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

João Guerreiro
Instituto Superior Técnico
Universidade de Lisboa
Lisboa, Portugal
joao.pereira.guerreiro@tecnico.ulisboa.pt

***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 is, already, a competition for agents playing Ms. Pac-Man. Most of the first agents back then 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 developed by [1], 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 [1] agent. The moves chosen by the MCTS agent, 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.**

***Index Terms*—Ms. Pac-Man, Monte Carlo Tree Search, Neural Networks**

I. Introduction

Since the early days of Artificial Intelligence (AI) until today, AI has evolved a lot, starting in 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 [2], 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 to AI enthusiasts and scholars that the research community created a competition. In 2007, the first tournament was organized by [3], 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 [4], 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, appeared a technique that uses the MCTS as an auxiliary tool, to create data to train a neural network, developed by [5]. 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 [3].

A. 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 [1]. 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. 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

II. Background

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.

A. Ms. Pac-Man

The game Ms Pac-Man is very similar to the original one, Pac-Man. However, there are some important differences, differences that change everything in the aspect that is important to this thesis.

In a Ms. Pac-Man maze 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. At the bottom, on the right, there are the fruits, that appear throught the maze. 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.

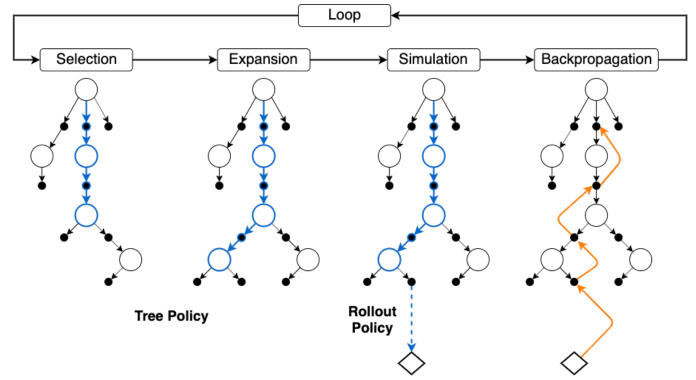


Fig. 1. Monte-Carlo Tree Search Steps

B. Monte-Carlo Tree Search

Monte-Carlo Tree Search, as mentioned by [6], 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.

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.

In Figure 1, 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

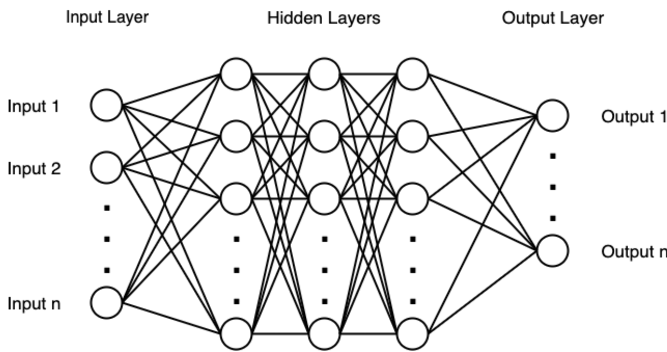


Fig. 2. Example of a Neural Network

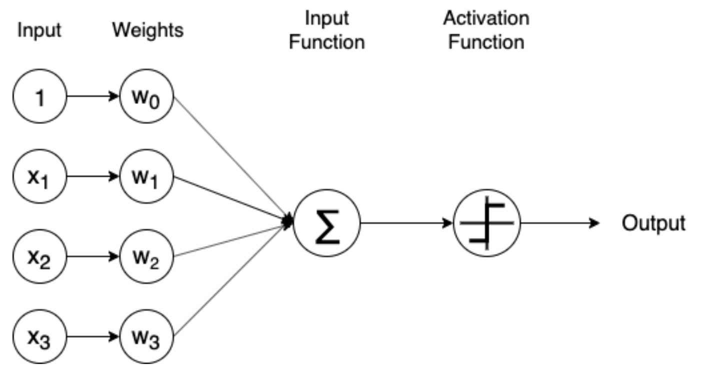


Fig. 3. Example of a Neuron

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.

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 [7], invented the UCT algorithm, derived from a multiarmed bandit algorithm, UCB1 (Upper Confidence Bound) by [8], 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.

C. 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.

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, 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, due to having multiple hidden layers can be called a deep neural network.

Figure 3, 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 0 or 1 or even initialize them following a distribution. For the tests in Chapter 5, it was used a normal distribution to initialize the weights and the Xavier technique, that tries to keep the variance of inputs and outputs in the same layer equal, balancing the weights accordingly. 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 an incredible success in 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 [9].

These 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

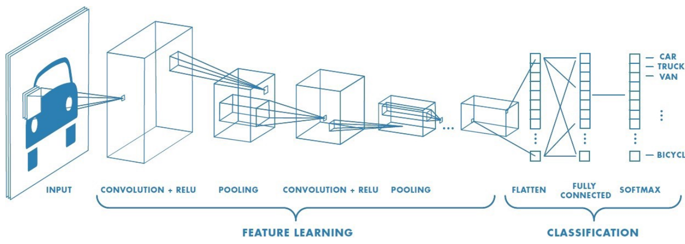


Fig. 4. Example of a convolutional neural network - Adapted from [10]

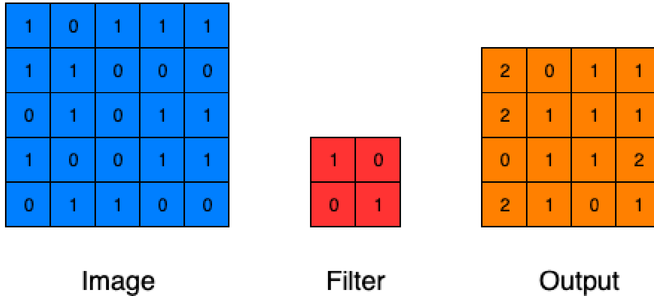


Fig. 5. Example of applying a convolutional filter to an image

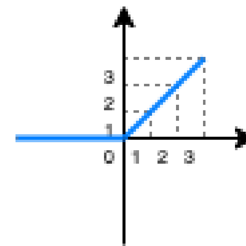


Fig. 6. ReLU Function Plot

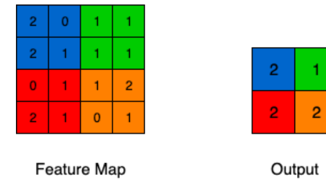


Fig. 7. Example of pooling a feature map

human deciding what features are important, these layers do that themselves.

As observed in Figure 4, 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 5, 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.

Non-Linearity, is a layer that applies a type of non-linear 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.

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 6. For example, a ReLU function applied to the output of the Figure 5, 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 7, 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.

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.

III. Related Work

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. [11] 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, [12] were also able, to use genetic programming to evolve heuristics for a MCTS agent.

[13], 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.

[14], 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.

[15], used evolved neural networks, to evaluate the best action for the agent. [16], 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.

[1], 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, two main papers will be discussed. One written by [1], which developed their agent using real-time MCTS and the second written by [5], that focused on creating their agent using the MCTS offline and then training a neural network with the information retrieved from the MCTS. This two papers were the base of the development for this thesis.

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

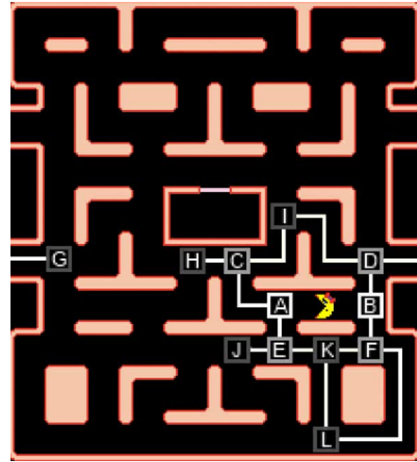
[1], created a controller for Ms. Pac-Man based on the MCTS algorithm that focuses on dealing with problems that rise due to this game being run in a real-time environment. This implementation won the CIG 2012 competition and came in second in WCCI 2012. This paper proposed four improvements, in relation 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 [17], 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 [1] 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. [1], 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 8, the distance between junction "F" and "L" is very different from the distance between "F" and "K".

The other papers, limit the depth of the tree by counting the number of nodes. [1], does it by a measure of distance in the



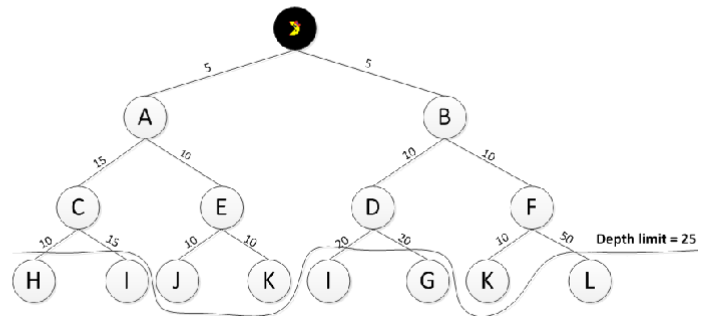


Fig. 8. Game state, and respective mapping in a tree - Adapted from [1]

maze. That is why, in Figure 8, 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 [17], 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.

[1] 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 [17], 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 1.

$$v_i = \begin{cases} M^i_{ghost} \times M^i_{survival} & \text{if tactic is pursuit} \\ M^i_{pellet} \times M^i_{survival} & \text{if tactic is feeding} \\ M^i_{survival} & \text{if tactic is survival} \end{cases} \quad (1)$$

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.

The main difference between [17] and [1], is the use of heuristics versus math. While [1], uses the value obtained as a reward, [17] 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, [1] 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.

[1], 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, [1]:

- 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. [1], achieves this by multiplying all the values stored at each node by a factor of λ at the beggining of each turn. Other methods of decaying, have already been proposed for UCB in multi-armed bandit problems, by [18].

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: [1], adds another modification upon the work of [17]: 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. [1], 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.

B. Deep Learning with Monte-Carlo Tree Search Planning

[5], goal was to surpass the recent innovation that was the DQN.

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.

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 I
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 II
PARAMETERS OF GHOSTS FROM GAME STATE STRING

IV. IMPLEMENTATION

This section 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.

A. Game State

One of the aspects in analysis on this thesis is the way the agent receives the game state. The approach implemented by [5], 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. The Tables I, II and III represent the game parameters obtained from the string. An example of a game string can be seen on Listing 1.

Listing 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,
111111111111111111111111111111111111
111111111111111111111111111111111111
111111111111111111111111111111111111
111111111111111111111111111111111111
111111111111111111111111111111111111
111111111111111111111111111111111111
1111,1111,-1,false,false,false,false,
false,false,false
```

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 III
PARAMETERS OF MAZE FROM GAME STATE STRING

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.

B. 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.

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 [1]. 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 **??**, 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.

2) Simulation: The simulation step, plays simulations of the game, 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:

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

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 V, 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.

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.

C. Creation of the Dataset

It was created two datasets, 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 above. 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 V.

D. Neural Network

The purpose of the thesis is to play the game in real time with a neural network. The base 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 chaning 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 V, 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 [1]. For the Neural Network, the results were lower, but overall showed potential. These results will be further analysed in the following Chapter.

V. Experimental Evaluation

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.

[1] https://deeplearning4j.org/

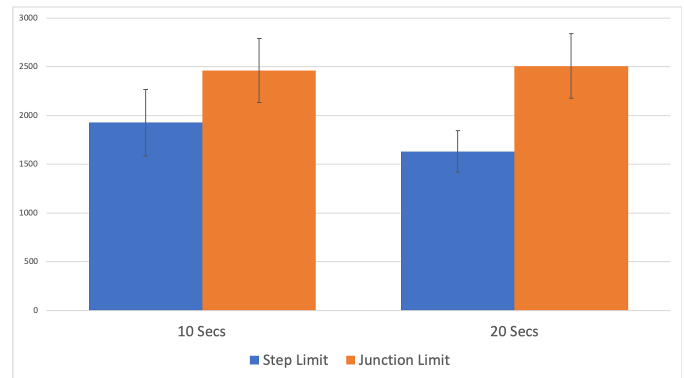


Fig. 9. Steps Limit vs Junction Limit

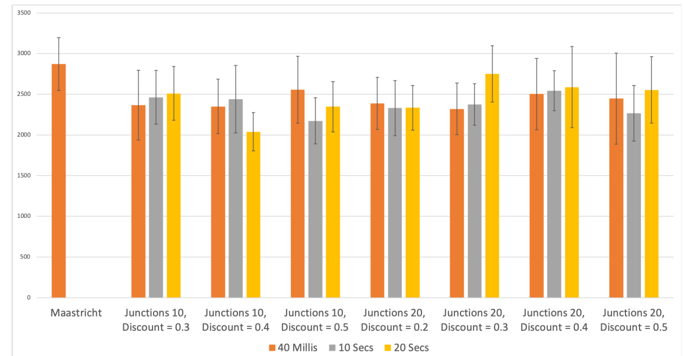


Fig. 10. Variation of Junction Limit and Discount Value

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 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.

A. 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.

Through Figure 9 it is possible to observe that the junction limit is quite better than the step limit option. 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.

The first column, of the Figure 10, represents the agent from [1]. 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 [1], 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.

B. 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. 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.

For the first sub-dataset, 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 with 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 and 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. The results however were not better than the one presented above, of 2103 average points. 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.

For the neural network trained through regression, the best architecture was a Xavier initialization, with Nesterovs as a solver and two hidden layers. Although this configuration still only managed 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.

VI. Conclusions

The idea for this thesis, surged on analysing the papers [1] and [5]. While [1] used the MCTS with modifications, in order to play Ms. Pac-Man in real-time competitively, [5] 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 [1], who managed to win the european competition, while using a game state obtained from a string instead of a frame like [5].

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 [1], when given 20 seconds to decide an action, with an average score of 2749 points against 2871 points achieved by the [1] 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 them, leaving the door open to the viability of using this technique.

A. 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.

References

[1] T. Pepels, M. H. Winands, and M. Lanctot, "Real-time monte carlo tree search in ms pac-man," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 6, no. 3, pp. 245–257, 2014.

[2] I. Millington, *AI for Games*. CRC Press, 2019.

[3] S. M. Lucas, "Ms pac-man competition," *ACM SIGEVOlution*, vol. 2, no. 4, pp. 37–38, 2007.

[4] S. Gelly and D. Silver, "Monte-carlo tree search and rapid action value estimation in computer go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.

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

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[7] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[8] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[9] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[10] S. Saha, "A comprehensive guide to convolutional neural networks - the eli5 way," Dec 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[11] A. M. Alhejali and S. M. Lucas, "Evolving diverse ms. pac-man playing agents using genetic programming," in *2010 UK Workshop on Computational Intelligence (UKCI)*. IEEE, 2010, pp. 1–6.

[12] ——, "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)*. IEEE, 2013, pp. 1–8.

[13] P. Burrow and S. M. Lucas, "Evolution versus temporal difference learning for learning to play ms. pac-man," in *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE, 2009, pp. 53–60.

[14] D. Robles and S. M. Lucas, "A simple tree search method for playing ms. pac-man," in *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE, 2009, pp. 249–255.

[15] S. M. Lucas, "Evolving a neural network location evaluator to play ms. pac-man." in *CIG*. Citeseer, 2005.

[16] M. Gallagher and M. Ledwich, "Evolving pac-man players: Can we learn from raw input?" in *2007 IEEE Symposium on Computational Intelligence and Games*. IEEE, 2007, pp. 282–287.

[17] N. Ikehata and T. Ito, "Monte-carlo tree search in ms. pac-man," in *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*. IEEE, 2011, pp. 39–46.

[18] L. Kocsis and C. Szepesvári, "Discounted ucb," in *2nd PASCAL Challenges Workshop*, vol. 2, 2006.