TÉCNICO LISBOA

# Learning Agent in the Ms. Pac-Man vs Ghosts game

**João Mário Pereira Guerreiro**

Thesis to obtain the Master of Science Degree in

**Information Systems and Computer Engineering**

Supervisor: Prof. José Alberto Rodrigues Pereira Sardinha

**Examination Committee**

Chairperson: Prof. Rui Filipe Fernandes Prada
Supervisor: Prof. José Alberto Rodrigues Pereira Sardinha
Member of the Committee: Prof. Pedro Alexandre Simões dos Santos

**January 2021**

# Acknowledgments

Thank you all. You know who you are.

# Abstract

Due to the success of the Monte Carlo Tree Search algorithm in several games, surged the idea to apply this method to the Ms. Pac-Man game. There was, already, a competition for agents playing Ms. Pac-Man. The first agents were rule-based, until this idea to use MCTS appeared. The idea for this thesis consist on using the MCTS algorithm, as an auxiliar agent that will play the game without the time restrictions and create a dataset a priori, to then train a neural network that will play the game in real time. These results were then compared to an agent, who came in second in the worldwide competition and first in the european competition. The results achieved were encouraging, with the MCTS agent achieving an average score of 2749 points after thirty games, comparing to the 2871 achieved by the agent used as benchmark. The moves chosen by the MCTS agent, in the thirty games, were then saved to a file and used to train two neural networks, one through classification using as labels the action chosen, and another by regression, using the values of each action for each game state. The best results achieved by both neural networks were 2103 points and 1437 points, respectively. This can, probably, be explained due to the low number of samples combined with a vast number of features in the dataset.

# Keywords

Ms. Pac-Man, Monte Carlo Tree Search, Neural Networks

# Resumo

Devido ao sucesso que o algoritmo de Pesquisa em Árvore Monte Carlo teve noutros jogos, surgiu a ideia de aplicar este algoritmo também ao jogo do Ms. Pac-Man. Já existia, antes, uma competição para agentes jogarem Ms. Pac-Man, mas a maioria usavam técnicas baseadas em regras. A ideia para esta tese, consiste em usar o MCTS, não em tempo real, mas sim como o agente auxiliar que irá jogar o jogo com um tempo limite maior, criando assim um dataset a priori, que depois será usado para treinar uma rede neuronal, esta sim, que irá jogar o jogo em tempo real. Os resultados foram depois comparados a um dos melhores agentes desta competição, ficou em segundo lugar mundialmente e em primeiro a nível europeu. Os resultados obtidos foram encorajadores, com o agente auxiliar de MCTS a alcançar uma pontuação média de 2749 em trinta jogos, comparando aos 2871 pontos alcançados pelo agente usado para comparação. Durante os trinta jogos, as acções escolhidas em cada estado foram guardados, assim como o estado. Este conjunto de treino, foi então usado para treinar duas redes neuronais. Uma através de classificação, com as classes para cada estado a serem as acções, e uma rede neuronal treinada por regressão, onde as classes eram os valores de todas as acções para cada estado. Os melhores resultados alcançados por cada rede neuronal, foram 2103 e 1437 pontos, respectivamente.

# Palavras Chave

Ms. Pac-Man, Monte Carlo Tree Search, Redes Neuronais

# Contents

# List of Figures

# List of Tables

# Listings

# 1

# Introduction

## Contents

Since the early days of Artificial Intelligence (AI) until today, AI has evolved a lot. The games started by using state-machines with simple reasoning behind it, all the way up to the famous neural networks. AI started to appear in every kind of algorithms for multiple purposes, such as finances, medicine, aviation.

Specifying to the scope of this thesis, video-games, also have an use for AI, mainly to control the actions of non-player characters (NPC's). Let us take for instance Pac-Man, AI controls the ghosts and a human plays with Pac-Man.

The original Pac-Man was the first game to possess what could be called real AI, it was very rudimentary, a simple state machine, as said by Millington [16], that controlled the ghosts. This was fine, but after a while people started seeing patterns in the ghosts' movement, the movement was predictable. In Ms Pac-Man, however, the movement was semi-random, making this game much more challenging from an AI standpoint.

This game became so intriguing for AI enthusiasts and scholars that the research community created a competition. In 2007, the first tournament was organized by Lucas [14], but it had restrictions. The agent could only gather information about the game in a form of screen-capture, having no access to the game-state itself. In 2011, the organizers created a second tournament format. In this new format, there were two types of agents, the competitors could enter with a controller for Ms Pac-Man that had the purpose of maximize the points, or enter with a controller for the ghosts that had the objective of doing the exact opposite, minimize the points of Ms Pac-Man.

While in the first iterations, the competition was dominated by teams with ruled-based algorithms, the success of Monte-Carlo Tree Search algorithms in other games like Go as explained by Gelly and Silver [8], made this an attractive alternative to use in Ms Pac-Man, this was justified by being the algorithm behind the winning controller on the 2012 competition.

However, Monte-Carlo Tree Search has limitations when executed in real-time, mainly in Ms. Pac-Man, where an agent has to decide an action to take in 40 milliseconds. This reduced time, limits the amount of simulations that the algorithm can do. Due to that, multiple agents implemented with a MCTS in real-time have modifications made to the algorithm, with the purpose of increasing its performance. However, more recently in 2014, appeared a technique that uses the MCTS as an auxiliary tool, to create data to train a neural network, developed by Guo et al. [9]. This way it is possible to give the MCTS the time it needs to do a proper assessment of the game state and produce the best possible action. In real-time the much faster neural network then processes the information, and outputs an action. This method was never used in the competition created by Lucas [14].

## 1.1 Contributions

The main purpose of this project, consist on comparing two agents, both using MCTS. One agent will use the MCTS in real-time, with modifications in order to improve its performance in Ms. Pac-Man, this agent was developed by Pepels et al. [18]. The second agent will be created from scratch by us. It will run the MCTS algorithm without modifications or improvements, in offline mode, this will allow for the creation of a dataset that will be used to train a neural network that will play the game in real-time.

This technique used for the second agent, has already been proven as an effective technique that can obtain excellent results in multiple games from the Atari 2600 console, as proven by Guo et al. [9]. However, it was only tested in a framework that allows the capture of images as input for the agent. The idea will be to use this same technique but altering the input features from an image to a set of parameters (game score, pills in the maze, ghost positions, etc), to understand if the results obtained can be comparable to a state of the art agent using MCTS in real-time.

With this said, the key contributions are:

- Creation of a standard MCTS agent capable of playing Ms. Pac-Man
- Creation of two datasets capable to be used in classification and regression.
- Analyse of an effective neural network architecture

## 1.2 Organization of the Document

The document is organized in chapters. Chapter 2 consists on a short analysis of several technologies necessary to the development of this thesis.

In Chapter 3, we analyze several papers that provide a solid start to the development of the agent. These papers, are research works on several technologies that can provide an idea of what works and doesn't on the development of an agent for Ms. Pac-Man.

Chapter 4, is the development of the agent. The implementation of the MCTS, the neural network and the solutions adopted for each step.

Chapter 5, consists on the presentation of the results obtained followed by its analysis. With the vast amount parameters varied, multiple tests were conducted in order to obtain a clear picture of the agent created.

Chapter 6, is the final chapter. This chapter mentions the conclusions obtained and possible options to analyse in the future.

# 2

# Background

## Contents

The controllers of Ms. Pac-Man, started by having a rule-based algorithm at its core. However, this technique was quickly surpassed by the MCTS algorithms, after the success it had in the game Go. More recently, convolutional neural networks, started making their success in games, overall.

This section will be focused on starting by explaining the differences between the original game, Pac-Man and Ms. Pac-Man. After that, the MCTS algorithm will be described followed by deep learning in general, in order to understand the papers presented in the next chapter.

## 2.1 Ms. Pac-Man

The game Ms Pac-Man is very similar to the original one, Pac-Man. However, there are some important differences.

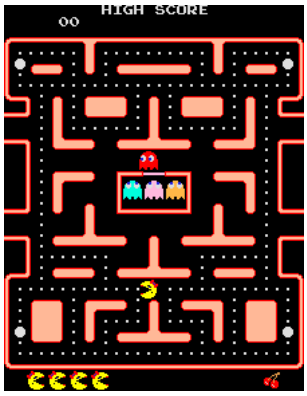


**Figure 2.1:** First Maze of Ms. Pac-Man

In Figure 2.1, it is possible to observe the first maze of the game. The small dots represent the pellets, the bigger dots represent the power-pellets. In the bottom, the number of Ms.Pac-Man are the remaining lives. Due to this figure showing the first level, the fruit at the bottom is the cherry. The score is showed at the top, together with the highest score achieved.

In the original game, there is only one maze with one tunnel, while the Ms Pac-Man version, has four different mazes, with three of them having 2 tunnels and one of the mazes having one tunnel.

The goal of the game stays the same, maximize the points earned. For that objective, Ms. Pac-Man presents no differences to the original game, Pac-Mac. The agent has to eat the pellets and the power pellets. Eating a power pellet allows the player to eat the ghosts, the more ghosts eaten in a power pellet the more points you get for each ghost eaten. The last way to win points, is the player eating fruits. These have a slight difference from the original game. In the original, the fruits would appear in the center of maze, in Ms Pac-Man the fruits wander through the maze. The fruit varies according to the level, in the first level it is the cherry; after level 7, all fruits can appear.

Lastly, probably the most important difference. While the ghosts in Pac-Man follow a predefined moving pattern according to the player movement, the movement of ghosts in Ms Pac-Man is more random, making impossible to analyse patterns in order to choose an action. Making it a more challenging game, for controllers.

## 2.2 Monte Carlo Tree Search

Monte-Carlo Tree Search, as mentioned by Browne et al. [4], is at its core a rollout algorithm. Through Monte-Carlo simulations, it is possible for the algorithm to calculate value estimates for each trajectory, discovering the best path. This can be used to solve MDP's.

The base idea is to run multiple simulations from the same node, extending trajectories that received a good evaluation from a previous simulation. The more simulations ran, the more precise and more accurate will the values of each edge be, due to the nodes keeping track of the number of times they were visited in the past and the results obtained. The actions selected during the simulation, follow a simple policy, called the rollout policy, usually this policy follows a uniform distribution. Like others Monte-Carlo methods, the value of each state-action pair is the average of the values returned from the simulations.

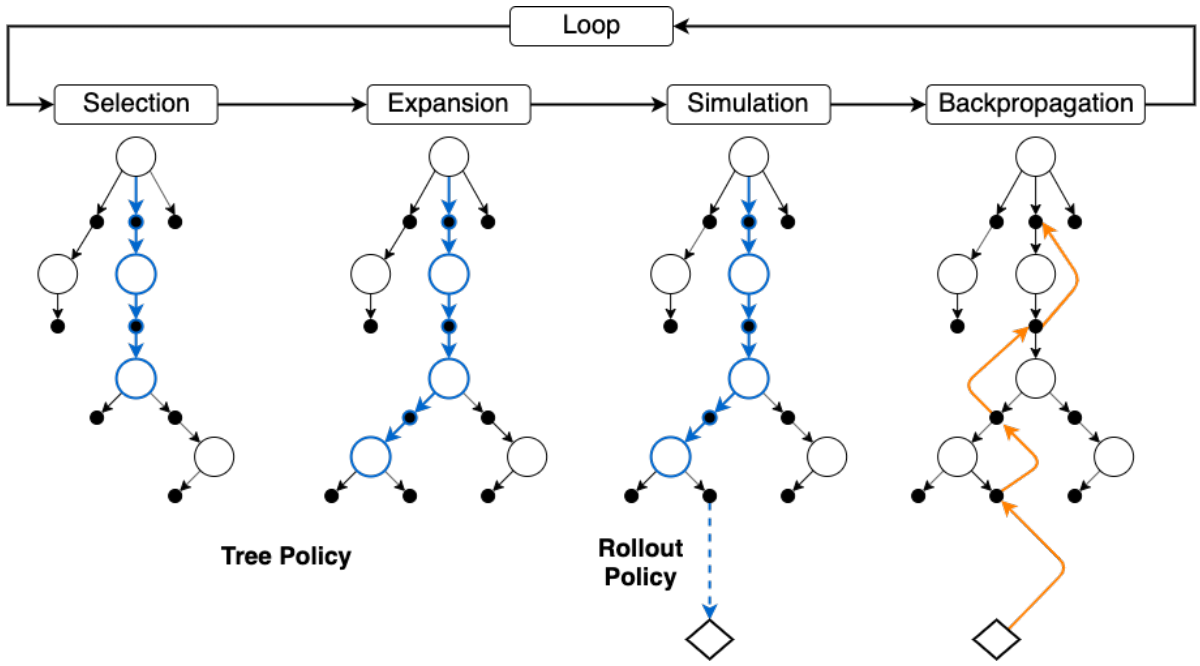


**Figure 2.2:** Monte-Carlo Tree Search Steps

In Figure 2.2, it is possible to observe the steps of the MCTS algorithm. The circles represent the states, while the black dots, represent the possible actions to take from that same state. The steps are

described as following:

- Selection. Begins at the root node, and starts selecting child nodes indicated by the tree policy based on the action values of each edge. This process will occur until a leaf is reached.
- Expansion. When the selected leaf is reached, one or more child nodes are created, through unexplored actions, expanding the tree.
- Simulation. One of the child nodes is then chosen and starting from that node, a random playout (or rollout) is executed until a final state is reached.
- Backpropagation. The value obtained by the rollout is then used to update the state-action values of each edge (black dot). This values will help the tree policy to perform a more informed decision in comparison to the previous iterations.

MCTS will continue to execute this steps until some condition is reached, it is usually either a time or computational restriction. When the environment passes to a new state, the MCTS will begin to run again, from the new root node, that represents the new state. Usually the tree is completly discarded with the exception of the nodes that connect to the root node.

One important aspect to have in consideration is the tree policy. While the rollout policy is commonly a policy that is simple, that does not require alot of computational effort. The purpose is, the simpler the policy, the faster the simulation runs, which allows for more simulations to be analysed. For the tree policy, it is needed a policy that can balance efficently between exploitation and exploration. For the MCTS, most papers use UCT.

### 2.2.1 UCT - Exploration vs Exploitation

The exploration versus exploitation problem, is a dilemma between the decisions of when to explore and when to exploit.

When the algorithm does not know the environment, by logic it should explore, it does not have information about anything to exploit. After some iterations, when some parts are already known, is when the problem appears, does the algorithm know enough to start exploiting?

To solve this problem Kocsis and Szepesvári [11], invented the UCT algorithm, derived from a multi-armed bandit algorithm, UCB1 (Upper Confidence Bound) by Auer et al. [3], modifying it to work on tree algorithms, like MCTS.

Like said before, for MCTS to work, it requires a selection policy during the selection step. This policy must be able to explore the tree for nodes with a high ”score”, but still be able to choose nodes from where not enough information as been gathered, again the exploration vs exploitation problem. The UCT algorithm allows for exactly this, a balance between both.

$$UCT_i = X_i + C * \sqrt{\frac{ln(n)}{n_i}} \tag{2.1}$$

To obtain a ”score” for each node, the Equation 2.1 is used. The $X_i$ is the discount reward, representing the amount of points that node obtained through simulations, this parameter represents the exploitation part of the equation. The exploration half is represented by the rest. The $C$ is a constant, it is common to use the value $\sqrt{2}$, but it should be obtained empirically. While the $n$ and $n_i$, represents the number of times the parent node and the child node are visited, respectively.

## 2.3 Neural Networks

Neural networks, are somewhat inspired in the human brain biology. A neural network possess neurons connected to each other in layers, forming a network.

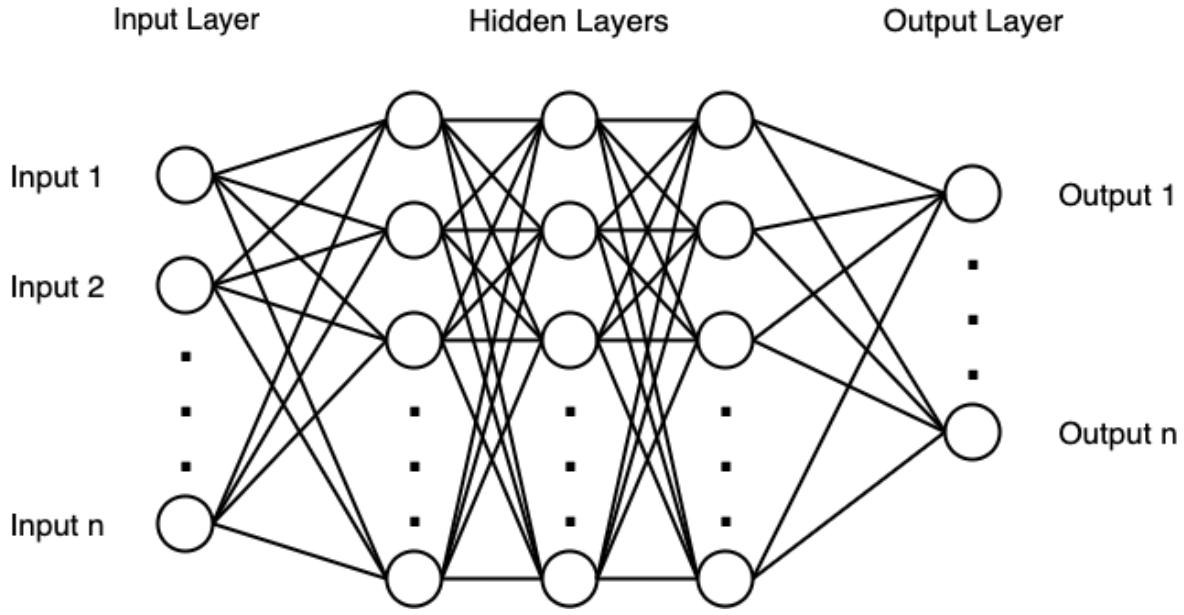


**Figure 2.3:** Example of a Neural Network

A neural network is usually, read from left to right. The input layer receives the various features of the sample, that the neural network is trying to predict. The hidden layers then perform calculations and the output layer will output the result predicted. Each neuron will perform calculations based on the input received. As seen in Figure 2.3, each neuron present in the hidden layers are connected to every other neuron present in the previous and next layers, this is called a fully connected neural network. The connections from neurons in the previous layer will act as input for the neurons in the next layers. These inputs will then be used to perform calculations on the current neuron, with this neuron then outputing a

value for the neurons present in the next layer to use as input, and so on. The network present in Figure 2.3, due to having multiple hidden layers can be called a deep neural network.

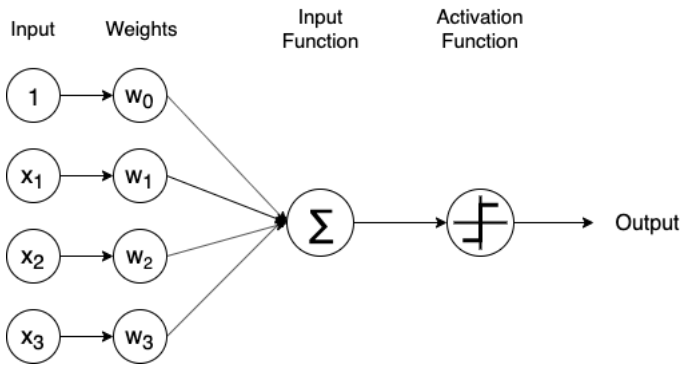


**Figure 2.4:** Example of a Neuron

Figure 2.4, shows a neuron up close. Every input is multiplied by the corresponding ”weight”, this weights can be initialized in several different ways, starting all the weights as 1 or even initialize them following a distribution. This parameter will be constantly changed during the training phase of the network, in order to improve the network accuracy, this accuracy is measured by a ”loss function”, with the purpose of the ”weights” being to minimize the ”loss function”. All the inputs, already multiplied are then summed all together and introduced in an ”activation function”.

There are multiple types of activation function, but they all have the same purpose, converting the sum to a value between a certain range. That will be used as input in the next neurons.

There are multiple types of neural networks, this one was just an example to introduce the concept. One type of neural network, currently having a great deal of success are the convolutional neural networks. This type of neural networks have showed a great capability classifying images. Some of the most recent development in the area of creating controllers for games in general, have this technology at its core. These networks are capable of extracting the image displayed on the screen and classify it correctly for the agent, allowing the agent later choose the correct action to take, with the most notorious case probably being the Alpha Go agent, that succeded in defeating a Go 9th Dan player, Lee Sedol. For this feat, Alpha Go used neural networks in combination with tree search algorithms, as explained by Silver et al. [23].

These convolutional neural networks stand out from the rest, due to the fact due to being able to automatically perform feature extraction, the features of the images, that are used as input. Instead of a human deciding what features are important, these layers do that themselves.

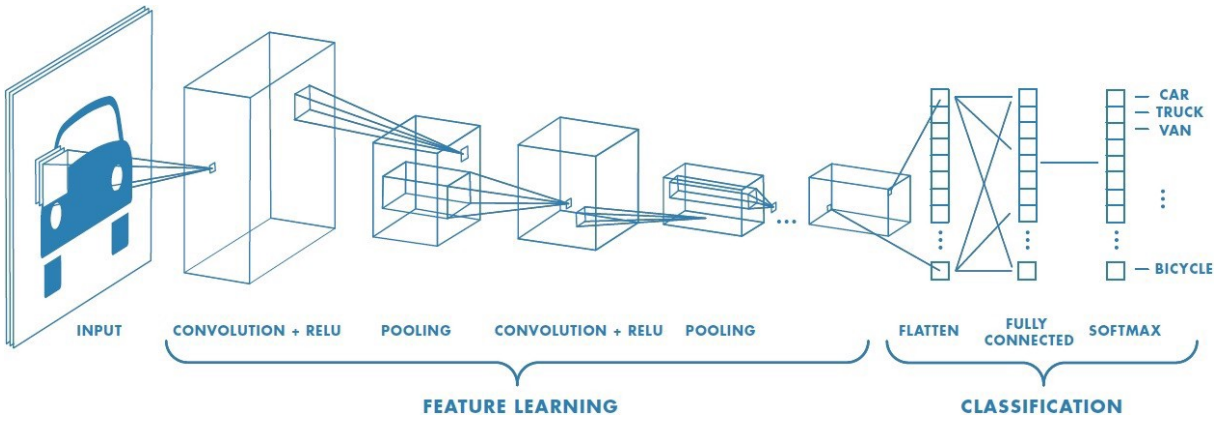


**Figure 2.5:** Example of a convolutional neural network - From Saha [21]

As observed in Figure 2.5, the first layers are used to extract the information from the input, in this case, an image, and then a fully connected network, like the one mentioned above is used to classify the image. For the feature extraction, there are three main types of layers: convolutional, non-linear and pooling.

**Convolutional**, is the main layer in terms of extracting the features of the input image. An image can be considered as a matrix, with each pixel representing a position of the matrix. The operation that the convolutional layer does is applying a filter through the matrix that represents the input. In the case, seen in Figure 2.6, the filter is going through the image with a stride of one, this means the filter moves one pixel at a time, to the side. The filter multiples each position element wise, and then sums all the values, and applies it to the output matrix. The output matrix is called a feature or activation map.

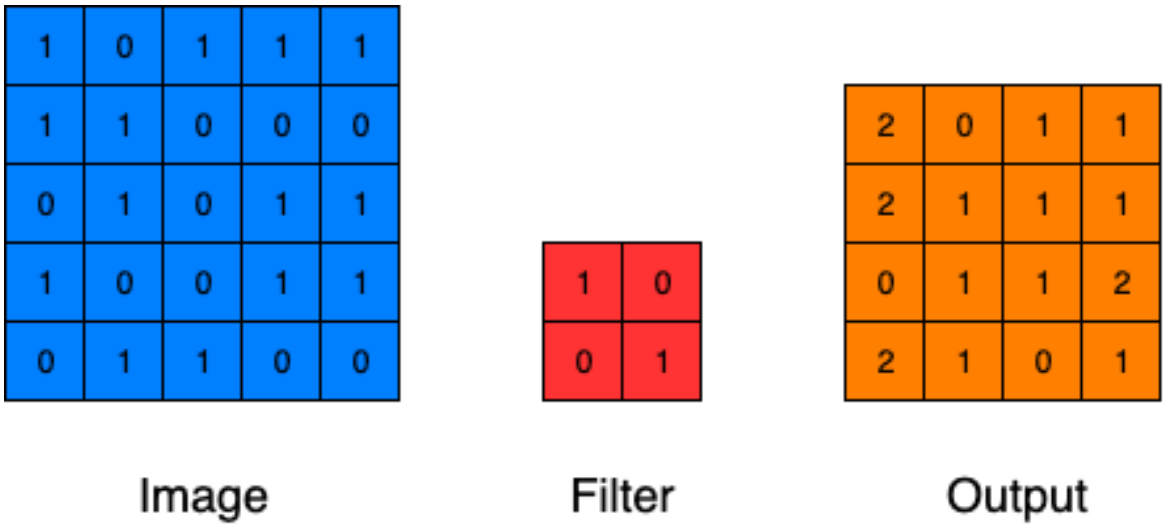


**Figure 2.6:** Example of applying a convolutional filter to an image

**Non-Linearity**, is a layer that applies a type of non-linear function, also known as an activation

function, in order to remove the linearity from the feature map. This step is required due to the operation made by the convolutional layer. The operation introduces linearity, when most things observed in the world do not possess such linearity, distorting the original data. One of the most common operation used in this step is the “ReLU” function.

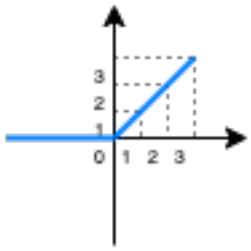


**Figure 2.7:** ReLU Function Plot

The function maximizes the value of each individual pixel in the feature map. It compares the value of the pixel with zero, and picks which value is higher, as seen in Figure 2.7. For example, a ReLU function applied to the output of the Figure 2.6, would change nothing. Other examples of non-linear operations are $tanh$ and the $sigmoid$ functions.

**Pooling**, consists on reducing the dimensionality of the feature map, basically downsampling. Pooling can be done with multiples functions, being the most commons, the max, average or sum. In Figure 2.8, a filter of 2x2, with stride 2 (moving 2 pixels at each time), is selecting the max value in that section of the feature map, producing the output matrix.

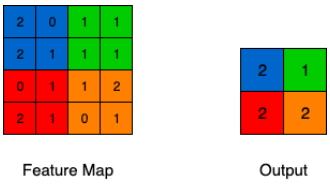


**Figure 2.8:** Example of pooling a feature map

These layers can be used multiple times in a neural network, but usually always follow this order. Convolutional layer, with a non-linear layer after, and in the end a pooling layer.

With the features extracted, the image is now ready to be classified by a fully connected layer.

The technologies mentioned in this chapter, are at the core of the research papers mentioned in the next chapter, as well as in the development of the agent created for this thesis.

# 3

# Related Work

## Contents

There are, already, multiple agents developed for Pac-Man and Ms. Pac-Man games, with the most diverse solutions, to be used in competitions, that are mentioned below. This was due to the fast increase in popularity, of the games in these tournaments, which attracted multiple academic teams.

One of the earliest researches, investigated the use of genetic programing, for the controller to choose one of the predefined rules. Alhejali and Lucas [1] were able to use this technique to evolve the behaviours of multiple Ms. Pac-Man agents, for multiple variations of the original game. Further down the line, Alhejali and Lucas [2] were also able, to use genetic programming to evolve heuristics for a MCTS agent.

Burrow and Lucas [5], compared the use of temporal difference learning against evolutionary algorithms for learning behaviours of the agent. They reached the conclusion that, for this game in particular, the evolutionary algorithms were better than TD-Learning.

Robles and Lucas [20], used a tree search algorithm, with the best path being evaluated through hard-coded heuristics. Still in rule-based systems, multiple teams also used the Dijkstra's algorithm to determine the way to follow.

Lucas [13], used evolved neural networks, to evaluate the best action for the agent. Gallagher and Ledwich [7], also used a neural network in combination with a evolutionary algorithm, and they were able to create agents that would learn the rules of the game, and show a begginer level of skill in the game.

Pepels et al. [18], were the first real team, to improve upon the current iterations of MCTS in Ms. Pac-Man. They proposed several enhancements and alterations, in order to improve the performance of the agent. They ended up in second, in the WCCI'12 (World Congress on Computational Intelligence) out of 63 competitors and in first in the CIG'12 out of 36 contestants.

In this section, multiple papers will be discussed. Starting by the implementations in MCTS for Ms. Pac-Man, as well as some more recent techniques that combine the use of deep learning with reinforcement learning, that present some incredible results in multiple games from Atari 2600, without changing any parameters.

## 3.1 Monte Carlo Tree Search

The idea of applying Monte-Carlo methods to Ms. Pac-Man, comes from its strong performance in other games. One of the most famous was the agent created by Silver et al. [23], to play Go. More recently, there have been developments in real-time games, more specifically in RTS (Real Time Strategy) games, as demonstrated by Chung et al. [6].

In this section, several papers will be mentioned that use MCTS to create an Ms. Pac-Man's agent. Each one has its own implementations and solutions for the problems encountered.

### 3.1.1 Max-n Monte-Carlo Tree Search

Samothrakis et al. [22], were one first teams to experiment with MCTS in Ms. Pac-Man.

They addressed the problem, by considering tree searches on a 5 player $max^n$ tree (this algorithm is explained below) with limited depth, due to Ms. Pac-Man having an end state only when it dies or the maze ends. Due to the lack of a standard process on how to apply MCTS, it was used a four step process based on empirical evidence. The first step, is analyse the number of agents and the information we can extract from the game while playing, with the objective of choosing the right tree. Since, they considered this game as 5 player game, the algorithm chosen was the $max^n$, developed by Luckhart and Irani [15]. Which allows for each individual player to maximize their own payoffs, independently from the other players.

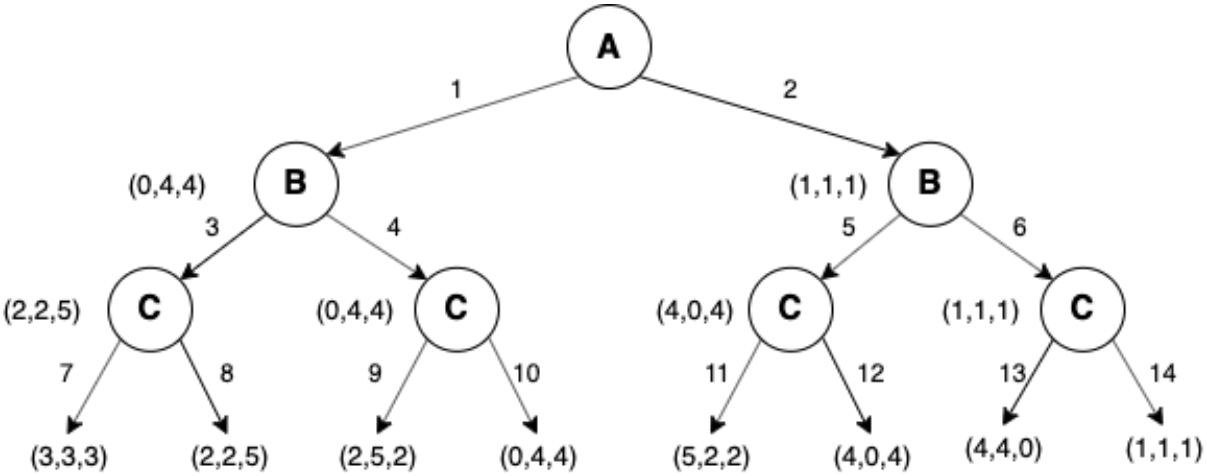


**Figure 3.1:** Example of a $max^n$ tree

The tree despicted on Figure 3.1, shows a game between three people, ”A”, ”B” and ”C”, each one with the objective of maximizing their individual reward. ”A” has to choose between action one or two. In order, to maximise its own reward (position one of each vector), the agent will choose the action number two. ”B” will follow the same idea, choosing action number six, because the second position (that corresponds to agent ”B”) of the reward vector is higher than if the agent took the action number five, and so on.

Considering the ghosts and Ms. Pac-Man the agents, it is a game with five players, each one maximizing their reward.

Second step, consisted on identify the policy to use. The policy used was one derived from the UCB, the UCB-TUNED as seen in equation 3.1, from Auer et al. [3]. The UCT was also considered, but after some tests, this one led to better results.

$$\overline{x_j} + \sqrt{\frac{ln(n)}{n_j} min\left\{1/4, \overline{x_j^2} - \overline{x_j}^2 + \sqrt{\frac{2ln(n)}{n_j}}\right\}} \tag{3.1}$$

Third step, was to consider a back-propagation policy. The end node, of the simulation has an array of rewards, one of each player in the game, in this case, five rewards. The rewards are then passed back, each one to its corresponding nodes, since one node can only belong to only one player.

The final step, consists on how to guide the Monte-Carlo simulations, in this case, it was done with the use of heuristics.

The main problem the MCTS faces in Ms. Pac-Man, is the rarity of an end state. Samothrakis et al. [22], solution for this specific problem is to limit the depth of tree. Another limitation imposed is the capability of Ms. Pac-Man moving backwards. This last part is very important, in the sense that if this rule was not implemented, the tree would grow exponentially. The simulation ends when the pac-man reaches a state where she is eaten, when there are no more pellets, or when the max depth is reached. Since the most probable case is the pac-man reaching the max depth of the tree, this would lead to pac-man wondering randomly in the maze. As a solution, it was created a function with objectives for the pac-man to achieve in determined situations, with the function deciding which is the ”better node” inside the tree depth limit. If a ghost is edible and is at a lesser distance than $x$, the pac-man follows the ghost, if there are no edible ghosts and the distance is to a pill is smaller than $y$, the pac-man goes after the pill and in last, as a condition to catch all cases, the pac-man goes to nearest node.

For the reward system, if the maze is cleared, pac-man receives 1 and the ghosts receives 0; If pac-man is eaten, it receives 0 and the ghosts 1. As for the other cases, the ghosts receive a reward, proportional to the inverse distance between them and pac-man; As for the pac-man, the rewards are attributed in relation to the current objective, if its reached. However, for the pac-man the values are hardcoded, and they were obtained after experimentation, without any real equation.

This first implementation of MCTS in Ms. Pac-Man tackled some of the problems that would make the use of this algorithm in this game unusable. The complications, derive from the fact that Ms. Pac-Man does not possess an end state and is being played in a real-time environment, in contrast to board games, where time constraints are not a factor, where the MCTS showed a great level of performance. While this paper, discards the time limitations and considers that improvements can always be made in terms of code optimization or in the form of a better CPU, the solutions proposed, tried to adhere to this constraint, avoiding solutions that would be improper for this implementation to work within time limitations. This method obtained, using the same simulator, better results, almost two orders of magnitude better, than methods using evolutionary, reinforcement learning and genetic programming methods, according to Samothrakis et al. [22].

### 3.1.2 Monte-Carlo Tree Search to avoid Pincer Moves

Ikehata and Ito [10], approached Ms. Pac-Man with the intent to avoid pincer moves. These moves, consist on blocking the possible paths of Ms. Pac-Man, leaving it with no possible way out. For this, it was used the MCTS in combination with UCT. To build a complete tree to be able to predict future situations, that could lead to pincer moves, requires a lot of processing, to generate trees deep enough to obtain enough information, but in real time that's not reasonable. What is possible, however, is to calculate the probability of being caught in such moves. Reducing this probability, also reduces the chances of being trapped.

The tree is built around, what Ikehata and Ito [10] calls "C-Paths". These paths are an interrupted route between two junctions. The root node represents the current position of Ms. Pac-Man, the rest of the nodes in the tree represent the junctions and the edges, the paths. The tree does not represent the ghosts' movement because that would have increased the number of situations.

During the selection step, the policy used was the UCT. While the policy used for this step was different from the Samothrakis et al. [22], in the simulation part of the MCTS, the choice was the same, the use of heuristics. Ikehata and Ito [10], also considered for this step a turn-based setting, where the pac-man and the ghosts move alternately; And again the rules of the game are different while in simulation, with both papers imposing restrictions, in order to ease the computational workload in a real-time environment.

The heuristics used for the simulation step are:

- Ms. Pac-Man will not go back to the C path she was on when choosing a path at a junction.
- When choosing a path at a junction, if there is a non-edible ghost on a C path, that pac-man can move to and the ghost is coming in her direction, she eliminates that C path, from the possible choices.
- If a unique C path is not determined by applying the rules above, a random path is chosen.
- If there is an non-edible ghost within a certain distance on the same C path, pac-man will reverse her direction at the spot.

For the ghosts, during simulation, an equation was designed to emulate the characteristic that they get more aggresive when there are fewer pellets in the maze. This equation will, in a probabilistic way, move the ghosts closer to pac-man.

$$P(G_{type}) = A(G_{type})\frac{AP - RP}{AP} \tag{3.2}$$

$AP$ means the total amount of pellets in the maze, $RP$ the remaining pellets. $A(G_type)$ indicates the

| Ghost Color | Aggressiveness | Target Points |
|---|---|---|
| Red | 0.9 | Ahead of Ms. Pac-Man |
| Orange | 0.8 | Behind of Ms. Pac-Man |
| Pink | 0.7 | Ahead of Ms. Pac-Man |
| Light Blue | 0.6 | Behind of Ms. Pac-Man |

**Table 3.1:** Ghosts' Aggressiveness

aggressiveness of the ghost, each one has a different level, as seen in Table 3.1. This Equation, like the following Table, allows for the simulation step to better emulate the behaviour of the ghosts.

$P(G_{type})$ is the probability of the ghost choosing a C path, that will lead closer to pac-man. As, also seen in the Table 3.1, the color of the ghost also affects the way the ghost approaches the pac-man. This allows the ghosts to do the so called "pincer move".

An important concept introduced by Ikehata and Ito [10], is "tactics". In Ms. Pac-Man there is not a win state, the game is infinite and maximizing the score may not be the best idea, since in the next move, pac-man might be eaten by a ghost. So Ikehata and Ito [10], considered three sub-goals:

- Avoid touching ghosts
- Eat all the pellets
- Eat as many ghosts as possible

This concept of tactics was created in order to achieve these sub-goals. The behaviour of pac-man will change in relation to the current tactic. There are also three tactics, "survival", that makes pac-man avoid ghosts at all costs, "feeding", that focuses on eating pellets, and "pursuit" that prioritizes on eating ghosts. The UCT value of a node, is what decide the current tactic in conjunction with the state of the ghosts. For example, if the current tactic is "survival", and the UCT values for all possible paths are above the threshold, then the tactic is changed to "feeding". The following rules, with the values obtained from UCT, are the conditions to change tactics:

- If the survival rate of the previous frame was above a certain threshold, then the tactics of "feeding" are adopted.
- If the survival rate of the previous frame was below a certain threshold, then the tactics of "survival" are adopted.
- If the ghosts are in the blue state and the closest ghost is within a certain distance, then the tactics of "pursuit" are adopted.
- If the ghosts are in the blue state and the closest ghost is beyond a certain distance, then the tactics of "feeding" are adopted.

- If all ghosts are active, then the tactics of “survival” are adopted.

With the tactics adopted, rises the need to select the correct path for each tactic.

- Survival - Chooses the action that maximizes the survival rate.
- Feeding - Chooses the action that has the most dots to eat and has the survival rate above the threshold.
- Pursuit - Chooses the action that leads to eat the most amount of ghosts and has the survival rate above the threshold.

In conclusion, this approach to try and reduce the amount of times the pac-man is cornered worked. Although that was the main goal of this paper, the most important part to keep, is the introduction of sub-goals and mostly tactics. These tactics would later be adopted by Pepels et al. [18], in their implementation of a controller for Ms. Pac-Man. Their implementation would go on to win CIG'12 and come in second in WCCI'12.

### 3.1.3 Monte-Carlo Tree Search Agent using Genetic Programming

Alhejali and Lucas [2], proposed the implementation of an MCTS controller for Ms. Pac-Man, that would use genetic programming to evolve and enhance the policy used during the simulation step. This idea, was supported by the successful implementation of MCTS and genetic programming agents in Ms. Pac-Man, mainly by, Samothrakis et al. [22] and Alhejali and Lucas [1].

Instead of using a random policy during the simulation step, this new policy would be able to use an agent capable of extracting information about the current game state, creating new simulations, with the only requirement being to keep the computational time low.

The system used, at the start would begin by creating a entire new population at random, with each one being passed to the agent for evaluation. To evaluate, the agent runs a simulation for each evolved tree. In the end, a new generation is created using crossover, mutation and reproduction beggining a new cycle. To achieve this, multiple operators were used, like numerical comparison, logical among others. As for the terminal, it was chosen three types. Each one with their purpose, action terminals direct the Ms. Pac-Man somewhere, a power-pellet, a ghost etc. The logical terminals answer questions like ”isEdible?”. The numerical terminals return numerical data, like distance to an object. Due to the time of simulation, the population size and the number of generations was reduced to 100 and 50 respectively.

The results showed what already was seen in other papers. The time is the main obstacle, with the MCTS system being able to equal a well-evolved agent if it had time to do a sufficient number of rollouts. For the evolved agents the problem was the same, Alhejali and Lucas [2] were forced to reduce the

number of evaluations to 5000 from 25000, also due to time constraints. Nevertheless the evolved agent was able to outperform the random implementation of MCTS by 18%.

### 3.1.4 Real-Time Monte-Carlo Tree Search Improvements for Ms. Pac-Man

Pepels et al. [18], created a controller for Ms. Pac-Man based on the MCTS algorithm that focuses on dealing with problems that rise due to this game being run in a real-time environment. This implementation won the CIG 2012 competition and came in second in WCCI 2012. This paper proposed four improvements, in relation to other controllers that also used the MCTS algorithm. A tree with variable-depth, strategies for the pac-man and ghosts, this strategies are derived from the implementation of Ikehata and Ito [10], long-term goals in scoring and reusing the search trees in order to save computational time.

#### Trees with Variable Depths

It is possible to map the mazes in trees, with the edges representing the paths between junctions and nodes representing, the said junctions. Through the tree, the agent can at the edges choose to go forward or reverse back, and at the nodes, choose between directions. While in the real game, it is possible for the agent to reverse multiple times in a row, in the tree of Pepels et al. [18] it is only possible to reverse once, the reason for this, is to allow a bigger margin between rewards for the available moves, otherwise it would be necessary more simulations per move to achieve decisive differences between moves.

The trees constructed, by these papers, are very similar between themselves. The difference is the control of the tree's depth. While the others, consider the distances between sequential junctions to be always the same. This makes the transition from the maze to the tree to not be accurate, and having a fixed depth rule to stop the rollout only worsens the situation. Pepels et al. [18], also uses the depth of the tree to stop the rollouts, but this depth is controlled by the distance between sequential junctions. This allows for the agent to avoid bad decisions, e.g. when in danger choose the shortest path instead of a longer one.

As seen in Figure 3.2, the distance between junction "F" and "L" is very different from the distance between "F" and "K".

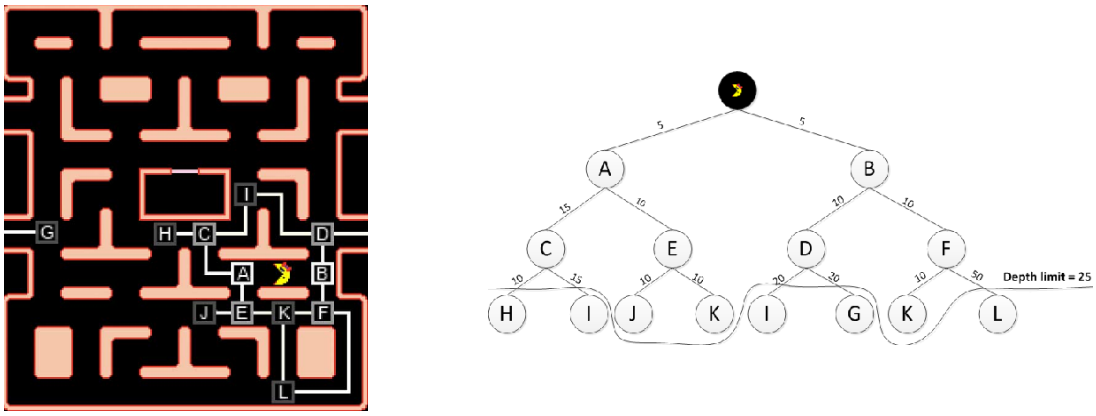


**Figure 3.2:** Game state, and respective mapping in a tree - Adapted from Pepels et al. [18]

The other papers, limit the depth of the tree by counting the number of nodes. Pepels et al. [18], does it by a measure of distance in the maze. That is why, in Figure 3.2, some nodes on the last level of depth are not reachable, the distance from the root to them is bigger than the limit of 25.

### Tactics

As defined by Ikehata and Ito [10], the use of the UCT method implies the calculation of a formula based on the winrate, but due to Ms. Pac-Man lack of a ”win” state, other arrangements had to be made. One can say the ”win” state is to maximize the points obtained, but that means close to nothing if in the next move, you are eaten.

Pepels et al. [18] iterated upon this idea, but considered the tactics in relation to the rewards. Each node would have three different reward values, one for each tactic and instead of using heuristics like Ikehata and Ito [10], decided to use a mathematical approach. There is a value for each tactic, that is saved at each node, and then later used to be applied to the UCT equation. Of the three values saved at a node, the one choosen is the one of the current tactic, according to the equation 3.3.

$$v_i = \begin{cases} M^i_{ghost} \times M^i_{survival} & \text{if tactic is pursuit} \\ M^i_{pellet} \times M^i_{survival} & \text{if tactic is feeding} \\ M^i_{survival} & \text{if tactic is survival} \end{cases} \tag{3.3}$$

This value, will later be used by the UCT to decide the paths to take in the selection step and to calculate the new values calculated by the backpropagation step with the $M^i_{tactic}$ representing the reward of each tactic.

The main difference between Ikehata and Ito [10] and Pepels et al. [18], is the use of heuristics versus math. While Pepels et al. [18], uses the value obtained as a reward, Ikehata and Ito [10] uses the rules as definitions in order to implement restrictions and more strict decisions.

### Search Tree Reuse

One of the main constraints these types of agents need to have in consideration in real-time environments, is the time that the agent has to decide each move. In Ms. Pac-Man that time is 40 milliseconds.

Due to the nature of the MCTS algorithm, and how its success depends in great part to the amount of simulations analyzed for better approximations, this short time of decision is far from ideal.

As an option, Pepels et al. [18] suggested reusing the search trees. However, neither solution is perfect. Reusing a search tree may result in bad decisions due to outdated values, making the agent biased. Rebuilding a tree at each junction may put the agent in a situation of going back and forth by deciding between actions with similar values.

Pepels et al. [18], decided to combine two techniques that allowed them to reuse past trees, saving time in the decision process. The first technique, decides if the game state is considerably different, indicating that the values stored in the tree are no longer meaningful, this technique is rule based. The second technique decays the values stored in the tree, making them less important in the decision process compared to more recent values. This technique ensures that the older values still influence the action to take, but due to the possibility of them being outdated, their importance is lowered.

First technique, when playing the game, an action can change the game state drastically, that was not predicted by the current search. This makes the current tree obsolete. Three rules were made, that if they occurred, a new tree would have be to build, Pepels et al. [18]:

- Pac-Man has died, or entered a new maze.
- Pac-Man ate a blue ghost or a power pill. This ensures that the search can immediately focus on eating (the remaining) ghosts.
- There was a global reverse. If all ghosts are forced to reverse, values of the previously built search tree no longer reflect the game state.

The second technique, is about decaying the values stored at each node. Pepels et al. [18], achieves this by multiplying all the values stored at each node by a factor of $\lambda$ at the beggining of each turn. Other methods of decaying, have already been proposed for UCB in multi-armed bandit problems, by Kocsis and Szepesvári [12].

However, the discount is only applied once per search, instead of everytime the node is visited. This way, the importance of the rewards further from the root is reduced, just like in markov decision processes. Due to the reduced time the agent has to decide on the action to take, the approximations are far from ideal, making further rewards instable and more abstract, and so its better to reduce their value and importance with this technique.

#### Long-Term Goals

Pepels et al. [18], adds another modification upon the work of Ikehata and Ito [10]: long-term goals. They consider two main long-term goals, eating ghosts as fast as possible, to reduce the risk of death, and at 10000 points, Ms. Pac-Man receives a life. Due to MCTS, looking at short-term goals, modifications had to be made for it to be able to calculate the rewards, and encourage the agent to go for these specific targets. The rewards in question are the $R_{Feeding}$ and $R_{Pursuit}$ rewards.

For the $R_{Pursuit}$ reward, in a long-term scenario, the reward is attributed according to the remaining time that the ghost was in an edible state. If this reward at the end is inferior to 0.5, the $R_{Feeding}$ reward is zero. The objective is to "show" to the agent, that the ghosts were too far away, and eating that power pellet was a waste. On the other hand, if the ghost reward ends up being above 0.5, the feeding reward is calculated as such: $R_{Feeding} = R_{Feeding} + R_{Pursuit}$. That was the method, to introduce a long-term reward for eating ghosts in the most efficent way.

For the other long-term goal, maximizing the overall score. The more lives, Ms Pac-Man has, the more safe she is, and for that, she has to score 10000 points fast and efficently. The game, also, has a time limit of 24000 time units, before it passes to the next maze. For these reasons, if the agent wants to maximize the points scored, it needs to do so in a fast way, due to the time limit. Pepels et al. [18], introduced an edge reward, when all the pellets in an edge are eaten. This is beneficial, in comparison to leave single pellets spread around the map, making harder and risky to eat. While on one hand, longer edges have a bigger scoring potential, they are also more risky to clear than smaller edges.

These four improvements alongside, the creation of rules for the simulation step, led to an increase in the score. It is important to note that, while each improvement led to a small increase to the score, the use of a rule-based simulation led to the most significant increase in the points. This supports the other papers. Due to the limited amount of time for a proper simulation to be executed in real-time using random methods, the results using heuristics prove to be much better.

Overall, all papers got to similar conclusions. With the limited time to process each move, the step of simulation can not be left to randomness, it has to be conditioned by heuristics. This factor, was the most critical in all the papers providing the biggest increase to the points. In combination with the heuristcs, the creation of tactics, introduced by Ikehata and Ito [10], also improved the results significantly. All other modifications, presented small improvements, but not significantly to be called breakthroughs.

## 3.2 Deep Reinforcement Learning

More recently and outside the scope of the competition, DeepMind, the company, pioneered a combination between two techniques, reinforcement learning and deep learning. Agents using reinforcement learning, usually have problems extracting information from the environment, when this environment is

too complex to be handcrafted or has a high-dimensional state space. It is in this part, that the advances in deep neural networks come in to play. These neural networks are able to extract information from highly complex environments. This combination, is called "Deep Reinforcement Learning". As mentioned by Mnih et al. [17], an agent with this technology, was tested using 49 different games from Atari 2600. With the agent, as input only receiving the pixels and the game score, was able to achieve higher scores than previous algorithms, without changing parameters or architectures.

### 3.2.1 Deep Reinforcement Learning for Atari Games

Guo et al. [9], succesfully presented a deep learning model, trained with a variant of Q-learning, that was able to extract information from images and to output a value function estimating future rewards.

One solution adopted by Guo et al. [9], was a technique called "experience replay", that uses random samples, which smooths the training distribution. "Experience replay", consists on storing the experiences $(s, a, r, s')$ that the agent passes through at each time-step. With the Q-learning value approximation function, updating experiences at random from the pool. After this, the agent selects an action based on an $\varepsilon$-greedy policy.

This approach leads to several advantages comparing to the standar Q-learning:

- Each experience is used in many weight updates, leading to a greater data efficiency.
- Using random samples, breaks the correlation between samples, reducing the variance of the updates.

Guo et al. [9], calls convolutional networks trained with this approach as "Deep Q-Networks".

This provides a platform, for the use of reinforcement learning algorithms in highly complex environments, allowing the use of these algorithms to train neural networks.

### 3.2.2 Recurrent Deep Q-Learning

Ranjan et al. [19], like the previous paper, also noted the limitations of using handcraft rules to map a game domain. A data-driven approach allows, for the networks to discover, by themselves, features that might be useful, as making the agent more abstract from the environment he is inserted.

Their idea, is to use a combination of convolutional neural networks and recurrent neural networks with LSTM (Long Short Term Memory) units to model the Q-values. The use of RNN (Recurrent Neural Networks), in combination with LSTM, increase the memory of the agent.

The agent was tested for the game Pac-Man, in the end, the agent improved significantly after the first 100 games, but stayed reasonably the same even after 300 games. Ranjan et al. [19], deduced that this happens due to the small number of iterations used to train the network.

In conclusion, the agent never achieved a level of human gameplay for Pac-Man. However, the same happen to Mnih et al. [17]. While this new approach was an improvement upon the previous algorithms, it was not an increase enough to equal a human. This can be justified, as said above by the difficulty of modeling a network good enough for Ms. Pac-Man and Pac-Man.

### 3.2.3 Deep Learning with Monte-Carlo Tree Search Planning

Guo et al. [9], goal was to surpass the recent innovation that was the DQN's.

The idea to be able to use these methods, consists on using them for creating training data, *a priori*, for the deep-learning algorithm to use in real-time play. Basically, developing methods that can keep the advantages of deep learning not needing the handcrafted rules, while being able to play in real-time, just like using model-free reinforcement learning techniques, but with characteristics of exploiting the data generated by UCT-planning agents.

The first agent, the one using UCT in MCTS offline, needs no training data. It is kept running multiple simulations ignoring the constraint of time. This way is possible to obtain the best possible values for a policy to train the convolutional neural network. It was used the value of 300 as maximum-depth and 100000 simulations, it can be expected the results would improve if bigger values were passed, but probably would not make a significant difference in the outcome, and it would take a lot more time.

#### Combining UCT-based RL with DL

Guo et al. [9], experimented with three different methods on how to convert the results obtained from the UCT agent to the deep learning part. Training by regression, by classification and by classification-interleaved. With the idea to rival the DQN implementations, the CNN architecture was the same for all the games in the Atari 2600, like the DQN implementations that do not change the architecture or the hyperparameters from game to game.

When using regression to train the CNN, the action-values resulted from the UCT-agent are used, these action-values are then paired to the last four frames of each state, and then used to train the CNN. To train the CNN through classification, the action-values are used to select (greedily) an action. This action is then combined with the last four frames from each state along the trajectory, and then passed to the CNN to use as training. For the training of CNN through classification interleaved the process is different. Only a small amount of training data created by the UCT-agent is passed to the CNN for training (through normal classification). The CNN will then, based on this dataset, create more samples with a 5% of choosing a random action to garantee exploration.

The conclusions to keep are the same conclusions obtained in the papers that used MCTS in real-time. While the model-based techniques are much better in terms of results, are impraticable in real-time scenarios. However in combination with deep learning, the use of model-based techniques surpass the

DQN. Especially, in this case, the agent trained with classification, beats the DQN in the seven games that Guo et al. [9] experimented on. Guo et al. [9], also predicted that the difference between the UCT agent, used to create the training data, and the other three agents in terms of input distribution would impact the results. Due to the agent trained with classification interleaved out scoring the other agent trained with simple classification, this seemed as a solution for the mentioned problem.

All these three papers, used a convolutional neural network, in order to analyse the game state and extract information. In the scope of this work, that will not be used, the information will be obtained from messages sent from the framework where the game is being run. However, in the case of this last paper, it showed that it is viable to use a neural network trained with data from a MCTS algorithm.

For the development of this thesis, due to the focus being on a comparison between an agent using MCTS to play Ms. Pac-Man live, versus an agent using a neural network trained with a dataset created by a MCTS, the main research papers used were Pepels et al. [18], for having one the of best agents using MCTS to play the game directly, and Guo et al. [9], for the creation of a neural network trained with a dataset created by MCTS, that was able to play several games from Atari 2600.

# 4

# Implementation

## Contents

This chapter will explain how the agent was developed. Starting on the extraction of the game state from the Ms. Pac-Man framework, passing through the implementation of the MCTS algorithm and concluding in the development of the neural network used to play the game in real-time.

## 4.1 Game State

One of the aspects in analysis on this thesis is the way the agent receives the game state. The approach implemented by Guo et al. [9], used a game state consisting on frames. The agent of this thesis, receives the game state in the form of a string from the framework itself. This string is composed by the attributes in table 5.4, 5.5 and 5.6, with a real example shown in Listing 4.1.

| **Variable** | **Description** |
|---|---|
| Maze Index | Current Maze Index |
| Total Time | How much time has passed |
| Score | Current score |
| CurrentLevelTime | How much time has passed on this level |
| LevelCount | How many levels were completed |
| CurrentNodeIndex | Current Ms. Pac-Man position |
| LastMoveMade | Last move made by Ms. Pac-Man |
| NLivesRemaining | Number of lives remaining |
| HasReceivedExtraLife | If Ms. Pac-Man has received an extra life |

**Table 4.1:** Parameters of Ms.Pac-Man from Game State String

| **Variable** | **Description** |
|---|---|
| CurrentNodeIndex | Current Ghost position |
| EdibleTime | Edible time left |
| LairTime | How much until ghost leaves the lair |
| LastMoveMade | Last move made by ghost |

**Table 4.2:** Parameters of Ghosts from Game State String

| **Variable** | **Description** |
|---|---|
| Pill String | A string of 0's and 1's, if the pill exists the number will be 1, otherwise it is 0 |
| PowerPill String | Same as above but for power pills |
| TimeLastGlobalReverse | Last time since the game reversed |
| PacManEaten | Was Ms. Pac-Man eaten |
| GhostEaten (x4) | Was the ghost eaten? For each ghost |
| PillWasTaken | If a pill was eaten |
| PowerPillWasTaken | If a power pill was eaten |

**Table 4.3:** Parameters of Maze from Game State String

**Listing 4.1:** Example of a Game State String

```
0,0,0,0,0,978,LEFT,3,false,1292,0,40,
NEUTRAL,1292,0,60,NEUTRAL,1292,0,80,NEUTRAL,1292,0,100,NEUTRAL,11111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111,
1111,
-1,false,false,false,false,false,false,false
```

With the game state is then possible to run simulations inside the MCTS algorithm, to calculate the best possible action. This will be further explained in the next section.

## 4.2 Monte Carlo Tree Search Algorithm

The agent created has two options, if the agent is not in a junction, it will continue its path forward, otherwise the MCTS algorithm is used.

The implementation of the MCTS algorithm was somewhat straightforward. With four main methods, one central method for the loop and the rest three methods for each step of the algorithm (the selection and expansion methods are done simultaneous).

```
while(time < timeLimit)
    node = expansion(rootNode, gameState)
    reward = simulation(node)
    backpropagation(node, reward)
    nodeChosen = chooseBestChild(rootNode.children)
return nodeChosen.actionMove
```

The code snippet, above, is the central method, that performs the execution of the MCTS algorithm.

### 4.2.1 Selection and Expansion

These two steps are done simultaneous, because in this implementation, the selection is done at the same time as expansion.

The nodes of the tree always represent junctions, the same way as Pepels et al. [18]. When the algorithm starts from the root node, it starts by selecting all the possible moves from the root node, and from there plays a simulated game until each direction reaches a junction. These new junctions reached

will be the children of the root node. This simulation between two junctions also allows to obtain a score for each junction. This way it is possible to select the best possible action between junctions. This selection is achieved through the UCT Equation 2.1, with a value of $C$ of $1/\sqrt{2}$. Starting from the node selected (junction), an expansion is made, beggining the simulation step from the child node chosen.

### 4.2.2 Simulation

The simulation step, plays simulations of the game using the framework, starting from the game state obtained, with the Ms. Pac-Man's actions randomized. During the simulation, each time that Ms. Pac-Man reached a junction, a calculation is made to compute the reward of that path. The Equation used was the following:

$$reward+ = (discountValue^{junctionDepth}) * (CurrentScore - PreviousScore) \tag{4.1}$$

This formula applies a discount factor, due to the furthest junction having less relevancy than those closest to the current game state. When the maximum tree depth allowed is reached, the reward is back propagated. The $CurrentScore$ is then subtracted to the $PreviousScore$, to obtain the score of that path only.

For this step it was considered two options for how to stop the roll-out. Since waiting for a final state (losing all lives or eating all pills) was not a plausible solution, even with the extended time provided to the algorithm. The first option was simply to consider each step of Ms. Pac-Man, an increment of tree depth until the max tree depth was reached. The second option was only to consider a step each time the Ms. Pac-Man reached a junction, meaning a path between junctions will only count as one step to increase the tree depth. In chapter 5, the tests conclude that the second option was the best and more inline with the purpose of the tree, since each node is represented by a junction.

### 4.2.3 Backpropagation

The backpropagation step updates the reward obtained, to each node, starting from the node where the simulation step began, iterating backwards until the root node.

## 4.3 Creation of the Dataset

Two datasets were created, one to train the neural network through classification and the other to train through regression. Both datasets consist on the same features, which are all the elements of the game state string mentioned before. The difference are the labels. While the dataset used for classification has as labels the actions chosen for each game state, the dataset used for regression has as labels the

discounted rewards of each action possible for each game state present on the dataset. It is important to mention, that not all features of the game state ended up being used. This will be explained in Chapter 5.

## 4.4 Neural Network

The purpose of the thesis is to play the game in real time with a neural network. The idea is the same as using the MCTS algorithm, meaning that the agent will only use the neural network to pick an action, when it is on a junction, otherwise it will continue to follow its path, without changing direction. This neural network was created using the Deeplearning4j [1] library.

It reads the dataset files to train itself before the game starts. During the game, it receives a game state and outputs an action. The conditions on how the datasets were created and composition of the neural network, will be further explained in chapter 5, due to the multiple variations of each for the tests conducted. The MCTS agent, that created the datasets, showed promise, obtaining results equaling the agent developed by Pepels et al. [18]. For the Neural Network, the results were lower, but overall showed potential. These results will be further analysed in the following Chapter.

[1] https://deeplearning4j.org/

# 5

# Experimental Evaluation

## Contents

Due to the random factor inserted in the MCTS algorithm, each test consists on the realization of thirty games. This allows for a more general idea of each alteration made to the parameters of the agent, eliminating possible ”lucky” or ”unlucky” games.

The same idea applies to the tests conducted for the neural network, in the initialization of the weights several techniques use randomness, for this reason, each test of the neural networks also requires the execution of thirty games.

With thirty results per test, it was calculated a 95% confidence interval. Due to sometimes the highest average score is not enough to decide if one parameter is better or not.

## 5.1 Monte Carlo Tree Search

Beginning with the MCTS tests. The first test was to decide which method was the best on deciding when to stop the simulation step. Stop the simulation based on how much steps Ms. Pac-main had walked or limiting based on many junctions had been crossed.

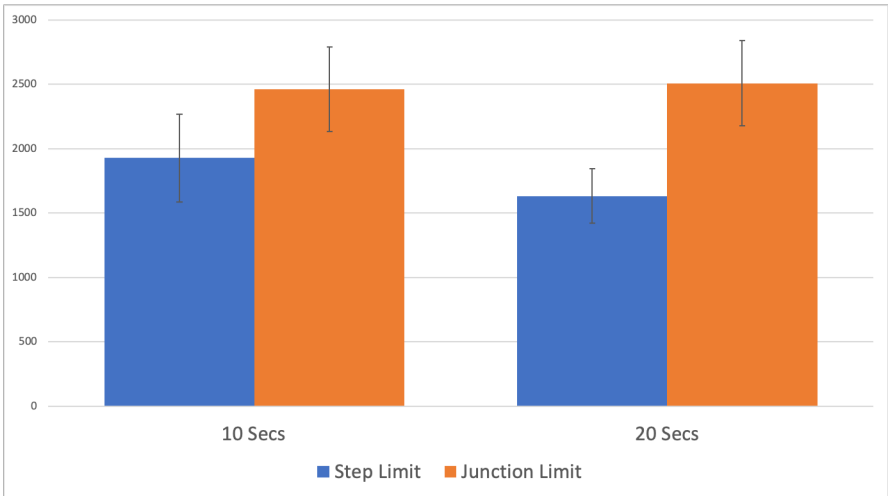


**Figure 5.1:** Steps Limit vs Junction Limit

Through Figure 5.1 it is possible to observe that the junction limit is quite better than the step limit option, in both cases, using 10 seconds to decide an action or 20 seconds. With this, the following tests all use the junction limit method. This can be possibly justified due to the fact that using a junction limit better represents the tree, where each node is a junction.

The following tests try to obtain the best value for the limit of junctions reached during the simulation step and the best discount value for the reward, also during the simulation.

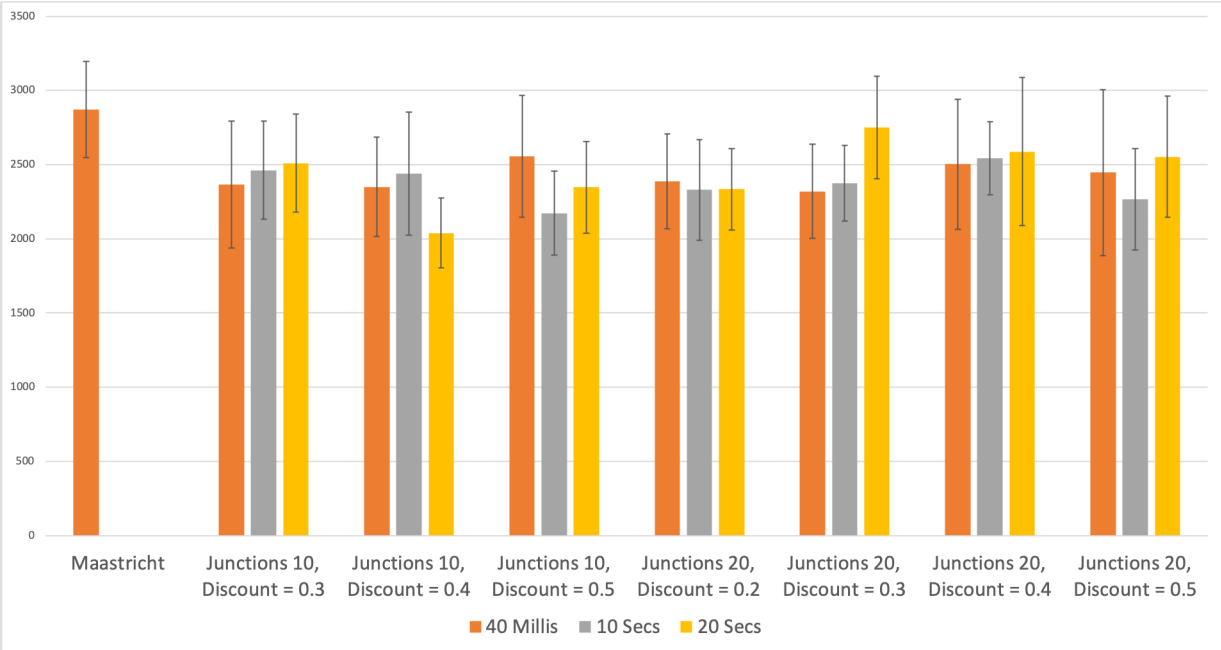


**Figure 5.2:** Variation of Junction Limit and Discount Value

The first column, of the Figure 5.2, represents the agent from Pepels et al. [18]. This agent is the benchmark, to which the thesis' agent is competing against. The following columns represent all the tests realized to obtain the best possible combination of junction limit and discount value. While the confidence intervals here, show a high variance in all the tests, including the control agent, it is possible to observe that agent with a value of 20 junctions as limit and a discount value of 0.3, is the best with the highest average score. Even with more time for our agent to choose an action, it did not equal or surpassed the agent developed by Pepels et al. [18], possibly due to the alterations made to the MCTS algorithm especifically made to increase its performance for Ms. Pac-Man, while our agent used a standard impletamentation of MCTS.

## 5.2 Train Dataset and Neural Network

In order to create the dataset for training, the big problem was the amount of features, since each pill counted as a feature and the dataset only had 2361 instances. The total amount of features was 257. Two sub-datasets were created, one with only 238 features and another with 230.

The neural network used, was the same for both sub-datasets. Using 1 hidden layer, with all layers having the same number of neurons (the number of features) and varying only the activation and loss functions, the output layer had in every single test 4 exits, one for each action.

In the following Tables, the items in red, represent the features that were discarded for the first sub-

dataset, this represent features that will not help the agent choose the better possible action, due to having no impact on the scenario of the game.

This string is composed by:

| Variable | Description |
|---|---|
| Maze Index | Current Maze Index |
| Total Time | How much time has passed |
| Score | Current score |
| CurrentLevelTime | How much time has passed on this level |
| LevelCount | How many levels were completed |
| CurrentNodeIndex | Current Ms. Pac-Man position |
| LastMoveMade | Last move made by Ms. Pac-Man |
| NLivesRemaining | Number of lives remaining |
| HasReceivedExtraLife | If Ms. Pac-Man has received an extra life |

**Table 5.1:** Parameters of Ms.Pac-Man from Game State String

| Variable | Description |
|---|---|
| CurrentNodeIndex | Current Ghost position |
| EdibleTime | Edible time left |
| LairTime | How much until ghost leaves the lair |
| LastMoveMade | Last move made by ghost |

**Table 5.2:** Parameters of Ghosts from Game State String

| Variable | Description |
|---|---|
| Pill String | A string of 0's and 1's, if the pill exists the number will be 1, otherwise it is 0 |
| PowerPill String | Same as above but for power pills |
| TimeLastGlobalReverse | Last time since the game reversed |
| PacManEaten | Was Ms. Pac-Man eaten |
| GhostEaten (x4) | Was the ghost eaten? For each ghost |
| PillWasTaken | If a pill was eaten |
| PowerPillWasTaken | If a power pill was eaten |

**Table 5.3:** Parameters of Maze from Game State String

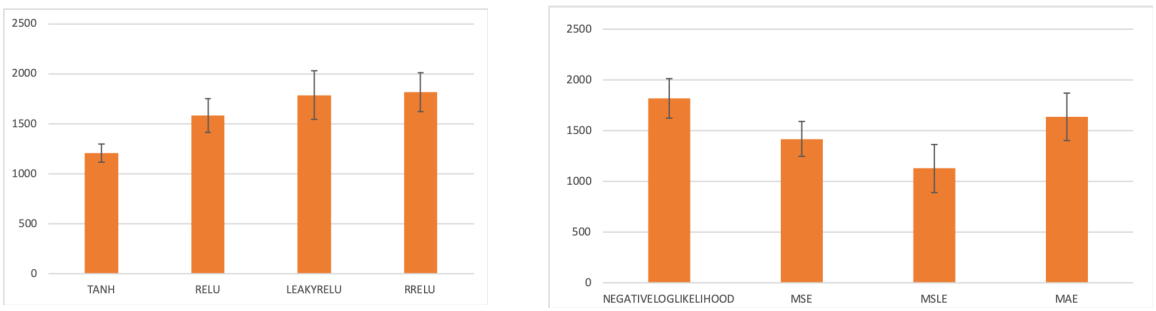


**Figure 5.3:** Comparing Activation and Losses Functions for Sub-Dataset 1

With this sub-dataset the results in Figure 5.4 were obtained. The best activation function, using Negative Log Likelihood as a loss function, was the Random RELU for the hidden layers and Softmax for the output layer. The best loss function being the Negative Log Likelihood, using the Random RELU as activation function with an average score of 1818. While, the results are far below the MCTS impletamentation, it is important to remember that a dataset with 238 features and only 2361 instances, is quite small for a neural network. This limitation is due to the fact that each test ran on the MCTS algorithm, with 20 seconds per action took almost more than 9 hours to complete. It was opted to perform more tests, in order to have a better understanding of the parameters, than running more games of the same test for a bigger dataset.

In the second sub-dataset, even more features were removed. Like in the previous sub-dataset, the red rows, represent the features ignored.

| **Variable** | **Description** |
|---|---|
| Maze Index | Current Maze Index |
| Total Time | How much time has passed |
| Score | Current score |
| CurrentLevelTime | How much time has passed on this level |
| LevelCount | How many levels were completed |
| CurrentNodeIndex | Current Ms. Pac-Man position |
| LastMoveMade | Last move made by Ms. Pac-Man |
| NLivesRemaining | Number of lives remaining |
| HasReceivedExtraLife | If Ms. Pac-Man has received an extra life |

**Table 5.4:** Parameters of Ms.Pac-Man from Game State String

| **Variable** | **Description** |
|---|---|
| CurrentNodeIndex | Current Ghost position |
| EdibleTime | Edible time left |
| LairTime | How much until ghost leaves the lair |
| LastMoveMade | Last move made by ghost |

**Table 5.5:** Parameters of Ghosts from Game State String

| **Variable** | **Description** |
|---|---|
| Pill String | A string of 0's and 1's, if the pill exists the number will be 1, otherwise it is 0 |
| PowerPill String | Same as above but for power pills |
| TimeLastGlobalReverse | Last time since the game reversed |
| PacManEaten | Was Ms. Pac-Man eaten |
| GhostEaten (x4) | Was the ghost eaten? For each ghost |
| PillWasTaken | If a pill was eaten |
| PowerPillWasTaken | If a power pill was eaten |

**Table 5.6:** Parameters of Maze from Game State String

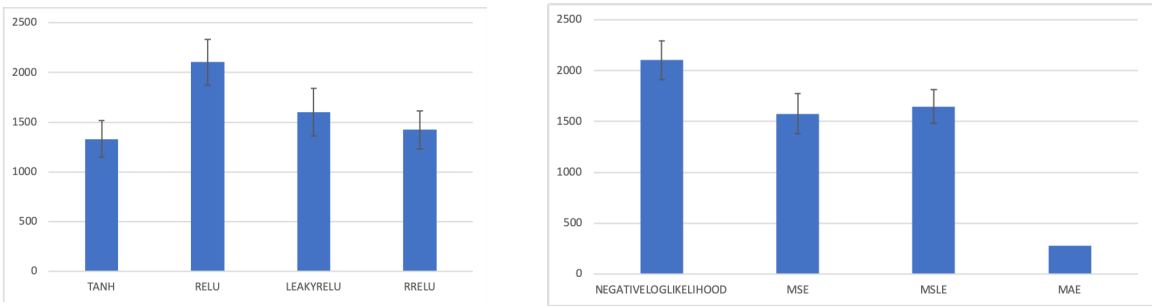


**Figure 5.4:** Comparing Activation and Losses Functions for Sub-Dataset 2

With this sub-dataset even more reduced, in terms of features, the results were somewhat better. The best combination of Relu with Negative Log Likelihood, achieved an average score of 2103 points. To remember that this sub-dataset is exactly equal to the previous one with the exception of features that were not used, as well as the neural network used.

With this version of the sub-dataset being the best, further tests were realized. Varying the amount of hidden layers and neurons per layer. Due to the results not being better than the one presented above, of 2103 average points, these tests will be shown in the Appendix A. These previous tests were conducted training the neural network with classification, which corresponds to using as labels the action to make. The tests for regression only used the second sub-dataset, the one with the smallest amount of features.

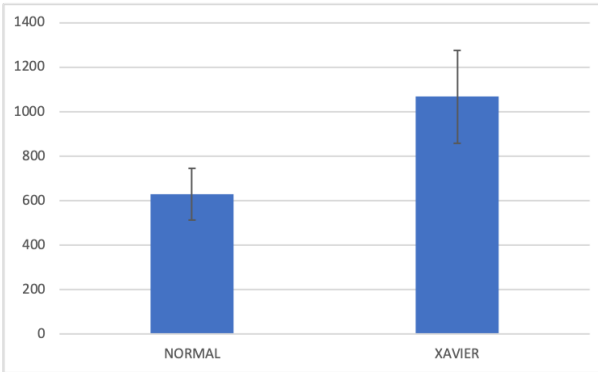


**Figure 5.5:** Comparison between Weight Initializers

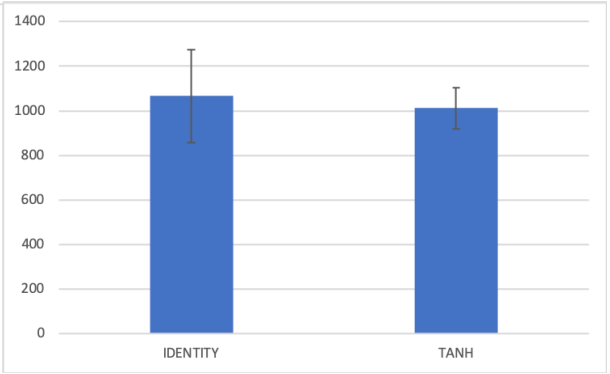


**Figure 5.6:** Comparison between Activation Functions

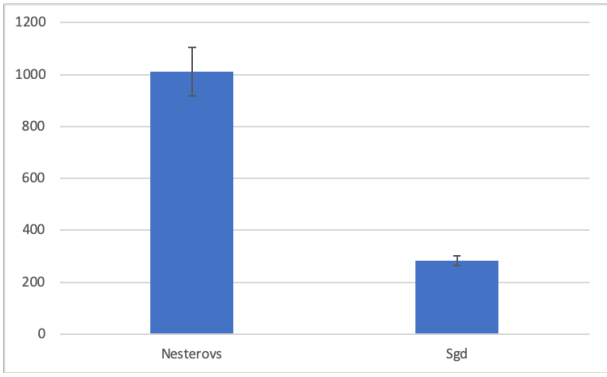


**Figure 5.7:** Comparison between Updaters

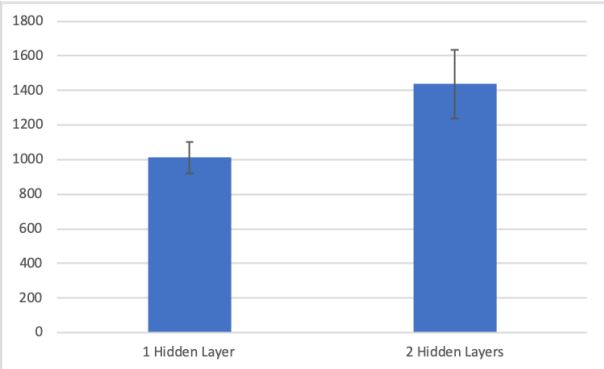


**Figure 5.8:** Comparison between 1 and 2 Hidden Layers

Starting with the Figure 5.5, the first test was to obtain the best way to initilize the weights. This test shows that a Xavier initialization is better than a normal distribution. Figure 5.6, shows a comparision between activation functions. The results indicate that there is almost no difference between these two functions, while the Identity function may have a slightly higher average score than the $tanh$, the difference is almost insignificant. As for the Solvers, Figure 5.7 indicates that the Nesterovs' Momentum is quite better than the conventional Stochastic Gradient Descent. In the end, Figure 5.8, shows that adding a layer can be beneficial. For this test in specific, the number of neurons on each layer was altered. The agent with two hidden layers, had half the number of neurons per layer than the other agent. This was done to try and reduce the complexity of adding another layer. This last neural network, was the best one, out of all the neural networks using regression, with an average score of 1437 points.

To summarize, both neural networks presented results below what was expected, atleast when comparing to the MCTS agent, this can, maybe, be explained due to the small size of both sub-datasets and the huge amount of features. While the classification neural networks managed to pass the 2000 points mark, the best regression neural network not even passed the 1500 points. This can be explained, due to the small amount of samples in the both sub-datasets.

# 6

# Conclusion

## Contents

The idea for this thesis, surged on analysing the papers Pepels et al. [18] and Guo et al. [9]. While Pepels et al. [18] used the MCTS with modifications, in order to play Ms. Pac-Man in real-time competitively, Guo et al. [9] suggested that it is possible to take advantage of a completly normal MCTS implementation and create a dataset to train a neural network to play in real-time. For this thesis, the idea was to analyse if that hypothesis, would be able to go against the agent developed by Pepels et al. [18], who managed to win the european competition, while using a game state obtained from a string instead of a frame like Guo et al. [9].

## 6.1 Conclusions

For the first part, the MCTS agent, the results obtained were very encouraging, when the technique to limit the simulations was altered, from a step based option to a junction limit. The agent even managed to equal the agent developed by Pepels et al. [18], when given 20 seconds to decide an action, with an average score of 2749 points against 2871 points achieved by the Pepels et al. [18] agent, although this agent only had 40 milliseconds to choose an action. Even that when limiting the time per action to the 40 milliseconds used in real-time, the agent was not that far off. The variation in parameters, of both, junction limit and discount value made almost no significant differences in the results obtained.

Passing to the second part of the thesis, the neural networks. Analysing the datasets used, can easily concluded that a dataset with less features, in this case, with few samples, obtained better results. Even looking at the features, one can say that some features like, current level time, level count, etc. will not help the neural network choosing an action. The neural network trained with classification obtained satisfatory results. With the best result being an average of 2103 points after thirty games. While the neural network trained through regression only a managed an average score of 1437 points on the best test.

To conclude, the results may not be ideal, but are encouraging, especially the fact that a MCTS algorithm without modifications can play a game well, given enough time. This makes it a prime candidate, when creating an agent to play different games. The neural network with all things considered did not behave that badly, considering the small dataset used to train both, leaving the door open to the viability of using this technique.

## 6.2 System Limitations and Future Work

For future work, the focus should be on the neural networks. While the MCTS algorithm already proved its value, the neural networks can also be viable. With further time, one could mainly expand the dataset and use feature engineering to reduce the amount of features, while also analising a better architecture

of a neural network that can achieve a good result, playing Ms. Pac-Man in real time.

# Bibliography


[1] Atif M Alhejali and Simon M Lucas. Evolving diverse ms. pac-man playing agents using genetic programming. In *2010 UK Workshop on Computational Intelligence (UKCI)*, pages 1–6. IEEE, 2010.

[2] Atif M Alhejali and Simon M Lucas. Using genetic programming to evolve heuristics for a monte carlo tree search ms pac-man agent. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[3] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[4] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[5] Peter Burrow and Simon M Lucas. Evolution versus temporal difference learning for learning to play ms. pac-man. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 53–60. IEEE, 2009.

[6] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte carlo planning in rts games. In *CIG*. Citeseer, 2005.

[7] Marcus Gallagher and Mark Ledwich. Evolving pac-man players: Can we learn from raw input? In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 282–287. IEEE, 2007.

[8] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[9] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, pages 3338–3346, 2014.

[10] Nozomu Ikehata and Takeshi Ito. Monte-carlo tree search in ms. pac-man. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 39–46. IEEE, 2011.

[11] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[12] Levente Kocsis and Csaba Szepesvári. Discounted ucb. In *2nd PASCAL Challenges Workshop*, volume 2, 2006.

[13] Simon M Lucas. Evolving a neural network location evaluator to play ms. pac-man. In *CIG*. Citeseer, 2005.

[14] Simon M Lucas. Ms pac-man competition. *ACM SIGEVOlution*, 2(4):37–38, 2007.

[15] Carol Luckhart and Keki B Irani. An algorithmic solution of n-person games. In *AAAI*, volume 86, pages 158–162, 1986.

[16] Ian Millington. *AI for Games*. CRC Press, 2019.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[18] Tom Pepels, Mark HM Winands, and Marc Lanctot. Real-time monte carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in games*, 6(3):245–257, 2014.

[19] Kushal Ranjan, Amelia Christensen, and Bernardo Ramos. Recurrent deep q-learning for pac-man. Technical report, Stanford Technical Report, 2016.

[20] David Robles and Simon M Lucas. A simple tree search method for playing ms. pac-man. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 249–255. IEEE, 2009.

[21] Sumit Saha. A comprehensive guide to convolutional neural networks - the eli5 way, Dec 2018.

[22] Spyridon Samothrakis, David Robles, and Simon Lucas. Fast approximate max-n monte carlo tree search for ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2): 142–154, 2011.

[23] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

# A

In the tables $nNodes$ is equal to $(nFeatures + nLabels)/2$

## A.1 Test Results for Multiple Neural Network Architectures using Sub-Dataset 2 with Classification

### A.1.1 Test 1

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | ReLU |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Error | | L2 |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nNodes |
| Hidden Layer #1 | Nodes In | nNodes |
| | Nodes Out | nNodes |
| Output Layer | Loss Function | Negative Log Likelihood |
| | Activation Function | Softmax |
| | Nodes In | nNodes |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 1854 |
| Standard Deviation | 533 |
| Confidence Interval | 191 |

### A.1.2 Test 2

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | ReLU |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Error | | L2 |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nFeatures |
| Hidden Layer #1 | Nodes In | nFeatures |
| | Nodes Out | nFeatures |
| Hidden Layer #2 | Nodes In | nFeatures |
| | Nodes Out | nFeatures |
| Output Layer | Loss Function | Negative Log Likelihood |
| | Activation Function | Softmax |
| | Nodes In | nFeatures |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 1321 |
| Standard Deviation | 463 |
| Confidence Interval | 165 |

### A.1.3 Test 3

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | ReLU |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Error | | L2 |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nNodes |
| Hidden Layer #1 | Nodes In | nNodes |
| | Nodes Out | nNodes |
| Hidden Layer #2 | Nodes In | nNodes |
| | Nodes Out | nNodes |
| Output Layer | Loss Function | Negative Log Likelihood |
| | Activation Function | Softmax |
| | Nodes In | nNodes |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 1654 |
| Standard Deviation | 407 |
| Confidence Interval | 145 |

### A.1.4 Test 4

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | ReLU |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Error | | L2 |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nNodes |
| Hidden Layer #1 | Nodes In | nNodes |
| | Nodes Out | nNodes/2 |
| Hidden Layer #2 | Nodes In | nNodes/2 |
| | Nodes Out | nNodes/3 |
| Output Layer | Loss Function | Negative Log Likelihood |
| | Activation Function | Softmax |
| | Nodes In | nNodes/3 |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 1728 |
| Standard Deviation | 707 |
| Confidence Interval | 253 |

### A.1.5 Test 5

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | ReLU |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Error | | L2 |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nNodes |
| Hidden Layer #1 | Nodes In | nNodes |
| | Nodes Out | nNodes/2 |
| Hidden Layer #2 | Nodes In | nNodes/2 |
| | Nodes Out | nNodes/4 |
| Output Layer | Loss Function | Negative Log Likelihood |
| | Activation Function | Softmax |
| | Nodes In | nNodes/4 |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 2014 |
| Standard Deviation | 549 |
| Confidence Interval | 196 |

### A.1.6 Test 6

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | ReLU |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Error | | L2 |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nNodes/2 |
| Hidden Layer #1 | Nodes In | nNodes/2 |
| | Nodes Out | nNodes/6 |
| Hidden Layer #2 | Nodes In | nNodes/6 |
| | Nodes Out | nNodes/6 |
| Output Layer | Loss Function | Negative Log Likelihood |
| | Activation Function | Softmax |
| | Nodes In | nNodes/6 |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 1922 |
| Standard Deviation | 826 |
| Confidence Interval | 295 |

### A.1.7 Test 7

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | ReLU |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Error | | L2 |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nNodes |
| Hidden Layer #1 | Nodes In | nNodes |
| | Nodes Out | nNodes/2 |
| Hidden Layer #2 | Nodes In | nNodes/2 |
| | Nodes Out | nNodes/4 |
| Hidden Layer #3 | Nodes In | nNodes/4 |
| | Nodes Out | nNodes/6 |
| Output Layer | Loss Function | Negative Log Likelihood |
| | Activation Function | Softmax |
| | Nodes In | nNodes/6 |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 2047 |
| Standard Deviation | 646 |
| Confidence Interval | 231 |

## A.2 Test Results for Multiple Neural Network Architectures using Sub-Dataset 2 with Regression

### A.2.1 Test 1

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | Identity |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nLabels |
| Output Layer | Loss Function | MSE |
| | Activation Function | Identity |
| | Nodes In | nLabels |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 276 |
| Standard Deviation | 36 |
| Confidence Interval | 13 |

### A.2.2 Test 2

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | Identity |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nFeatures |
| Hidden Layer #1 | Nodes In | nFeatures |
| | Nodes Out | nFeatures |
| Output Layer | Loss Function | MSE |
| | Activation Function | Identity |
| | Nodes In | nFeatures |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 787 |
| Standard Deviation | 196 |
| Confidence Interval | 70 |

### A.2.3 Test 3

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | Identity |
| Weight Initialization | | Normal |
| Solver | | Stochastic Gradient Descent |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nNodes |
| Hidden Layer #1 | Nodes In | nNodes |
| | Nodes Out | nNodes |
| Output Layer | Loss Function | MSE |
| | Activation Function | Identity |
| | Nodes In | nNodes |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 290 |
| Standard Deviation | 109 |
| Confidence Interval | 39 |

### A.2.4 Test 4

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | Tanh |
| Weight Initialization | | Xavier |
| Solver | | Nesterovs |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nNodes |
| Hidden Layer #1 | Nodes In | nNodes |
| | Nodes Out | nNodes |
| Hidden Layer #2 | Nodes In | nNodes |
| | Nodes Out | nNodes |
| Output Layer | Loss Function | MSE |
| | Activation Function | Identity |
| | Nodes In | nNodes |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 860 |
| Standard Deviation | 453 |
| Confidence Interval | 162 |

### A.2.5 Test 5

| **Parameters** | | **Values** |
|---|---|---|
| Activation Function | | Tanh |
| Weight Initialization | | Xavier |
| Solver | | Nesterovs |
| Input Layer | Nodes In | nFeatures |
| | Nodes Out | nNodes |
| Hidden Layer #1 | Nodes In | nNodes |
| | Nodes Out | nNodes |
| Hidden Layer #2 | Nodes In | nNodes |
| | Nodes Out | nNodes |
| Hidden Layer #3 | Nodes In | nNodes |
| | Nodes Out | nNodes |
| Output Layer | Loss Function | MSE |
| | Activation Function | Identity |
| | Nodes In | nNodes |
| | Nodes Out | nLabels |

| | |
|---|---|
| Average | 1430 |
| Standard Deviation | 784 |
| Confidence Interval | 280 |