced that players exhibit unique behaviours that can distinguish them from other players. Furthermore, the network shows potential for the recognition of game-specific actions. Lastly, the technique displays some ability to identify which game mode the player is playing. Therefore, it is concluded that using deep learning techniques to recognize playstyles can facilitate creating a more dynamic experience for the player.

Acknowledgements

Throughout the making of this project and thesis, I have received a great deal of support.
First of all, I'd like to thank my supervisors, David Hörchner and Robbie Grigg, whose feedback helped with formulating the research question and steered the project towards a clear goal.
Besides my supervisors, I'd like to thank my teachers. Especially Thomas Buijtenweg, who was always available for feedback or questions.
Furthermore, I am grateful to all my classmates for their continued help whenever needed. I'd like to point out Brent Op de Beeck in particular, who helped me stay motivated.
Additionally, I'd like to thank my friends, who helped me whenever they could, but also allowed me to forget about the project once in a while.
Last but not least, a heartfelt thanks goes to my parents, who supported me through it all, doing whatever they could to help and motivate me.

Contents

List of Abbreviations

AI	Artificial intelligence
BPTT	Backpropagation through time
CNN	Convolutional neural network
GA	Genetic algorithms
ML	Machine learning
NN	Neural network
MLP	Multi-layer perceptron
LSTM	Long short term memory
RCNN	Recurrent convolutional neural network
RNN	Recurrent neural network
VRAM	Video random-access memory

Chapter 1: Introduction

Deep learning has completely changed how problems in multiple areas of computer technology are being tackled. Recently, the use of deep learning in real-time applications has grown in popularity, including in games. They offer solutions to various different problems that hindered game development in the past. The ability to train neural networks (NN) in an offline setting, but apply them in real-time, allows developers to benefit from their many applications without having to worry about the computational power required for the training.

1.1 Adaptive AI in Games

Adaptive artificial intelligence (AI) already exists in games. Over the past few years, developers have experimented with different techniques to make their AI agents more dynamic and flexible.

However, most of these techniques don't actually change the behaviour of the AI agents. They look at how well the player is doing and then adjust the difficulty based on that. Their actions, rotations, movements, and strategies remain static. In some games, this is exactly what the developers want. Nevertheless, creating an AI agent that can adapt its behaviour based on how the player interacts with it may create more compelling game experiences (Elkin, 2013), especially in today's complex games where "dumb" AI can ruin one's immersion. For example, when an opponent AI agent keeps taking the same route despite being punished for it every time, or when an allied AI agent forces a certain approach to a situation (i.e. attacking directly, while the player wanted to take a stealth approach).

1.2 Deep Learning for Adaptive AI

Although NNs have been successfully used in shipped games such as Hello Neighbor (2017), Forza Horizon 2 (2014), and NERO (2005), the techniques used are not applicable to most genres of games. This study attempts to prove that it is possible to use deep learning in a more generalized fashion. NNs may be capable of identifying patterns in the movements and actions of players. These patterns could be indications of the playstyle the player is using (in this context, playstyle refers to how a player approaches the objectives within the game). If so, the network could analyse in real-time which playstyle a player is using, which would allow the developer to adapt the agent's

behaviour to the player's playstyle at runtime. This would not only make the content of the game more dynamic; it would also force the player to switch between playstyles and use multiple strategies to achieve their goal. Thus creating a more dynamic experience as well as a continuous challenge.

1.3 Thesis Structure

Chapter two will cover earlier research done on adaptive AI and pattern recognition in spatio-temporal data in the different fields, while chapter three will go over how the experiment will be conducted, and how its success will be evaluated. The execution and any changes made will also be explained. Chapter four covers the results of the experiments. Chapter five discusses the results found in chapter four, covers its applications and some of the limitations of the experiment. Finally, chapter six contains a quick summary, with the conclusion and potential future work.

Chapter 2: Background Information

2.1 Goal of AI in Games

Previous research has shown that, among other things, the challenge in a video game is what brings enjoyment (Malone, 1980). However, despite modern day game AI's being quite advanced, their behaviour remains static and their scripts often get very complex and hard to maintain (Brockington & Darrah, 2002). Using deep learning to create an adaptive game AI has been suggested multiple times before. But as Tozour (2002) suggests, the goal of a game AI agent is not to be as effective at defeating the player as possible. Their goal is rather to pose a challenge that can be overcome by the player. Thus, using deep learning to create an unbeatable AI is not useful for the development of game AI opponents. Moreover, it is very difficult to debug an agent that was trained with deep learning (Woodcock, 2000). For these reasons, deep learning techniques have thus far rarely been used to drive AI agents in games.

2.2 Current State of AI

Instead of using deep learning, most developers use techniques such as finite state machines and behaviour trees, among others. The majority of these techniques usually face two major problems, namely the problem of complexity and the problem of adaptability (Spronck, Sprinkhuizen-Kuyper, & Postma, 2003). The complexity problem refers to the scripts getting very large and complicated, which often results in unforeseen situations and exploitable behaviour. Both of these can ruin the challenge or player experience. The problem of adaptability is a direct result of the static nature of these techniques. They are not capable of changing their own behaviour, and thus cannot adapt to what the player does.

Despite this, adaptive AI has made its appearance before in games. An example of this can be found in Middle-earth: Shadow of Mordor (2014), which featured the "Nemesis" system. This system featured agents that would increase in strength if they managed to beat the player, to enhance the feeling of rivalry between the player and the NPC. while on a surface level it may seem like the agents adjust themselves, what the system actually does is remove an exploitable weakness of the agent with every iteration (Hoge, 2018). Another example of adaptive AI used in games can be found in Left 4 Dead (2008), in which the game pacing is adjusted algorithmically. Depending on

how intense the gameplay experience is (calculated using the player's health and amount of enemies), the AI will spawn more or fewer agents. While this does create a more dynamic experience, it only adjusts the difficulty on the fly. The AI's strategies do not change and therefore do not solve the problem of adaptability. Additionally, a game getting harder does not mean the player enjoys it more because the balance between challenge and fun depends on the profile of the player (Elkin, 2013).

One of the most interesting examples of adaptive AI in games is found in Hello, Neighbor (2017) by Dynamic Pixels. In this game, the player has to try and infiltrate into an AI agent's house. However, the agent tracks how the player moves around the house. Using machine learning techniques, the AI will then learn where to place its traps to try and catch the player. Thus, Hello, Neighbor proves that it's viable to use machine learning in real-time applications. However, the game is built around this mechanic. Traps and other interactables are placed in designated positions. These positions are used to determine where the AI agent will move towards. This technique cannot be directly generalized to be used in other genres of games. Crafting larger worlds like this likely requires a lot of manual work. Besides that, only one AI agent is present in the game. Thus only one simple network has to be trained, making it computationally cheap in comparison to most other games.

2.3 Reinforcement Learning

Hello, Neighbor (2017) was certainly not the first time deep learning was introduced to gaming. Games are often used as a tool to measure the intelligence of AI, as they simulate complex environments often with many influential decisions the player can make. Reinforcement learning is by far the most used DL technique used to create AI agents that play games. In reinforcement learning, an agent tries to maximize a reward function and learns (through trial and error) which actions to use in which scenarios.

A reinforcement learning network takes the game-state as input. This can be achieved in multiple ways, and the most optimal way depends on the task the agent has to learn (Risi & Togelius, 2015). The output of the network specifies which action the agent should take. Thus, it learns a mapping between the game environment and the preferred action in said environment.

This technique has been around for quite some time but has been gradually gaining more and more interest as advancements are being made. In 2013 Google Deepmind created a neural network capable of beating Atari games. In some cases, they were trained to superhuman levels, only being given pixel information as input (Mnih et

al., 2013). Later on, in 2015, the AlphaGo program managed to create a reinforcement learning agent that beat professional human players at the very complex game of Go (Silver et al., 2017). Similarly, OpenAI has been working on a neural network that plays the popular online battle arena game Defense of the Ancients 2 (2013), also known as Dota 2. In 2017, they demonstrated a bot that was able to defeat one of the best players in a one versus one matchup. They then continued to work on what is called OpenAI Five, a full team of AI-controlled agents. In 2018, they managed to beat semi-professional teams (Rodriguez, 2018). These achievements proved that reinforcement learning is capable of dealing with complex environments and long term decision making.

However, all these advancements have not actually found their use yet for game development. The main issue with reinforcement learning is that agents can easily reach a superhuman level. These agents can be considered as adaptive, as they'll always end up making the correct decisions, no matter how the player plays. But as mentioned earlier, hard AI does not equal fun AI. Using reinforcement learning to control AI agents in games has been suggested, but finding a balance between fun and smart AI is thus far an unsolved challenge.

2.4 Possibilities for Adaptive AI

When looking at the options for real-time adaptation through deep learning, there are 2 main categories: Online and offline training. Online training would always be preferable. Online training allows the network to adapt to unseen situations and learn directly from the player. However, with today's state of games, it is hard to preserve enough computational resources to train a neural network as the game is running. Although it is certainly not impossible, as proven by Dynamic Pixels (2017).

Utilizing offline learning is also possible, provided that the training data is acquirable for initial training. However, the network can only learn from what is shown in the training data. If there is not enough training data, the agent might perform poorly in new situations. Moreover, as mentioned before, this might result in an unbeatable AI. Because of the difficulty of debugging a neural network, manual adjustments to try and simplify it are hard. The main advantage of offline learning is that once trained, a neural network requires little computational power, which is perfect for any real-time application.

2.4.1 Online learning techniques

Academic research has been conducted towards the use of online learning in games. Antonio Ricciardi & Patrick Thill (2008) show that in simple games with few actions -such as a 2D fighting style game like Tekken (1994) or Street Fighter (1987)- it is possible to record player actions and train an AI agent at runtime against them utilizing the collected data. By using a modified version of reinforcement learning, they were able to achieve promising results. The research was conducted using a demo game in order to have access to all necessary data at runtime. This does imply that the game was lacking in several areas that would affect performance. Another experiment regarding online learning involved what is known as dynamic scripting, a technique developed by Spronck et al. (2003). Dynamic scripting uses an unsupervised neural network and trains it using genetic algorithms (GA). GAs use the "survival of the fittest" principle. Starting with a randomly generated generation, after which each new generation is based on the previous. Using techniques such as selection and crossover, traits from the most successful agents are passed onto the new generation. The success of an agent is assessed by a fitness evaluation function. By applying mutation, agents of new generations also receive random traits to avoid getting stuck at a local maximum fitness (Carr, 2014). At runtime, dynamic scripting maintains multiple rulebases: one for each type of AI controlled agent or entity. These agents are then matched up with human players. Using information collected during these encounters, a reinforcement learning inspired technique is applied to update the rulebases. This way, high priority rules become more prevalent over time. These new rulebases are then used to create the scripts that control newly generated agents. Spronck et al. (2003) found that by using dynamic scripting, agents would outperform players after 30 encounters on average. While this technique showed promising results, it may have unpredictable behaviour when used in a larger sample, and it remains hard to debug. Following that, it also relies on the usage of generations. This implies that multiple encounters are needed before the agent can properly adapt to the player, which is, at least up until this moment, a major limitation of dynamic scripting. Many genres of games do not have content that allows multiple encounters with a single type of agent. For example, in Monster Hunter (2004), the player must defeat a wide variety of monsters but fights most types of monsters only once (unless multiple attempts are needed).

Majchrzak, Quadflief & Rudolph (2015) did expand on this concept. They used dynamic scripting for a fighting game but updated the network every four seconds, after

which they generated a new script for their AI agent. This technique, although successful, requires a constant evaluation of fitness, which is not possible in every genre of game. In a fighting game, it is relatively easy to assess how well an agent is doing (did it do damage/did it lose health); however, in most other games this is harder to measure.

2.4.2 Offline learning techniques

Offline techniques have also been suggested for adaptive AI. Offline learning suffers from one major issue, namely that it can only learn to adapt to anything it is shown in the offline environment. Yet, there are still many possible applications of offline learning. It can be used to find holes or exploits in more traditional (static) AI, by having a neural network train to try and beat the game's AI according to Spronck, Sprinkhuizen-Kuyper & Postma (2002). Furthermore, they found that offline learning can be used to create new tactics that a static AI can use. Ponsen, Spronck, Muñoz-Avila & Aha (2007) applied this knowledge in combination with the earlier mentioned dynamic scripting technique. In their experiment, dynamic scripting was applied to a real-time strategy game called Wargus, a mod for Warcraft II: Tides of Darkness (1995). The dynamic AI controlled player was given a list of actions it will use during a match. After a match is played, the actions used are evaluated using a fitness function. The fitness depends on the result of the match (win or loss), but it also depends on the fitness of every separate action used. The fitness of an action is evaluated by the direct results of this action (calculated with in-game statistics such as military units killed etc). Using GA, the most successful actions are then used in the next match. The dynamic AI was matched up with four different scripted (static) AI's, each using a specific tactic that was popular among human players. In the first condition, the possible actions that could be chosen by the dynamic AI (Controlled by dynamic scripting) were predefined by the designers. With this approach, the dynamic AI was able to successfully adjust to two of the four scripted tactics. Ponsen et al. (2007) concluded that the predefined list of actions most likely did not include a way to deal with the two tactics it lost to.

In the second approach, offline reinforcement learning was applied against each of the 4 tactics separately, and then new actions were extracted from these results manually. This resulted in an increased effectiveness of the dynamic learning, but the dynamic AI still failed against the same 2 tactics. In the third condition, they completely stepped away from manual selection. The predefined list of actions the AI could choose from was completely made from actions found to be effective during the offline

reinforcement learning. This once again resulted in an increased effectiveness, and the AI was able to successfully adjust to 3 out of the 4 tactics and showed an increased performance against the unbeaten tactic. This experiment proved that tactics generated through offline learning can be extracted and used in real-time to adjust AI entities. The experiment was limited by only being tested against static opponents. They did, however, test the AI (under the third condition) against unseen static scripts and had successful results there as well. Therefore, they concluded that reinforcement learning also found more generalized tactics that can be applied in a wide range of situations. These findings are quite promising but still struggle with the original problem of dynamic scripting, namely that it needs iterations in order to adjust to the player.

2.5 Pattern Recognition in Spatio-Temporal Tracking Data

Thus far, all implementations of adaptive AI lack the capability to identify how a player is playing at runtime. In order to do this, the AI has to be aware of the implications of the movements and actions of the player's agent(s). Movement analysis has to date not been done in games. There has, however, been plenty of research on analysing movements in sports. Thanks to the deployment of player tracking systems in various sports such as soccer and basketball, the amount of spatio-temporal tracking data has increased significantly. This has facilitated the analysis of player movement patterns. Bialkowski, Lucey, Carr, Yue, Sridharan,& Matthews (2015) were able to train a neural network to identify teams given the spatio-temporal tracking data. By applying k-means clustering, different styles were recognized, which in turn could then be used to identify teams given unlabelled spatio-temporal data.

Similar research was also conducted in basketball. Spatio-temporal tracking data was used to train a network to classify different strategies (Wang & Zemel, 2016). The acquired data was converted into pictorial data: images in which pixels represent the positions of players. This way, it could be read by using a convolutional neural network (CNN). This conversion was done because the spatial data came in the form of coordinates (x, y). These are numerical values and cannot be logically interpreted by a neural network. The network tries to map the input sequence to a type of play. A position with high value coordinates does not necessarily reflect a higher signal for a certain play call.

2.6 Convolutional Neural Networks

CNNs are designed to process images (Krizhevsky, 2012). Traditional multilayer-perceptrons (MLP) (the "vanilla" neural networks) are capable of processing images but are not as efficient at it. MLP's suffer from the resolution curse: as the resolution of the input image increases, the amount of input neurons also increases, but at an exponential rate. CNNs are specifically made to enhance the efficiency of analyzing visual imagery. CNNs distinguish themselves from MLP's by using convolution layers. This layer is inspired by the behaviour of the visual cortex. It applies a filter to a small area of the input image. This filter recognizes specific features in the image. Early layers find features such as edges, while deeper layers combine earlier filters to find more complex features of an image, such as shapes and textures. CNNs also make use of pooling layers. These layers are designed to reduce the dimensions of previous layers while retaining important information. Downsampling the resolution greatly reduces the computing power necessary to train the network.

While CNNs are great at image classification tasks, they cannot deal with sequenced data. The spatio-temporal tracking data Wang and Zemel (2016) used came in the form of sequences of positions. As standard CNNs are purely feed-forward networks, they can only look at images one at a time. This means they have no way of retaining information of previous samples when a video (or any other sequenced data) is used as input.

2.7 Recurrent Neural Networks

In order to have a neural network make sense of sequential data, it somehow needs access to the whole sequence. This is what recurrent neural networks (RNN) were designed to do (Ghewari, 2017). The output of an RNN at any point in a sequence depends on the previous computations in that sequence. The RNN does this by making use of a hidden state. This state can be seen as the memory of the network as it captures information about every step in the sequence. The hidden state is recalculated at every step in the sequence by using the previous hidden state and the current input. The output of the network is then calculated using the new hidden state. By using this hidden state, they cannot just use sequences as input, RNN's can also have a sequential output. Sequential outputs are useful for prediction tasks such as word prediction. Unlike normal neural networks, neither the input nor the output of an RNN has to be of a fixed size. This is most useful as the movement data sequences produced by a player can have any size. Vanilla RNNs do often fall victim to the vanishing gradient problem (Bengio,

Simard & Frasconi, 1994). This problem is a result of the backpropagation algorithm and causes the learning rate to become negligible if the network is too large. This is a problem that can appear in any neural network, but it is most prominent in RNN's, as they use what is called "backpropagation through time" (BPTT). BPTT allows the network to use backpropagation on sequenced data such as a time series. But it also means that for every step in the sequence, the gradient gets exponentially smaller. This problem can be solved by using a technique called long-short term memory (LSTM). This technique was introduced by Hochreiter and Schmidhuber (1997). LSTM not only solves the vanishing gradient problem, but it also enhances the capabilities of RNNs to learn long term dependencies (Gers, Schraudolph & Schmidhuber, 2002).

2.8 Imitation Learning

To train these deep learning networks, a large quantity of training data is needed. Wang & Zemel (2016) were able to use the spatio-temporal tracking data provided by SportVU. However, such data is not (publicly) available for most games. Gathering the data itself should provide no issues, as it just comes down to storing coordinate data for relevant entities. Yet, to get large quantities of this data during development, numerous playtesters would have to play the game for hours on end. Alternatively, a technique called imitation learning can be applied. In imitation learning, a network is trained to replicate human behaviour. It can be used to speed up the learning process of other machine learning techniques (Thurau, Sageger & Bauckhage, 2004) as it reduces the search space for possible solutions to a problem. It has been used successfully in games before (Forza Horizon 2, 2014), and the potential of the technique for game development is slowly getting recognized (Hussein, Gaber, Elyan & Jayne, 2017). The game engine Unity (Version 2018.3.0.f2, Unity Technologies, 2018) includes libraries that allow imitation learning to be directly applied to a player playing the game (Juliani, Berges, Vckay, Gao, Henry, Mattar & Lange, 2018). By training different imitation learning AI agents, and using them to simulate gameplay, data can be generated with a limited amount of original playtesting gameplay.

2.9 Supervised Learning

Wang & Zemel (2016) used labelled data to train their networks in a supervised manner. Supervised learning is based on human feedback. A training data set is given to the neural network, where each piece of data in that set already has a corresponding output. By processing this training set, a neural network can teach itself a set of rules

that defines how input is mapped to output. This mapping is then used to predict the outputs of unseen data. In general, when dealing with classification problems, and plenty of labelled data is present, supervised learning is preferable. Labelled data provides clear criteria for model optimization, thus making supervised learning, in general, more efficient (Cord, Delany & Cunningham, 2008).

2.10 Unsupervised Learning

Wang & Zemel (2016) labelled their data manually, which requires a lot of work, but as basketball has been around for a very long time, different strategies and plays have developed over the years, and a clear terminology exists to differentiate them (FIBA, 2014). Therefore, despite the labelling process being labour-intensive, it is accurate. In games, this is not always the case. Long-standing games may have existing strategies and similar terminology, but generally speaking, labelling data manually is suboptimal.

Even without labelled data, a neural network can still be trained for pattern recognition. This process is called unsupervised learning. Unsupervised learning is primarily used for pattern discovery (Boutros & Okey, 2005). Most variants of unsupervised learning try to cluster data together based on similarities it finds. There is a wide variety of clustering algorithms, but the most prevalent are k-means clustering, hierarchical clustering, and self-organizing maps. Each algorithm has different advantages and flaws. The most optimal technique depends on the input data used and the expected results.

Chapter 3: Methodology

3.1 Research Question

The goal of this research is to find out what kind of patterns can be found in the movements of a player using deep learning. In particular, it aims to classify movement patterns that indicate different styles of gameplay.

The hypothesis is that neural networks are capable of classifying sequences of movement data if labelled data is available. However, it may be hard to label movement sequences as a particular playstyle. The method to achieve this may differ per game. Yet, if supervised learning yields successful results, this opens up opportunities for research towards unsupervised learning in this area.

3.2 Data Collection

Given it presents little difficulty to store positional data of game entities at runtime, it is possible to quickly acquire plenty of training data. It would be very beneficial if we could have a neural network distinguish different gameplay behaviour patterns given this data. As Wang et al. (2016) were able to classify different kinds of basketball plays using spatio-temporal tracking data, it could be possible to do the same in video games.

In an ideal situation, the spatio-temporal data can be collected near the end of development of a game, as the data can then be used to train agents which can be used when the game releases. However, data collected post-release is still useable for further training, or initial training if this was not done during development. The sole purpose of this experiment is to deduce what kind of patterns can be found in the data, and how these patterns are related to the playstyle of the player. Therefore it does not matter if prerelease or postrelease data is used.

3.2.1 The demo game

In order to test whether or not gameplay patterns can be extracted from spatio-temporal data, a game where movement is a major part of the gameplay is required. Furthermore, access to the source code of the game would be optimal to retrieve the positional data. Lastly, the game must have enough depth to allow for different playstyles. For these reasons, Unity's open-source Tanks! game (2015) was used for the experiment. The game features a deathmatch between two or more tanks.

Each tank can move forward and backwards, and turn around its axis. Additionally, a tank can fire shells that explode on impact. The distance these shells fly depends on how long the attack was charged. The goal is to eliminate all opponents. The environment in the game is kept static, to avoid creating an abundance of movement data.

The game was slightly modified to give it more depth. An ammunition system was added, along with two kinds of pickups. One that heals the consumer, and the second giving them a short period of infinite ammo. Two additional game modes were introduced as well. The *timer* game mode introduces a time limit in which the player has to eliminate all the enemy tanks. The *low health* game mode makes the player start the game with only one health point remaining. These game modes were added to enforce different playstyles on the player.

The game was extended using Unity's machine learning (ML) agents toolkit to allow training of reinforcement and imitation learning agents. This toolkit is built on top of the open-source library Tensorflow and allows the user to directly train ML agents in the editor, or a build of the game. It supports reinforcement learning and imitation learning, among other machine learning techniques.

3.2.2 Collecting movement data

At runtime, the coordinates and rotations of all relevant game entities are saved at a fixed timestep. A sequence of coordinates of any length can be saved at any point in time. Sequences of full matches are saved whenever the match is concluded (either the player dies or all enemies are eliminated). These sequences may have to be sliced, however, as they may prove too long to analyse using state-of-the-art RNN techniques. At the end of the play session, these sequences are saved to a file. Additionally, whenever a certain event in-game happens, such as a pickup getting consumed, or the player getting hit, the timestamp of this event is also saved.

Considering the game is not a commercial game, there is no pre-existing movement data. To gather the data, a small group of people with varying backgrounds in gaming will be asked to play the game. Due to the nature of neural networks, a large amount of training data is preferable. For this reason, and because of the lack of available players, additional data may be required. This data can, in theory, be generated through imitation learning. This NN technique analyses how a player plays the game, and tries to imitate what they do. Thus after training several AI agents using different players, the movement data of the trained agents can be used to represent the different players. By simulating games with these imitation-learned AI agents, any amount of data

can be produced. Though it must be considered that this data might not be the most accurate, and may vary slightly from actual player movement data. Imitation learning is supported by the Unity ML agents toolkit. In order to use it to gather training data, a performant configuration for the network must first be found to train the agents. Many factors can have an influence on the learning rate of a neural network. Considering it will be difficult to acquire a lot of data from a single player in a non-commercial game, an optimal setup is required so the toolkit does not require too many demonstrations to train an accurate imitation agent.

3.2.3 Representing the data

Once the data is collected, the goal is to use it as input for a deep NN. However, the format of our data comes in the form of position coordinates. As explained in chapter 2.4, coordinates are numerical values and cannot be logically read by a NN. Therefore, the data must be represented in a different way. As suggested by Wang et al. (2016), pictorial representations of the positions over time can be used, as they are comprehensible for humans, and NNs can more easily interpret them. However, this does limit the experiment to two dimensional (2D) games (or 3D games that constrain their movement to a 2D plane), but this can later be expanded upon by using a different representation of the coordinates. The path of individual entities can be turned on or off in a sequence, allowing different levels of depth in the pictorial representations (i.e. only showing the player's trajectory without information about enemies etc.).

3.2.4 Conversion to pictorial data

As the learning will not happen at runtime, any software can be used to create the pictorial representations. However, if it turns out the network is capable of identifying patterns in the movement, a method to create these pictorial representations at runtime will be required in order to identify the player's playstyle at runtime.

For this experiment, a simple tool was created that converts the positional data collected at runtime into pictorial data. This was done using Unity's built-in Texture2D class. This class is a simple container for pixel values with some added utility such as being convertible to other image file formats. At the start of a playing session, an instance of this class is created and a layout of the current level is generated. Any geometry the player can collide with is drawn into the texture in black, while the remaining pixels are coloured white. Once the session is concluded, all saved positions are converted to paths and added to the texture frame by frame. All agents are

represented by a small circle. Other relevant entities such as bullets are also drawn using a smaller circle. Each type of entity has a distinct colour. Every texture created this way, is then converted to a PNG and saved locally. The level of detail of these images can be adjusted if necessary but is kept as low as possible to decrease file size and training time (see Appendix A.1).

Another advantage of using Unity's built-in Texture2D class is that these can easily be created and edited at runtime and can, therefore, serve as input for the network once it is trained. It must be considered, however, that this has performance implications. For this experiment, this likely won't be an issue, but it must be noted that this is certainly not the optimal way to convert the data. Ideally, a method to directly use positional data as input should be used, causing very minimal loss of performance.

3.3 Recurrent Convolutional Neural Networks

The saved image sequences will serve as input for a recurrent convolutional neural network (RCNN). Considering the aforementioned sequences are often built up over longer periods of time, it is important for the neural network to be able to find long term dependencies. As was mentioned by Felix et al. (2002), LSTMs are capable of storing information over longer periods of time than regular RNN's. Therefore, the LSTM variation of RNN's will be used.

Furthermore, the coordinate data was converted towards pictorial data, thus the network will be dealing with images as input. As explained by Ghewari (2017) CNNs are designed to process images.

Several techniques exist to combine the capabilities of CNNs with those of RNNs, and doing so is supported by most major DL libraries. The order of the layers in an RCNN can have a major impact on the results, as it can change how it interprets the data. For example, a network can be modelled to interpret an entire sequence at once or to evaluate it frame by frame instead. Multiple NN setups may have to be tested to achieve optimal results.

3.4 Training the Neural Network

Once movement data is acquired and converted into pictorial data, it can be used as the input for the RCNN. In a perfect scenario, applying unsupervised learning to the data would result in the network categorizing each sequence depending on the playstyle used by the player. However, this is not very likely but may still result in interesting movement patterns being found.

The second option is to label the data and apply supervised learning to it. Labelling may prove difficult, considering there is no clear existing terminology or definition for playstyles in the Tanks! (2015) game. Yet, due to the uncertainty of the results of applying unsupervised learning, the experiment will be conducted using supervised learning.

There is no clear way to label data according to playstyle other than doing it manually, which would be an immense amount of work, and would also be error-prone. Thus, for this experiment, data will be labelled automatically. Three options for labelling the data will be tested.

3.4.1 Labelling per player

Bialkowski et al. (2015) trained their network to categorize sequences per player. Labelling data per player may not be difficult in the case of a game. However, even if the network is able to identify which player is playing, these categories do not necessarily represent playstyles. Nevertheless, if the network can successfully classify sequences per player, it still proves that players move through the levels in a unique way. This, in turn, would open up the possibility to research what exactly differentiates two players from each other.

3.4.2 Labelling per game mode

As the goal is to identify different playstyles, and there is no clear way to mark a sequence as one particular playstyle, a different approach is needed. For this reason, the aforementioned game modes were introduced. By restricting the time a player has to complete the level, a more offensive approach is needed. On the other hand, by putting the player at a health disadvantage, they are forced to play more defensively. A sequence can then be labelled as being one game mode or the other.

3.4.3 Labelling per action

A third option to label the sequences is to save a sequence whenever an action happens in-game. In the example of the Tanks! (2015) game, a sequence can be saved whenever the player consumes a pickup, an enemy gets killed or a high amount of damage is dealt in a short timespan, among other scenarios. This would allow a trained network to identify or even predict the action the player is doing based on their movements. Analysing how often and in which scenarios a player chooses to go for an action may also indicate the usage of a certain playstyle.

3.5 Applying the Results

The results of the supervised learning will determine which kinds of patterns can be found in the movements of a player for this specific game. Nonetheless, it can be expected that the techniques used to conduct the experiment can be applied to other games as well considering movement is a major factor of gameplay in the majority of entertainment games.

While there may be several usages for each kind of pattern that can be found, the main goal of this research is to find out whether or not we can use these patterns to deduce the player's playstyle and create an AI that adapts to it. If the network is capable of finding gameplay behaviour patterns, and thus being able to recognize them at runtime, this information can be exposed to the game designers and used when implementing an AI agent's behaviour. To avoid this new information further increasing the complexity of scripts, an approach using utility AI is suggested. Utility AI is a system that chooses which action an agent will take by comparing their expected utility values. These values are calculated depending on the game state (Evans, 2015). By increasing or decreasing the expected utility value of specific actions depending on the player's current playstyle, an agent can be steered towards a more fitting playstyle against that player. For example, when playing versus a defensive playstyle, all offensive actions can be given an increase in expected utility value. This adaptation can be implemented by giving each action a designated expected utility curve per variable per playstyle. This approach minimizes the complexity problem and partially solves the problem of adaptability while remaining debuggable and easy to tune by the developer.

3.6 Implementation of the project

Given the Tanks! (2015) game was made in Unity, the most logical solution to save and convert the data was to do it in the engine. Due to the performance consequences of saving images at runtime, this was done whenever the player finished playing. At the end of a session, a folder structure was created on the hard drive, where each sequence of images was saved to a directory specified by its label. The actual labelling process was done in Python, where the sequences got converted to a 5D array of integers. The first dimension specifies the colour of a pixel. The following two dimensions are used for the width and height of the image, while the latter two dimensions specify the number of frames per sequence and the number of sequences in the training set respectively. This array was then used as the input of the network.

Alongside it, a second 1D array was also created to hold the label of each sequence, which depended on where in the folder structure the sequence was found.

3.6.1 Imitation learning

As explained earlier, additional training data was to be created through imitation learning. The ML-agents toolkit was added to the project, and multiple input setups were tested to train both reinforcement and imitation learning agents. Initially, raycasts were used to scan the nearby environment, calculating the distance to buildings, enemies projectiles and pickups around the agent. However, this method proved to be inefficient, as an abundance of rays would have to be shot to handle the full environment. Especially if small colliders were far away from the agent, the likeliness of the collider ending up between two rays was too high. Furthermore, it also meant an agent could only see around itself, while a player plays with "perfect" information (they see the full map and all of its entities at all times). If an imitation learning agent was to accurately imitate player behaviour, it would need the same level of information. To achieve this, angles and distances to specific targets were added to the input. A minimal amount of rays was kept to allow the agent to scan for nearby buildings. This way, even if an obstacle obscured the path towards an opponent, the agent still knew its position. For each target, a raycast was added to show whether or not any obstacles were in between the agent and the target. Still, this technique failed to get good results, most likely due to the inability to learn the layout of the map, and therefore still not having a clear path towards a target if any obstacle was in the way. Considering the issue with both of these methods was the agent having imperfect information, the last option that was tested was to use a camera as input. Theoretically, this gives the agent the same input as a player would receive. However, the major drawback of this approach is that image inputs take a lot longer to process (despite using a simplified, downscaled sight). Considering only short demonstrations would be available to train the imitation learning agents, this was an issue. Short tests were done to validate the usability of this approach, but they were not successful.

However, as proven by Forza Horizon (2014), imitation learning is still a reliable technique to mimic player behaviour, and should not be disregarded for future research. Forza Horizon does have the advantage of being a racing game, which in turn allows very simple inputs (in the form of raycasts in front of the vehicle) to accurately convey all necessary information in order to get human-like behaviour. OpenAI did prove that it's possible to do so in more complex games as well (Rodriguez, 2018). However, due to

time constraints and the inability to prove the equivalence between an imitation agent and the player, imitation learning was dropped for this experiment and instead, all data was gathered directly from real players.

3.6.2 Building the network

The RCNN was created in Keras. Keras is a high-level neural networks API for Python. It is capable of running on top of Tensorflow, among other DL toolkits. It allows for fast iterations of the training configuration and layer setup of a network. Moreover, in combination with CUDA, Keras has the ability to train networks using the GPU, which speeds up the training process significantly.

A sequential model was used, which is a linear stack of layers. The first layer of the network is a Gaussian noise layer. This applies additive zero-centred noise to the input images, which prevents the model from overfitting. Afterwards, in order to find spatio-temporal features, a recurrent convolutional layer was needed to process the sequences of images. Keras provided this in the form of a convolutional LSTM layer (ConvLSTM2D). This was followed by a batch normalization layer, which, as the name suggests, normalizes the activations of the previous layer at each batch. This process was repeated multiple times. The output of the final normalized LSTM layer then had to be flattened. Finally, the network was finished with a dense layer. It was given a softmax activation function as the network was used for classification tasks. The number of output neurons was equal to the number of classes. The model used a categorical cross entropy loss function. In cross entropy, the loss increases as the predicted probability diverges from the correct label (Shibuya, 2018). The categorical version of this function was used as the sequence labels are one-hot encoded. The model was compiled using the Adadelta optimizer. An optimizer handles how the network adapts using the output of the loss function. The Adadelta optimizer adapts the learning rate to individual features. Small updates are used for frequently occurring features, and large updates for less frequent features (Ruder, 2017). This allows the network to continue learning even after many updates have happened. This optimizer is widely used for sparse datasets.

Due to the small size of the dataset, additional data was created by the network. This was done in a generator function that rotates and shifts the images randomly. Keras models can take such a function as input, instead of a full dataset. It allows loading in small batches of data to train on step by step. This was done to prevent having to load in all the data at once and was also utilized to augment and enlarge the dataset. The

augmentations applied in this case also served the purpose of forcing the network to find patterns in the movements, rather than the positions of entities.

3.6.3 Training the network

The network was trained on a GeForce GTX 1070. Consequently, the training of the network was constrained by the limited amount of VRAM available (8 gigabytes). This meant that suboptimal training parameters had to be used. An optimal balance between all memory consuming variables had to be found. The pictorial representations were saved as images of 100x100 pixels, to try and capture detailed trajectories. However, this meant the full games had to be sliced into sequences of 15 frames each. This was lower than initially planned, and may have caused important information being lost. The batch size was limited to 4, which meant training was fast, but accuracy may have suffered from this. Despite these limitations, the network still managed to perform, but it is plausible that better results could have been achieved with better hardware.

Chapter 4: Details of Experiment and Results

4.1 Data Collection

Due to the inability to use imitation learning to generate data, all the training data had to be collected manually. This meant more players than initially expected would have to play the game. For this reason, the process of collecting the data was simplified. The game, along with all the systems responsible for saving, converting and labelling all the movement data was built into an executable to allow for easier distribution. To make the executable accessible and clear in its intentions, various user interface elements were added. Additionally, a 'session' option was added to the game. A session consisted of 33 games and was aimed towards players playing the game for the first time, with the goal of generating the necessary data. To allow the participants to familiarize themselves with the controls and the mechanics of the game, the first three games of a session did not save any data. Afterwards, the participants each got to play 10 games of both the regular and the two added game modes. As mentioned before, all data was saved during gameplay and labelled automatically. After the session, it got converted and saved to a folder on the hard drive.

To keep the format of the data consistent, only one level layout was used, and the number of enemies was kept static, three to be specific. This number was chosen to ensure at least a certain level of engagement but still allow inexperienced participants to beat the game. This consistency was necessary to prevent the network from basing its results on factors like map layout and enemy amount. As the goal of the network was to find patterns in movements, having as little variation in all other aspects was important for the experiment. For the same reason, participants were not asked to play against each other, which was the original intention of the 'Tanks!' game. Instead, giving each participant the exact same environment made sure their approach was purely decided by themselves, and not by the game, or the opposing player.

As explained in chapter 3.2.2, rotational data was also saved but not used to create the pictorial representations. Instead, the raw data of all sequences also got stored. Players received the option to also store all of this raw data into a JSON file (State Software, 2001). JSON stands for Javascript Object Notation and is a filetype that can easily be generated and parsed in C# (Crockford, 2002). This was done in case a different representation that includes rotation had to be created at a later point. Because of the large nature of this file, the time required to save a full session with raw data

became almost tenfold of only saving the pictorial representations. Thus, for the convenience of the player, this option could be disabled.

4.2 Verification Data

In order to be able to verify how well a network performs once trained, validation data is needed. Usually, a part of the training set is used for this purpose. However, in this experiment, being able to visualize what the network classifies as game mode A or B, could prove useful if the results are not as expected. Thus, the gameplay that was used as a validation set, was also recorded. This way, comparisons could be drawn between the results of the network and the gameplay. If a sequence was wrongly classified, it could still be compared to other sequences that were classified in the same category, possibly providing a better understanding as to why it was classified improperly. However, due to the file sizes of recordings, this was only done for the validation data set, and not for the training data set.

4.3 Participants

Nine participants were invited to play one of the aforementioned sessions. Each player was given a short explanation of what a full session consisted of, and was then asked to play the full session without breaks. Afterwards, additional games (in any game mode) could be played if the participant wished to do so. As previously mentioned, the participants had varying backgrounds in gaming. Two of the participants indicated to have little to no prior experience with games. Three said to have some experience but did not consider themselves gamers. Four participants said they played games on a regular basis. The ages of participants varied between 18 and 27 years. Each participant played at least one session of 33 games, but three of them were eager to play more afterwards. Most inexperienced participants had a hard time in their first few games, but soon learned the mechanics allowing them to still beat the game, or get close to doing so.

4.4 Final Data

A total of 372 games were played (see Appendix A.2.1). Most players, both inexperienced and experienced, seemed to enjoy the game. However, two of the participants felt too inexperienced with computer games and their controls to properly enjoy the game.

Inexperienced players proved to be clearly less adept at completing the game, dying more often and scoring fewer kills (see Appendix A.2.2). Players who played multiple sessions showed clear improvement, scoring more kills and dying fewer times in later games. The participants developed different strategies to tackle the game as they played. Notable was how inexperienced participants tried multiple approaches across their games, and across each game mode, while experienced participants tended to quickly stick to one particular strategy. The latter tended to have longer games as well, probably owing to the fact that they won more often.

4.5 Processing of the Data

For each action happening, a sequence was saved containing the movements that lead up to that action. The length of these sequences was chosen to be 15, striking a balance between capturing all relevant movements that lead up to the action, without processing an excessive amount of frames.

Certain actions happened more often than others did. For example, the *get hit* action had significantly more stored sequences compared to the *kill* or *death* actions. This was expected, as those actions could only occur a limited amount of times per game. However, this did mean infrequent actions had fewer examples for the network to train on.

Sequences that were to be classified as a specific player or game mode were initially saved as sequences of full games. However, using varied sizes of input would require additional work on the neural network. On top of that, using long sequences significantly increases the time required for training. Consequently, the sequences were sliced into smaller sequences of constant size. The original sequences were often built up of multiple actions. The order and time between actions can be indicative for the classification, thus making the sequences too short could result in important information being lost. Initially, the games were sliced into sequences of 50 frames. However, as explained in chapter 3.6.3, they were shortened to 15 frames because of the hardware limitations, which may have caused some of them to lose valuable information. This did increase the amount of labelled data. Considering there was little training data to begin with, this was a major advantage.

Images were saved with a size of 100x100 pixels. This size was chosen to still have a clear representation of the geometry in the scene. The images can always be downscaled if faster training times are required, at the expense of some loss of detail. While it would be logical to save full resolution images and downscaling them

afterwards (so that any size could be used), this would result in longer save times and a larger dataset. Hence this size was chosen to minimize these two issues, while still retaining enough detail to find useful patterns.

4.6 Output of the Neural Network

Three different iterations of the network were trained and verified on the gathered data. Each iteration adjusted to tackle one of the aforementioned classification problems (classifying per player, game mode or action). For each iteration, multiple setups were tested, varying the augmentations, the noise applied and the training parameters of the network.

4.6.1 Initial results

The initial testing was done using the network designed to classify players. The first training sessions used non-augmented data without any noise. The results showed potential during training. During each batch of processed data, a part of that batch was used as validation data. While the validation step showed similar accuracy to the training data, it became clear that the network was not classifying the data according to spatio-temporal features. Instead, after about 80 iterations of training, the network reached an accuracy rate of 100% but performed very poorly on post-training validation, which used unseen sequences. Classifying players, actions and gameplay styles resulted in the same outcome. This was likely due to the slicing of sequences, allowing the network to memorize positions rather than classifying the sequences according to spatio-temporal patterns.

The augmentations explained in chapter 3.6.2 were added to prevent this from happening. After adding them, the network showed more natural behaviour. A slow and steady increase in accuracy took place during training, without the network showing signs of overfitting. Both the validation tests during and after training had similar accuracy. Further testing was done on each iteration of the network separately.

4.6.2 Classifying per player

The initial setup for the network designed to classify players used the unaltered (besides the aforementioned augmentations) input data, gathered as described in chapter 3.2. There were nine different classes, one for each player, giving the network a chance performance of $1/9 \approx 11\%$. Training the network resulted in the accuracy

stagnating around 33%. After this, the loss saw a slight decrease over the following epochs, but no further improvements were measured in accuracy. Although these results were decent, the network was expected to perform better. In an attempt to increase the accuracy, further setups were tested. The input data was made to be as similar as possible to the data used by Bialkowski et al. (2015), who achieved better results (67% accuracy with a chance performance of 5%) using a similar technique in sports. The pictorial representations made for this experiment captured a lot of additional information next to the player's trajectory, such as the movements of enemies, bullets, the locations of pickups etc. Thus, to avoid the network focussing on unimportant information, enemies and bullets were both removed from the pictorial representations. This resulted in both an increase in the learning rate, as well as a higher maximum accuracy reached by the network, peaking at 53% (see Appendix A.3.1).

4.6.3 Classifying per action

The second iteration of the network was built to classify sequences per action. The first setup used was identical to the initial setup of the previous experiment: information about all moving entities was conveyed to the neural network. With all augmentations added (noise, rotations and shifts), the network managed to reach an accuracy of 29% when classifying eight different actions (therefore having a chance performance of 12.5%). Considering Wang & Zemel (2016) used this approach to classify play calls in basketball and achieved an accuracy of 66% with a chance performance of 1/11 ≈ 9, a better result was expected.

Given the increase in accuracy when removing the enemy and bullet trajectories from the pictorial representations in the previous experiment, this was also tested for this iteration of the network. Despite these changes, accuracy still only reached 33%.

In later tests, the *enemy hit* and *get hit* actions were removed to test if the performance of the network could be linked to those actions being too similar with the *kill* and *death* actions. The results improved but were still not optimal. Accuracy peaked at 52% (see Appendix A.3.2). Bearing in mind that the chance performance dropped to 1/6 ≈ 17%, a slight relative increase in accuracy occurred.

4.6.4 Classifying per game mode

The results of the network ordered to classify sequences per game mode were not as successful as expected. The first tests trained the network to classify all three game mode options: The default game mode and the added *low health* and *timer* modes.

The network's accuracy reached up to 40%, which was rather poor as the chance performance was only 1/3 ≈ 33%. Considering the default game mode does not enforce any gameplay behaviour, the decision was made to only classify the added game modes instead, anticipating a relative increase in accuracy compared to the chance performance. Accuracy increased up to 70%, which likely means it was part of the problem. However, taking into account that the chance performance dropped to 1/2 = 50%, these results were still very poor.

Next, similar to the previous networks, a setup showing only the player trajectory was tested. This allowed the network to reach an accuracy of 72% (see Appendix A.3.3) when classifying the *low health* and the *timer* game modes.

Chapter 5: Discussion of Data and Results

5.1 Breakdown of the Input Data

The data gathered from participants shows how they experienced the game. By looking at the number of times certain actions happened, the conclusion can be drawn that most inexperienced participants spent their first session getting familiar with the mechanics and figuring out how to play. Those that played additional sessions afterwards showed clear improvements. This does imply that the initial games of inexperienced participants are not representative of most players in established games, and may not contain valuable movement information.

Another feature of the dataset to consider is the length of sequences. While actions have a static size per sequence, full games were recorded to be labelled as player or game mode. In this experiment, these sequences were cut into smaller pieces of equal size. However, a full sequence may still contain more in-depth information than the sum of the cut pieces. For example, a movement pattern that indicates a certain playstyle may take longer than the size of a cut sequence, and thus it may be misinterpreted and wrongly labelled by the network.

Another slight issue with the data is that an optimal training set for classification problems contains about the same amount of items per class. However, as certain actions tend to occur more often, this is not the case for the acquired dataset. Nonetheless, considering that a decent amount of games was played, enough examples of these uncommon classes were still recorded. The dataset was manipulated so more frequent actions had an equal chance of being chosen during training, to avoid having an imbalanced class representation.

Lastly, as mentioned in chapter 3.2.2, rotational data was saved alongside the positional data of the game entities. The pictorial representations Wang & Zemel (2016) used included a faint representation of previous frames, which allowed the network to deduce the current rotation of players. However, this was not applicable in the Tanks! (2015) game, as players could rotate without moving. As there was no clear way to add rotations to the images, no representation was made that conveyed both rotations and positions of game entities.

5.2 Breakdown of the Classification Process

The results of the trained neural networks showed varying results. While the accuracy of the player classification process is promising, both other networks showed potential but performed poorer than expected. This section tries to explain why this may be the case and covers the implications of these results.

5.2.1 Classifying players

The most successful of the three experiments was the player classification process, achieving an accuracy of 53% with a chance performance of ~11%. From these results, it can be concluded that the network can identify spatio-temporal patterns that are unique to each player. However, it cannot be assumed that these patterns indicate a specific playstyle. They may be related to factors such as experience, stress level, etc. As most of these factors were not measured for this experiment, it cannot be established which kinds of patterns the network used to classify its input.

The increase in accuracy after removing enemy and bullets from the pictorial representations is likely related to the depth of the network. Due to hardware limitations, the network was severely restricted in size. Neural networks can learn a more complex, non-linear function as they become deeper, and can, therefore, find more complex patterns. A deeper network may achieve better relative results using the pictorial representations that show all entities compared to only showing the player trajectory along with the (static) interactable items.

5.2.2 Classifying actions

Wang & Zemel (2016) were able to achieve an accuracy of 66% when classifying different play calls. It was thus expected that a similar accuracy could be achievable when the same technique is applied to games. However, the results of this experiment showed otherwise. Tests using all recorded actions showed insignificant results. Therefore, in the second experiment, all indistinguishable actions were left out. The *hit opponent* and *get hit* actions were dropped due to their similarity to the *kill* action and *death* action respectively. The relative increase in accuracy of the network shows that the network had trouble differentiating these actions. After this, the highest accuracy reached by the network was 52%. This could have been caused by a number of reasons. It is conjectured that not all the classes were distinct actions, but rather events that happened, not necessarily related to the player's movements. For example, the player

hitting themselves was often misclassified, but was probably never intended by the player, and therefore not reflected in their movements.

5.2.3 Classifying game modes

The original idea behind the network classifying sequences per game mode was that players would use different gameplay styles for each game mode. However, the network showed inadequate results. This can be due to a number of reasons. First of all, the inexperience of the participants could be the culprit. Although they were introduced to different game modes, it is not guaranteed they actually used different approaches to these game modes. The participants were only asked to play 30 games, which might not have been enough to get used to the core mechanics of the game for most them, especially considering that not all participants were experienced with the platform or games in general. Thus the game mode may have been neglected in a majority of their games. Resolving this problem is possible by applying the experiment to a game with established tactics, such as Starcraft II (2010) or DotA 2 (2013).

The failing of the network could also be linked to either the lack of or the overwhelming amount of information. Simply showing the network the movements of the player may not be sufficient, as it may not provide the network with all the information it needs to classify a sequence. However, adding additional information (movement information of other entities) may cause the network to overfit. Resolving this problem is trickier. The original hypothesis was that the network would require information such as enemy positions, so it could find patterns in the way the player moved relative to their opponents. However, in the discussed experiment, only showing player movements achieved the highest accuracy. Although it is likely that improved results could be achieved when all gameplay related information is processed, this would require a more advanced network as well as more processing power.

5.3 Possible Improvements

One of the major reasons the network for the poor performance of the network was the lack of processing power available for training. Due to a lack of video random-access memory (VRAM), suboptimal parameters had to be used when training the networks. Because of this, a low batch size had to be used. The length of processed sequences also suffered from this lack of VRAM. Sequences only consisted of 15 frames, while the majority of games took several hundred frames.

A higher amount of VRAM would also allow a deeper neural network to be created, which gives the network the potential to find more complex patterns. This could possibly also result in better accuracy when using the full pictorial representations, including enemy and bullet trajectories. Despite the results of the experiment proving otherwise, exposing all this information to the network should increase its maximum potential accuracy.

As mentioned earlier, the experiment may also be more successful when applied to an existing game with established gameplay styles. First of all, players would already have experience with said game, eliminating the learning process which hindered the creation of accurate data in this experiment. It would also guarantee that different gameplay styles are being used. Depending on the game, labelling manually may still prove a problem, but plenty of games exist where the chosen gameplay style of a player is clearly visible (i.e. Starcraft II (2010) where each faction has certain strengths and weaknesses which the player has to use to their advantage and forces them to use certain strategies).

Furthermore, the network could be made to work in an unsupervised manner. In an optimal scenario, the network does not need labelled data to train itself. Unsupervised learning may result in the network clustering the data according to the playstyle used by the player. However, it cannot be predicted what patterns the network will focus on. If the network can somehow be steered towards clustering the data per playstyle, this would eliminate the labelling process, allowing the automation of the whole process.

Another possible improvement would be to somehow add player actions to the input of the network. Giving the network access to full information of which actions the player is performing should allow the network to identify patterns more easily.

5.4 Possible Applications

The different kinds of patterns that can be detected in player movement can each be used in various fields of game development. First of all, as explained in chapter 5.2, patterns can be found that are unique to each player. These patterns may indicate how the player is experiencing the game. If the exact origins of these patterns could be determined, this information can be used by the game developer to adjust gameplay at runtime. For example, they could be an indication of stress or boredom. Access to the current stress level of a player can allow the developer to enhance stressful moments and avoid boring experiences. Alternatively, they may also indicate whether or not a

player is struggling with controls or the difficulty of the game. Such information could allow for a dynamic difficulty system or prompt a tutorial popup.

Furthermore, it could be used to prevent people from account sharing. These practices are against the rules of many games. For example, competitive games frequently struggle with skilled players *boosting* other players (increasing their in-game rank by playing on their account, often in exchange for money). Such abuses could be detected using the methodology proposed in this thesis.

Action detection, on the other hand, could be useful when designing the AI of a game. The capability of an RCNN to predict which action a player is about to execute, may in some cases allow the AI designer to create a more compelling experience.

The original goal of this research was to find patterns that indicate the usage of a specific playstyle. As explained in chapter 3.2.1, the *low health* and *timer* game modes were implemented to enforce the player to use different approaches towards the level and thus use different playstyles. Although the accuracy of the associated network wasn't particularly high, it still showed potential despite the earlier mentioned issues with the experiment. If a clear definition can be given to the different playstyles available in a game, and the labelling process is improved, it is very likely that the network can accurately recognize them. This would allow AI designers to use this information when implementing the behaviour of the game's AI agents. Creating the illusion of an AI that actively adapts to the player, while still giving the designer full control over its behaviour. To prevent this from further complicating the scripts, an approach using utility AI is suggested and explained in chapter 3.5.

Another possible application would be the detection of *bots* in games. These are AI systems that play games instead of players. They are often abused to farm in-game currencies. The experiments proposed in this thesis do not cover the possibility to classify real players from bots. However, the discussed neural network may be capable of doing so.

5.5 Answering the Research Question

The goal of this paper was to find out what kind of patterns could be found in the movements of a player in video games using deep learning techniques, and if these patterns could be used to indicate if a player is using a particular playstyle. Results showed that a neural network is capable of finding several different spatio-temporal patterns in player trajectories. First of all, patterns that were unique to each player were found. While the exact origin of these patterns is yet to be found, they are either

connected to the player's gameplay behaviour or the way they are experiencing the game.

Furthermore, the network showed some capability to identify which game mode a sequence came from. The only difference between these sequences is how the players approached the level. Therefore, while the network's accuracy was not particularly high, it can still be concluded that the network was able to identify patterns that indicate a certain playstyle.

Lastly, patterns unique to certain actions were also found. While some actions were often misinterpreted by the network, others were properly classified the majority of the time.

Chapter 6: Conclusion and Future Directions

6.1 Conclusion

The goal of this research was to deduce whether or not neural networks are capable of finding spatio-temporal patterns in the movements of players in games. In particular, patterns that indicate the usage of a certain playstyle. A recurrent convolutional neural network was created and tested in three different classification tasks. Evaluation of its accuracy shows that the network is capable of recognizing different patterns in player movements. From the results, it is deduced that players exhibit unique movements that differentiate them from other players. The network also demonstrates the ability to recognize game-specific actions purely based on player trajectories. However, some actions were often misclassified due to their similarity to other actions. Lastly, the model showed potential when classifying movement sequences per game mode, which infers that it can distinguish different playstyles. Although the accuracy of the trained network was inadequate to apply the technique in practice, it is conjectured that better results may be achieved if more training data is available and the labelling process is improved.

Even though the proposed neural network does not directly implement an adaptive AI, exposing information such as the player's current playstyle to the game developers can allow them to create a more dynamic experience.

6.2 Future Directions

While this research demonstrates the potential of deep learning techniques in spatio-temporal pattern recognition, it does not delve deep into the exact origin of these patterns. Many factors may have an influence on how a player moves throughout a level, such as stress, focus, experience with the platform, etc. These factors were not measured and thus not taken into account, but they still may be related to the results. Future research could provide more insight into how these factors relate to player movements.

The network may also benefit from using a different representation of the data. The pictorial representations used can only display spatio-temporal information. However, the network may perform better if information such as player actions (shooting, reloading, …) is also conveyed. This may be achieved by simply processing play inputs instead of their movements, along with information about their surroundings. Not only would this provide the network with more useful information,

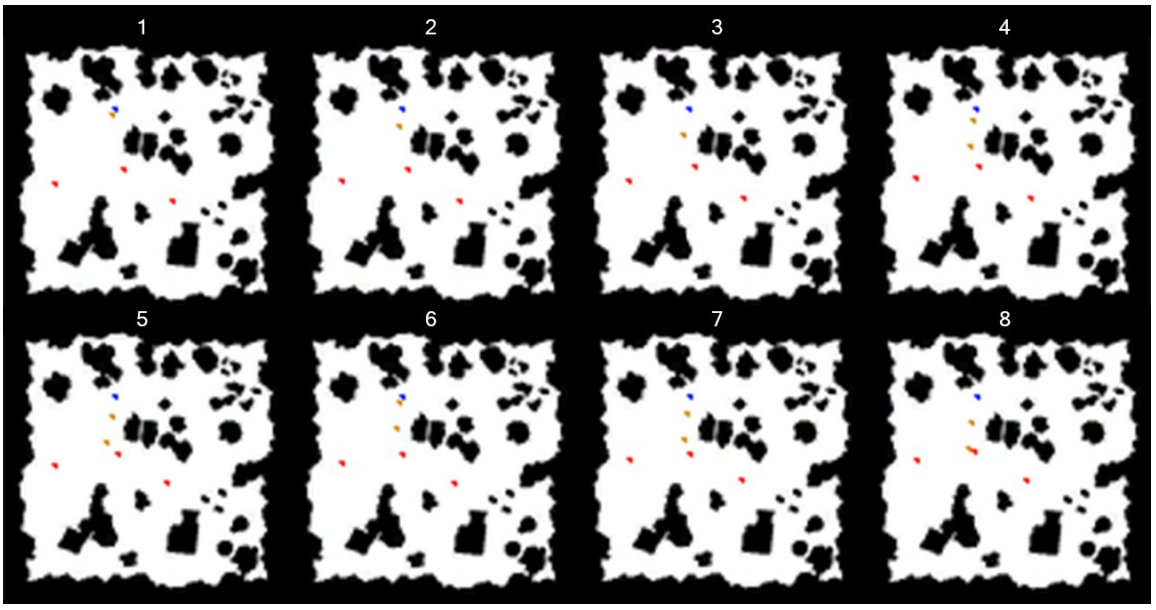
but it may also reduce the computational power necessary to train and use the network. As other methods are yet to be tested, it is still unknown which representation would yield optimal results.

Furthermore, the network was not tested in a real-time application. It is plausible that the network, once trained, will not have an excessive impact on performance. However, only an actual implementation can provide accurate statistics on the computational power needed to evaluate player movements at runtime.

Lastly, the network may also have potential in other fields of game development. It was briefly covered how it may be implemented to discover players breaking in-game rules such as account sharing or botting. These possibilities have not been tested in this research but could benefit the game sector as well.

Appendix

A.1 Pictorial representations



A snippet of 8 frames from a sequence showing the player (blue), enemies (red), and bullets (orange).

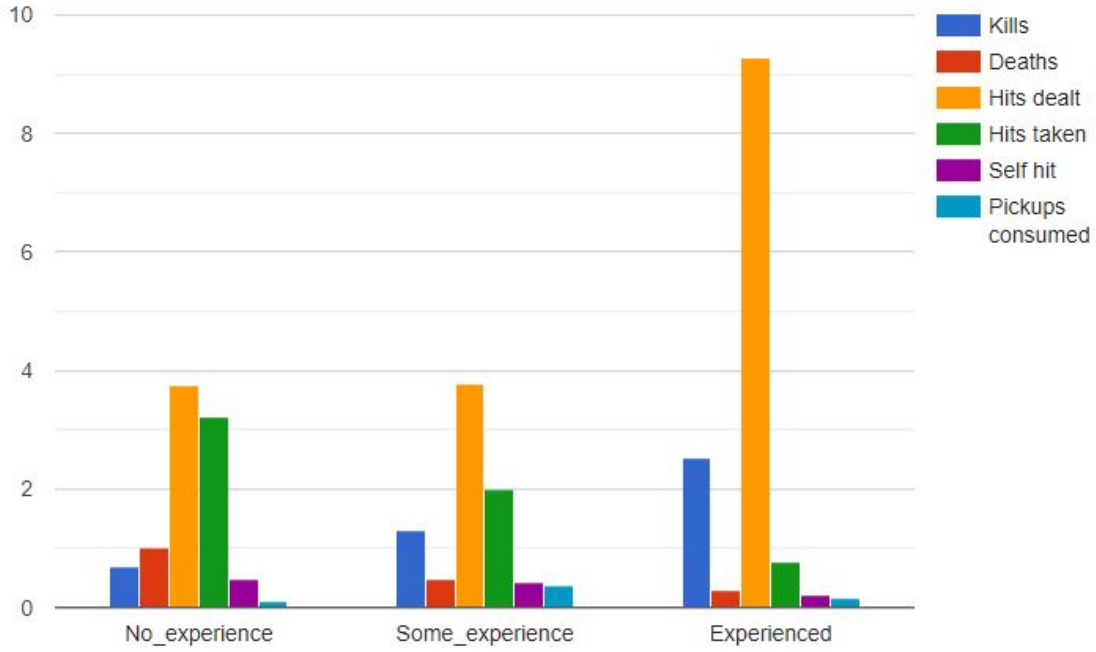
A.2 Participant Data

A.2.1 Actions performed per participant

Participant	Experience with games (1-7)	Games played (Sessions)	Kills	Deaths	Hits	Hits taken	Self hit	Pickups consumed
A	Some experience	66 (2)	112 (47, 65)	21 (13, 8)	237 (112, 125)	87 (48, 39)	30 (14, 16)	23 (12, 11)
B	Experienced	33 (1)	78	6	237	28	4	4
C	Experienced	33 (1)	69	8	234	32	2	6
D	No experience	33 (1)	18	30	/	/	/	/
E	No experience	33 (1)	23	30	/	/	/	/
F	Some experience	66 (2)	48 (16, 32)	36 (20, 16)	199 (70, 129)	172 (82, 90)	21 (18, 3)	26 (5, 21)
G	Some experience	33 (1)	37	15	130	42	14	5
H	Experienced	33 (1)	79	6	236	15	15	5
I	Experienced	42 (1)	102	18	498	24	7	7

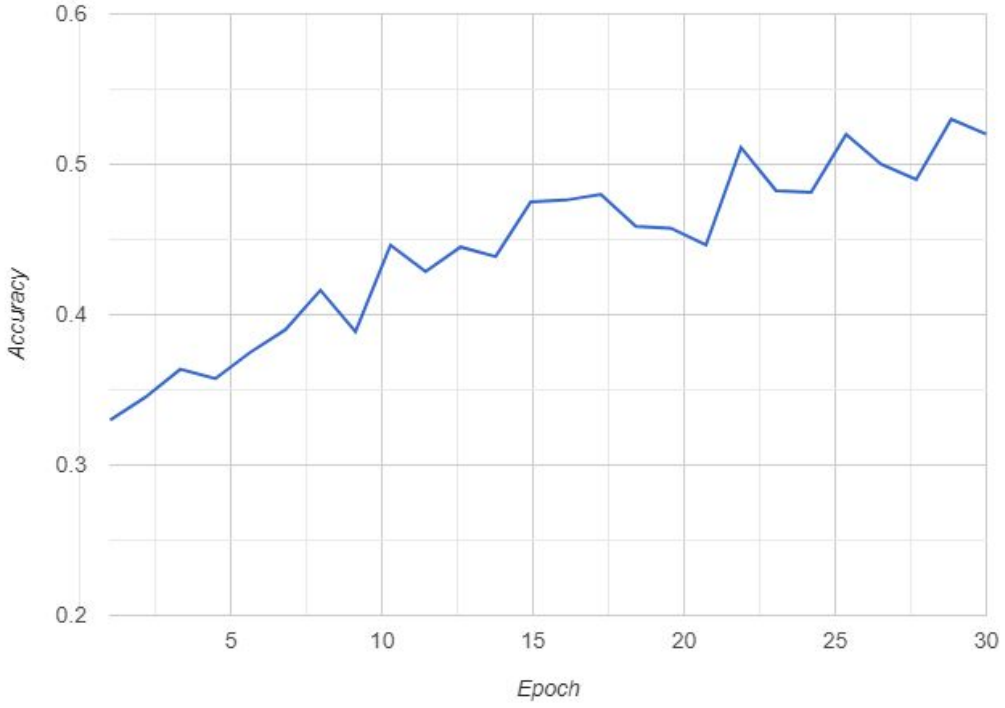
Participants and the amount of games they played, along with the number of actions performed. Note that a few numbers were overwritten due to a bug in the saving process and were left out.

A.2.2 Number of actions performed per game

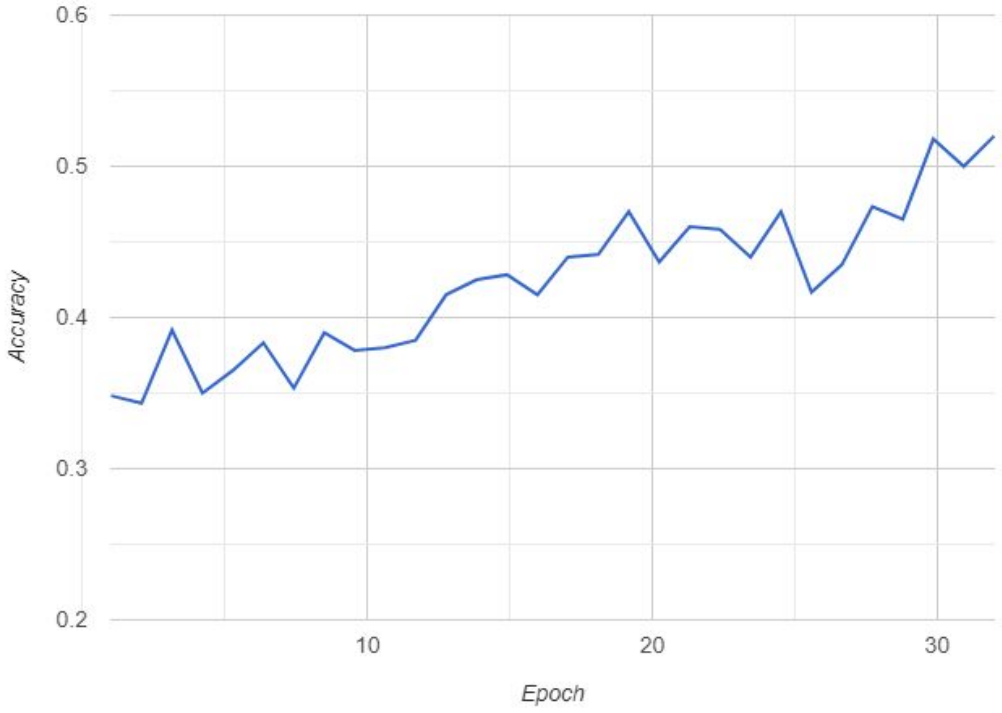


A.3 Accuracy per Epoch per Classification Task

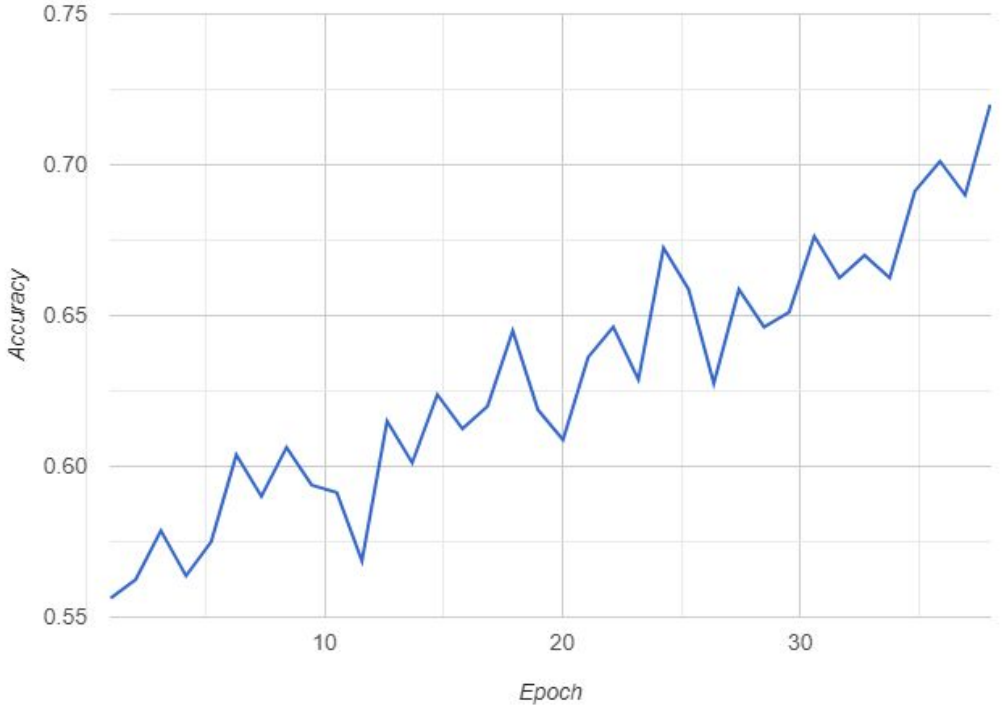
A.3.1 Accuracy increase over epochs for player classification



A.3.2 Accuracy increase over epochs for action classification



A.3.3 Accuracy increase over epochs for game mode classification



Bibliography

Bandai Namco Studios (1994). “Tekken” (Video game). Bandai Namco Entertainment & Sony Computer Entertainment.

Bengio, Y., Simard, P., and Frasconi, P. (1994). “Learning long-term dependencies with gradient descent is difficult”. In: IEEE Transactions on Neural Networks, pp. 157–166. URL: http://www.iro.umontreal.ca/~lisa/pointeurs/ieeetrnn94.pdf

Bialkowski, A., Lucey, P., Carr, P., Yue, Y., Sridharan, S. & Matthews, I. (2015). “Identifying Team Style in Soccer Using Formations Learned from Spatiotemporal Tracking Data”. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.670.2128&rep=rep1&type=pdf

Blizzard Entertainment (1995). “Warcraft II: Tides of Darkness”. Davidson & Associates.

Blizzard Entertainment (2010). “Starcraft II: Wings of Liberty”. Blizzard Entertainment.

Brockington & Darrah (2002). “How Not to Implement a Basic Scripting Language”. In: Rabin, S., ed., AI Game Programming Wisdom, Charles River Media, pp. 548-554. URL: http://planiart.usherbrooke.ca/files/Rabin%202002%20-%20AI%20Game%20Programming%20Wisdom.pdf

Capcom (1987). “Street Fighter”. Capcom.

Carr, J. (2014). “An introduction to Genetic Algorithms”. URL: https://www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf

Cord, M., Delany, S. & Cunningham, P. (2008). “Supervised Learning”. In: Machine Learning Techniques for Multimedia, pp. 21-49. Berlin, Germany: Springer. URL: https://www.springer.com/gp/book/9783540751700

Crockford, D. (2002). “Introducing JSON”. URL: http://www.json.org/

Dynamic Pixels (2017). “Hello Neighbor” (Video game). tinyBuild.

Elkin, A. (2013). “Video Game Satisfaction with Adaptive Game AI”. URL: https://pdfs.semanticscholar.org/bf82/a80e9214cb850428b48505620494909de6f9.pdf

Evans, R. (2015). “Modelling individual personalities in The Sims 3”. In: GDC Vault, pp. 36-38. URL: https://www.gdcvault.com/play/1012450/Modeling-Individual-Personalities-in-The

FIBA (2014). “Official Basketball Rules 2014”. FIBA. URL: http://www.fiba.basketball/documents/2015/Official_Basketball_Rules_2014_Y.pdf

Gers, F., Schraudolph, N. & Schmidhuber, J. (2002). “Learning Precise Timing with LSTM Recurrent Networks”. In: Journal of Machine Learning Research, pp. 115– 143. URL: http://www.jmlr.org/papers/volume3/gers02a/gers02a.pdf

Ghewari, R. (2017). “Action Recognition from Videos using Deep Neural Networks”. URL: https://escholarship.org/uc/item/2mr798mn

Hochreiter, S., Schmidhuber, J. (1997). “Long Short-Term Memory”. In: Neural Computation, pp. 1735-1780. URL: https://www.bioinf.jku.at/publications/older/2604.pdf

Hoge, C. (2018). “Helping Players Hate (or Love) Their Nemesis” (Video). URL: https://www.gdcvault.com/play/1025150/Helping-Players-Hate-(or-Love)

Hussein, A., Gaber, M., Elyan, E. & Jayne, C. (2017). “Imitation Learning, a Survey of Learning Methods”. In: ACM computing surveys. URL: https://openair.rgu.ac.uk/bitstream/handle/10059/2298/HUSSEIN%202017%20Imitation%20learning.pdf?sequence=3&isAllowed=y

Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M. & Lange, D. (2018). “Unity: A General Platform for Intelligent Agents”. URL: https://arxiv.org/pdf/1809.02627.pdf

Majchrzak, K., Quadflieg, J. & Rudolph, G. (2015). “Advanced Dynamic Scripting for Fighting Game AI”. In: Lecture Notes in Computer Science, pp. 86-99. URL: https://hal.inria.fr/hal-01758421/document

Malone, T. (1980). “What Makes Things Fun to Learn? A Study of Intrinsically Motivating Computer Games”. URL: https://hcs64.com/files/tm%20study%20144.pdf

Mehrasa, N., Zhong, Y., Tung, F., Bornn, L., Mori, G. (2017). “Deep Learning of Player Trajectory Representations for Team Activity Analysis”. URL: http://www.lukebornn.com/papers/mehrasa_ssac_2018.pdf

Mnih, V., Kavukcuoglu, K., Silver, D., Gravis, A., Antonoglou, I., Wierstra, D., Riedmiller, I. (2013). "Playing Atari with Deep Reinforcement Learning". Technical Report arXiv:1312.5602 [cs.LG], Deepmind Technologies. URL: https://arxiv.org/pdf/1312.5602v1.pdf

Monolith Productions & Feral Interactive (2014). "Middle-earth: Shadow of Mordor" (Video game). Warner Bros. Interactive Entertainment.

Krizhevsky, A., Sutskever, I., Hinton, G (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: Advances in Neural Information Processing Systems, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Ponsen, M., Spronck, P., Muñoz-Avila, H., Aha, D. (2007). "Knowledge Acquisition for Adaptive Game AI". In: Science of Computer Programming, pp. 59-75. URL: https://www.sciencedirect.com/science/article/pii/S0167642307000548

Ricciardi, A. & Thill, P. (2008). "Adaptive AI for Fighting Games". URL: http://cs229.stanford.edu/proj2008/RicciardiThill-AdaptiveAIForFightingGames.pdf

Risi, S. & Togelius, J. (2015). "Neuroevolution in Games: State of the Art and Open Challenges". URL: https://arxiv.org/pdf/1410.7326.pdf

Rodriguez, J. (2018). "The science behind OpenAI Five that just Produced One of the Greatest Breakthrough in the History of AI". URL: https://towardsdatascience.com/the-science-behind-openai-five-that-just-produced-one-of-the-greatest-breakthrough-in-the-history-b045bcdc2b69

Ruder, S. (2017). "An overview of gradient descent optimization algorithms". URL: https://arxiv.org/abs/1609.04747

Shibuya, N. (2018). "Demystifying Cross-Entropy". URL: https://towardsdatascience.com/demystifying-cross-entropy-e80e3ad54a8

Spronck, P., Sprinkhuizen-Kuyper, I. & Postma, E. (2003). "Online Adaptation of Game Opponent AI in Simulation and in Practice". URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.323.7253

Spronck, P., Sprinkhuizen-Kuyper, I. & Postma, E. (2002). “Improving Opponent Intelligence Through Offline Evolutionary Learning”. URL: https://pdfs.semanticscholar.org/631b/2592a01d2b59e744c69ee416900c01217cfc.pdf

Stanley, K. (2005). “Neuro Evolving Robotic Operatives” (Video game). URL: http://nn.cs.utexas.edu/nero/index.php

State Software (2001). “JSON” (File format). URL: https://www.json.org/

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., … Hassabis, D. (2017). “Mastering the game of Go without human knowledge”. Nature, 550:354– 359. URL: https://www.nature.com/articles/nature24270

Thurau, C., Sagerer, G. & Bauckhage, C. (2004). “Imitation Learning at all Levels of Game AI”. In: Proceedings of the international conference on computer games, artificial intelligence, design and education. URL: https://www.researchgate.net/publication/228474437_Imitation_learning_at_all_levels_of_game-AI

Tozour, P. (2002). “The Perils of AI Scripting”. In: Rabin, S., ed., AI Game Programming Wisdom, Charles River Media, pp. 541-547. URL: http://planiart.usherbrooke.ca/files/Rabin%202002%20-%20AI%20Game%20Programming%20Wisdom.pdf

Turn 10 Studios, Playground Games & Sumo Digital (2014). “Forza Horizon 2” (Video game). Microsoft Studios.

Unity Technologies (2015). “Tanks!” (Video game). URL: https://github.com/lukearmstrong/unity-tutorial-tanks

Unity Technologies (2018). “Unity 2018.3.0f2” (Computer software). URL: https://unity.com/

Valve Corporation (2013). “Defense of the Ancients 2” (video game). Valve Corporation.

Valve Corporation, Turtle Rock Studios, Certain Affinity (2008). “Left 4 Dead” (Video game). Valve Corporation & Electronic Arts.

Wang, K., Zemel, R. (2016). “Classifying NBA Offensive Plays Using Neural Networks”. URL: http://www.sloansportsconference.com/wp-content/uploads/2016/02/1536-Classifying-NBA-Offensive-Plays-Using-Neural-Networks.pdf

Woodcock, S. (2000). “Game AI: The State of the Industry”. In: Game Developer Magazine. URL: https://www.gamasutra.com/view/feature/3570/game_ai_the_state_of_the_industry.php?print=1

This template is based on a template by:
Steve Gunn (http://users.ecs.soton.ac.uk/srg/softwaretools/document/templates/)
Sunil Patel (http://www.sunilpatel.co.uk/thesis-template/)