Augmenting Tree Search with Neural Networks in an AI Agent for Pac-Man

Student Name: J. Huntbach

Supervisor Name: G. Mertzios

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

Abstract –

Background: AlphaGo Zero used a neural network in combination with Monte-Carlo Tree Search (MCTS) to achieve superhuman performance in the game of Go, and consequently inspired many others to experiment with this approach. Several AI game agents have found success by implementing these methods, but there are many ways in which a neural network can augment a tree search

Aims: The aim of this project is to combine tree search and neural networks in a different way to 'the AlphaGo Zero method', to create an AI agent for Pac-Man. The project investigates whether the resulting agent is any more successful than agents that use tree search or a neural network exclusively, and additionally aims to highlight some variables which can affect the performance of agents using tree search and/or neural networks within a game environment.

Method: AlphaGo Zero used a neural network to evaluate leaves within the tree search, replacing the simulation phase of MCTS. This project will instead use a neural network to control the gameplay within each playout of the search. The scores achieved by this hybrid agent will be compared to the scores achieved when the agent is restricted to using just the search tree or the neural network, to reveal whether or not the hybrid approach is effective. On top of this, the scores of some human players will be used to indicate the overall effectiveness of the agents produced.

Results: The agent was able to achieve an average score of 12447 when using just MCTS, and a score of 3425 when using just the neural network. When using the proposed combination of MCTS and neural network, the agent achieved an average score of 19437 (a 56% improvement on the pure MCTS agent). In contrast, human players were able to achieve scores anywhere in the range of 2380 to 18490.

Conclusions: The proposed approach of using MCTS with a neural network controlling Pac-Man during the simulation phase can indeed increase the performance of an AI agent in Pac-Man, with the proposed design allowing an agent to outperform all of the human players in the trial.

Keywords – MCTS (Monte-Carlo Tree Search), NEAT (Neuro-Evolution of Augmenting Topologies), Reinforcement Learning, Neural Networks, Genetic Algorithms, Pac-Man, Artificial Intelligence (AI)

I INTRODUCTION

The introduction to this paper begins by outlining the rules and objectives of Pac-Man and explaining why the game provides an interesting challenge for AI research. We then discuss the inspiration behind this project and the research question that we aim to address, as well as the project's objectives and deliverables.

A Pac-Man

Pac-Man is a challenging, classic arcade game with a relatively simple set of rules. The player must navigate Pac-Man (the yellow character) around the maze shown in figure 1, collecting small orange dots to increase their score. While doing so, the player must avoid four ghosts, who will chase, and try to eat Pac-Man. If Pac-Man eats one of the four big orange dots, known as 'energisers', then the ghosts become temporarily vulnerable, and Pac-Man may eat them for extra points. After being eaten, a ghost's eyes will return to the centre of the maze for a short time, before exiting to chase Pac-Man once again. Once the entire maze has been cleared of dots, the level is reset and the game continues. The aim of the game is to achieve the highest score possible before losing all three lives.

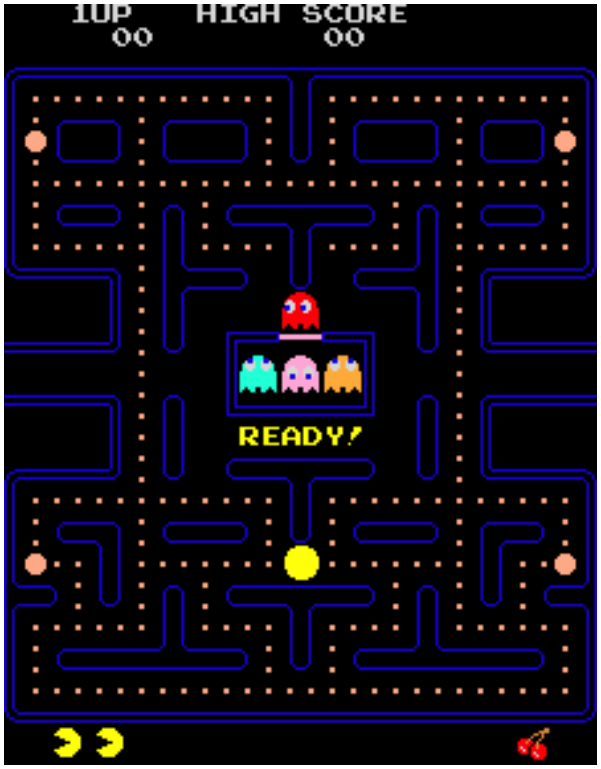

Figure 1: The game screen in Pac-Man. Accessed from: https://en.wikipedia.org/wiki/Pac-Man

Despite the simplicity of the game's objective, complex strategies are required in order to achieve high scores. Additionally, although the ghosts do have predetermined movement strategies, while they are in their vulnerable state they move pseudo-randomly, meaning that a player cannot just memorise a sequence of turns to achieve high scores. The game therefore provides an interesting platform for AI research, and as such, many people have developed their own agents for playing the game. In fact, an annual competition was established in 2007 to determine the best Pac-Man AI. Many different approaches to the challenge have been tried, with the best-performing agents initially being those which used a hand-crated set of rules. However, due to the growing success of MCTS based approaches in other games, agents were inevitably developed for this competition that employed these methods, and since 2011 the most successful agents in these competitions have been those that utilised some form of tree search in their decision making.

B This Project

The work of the DeepMind team has been a significant influence in recent developments in AI. Specifically, AlphaGo Zero (Silver et al. 2016) and the later AlphaZero (Silver et al. 2018) were hugely successful agents which managed to convincingly beat the previous champion players of Go, with AlphaZero also beating the champion chess and shogi agents. These agents made use of a neural network to augment Monte Carlo Tree Search. More precisely, as a replacement of the playout phase of MCTS, a neural network was used to evaluate a leaf node, estimate the probability of the player winning from that position, and provide a vector of move probabilities.

The aim of this project is to implement an AI agent for Pac-Man using methods similar to those used by AlphaGo Zero. However, instead of replacing the playout phase of MCTS with a neural network, this project aims to use the network to control gameplay within the playout phase. The motivation behind this was as follows. AlphaGo Zero uses a neural network to estimate the probability of winning from a given game state, and the training process involved improving this estimate. These probabilities are used to guide the agent's decision making; moves should be taken that lead to the leaves of the MCTS tree which have a greater estimated probability of winning the game. However, in Pac-Man, an agent cannot win the game, so the network's purpose no longer makes sense. Additionally, Pac-Man provides its own performance measure, namely the in-game score, and as such the agent can simply make decisions that lead it towards the game state discovered with the greatest score. A neural network could still be used in a similar way to the one in AlphaGo Zero, by estimating the final score achieved from a given game state, instead of a probability for a winning/losing. However, this is a much more complex problem than estimating a binary condition and has a greater range for error. This is why we decided to try using a neural network differently, i.e. using it to influence the playout phase of MCTS. Thus, we have a research question:

Can a Neural Network be used to enhance the playout phase of Monte Carlo Tree Search, resulting in an effective game agent?

C Objectives / Deliverables

Before development could begin on an AI agent, this project required a copy of the Pac-Man game with which the agent can interact. With this in place, an MCTS framework and a neural network had to be created, each allowing an agent to select moves and play the game. What remained was to develop a third agent, which uses the Monte Carlo Tree Search, but hands control over to the neural network during the playout phase. Implementing all of the above provides a minimum deliverable: an AI agent that can play the game of Pac-Man, using a neural network in addition to Monte Carlo Tree Search.

For an intermediate objective, the aim is for the AI agent to be able to collect all of the dots on the first level of the game within three lives. To achieve this goal, we will experiment with different input sets, architectures, and training methods for the neural network. Additionally, there are several ways in which we can alter the MCTS algorithm. Section IV outlines the results of this experimentation.

The advanced objective for this project is for the AI to be able to score more points in the game than this project's author. More experimentation will be required for the agent to achieve such high scores. For example, while the obvious approach is to combine the best-performing tree search agent with the best neural network agent, it may be the case that a neural network

which does not perform as well on its own results in a better agent when combined with the MCTS. Additionally, a menu should be added to the game, allowing the user to select different options without having to edit the code. The menu will allow a game to be played by a human or the AI agent, as well as including an option to train a new neural network. The two components of the AI agent (tree search and neural network) should also be able to be toggled on/off independently.

II RELATED WORK

This section provides an overview of existing work related to this project, including a detailed explanation of the Monte-Carlo Tree Search algorithm and a brief discussion regarding how MCTS can be utilised within the Pac-Man game environment. We then outline some of the different approaches that can be used for neural networks in Pac-Man and highlight that there exists more than one way of training these networks.

A Monte-Carlo Tree Search

In 1997, Deep Blue (Campbell et al. 2002) became the first computer chess-playing system to win a chess match against a reigning world champion under regular time controls. This system used alpha-beta search, which seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree (where minimax optimises the worst result that could come from an opponent's move). However, Deep Blue was essentially a brute-force player, evaluating 200 million positions per second. When AI researchers turned their attention to the game of Go, which has 5.6 times as many board positions as chess and an average of 5 times as many legal moves per turn, new strategies had to be developed.

Monte Carlo Tree Search is a heuristic search algorithm. Instead of attempting to search the entire space, the algorithm explores the search space with a bias towards moves which are found to produce better results. When applied to gameplay, the search tree is built by iteratively performing the four steps outlined below. While a greater number of iterations of this process leads to a more explored search space and a better tree search, we can terminate the MCTS at any time, at which point the AI selects the move represented by the root node's highest scoring child. Note that each node within the tree represents some game state.

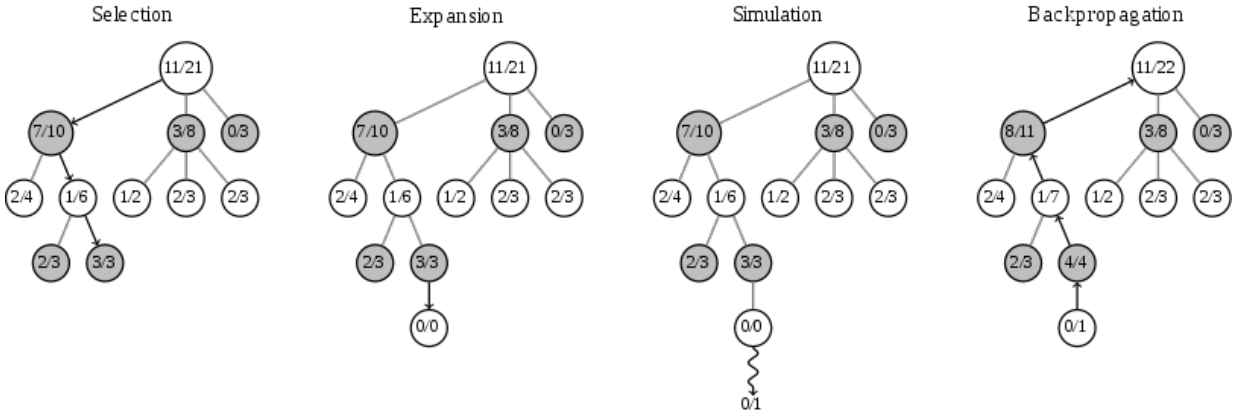

Figure 2: The basic operation of Monte-Carlo Tree Search (Chaslot et al. 2008)

1. **Selection**. Starting at the root node, which represents the current game state, children are recursively chosen according to some selection policy. When a leaf node is reached that does not represent a terminal (game ending) state, it is selected for expansion.

2. **Expansion**. The selected leaf node is expanded. Children are added to the node, with each child representing a legal move from the game state encoded by the leaf.

3. **Simulation**. Starting from one of the child nodes expanded in the previous stage, a playout is run to completion; this means simulating game play until either the game ends, or some other termination criteria is met (such as a limit on the number of moves taken). In pure MCTS, moves are selected at random during this playout, however, some heuristic is often employed here to improve performance. This project aims to utilise a neural network to select these moves.

4. **Backpropagation**. Each node maintains a visit count, as well some measure of the success of playouts that follow from that node. The result of the simulated playout is propagated up the tree, updating these statistics.

As mentioned earlier, many of the best performing Pac-Man agents currently use this search algorithm, and Ikehata & Ito (2011) proposed the following framework. During the selection phase of the search, UCT (Upper Confidence Bound applied to Trees) is used, as formulated by Kocsis & Szepesvári (2006). However, this formula was devised for win/lose games, so to adapt it for use in Pac-Man, Ikehata & Ito (2011) use the following expression, where v_i is the mean score of playouts from the child node, n_p is the visit count of the parent node, and n_i is the visit count of the child node.

$$v_i + c\sqrt{\frac{\ln n_p}{n_i}}$$

For the expansion phase of the search, a child node is added to the leaf for each valid direction in which Pac-Man can move, as shown in figure 3.

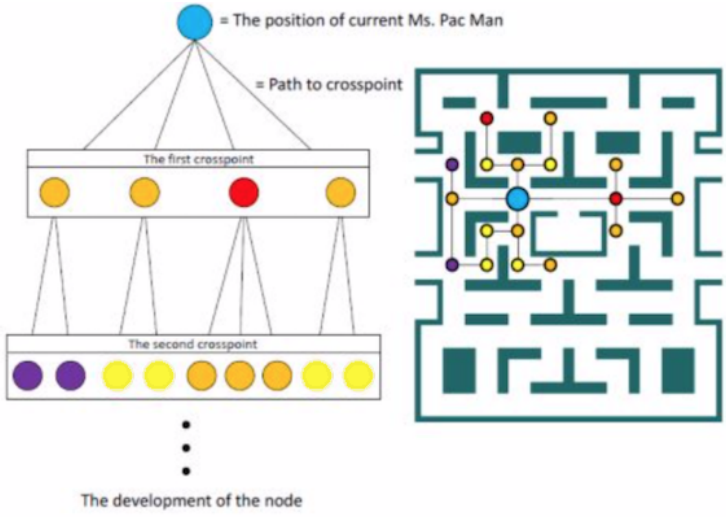

Figure 3: MCTS within a Pac-Man environment. (Ikehata & Ito 2011)

The majority of the rest of Ikehata & Ito's paper (2011) is concerned with heuristics for the simulation phase of Pac-Man, which are not relevant for this project since we will use a neural network instead, but one other non-trivial feature of their agent is the reward mechanism in the backpropagation phase. Instead of updating nodes with the in-game score achieved, the result of the simulation is scored on a scale from 0 to 3. The playout is awarded 1 point if Pac-Man is still alive at the end of the playout. Additionally, the number of dots eaten will be divided by the number of dots that were present at the start of simulation and the number of ghosts eaten will be divided by the number of ghosts active at the start of simulation, providing a total of 2 additional points if all ghosts and all dots are eaten within a single playout. The use of a non-trivial reward mechanism is an interesting one, which we shall experiment with later in the project.

Having discussed the MCTS algorithm in more detail, it should now be clear how neural networks were used in AlphaGo Zero, and what this project's aims are. As a reminder, AlphaGo Zero performs selection and expansion as normal, but then instead of simulating gameplay in the playout phase, they use a neural network to evaluate a leaf node, predict the probability of the game state leading to a win/loss, and use those predictions in the backpropagation phase. This project instead aims to use a neural network to dictate the moves taken during the simulation phase of MCTS. It is therefore pertinent to dedicate some time now to the discussion of neural networks.

B Neural Networks

A neural network is an algorithm, the design of which was inspired by biological brains. The network works through a series of nodes, often arranged in layers, as shown in figure 4. The nodes in the input layer receive values, which propagate throughout the network via connections. Each node takes the weighted sum of its input values (labelled as the 'combination function' in figure 4) passes the result through some function (the 'transfer function', also commonly referred to as an 'activation function') then outputs that result to the next layer of nodes. This process continues, until each node in the output layer has a value.

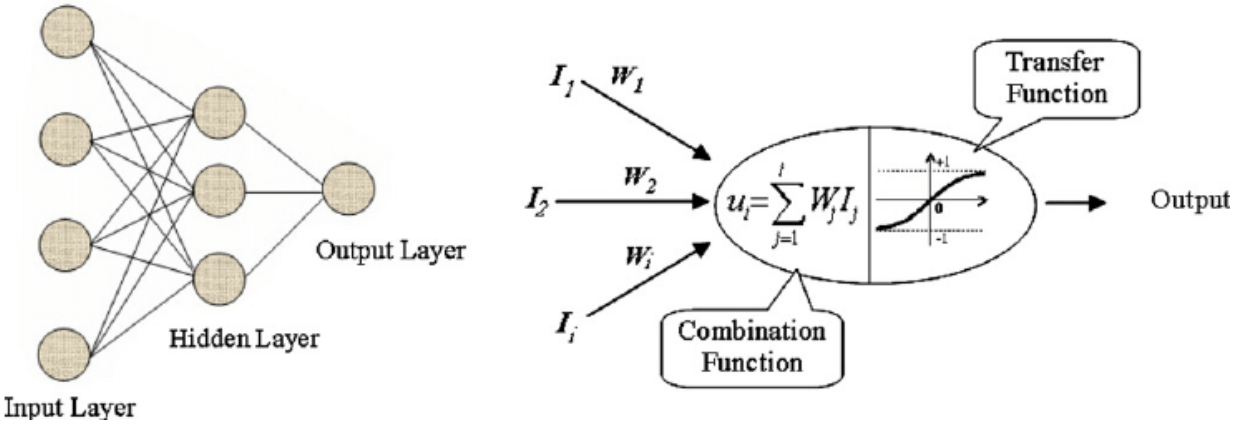

Figure 4: A simple neural network, and the mechanism behind each node (Rezaee et al. 2007).

Neural networks can be trained to produce better outputs; this is done by repeatedly adjusting the weights of each connection, aiming to minimise the difference between the desired output and the actual output.

However, there is a limit to how well a given network structure can perform, regardless of how much the connection weights are altered. To get the best results, one must also experiment with different structures. Not only is there scope for experimentation with the internal structure of the network, i.e. the hidden nodes, but also what inputs/outputs to use. Gallagher & Ledwich (2007) found some success using three N x N windows as the network inputs, centred on Pac-Man, with a window encoding the nearby dots, walls or ghosts. However, this input set is limited in that it only provides Pac-Man with a local view of the level, so if there are no available dots nearby, the agent could struggle to find them within the maze. To mitigate this the window sizes could be increased, but increasing the input size to a neural network increases the training time, as there are more connections to adjust. Additionally, the greater the window size, the less relevant the data on the extremities of the window will be. On top of this, in the case where Pac-Man is in the corner of the maze, three quarters of the window will be outside of the bounds of the play area, leading to a lot of useless inputs. An alternative input scheme is to provide the network with values for different variables, such as the distance to each ghost, binary values indicating the directions in which Pac-Man can turn, etc. An evaluation of different inputs and outputs follows in the results section of this report, but for the internal structure, this project employs neuro-evolution, the details of which are covered in section III:C.

As far as training the network is concerned, there is some evidence to suggest that incremental learning would prove fruitful; this is the process of splitting up the task into smaller stages. For example, the first stage could be training the network to navigate the level, collecting dots, without the ghosts. Once the network can clear the maze of all dots, we would then introduce the ghosts and allow it to learn to avoid them. Finally, the energisers would be added, and the network could learn that it can use these to eat ghosts and increase the score. This training method has been shown in some cases (Elman 1993) to be more successful than trying to train a network on such a complex set of rules all at once. Additionally, this method draws parallels to the neuro-evolutionary approach of evolving a complex network structure gradually, beginning with a simple one. It would also be interesting to try incremental learning in a different order and to compare the networks produced; it may be the case that the network learns more quickly, or is able to achieve a better final performance, if its learns first to avoid the ghosts, and is only then introduced to the dots that it can collect, rather than the other way around. The results of this experimentation are also discussed later.

III SOLUTION

This section presents the design and implementation details of this project's solution. The project lifecycle followed the below stages, so these bullet points will form the subsections for this part of the report.

A) **Pac-Man**: Creating the game.

B) **MCTS**: Developing an MCTS framework able to control Pac-Man.

C) **NEAT**: Implementing a genetic algorithm for neural networks.

D) **Interfacing**: Combining the tree search with the neural network.

E) **Finalising**: Adding a menu, testing, verification and validation.

Whilst there were some open source versions of the game available online, none were of very high quality. Additionally, when creating an AI agent, it is beneficial to understand exactly how the game works, and developing a copy exclusively for use in this project provided useful experience. It also allowed much more freedom in deciding how the agent would interface with the game. The entire project was developed in Java, using the Java FX library for the GUI. The game sprites (dots, walls, Pac-Man and the ghosts) were all created specifically for this project in GIMP.

A Pac-Man

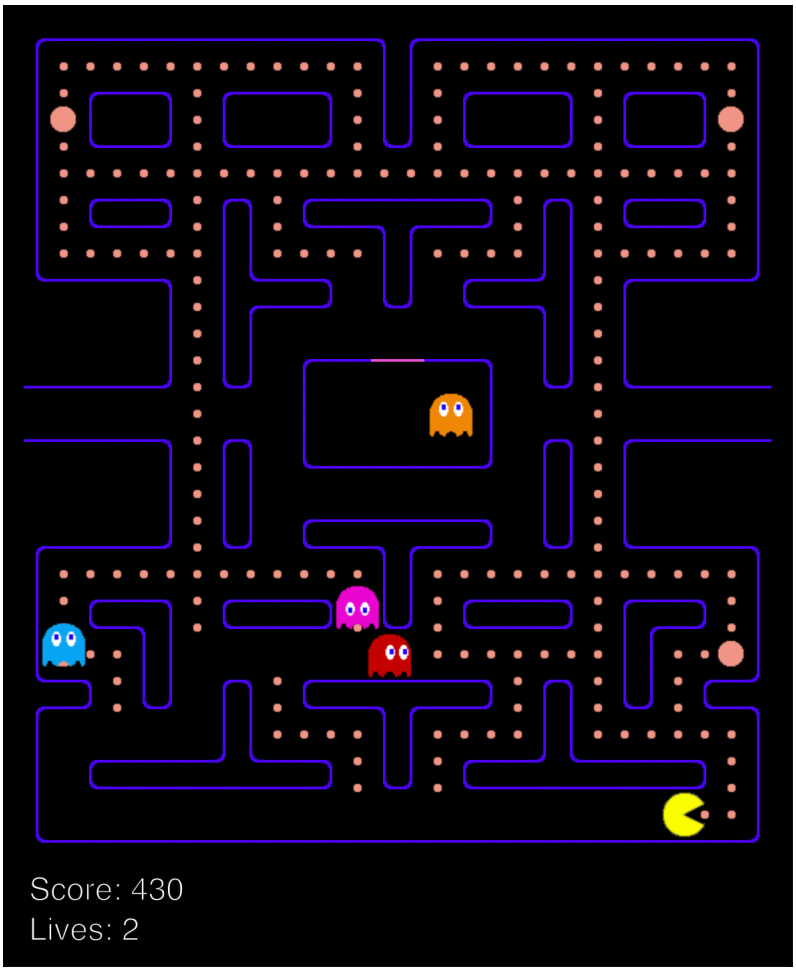

Figure 5: A screenshot of my copy of the game.

There are many subtle details in Pac-Man, and this project's implementation was developed to recreate the original game as accurately as possible. An example of these small details is that each of the ghosts has a different 'personality', in that they target different positions around Pac-Man - this was added to prevent the ghosts chasing after Pac-Man in a single file line, and to keep the gameplay interesting for more advanced players. The red ghost is called Blinky and targets Pac-Man's current position at all times. The pinky ghost is called Pinky and will target the position four game tiles in front of Pac-Man. For some indication of how far this distance is,

note that each dot is centred on its own tile. The blue ghost is called Inky and has more complex behaviour. A vector is drawn between the current position of Blinky and the position two tiles in front of Pac-Man. This vector is then extended to twice its length, with the base still at Blinky's position, and the end is Inky's target. The orange ghost is called Clyde, and targets Pac-Man's position until it is within eight tiles of Pac-Man, at which point Clyde targets the bottom-left corner of the maze.

The path-finding used by the ghosts to reach their target positions is somewhat rudimentary. At each intersection, they query the tile immediately in each of the possible directions they can turn, and move towards the one with the shortest straight line distance to their target. There are some exceptions to the ghosts' normal movement rules, one being that there exist four junctions on the level where a ghost can never turn upwards. Another is that ghosts cannot turn backwards at any time except when switching between movement modes, when they are forced to reverse. The ghosts have two modes, 'scatter' and 'chase'. At the beginning of each level, the ghosts are in scatter mode. In this mode, each ghost targets a specific tile, causing them to retreat to separate corners of the maze. After some time (approximately 7 seconds, although these timings vary depending on the level), the ghosts switch to chase mode, in which they target the tiles as described above. The ghosts alternate between scatter and chase mode at fixed points in each level, entering scatter mode a total of four times. On top of this, the ghosts each enter the level at different times. Blinky always starts the level outside of the ghost house (the box in the middle of the level, which is inaccessible to Pac-Man), whilst the other three ghosts start inside. Pinky is released immediately at the start of each level, Inky is released after 30 dots have been collected, or 4 seconds have passed between collecting any dots, and Clyde is released after an additional 60 dots have been collected, or following another 4-second pause in collecting any dots. The number of dots required to release the ghosts decreases each level, until level four when all ghosts exit the ghost house immediately. The work of Pittman (2009) was instrumental in discovering these intricacies, and producing this version of the game.

It should be mentioned that this game does not include some of the less noticeable features; for instance, in the original game, fruits occasionally appear on the map and award the player bonus points if they can be collected within a short time limit. This would have required more work to implement, and interesting results can be attained without including it. Additionally Pac-Man is supposed to pause for 1/60th of a second after eating each dot, but can turn corners slightly faster than the ghosts. The exact details of these speed differences are fairly convoluted, but balance out to not being that dissimilar compared to not including them at all. Given the time it would have taken to implement these features, they have been excluded from this project.

B MCTS

Section II:A of this paper extensively details how Monte Carlo Tree Search operates, and how it can be used within a Pac-Man environment, so this section will briefly discuss how the algorithm was implemented, and any design choices that had to be made.

Implementing MCTS first requires a Node class; this maintains a visit count, a list of child nodes, a reference back to a parent node (or null if the node is the root of the tree) and a variable to contain the maximum and average scores of all games which have been played out from this node. We also need to implement a TreeSearch class, which contains a pointer to the root node and implements the functionality for each of the four steps that comprise MCTS (selection, expansion, simulation and backpropagation). For the selection policy, we use the UCT equation, as

discussed earlier (Ikehata & Ito 2011), with a minor adjustment, namely that any node encountered with a visit count of 0 is immediately selected. This is to save evaluation time - there are no heuristics to predict the outcome of a playout from a node that has never been visited before, so we should select it immediately for expansion. The details of expansion were discussed earlier, so the next step is simulation. Gameplay resumes, with Pac-Man moving towards the position associated with the selected leaf node. It may be the case that in navigating to the state associated with a leaf node, Pac-Man is caught by a ghost. In this case, the current node in the tree search is marked as leading to death, and this iteration of the tree search is terminated early. Then, in the next selection phase, this node and all of its children can be ignored - we do not want to waste any more time attempting to explore a path which has been confirmed to lead to death. If navigating to the selected leaf node does not result in Pac-Man's capture, the game continues from the leaf node's position with Pac-Man moving in a random direction each time he reaches a junction in the level. Note that this is the stage of the tree search where we aim to replace random movement with a neural network controlling Pac-Man's movement. The simulation stage continues until either a ghost eats Pac-Man, or a maximum number of moves has been made; at this stage, the game state is restored to what it was before the simulation ended.

The final step is backpropagation. From the leaf node selected in the selection phase, we traverse up the tree, incrementing each node's visit count, and adjusting its maximum and average score variables as appropriate.

These steps can be repeated an arbitrary number of times; the number is a tradeoff between better decision making, as a result of a more thoroughly explored search space, and shorter computation time. 20 rounds seem to give a sufficient level of exploration while maintaining a relatively short 'thinking time', but this variable will be revisited later. When the tree search is completed, the AI must decide in which direction Pac-Man should move. To do this, we simply pick the root node's child which has discovered the greatest maximum score and move in the direction towards that node's associated position.

C NEAT

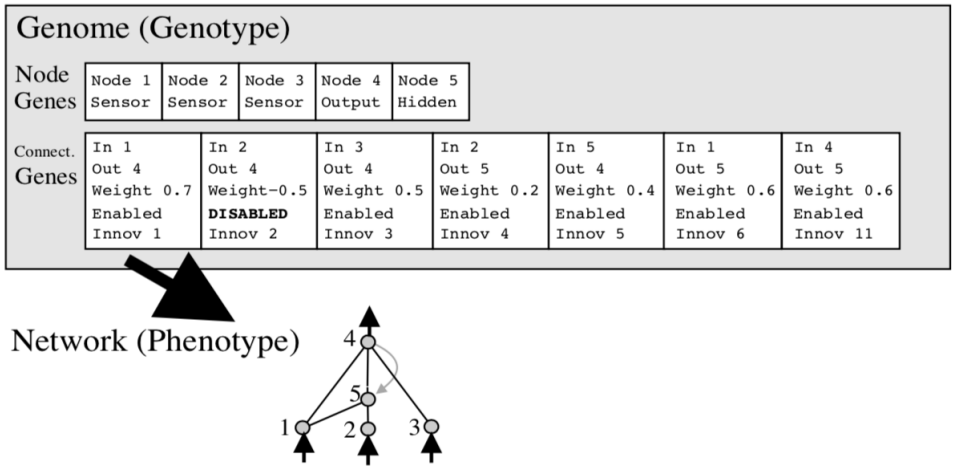

Figure 6: An example of a NEAT genome. The second connection gene is disabled, so the connection that it specifies is not expressed in the network. (Stanley 2004)

Neuro-Evolution of Augmenting Topologies is a genetic algorithm which evolves neural networks (Stanley 2004). Each node and connection within the network is encoded as a gene. Node genes specify what type of node they are (input, output or hidden), and connection genes list an in-node, an out-node, a weight, and whether the connection is currently enabled/disabled. Additionally, each gene contains an ID number, referred to as the 'innovation'. The innovation must be unique for each node and each connection. A genome represents a single network and contains a list of node and connection genes, as shown in figure 6.

The NEAT algorithm describes methods to create new genomes through crossover (producing a new genome from two old ones) and mutation (altering an existing genome). In fact, the genetic encoding scheme described above was designed specifically to allow corresponding genes to be easily lined up during crossover, avoiding the Competing Conventions Problem (Schaffer et al. 1992). To create a new genome, two existing genomes are selected with probability proportional to their evaluated fitness (the score achieved when the network is used to play Pac-Man). All genes present in both parents are passed onto the child genome, with connection weights being randomly chosen from either parent. The genes present in the more fit parent are also passed onto the child genome. If a connection gene is disabled in either parent, then it has some fixed probability of being disabled in the child genome too; this probability is a parameter of NEAT.

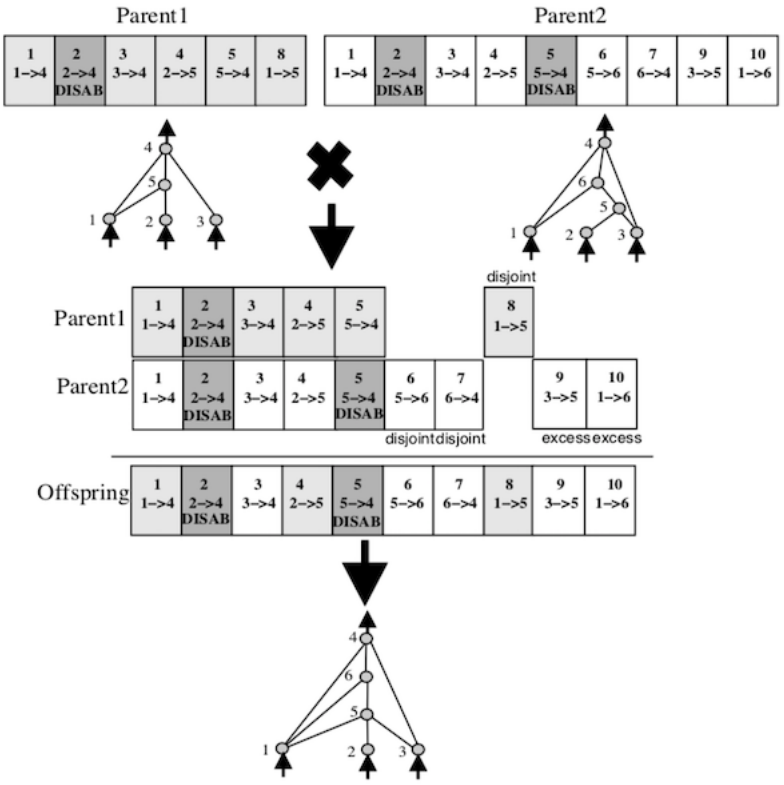

Figure 7: An example of a crossover between two genomes in NEAT. (Stanley 2004)

Once a new genome has been created, it undergoes mutation. There are three different types of mutation within NEAT, and a new genome has different probabilities of undergoing each type.

- Add Connection: Two non-connected nodes are selected, and a connection is added between them with a random weight.

- Add Node: A random connection, connecting node X to node Y, is selected and disabled. A new node, Z, is created, as well as connections from X to Z, and Z to Y.
- Mutate Connection: A random connection is selected. Its weight is either set to a new random value, or multiplied by some random value, according to a fixed probability.

The final idea behind NEAT is protecting innovation through speciation. The idea is to divide the population into species, such that similar network topologies are in the same species, and to carry the fittest member from each species through to the next generation. This allows genomes to compete primarily within their own species, instead of with the population at large, so that topological innovations are protected and have time for their structures to be optimised. We can measure the compatibility distance δ of two genomes, and speciate them using this as a threshold.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \bar{W}$$

In the above equation, E is the number of excess genes (see figure 7), D is the number of disjoint genes, and $\bar{W}$ is the average weight difference of matching connection genes. The coefficients c_1, c_2 and c_3 allow us to adjust the importance of the three factors, and N is the number of genes within the larger of the two genomes.

In each generation, genomes are sequentially placed into species. Each existing species is represented by a random 'mascot' genome, which was present in the species during the previous generation. A given genome g is placed in the first species with which g is compatible with the species' mascot, where two genomes are compatible if their compatibility distance δ is below a fixed value. If g is not compatible with any existing species, a new species is created with g as its mascot. Over time, it may be that case that the population becomes full of discreet species, each of which is protected, and no further evolution can take place. To avoid this, we introduce a stagnation counter. If the maximum fitness of a species does not improve in 15 generations, then the species, and all of its members, are removed from the population. Additionally, when the number of species is greater that 15% of the population size, we increase δ, and if the number of species is one, we decrease δ. This helps to ensure that there are enough species to protect innovation within the population, but not so many that no new mutations are experimented with.

Having implemented these details, we can now create a minimal genome, then apply the NEAT algorithm to evolve a network structure over many generations. However, the structure of a 'minimal genome' is not completely intuitive. One obvious approach would be to initialise a fully connected network with no hidden layers, and allow a complex structure to form over time. On the other hand, Whiteson et al. (2005) found that it's often beneficial to begin with networks that have no initial connections. To understand how this could be effective, we can use an example from Pac-Man; if one input were to represent whether there are dots to be collected directly below Pac-Man, this node only really needs to be connected to the output which corresponds to Pac-Man moving down. Of course, with a fully connected network, the other connections may eventually evolve to have very low weights, or even be disabled all together, but in this will only result in an increase in the time required to train an efficient agent, particularly in larger networks.

Regarding the input/outputs for the network, a few different ideas were tried. A discussion of what worked well, and what didn't, will follow in the results section of this paper.

D Interfacing

This stage of the project ensured that the two - up until this point separate - components (MCTS and NN) were able to operate together, as initially planned. The MCTS code was already in place, but using a random move each time frame during the simulation phase. Additionally, methods had already been created for initialising a neural network from a given genome, getting the required inputs, passing these inputs to the network and calculating the outputs, from which a control for Pac-Man could be extracted. All that this stage required was to initialise a neural network at the beginning of the game, and query the network for a decision instead of getting a random move during MCTS playout.

To enable the use of a previously trained genome in the game, a custom file format was created, storing information about each connection within the network. Specifically, connections are written to their own lines, with vertical bars separating each variable (in node ID, out node ID, connection weight, is the connection expressed, and connection ID). At the end of a NEAT training session, the genome with the highest score is written to the file. Note: while training, this project also catches signal interrupts, so that the best genome can be saved even if the training process is terminated prematurely. To recreate a network, we instantiate a minimal genome as if we were about to commence training; this is for the sole purpose of initialising the input and output nodes. We then read the genome file, adding connections between nodes as we find them. When we come across a new node ID, we can simply create a new hidden node. This process exactly recreates the saved network.

E Finalising

By this stage of the project, all of the required functionality had been implemented, but some useful features were added to increase the ease of use of software. A menu screen was created with options for playing the game as a human, allowing the AI agent to play the game, or training a new genome from scratch. Additionally, an options menu allowed the MCTS and NN parts of the agent to be turned on/off independently, for the purposes of later testing, and also allowed the ghosts, dots and energisers to be turned on/off, so that we could attempt incremental learning during NEAT. Of course, all of these things could be changed manually in the code, but the menu was a nice final touch for the project, and would make it easier to collect some final results.

Testing: While developing the game, manual unit testing was used each time a new feature was added. For example, one of the first implemented methods was called canMove; the method takes a string and returns a boolean value to indicate if Pac-Man can move in this direction. To test this method worked properly, Pac-Man was moved to several different locations of the maze, and the control flow of this function was analysed to ensure that it worked as intended. However, as the number of methods increased, the testing approach depended more on integration testing. Most bugs typically presented themselves very quickly when trying to play a game, and a great many games have been played, both by human players when testing, and later by the AI when experimenting and collecting results.

Verification & Validation: The requirements and specifications of this project's software were only brief; a Pac-Man game needed to be produced, and an AI agent had to be developed that was able to play it. The validation process was reasonably straightforward, but the design report and project brief were frequently consulted to ensure that the software met the needs of this project. The primary concern here was that the game was developed in a way that accommodated

simple interaction with an AI agent. To meet this criterion, every game loop, before updating the game state, a method is queried to get the next control input. When a human is playing the game, this method gets the last arrow key pressed, and returns the appropriate direction. When an AI agent is controlling the game, this method either performs the required MCTS steps or queries a neural network, as appropriate.

IV RESULTS

As mentioned earlier, the aim of this project was to experiment with different ways of implementing MCTS and neural networks within a Pac-Man environment, as well as to combine the two approaches and evaluate the hybrid agent's performance. This section will begin with an analysis of different MCTS approaches, revisiting ideas mentioned in sections II:A and III:B. Once we have established what works well for MCTS, we will evaluate different neural network architectures and training methods. Having discussed both MCTS and NNs in detail, we shall finally combine the approaches and compare the final agent to the results already obtained. Additionally, some human players' scores will be included to provide a measure of how these agents compare.

Given that we have a lot of different approaches to experiment with, each agent will be allowed to play just three games, and the average scores will be used as a comparison. The human players will also be allowed to play three games, but their maximum scores will be used, as a human player takes some time to understand the rules of the game, and adjust to the controls, whereas a computer does not require this time to 'warm up'.

A MCTS

When describing a new approach for the tree search, square brackets will indicate which row in Table 1 is currently being discussed.

The initial MCTS implemented in this project was as described in previous sections. Once the search tree had been built and a move needed to be taken, the child node of the root element which had found the greatest score in a playout was selected, with the score simply being the in-game score achieved [1]. The first experiment was instead selecting the child node with the highest average score of all playouts [2], which resulted in better performance.

Next, it was noticed that if the tree search is unable to reach any dots in a playout, Pac-Man sometimes moves straight towards a ghost. To reduce the likelihood of this happening, a penalty of -500 points was added to the score if the playout ended with Pac-Man being eaten [3,4]. In the event where every playout ends in death this will have no effect, but if even one playout ends with Pac-Man being alive, despite having collected no dots, this path should now be taken. Indeed, this typically increased the score achieved, as the agent will try to evade the ghosts for longer, and end up moving towards the remaining available dots on the level.

Pepels et al. (2014) suggested that when expanding a node, its children should not include its parent, preventing backtracking. They state that if reverse moves are expanded, more simulations per move are required in order to have conclusive differences in rewards, so not allowing them allows more nodes to be expanded, and reduces the amount of similar paths in the tree [5-8].

Finally, Ikehata & Ito's (2011) bespoke reward scheme for the playouts was tested, although this did not provide good results [9, 10]. An explanation for this may be is that they used this system as part of a greater framework. In their approach, Pac-Man was assigned different strate-

gies, depending on the current state of the game, and their reward equation was actually changed according to the current strategy. This may indeed result in the given reward system producing good results, but in the case of this project, it was not so.

From these experiments, we find the best MCTS framework to be significantly better than the all others. Disabling backtracking, as proposed by Pepels et al. (2014), using the average score instead of the maximum score for decision making, and using the in-game score as the reward system seems to be the best configuration within this environment.

Table 1: Scores achieved by MCTS agents

Score Used	Reward System	Backtracking	Game 1	Game 2	Game 3	Average
Max	In-game score	Allowed	3320	3400	7940	4887
Average	In-game score	Allowed	6340	7280	3710	5777
Max	500pt death penalty	Allowed	6850	6220	6970	6680
Average	500pt death penalty	Allowed	5940	7020	4170	5710
Max	In-game score	Disabled	7540	8620	4250	6803
Average	**In-game score**	**Disabled**	**12240**	**11310**	**13790**	**12447**
Max	500pt death penalty	Disabled	9930	7636	5800	7789
Average	500pt death penalty	Disabled	13980	8960	2890	8610
Max	Different rewards	Disabled	2900	3670	3230	3267
Average	Different rewards	Disabled	3160	3000	2740	2967

B Neural Networks

In designing a neural network, three different input sets were evaluated.

1. A vector of 32 different values, encoding normalised values for: the distance and direction to each of the four ghosts; whether or not each ghost is edible, and if the ghost is moving towards Pac-Man; distance and direction to the closest dot; distance and direction to the closest energiser; Pac-Man's current position on the screen; the directions that Pac-Man is currently able to move in.

2. A sliding window, centred on Pac-Man, with two channels. One channel contained +1 for each dot and -1 for each wall tile, and the other channel had a +1 for each edible ghost and a -1 for each non-edible ghost, with all other inputs being zero.

3. A simpler input vector with just 12 values, encoding for each of the four directions: if Pac-Man can turn in that direction; whether there is a dot in direct line-of-sight; the distance to the nearest ghost, starting one tile in that direction.

The first input set did not result in any great performance; it was unable to produce a network capable of scoring more than around 1500 points, even after experimenting with different parameters for the NEAT algorithm and different activation functions for the neural network. This was with a population size of 100, and poor results were obtained even when leaving the training process running over night, reaching 200 generations. It may be the case that increasing

the population size and number of generations further would allow this input set to produce a competent agent, but given that the neural network is only a part of this project, the decision was made to try something else.

With the second approach, we initally tried a 13 x 13 window. It stands to reason that the greater the window size, the more information the AI would have, leading to better performance. However, this also means that there are many more possible connections, so the Neuro-Evolution takes longer to find ones which work well. Additionally, the further the input node is from the centre of the window, the less relevant it should be to Pac-Man's decision making, so not only are there lots of potential connections to be made, but to randomly find a configuration that works well takes a long time. Decreasing the window size to combat this results in Pac-Man being able to detect threats from less distance, which is not ideal. A heuristic that was experimented with here was when adding a connection, instead of randomly selecting an input node, sample from a normal distribution so that connections are more likely to be added to inputs near the centre of the window. However, as soon as hidden nodes are added to the network through mutation, this idea does not work. In general, we did not find much success with this network, even when using incremental learning; to accommodate this training technique, the ghosts were disabled and the inputs halved to just encode the relative positions of nearby walls and dots, but the performance was still poor. The best genomes produced were able to achieve scores in the range of 1000-2000, and using incremental training seemed to have no effect on these scores.

The third input set was slightly more successful. The first four values indicate whether Pac-Man can move up, down, left and right, with a 1 if he can, or a -1 if not. The next four values indicate if there is a dot in line-of-sight in each direction, with a 1 if there is, else a 0. Initially, training was performed on just the dots, and the agent seemed to be performing better than the previous methods, so the ghosts were introduced. To give the network some indication of the danger of turning in each direction, it is provided with one more input per direction. If Pac-Man can't move in a direction, this input is again zero; otherwise, a value inversely proportional to the distance to the nearest ghost is used. This was implemented by searching pixel-by-pixel in a straight line until either a ghost or a wall is reached. If a wall is reached, we check if the current position is a corner or an intersection. If it's a corner, then we turn the corner and continue the pixel-wise search. If a junction is reached, A* search is used to find the shortest path to each ghost from the junction. The distances of these paths can then be used to calculate the shortest distance to a ghost. The input value is then 50 divided by the number of pixels to the closest ghost. Additionally, if that ghost is edible, the number is positive, otherwise it is negative. This allows the AI to navigate towards ghosts Pac-Man can eat, but away from ones that will eat Pac-Man.

This third neural network agent was able to perform better, but it still wasn't achieving particularly high scores. Taking inspiration from an online video of a similar agent (CodeBullet 2018), the inputs and outputs were changed so that instead of providing information about each of the four cardinal directions (up, down, left and right), the information is provided relative to Pac-Man's current direction (forwards, backwards, left and right). The theory behind this is that instead of having to learn rules for each direction, for example, if there is wall immediately above Pac-Man, don't try to turn up, if there is a wall immediately to the right of Pac-Man, don't turn right etc., the agent could move much further with a single rule: if there is a wall in front of Pac-Man, don't keep trying to move forwards. This is obviously a simplification for the nuances of neural networks, but the theory seems to hold up, and training immediately began to produce

better results, even for early network generations. This final change was enough that the network was occasionally able to clear the first level of dots, which was deemed enough progress to continue with the project.

C Final Agent

To produce the final agent for this project, the best MCTS agent was combined with the best trained network. Each of these agents' scores are included in Table 2, along with the scores of the agent resulting from their combination.

Table 2: Scores achieved by final agents

Agent	Game 1	Game 2	Game 3	Average
MCTS	12240	11310	13790	12447
Neural Network	3200	2930	4150	3427
Hybrid Agent	16000	15450	26860	19437

V EVALUATION

The results in Table 2 show that the hybrid approach of using MCTS with a neural network controlling Pac-Man during the playout phase can indeed increase the performance of an AI agent in the game, therefore positively answering the research question. In fact, from the games recorded in the results section, we see a 56% improvement in the average score achieved by the AI.

In the introduction of this paper, three levels of aims were outlined. As a minimum deliverable, we set out to develop an AI agent that can play the game of Pac-Man, using a neural network in addition to Monte Carlo Tree Search. For an intermediate objective, the AI agent should be able to collect all of the dots on the first level of the game within three lives, and an advanced objective was for the AI to be able to score more points in the game than this project's author. At the project's conclusion all of these aims have been met. In this respect, the project was a great success. However, there is still much experimentation that could result in improvements to the AI. For example, while the tree search part of the agent can perform well on its own, the neural network achieves much lower scores, and its performance plateaued after very few generations of training. With a better performing neural network, we may see more improvement in the hybrid agent. One idea that could improve the network's performance is adding an element of randomness to the initial game state during training. When training, we noticed that some networks are successful only because their network structure and the initial configuration of the game coincidentally allow them to collect multiple points before being eaten. However, when these networks are used to control Pac-Man from a different starting configuration, as is often the case when trying to play the game, they perform very poorly; they have simply not evolved a structure capable of reacting to the ghosts positions and the maze environment. We postulate therefore that more successful networks could be produced in a bespoke training environment, where the initial positions of the ghosts are randomised, and potentially even the structure of the maze. The networks would have to become much more generalised and dynamic to enjoy

success in this environment. Alternatively, the NEAT algorithm could be replaced entirely with a larger, pre-built neural network. Since starting this project, Dienstknecht (2018) has found some success using a Convolutional Neural Network within a Pac-Man AI.

Despite the limited performance of the neural network, this project was a success. As stated earlier, the objectives were met, and the timeline roughly followed the initial draft that was laid out in the first documents. Towards the end of the epiphany term, the project had to be put on hold due to time pressures for other university coursework, meaning that some experiments for the neural networks still had to be finished over the easter holidays. However, we were able to catch up in time to get some positive results and write them into this dissertation.

It was mentioned earlier that some scores of human players would be included, in order to gauge how the final agent compares; Table 3 shows these results. In contrast to the AI agents, when collecting scores from human players, the final column in the table highlights the highest score from three games rather than the average. This is because it takes time for a human to become accustomed to the rules of the game, and the controls, so their first game is expected to be lower than their skill level, whereas this learning process does not take place with an AI agent. In this table, I, the author of this dissertation, am labelled as player 1. Whilst I am by no means a professional Pac-Man player, I have nonetheless spent a lot of time playing in the past few months, while testing the game and its features, and know exactly how the game is implemented having written it myself, so can - to some extent - predict the motion of the ghosts. My scores therefore reflect the level of a skilled player, while the rest of the scores demonstrate a range of abilities from novice to mid-level. It should be noted that human players have managed to get a perfect score in Pac-Man, meaning that they have played 255 consecutive levels, eating all four ghosts each time they eat an energiser dot, and not losing a single life, resulting in a score of 3333360. Therefore, the average score of 19437 achieved by the AI agent in this project is by no means superhuman. However, it is a higher score than any recorded in Table 3, and very few humans have ever achieved a perfect score. It is thus reasonable to suggest that the final AI produced plays at the level of a skilled human player, albeit not quite a professional.

Table 3: A comparison of human players with the final agent

Player	Game 1	Game 2	Game 3	Final Score
1	3390	18490	17800	18490
2	3590	6110	1980	6110
3	3530	5040	8800	8800
4	2240	4250	2420	4250
5	6140	8830	5100	8830
6	1190	1620	4330	4330
AI	16000	15450	26860	19437

Note: The ‘final score’ column displays the high score of humans, but the average score for the AI.

VI CONCLUSIONS

In this project, a neural network has been used to control the playout stage of a Monte Carlo Tree Search in a Pac-Man AI agent, resulting in an average score increase of 56%. In designing

the tree search, ideas were taken from existing agents (Ikehata & Ito 2011), including the 2012 IEEE CIG Conference 'Pac-Man Versus Ghost Team' competition winner (Pepels et al. 2014). Neural networks were then evolved using NEAT (Stanley 2004) for a few different input and outputs sets. The most successful network was then used within the MCTS search, resulting in the aforementioned improvements.

In the evaluation, it was mentioned that the neural network was not able to achieve very high scores on its own, and some enhancements were proposed in order to improve upon this. To extend the project further, another potential aim could be to reduce the computation time of the MCTS. While the tree search is limited to performing 20 rounds of simulation, and 20 moves per simulation, this is too much computation for the agent to play the game in real time. The number of simulations and the depth of the playouts could be decreased in order to reduce computation time, but this would come at the cost of lower performance. Additionally, the agent could estimate what the game state will be by the time Pac-Man reaches the next junction in the maze and start running MCTS in a different thread, from this game state, so that the game does not have to pause for as long at each junction. On top of this, Pepels et al. (2014) have created some heuristics and alterations to the MCTS algorithm allowing for real-time MCTS within a complex game environment, such as reusing the search tree for several moves with a decay factor, and these ideas could be worked into this agent.

References

Campbell, M., Hoane, A. & hsiung Hsu, F. (2002), 'Deep blue', *Artificial Intelligence* **134**(1), 57 – 83.

Chaslot, G. M. J.-B., Winands, M. H. M., van den Herik, H. J., Uiterwijk, J. W. H. M. & Bouzy, B. (2008), 'Progressive strategies for monte-carlo tree search', *New Mathematics and Natural Computation* **4**, 343–357.

CodeBullet (2018), 'AI learns to play pacman using neat'. Accessed: 22-January-2019. **URL:** *https://www.youtube.com/watch?v=QpyHYRBKy8U*

Dienstknecht, M. (2018), Enhancing Monte Carlo Tree Search by Using Deep Learning Techniques in Video Games, PhD thesis, Department of Data Science and Knowledge Engineering, Maastricht University.

Elman, J. L. (1993), 'Learning and development in neural networks: the importance of starting small', *Cognition* **48**(1), 71 – 99.

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

Ikehata, N. & Ito, T. (2011), Monte-carlo tree search in Ms. Pac-Man, *in* '2011 IEEE Conference on Computational Intelligence and Games (CIG'11)', pp. 39–46.

Kocsis, L. & Szepesvári, C. (2006), Bandit based monte-carlo planning, *in* 'Proceedings of the 17th European Conference on Machine Learning', ECML'06, Springer-Verlag, Berlin, Heidelberg, pp. 282–293.

Pepels, T., Winands, M. H. M. & Lanctot, M. (2014), 'Real-time monte carlo tree search in ms pac-man', *IEEE Transactions on Computational Intelligence and AI in Games* **6**(3), 245–257.

Pittman, J. (2009), The Pac-Man dossier, Technical report, Gamasutra. Accessed: 17-January-2019.
URL: *http://www.gamasutra.com/view/feature/3938/thepacmandossier.php*

Rezaee, R., Kadkhodaie, A. & Alizadeh, P. (2007), 'Intelligent approaches for the synthesis of petrophysical logs', *Journal of Geophysics and Engineering* **5**, 12.

Schaffer, J. D., Whitley, D. & Eshelman, L. J. (1992), Combinations of genetic algorithms and neural networks: a survey of the state of the art, *in* '[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks', pp. 1–37.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. & Hassabis, D. (2016), 'Mastering the game of Go with deep neural networks and tree search', *Nature* **529**(7587), 484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. & Hassabis, D. (2018), 'A general reinforcement learning algorithm that masters chess, shogi, and go through self-play', *Science* **362**(6419), 1140–1144.

Stanley, K. O. (2004), Efficient Evolution of Neural Networks Through Complexification, PhD thesis, Department of Computer Sciences, The University of Texas at Austin.

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R. & Kohl, N. (2005), Automatic feature selection via neuroevolution, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference'.