Evolutionary Computing

1. Introduction

There is a great desire to understand intelligence and apply it, computationally, to various tasks. The field of Artificial Intelligence (AI) is an endeavour to ‘solve’ intelligence, and use it enhance the speed, precision, and effectiveness of human efforts. To create these ‘learning algorithms’, heavy inspiration has been taken from the fields of Mathematics [1], Neuroscience [2] and perhaps most interestingly, human evolutionary history [3].

Games are a perfect testbed for AI algorithms as they act as a rich and noisy source of optimisation problems [4]. Specifically, stochastic elements in a game make it difficult to optimise against. This could be the use of random terrain generation, a roll of dice or unexpected moves from an opponent. Moreover, an AI can play the game autonomously and repeatedly until the goal is achieved, all without human intervention. This, paired with the stochastic nature of games, makes it very appealing to test the performance of new AI strategies on games. If an AI has a high chance of success at winning a game despite the natural stochasticity it faces, it is a good indication that something has been ‘learned’ by the model (AI).

The game used for this report is called Snake. Snake was very popular as it was one of the few free games available on a Nokia phone, at the start of the mass adoption of mobile phones in the 2000s [5]. The player controls a food-hungry snake by traversing a 2D grid and collecting one food item (random spawns) at a time. At each time-step, the snake travels a unit in its current direction and can be controlled to move up/down/left/right. The size of the grid is limited by its surrounding walls (boundary) and colliding into these walls results in a loss. Consuming food increases the length of the snake by a unit, although a loss occurs if the snake turns into itself. Here lies the challenge: a player must avoid all collisions (with wall and the snake itself) and obtain food items with the escalating pressure an increasing snake length brings.

Aim and Objective

The aim is to create an agent that can win the game of Snake. This will be done using an Evolutionary Algorithm (EA). Maximising the score will be the agent’s primary objective. In the best-case scenario, the maximum score would be achieved and in the worst-case, the score would always be 0. The snake can traverse a 14x14 grid space (number of legal moves excluding walls on each side) and is initialised with 11 units. Thus, a maximum score would be equal to 185. Achieving this can be conceptualised visually, whereby the snake would cover the entire grid if successful.

Literature Review

Perhaps due to the natural human curiosity surrounding games and AI, there is a vast amount of research on the subject. The goal of much of this research is to generate adaptive, intelligent, and responsive agents that can ‘win’ a game with human-like or super-human intelligence. The focus of this review will be on a select few papers that provide the most relevance to the report.

The game of Tetris is played on a 2D grid where a tile enters the grid from the top and falls at a specified pace towards the bottom. It can be rotated and moved horizontally to connect to a previous tile(s) already at the bottom of the game grid. The connection scores points if there exists a line that is completely filled with tiles, causing the line to vanish. The objective is to score as many points as possible. Böhm et al. [6] used an EA and reduced this problem to finding a good ‘rating’ function. Each possible game-grid that could be reached using the incoming tile was rated and the best-rated one was taken as the path. This rating involved 10 different factors, the most interesting being finding the sum of all the gaps between the tiles and attempting to minimise that. An EA was used to find the optimal weights for this ‘rating’ function. The fitness function was chosen to be the number of placed tiles. Interestingly, the score was not considered as the researchers found that the number of placed tiles was more proportional to the number of cleared lines than any other factor, and hence gave the best results.

A solution to Pac-Man using an EA was proposed by Tan et al. [7] in 2011. They used a hybrid approach of having evolution strategies (ES) combined with a feed-forward artificial neural network (ANN) generate controls that allow an agent to win the game. The fitness function was the score, and each generation had a population of weights (Pac-Man automated agents) that could then create offspring (based on tournament selection) and mutate with a probability of 0.7. They compared the EA’s performance to a random ANN and used statistical methods (i.e., t-test) to find that their model was effective. It is notable that their relatively simple methodology (basic EA) resulted in such good scores.

For the game of Snake in particular, Yeh et al. [8] used a ‘rating’ function, similar to the one seen in [6]. Factors such as smoothness (turns the snake makes), space (number of reachable positions), and food (impact on other factors if snake paths towards the food) all feed into the aggregate rating function. These weights are then optimised by an EA with a similar procedure used in [7] (described above). This is then fed to an ‘direction control module’ (presumably an ANN) that can execute moves for the snake and receive updated game states in return. The objective is to maximise the score.

Alternative research in this area uses reinforcement learning and deep learning [9] but will not be discussed for the purposes of this report (since only an EA can be used).

Approach used

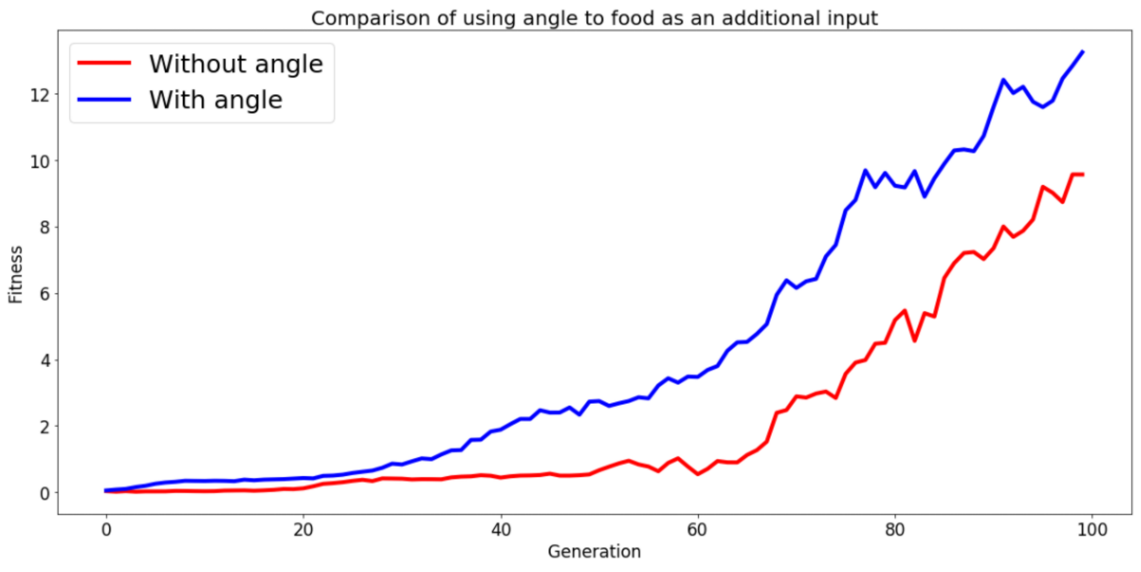
The approach used in the latter half of this report will be an EA that evolves the weights of an ANN. Since ANNs are good at pattern recognition given a lot of data, it is justified to use them in a game with a lot of repeated simulations present. Each state of the game can be thought of as a pattern that the ANN will have to recognise, and then approximate a solution with its general weights (prevent overfitting). However, just this alone will not suffice as it can get stuck in a local optimum (e.g., snake may traverse in a loop). Thus, an EA is used to evolve these weights (produce offspring via selection process and mutate) in order to get the best result possible.

2. Methodology

A neuroevolutionary approach is used to design the model. Conceptually, this made sense as the EA compensated for weaknesses of the ANN, specifically the issue of getting stuck at a local optimum. An alternative approach, Genetic Programming (GP), has already been used many times in different variations to solve Snake and so was not used. It would be more interesting to try a relatively more unexplored method. This is not to say GP is harder to conceptualise. One may argue that as a human who understands biology, identifying why the snake has a particular strategy through the lens of GP may be easier than the pattern-recognition approach taken by an ANN. Nonetheless, neuroevolution was used for reasons described earlier.

The inputs to this ANN are the four directions the snake could take (i.e., up, down, left, or right) and the angle from the snake's head to the food. This essentially encompasses both rating functions (smoothness and food) used in [8], but perhaps in a more succinct and valid way. The intuition here is that the score will increase when this angle is small, since the snake is closer and so the chance of it reaching the food is greater. In turn, to keep the angle small, the snake would zig-zag to the food and then be able to 'straighten' itself out around the edges to prevent any collisions. To test this idea, a small preliminary run was conducted. Graph 1 shows that having the additional input of the angle between the snake and the food produces a suitably higher score, justifying its inclusion to the ANN. The outputs are simply 4 nodes corresponding to what action the snake agent should take next (up, down, left, or right). The failure conditions are if the snake collides with an object (wall or itself), or if it traverses the map for too long without getting food ('too long' defined as 85% of the maps total traversable area).

Graph 1



The fitness function used is final score. This is taken as additional length (starting from initial length = 11) the snake gains by the end of the game. An alternative fitness function may be to minimise the total number of empty spaces in the grid, as used in the Tetris example earlier [6], however this is simply another way to visualise the problem and would not be more beneficial. A worse fitness function would be to maximise the time spent alive, since this may correlate with eventually obtaining food. Nonetheless, the inefficiency of this method was why it was disregarded.

The choice to not evolve the topology of the network was made. Instead, a static topology with evolving weights would be used. The primary reason for not using augmenting topologies is the assumption that the model can perform well without having to incur the high computational cost of topology augmentation. One way to test this assumption is to have 3 different ANN structures and plot their performance on a graph. These can be seen in Table 1 below.

Table 1

ID	No. hidden layers	No. hidden nodes / layer
NN 1	2	5
NN 2	2	15
NN 3	2	30

Note that each variation will be run for 250 generations with a population size of 500. The number of hidden layers was kept constant as it is assumed also that the nature of the task does not require more for it to produce a good solution. Increasing the number of hidden layers risks overfitting and decreasing it may mean that a good function cannot be approximated. If the results show that the model fails to progress relatively early into the cycle, then changing the topology/adding more hidden layers could be justified.

Once this method to determine the best structure for the network is carried out, the best model should be run for a further 250 generations (500 total) using the same population size (500) for fairness. This will be done in the results section below.

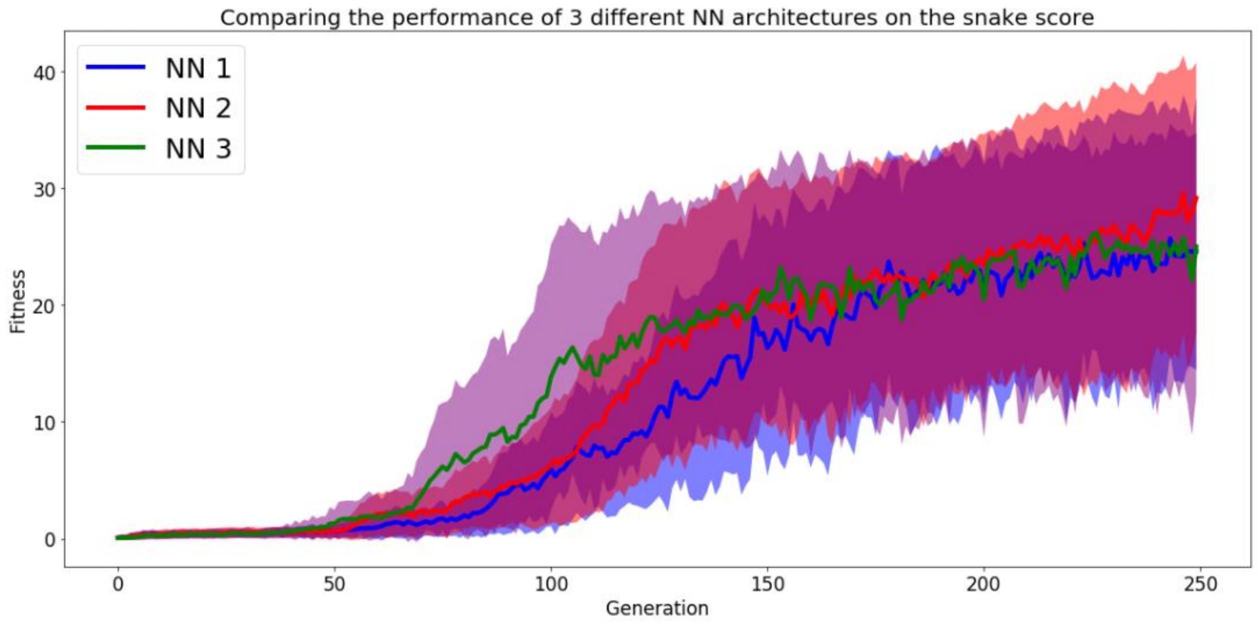
Note that the first step for this algorithm is to generate a population of 500 neural network weights. Genes (each gene is a weight between 2 neurons) are initialised uniformly between [-1, 1] as it is a reasonable spread; they could also be randomly initialised. Then, the weights are loaded into the ANN and used to play the game, one set after another (until all 500 run). Each individual play's 3 times and an average score is taken to avoid stochasticity. Specifically, one member may get lucky if the food spawns next to it, hence the average is used. The fitness value for each member of the population is calculated as the game score. Now, the goal is to pass on the best performing individuals to the next generation. This involves using tournament selection, whereby 3 individuals are paired against each other at random. Those with higher fitness values have a higher chance of winning the tournament and being selected for the next generation. This process is repeated for the population for each generation, so the future generations are stronger than the previous (ideally).

To introduce diversity into the populations, we make use of mutations. Every gene for every individual has an 8% chance to be mutated, which replaces it with a random value from a Gaussian distribution (mean=0, std=0.5). Alternatively, a crossover method was considered to introduce diversity but due to the competing convention problem, this was ruled out and so cloning was used. This was all done in Python using NumPy due to its simplicity, effectiveness, and ease of use.

3. Results

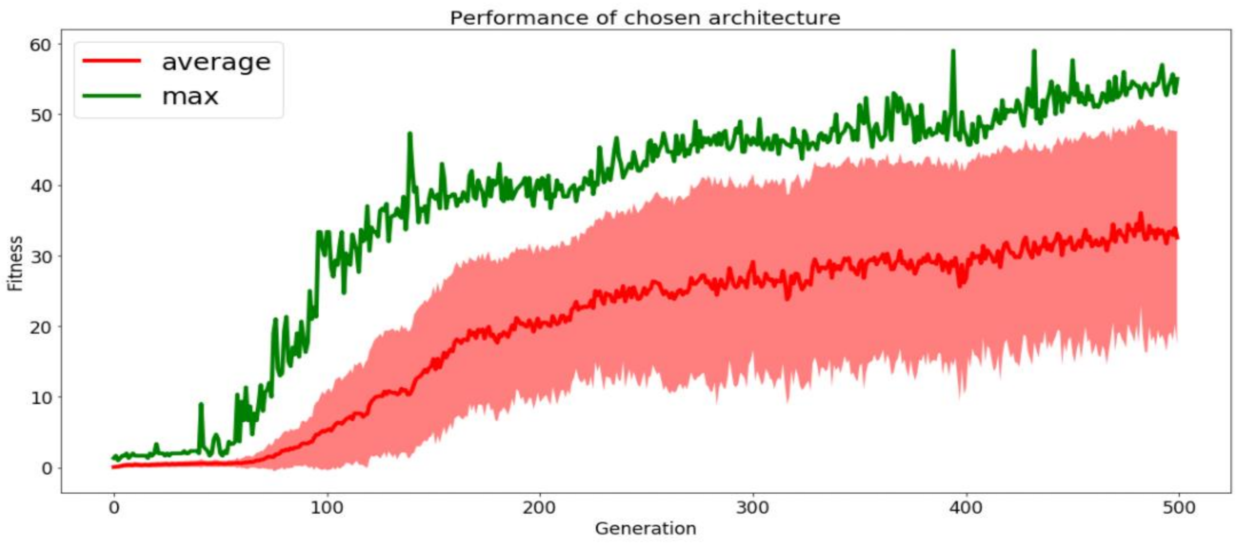
The first result tells us which ANN performed the best after 250 generations. The results for each network are overlayed on Graph 2.

Graph 2



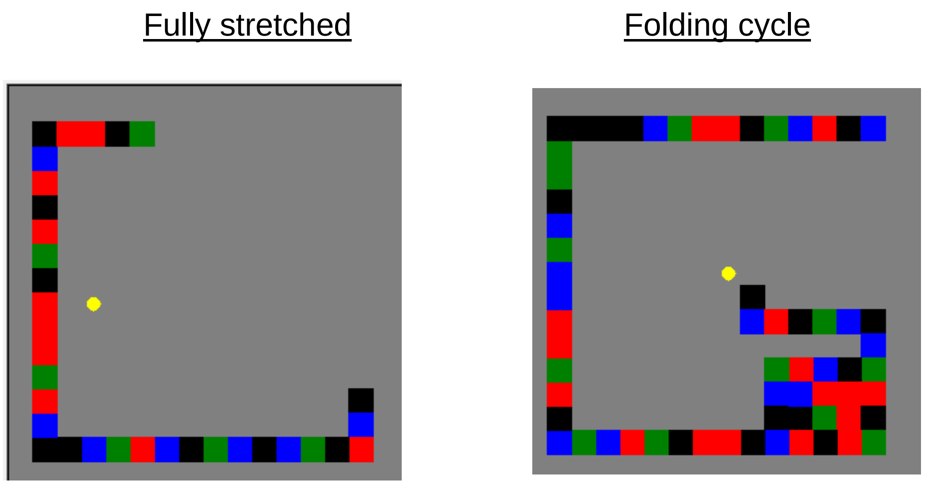
Here, we see that all 3 networks are very similar. The trend starts sharply for all, which was to be expected. It simply means that all the networks start to learn something about the game. The smaller the network, the greater the advantage as it has less weights to evolve through. However, after generation 150, a more discernible pattern appears. NN 1 starts to plateau, perhaps due to 5 hidden neurons in each layer not being enough to learn more complex improvement patterns. NN 3 follows in a similar vein, although it may require many more generations to show improvement (lot more weights). However, the average (line) for NN 2 and its standard deviation (surrounding shade) trends upwards. The deviation is enough to include fitness values from the other methods, and so NN 2 is used for the rest of the results.

Graph 3



To examine the results closer, we can use Graph 3. This can be seen above and is the performance of NN 2 on 500 generations. The average score at the end of our training is 31, out of a total of 185. However, the standard deviation reaches as far as 47 and the max score achieved by a member of the final population was 56. Although nowhere near the maximum target of 185, it is clear that our model performs relatively well for the game. There are clear signs of constant evolution as is seen in the upward trend. It is likely that the plateau after generation 300 is due to the agent attempting to optimise one specific strategy as opposed to evolving entirely to find a new one.

Figure 1: Pattern used to obtain food



In fact, the most common strategy confirms the prediction and preliminary data presented in the method. By having the angle as an input, the snake 'folds' itself to get food and then stretches itself along the side to return and repeat the process. This can be seen from Figure 1 above. The dominant strategy is to traverse the edge until the entire length of the snake has been stretched, then to 'zero in' on the food item by folding at each layer (see bottom right of 'Folding cycle'). In fact, the folds always tend to happen in the bottom right of the map. Moreover, the cyclical pattern is always anti-clockwise and never changes. This can be seen in Figure 2, where in all 3 cases the snake head is the end closest to the food and the anti-clockwise loop is present in every run. Perhaps this is due to the snake spawning facing the right wall every time, and so always prefers to take the upward path where the food is more likely to spawn (more area exposed).

Figure 2: Clockwise pattern examples

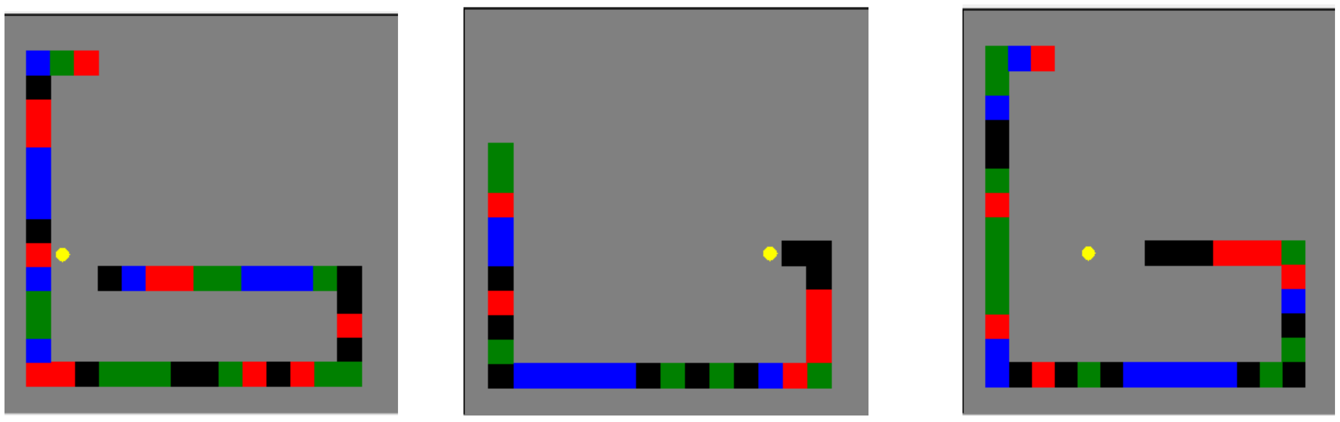
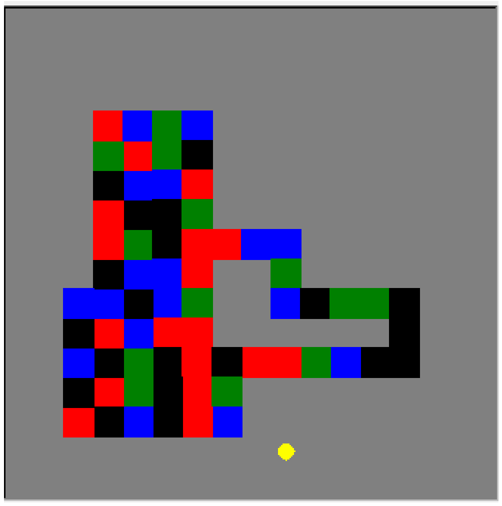
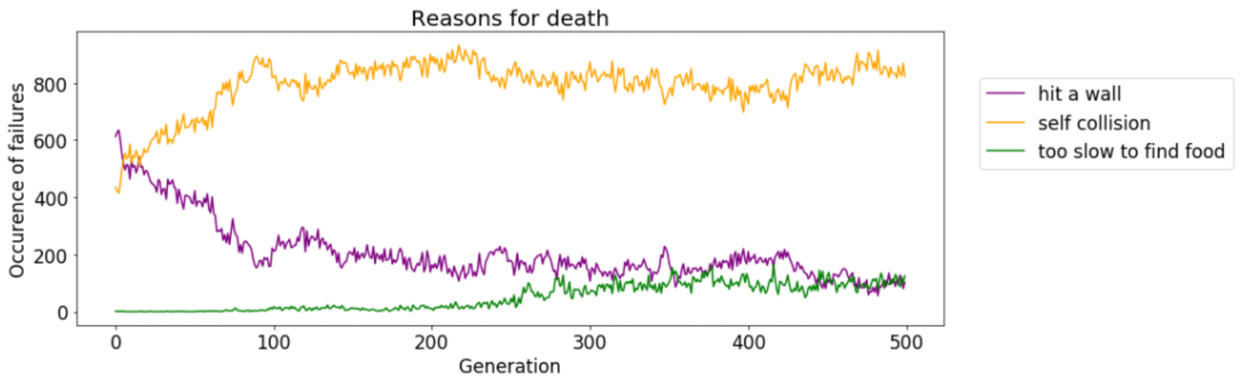


Figure 3: Crash example



A typical crash can be seen in Figure 3. This particular game had a score of 58 and followed all the patterns mentioned above. When the length exceeds the average, the risk of the snake trapping itself is high. This tends to happen when the food spawns under its folding pattern. In an attempt to reach it quickly, it takes the shortest route into itself. A condition in the method mentioned above was that the game ends if the snake traverses the map for too long. The 'too long' parameter is defined as 85% of the maps total traversable area. To unfold properly to reach the food, the snake needs to travel more. However, the training clearly disincentivises that. This may be the reason for the crashes and adjusting this parameter may result in fewer crashes but a longer training time.

Graph 4



Graph 4 plots the reasons for death/occurrence of the failure at each generation. Since each member if evaluated 3 times, the occurrences are scaled to account for each failure. The decreasing wall collisions make sense, as the model has already been seen to be improving over time. Self-collisions are likely due to an increased success of finding food (and hence increasing length) and are complemented with failures to find food for a given distance. This confirms our earlier hypothesis that the longer the snake gets, the more time it takes to unfold and the failure to take the outer path in a clockwise manner results in the snake trapping itself and losing.

4. Conclusion

Initially, the scope for this report was introduced. The scope lies in between the fields of AI and Biology, specifically Evolutionary Algorithms (EA). The point on why games are good to test novel AI methods was made, i.e., there exists an ability to continuously repeat the game in order to learn and create general approximations to combat the natural stochasticity present. The hope is, then, to use these tried and tested algorithms on efforts that can help humans. Examples of similar applications from the literature were drawn. Similarities included the use of common 'rating' functions and differences included the unique way in which fitness could be evaluated (empty spaces for Tetris, score for Pac-Man etc.). [6] [7]

From this overview, it was clear why the neuroevolutionary methodology was chosen. It provides a simple pattern recognition perspective combined with an evolutionary strategy that encourages evolution through cloning (after tournament selection) and mutations. The fitness function was kept simple, unlike [8], and a new input that calculates the angle between the snake head and the food was discussed. A graph showing a preliminary evaluation showed clearly that this angle provides a significant increase in performance, and so it was used as part of the official method. 3 different architectures were proposed, and it was made clear that a short evaluation should be carried out and the results should be generated for the best network.

The initial step to getting the results was to find the best architecture. This was found to be NN 2, which had 2 hidden layers with 15 neurons each. This provided a reasonable trade-off between overfitting (risk with more neurons) and approximating correctly (unlikely with less neurons). The performance of NN 2 was discussed in addition to the specific patterns that it generated after training for 500 generations with a population size of 500. Specifically, this included the anti-clockwise looping pattern and the folding cycle to obtain food. Improvements on this work could involve changing parameters (mutation rate, distributions, starting position for snake etc.) to test if the score can be increased. Further training on larger populations could itself result in a higher score but requires a lot of computational resources and time. Towards the end of training, the reasons for death were trending in the correct direction (greater self-collisions, fewer wall-collisions) but it was evident that the snake took too long to traverse the map to reach the food in the fewest legal moves available. If this final hurdle can be overcome, perhaps by progressively training on different map sizes or by using an entirely different method (e.g., Genetic Programming, evolving a CNN with an image as the input, using 'attention' based networks), then it may be possible to consistently reach the maximum score of 185.

END OF PAPER

References

[1] G. Hinton et al., “Learning representations by back-propagating errors”, Institute for Cognitive Science, California, 1986.

[2] F. Rosenblatt, “The Perceptron: A probabilistic model for information storage and organisation in the brain”, Cornell Aeronautical Laboratory, 1958.

[3] D. Corne, M. Lones, “Evolutionary Algorithms”, Heriot-Watt University, Edinburgh, 2018.

[4] D. Charles, “Enhancing gameplay: challenges for artificial intelligence in digital games”, Digital Games Research Conference, Netherlands, 2003.

[5] P. Krishnankutty, “Nokia’s Snake, the mobile game that became an entire generation’s obsession”, ThePrint, 2020.

[6] N. Böhm et al., “An Evolutionary Approach to Tetris”, Proceedings of The Sixth Metaheuristics International Conference, 2005.

[7] T. Tan et al., “Nature-Inspired Cognitive Evolution to play MS. PAC-MAN”, International Conference Mathematical and Computational Biology, 2011.

[8] J. Yeh et al., “Snake Game AI: Movement rating functions and evolutionary algorithm-based optimization”, Conference on Technologies and Applications of Artificial Intelligence, Taiwan, 2016, pp. 256-261.

[9] Z. Wei et al., “Autonomous Agents in Snake Game via Deep Reinforcement Learning”, IEEE International Conference on Agents (ICA), 2018.