Master Thesis

Enhancing Monte Carlo Tree Search by Using Deep Learning Techniques in Video Games

M. Dienstknecht

Master Thesis DKE 18-13

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science of Data Science for Decision Making
at the Department of Data Science and Knowledge Engineering
of the Maastricht University

Thesis Committee:

Dr. M.H.M. Winands
Dr. K. Driessens

Maastricht University
Faculty of Science and Engineering
Department of Data Science and Knowledge Engineering

August 24, 2018

Preface

Playing video games has been a passion of mine for almost two decades now. Looking at the process the field of video game AI has made during this time is stunning. Developing an AI that could easily beat humans in a video game might sound like nonsense to some people but actually, it is not. This thesis shows the process of enhancing an agent that uses a Monte-Carlo Tree Search by combining it with a Convolutional Neural Network. The final agent participated in the "Ms. Pac-Man vs Ghost Team" tournament. The results of this tournament were presented at the 2018 IEEE Conference on Computational Intelligence and Games. First of all, I want to thank Dr. Mark Winands for supervising my thesis and supporting me throughout my research. Additionally, I want to thank Dr. Kurt Driessens for also supervising my thesis. Moreover, I am grateful that I could use the hardware that was provided by Maastricht University and the RWTH Compute Cluster of the RWTH Aachen University to run all the different experiments. Although several maintenances did slow the process down, it would have never been possible to run all of the time-consuming evaluations in that short amount of time. Finally, I want to thank my family for giving me the best support they could.

Markus Dienstknecht
Maastricht, August 2018

Abstract

The main goal of this master thesis is to extend Monte Carlo Tree Search (MCTS) in real-time video games by using Deep Learning. MCTS is a heuristic search algorithm which is most often used for decision making in games. It tries to determine a good move given the current game state.
Although, an MCTS has important advantages compared to $\alpha\beta$ search and similar algorithms it does not converge very fast. Additionally, the search may not investigate a path, which leads to a loss. Consequently, that path cannot be taken into account. By using Deep Learning, the MCTS should be improved to for example find optimal paths easier by better determining the value of a given game state. For that, a Convolutional Neural Network (CNN) is trained by using the Java library DL4J.
CNNs usually take an image as their input, which is then passed through the network to classify certain aspects of that picture. In this case, the network takes a game state as its input and estimates a move policy for that game state. By that, an estimation for the player's best next move is given.
This concept is similar to what Deepmind's AlphaGo does. The agent uses CNNs to determine either a value for a certain game state or a move policy that could then be used by an MCTS. This improves the search algorithm to work with better estimates and converge faster.
In this thesis, a concept that resembles the AlphaGo approach is applied to the real-time video game Ms. Pac-Man. Furthermore, it is successfully shown that a CNN can in principle improve an MCTS in this real-time video game. However, Deep Learning techniques are time-consuming and the time a CNN needs to process a game state is too much to compensate for the decreased number of play-outs in the MCTS.
Moreover, this thesis provides information about how to combine a CNN with an MCTS, how to configure a good performing CNN for real-time video games and possibilities of training such a network with game state samples.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives an introduction to the topic of this thesis. First, Artificial Intelligence for games, in general, is described in Section 1.1. Afterward, Section 1.2 gives an introduction to a heuristic search algorithm named Monte Carlo Tree Search. Furthermore, the reference to machine learning is made in Section 1.3. Next, the problem statement and the three research questions which this thesis is trying to solve are presented in Section 1.4. Finally, this chapter provides an outline of the remainder of the thesis in Section 1.5.

1.1 Game Playing AI

Artificial Intelligence (AI) that is able to play video games has become more and more popular over the past years and is one of the main research fields in AI. It is an easy way to show what AI is capable of and to compare its strength to that of human beings. There exist several techniques that can be applied to create a good game playing AI. The most famous ones are $\alpha\beta$ search (Knuth and Moore, 1975) and Monte Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006) (Coulom, 2007). The $\alpha\beta$ search usually tries to investigate all possible moves in a search space and by that finding the best one given the current game state. After taking a default $\alpha\beta$ framework and adding different search control methods to it, the human world champion in chess could be beaten by the $\alpha\beta$ framework Deep Blue in 1997 (Campbell, Hoane Jr, and Hsu, 2002).

1.2 Monte Carlo Tree Search

Once the search space gets bigger, $\alpha\beta$ search quickly reaches its limits because it cannot achieve a proper lookahead in a reasonable time. This problem can be tackled by using MCTS. Usually, an MCTS focuses more on promising paths in the tree and can, therefore, spend its computational resources more efficiently. Furthermore, the dependency on explicit human knowledge can be handled better. The reason for that lies in the evaluation method of MCTS. It relies on

evaluating different game states by running simulations and investigating the search space based on the resulting scores. This allows the search method to work without human knowledge. An example of this field is the Asian game Go. Although the game is so complex that $\alpha\beta$ searches performed weakly, DeepMind was able to build an MCTS based AI that could beat a professional Go player in 2016 (Silver *et al.*, 2016). This AI was called AlphaGo and was improved even more till then. Only one year later, the same researcher's group published a newer version called AlphaGo Zero. This AI could even beat the human world champion in Go (Silver *et al.*, 2017b) and therefore reached superhuman status. DeepMind accomplished that by extending the MCTS with Machine Learning. They used a Neural Network to either estimate the next best moves or to estimate the value of a specific game state. Given that information, the MCTS could run faster and more precise than before. What is even more stunning is that AlphaGo Zero did not use any human data at all. It learned the game of Go completely by itself by playing and evaluating its own moves.

1.3 Combining Search and Machine Learning

The performance of a search technique is influenced by many parameters. Tuning all of these parameters by hand can take much time and effort but may also be unfeasible in some cases. The exact best configuration can vary from application to application and therefore even existing research cannot always provide these parameters. However, there exist different machine learning techniques that can help to find the correct values (Chaslot *et al.*, 2008). An important aspect is that these methods must not rely on the availability of an analytic fitness function because it is most likely not given in a game. One group of methods that does not face this problem is called temporal-difference methods. These methods have been proven in the past to work successfully for Backgammon, Chess, Checkers, and LOA (Tesauro, 1995; Baxter, Tridgell, and Weaver, 1998; Schaeffer, Hlynka, and Jussila, 2001, Winands *et al.*, 2002). However, every other method that does not need a fitness function could also be used. For example, so-called "Finite-Difference Stochastic Approximations" have been used in chess for the agent Crafty (Björnsson and Marsland, 2001) and "Resilient Simultaneous Perturbation Stochastic Approximation" has been used in Poker and LOA (Kocsis, Szepesvári, and Winands, 2005).

Another way of using machine learning in the context of video games is the use of Neural Networks. Starting in the year 1995 (Tesauro, 1995), Neural Networks are still used today for playing video games. They are mainly applied to evaluate actions or game states and by that determine good moves. One of the greatest achievements of the past years is the Go playing AI AlphaGo Zero. DeepMind used a Convolutional Neural Network (CNN) as a machine learning algorithm for their AI and trained it without any human knowledge (Silver *et al.*, 2016; Silver *et al.*, 2017b). Combining this CNN with an MCTS resulted in an agent that outperformed every other Go playing AI

that is known yet. The agent was continuously improved during training by making use of Reinforcement Learning. The agent received an instant feedback about the outcome of a game once it was finished and uses this knowledge for the next games. Agents which are making use of a CNN framework have shown to perform extremely well and even surpass human champions. The most recent achievement of DeepMind is called AlphaZero (Silver *et al.*, 2017a). This agent also makes use of an MCTS, a CNN, and Reinforcement Learning but is more generic and can play Shogi, Chess, and Go on a superhuman level.

1.4 Problem Statement and Research Questions

As described before, DeepMind has been able to build an AI with superhuman status. However, in board games like Go each turn can take several seconds to give the player time to think about the next move. To push the boundaries even further, the next step is to apply this concept in real-time games. Most simple Arcade Games provide a good basis for a real-time AI. Therefore, there exist many competitions in this field to test the capability of such an Artificial Intelligence. Since 2016, the "Ms. Pac-Man vs Ghost Team Competition" (Williams, Perez-Liebana, and Lucas, 2016) gives participants the opportunity to compete in the real-time Arcade Game Ms. Pac-Man. Taking the concept of DeepMind and adapting it to the field of real-time games leads the problem statement:

> *Enhancing Monte Carlo Tree Search by using Deep Learning techniques in the context of real-time video games.*

Most Deep Learning techniques such as Convolutional Neural Networks (CNNs) (Krizhevsky, Sutskever, and Hinton, 2012) can take some time to compute depending on the problem and the hardware. Therefore, a trade-off between performance and efficiency needs to be found to make the combination of MCTS and Deep Learning applicable in real-time. The approach of using a CNN on top of an MCTS can add even more restrictions regarding computation and time management. Therefore, the first research question addresses exactly that problem and tries to answer if this enhancement can be applied without making the performance suffer too much:

1. *Can the* AlphaGo *approach be adapted to work for real-time video games?*

Furthermore, the CNN can enhance the MCTS in different ways. As previously described, DeepMind used it to evaluate the current game state and to get the best moves given a specific state. Besides these two approaches, there are even more ways to enhance the search. This leads to the second research question:

2. *Which different approaches can be used to combine Convolutional Neural Networks and Monte Carlo Tree Search?*

Even though today's Deep Learning frameworks make Machine Learning more and more easy, there are still many things that can be adjusted. From hyperparameters to the exact structure of the network layers, everything has a large

influence on the performance of the CNN. Especially in a real-time video game environment, everything depends on providing the best results given strict time limitations. Therefore, the last research question is:

3. *What is the best architecture of a Convolutional Neural Network given the real-time Arcade Game Ms. Pac-Man environment?*

1.5 Thesis Outline

This thesis first introduces the reader to the background knowledge which is required to understand the main part. Chapter 2 describes the Arcade Game Ms. Pac-Man and its rules, which are used in the thesis. Furthermore, it introduces the competition in which the developed AI competes in. Chapter 3 shows the concept of an MCTS, which is the algorithm this thesis aims to enhance. Chapter 4 describes Deep Learning in general and focuses more on Convolutional Neural Networks of which one is used for enhancing the MCTS. Chapter 5 presents related work in this research field and shows concepts of combining an MCTS and Deep Learning. Chapter 6 describes the final "Pac-Man Agent" of this thesis. Chapter 7 shows the experiments that were run to evaluate the best configuration and Chapter 8 concludes the thesis with a discussion about the problem statement and the research questions and finishes with eventual future work that can be done.

Chapter 2

Pac-Man

Pac-Man is a single-player arcade game which was invented in 1980 by Toru Iwatani, an employee at the Japanese company Namco. The game title was originally called "Puck Man" which means "repeatedly open and close the mouth". One Year later, in 1981, the American company Midway licensed the game and it was named "Pac-Man" from there on. The game itself became so famous that it was soon played all around the world. The game was originally played using a joystick that could be tilted in four directions.

In this chapter first, the game Pac-Man itself is explained in Section 2.1, Section 2.2, and Section 2.3. Afterward, Section 2.4 describes its most relevant sequel, which is also the game that is used for research in this thesis. Finally, the topic will be connected to the field of AI and therefore, competitions are presented which were run to let different AIs compete against each other in Section 2.5.

2.1 Rules

The game takes place in a maze-like structure consisting of different corridors as shown in Figure 2.1. All of these corridors are filled with pills. Additionally, there are four power pills (also called energizers), each located in a different corner of the maze. On the left and the right side of the maze, two tunnels are connected horizontally allowing a character to move through them to reach the opposite side of the maze faster. The player controls the main character, Pac-Man, with the goal to eat these pills while he is getting chased by four different ghosts. In the beginning, the ghosts start in the middle of the maze in a closed area, the monster pen, and Pac-Man starts in the center of the bottom half.

Starting with three lives, every time Pac-Man gets caught by a ghost he loses a life. However, eating one of the four power pills make the ghosts edible for a short period. Eating a ghost during that time rewards the player with additional points and the ghost is returned to the monster pen for a short time.

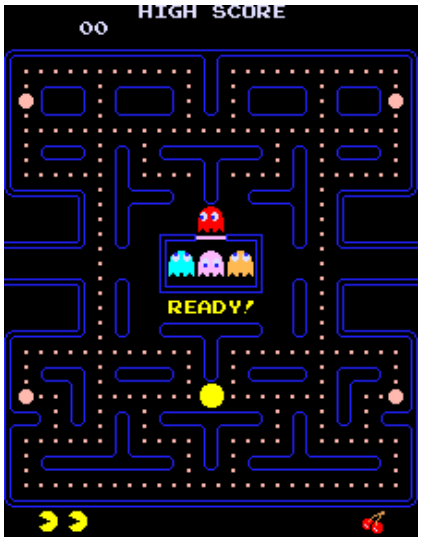


Figure 2.1: Initial state of the arcade game Pac-Man

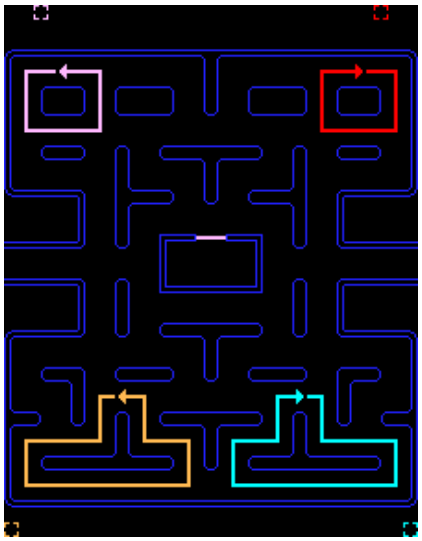


Figure 2.2: Positions of the ghosts in scatter mode

After reaching level 21, the power pills lose their effect meaning they still reward the player with extra points but the ghosts no longer become edible. Furthermore, different fruits which also increase the score can spawn during the game. A maze is completed once all pills are eaten. In addition to getting a lot of points for completing a level, Pac-Man also gets rewarded with an extra lifeline. Obviously, the game is lost once Pac-Man lost all lives.

2.2 Ghost Behavior

Each ghost has a different color and also a different behavior (Birch, 2010). However, every ghost acts deterministically. First of all, there exist three different base strategies which the ghosts follow (Pittman, 2009, accessed February 20, 2018):

- **Chase**: In this mode, each of the ghosts is willing to catch Pac-Man. Moreover, during that strategy, the different behaviors of the ghosts come into play.
- **Scatter**: As shown in Figure 2.2, the ghosts can occasionally give up their chasing behavior for a few seconds and scatter around to the maze corners.
- **Frightened**: Once Pac-Man eats a power pill, the ghosts become edible and wander around the maze randomly until they either get eaten or the effect of the power pill runs out.

As mentioned before, during chase mode each of the ghosts approaches to catch Pac-Man differently. Knowing all of the different strategies gives the player the opportunity to play around the movements of the ghosts. The following shows the four different ghosts and their default behavior:

- **Blinky**: The red colored ghost was originally called Oikake and his personality is described by the word "shadow". Blinky usually chases Pac-Man using a shortest path method unless he is frightened or in scatter mode. However, if a certain amount of pills, depending on the level, have been eaten by Pac-Man this ghost will stick to the chasing behavior even if the other ghosts start to scatter around the maze.

- **Pinky**: As the name already suggests, this ghost is colored pink. The main behavior is to target the tile which is four tiles ahead of Pac-Man. By that, this ghost tries to cut off the path to which the main character is heading.

- **Inky**: Inky is the least predictable and cyan colored ghost. The ghost's target location is calculated using Pac-Man's target and Blinky's position. In exact, the target consists of taking the distance between Blinky's position and the tile which is two steps away from Pac-Man in the direction he is facing. Doubling that distance results in Inky's target. This strategy allows the ghost to keep roughly the same distance to Pac-Man as Blinky does.

- **Clyde**: The orange ghost is the easiest to avoid. While he is more than 8 tiles away from Pac-Man measured in Euclidean distance, the ghost acts similarly as Blinky. In that case, he tries to get as close as possible towards the position of Pac-Man. However, once reaching the 8 tile radius around Pac-Man, Clyde instantly switches to the scattering behavior and therefore tries to reach the bottom left corner. This usually makes the ghost alternating between the two modes very often and due to that staying 8 tiles away from Pac-Man. Consequently, the player needs to be cautious while being in Clyde's corner.

2.2.1 Speed

During the course of the game the speed of the ghosts and Pac-Man increases. After eating a power pill, the game enters the so-called "frightened mode". In this mode, Pac-Man's speed is increased and the ghosts' speed is decreased. This makes the ghosts easier to catch. However, eating pills makes Pac-Man always slower regardless of the current mode. By default, the ghosts are 5% slower than Pac-Man until reaching the 21st level. From there on, the ghosts become 5% faster than Pac-Man and the game becomes harder. Moreover, while passing through a tunnel the ghosts are always slower than in any other situation regardless of their current behavior. Originally shown by Pittman (2009, accessed February 20, 2018), Table 2.1 shows the exact speeds.

	Pac-Man				Ghosts		
Level	**D**	**DE**	**F**	**FE**	**D**	**F**	**T**
1	80%	71%	90%	79%	75%	50%	40%
2 - 4	90%	79%	95%	83%	85%	55%	45%
5 - 20	100%	87%	100%	87%	95%	60%	50%
21+	90%	79%	-	-	95%	-	50%

Table 2.1: Speed of the characters in Pac-Man regarding their current strategy and the current level. Column **D** represents the default speed, column **F** represents the speed in frightened mode, and column **T** means a ghost being in a tunnel. Additionally, **E** indicates that Pac-Man is eating a pill.

2.3 Goal

Because of the reason that the internal level counter was an 8-bit value, the maximum playable level is 255. However, there is a level 256 which cannot be completed due to bugs which occur because of the overflowing level counter. Achieving a perfect score means eating all pills and all fruits in each one of the 255 mazes and additionally eating every ghost each time Pac-Man consumed a power pill. The world record for a perfect play without losing a life is held by David Race and took him 3 hours, 28 minutes, and 49 seconds.

2.4 Ms. Pac-Man

Due to its fame, Pac-Man has been remade on different platforms and with slightly changed rules several times. The most famous Pac-Man clone is called Ms. Pac-Man and was released in 1981 in the United States by the company General Computer Corporation. Furthermore, it was originally called Crazy Otto and was sold without Namco's permission. Later, the creators of Ms. Pac-Man struck a deal with Namco to officially license the game as a sequel. Today, many people consider Ms. Pac-Man to not only be better than the original version but the best of all Pac-Man clones.

2.4.1 Changes

The game itself now has a female main character and includes several innovations compared to the original Pac-Man game "including faster gameplay, more mazes, new intermissions, and moving bonus items" (Weiss, 2012). The most interesting change is the update of the ghost AI. As mentioned before, in the default Pac-Man game the ghosts behaved purely deterministic so that knowledge about their behavior would lead to an easy game. This means that "the determinism of the ghosts implies that there is a pure strategy that is optimal for playing the game" (Rohlfshagen *et al.*, 2017). However, the ghosts in Ms. Pac-Man can randomly act in an unpredictable way which makes the game more challenging. It can be hard not only for the players themselves but also game AIs to find the

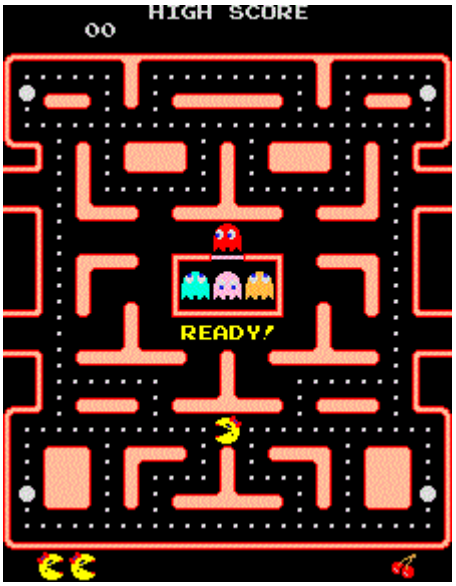


Figure 2.3: Initial state of the arcade game Ms. Pac-Man

best move given several random factors. While the main goal of the game stays the same, a few other things were changed:

- Instead of one maze, there are now four mazes which differ in their structure and are displayed in different colors. Each maze has a different number of pills. Furthermore, three of the four mazes include two tunnels and the other one has only one tunnel which allows Ms. Pac-Man to directly travel from one side of the maze to the opposite one.
- The bonus fruits no longer spawn in the middle of the maze but now can spawn randomly at any position. Moreover, they are not stationary anymore so that they can also move around the maze.
- The graphics were updated to fit a 10 years newer style as shown in Figure 2.3. Additionally, the sound effects and music were replaced and the orange ghost was renamed.

2.5 Competitions

Due to the newly introduced nondeterministic ghost AI with the release of Ms. Pac-Man, creating an AI for the main character got challenging again. Therefore, several competitions were held in the last 10 years introducing different techniques to observe the game and to control the character.

2.5.1 Ms. Pac-Man Screen-Capture Competition

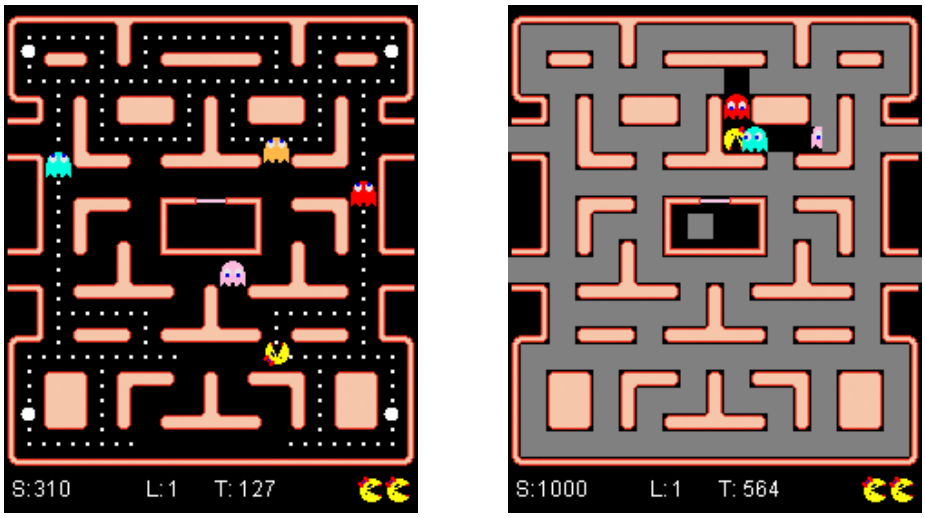
The first famous competition that was held in this context was called Ms. Pac-Man Screen-Capture Competition (Lucas, 2007). As the name already suggests, the game was observed by capturing the current screen and converting it into an internal game state. The competition ran from "2007 to 2011, with the last proper run in 2011 at the IEEE Conference on Computational Intelligence an Games (CIG)" (Rohlfshagen *et al.*, 2017). The game was either provided "as a Microsoft Windows application or as a Java applet" (Rohlfshagen *et al.*, 2017). Furthermore, a basic screen-capture kit was provided to the participants. The main effort of the competition winners was to build a good feature detection that provided the controller with all required information. For comparison, multiple runs were executed resulting in the controller with the highest average score being the winner. The best results were achieved by implementing a copy of the default ghost AI to achieve high performance. However, the results were still worse than the human high score.

2.5.2 Ms. Pac-Man vs Ghosts Competition

The main differences to the previous competition are that the Ms. Pac-Man vs Ghosts Competition (Rohlfshagen and Lucas, 2011) firstly no longer includes the screen-capturing aspect and secondly a controller for both, Ms. Pac-Man and the ghosts, could be provided making it a two-player game. The goal of Ms. Pac-Man stayed the same as before whereas the goal of a ghost controller is to minimize the score of Ms. Pac-Man. The game was completely rewritten in Java which provides an interface for the competitors. However, some key aspects of the original game are missing and therefore the results cannot really be compared to the previous competition. The main differences are that there are no moving fruits, the characters use at a constant speed unless the ghosts are edible, and the tunnels are shorter than they used to be. The competition was held every half a year for a total of two years. Each participant got a score assigned based on wins against other players using a Glicko rating. Afterward, pairings were built by using these scores. Winning or losing such a pairing then resulted in an adjustment of each player's score (cf. Pepels *et al.*, 2014). In order to stick to the real-time aspect, the controllers were given a total of 40ms to compute the next move. The competition included 16 levels with a maximum time limit of 3000 steps for each. After the time limit ran out, Ms. Pac-Man was rewarded with half of the points for the remaining pills before moving on to the next level. This encourages the ghosts to be more aggressive and prevents them to group around a group of pills which Ms. Pac-Man would normally require to get to the next level.

2.5.3 Ms. Pac-Man vs Ghost Team Competition

Ms. Pac-Man vs Ghost Team (Williams *et al.*, 2016) is the most recent competition in this field and is still held every year from 2016 until now. The difference



(a) Full observability (b) Partial observability

Figure 2.4: Comparing full observability to the newly introduced concept of partial observability

to the previous competition lies in the observability. As well Ms. Pac-Man as the ghosts can only see the corridor they are currently in. Additionally, the range of sight can be constrained to an arbitrary distance. This partial observability is compared to the fully observable version in Figure 2.4. Furthermore, the ghosts can now communicate with each other in order to make better decisions. This also allows having different controllers for each ghost such that they can act individually. Additionally, the four ghosts share the 40ms to compute a move which allows dividing the time between them so that one ghost can perform a more complicated move whereas other ones just follow a simple pattern. To determine the winner, usually, a round-robin tournament is used. However, in cases with too many competitors, each team gets assigned a score according to the Glicko2 algorithm (Glickman, 2013). Afterward, the 10 best candidates still need to compete in a round robin tournament.

2.5.4 Ms. Pac-Man Controller Approaches

There already exist several different approaches on how to control Ms. Pac-Man to achieve the best results (Williams *et al.*, 2016). For instance, an Evolutionary Algorithm (EA) was used to train a Neural Network (Lucas, 2005). Although the trained Network consisted of only two layers, the agent performed well against the deterministic ghosts.
Another Machine Learning approach compared an EA and Temporal Difference Learning (TDL) with each other (Burrow and Lucas, 2009). These two methods were used to train a Multi-Layer Perceptron to play the game. In the end, the

EA was always superior to the TDL method.
Another method was to create an Influence Map that can be checked into each of the four directions (Wirth and Gallagher, 2008). Pills and edible ghosts have a positive influence while nonedible ghosts have a bad one. The agent chooses the move that maximizes the corresponding influence.
Another team of contestants used Genetic Programming to evolve heuristics that control Ms. Pac-Man (Alhejali and Lucas, 2010; Alhejali and Lucas, 2013). The main issue was that the agent focused too much on eating the ghosts instead of clearing the mazes and progress to the next maze.
There also exist several approaches which try to use a classical tree search method to find a best move for the corresponding game state. For example, a method was investigated where the tree represented each possible path in the maze and depth limit 10 (Robles and Lucas, 2009). Each node encoded the information about the ghosts and pills that are currently present. The tree was then traversed by a simulator and the different states were evaluated.
Furthermore, a five-player max^n tree was used where the players were the four ghosts and Ms. Pac-Man (Samothrakis, Robles, and Lucas, 2011). The tree was limited in depth and the approach achieved very good results.
Another quite successful approach used an MCTS as a search method (Pepels *et al.*, 2014). The MCTS was improved with several different enhancements specifically for the Ms. Pac-Man domain. The MCTS of this agent was also used for this thesis and is described in more detail in Chapter 6.
The controller Pax-mAnt makes us of an Ant Colony Optimization that was chosen with regard to two objectives. One of them is to maximize pill collection and the other one is to minimize the chance of getting eaten by a ghost. The parameters were optimized by using a Genetic Algorithm.
The last tree search based approach converted the maze into a connected graph of cells (Foderaro, Swingler, and Ferrari, 2012). The authors aimed for an algorithm that can quickly adapt to unexpected behaviors and dynamic environments.
Besides the machine learning approaches still a rule-based participated in the competition (Flensbak and Yannakakis, 2008). While avoiding ghosts that are around Ms. Pac-Man in a 4×4 range, the controller mainly concentrates on collecting pills. This approach tries to avoid the ghosts as much as possible.

2.5.5 Ghost Team Controller Approaches

Due to the fact that the competition previously accepted controllers for Ms. Pac-Man only, there has not been much research on how to control the ghosts. One team, who also won the competition, used an MCTS controller for three of the ghosts while the fourth one was completely rule-based (Nguyen and Thawonmas, 2011). Each ghost has its own search tree and the goal was to create a ghost team that can adapt to many different kinds of Ms. Pac-Man controllers.
Another team used Neural Networks for the ghosts (Wittkamp, Barone, and Hingston, 2008). Each of them is evolved separately and the weights and structure are evolved by a Neuro-Evolution Augmenting Topologies algorithm.

Last, a team used Swarm Intelligence to control the behavior of the ghosts (Liberatore *et al.*, 2014). They created sets of different rules which are optimized to play against different Ms. Pac-Man controllers.

Chapter 3

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm which is often used in a gameplay context (Coulom, 2007; Kocsis and Szepesvári, 2006; Browne *et al.*, 2012; Chaslot *et al.*, 2007). As the name already suggests, MCTS works by building up a search tree. The nodes in the tree represent different game states. The edges represent all possible moves which can be played in a game state. Consequently, the children of a node are the game states, which are reached by playing the move that corresponds to the edge between them. A big advantage of MCTS is that it does not need domain knowledge to compute its results. Compared to most other search frameworks no evaluation method is needed. The "value" of a game state is purely evaluated by running many simulations and looking at their outcome. The tree search consists of four different phases (Figure 3.1), which can be repeated an arbitrary number of times to improve the result. The following chapter explains these four phases and shows how they help to progress through the tree.

This chapter is structured as follows. First, Section 3.1 describes the method for selecting nodes in the tree. Section 3.2 shows what happens once the selection method reaches a leaf node. In Section 3.3 the method of expanding the tree is described. Once a new node was added to the tree, its value needs to be backpropagated to the root. This functionality is explained in Section 3.4. Finally, Section 3.5 explains how the result of an MCTS can be used to extract the assumingly best move from the tree.

3.1 Selection

The root always is the game state for which the best move needs to be found. As previously mentioned, MCTS continuously builds up this tree and therefore needs a selection method to traverse through it. This selection method decides which node will be investigated next. The most commonly used method is called "Upper Confidence Bound" (UCB1) (Auer, Cesa-Bianchi, and Fischer, 2002). A

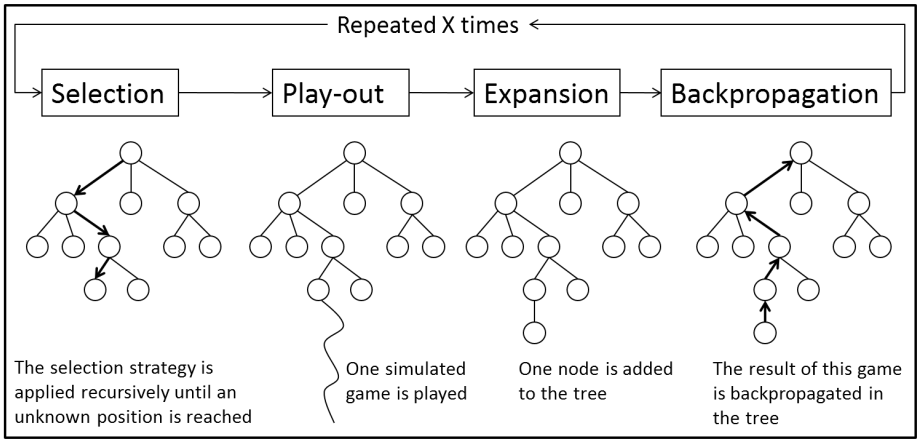


Figure 3.1: The four phases in a Monte Carlo Tree Search. Figure taken from Winands (2017).

MCTS that uses this selection method is called "Upper Confidence Bound 1 applied to Trees" (UCT) (Kocsis and Szepesvári, 2006). Among all the children of a parent node p, the child c is selected that maximizes Equation 3.1.

$$UCB1 = \frac{w_i}{n_i} + C \times \sqrt{\frac{ln(n_p)}{n_i}} \tag{3.1}$$

This equation tries to find a balance between exploration and exploitation.
The first component is considered to be the exploitation part. Parameter w_i represents the number of the considered wins for the node. Parameter n_i stands for the total number of simulations that have been done for the node. Consequently, the fraction represents an estimation of the winning probability which leads to exploitation of moves with a higher win ratio.
The other component can be seen as the exploration part. n_p stands for the total number of simulations that have been executed in the subtree and n_i is again the number of simulations for the current child node. If the node is selected only a few times or was not yet selected at all, n_i stays very low. Due to the fact that n_p gets increased with every simulation in the subtree while n_i only gets increased for visits of the child node, the fraction in the square root becomes higher for nodes that were selected less often. Therefore, the second component encourages the selection function to explore these nodes.

Consequently, variable C takes care of the relevance of the exploration part. Choosing a lower value for C prefers nodes that have a promising win probability. Choosing a higher value for C prefers nodes that have been investigated fewer times than other nodes. A choice for that parameter can be around 0.4 (Winands, 2017).

3.2 Play-out

Once the selection process reaches a leaf node which does not result in a clear win or loss, a play-out step needs to be done. The game is usually played with quasi-random moves for each player. These moves are sampled from a probability distribution. This way, more promising moves are preferred but are not guaranteed. The reason behind this is to keep the computations as cheap as possible and also make runtime for one play-out faster. However, this cannot always be guaranteed. Sometimes, playing random moves might be reducing the number of computations as the games can take much longer than play-outs that follow a certain policy. Usually, a play-out runs until the game is over and the result is stored. However, if the games take so long that not even one play-out can be run to an end, they are only run until a certain depth. Keeping the runtime of play-outs shorter allows running many consecutive simulations and increases the accuracy of the approximated value.

3.3 Expansion

The leaf node which was selected for the play-out phase had no known children yet. Therefore, once the play-out phase is over, the corresponding leaf node gets expanded with the newly achieved results. One method is, to add the whole play-out branch to the tree. This way, all played actions are preserved. However, storing this information can often lead to memory issues. Therefore, the leaf node usually gets expanded with only one child node corresponding to the first action of the play-out phase. Afterward, the value gets backpropagated recursively through the tree up to the root node.

3.4 Backpropagation

As shown in Figure 3.1 the backpropagation process updates all nodes on the path to the leaf node. In a standard UCT, the visit count n of every node on the path gets increased by 1 and the win count w gets increased by 1 depending on the outcome of the simulations. These updated values are then used for the new UCB1 scores as shown in Equation 3.1.

3.5 Final Move Selection

Once the backpropagation is done, the whole 4-phase process starts over again and is repeated until the computational budget is used. If needed, the MCTS may also stop if a guaranteed win was found. A big advantage of this process is that it can be interrupted at any given point and can then return the so far best move according to the selection function.

This best move can be chosen according to different criteria (Chaslot *et al.*,

2007). Using the UCB1 function at the root again to return the best move is usually not done as there exist many better alternatives. The most often used methods are to either return the child node that was investigated most often or the one that was discovered to have the highest win probability. Which of these methods is chosen can also depend on the game for which the MCTS was run.

Chapter 4

Deep Learning

This chapter first describes Neural Networks in general in Section 4.1. Afterward, Section 4.2 describes a specific type of Neural Networks, namely Convolutional Neural Networks (CNNs). Furthermore, the advantages compared to other Deep Learning techniques are shown. Moreover, different types of layers that can be used in a CNN and their purpose are explained. Additionally, a way of how to combine these layers is explained. Finally, a description of how a CNN can classify different inputs is given.

4.1 Neural Networks

Deep Learning (DL) is a subclass of machine learning which is mainly based on Artificial Neural Networks (Schalkoff, 1997). Artificial Neural Networks are built in a layer-wise structure as shown in Figure 4.1. Passing a specific number of input features to the input layer is the first step in a Neural Network. The arrows between the different layers in Figure 4.1 represent the connections of the nodes. Each connection can have a different value and therefore contributes to the next layer with a different weight. Between input and output, there can be an arbitrary number of hidden layers depending on the problem the network tries to solve (Haykin, 1999). Such a network is called Deep Neural Network (DNN). Usually, DNNs are used for speech recognition, natural language processing, audio recognition, bioinformatics etc. With their structure, Neural Networks try to process information like a biological nervous system. For multi-dimensional problems like image or object classification, CNNs are used. They accept multi-dimensional inputs and convolute them during the process.

4.2 Convolutional Neural Networks

CNNs are mainly used to process image and audio data (Krizhevsky *et al.*, 2012). This type of data can be seen as multi-dimensional matrices. Images have a certain height and width and in the case of RGB images, they consist of

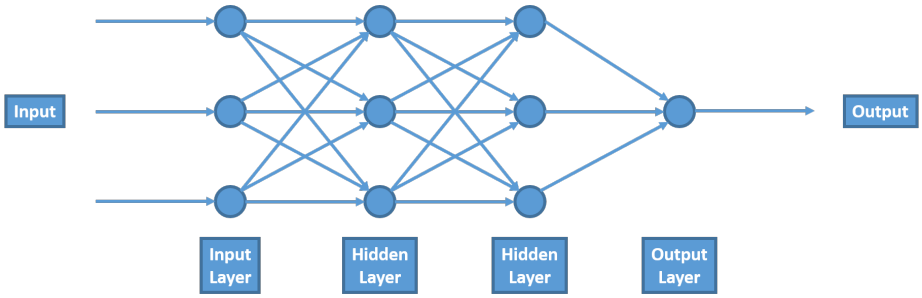


Figure 4.1: Example structure of an Artificial Neural Network

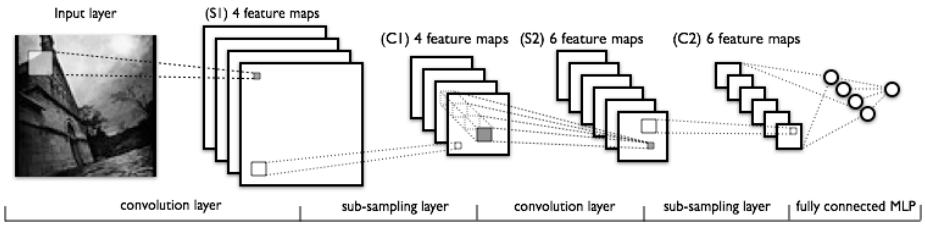


Figure 4.2: The example structure of a Convolutional Neural Network. Figure taken from Skymind (2017, accessed February 26, 2018).

three channels. Audio data can be split up into multiple short periods of time which represent the different channels. The width and height of one channel then depend on the length of such a period and the amplitude of the signal (Hershey *et al.*, 2017). The general purpose of CNNs is to perform classification or tagging on the given input data. Figure 4.2 shows a possible structure of a CNN which is used to classify objects on different images.

4.2.1 Layers

As the name already suggests, the layers in a CNN perform convolution operations on the data. These operations can be seen as filters which are applied. Similar to the input, the resulting channels are also multi-dimensional. This section briefly describes the most important types of layers which were later also used for the final product of this thesis.

Convolutional Layer

The most commonly used layers in a CNN are Convolutional Layers. A filter with a predefined size is shifted along the input matrix and at each position, a convolution is applied. In each of these convolution steps, every node of the input is multiplied by the corresponding node in the filter. The sum of these multiplications is then the result of this convolution. If the input consists of multiple channels, the filter is applied to each channel simultaneously and the

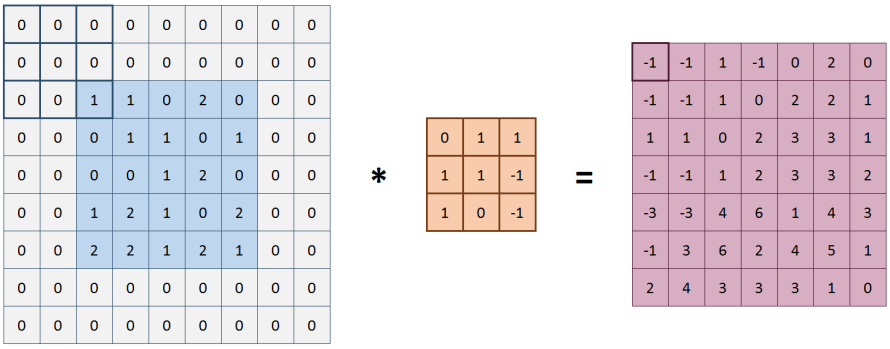


Figure 4.3: Example of a Convolutional Layer

results are again added up. Furthermore, a stride for the horizontal and vertical directions needs to be configured. This stride implies how much the filter gets shifted to the right after one convolution was applied and how much the filter is shifted to the bottom after the filter reaches the right side of the input. The shape of a filter is usually quadratic.

A problem that comes with convolutional filters is that they reduce the size of the input. If a 2×2 filter is applied, the width and height of the output are reduced by 1 compared to the input. For instance, if a 3×3 filter is applied, the width and height are reduced by 2. To avoid this issue, a method called Padding is introduced. The thought behind padding is to artificially extend the input to each direction such that the shape of the output will have the same shape as the original input. The most often used technique is called Zero Padding where zeros are getting appended to each side of the input matrix. An example of a convolution that includes Zero Padding is shown in Figure 4.3.

Because of the described process, every filter only generates a 2-dimensional matrix regardless of the number of channels in the input. However, not only the size and stride of a filter but also the number of filters can be configured. Each of these filters creates its own output matrix. Consequently, the number of channels in the output is equal to the number of filters that are applied.

Activation Function Layer

Activation functions are most often used directly after a Convolutional Layer in state-of-the-art CNNs. The most commonly used activation function is called "Rectified Linear Unit" (RELU). A RELU is used to remove negative values after the data was filtered. Compared to other activation functions RELUs usually improve the training of a Neural Network the most (Glorot, Bordes, and Bengio, 2011). Another activation function which is used in the final product is called "Softmax". This function is used to represent the data as a probability

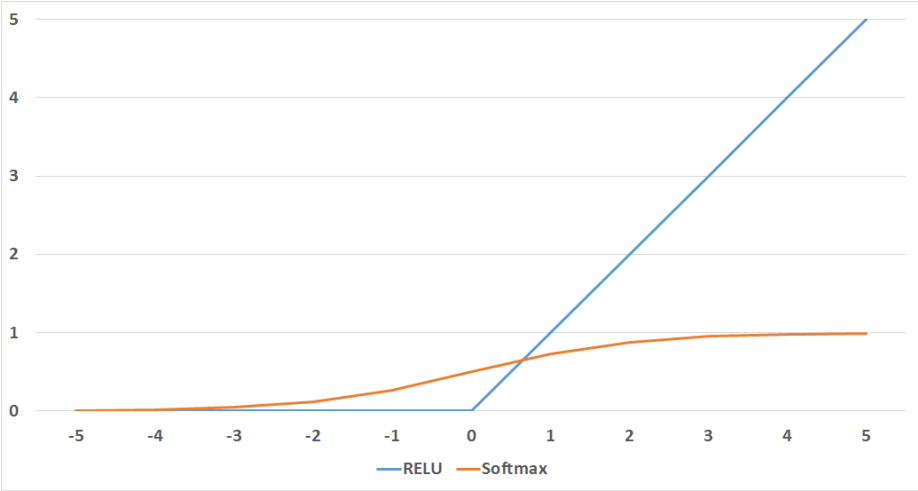


Figure 4.4: Example of the RELU and Softmax activation functions

distribution over all different outcomes. An example of these two activation functions is provided in Figure 4.4.

Pooling Layer

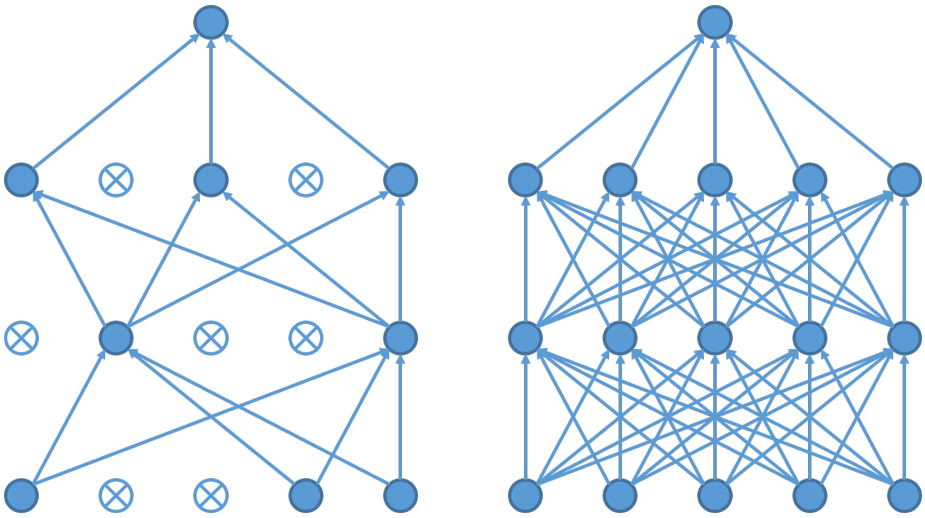
Pooling is a technique to reduce the size of the data and by that increasing the speed of the network. There exist several different types of pooling but the most common one is "Max Pooling". Similar to the Convolutional Layer the size of the filter and the stride can also be adjusted but a common choice is 2×2 with a stride of 2 (Giusti *et al.*, 2013). The filter outputs only the maximum value in the current window meaning a filter of size 2×2 would half the size of the input data. Reducing the size can have different effects. Besides decreasing the number of computations needed, also often minor details in the data are negligible and can, therefore, be removed by Max Pooling.

Fully Connected Layer

Compared to other layers in a CNN where connections between different neurons are not mandatory, a neuron in a fully connected layer is connected to every neuron of the previous layer just like in typical Artificial Neural Networks. This type of layer is also often applied to flatten the network and reshape the data to fit the output format.

Dropout

Dropout is used to avoid over-fitting of the network (Srivastava *et al.*, 2014). Dropout can theoretically be applied to any type of layer but is most often used



(a) Dropout active (b) Dropout inactive

Figure 4.5: Example of Dropout

in a fully connected layer. If the dropout technique is used some randomly chosen neurons in the network become deactivated for the next iteration as shown in Figure 4.5. This encourages the network to learn the same representation with different neurons. Because of that, Dropout also makes the network more robust to noisy data and improves generalization. Usually, Dropout is only applied to the Fully Connected Layers at the end of a CNN with 50% of the neurons getting disabled (Hinton *et al.*, 2012). However, other studies show that applying Dropout to Convolutional Layers can also lead to good results (Park and Kwak, 2016). In these cases, the percentage of disabled neurons was set to 10%-20%. It is important to mention, that Dropout should not be applied during the prediction phase because it only aims to enhance the training process.

4.2.2 Classification

A CNN is often used to perform classification. Usually, the output layer is a Fully Connected Layer and the number of neurons corresponds to the number of possible classes. After prediction, every output neuron contains a value describing how probable the input fits the corresponding class. This output can then be treated in different ways. Either the maximum is taken as the final outcome, or the order of different probabilities can be investigated further. Often, a Softmax is applied to the output neurons normalizing their values, to sum up to 1 and by that build a probability distribution.

Chapter 5

Combining MCTS and Deep Learning

The field of using MCTS as a search method for games has become more and more popular (Browne *et al.*, 2012). The idea to combine it with Neural Networks to improve it even further has become even more popular after Google DeepMind published their Go game AI Alpha Go (Silver *et al.*, 2017b). There are several ways of using a Neural Network in combination with an MCTS. The following chapter introduces different approaches that have already been used to enhance an MCTS by using DL techniques.

5.1 Real-Time Atari Games

Already in 2014, a group of researchers tried to improve an MCTS architecture with neural networks (Guo *et al.*, 2014). Until then model-free environments combined with DL were the best real-time agents. The authors built 4 different types of MCTS-based agents combined with convolutional neural networks and compared them to the raw neural network approach.

- **UCTtoRegression**: A regression-based network that tries to predict the action values of the tree search.
- **UCTtoClassification**: A classification based network that tries to predict the action choice of the tree search.
- **UCTtoClassification-Interleaved**: To avoid overfitting the training data a third type is introduced. Both of the previously mentioned methods are trained only on data generated by the MCTS. This approach starts the same way but continues to let the network itself decide actions for the MCTS. Once the MCTS evaluation has finished, the convolutional neural network is updated accordingly. By that, new artificial training instances are generated.

- **off-line UCT**: This model works without any real-time constraints and was added for comparisons of performance due to the time limitations that come along with real-time agents.

The agents were trained to play a collection of six different ATARI real-time games. The Atari Learning Environment (ALE) contains among other things emulators for several ATARI games. The inputs of the networks are screen capture based so that the images had to be preprocessed. Therefore, the images were converted to gray-scale and were cropped to only include the playing area of the game. Additionally, the pictures were down-sampled and the pixels were normalized to lie in a range between -1 and 1.

During training, the exploration rate is set to 5% meaning that the MCTS selects a random move by that chance. For reasons of completeness, the authors also include a version of each agent which selects actions greedily every time. It is worth to mention that these greedy versions perform better on average but might suffer more from overfitting because no random actions are explored during training. As expected, the off-line agent usually outperforms the other ones by far. Depending on the game it can achieve more than three times more points compared to the second best method. Surprisingly, all of the MCTS-based methods work better than the raw Neural Network approach. On average the UCTtoRegression method performs worse than the classification agents. This indicates that given that infrastructure a move predicting architecture works better than a value predicting one. Comparing the two classification methods, it turns out that the interleaved version works better on average. This strengthens the author's suggestions about the overfitting problem that comes with the UCTtoClassification agent.

All in all, the researchers showed that there was still a lot of room for improvements at that time. The best method until then was easily outperformed by their newly introduced architecture. Furthermore, it seems that a move predicting network works better than a value predicting network in an MCTS.

5.2 AlphaGo

Go is an Asian 2-player board game. Each turn, a player gets to place a stone on one of many predefined positions. The goal is to capture areas by enclosing them. In the end, the winner is the one who captured the bigger part of the board. Because of the enormous search space, Go was thought to be competitively playable by computer programs not earlier than a decade away from now. However, by using Neural Networks, Google DeepMind was able to overcome these complexity issues and to build a champion defeating game AI AlphaGo in 2016 (Silver *et al.*, 2016).

First, two different Convolutional Neural Networks were built which both take the board positions as a 19×19 image as input. One of the networks is used

as a policy provider and returns probabilities of moves which are considered to be good. The other network is trained as a value network and can output a score for the current player given a certain board. In the end, both networks are combined in an MCTS.

The policy network was trained using two different learning techniques. First, the aim was to predict expert moves by using supervised learning. For that, the KGS dataset consisting of a large number of expert moves was used. After training the network, it achieved an accuracy of 57% which is on average 13% higher than similar networks were capable of at that time.

Subsequently, the researchers use reinforcement learning to improve the network even further. Initially, the weights from previous training were kept and games were simulated by letting the newest version of the policy network and a randomly selected older version compete against each other. The results were back-propagated each time to maximize the expected outcome. By that, the new version of the policy network won more than 80% of the games against the older version that was trained using purely supervised learning. Furthermore, 85% of the games against PACHI (Baudiš and Gailly, 2011), an open-source Go AI, resulted in wins for the DeepMind's policy network whereas other state-of-the-art networks only won 11-12% of the matches.

The value network also makes use of reinforcement learning. First, state-outcome pairs were built using the same KGS dataset. Additionally, a self-play dataset was generated by sampling to avoid overfitting to the expert moves. The resulting MSE was close to being as good as the MCTS rollout results but used "15.000 times less computation" (Silver *et al.*, 2016). The training itself was done first on a single machine with "40 search threads, 48 CPUs, and 8 GPUs" (Silver *et al.*, 2016). Later, a distributed version was implemented that made use of multiple machines resulting in a total of "40 search threads, 1202 CPUs, and 176 GPUs" (Silver *et al.*, 2016).

Combining the two networks in an MCTS led to even better results. The search traverses the tree, as usual, using a function that determines which action to explore. After reaching a leaf node, the probabilities that are yielded by the policy network are considered for the expansion step. Afterward, the evaluation of a newly expanded child is done by using the value network as well as by doing a play-out to a certain depth. Eventually, both evaluations are combined to give a final leaf evaluation.

Another important thing to mention is that they tested several versions of their generated networks. The policy network that was trained by solely supervised learning (policy1) and the one that was additionally trained with reinforcement learning (policy2). Furthermore, one value network was generated from the supervised learning only policy network (value1) and another one from the final version of the policy network (value2). Consequently, 2 networks of each

type were evaluated. It turned out that the policy1 actually performed better than policy2 which is explained by the fact that humans usually play towards an overall situation whereas the network after reinforcement learning tries to go for the currently best-considered move regardless of the overall situation. However, as expected value2 performed way better than value1 because it was simply trained on a higher number of different scenarios.

Using the explained methods to determine the best move, AlphaGo won 99.8% of the games against other Go programs. Even with different handicaps, the win-rate stayed above the 75% mark. Furthermore, AlphaGo was the first computer program that was able to defeat a professional human player. Consequently, DeepMind successfully overcame the problem to find good moves in an extremely large search space.

5.3 AlphaGo Zero

In 2017 the same company, Google DeepMind, took it to a new level. They introduced a new game AI AlphaGo Zero (Silver *et al.*, 2017b) that completely gives up the idea of supervised learning and works completely without human data. Another big difference towards their previous version is the combination of the policy and value networks into one residual network that can do both. Furthermore, the rollout step of the MCTS was replaced and the selection now only relies on the value output of the network.

The input of the new network now consists of the raw board representation plus the history of the board. The researchers built the network by using "residual blocks of convolutional layers with batch normalization and rectifier nonlinearities" (Silver *et al.*, 2016). The training itself is done in a reinforcement learning self-play manner. While playing the game against itself, the network is repeatedly updated and the newer version is used in the next iteration. The selection process is done, similar to the previous version, by traversing through the search tree using a specific function that balances exploration and exploitation. Once a leaf node is reached, the expansion is done by using the neural network. Probabilities of the newly encountered child nodes are set according to the probabilities the network yields and the value of the investigated node is updated. Afterward, the value is back-propagated as usual. After the search is done, the resulting search probabilities and values are passed to the neural network for training. This process is repeated always using the newest version of the neural network. In comparison to the previous version of AlphaGo, AlphaGo Zero relies on less computational power. AlphaGo Zero is run on only a single machine with 4 Tensor Processing Units. The previous version "was distributed over many machines and used 48 TPUs" (Silver *et al.*, 2017b).

Starting from completely random play, AlphaGo Zero quickly improved in its play style. The process of learning is illustrated using a function that displays

the Elo rating (Coulom, 2008) over time. Although the previous champion-defeating version of AlphaGo achieves a higher score at the beginning of the training, it is surpassed by AlphaGo Zero after only three days of training. That can be explained due to the fact that the previous version already gets examples of professional plays whereas AlphaGo Zero has to learn everything from scratch. The final version of AlphaGo Zero was trained for a total of 40 days on the above-mentioned architecture. All in all, DeepMind has shown successfully that learning without supervision is indeed possible and can even lead to better results than similar supervised methods.

In the same year, DeepMind published a more generic version of AlphaGo Zero called AlphaZero (Silver *et al.*, 2017a). The training process itself is similar to what happened in AlphaGo Zero. However, besides being able to play Go, AlphaZero is also capable of playing Shogi and Chess with superhuman performance. The agent could defeat world-champion programs in each of the three games.

5.4 Reward Design in Real-Time Atari Games

Four of the five researchers who collaborated in the Real-Time Atari Game Play paper (Guo *et al.*, 2014) published a new paper in 2016 about adding an internal reward-bonus function to an MCTS (Guo *et al.*, 2016). This function is trained by using DL and tries to overcome the computational issues of MCTS and at the same time allow to provide a reward function that can be independent of the in-game score.

A disadvantage of MCTS considering real-time games is the simulation step. The number of simulations that can be done while staying beneath a certain time limit is severely limited. Therefore, the resulting values can be inaccurate and are often incomparable to the actual values. By building a "Policy-Gradient for Reward Design with Deep Learning" (Guo *et al.*, 2016) (PGRD-DL), it becomes possible to not only get a value given a certain state faster but also making that value dependent on self-learned features.

The reward function that is considered at each node is adjusted by not consisting of only the objective reward like usually but also include that output of the CNN. Adding these two terms results in an internal reward which is then used for action selection in the MCTS. For time-saving reasons, a frame-skipping technique (Mnih *et al.*, 2015) is used. This technique only selections actions on every 4th frame but the evaluation can still keep up with one that evaluates every frame. Similar to their previous approach, they again downsample, gray-scale and normalize the input images. Their proposed CNN consists of three hidden layers, each followed by a rectifier non-linearity unit. The learning rate is initially set to 10^{-4} and is divided by 2 every 1000 game simulations.

The authors compare the MCTS that uses internal rewards with the one that uses objective rewards only. Additionally, the scores for a deeper and a wider MCTS are added to have a comparison towards other improvements. A total number of 25 ATARI games are compared using these four agents. In 18 out of these 25 games, the MCTS with internal rewards improves the default MCTS. Furthermore, in 15 out of the 18 improved versions, the internal reward MCTS even outperforms the deeper and wider versions of MCTS. This indicates, that resources for improvements should better be laid on improving the reward function instead of increasing the search space.

In their paper, the authors successfully showed that a reward function built by DL can improve the performance of an MCTS better than other methods. Additionally, the concept of learning features which cannot be considered the usual way is proven.

5.5 Search Node Replacements

In 2016 Tobias Graf and Marco Platzner investigated methods to reduce the number of nodes in an MCTS by also using convolutional neural networks (Graf and Platzner, 2016). Speeding up the execution time of the search framework is a recent topic because such an environment is often used in real-time games where the computation time for each move is limited. To have a comparison, they additionally include an upper bound. This upper bound is estimated by neglecting the real-time constraint.

Altogether, the authors compare four different strategies to reduce the number of evaluated nodes in the tree search. The authors assume that the nodes are usually getting evaluated by using a fast move predictor. A node getting replaced means that instead of evaluating with that predictor, the knowledge of the learned CNN is applied.

- **Replace by Depth**: All nodes with depth $\leq D$ are getting neural network knowledge while the others only use the fast classifier.
- **Replace in Principal-Variation**: The nodes that are considered the most important get CNN knowledge while others do not. Therefore, some nodes already need to be assumed as being important in the beginning. These nodes are then initialized using the CNN knowledge while all the other nodes are initialized with the fast classifier. This strategy also allows updating moves with the CNN knowledge if they become considered as important. Although this guarantees an early initialization with the neural network knowledge, the nodes that are considered most important in the beginning are only assumed and not proven.
- **Replace by Threshold**: At the beginning, all nodes are initialized using the fast classifier. Once a node was visited more than T times the

fast knowledge gets replaced by the CNN knowledge. The advantage is that only often visited nodes are updated but on the other side the CNN knowledge is applied very late.

- **Increase Expansion Threshold**: This strategy replaces the fast classifier knowledge in a node once it was simulated a certain number of times.

Each replacement strategy can either be applied synchronously or asynchronously meaning that in the asynchronous version quality of knowledge is traded for computation time.

The neural networks themselves are built of several convolutional layers followed by a softmax layer. The input consists of 20 different features that are encoding the current position. In their first experiment, the authors compare different convolutional neural networks regarding the widths of the convolutional layers. Different widths between 3×128 and 12×256 nodes were tested. The results show that the more the width of the layers is increased, the better the convolutional neural network performs. However, it also takes longer to compute the results. Therefore, the 12×128 network was used for further experiments as a trade-off between better network and speed.

For each of the proposed strategies, the authors ran several experiments with different parameters. The Replace by Depth strategy usually gets outperformed by the Increase Expansion Threshold strategy if the threshold is set high enough. However, this also means that the CNN knowledge is only applied in a few nodes. In the principal-variation replacement strategy, the knowledge is also applied late but not as late as in the Increase Threshold case. Moreover, this strategy shows a decent performance for all different parameters. The best working strategy is the Replace by Threshold strategy. However, it is sensitive to the threshold parameter. If it is set too low or too high, the results become worse but are still better than the results of other strategies. Another thing worth to mention is, that all methods work better when they are executed asynchronously meaning that lower quality knowledge is applied faster.

In their paper, the authors successfully extended an MCTS with DL and by that improved the results. Furthermore, several techniques were tested and evaluated against each other. The best strategy applies the knowledge learned by the convolutional neural network after a specific number of simulations passed through a node.

Chapter 6

Pac-Man Agent

The final Pac-Man agent is called PacMaas. This chapter first describes the MCTS which performs as the underlying framework for the move selection in Section 6.1. Next, Section 6.2 describes the concept of so-called Belief Games, which try to deal with the partial observability. Afterward, the CNN that extends the search algorithm is explained in Section 6.3 and the way the data has to be preprocessed is shown. Moreover, the approach of combining the MCTS and the CNN is explained in Section 6.4. Finally, Section 6.5 shows the performance of the agent in the 2018 Ms. Pac-Man vs Ghost Team competition.

6.1 Monte Carlo Tree Search

The MCTS for the agent was originally built for the CIG'12 edition of the "Ms. Pac-Man Vs. Ghosts" competition (Pepels and Winands, 2012; Pepels *et al.*, 2014). The agent was able to win the competition and furthermore achieved the 2nd place in the WCCI'12 of the same competition. The agent was called Maastricht. The MCTS was improved with different enhancements that either speed up the search or introduce domain knowledge. For the new "Ms. Pac-Man Vs. Ghost Team" competition, a few adaptions were needed to get the MCTS running with the new framework. The following section describes the structure of the search tree and the features that were used to enhance the MCTS.

6.1.1 Search Tree

The tree for the search is built to fit into the maze structure (Pepels *et al.*, 2014). Each node in the tree corresponds to a junction in the maze and each edge represents a corridor between two contiguous junctions. Therefore, the value of one edge is equal to the distance between the two corresponding junctions. An example of a maze being discretized as a tree is shown in Figure 6.1.

When playing against an opponent, the player usually changes with each level of the search tree. However since the ghosts' behavior is unpredictable, the tree was chosen to be a single player tree. The moves of the ghosts are simulated while traversing through the tree leading to approximations for each node.

Furthermore, the tree allows no reverse moves meaning that the children of a child of node n_p do not include node the node itself again. This tries to make the difference in rewards between available moves larger (Pepels *et al.*, 2014). Furthermore, allowing reverse moves would lead to a larger branching factor and the tree would contain duplicate paths with similar rewards.

There exist three different tactics called "ghost", "pill", and "survival" which are explained in more detail in Subsection 6.1.3. Each node in the tree needs to store a value for each available tactic. Such a value is computed with the help of Equation 6.1 where p is the current node, N is the number of times node p has been visited, and $R_{tactic,n}$ is the reward of simulation n with the use of the given tactic. Thus, the score total for any tactic for node p is the cumulative sum of rewards for every simulation of the node.

$$S^p_{tactic} = \sum_{n=1}^{N} R^p_{tactic,n} \tag{6.1}$$

Consequently, the mean reward is defined as shown in Equation 6.2.

$$\bar{S}^p_{tactic} = \frac{S^p_{tactic}}{N} \tag{6.2}$$

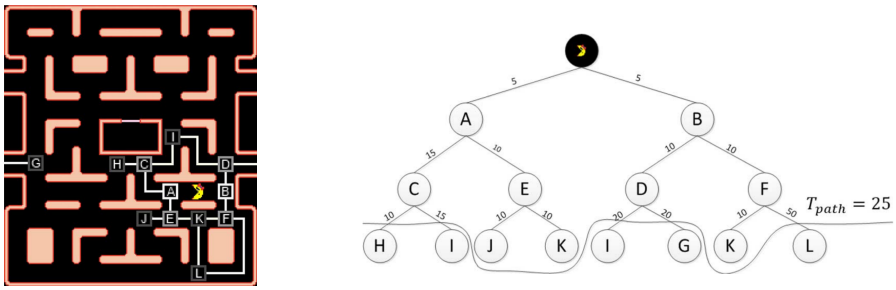
The maximum mean reward for node p is computed recursively as shown in Equation 6.3. This equation recursively selects the child that maximizes the maximum mean reward until a leaf node is reached. In other words, the maximum mean reward of node p is equal to the mean reward of the leaf node i among all leaf nodes that can be reached by traversing through node p that maximizes $\bar{S}^i_{tactic}$.

$$M^p_{tactic} = \begin{cases} \bar{S}^p_{tactic} & \text{if } p \text{ is a leaf} \\ -\infty & \text{if } p \text{ is not in the tree} \\ max_{i \in C(p)} M^i_{tactic} & \text{otherwise} \end{cases} \tag{6.3}$$

The reward values $\bar{S}_{tactic}$ and M_{tactic} are both in range [0,1].

6.1.2 Variable Depth

Other agents restricted the tree to not exceed a maximum number of edges (Ikehata and Ito, 2011). However, due to the fact that the edges represent different distances within the maze, this agent uses a maximum distance measurement to restrict the depth of the tree (Pepels *et al.*, 2014). This way, a leaf node



(a) Maze

(b) Tree

Figure 6.1: Representation of a maze (6.1a) as a tree (6.1b). Figure taken from Pepels *et al.* (2014).

only gets expanded if the distance to the root does not exceed the maximum distance T_{path}. An example of how this maximum distance looks like in a tree can be seen in Figure 6.1b.

This method has two benefits. First, it tries to maximize the short-term outcome which can be helpful in situations where Ms. Pac-Man is in danger. Furthermore, the score in the game increases over time. Due to the fact that all paths are restricted to the same distance, each of them gets the same scoring potential.

6.1.3 Tactics

As in Subsection 6.1.1, there exist three different tactics which are used according to the current game state (Pepels *et al.*, 2014). They mainly depend on how safe the current position is for Ms. Pac-Man. A threshold $T_{survival}$ is set based on the survival rate of the current simulations. Before the first MCTS iteration, a tactic is chosen according to that threshold and the following rules:

- If the maximum survival rate is above the survival threshold $T_{survival}$ and no edible ghosts are in range of Ms. Pac-Man, the "pill" tactic is chosen.
- If the maximum survival rate is above the survival threshold $T_{survival}$ and there are edible ghosts in range of Ms. Pac-Man, the "ghost" tactic is chosen.
- If the maximum survival rate falls below the survival threshold $T_{survival}$, the "survival" tactic is chosen.

These tactics play an important role in the UCT value which is used for selection, backpropagation, and final move selection. Equation 6.4 shows how the UCT value is calculated.

$$X_i = v_i + C \times \sqrt{\frac{ln(n_p)}{n_i}} \tag{6.4}$$

The exploitation part depends on value v_i which is chosen according to Equation 6.5.

$$v_i = \begin{cases} M^i_{ghost} \times M^i_{survival} & \text{if tactic is ghost} \\ M^i_{pill} \times M^i_{survival} & \text{if tactic is pill} \\ M^i_{survival} & \text{if tactic is survival} \end{cases} \tag{6.5}$$

This way, value v_i is either the maximum mean survival reward if the survival tactic is chosen or the maximum mean reward for the currently active tactic multiplied by the maximum mean survival reward.

6.1.4 Play-out Strategies

In every play-out, the moves for both, Ms. Pac-Man and the ghosts, need to be simulated. However, in the competition, a game only ends if either Ms. Pac-Man loses all her lives or after 10000 time-steps have passed. Unfortunately with the limit to evaluate single-threaded only, it is not computationally achievable to run enough simulations to an end within the 40ms time limit. Therefore, three different stopping criteria are introduced (Pepels *et al.*, 2014).

1. A predefined time limit T_{time} has been reached.
2. Ms. Pac-Man either died or is currently trapped by the ghosts with no path to escape left.
3. The current maze was completed and the simulation switched to the next one.

T_{time} is the difference between the maximum path length in the tree T_{path} and the time limit for one MCTS iteration $T_{simulation}$. This way, every iteration gets the same time limit and has, therefore, the same potential to score points. Once a simulation three values are stored, one for each tactic (Pepels *et al.*, 2014). These rewards are later used to compute the total score of a node as shown in Equation 6.1. The three rewards are defined as follows:

1. $R_{survival} = \begin{cases} 0 & \text{if Ms. Pac-Man died} \\ 1 & \text{if Ms. Pac-Man survived} \end{cases}$
2. R_{pill} is equal to the proportion of pills eaten with respect to the total number of pills at the start of the level.
3. R_{ghost} is the number of ghosts eaten and is normalized by the total edible time at the start of the simulation to fall into the range [0,1] as well.

Following these three reward values, the goal of Ms. Pac-Man during a simulation is to achieve the highest possible score while losing as few lives as possible.

At the same time, the ghosts try to reach the following three goals as best as possible:

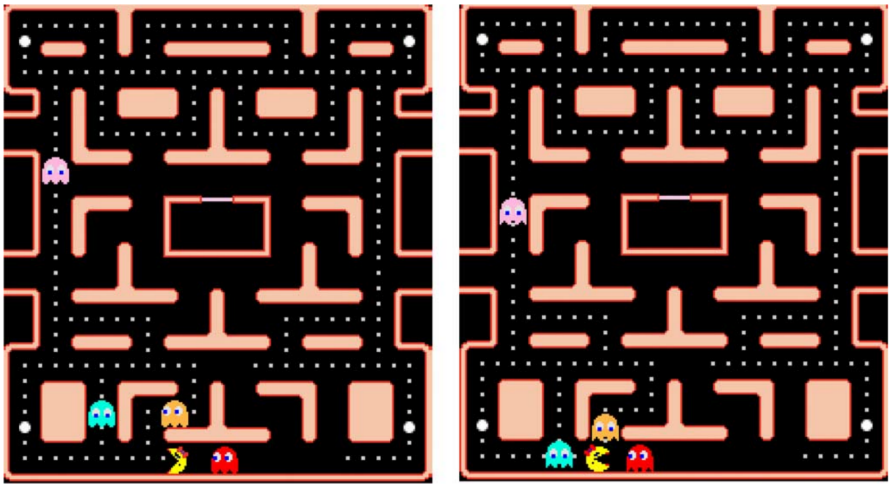


Figure 6.2: Pincer Move example taken from Pepels *et al.* (2014).

1. Trap Ms. Pac-Man so that she cannot escape and eventually loses a life. Performing such a maneuver is also called "Pincer Move" and is shown in Figure 6.2.
2. Try to keep the ghosts reward R_{ghost} as low as possible. A higher ghost reward indicates that more ghosts were eaten by Ms. Pac-Man.
3. Prevent Ms. Pac-Man from eating pills and thereby keep the number of eaten pills as low as possible.

Ghost Simulation

In the beginning, each ghost gets a random target-location vector $\vec{target}$ assigned (Pepels *et al.*, 2014). This vector determines whether the ghost should approach Ms. Pac-Man from the front or the back and prevents a ghost to approach Ms. Pac-Man from the same direction as the other ghosts. Furthermore, the ghosts follow a predefined strategy with a probability of $1 - \epsilon$ with $\epsilon = 0.2$. Otherwise, they just make a random move. The strategy itself can be divided into three different cases.

- First, if ghost g_i is currently not edible, the following rules hold in the given order.
    1. If there is a path available that traps Ms. Pac-Man, this path is chosen.
    2. If the ghost is within the range of six tiles away from Ms. Pac-Man it greedily chooses the closest path toward her.

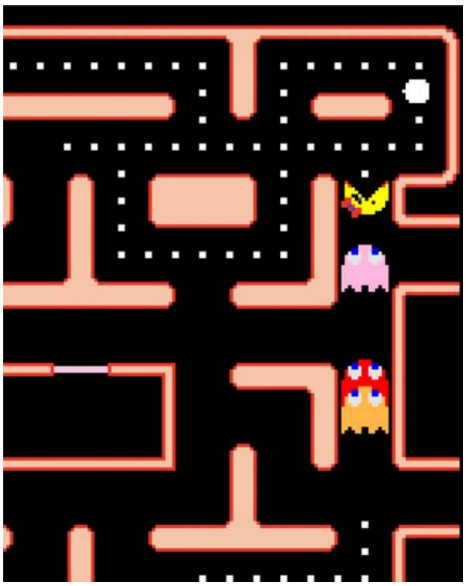


Figure 6.3: Ghosts chasing Ms. Pac-Man from the same direction example taken from Pepels *et al.* (2014).

3. If the ghost's target-location vector $\vec{target}$ indicates that ghost g_i should approach Ms. Pac-Man from the front and the g_i is closer to a junction Ms. Pac-Man is facing than Ms. Pac-Man herself, than g_i moves to exactly that junction.
4. If g_i is on a junction which is directly connected to a corridor on which Ms. Pac-Man is located on and no other ghost moves toward that direction, then the ghost moves toward that corridor.
5. If none of the aforementioned rules can be applied, the ghost moves according to its target-location vector $\vec{target}$.

- Next, if ghost g_i is edible, it tries to maximize the distance between him and Ms. Pac-Man.
- The third case overrules the other two and is applied if ghost g_i is moving toward a corridor that is already occupied by another ghost $g_{j \neq i}$. In this case, simply a random move is selected. This method ensures that spread through the maze and increases their chances to catch Ms. Pac-Man. Furthermore, it prevents the ghosts from approaching Ms. Pac-Man from the same direction or distance as shown in Figure 6.3.

Ms. Pac-Man Simulation

The simulation for Ms. Pac-Man tries to maximize the score while also playing safe (Pepels *et al.*, 2014). If more than one move turns out to be the best

according to these criteria, one of them is chosen randomly. To increase the safeness of Ms. Pac-Man a so-called "Safe Move" is performed which is defined as follows:

- The move leads to a corridor which is not occupied by a nonedible ghost that faces towards Ms. Pac-Man.
- The move leads to a corridor which next junctions are safe. This means that Ms. Pac-Man can reach all junctions before any of the ghosts can.

The simulation rules for Ms. Pac-Man differ depending on her being at a junction or a corridor. If she currently is at a junction the following rules hold in the given order:

1. If Ms. Pac-Man can reach an edible ghost before the ghost's edible time runs out, she chooses the shortest Safe Path towards that ghost.
2. If there is a corridor nearby that still contains pills and there exist a Safe Move towards it, this move is performed.
3. If there is no corridor around with pills in it, a random move is chosen that leads to any safe corridor.
4. If no Safe Move can be found, a random move is performed.

If Ms. Pac-Man is currently located in a corridor, she can either follow its path or reverse her movement. The following rules allow a reversal:

- A nonedible ghost is approaching the front of Ms. Pac-Man on the current corridor.
- A power pill was eaten and the shortest path towards an edible ghost can be reached by reversing.

Furthermore, may only reverse her direction once before leaving a corridor. Moreover, the first condition is only checked if the last move made at a junction was an unsafe move. In every other case, Ms. Pac-Man just follows the corridor without reversing.

6.1.5 Long-Term Goals

A standard MCTS only looks at the short-term rewards. However, in Ms. Pac-Man also the long-term goals can matter a lot. After eating a power pill, the ghosts need to be eaten as fast as possible to score the maximum number of points possible. Furthermore, remaining longer in one maze than needed means a higher risk of getting captured. Even more important is the fact, that Ms. Pac-Man gets rewarded with an extra life after achieving 10000 points.

These long-term goals have to be encoded in the rewards to influence the outcome of the MCTS. Therefore, the reward for eating a ghost R_{ghost} is increased

by the remaining edible time after a ghost was eaten (Pepels *et al.*, 2014). This enforces Ms. Pac-Man to eat the ghosts faster. Moreover, after eating a power pill Ms. Pac-Man has to achieve a ghost reward higher than 0.5 in this playout. Otherwise, Ms. Pac-Man could not reach enough ghosts and the pill reward R_{pill} is set to 0 as a punishment. However, if Ms. Pac-Man can achieve a ghost reward that is higher than 0.5, the pill reward is additionally increased by this achieved ghost reward. This method ensures that Ms. Pac-Man waits for the ghosts to be close and group up before she eats a power pill.

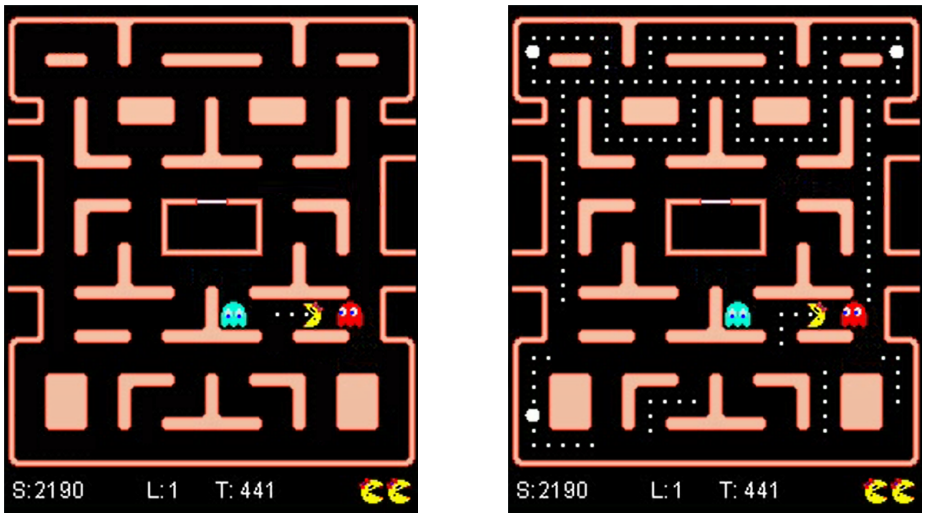
The mazes need to be cleared as fast as possible because the game progresses to the next maze after 10000 time-steps and the remaining pills are lost. To enforce a better clearing of the pills and by that, a faster progression through the maze, a so-called "Edge Reward" is introduced. This means that the pill reward R_{pill} is only increased if the complete corridor is cleared. Thus, Ms. Pac-Man tries to eat all pills in a corridor instead of leaving a few behind which can become hard to reach at a later stage. Ms. Pac-Man should clear long corridors when it is safe to do so but it could happen that clearing multiple short corridors results in a higher reward. To counteract this, the corridor reward is defined as $R_{edge} = num_{pills}(e_i)^p$. The exponent p falls into the interval [1,2] and e_i is the cleared edge. Finally, to also reward Ms. Pac-Man even if no corridors were cleared completely, the pill reward is set to $R_{pill} = max(R_{pill}, R_{edge})$ and is normalized with regard to the total number of pills in the maze.

6.2 Belief Game

Since the complete framework has become partially observable only with the new "Ms. Pac-Man Vs. Ghost Team" competition, the MCTS had to be adapted to support the new requirements. Therefore, so-called "Belief Games" are introduced which make use of determinizations. The problem of having imperfect information about the current game state occurs in many games. Determinizing these game states has among other been tested in Scotland Yard and StarCraft (Nijssen and Winands, 2012; Uriarte and Ontañón, 2017). In both cases, the experiments lead to promising results. Instead of knowing the real positions of the enemies they are sampled by assumptions and get updated once the agent gets new knowledge. For this reason, a so-called "Pill Model" and a "Ghost Model" were implemented. Combining both of these two concepts results in one Belief Game instance. Obviously, the Belief Game must be reset once the game continues to the next level, i.e. the maze changed.

6.2.1 Pill Model

While traveling through the maze, the framework does not allow to check the presence of pills that are not in her range of sight. However, at the beginning of each level, no pill was eaten yet and the layout of the maze is known by the agent. Therefore, the exact position of each pill and power pill can be stored.



(a) Pill Model inactive (b) Pill Model active

Figure 6.4: Representation of the game with the Pill Model inactive (6.4a) and active (6.4b). With an inactive Pill Model, Ms. Pac-Man can only see the pills and power pills which are in the same corridor as her. With an active Pill Model, the positions of the remaining pills are known exactly.

Due to the fact that no one else than Ms. Pac-Man can eat a pill, the information does only need to be updated when the agent itself eats one of the remaining pills. This way, the agent can be introduced with perfect knowledge about each pill although the framework is still partially observable. Figure 6.4 shows the partially observable framework working without the Pill Model in comparison to an activated Pill Model.

6.2.2 Ghost Model

In contrast to the pills, the ghost's positions cannot be assumed exactly. They can either move with respect to different unknown policies or even in a random way in some situations. Therefore, three different types of Ghost Models were implemented which try to imitate the ghosts' behavior as good as possible. One that makes the ghosts follow a "Pincer Move Policy", one the lets them behave in a random way, and one that considers every possible move the ghosts can make. The models are then used to determinize the information about the ghosts by sampling (Whitehouse, Powley, and Cowling, 2011). The following subsection first shows the general approach of storing information about the ghosts. Next, the three different types of models are described. Finally, the method of sampling information about the ghosts from a Ghost Model is explained.

General

To store every information that is needed to guarantee a working Ghost Model, some variables are introduced:

- A variable *maze* holds the information about the structure of the maze. This is needed to know which corridors the ghosts can choose when they arrive at a junction.
- The *probabilities* variable stores a probability for each position in the maze of being occupied by a ghost. The 3D-array holds the structure of the maze for each of the ghosts where the probabilities for one ghost sums up to 1. This way, a probability distribution for each ghost is created.
- The direction the ghosts are facing plays an important role in Ms. Pac-Man. Due to the fact that the ghosts can usually not turn around in a corridor, this information needs to be stored as well. Therefore, a variable *moves* is declared which holds the information of a ghost's last move.
- If ghosts are edible, the remaining time of each ghost is individually saved in the $ghostEdibleTime$ array.
- Finally, the time for one ghost being kept in the lair is independent of the other ghosts. Therefore, this information is stored in the $ghostLairTime$ array.

In the beginning, each of the ghosts starts in the lair and remains there for a certain amount of time before it is allowed to leave. After the game starts or proceeds to the next level, the Ghost Model is introduced to the layout of the new maze. Furthermore, the *probabilities* are set to 1 for the lair position, the *moves* are set to "neutral", the $ghostEdibleTimes$ are set to zero, and the $ghostLairTimes$ are set according to the initial lair time. It is important to mention that each ghost has its own initial lair time and they leave the lair at the start with a certain delay.

At each time step of the real game, the ghost model gets updated as well. For that, each ghost is handled individually according to the following steps in the given order:

1. If the ghost's lair time l_i is greater than zero, the lair time is reduced by 1.
2. If the reduction of step 1 results in l_i being exactly zero this means that the ghost is now allowed to leave the lair. Therefore, its probability of being at the exit of the lair is set to 1 and the move is set to "neutral" since it is unknown. After that, the loop continues to the next ghost.
3. If the ghost's lair time l_i is not greater than zero and the ghost is edible right now, its edible time e_i gets reduced by one. Edible ghosts move at a

slower speed than nonedible ghosts. The framework realizes this slowdown by allowing the ghosts to move only every n-th time step where parameter n is provided by the framework. Therefore, before allowing the ghost to move, it first needs to be checked whether this current time step is the n-th one or not. If this is not the case, the loop simply continues to the next ghost. Otherwise, the ghost moves according to one of the three different Ghost Models.

4. If the ghost's lair time l_i is not greater than zero and the ghost's edible time e_i is not greater than zero as well, the ghost also moves according to one of the three different Ghost Models.

Besides these conditions, also other scenarios can happen which affect the variables in the Ghost Model:

- One of the most important events is the observation of a ghost g_i. If Ms. Pac-Man can see a ghost in her current line of sight, the position and direction of that ghost are known exactly. These values can then be updated accordingly. Therefore, the probability distribution for this ghost is set to be zero everywhere except for the current position which has probability one. Furthermore, the direction of the ghost can be updated and the *moves* variable is updated.

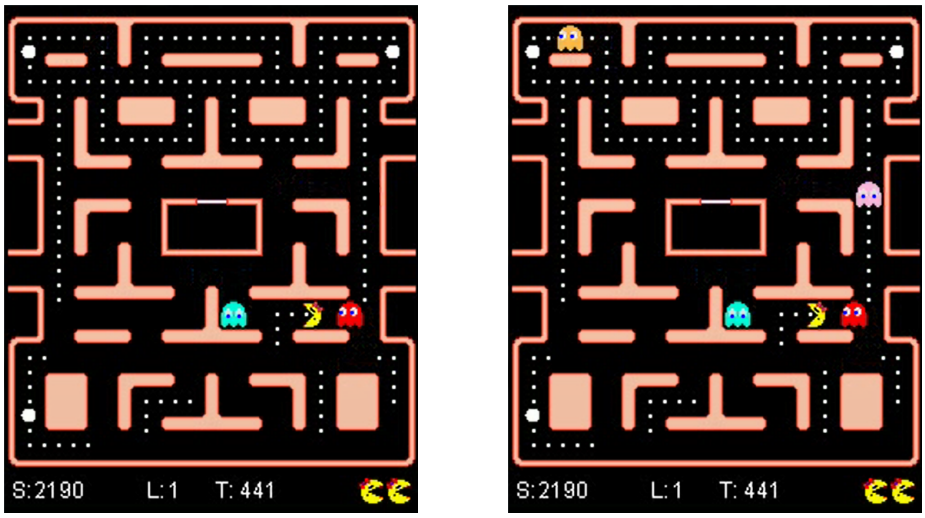
- Equally important is the observation of a ghost not being in the current line of sight of Ms. Pac-Man. In this case, the probabilities for the corresponding position are set to zero and the probabilities for the remaining positions are increased equally to still sum up to one.

- With a probability that is given by the framework, a global reversal can happen. This only affects the direction of the ghosts. Therefore, each move in the *moves* variable is changed to be its opposite move.

- If Ms. Pac-Man eats one of the four power pills, the ghosts become edible for her. However, this only happens for the ghosts which are currently not in the lair. Therefore, the edible time of each ghost which fulfills this condition is set to the initial edible time. This time also depends on the level Ms. Pac-Man is currently in. These values are provided by the framework. Furthermore, a global reversal event is triggered if Ms. Pac-Man eats a power pill.

- Once an edible ghost gets eaten by Ms. Pac-Man, it is directly sent to the lair. Therefore, all probabilities are set to zero except for the layer position and the move is set to be neutral. Moreover, the edible time e_i is set to zero and the lair time l_i is set to a specific value that also depends on the level Ms. Pac-Man is currently in.

- If Ms. Pac-Man gets caught by one of the nonedible ghosts, she loses a life. In this case, the ghost model must be reset. The probabilities are again set to be one at the lair position and the moves are set to be neutral.



(a) Ghost Model inactive (b) Ghost Model active

Figure 6.5: Representation of the game with the Ghost Model inactive (6.5a) and active (6.5b). With an inactive Ghost Model, Ms. Pac-Man can only see the ghosts which are in the same corridor as her. With an active Ghost Model, the positions of the remaining ghosts are assumed.

Furthermore, the initial lair time that is different for each ghost is stored in the $ghostLairTime$ array.

Following each of these steps, the Ghost Model can hold every information that is needed to create reasonable assumptions about the positions of each ghost. Figure 6.5 shows the difference between a game with an inactive Ghost Model and one with an active Ghost Model.

Random Ghost Model

The Random Ghost Model stores every possible position for the ghosts and assigns different probabilities to each of them. However, this method does not consider any policy for the ghosts but still lead to reasonable results. Each time a ghost is allowed to move there are two different options. First, the ghost can be in a corridor. Handling this case is straightforward because the ghost can only move along the corridor without changing its direction. Therefore the stored probability just gets shifted from the old position to the new one. The second and more complex option is that the ghost has reached a junction and needs to decide for one of the adjacent corridors. This is where the probabilities come in. To keep the Ghost Model as generic as possible, every possible move that a ghost can take at a junction is considered. Therefore, the probability of the ghost reaching the current junction is divided uniformly into the corresponding number of corridors. This procedure is also shown in Figure 6.6. However, if the

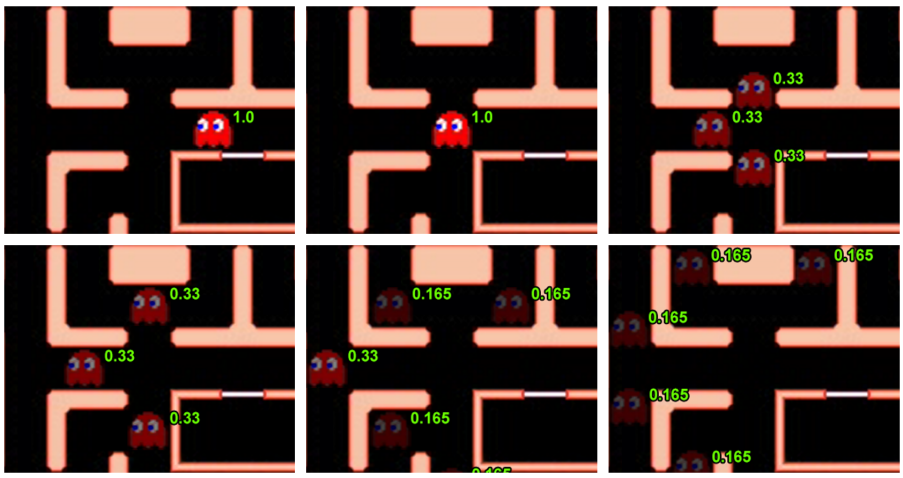


Figure 6.6: Example behavior of the Random Ghost Model

probabilities become too low i.e. fall below a certain threshold $T_{junction} = \frac{1}{256}$, these paths are not considered anymore. That is because the assumption about the ghost would become so uncertain that it simply would confuse the agent to think a ghost could still be there.

Biased Ghost Model

The problem with the Random Ghost Model is that the ghosts are not fixed to a location. They can be anywhere if the probability at that location is greater than zero. Therefore, sampling from the Random Ghost Model results in different outcomes every time. A problem is that one ghost could approach her from two sides with the same probability as shown in Figure 6.7. Sampling both of these states in two consecutive time-steps creates an impossible sequence of game states and can completely change the behavior of Ms. Pac-Man. To overcome this problem the Biased Ghost Model is used. Instead of dividing the probabilities once a ghost reaches a junction, one random path for the ghost is chosen (Figure 6.8). This way, two consecutive samples always represent a valid sequence in the game. Furthermore, the probability for one ghost always stays at 1 and is only shifted along the path of the ghost. This also needs less computational power.

Pincer Ghost Model

Similar to the Biased Ghost Model the Pincer Ghost Model also fixes the position of a ghost to one location. Additionally, instead of using a random policy, this model makes use of a more intelligent policy. The Pincer Ghost Model tries to trap Ms. Pac-Man by cutting off all of her escape paths as it was shown in

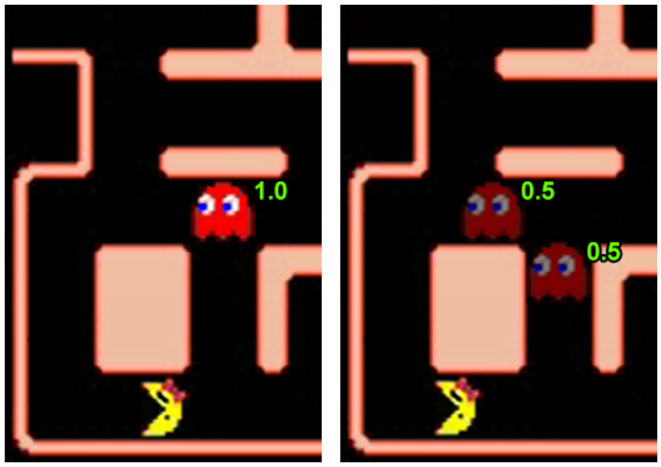


Figure 6.7: Disadvantage of the Random Ghost Model where Ms. Pac-Man can be trapped from all sides by one ghost.

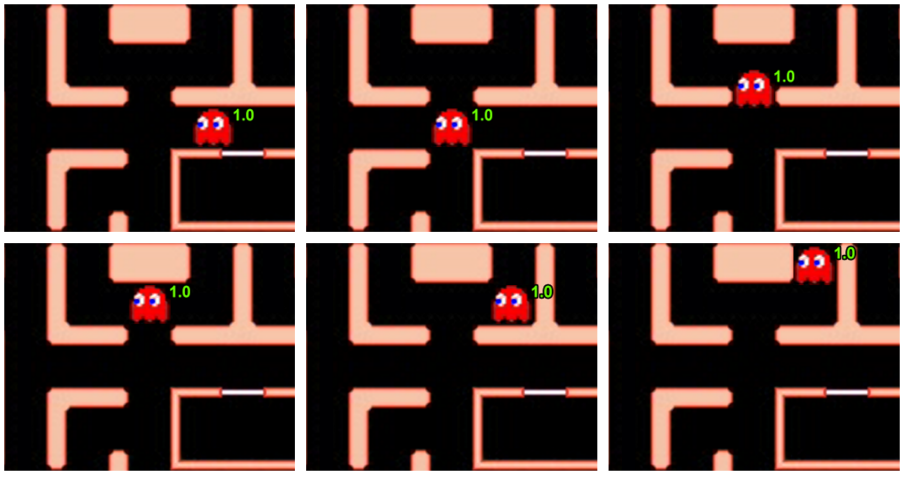


Figure 6.8: Example behavior of the Biased Ghost Model

Figure 6.2. The ghost strategy from Pepels *et al.* (2014) that was described in Subsection 6.1.4 is able to perform pincer moves. The same strategy is therefore used for the Pincer Ghost Model. By threatening Ms. Pac-Man as much as possible she is forced to play as save as possible. This ensures that the chance of being caught by the ghosts is kept low.

Sampling

For the final input for the MCTS one game state containing the four ghosts needs to be discretized. If either the Biased Ghost Model or the Pincer Ghost Model is used, the fixed locations are sampled for the resulting game state. However, the Random Ghost Model does not rely on fixed locations and sampling, therefore, needs to be done in a different way. To create a sampling technique that does also allow unlikely locations to be sampled with a low probability, first a pseudorandom number x between zero and one is chosen for each ghost. Afterward, the variable *sum* that represents the sum of probabilities of all visited locations in the maze is initialized with value zero. Then the algorithm iterates over each possible location for a ghost in the maze. If the cumulative sum of the so far visited locations surpasses the previously defined random number x, this location and the corresponding move are sampled for the ghost. This allows each location to be sampled at its own probability. Therefore if the ghost's location is exactly known, this location will be sampled. Algorithm 1 shows the exact procedure.

Algorithm 1 Sample Ghost Locations

```
1:  function sampleLocations( )
2:      locations ← Map < Ghost, Location >
3:
4:      for all g ∈ ghosts do
5:          x ← randomnumber ∈ [0, 1]
6:          sum ← 0
7:
8:          for all l ∈ mazeLocations do
9:              sum ← sum + probabilities[g][l]
10:
11:             if sum ≥ x then
12:                 locations[g] ← l
13:                 break                          ▷ continue with next ghost
14:             end if
15:         end for
16:     end for
17:
18:     return locations
19: end function
```

6.3 Convolutional Neural Network

The CNN that was used needed to be configured precisely to guarantee a good outcome while still delivering results in reasonable time. In the end, the CNN has to work together with the MCTS and the time limit is 40 milliseconds. Adding too many layers will slow down the output of the CNN. Nevertheless, adding a Max Pooling Layer will increase the speed because it reduces the size of the data for the remaining layers. Therefore, a trade-off between these factors has to be found.

6.3.1 Data Handling

To guarantee that the CNNs get every information about the game that is needed and is not biased by getting too much information, the data needs to be preprocessed. This subsection first explains how the training data is gathered and how it is preprocessed. Afterward, the meaning of the output and the way it is processed is described.

Input

The information about a game state is divided into different features which are then passed as different channels into the CNN. Storing the data in real-time is an important aspect because the features also include the history of a game state which therefore needs to be referenced. An overview of the different features in a channel representation is given in Figure 6.9 where 6.9b - 6.9g show the different channels and 6.9a corresponds to the original representation of these channels in the framework.

For the history of one game state, it is not important to include the layout of the maze again. Therefore, each element in the history adds another 5 channels to the input of the CNN (6.9c - 6.9g). For the first samples in a game, the history good refer to game states that happened before time-step 0. In these cases, simply the starting position where the ghosts are in the lair and Ms. Pac-Man is at the initial position is taken. How many elements the history should take into account and how many time-steps lie between two consecutive elements can be adjusted.

The training samples for the CNN can be considered in two different ways. Either only decisions for the junctions are taken into account and the CNN is purely trained on these examples, or also the decisions in the corridors are observed. Furthermore, the representation of the history can also change between these two functionalities. However, if the history only includes samples for the junctions, it is important to provide the corresponding time-step in the game as well because the gaps between two consecutive elements in the history can be

irregular. In this case, the corresponding time-step is provided in the top-left corner of the Ms. Pac-Man Channel. This position can never be reached by Ms. Pac-Man because each of the four mazes has a border there.

Output

The output layer in each CNN consists of four output nodes which correspond to the four different directions Ms. Pac-Man can take. Furthermore, the output is normalized to sum up to one such that a probability distribution for the four different moves is created. To select the assumingly best move that is provided by a CNN, the moves are ordered according to their probabilities. Afterward, it is checked, which moves are even possible to perform given the current game state. Removing unfeasible moves while keeping the order results in a ranking of the remaining moves. This ranking can then be used to lead the MCTS into a promising direction.

6.4 Combining MCTS and CNN

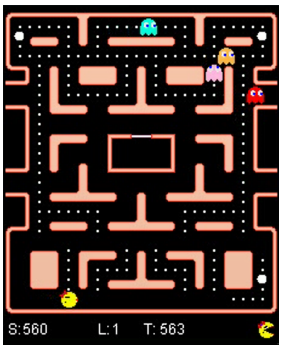
A faster CNN takes about 20 milliseconds to process an input given the hardware restrictions for the competition and the fact that the framework is allowed to only use one thread at a time. With a time limit of 40 milliseconds, it is only possible to run the CNN once per time-step and pass the output to the MCTS. To bias the MCTS the most, this output is computed for the root of the tree. To process this output two different ways of biasing an MCTS were implemented. In both of the two different approaches, H_i corresponds to the probability the CNN outputs for a specific move.

First, the so-called "Progressive Bias" can be used (Chaslot *et al.*, 2007; Graf and Platzner, 2016). The Progressive Bias is simply added to the already existing UCT function to bias it ($UCT + f(n_i)$). However, there exist different forms of it. Different examples of the Progressive Bias function are given in Equations 6.6 and 6.7. Due to the fact that the probabilities H_i fall into the range [0,1], an additional factor W can be added that enhances the contribution of the Progressive Bias ($UCT + W \times f(n_i)$).

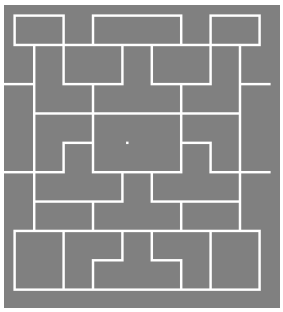
$$f(n_i) = \frac{H_i}{n_i} \tag{6.6}$$

$$f(n_i) = \frac{H_i}{\sqrt{n_i}} \tag{6.7}$$

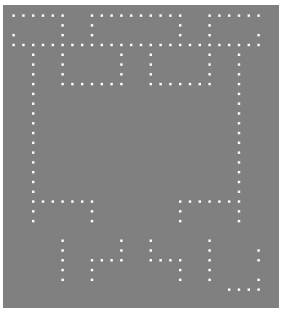
The other option is to use the "Polynomial Upper Confidence Trees (PUCT) algorithm" (Rosin, 2011). For that, the exploration part of the UCT function is changed to accept a bias. This is done by multiplication which biases the UCT equation more than an addition. Therefore, the PUCT algorithm makes the MCTS more sensible to the CNN output. Variable C, which is originally used to balance between exploration and exploitation, is replaced by the PUCT



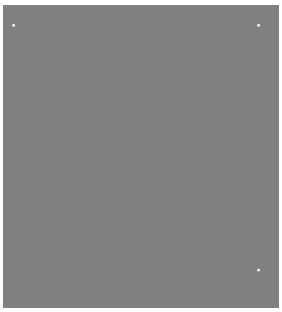
(a) Original representation in the game



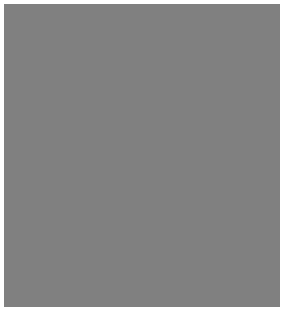
(b) Channel 1: Structure of the maze



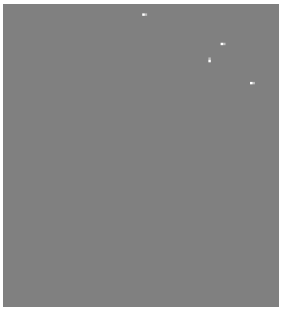
(c) Channel 2: Remaining pills



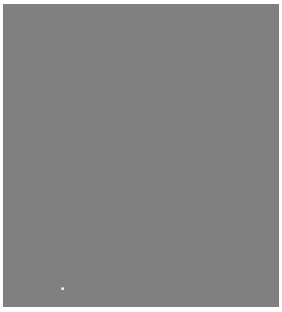
(d) Channel 3: Remaining power pills



(e) Channel 4: Positions of the edible ghosts



(f) Channel 5: Positions of the nonedible ghosts



(g) Channel 6: Position of Ms. Pac-Man

Figure 6.9: The default channels that are taken as an input for the CNN and the corresponding representation in the game. Figures 6.9e and 6.9f also encode the directions the ghosts are currently facing in a lighter gray color.

variable C_{puct}. This factor is a multiplication of the heuristic value for a move H_i and a weight W ($C_{puct} = H_i \times W$). Similar to the Progressive Bias approach, the weight can be adjusted to enhance the contribution of the PUCT bias. Again, there exist different implementations of a PUCT algorithm which are shown in Equations 6.8 and 6.9.

$$UCB1 = \frac{w_i}{n_i} + C_{puct} \times \sqrt{\frac{ln(n_p)}{n_i}} \tag{6.8}$$

$$UCB1 = \frac{w_i}{n_i} + C_{puct} \times \frac{\sqrt{ln(n_p)}}{n_i} \tag{6.9}$$

The four different equations will be evaluated on their performance to show their influence on the MS. Pac-Man MCTS. Furthermore, different values for the weight parameter will be tested to see to what extend they can change the combination.

6.5 Final Performance

The agent PacMaas participated in the 2018 Ms. Pac-Man vs Ghost Team competition. Among 5 real agents, PacMaas placed in the 4th position with a score of 6275 points. However, the competition did not allow to add external files to the submission. Therefore, the trained CNN could not be used but instead, a new untrained CNN was used. Obviously, this means a drop in performance. Participating without using a CNN would have probably increased the performance in the tournament.

Chapter 7

Experiments

To determine the settings for the best Ms. Pac-Man agent the parameters for the MCTS and the CNN have to be adjusted precisely. Furthermore, different implementations of the enhancements have to be tested. This chapter first describes the general setup and the machines the experiments were performed on in Section 7.1. Afterward, Section 7.2 shows the experiments that were performed to evaluate the different models for the partial observability. Section 7.3 shows the performance of the MCTS without a CNN to compare it to the other results. Next, Section 7.4 describes the different CNNs that were tested and evaluated on different configurations. Furthermore, the experiments for combining the MCTS and a CNN are shown in Section 7.5. Finally, the general performance against different ghost agents is shown in Section 7.6.

7.1 Setup

Most of the following experiments were executed on the "RWTH Compute Cluster" of the RWTH Aachen University. The cluster provides a batch system where jobs can be scheduled on high-end hardware. To speed up the training process of the CNNs, the training was primarily performed on NVIDIA GPUs of the Kepler architecture. In general, GPUs can perform convolutions much faster than CPUs.

The evaluations for the different Ms. Pac-Man agents were executed on the CPUs because the game and the MCTS have to run on a CPU. It turned out that there was no benefit in outsourcing the CNN output to a GPU because the time it took to get the output back on the CPU was longer than computing the output directly on the CPU itself. Furthermore, the evaluations were run manually to ensure the use of the same hardware every time. The machine is using 48 CPU cores of model type "Intel Xeon E5-2650 v4" with 2.2 GHz each. To guarantee an overall good estimation of the agent's performance, each evaluation considers the average performance after executing at least 300

runs of the game. The runs themselves were played against a ghost team called "POCommGhosts". This ghost team was provided by the Ms. Pac-Man framework and was specifically designed for the partially observable framework. The four ghosts are communicating with each other to share their knowledge and catch Ms. Pac-Man easier. Once one of the ghosts can see Ms. Pac-Man, the others are immediately informed about her position. If Ms. Pac-Man is either near a power pill or if she just ate one, the ghosts try to maximize the distance to her. Otherwise, they simply attack Ms. Pac-Man and try to catch her.

Depending on a CNN's topology it is possible to reach an accuracy of 70% or more. However, to train a CNN to that state it takes multiple weeks even with the computational power of the "RWTH Compute Cluster". Therefore, only the most promising CNNs could be trained and evaluated.

7.2 Models

The following section describes the experiments that were done to evaluate the performance of the Ghost Models and the Pill Models. The Models are compared against each other to determine which of them works best in the partially observable framework. Subsection 7.2.1 shows the evaluations of the different Ghost Models and Subsection 7.2.2 shows the improvement of an active Pill Model.

7.2.1 Ghost Model

The different types of Ghost Models help the Ms. Pac-Man agent to overcome issues of the partial observability. The Ghost Models can work independently of the CNN and are therefore compared by running simulations. Besides the three proposed Ghost Models a fourth one, which is called "Unaware Ghost Model", is added to the evaluations. This model does not store any information about the ghosts at all and Ms. Pac-Man does only know about a ghost if she can really see him in the maze. The Unaware Ghost Model was added to show the increase in performance if intelligence is added to assume the behavior of the ghosts. All of the four models were evaluated with the Pill Model proposed in Section 6.2. The results are presented in Figure 7.1.

As expected, the Unaware Ghost Model performs the worst by far. All other models score at least three times as many points on average. Moreover, the 95% confidence interval is really narrow which probably means that the Unaware Ghost Model most often cannot escape from the ghosts once Ms. Pac-Man got spotted by them. On the other side, the three other models perform much better. Although the Random Ghost Model considers every possible situation, the Biased Ghost Model Performs slightly better on average. This can be explained by the sampling technique described in Subsection 6.2.2. Due to the probabilistic sampling, it can happen that the Belief Game situation changes

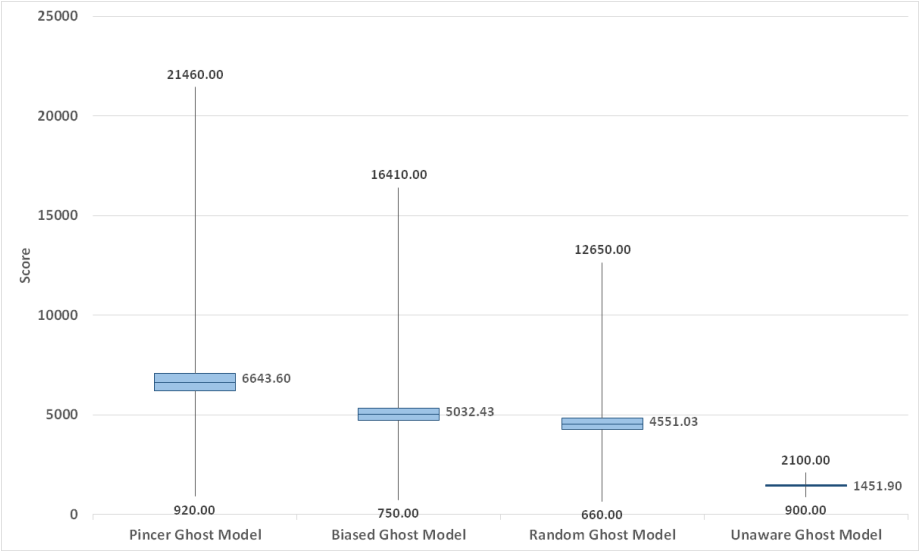


Figure 7.1: Scores of the different ghost models. The numbers show the minimum, maximum, and average score of each model respectively. The boxes show the 95% confidence intervals.

completely from one time-step to the next one. This causes confusion and irrational behavior Ms. Pac-Man. For example, a corridor is blocked by a ghost in one time-step and in the next time-step, the same ghost blocks another corridor because it was sampled at a different location. This cannot happen by using the Biased Ghost Model where the ghosts' locations are deterministic and not probabilistic. The Pincer Ghost Model performs best among all four different ghost models. It does not suffer from the described problem of probabilistic sampling and additionally outperforms the Biased Ghost Model by following an intelligent policy instead of a random one. Therefore, for further evaluations, the Pincer Ghost Model was used.

7.2.2 Pill Model

Similar to the Ghost Models an Unaware Pill Model was added to show the improvement of the Default Pill Model. The Unaware Pill Model does not store any information about the pills and a pill can only be seen once Ms. Pac-Man reaches the corresponding corridor. The two different Pill Models were evaluated with the Pincer Ghost Model. The results of the evaluations are given in Figure 7.2.

The Default Pill Model more than doubles the overall performance. However, this result was obvious because many enhancements of the MCTS, such as the play-out strategies or long-term goals, rely on knowing the positions of the re-

maining pills and power pills. Remaining corridors cannot be cleared and Ms. Pac-Man cannot proceed to the next level unless she randomly moves to one of the remaining nonempty corridors. Consequently, the Default Pill Model was activated for all other evaluations.

7.3 MCTS

This section shows the evaluation of the MCTS that was adapted to work with the new version of the game in general and especially with the introduced partial observability restrictions. The comparison to an MCTS in a fully observable framework can be seen in Figure 7.3 and Figure 7.4. Obviously, there is a performance loss when comparing the fully observable configuration to the partially observable configuration. The main reason is that the game becomes harder with the observability restrictions. Furthermore, many of the MCTS's enhancements were originally built to work in a fully observable situation. Some of them perform worse in a partially observable framework and do not increase the performance as much as they usually would do. One of the most obvious of these enhancements is the luring behavior of the agent. Before eating a power pill, Ms. Pac-Man waits until all ghosts are near her such that it will be possible to eat all of them while they are under the effect of the power pill. However, in a partially observable framework, the assumed positions do not need to be correct and waiting for the ghosts can create crucial situations.

7.4 CNN

The following section describes the experiments for the CNNs. In Subsection 7.4.1 different architectures of CNNs are compared. Subsection 7.4.2 shows how long each type of network takes to give an output. Two different methods of training a CNN are explained in Subsection 7.4.3. Subsection 7.4.4 compares the performance of the CNN under various conditions. Finally, Subsection 7.4.6 shows the influence of adding a history to the CNN input and also describes evaluations about the length of the history.

7.4.1 CNN Setup

The first versions of the CNNs were coded and tested in Python using the high-level neural networks API Keras. Keras can run on top of different Deep Learning frameworks like Tensorflow, CNTK, and Theano. In this case, the Tensorflow backend was chosen. For a shorter and better representation, the following abbreviations are used for the different types of layers and their configurations:

- **Zero Padding Layer**: Z-(size of padded rows and columns)
- **Convolutional Layer**: C-(number of filters)-(size the filter)-(stride)

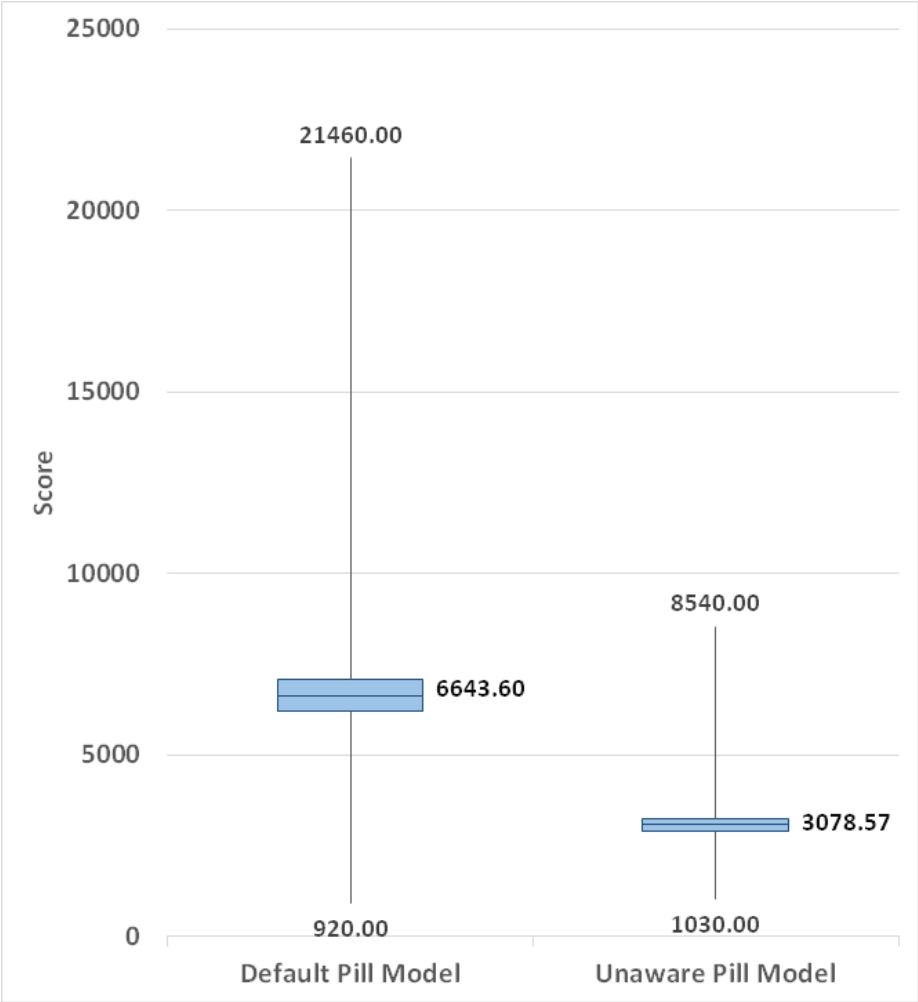


Figure 7.2: Scores of the different pill models. The numbers show the minimum, maximum, and average score of each model respectively. The boxes show the 95% confidence intervals.

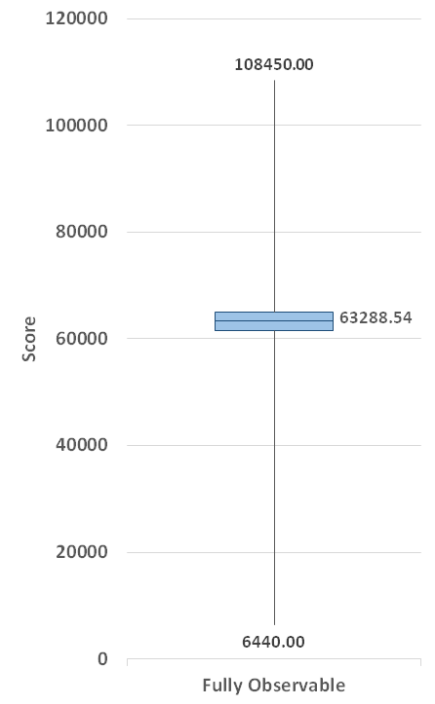


Figure 7.3: MCTS performance in a fully observable framework. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

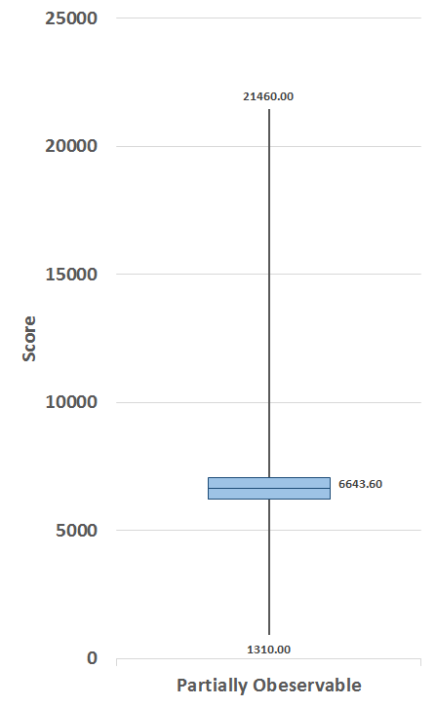


Figure 7.4: MCTS performance in a partially observable framework. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

- **Max Pooling Layer**: M-(size of the pool)-(stride)
- **Fully Connected**: F-(number of nodes)

The CNNs that were created in Keras are the following:

1. Z-2 → C-32-3-1 → M-2-2 → Z-2 → C-64-3-1 → M-2-2 → F-1024 → F-512
2. Z-2 → C-32-3-1 → M-2-2 → Z-2 → C-64-3-1 → M-2-2 → F-1024
3. Z-2 → C-32-3-1 → M-2-2 → F-1024
4. C-32-3-1 → M-2-2 → F-1024
5. Z-2 → C-64-3-1 → M-2-2 → F-1024
6. Z-2 → C-32-3-1 → F-1024
7. Z-2 → C-32-3-1 → M-2-2 → F-512
8. Z-2 → C-32-3-1 → Z-2 → C-32-3-1 → M-2-2 → Z-2 → C-64-3-1 → Z-2 → C-64-3-1 → M-2-2 → F-512

Due to the fact that the Ms. Pac-Man framework was written in Java, an interface would have need to be created that allows communication between the CNN in Python and the game itself. Obviously, this would slow things down even further and therefore the CNN for the final agent was created in Java itself. The Java library DeepLearning4j provides methods to create and train different types of Neural Networks. The CNNs that were created in DL4j are the following:

9. Z-2 → C-32-3-1 → M-2-2 → Z-2 → C-64-3-1 → M-2-2 → Z-2 → C-64-3-1 → M-2-2 → Z-2 → C-64-3-1 → M-2-2 → F-256
10. C-32-1-1 → C-64-1-1
11. Z-4 → C-32-5-1 → M-2-2 → Z-4 → C-64-5-1 → M-2-2 → F-1000
12. Z-3 → C-16-4-2 → Z-2 → C-64-3-1 → M-2 → Z-2 → C-64-3-1 → M-2-2 → Z-2 → C-64-3-1 → M-2 → F-256

All of these CNNs use a RELU activation function in all layers except for the output layer where a Softmax activation function is applied. Furthermore, a regularization value of $5e^{-4}$ and a learning rate of $1e^{-2}$ were chosen. Moreover, the "Nestrovs Updater" with a momentum of 0.9 was used. As the optimization function "Stochastic Gradient Descent" was used and the loss function in the output layer is "Multiclass Cross Entropy". The weights were initialized with the "Xavier Weight Initializer".

7.4.2 Output Time

The time the framework allows to compute a move is 40ms and the CNN should not only stay below this limit but also leave some time for the MCTS to process the CNN output. Therefore, the CNNs need to be tested for their individual output times. To evaluate the CNNs, that were originally implemented in Python, they were adapted to Java to guarantee the same conditions for each of them. After generating at least 10,000 samples, these samples were plugged into each network and the output time was measured. Each network and its corresponding output time is shown in Table 7.1.

Changing a simple aspect in a CNN and comparing the output times clearly shows which parts of the configuration influences the output time the most. The difference between CNN-1 and CNN-2 is a single Fully Connected Layer that was added at the end. However, the output time only differs in 1 ms and is basically negligible. In contrast, the influence of a Zero Padding Layer is much higher. CNN-3 takes about 8 ms longer than CNN-4 because the padded zeros also influence the dimensions of the following convolutions. Increasing the number of filters that are applied in a Convolutional Layer also increases the output time. While CNN-3 has 32 filters and takes 136 ms, CNN-5 has 64 filters and takes nearly twice the time to give an output. However, this is a logical consequence because the convolutions take most of the time in a CNN and doubling the filters also doubles the operations that are performed in that layer. Removing a Max Pooling Layer has even more impact on the output time. While CNN-3 includes a Max Pooling Layer and takes 136 ms, CNN-6 has the same configuration without the Max Pooling and takes almost four times as long. Furthermore, reducing the number of nodes in a Fully Connected Layer also reduces the output time. This can be seen by comparing CNN-3 and CNN-7. Although CNN-8 is built with many more layers than the previous CNNs, it still takes less time than all of them except CNN-7. The reason for that is that it contains multiple Max Pooling Layers which significantly reduce the size of the data and by that also reduce the computations for the following layers.

Looking at the results, it is obvious that the size of the input and the topology of the CNN do influence the output time. Adding more Max Pooling Layers results in significantly less computation time. Furthermore, increasing the dimensions with Zero Padding, adding more filters to the Convolutional Layers, or increasing the number of nodes in the Fully Connected Layers increases the output time. Contrary to expectations, these factors have by far more influence on the computation time than the number of layers itself.

7.4.3 CNN Training

Accuracy

The training itself can happen in two different ways. Either the CNN is trained on samples which are already biased by the CNN or on the pure output of the

CNN	Output Time
1	80 ms
2	81 ms
3	136 ms
4	128 ms
5	262 ms
6	501 ms
7	70 ms
8	75 ms

CNN	Output Time
9	20 ms
10	19 ms
11	98 ms
12	16 ms

Table 7.1: Output Time of the CNNs. The different numbers represent different configurations of the CNNs which are shown in Subsection 7.4.1

MCTS only. Figure 7.6 shows how the accuracy of the two different CNNs changes during training. However, the combined approach in the figure does neither use PUCT nor Progressive Bias. The accuracy data was gathered at a point where none of these two methods were introduced yet. At that stage, the combination happened by investigating the most promising move in the root according to the CNN first instead of starting with a random move. For reasons of simplicity the CNN that was trained on raw MCTS data is from here on referred to as "MCTS-Only-trained CNN" while the second CNN is referred to as "Combined-trained CNN". One step on the x-axis represents one training iteration that consists of training on a set of 10,000 samples for 10 epochs. One iteration of training is shown in Figure 7.5. Both of the CNNs were trained for about 80 iterations and the samples did not only include decision for the junctions but also for the corridors and were sampled from a fully observable framework.

The graphs clearly show that training on the raw MCTS data is by far more stable than training on the samples that were generated with a combination of the MCTS and the CNN. However, both CNNs are successfully learning. The MCTS-Only-trained CNN reaches an accuracy of over 90% after only 13 iterations of the training process and stays above 90% from there on. On the other side the Combined-trained CNN needs 40 iterations of training but already falls below 90% only four iterations later. It looks like the Combined-trained CNN is passing different generations during training which makes sense because it is training on data that is influenced by itself. However, the training process becomes much slower and more unstable.

Accuracy

The accuracy over time alone does not tell much about the final performance of a CNN when it is used in the framework. Therefore, these two CNNs are additionally compared by evaluating the score they can achieve. Due to the fact that the decisions in the corridors are by far less complex than the decisions at

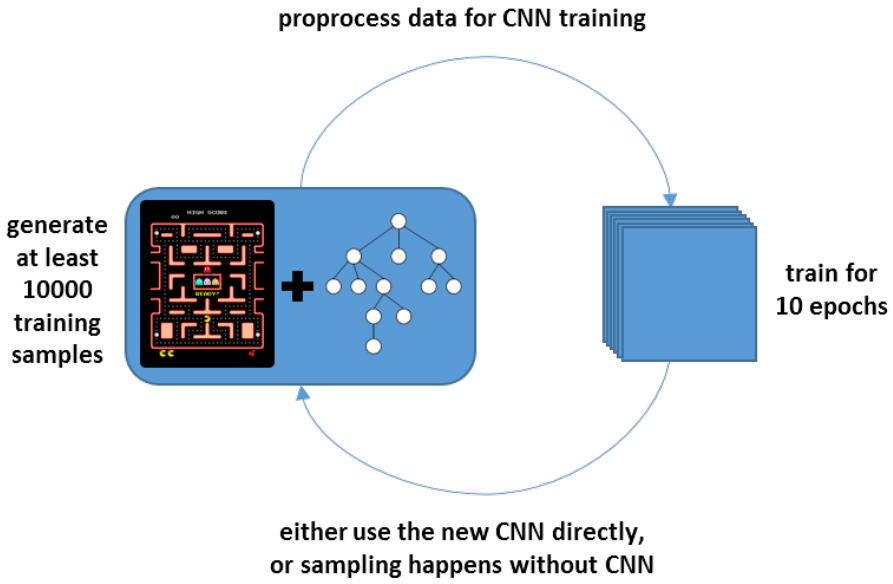


Figure 7.5: Example of one iteration of training

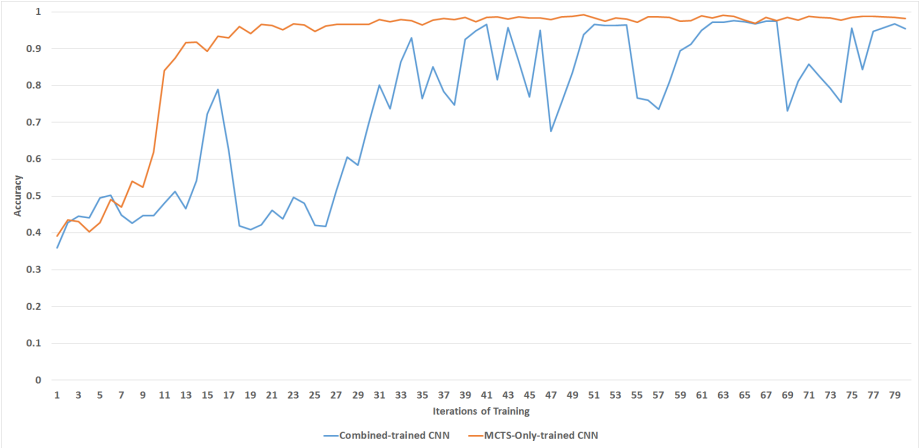


Figure 7.6: Accuracy of the CNN during training. One step on the x-axis corresponds to training the CNN for 10 epochs on at least 10,000 training samples. The blue graph corresponds to the CNN which was trained on data that was generated by an MCTS that is already biased by the CNN. The orange graph corresponds to the CNN which was trained on the pure output of the MCTS only.

the junctions, the CNNs were only trained for the complex decisions. This does not only specialize the CNNs on the most crucial decisions but also stabilizes the learning process because the samples become more consistent. Considering the two different training approaches of an MCTS-Only-trained CNN and a Combined-trained CNN plus the fact that decisions at the junctions can be made by the CNN only or in a combined way as well, leads to the following four configurations:

1. Making decisions by combining the MCTS with the MCTS-Only-trained CNN
2. Making decisions purely based on the output of the MCTS-Only-trained CNN
3. Making decisions by combining the MCTS with the Combined-trained CNN
4. Making decisions purely based on the output of the Combined-trained CNN

The two CNNs were evaluated at a stage where the agent did not make use of the Ghost Model and the Pill Model and the MCTS did not use the optimal settings yet. Therefore, the CNNs can be compared to each other but the scores are in no relation to the scores of other experiments. Moreover, the CNNs were evaluated in a fully observable framework and the combination with the MCTS happened by investigating the most promising move according to the CNN in the root first. During evaluation, it turned out that if the decisions at the junctions are made by the CNN only, the agent does never score more than 280 points. Therefore, only the two approaches in which the decisions are made in a combined way are compared. The evaluations were run in a partially observable framework and the results are shown in Figure 7.7. With a p-value of 0.784 the two different approaches seem to be not significantly different. However, the training process of an MCTS-Only-trained CNN is much faster and more stable. Therefore, this training method was chosen for all later CNNs.

7.4.4 Final CNN Performance

After introducing these partial observability models and adjusting the MCTS settings, a new version a the CNN was trained for multiple weeks until the accuracy stagnated at around 70%. By considering the partial observability models, the decision became more complex.

The performance in a partially observable framework differs from the performance in a fully observable framework. A CNN, that was trained in a fully observable framework might still perform better in a partially observable framework than a network that was especially trained for it. The reason for that is that the fully observable CNN has seen more situations and is, therefore, more robust to new situations in the game. Furthermore, the CNN was originally

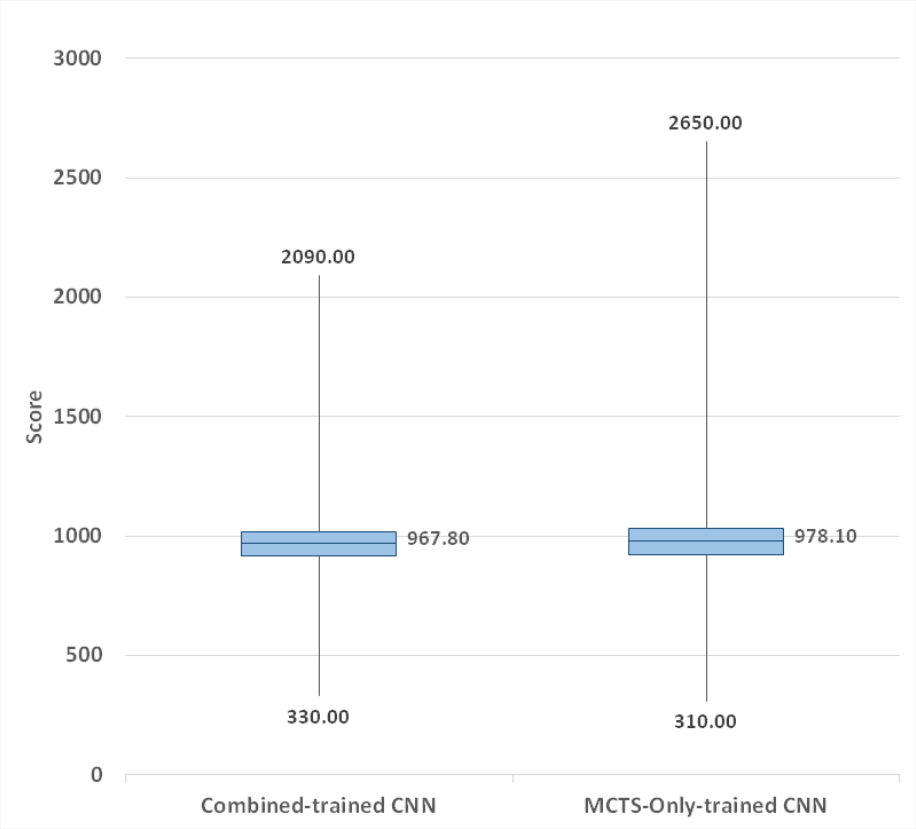


Figure 7.7: Difference in performance of two different CNNs. One CNN was trained on raw MCTS data (MCTS-Only-trained) while the other one was trained on MCTS data that is already biased by a CNN (Combined-trained). The CNN integration itself was done by prioritizing the best move according to the CNN in the root. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

trained to make decisions for the junctions only but it could be that it also outputs reasonably good decisions for the corridors as well. Moreover, there is a chance, that the CNN performs well on its own but becomes worse when it is combined with the MCTS. Taking these three different parameters into account leads to the following eight configurations:

1. Use in a fully observable framework and make decisions at the junctions using the CNN only
2. Use in a fully observable framework and make decisions at the junctions using the CNN and MCTS combined
3. Use in a fully observable framework and make decisions at every location using the CNN only
4. Use in a fully observable framework and make decisions at every location using the CNN and MCTS combined
5. Use in a partially observable framework and make decisions at the junctions using the CNN only
6. Use in a partially observable framework and make decisions at the junctions using the CNN and MCTS combined
7. Use in a partially observable framework and make decisions at every location using the CNN only
8. Use in a partially observable framework and make decisions at every location using the CNN and MCTS combined

For the evaluations, the Progressive Bias approach $UCT + W \times \frac{H_i}{n_i}$ with $W = 1$ was used if a combination of the CNN and MCTS was needed. The results are shown in Figure 7.8, Figure 7.9, Figure 7.10, and Figure 7.11. Although the new CNN could not reach the 90% mark during training, the performance is much better compared to the previous one.

In every case, letting only the CNN make decisions in the corridors results in relatively bad scores. This was to be expected because the CNN was trained on samples at the junctions and not in the corridors. Furthermore, for all of the four different setups, better scores can be achieved by combining the MCTS and the CNN instead of using only the CNN to make decisions. Although the previous experiments showed that the older CNNs could never achieve more than 280 points on their own, the new CNN achieves good results even when it is not combined with the MCTS. In the fully observable framework, the CNN can achieve 13,000 points on its own, which is about 20% of the performance of the agent that only uses the MCTS. In the partially observable framework, it achieves about 4,400 points, which corresponds to 66% of the performance of the agent that only uses the MCTS. The CNN clearly learned to make reasonably good moves on its own and especially in the partially observable framework it

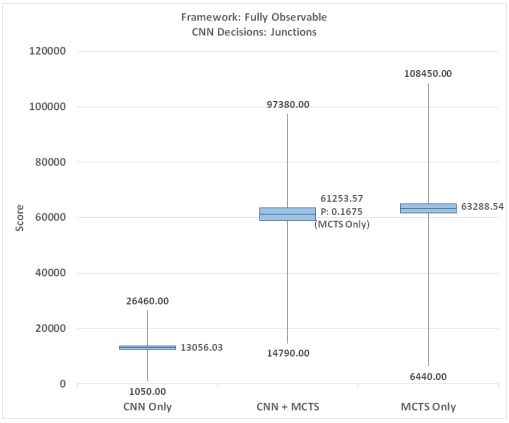


Figure 7.8: Evaluation of the CNN in a fully observable framework while taking the CNN output into account at the junctions only. Additionally, the evaluation of the agent that only uses the MCTS in a fully observable framework was added for comparison reasons. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

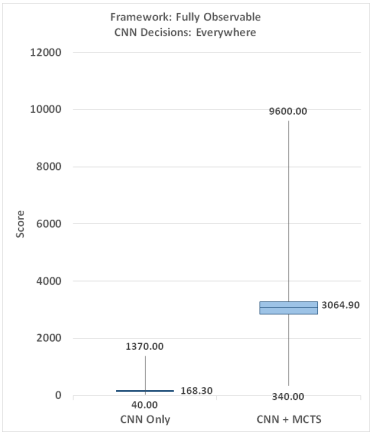


Figure 7.9: Evaluation of the CNN in a fully observable framework while taking the CNN output into account everywhere. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

performs reasonably well. Obviously, comparing the performance of a configuration in a fully observable framework with the same configuration in a partially observable framework, the agent always performs better without the restrictions in observability. For both types of observabilities, the best performing configuration is to use a CNN at the junctions only in combination with the MCTS. However, this configuration still performs a bit worse than the agent that only uses the MCTS for move prediction. Nevertheless, with a p-value of 0.1675 for the fully observable case and a p-value of 0.5128 for the partially observable case, the scores are not significantly worse than the scores of the agent that uses the MCTS only and no CNN at all. It could still be that the contribution of the CNN does helps the MCTS but the time it consumes would better be used to run more simulations for the MCTS. This assumption is checked in the next experiments in Subsection 7.4.5.

7.4.5 MCTS Performance

An agent that uses the combined approach can achieve 6493.23 points in the partially observable framework. An agent that uses an MCTS only achieves 6674.30 points on average. Comparing the two agents (Figure 7.9 and Figure 7.4) does not show a significant difference in performance. It seems that the CNN works reasonably good on its own but cannot enhance the MCTS. The

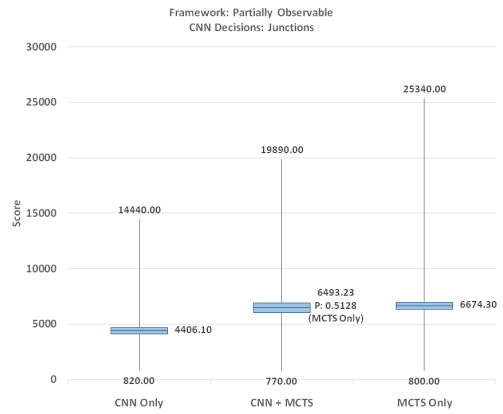


Figure 7.10: Evaluation of the CNN in a partially observable framework while taking the CNN output into account at the junctions only. Additionally, the evaluation of the agent that only uses the MCTS in a partially observable framework was added for comparison reasons. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

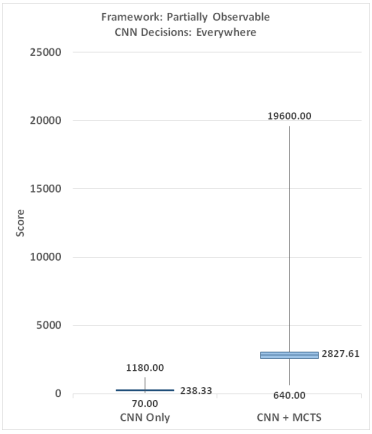


Figure 7.11: Evaluation of the CNN in a partially observable framework while taking the CNN output into account everywhere. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

reason for that might be that the CNN consumes too much time such that the MCTS cannot improve the result enough with the remaining time. To test this assumption, two other experiments were run.

The first experiment tries to find out if the MCTS does not have sufficient time to enhance the output of the CNN even further. Therefore, instead of giving the CNN and MCTS 40ms in total to predict a best move, the time consumption of the CNN is ignored. Consequently, the MCTS has 40ms to predict a move as usual but it is still biased by the CNN output. This corresponds to an agent that uses the MCTS as usual and gets the output of the CNN for free. This agent is referred to as "MCTS + Free CNN". If the score increases, the problem lies in the time the CNN consumes to compute the move probabilities. If the score does not increase, this indicates that the MCTS in its current state has reached its limits and cannot improve any further given the time restriction of 40ms.

The second approach deals with the same question in a different way. Instead of increasing the time for the CNN and MCTS, it is to be determined whether the MCTS can even contribute good results in the remaining time. Therefore an evaluation is run by using the MCTS only but the time limit for the move prediction is reduced by the output time of the CNN. Consequently, the amount the MCTS can contribute to the move prediction after the CNN consumed its 20ms is evaluated. This agent is referred to as "MCTS Less Time". If the score decreases significantly, this means that the MCTS simply has too little time to compute reasonable results. If the score still stays high the MCTS cannot improve further given the time restrictions of 40ms.

Moreover, three more evaluations are added for clarification and comparison reasons. The first one is referred to as "MCTS More Time" and represents the performance of an agent that can use a larger time window than the default 40ms. In exact, the time is increased by 20ms, which corresponds to one CNN output. The second agent corresponds to a default "MCTS Only" agent and has 40ms to predict a move by using only the MCTS. The last agent also uses a free CNN but the time for the MCTS is reduced to 20ms to compare it to the "MCTS Less Time" agent. Consequently, this agent is called "MCTS Less Time + Free CNN". A graphical representation of the timing configuration for each of the agents is shown in Figure 7.12. In the figure, the time-step marking corresponds to the default time-step in which both of the agents have to predict a move. Everything that is not included in that time-step is added for free to the Ms. Pac-Man agent.
Although the results in Figure 7.13 show that the agent with a free CNN can achieve about 70 more points on average compared to the agent that only uses the MCTS, this difference is not significant. The p-value for the two evaluations is 0.7077. However, this could simply be because the MCTS is working at its limit and increasing the time window only adds a minor improvement in performance. To check this assumption, the results need to be compared to

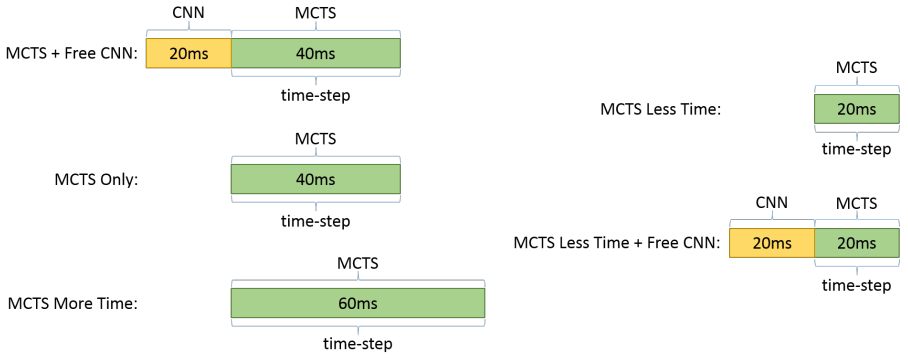


Figure 7.12: Graphical representation of the timing for one move prediction in the different experiments.

the "MCTS More Time" agent. The performance is almost identical, proven by the p-value of 0.8051. This indicates, that the MCTS works near its limits in the partially observable framework given the short time window. Due to diminishing returns, the score does only increase in very small steps although the number of play-outs is strongly increased.

Comparing the agent that only uses an MCTS but has less time for it (MCTS Less Time) with the agent that uses a CNN and the MCTS in the default 40ms time window (MCTS Less Time + Free CNN) makes the enhancement of the CNN clearer. If a free CNN is added to the MCTS agent that has less time, there is an improvement of more than 250 points on average. The p-value of 0.0312 indicates, that improvement of the agent is significant. The only problem is that the CNN consumes too much time to compensate the reduced number of play-outs that are lost due to the CNN output.

Wrapping the two experiments up, the MCTS seems to work near its limit given the short time window. The agent can be slightly improved in average score, but the improvement is not significant. However, the second experiment clearly shows, that the CNN can significantly improve an MCTS that has not reached its maximum capabilities yet.

7.4.6 History Elements

The previous CNNs were trained in a fully observable framework. However, the agent has to perform in a partially observable framework and training a CNN with these restrictions could increase the overall performance. By training in a partially observable framework, it could also be beneficial to add a history to the CNN input. Taking the history as an additional feature into account creates a new way to assume the current positions of the ghosts better. Adding more elements to the history means that more data needs to be processed by

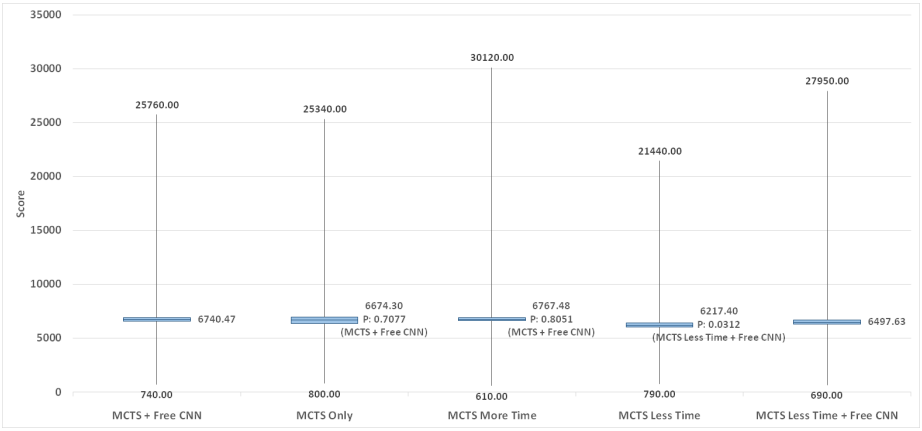


Figure 7.13: Difference in performance of an MCTS that has 40ms to predict a move regardless of the time the CNN consumed before and an MCTS that predicts a move on its own but has reduced a time window for prediction. The MCTS Only, MCTS More Time, and MCTS Less Time + Free CNN evaluations were added for comparison reasons. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

the CNN. Each element adds five more channels to the input. Obviously, this increases the output time of the CNN which is a crucial factor in the real-time framework. Therefore, experiments were run to see the change in output time depending on the number of elements that are added to the history. Figure 7.14 shows the different output times in form of a graph.

With an increasing number of history elements, the output time seems to increase in a linear way. This also makes sense because the convolution happens for each channel separately and no 3D convolution is performed. Given the 40ms time restriction to predict a move, it is possible to add at most six history elements to the CNN input. However, adding one element does not only increase the computational power that is needed but also the disk space that is needed to store each history element. This adds more limitations to the use of a history. To test the difference in performance when using a history, two CNNs were trained using the partially observable framework. One of them does not use a history for the CNN input at all while the other one uses one history element that represents the situations five time-steps ago. Figure 7.15 clearly shows that adding a history element drastically decreases the performance of the agent. Although both CNNs were trained to 70% accuracy where they stagnated, the one without a history performs significantly worse than the CNN that uses a history. However, when comparing the performance to the agent that is using a CNN that was trained in a fully observable framework without a history, the fully observable trained CNN agent performs significantly better with a p-value of 0.0009. Consequently, training a CNN in a fully observable

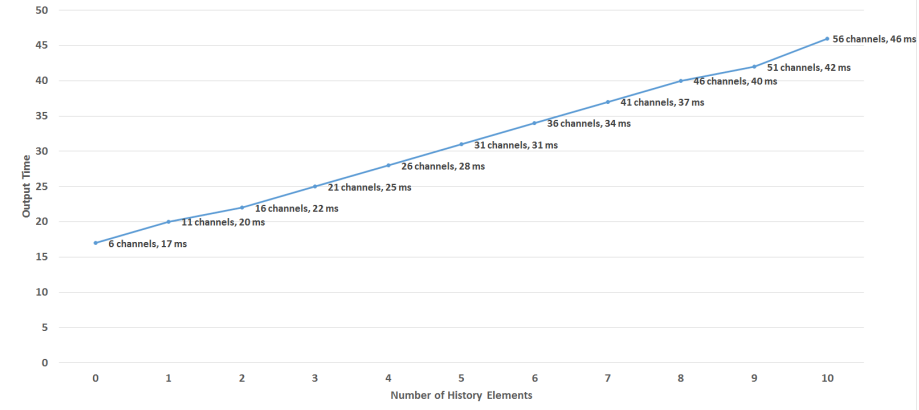


Figure 7.14: Difference in output time in milliseconds if the number of history elements changes. Additionally, the labels show the corresponding number of channels in the input.

framework results in a better performing agent compared to one that uses a CNN that was trained in a partially observable framework. Moreover, adding a history to the input of the CNN decreases the performance of the agent.

7.5 CNN Integration

As discussed in Section 6.4, there exist different ways of combining the CNN and the MCTS. Figures 7.16, 7.17, 7.18, and 7.19 compare the four different equations and shows their performance while using different weights for combination.

Although the different equations differ much and influence the MCTS in different ways, the average scores stay on the same level. Even when using different weights to adjust the influence of the CNN, the score does not become significantly different. The MCTS seems to have enough time to lead the prediction in the same direction every time regardless of the way it is influenced by the CNN.

7.6 General Performance

All of the previous experiments were run in games where Ms. Pac-Man was playing against one type of ghost agents. These agents were explicitly created to work in a partially observable framework and are able to communicate with each other. However, the framework also provides the controllers for six more different ghost agents. To determine whether the CNN is not overfitted to the ghost agents that were used for training, evaluations against all of the other ghost agents were run. Figure 7.20 shows the results of the agent that uses a

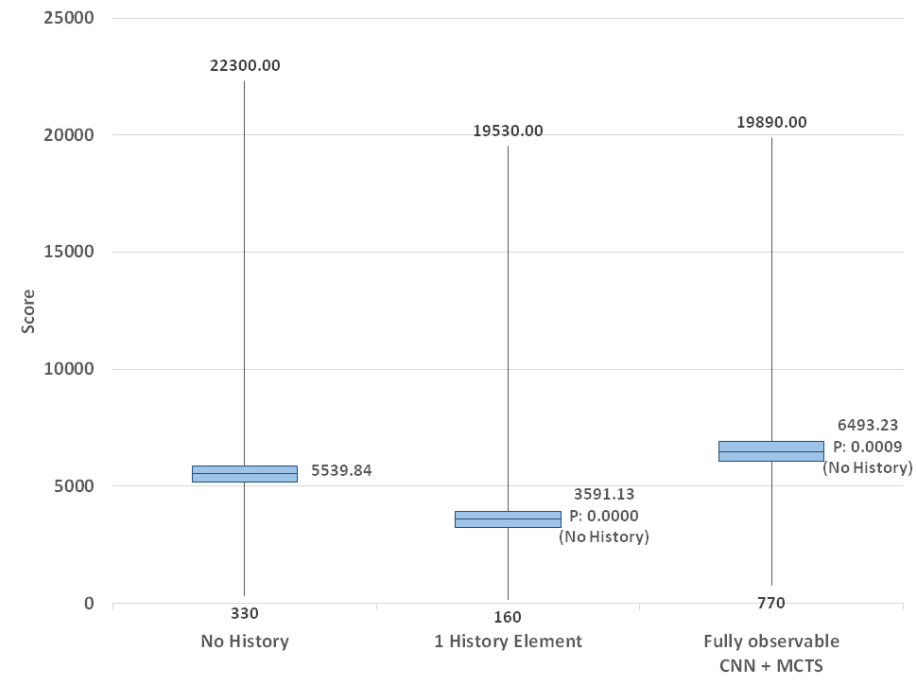


Figure 7.15: Difference in performance depending on the history of a CNN that was trained in a partially observable framework. Additionally, the evaluation of the agent that uses a CNN that was trained in a fully observable framework was added for comparison reasons. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

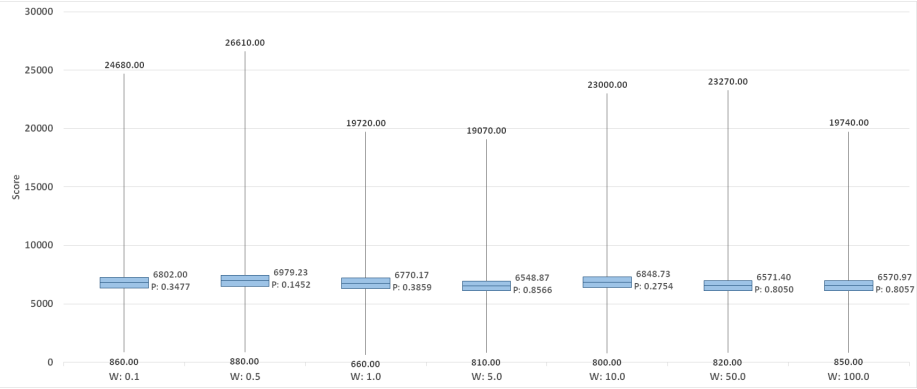


Figure 7.16: Evaluation of the CNN combination with different weights when using Equation 6.6. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

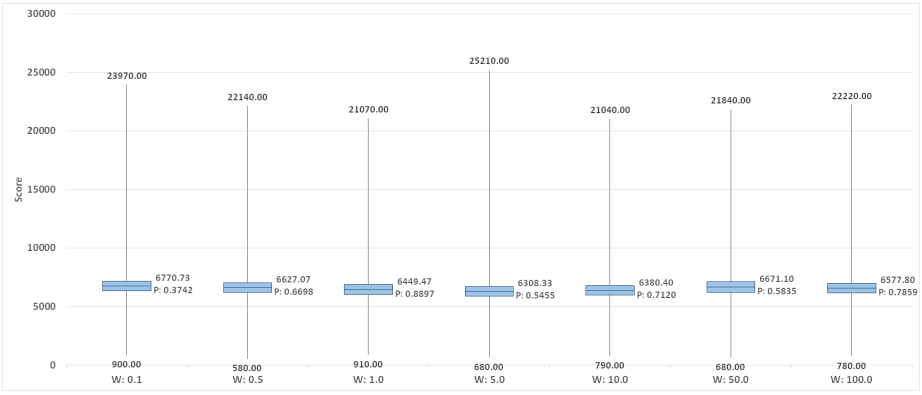


Figure 7.17: Evaluation of the CNN combination with different weights when using Equation 6.7. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

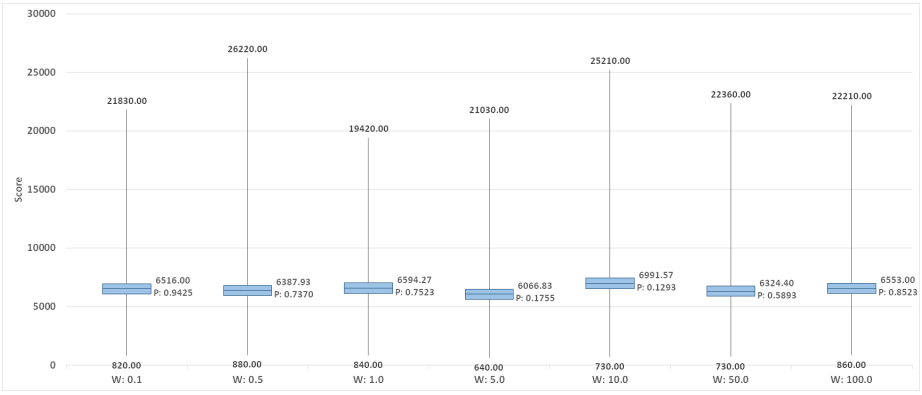


Figure 7.18: Evaluation of the CNN combination with different weights when using Equation 6.8. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

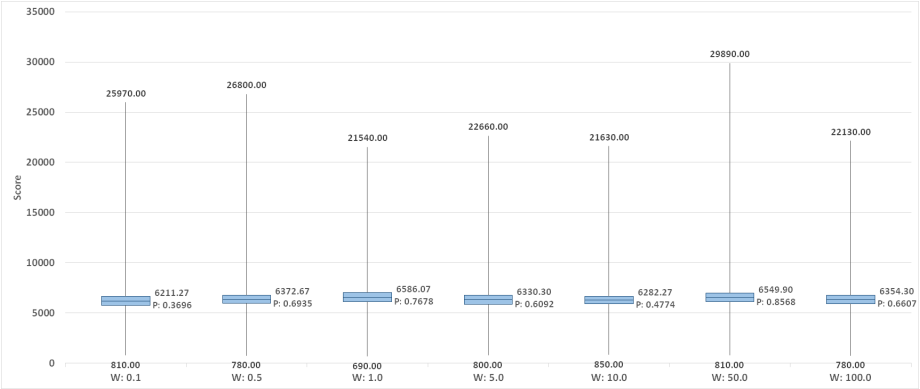


Figure 7.19: Evaluation of the CNN combination with different weights when using Equation 6.8. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

CNN and the MCTS and the results of the agent that only uses the MCTS. Although the CNN was only trained against one ghost agent team, the final agent performs well against the other agents as well. The agent that uses the combination of a CNN and the MCTS usually performs on the same level as the agent that only uses the MCTS. Depending on the enemy ghost team, it can sometimes even outperform an agent that only uses an MCTS. Only against the Legacy2TheReckoning ghost team, the combined agent performs significantly worse than the pure MCTS agent. Overall, the final agent shows a good performance regardless to the enemy it is facing.

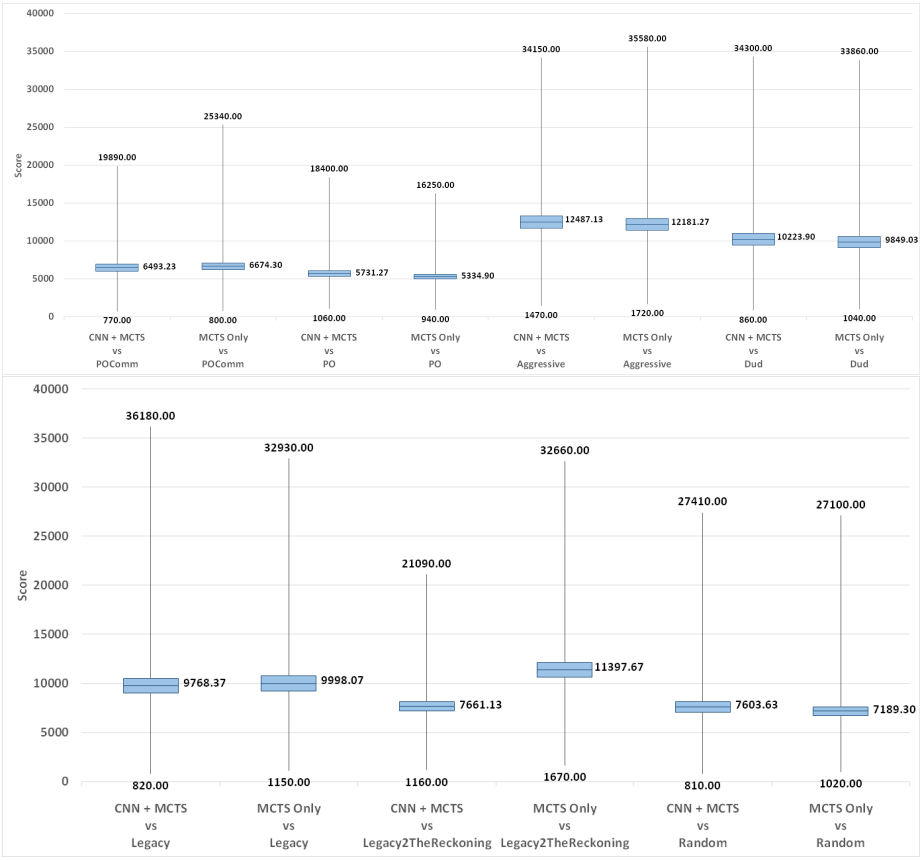


Figure 7.20: Comparison of performance against the different ghost agents that are provided by the Ms. Pac-Man framework. The numbers show the minimum, maximum, and average score. The boxes show the 95% confidence intervals.

Chapter 8

Conclusion and Future Work

This chapter provides a conclusion of the thesis. The general problem statement and the research questions that were presented in Chapter 1 are revisited in Section 8.1 and Section 8.2 by taking the results into account. Finally, examples of future work in form of improvements and new ideas are given in Section 8.3.

8.1 Research Questions

This section lists the three research questions and provides an answer to each of them based on the results of this thesis.

1. *Can the* AlphaGo *approach be adapted to work for real-time video games?*

The original AlphaGo approach needed to be adapted slightly to work in a real-time video game setup. The original approach trained the CNN after every prediction such that the newly learned information could instantly be used for the next move prediction. Training a CNN in a single-threaded real-time environment consumed too much time to ensure a good prediction. Therefore, the training process was outsourced to create an iterative learning process. First, the game is played in a usual way and at each time-step, a snapshot of the current state and the predicted move is taken. After generating a sufficient number of samples, the CNN is trained on them. This process is repeated until the CNN either reaches a predefined accuracy or does not improve anymore. Combining the CNN with the MCTS to generate the training samples did not decrease the final performance of the CNN but slowed down the training process and made it more unstable. Another important factor is that the CNN cannot be used in every node of the tree. One CNN output consumes almost half of the time that the agent has to return a move. To still ensure a high impact on the MCTS, the tree can only be biased in the root node. Consequently, the AlphaGo approach could only be adapted to work for real-time video games after outsourcing the training process out of the real-time move prediction and adjusting the way the CNN influences the MCTS.

2. *Which different approaches can be used to combine Convolutional Neural Networks and Monte Carlo Tree Search?*

The methods to bias an MCTS with the output of a CNN proposed in this thesis are "Progressive Bias" and "Polynomial Upper Confidence Trees". Both of these approaches can successfully bias the MCTS and do increase the score slightly. Although the two methods work in a different way, the results are similar because the MCTS has enough time to run many simulations and adjust the prediction accordingly. Moreover, changing the weight to adjust the contribution of the bias also does not change the result significantly.

3. *What is the best architecture of a Convolutional Neural Network given the real-time Arcade Game Ms. Pac-Man environment?*

The architecture highly depends on the real-time constraint. By having only 40ms to predict a move and still leave time for the MCTS, it is mandatory to keep the computations as simple as possible. Decreasing the size of the data that traverses through the network reduces the output time significantly. Moreover, the number of layers itself does not have a high impact on the output time but the type of layers does. Adding Max Pooling Layers and reducing Zero Padding as much as possible results in topologies that can be applied in a real-time framework. Taking these factors into account while still ensuring a CNN that is able to learn leads to the following configuration using the notation proposed in Section 7.4:

> Z-2 → C-32-3-1 → M-2-2 → Z-2 → C-64-3-1 → M-2-2 → Z-2 → C-64-3-1 → M-2-2 → Z-2 → C-64-3-1 → M-2-2 → F-256

Training the CNN on the raw output of the MCTS results in a more stable learning process and a steeper accuracy curve then training on samples that are already biased by the CNN. Furthermore, a CNN that was trained on samples that were generated in a fully observable framework does perform better than a CNN that was trained on samples from a partially observable framework. Moreover, the experiments show that the agent performs better if the CNN only supports the MCTS at the junctions. The decisions at the corridors are relatively simple and letting a CNN, that was trained on junction data, do a move prediction for the corridors results in unnecessary time consumption and performance decrease. Finally, a combination of the MCTS and the CNN always leads to better results than using solely the CNN to predict a best move.

The remaining configuration and hyperparameters of the best performing CNN are as follows:

Activation Function	RELU
Activation Function Output Layer	Softmax
Regularization Value	$5e^{-4}$
Learning Rate	$1e^{-2}$
Updater	Nestrovs Updater
Momentum	0.9
Optimization Function	Stochastic Gradient Descend
Loss Function Output Layer	Multiclass Cross Entropy
Weight Initializer	Xavier Weight Initializer

8.2 Problem Statement

The general problem statement of the thesis was defined as follows:

> *Enhancing Monte Carlo Tree Search by using Deep Learning techniques in the context of real-time video games.*

Although the MCTS can in principle be enhanced by a CNN to achieve a higher average score with a confidence of 95%, the improvement is only beneficial if the CNN would be for free, i.e. consume no time. The time consumption of the CNN reduces the number of play-outs so much that the CNN cannot compensate for it. Therefore, decreasing the time the CNN needs to predict a move could make an enhancement of the MCTS possible.

In this thesis, a CNN was successfully trained to learn a good move prediction for the real-time video game Ms. Pac-Man. The output of this CNN is used to bias the selection in the root node of the tree. The CNN that was used for the final agent was only trained against one specific enemy but also performs well against other enemies. The partial observability added another constraint which needed to be handled. The CNN can successfully be used in a fully observable framework as well as in a partially observable framework. Finally, it can be said that applying Deep Learning techniques in real-time video games is theoretically possible but the real-time constraint makes it hard to use time-consuming techniques. Especially CNNs need much time and depending on the given time limit they are too computationally expensive. The provided framework added the restriction to work single-threaded so removing this restriction might add other possibilities to enhance an MCTS.

8.3 Future Work

Although the CNN was able to improve the MCTS, there is still room for more improvements. Especially the partial observability is causing unexpected behavior of Ms. Pac-Man in some situations.

The main problem of an agent that uses a combination of a CNN and the

MCTS is that the CNN consumes too much time and reduces the number of play-outs too much. As explained in Section 7.1 the data pipeline is too slow such that a computation on the GPUs is not worth it. By minimizing the time consumption of the data transfer, the computation time could be decreased and the CNN could compute the output faster. Furthermore, a CNN with a different topology could be used that can compute the move probabilities faster. Nevertheless, the CNN would still need to guarantee a reasonably good performance, which can be hard if the output time is reduced even further.

Another aspect that can be improved is the implementation of the history. Considering the results of the experiments, adding a history to the CNN did decrease the performance of the agent. Instead of adding the history as new channels to the input of the CNN, so-called "Long short-term memory" (LSTM) nodes (Hochreiter and Schmidhuber, 1997) could be used in the CNN to make it consider the history of the current game state. These nodes can store the data from previous iterations and use it for the later ones. However, this also means that the complete training process has to be reworked. Instead of shuffling all samples and generating a training and testing set, the order needs to be maintained to ensure a chronological order. Furthermore, by starting a new game, the memory in the nodes also would need to be cleared every time because the new game does not have a history yet.

DeepMind was using Reinforcement Learning strategies to train their agents. By getting a feedback instantly after a game is over, the agents could use that information for the next games. The PacMaas agent did not make use of Reinforcement Learning at all and implementing an algorithm for that, could enhance the learning process and the final performance.

Moreover, not only the CNN but also the MCTS can be adjusted. As described in Section 6.1, the MCTS was originally built for a fully observable framework. Although it was adapted to work in a partially observable framework by adding enhancements like partial observability models, the main part was left mainly untouched. There exist enhancements, that can help Ms. Pac-Man in a fully observable framework but might harm Ms. Pac-Man in a fully observable framework. Reworking enhancements like the luring behavior and similar ones could, therefore, increase the performance of the agent.

Furthermore, the Ghost Model currently works by assuming the positions of ghosts that cannot be seen. However, this often leads to a scenario where Ms. Pac-Man reaches a position in which the model assumed a ghost but in reality there is none. Consequently, the assumed position of the corresponding ghost needs to be shifted to a new location. Currently, the position is set to the lair and the ghost stays there until it is spotted again. Although this avoids impossible situations, all information about the corresponding ghost is lost. Adjusting this behavior such that the ghost is shifted to another position in the maze keeps up the thread of Ms. Pac-Man getting caught. Handling these situ-

ations differently can, therefore, help the model to play safer and survive longer.

Another improvement could be the distinction between "known" and "assumed" ghosts. Neither the Belief Game nor the CNN does distinguish between ghosts of which the location is known exactly and ghosts of which the location can only be assumed. However, handling both of these types differently could help Ms. Pac-Man to escape from dangerous situations. For example, if she is chased by two ghosts and an "assumed" ghost is blocking the last corridor to escape, she eats as many pills as she can before losing a live. According to the Ghost Model, every corridor leads to a loss. However, choosing the corridor with the "assumed" ghost could still result in a possible escape. Moreover, the CNN could use this new feature to help the MCTS choosing exactly these corridors.

Currently, the best working Ghost Model only follows one strategy in which the ghosts are trying to create a pincer situation. This leads to a monotone behavior of the ghosts, which the CNN adapts. Consequently, it performs worse in situations where the ghosts are not performing a Pincer Move. By reworking the Ghost Model and adding more features to it, the CNN will become more robust to different tactics and situations.

In the experiments, the combination of the CNN and the MCTS only happened in the root node of the tree search. If the output time of the CNN could be reduced the CNN could theoretically be applied to more nodes than only the root. Unfortunately, the provided Ms. Pac-Man framework did not allow to run that experiment. The duration of one time-step needs to be configured before starting a game and it will be the same for every time-step in the game. However, the number of play-outs can differ for every step and are not previously known. Therefore, the framework has to be changed to support this experiment. Furthermore, this would make the use of LSTMs even more complex. In that case, the CNN needs to be cloned for every branch in the tree to ensure a correct history every time.

Finally, instead of using a CNN to predict moves, the CNN could also be used to predict the positions of the ghosts. This is a completely different way to deal with the partial observability and might improve the performance of the MCTS in a partially observable framework.

Bibliography

[1] Alhejali, Atif M and Lucas, Simon M (2010). Evolving Diverse Ms. PacMan Playing Agents Using Genetic Programming. *UK Workshop on Computational Intelligence (UKCI 2010)*, pp. 1–6, IEEE. [12]

[2] Alhejali, Atif M and Lucas, Simon M (2013). Using Genetic Programming to Evolve Heuristics for a Monte Carlo Tree Search Ms Pac-Man Agen. *IEEE Conference on Computational Intelligence in Games (CIG 2013)*, pp. 1–8, IEEE. [12]

[3] Auer, Peter, Cesa-Bianchi, Nicolo, and Fischer, Paul (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, Nos. 2–3, pp. 235–256. [14]

[4] Baudiš, Petr and Gailly, Jean-loup (2011). Pachi: State of the Art Open Source Go Program. *Advances in Computer Games (ACG 2011)*, pp. 24–38, Springer. [25]

[5] Baxter, Jonathan, Tridgell, Andrew, and Weaver, Lex (1998). Experiments in Parameter Learning Using Temporal Differences. *International Computer Chess Association Journal*, Vol. 21, No. 2, pp. 84–99. [2]

[6] Birch, Chad (2010). Understanding Pac-Man Ghost Behavior. *Game Internals.(online) Disponível em< http://gameinternals.com/post/2072558330/understandingpac-man-ghost-behavior>. Acesso em*, Vol. 5, p. 2010. [6]

[7] Björnsson, Yngvi and Marsland, Tony A (2001). Learning Search Control in Adversary Games. *Advances in Computer Games (ACG 2001)*, Vol. 9, pp. 157–174. [2]

[8] Browne, Cameron B, Powley, Edward, Whitehouse, Daniel, Lucas, Simon M, Cowling, Peter I, Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG 2012)*, Vol. 4, No. 1, pp. 1–43. [14, 23]

[9] Burrow, Peter and Lucas, Simon M (2009). Evolution versus Temporal Difference Learning for learning to play Ms. Pac-Man. *IEEE Symposium on Computational Intelligence and Games (CIG 2009)*, pp. 53–60, IEEE. [11]

[10] Campbell, Murray, Hoane Jr, A Joseph, and Hsu, Feng-hsiung (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 57–83. [1]

[11] Chaslot, Guillaume M JB, Winands, Mark HM, Herik, H Jaap van den, Uiterwijk, Jos WHM, and Bouzy, Bruno (2007). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [14, 16, 46]

[12] Chaslot, Guillaume MJ-B, Winands, Mark HM, Szita, Istvan, and Herik, H Jaap van den (2008). Cross-Entropy for Monte-Carlo Tree Search. *Icga Journal*, Vol. 31, No. 3, pp. 145–156. [2]

[13] Coulom, Rémi (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *International Conference on Computers and Games (CG 2006)*, pp. 72–83, Springer. [1, 14]

[14] Coulom, Rémi (2008). Whole-History Rating: A Bayesian Rating System for Players of Time-Varying Strength. *International Conference on Computers and Games (CG 2008)*, pp. 113–124, Springer. [27]

[15] Flensbak, Jonas and Yannakakis, Georgios N. (2008). Ms. Pacman AI controller. [12]

[16] Foderaro, Greg, Swingler, Ashleigh, and Ferrari, Silvia (2012). A model-based cell decomposition approach to on-line pursuit-evasion path planning and the video game ms. pac-man. *IEEE Conference on Computational Intelligence and Games (CIG 2012)*, pp. 281–287, IEEE. [12]

[17] Giusti, Alessandro, Ciresan, Dan C, Masci, Jonathan, Gambardella, Luca M, and Schmidhuber, Jurgen (2013). Fast Image Scanning with Deep Max-Pooling Convolutional Neural Networks. *20th IEEE International Conference on Image Processing (ICIP 2013)*, pp. 4034–4038, IEEE. [21]

[18] Glickman, Mark E (2013). Example of the Glicko-2 System. *Massachusetts, Boston. Boston University, Department of Health Policy and Management.* [11]

[19] Glorot, Xavier, Bordes, Antoine, and Bengio, Yoshua (2011). Deep Sparse Rectifier Neural Networks. *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*, pp. 315–323. [20]

[20] Graf, Tobias and Platzner, Marco (2016). Using Deep Convolutional Neural Networks in Monte Carlo Tree Search. *International Conference on Computers and Games*, pp. 11–21, Springer. [28, 46]

[21] Guo, Xiaoxiao, Singh, Satinder, Lee, Honglak, Lewis, Richard L, and Wang, Xiaoshi (2014). Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. *Advances in Neural Information Processing Systems (NIPS 2014)*, pp. 3338–3346. [23, 27]

[22] Guo, Xiaoxiao, Singh, Satinder, Lewis, Richard, and Lee, Honglak (2016). Deep Learning for Reward Design to Improve Monte Carlo Tree Search in ATARI Games. *International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pp. 1519–1525. [27]

[23] Haykin, Simon S. (1999). Neural Networks: A Comprehensive Foundation. ISBN 978–8–120–32373–5. [18]

[24] Hershey, Shawn, Chaudhuri, Sourish, Ellis, Daniel PW, Gemmeke, Jort F, Jansen, Aren, Moore, R Channing, Plakal, Manoj, Platt, Devin, Saurous, Rif A, Seybold, Bryan, et al. (2017). CNN Architectures for large-scale Audio Classification. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2017)*, pp. 131–135, IEEE. [19]

[25] Hinton, Geoffrey E, Srivastava, Nitish, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan R (2012). Improving Neural Networks by preventing Co-Adaptation of Feature Detectors. *arXiv preprint arXiv:1207.0580*. [22]

[26] Hochreiter, Sepp and Schmidhuber, Jürgen (1997). Long short-term memory. *Neural computation*, Vol. 9, No. 8, pp. 1735–1780. [75]

[27] Ikehata, Nozomu and Ito, Takeshi (2011). Monte-Carlo tree search in Ms. Pac-Man. *IEEE Conference on Computational Intelligence and Games (CIG 2011)*, pp. 39–46, IEEE. [31]

[28] Knuth, Donald E and Moore, Ronald W (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [1]

[29] Kocsis, Levente and Szepesvári, Csaba (2006). Bandit Based Monte-Carlo Planning. *European Conference on Machine Learning*, pp. 282–293, Springer. [1, 14, 15]

[30] Kocsis, Levente, Szepesvári, Csaba, and Winands, Mark HM (2005). RSPSA: Enhanced Parameter Optimisation in Games. *Advances in Computer Games (ACG 2005)*, pp. 39–56, Springer. [2]

[31] Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E (2012). Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, pp. 1097–1105. [3, 18]

[32] Liberatore, Federico, Mora, Antonio M, Castillo, Pedro A, and Guervós, Juan Julián Merelo (2014). Evolving evil: optimizing flocking strategies through genetic algorithms for the ghost team in the game of Ms. Pac-Man. *European Conference on the Applications of Evolutionary Computation*, pp. 313–324, Springer. [13]

[33] Lucas, Simon M (2005). Evolving a Neural Network Location Evaluator to Play Ms. Pac-Man. *IEEE Symposium on Computational Intelligence and Games (CIG 2005)*, pp. 203–210. [11]

[34] Lucas, Simon M (2007). Ms Pac-Man Competition. *ACM SIGEVOlution*, Vol. 2, No. 4, pp. 37–38. [10]

[35] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. (2015). Human Level Control Through Deep Reinforcement Learning. *Nature*, Vol. 518, No. 7540, pp. 529–533. [27]

[36] Nguyen, Kien Quang and Thawonmas, Ruck (2011). Applying Monte-Carlo Tree Search To Collaboratively Controlling of a Ghost Team in Ms PacMan. *IEEE International Games Innovation Conference (IGIC 2011)*, pp. 8–11, IEEE. [12]

[37] Nijssen, Pim and Winands, Mark HM (2012). Monte-Carlo Tree Search for the Hide-and-Seek Game Scotland Yard. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG 2012)*, Vol. 4, No. 4, pp. 282–294. [37]

[38] Park, Sungheon and Kwak, Nojun (2016). Analysis on the Dropout Effect in Convolutional Neural Networks. *Asian Conference on Computer Vision*, pp. 189–204, Springer. [22]

[39] Pepels, Tom and Winands, Mark HM (2012). Enhancements for Monte-Carlo Tree Search in Ms Pac-Man. *IEEE Conference on Computational Intelligence and Games (CIG 2012)*, pp. 265–272, IEEE. [30]

[40] Pepels, Tom, Winands, Mark HM, and Lanctot, Marc (2014). Real-Time Monte Carlo Tree Search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG 2014)*, Vol. 6, No. 3, pp. 245–257. [III, 10, 12, 30, 31, 32, 33, 34, 35, 37, 44]

[41] Pittman, Jamey (2009, accessed February 20, 2018). The Pac-Man Dossier. https://deeplearning4j.org/convolutionalnetwork. [6, 7]

[42] Robles, David and Lucas, Simon M (2009). A Simple Tree Search Method For Playing Ms. Pac-Man. *IEEE Symposium on Computational Intelligence and Games (CIG 2009)*, pp. 249–255, IEEE. [12]

[43] Rohlfshagen, Philipp and Lucas, Simon M (2011). Ms Pac-Man versus Ghost Team CEC 2011 Competition. *IEEE Congress on Evolutionary Computation (CEC 2011)*, pp. 70–77, IEEE. [10]

[44] Rohlfshagen, Philipp, Liu, Jialin, Perez-Liebana, Diego, and Lucas, Simon M (2017). Pac-Man Conquers Academia: Two Decades of Research Using a Classic Arcade Game. *IEEE Transactions on Games (TG 2017)*. [8, 10]

[45] Rosin, Christopher D (2011). Multi-armed Bandits with Episode Context. *Annals of Mathematics and Artificial Intelligence*, Vol. 61, No. 3, pp. 203–230. [46]

[46] Samothrakis, Spyridon, Robles, David, and Lucas, Simon (2011). Fast Approximate Max-n Monte Carlo Tree Search for Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG 2011)*, Vol. 3, No. 2, pp. 142–154. [12]

[47] Schaeffer, Jonathan, Hlynka, Markian, and Jussila, Vili (2001). Temporal Difference Learning Applied to a High-Performance Game-Playing Program. *Proceedings of the 17th international joint conference on Artificial intelligence-Volume 1*, pp. 529–534, Morgan Kaufmann Publishers Inc. [2]

[48] Schalkoff, Robert J (1997). *Artificial neural networks*, Vol. 1. McGraw-Hill, New York. ISBN 978–0–070–57118–1. [18]

[49] Silver, David, Huang, Aja, Maddison, Chris J, Guez, Arthur, Sifre, Laurent, Van Den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, et al. (2016). Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, Vol. 529, No. 7587, pp. 484–489. [2, 24, 25, 26]

[50] Silver, David, Hubert, Thomas, Schrittwieser, Julian, Antonoglou, Ioannis, Lai, Matthew, Guez, Arthur, Lanctot, Marc, Sifre, Laurent, Kumaran, Dharshan, Graepel, Thore, et al. (2017a). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv preprint arXiv:1712.01815*. [3, 27]

[51] Silver, David, Schrittwieser, Julian, Simonyan, Karen, Antonoglou, Ioannis, Huang, Aja, Guez, Arthur, Hubert, Thomas, Baker, Lucas, Lai, Matthew, Bolton, Adrian, et al. (2017b). Mastering the Game of Go without Human Knowledge. *Nature*, Vol. 550, No. 7676, p. 354. [2, 23, 26]

[52] Skymind (2017, accessed February 26, 2018). A Beginner's Guide to Deep Convolutional Neural Networks (CNNs). https://deeplearning4j.org/convolutionalnetwork. [III, 19]

[53] Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research*, Vol. 15, No. 1, pp. 1929–1958. [21]

[54] Tesauro, Gerald (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, Vol. 38, No. 3, pp. 58–68. [2]

[55] Uriarte, Alberto and Ontañón, Santiago (2017). Single Believe State Generation for Partially Observable Real-Time Strategy Games. *IEEE Conference on Computational Intelligence and Games (CIG 2017)*, pp. 296–303, IEEE. [37]

[56] Weiss, Brett (2012). *Classic Home Video Games, 1985–1988: A Complete Reference Guide*. McFarland. ISBN 978–1–476–60141–0. [8]

[57] Whitehouse, Daniel, Powley, Edward Jack, and Cowling, Peter I (2011). Determinization and Information Set Monte Carlo Tree Search for the Card Game Dou Di Zhu. *IEEE Conference on Computational Intelligence and Games (CIG 2011)*, pp. 87–94, IEEE. [38]

[58] Williams, Piers R, Perez-Liebana, Diego, and Lucas, Simon M (2016). Ms. Pac-Man Versus Ghost Team CIG 2016 Competition. *IEEE Conference on Computational Intelligence and Games (CIG 2016)*, pp. 1–8, IEEE. [3, 10, 11]

[59] Winands, Mark HM (2017). Monte-Carlo Tree Search in Board Games. *Handbook of Digital Games and Entertainment Technologies* (eds. Ryohei Nakatsu, Matthias Rauterberg, and Paolo Ciancarini), pp. 47–76. Springer. [III, 15]

[60] Winands, Mark HM, Kocsis, Levente, Uiterwijk, JWHM, and Herik, H Jaap van den (2002). Temporal Difference Learning and the Neural MoveMap Heuristic in the Game of Lines of Action. *GAME-ON*, pp. 99–103. [2]

[61] Wirth, Nathan and Gallagher, Marcus (2008). An Influence Map Model forPlaying Ms. Pac-Man. *IEEE Symposium on Computational Intelligence and Games (CIG 2008)*, pp. 228–233, IEEE. [12]

[62] Wittkamp, Mark, Barone, Luigi, and Hingston, Philip (2008). Using NEAT for continuous adaptation and teamwork formation in Pacman. *IEEE Symposium on Computational Intelligence and Games (CIG 2008)*, pp. 234–242, IEEE. [12]