SnapAI

Kieran Kelly kell1170, Wyatt Kormick kormi001

May 11, 2018

Abstract

Pokemon Snap is a game without randomness, which creates an ideal situation for creating a neural network learning through a genetic algorithm. Testing was done on different scoring functions, different breeding algorithms, and different network topographies in an attempt to create the most successful neural network we could. Our results weren't promising in our testing, with our best networks not coming close to what we had hoped. Further testing is necessary for this AI to function as intended.

1 Introduction

Pokemon Snap is a video game made for the Nintendo 64 gaming console where the player rides through levels in a mine-cart and takes photos of Pokemon in the level for points. The game has seven levels, and the game has no randomness involved making it conducive for the creation of an AI. Our goal is to create an neural network that learns through a genetic algorithm that can successfully conquer the first level of the game. The choice of the first level is for the simplicity of it. Only five controls are available as options for possible actions, unlike later in the game with new actions are introduced and complicate the game much more.

The heart of the problem lies in image recognition. Creating the ability to look at images on a pixel by pixel basis and distinguish what images are worth taking pictures of and what is not worth taking a picture of. A neural network was decided on as the best approach for tackling this problem as image recognition has been tackled before through it. Training the neural network was another obstacle that we determined a genetic algorithm would be best. The issue with the genetic algorithm arises with scoring each possible neural network to get the results we want while also breeding the more successful networks in a way that facilitates the learning we hope to achieve. The size of the images also proves difficult to work with as it slows down testing significantly having to constantly create new networks with close to 4 million weights and biases that have to determined for each one. This number is higher or lower depending on the network topography we chose which we did vary in our experiment.

2 Literature Review

2.1 Other Object Detection Approaches

2.1.1 Genetic Algorithm Based Optimization

This method uses only a genetic algorithm to find natural edges between objects in an image. The genetic algorithm takes in a gray-scale image and produces an image of the same size. The pixels of the output image have two values, 1 or 0 (white or black), corresponding to whether or not the pixel belongs to an edge. The fitness function used calculates probabilities of a pixels in the image belong to an edge based on dissimilarities between neighboring regions pixels and compares the output image to these values. The first generation were given an output image consisting of randomly selected values for pixels. Each generation used 512 images, and finished when there were 15 consecutive generations where the fittest image did not change. Reproduction involved trading regions of pixels between mates, and mutations involved flipping the values of pixels. The higher the fitness of an image, the higher its probability to reproduce. This method of edge detection was tested, along with simulated annealing and local search algorithms, on a series of natural and synthetic images, and was found to perform much better than both in terms of time to find the solution and robustness to noise.

2.1.2 Reinforcement Learning with Classifiers

This method uses a classifier, trained on a data set of 12,000 human-labelled images to detect boundaries between objects in an image. This classifier sampled circular regions of the input image, split them in half, checked the brightness, color, and texture in the regions, and if it found them to be sufficiently different, put a border of black pixels between the two regions in the output image [MFM04]. After the classifier put where it thought there would be boundaries in the image, it would check it against where the human place boundaries in the image. Depending on how far away the classifier's output image was from the human's the classifier would adjust the weights and biases of the its network through a process called back-propagation. The goal of this process is that after training itself on a large training set, it would be able to find the boundaries in images it had never seen before. This classifier was found to be able to detect objects in images with low noise levels at a rate of success of just above 90%, but as noise levels increased, the success rate declined exponentially.

A similar method was used in [Giu+16] to detect a usable path to traverse. This classifier implemented a deep neural network to tell the camera which direction to turn based on what the camera saw moment-to-moment. Like the other classifier, this neural network was trained on thousands of images. This method, however, used 17,000 images of trails, and the classifier's job was to say whether the camera needed to be turned left, turn right, or stay the same in order for the camera to be facing the trail straight on. In an experiment, the authors found that their deep neural network managed to classify a sample set of images just as well as human observers could, with a consistent 85% success rate.

2.1.3 Bayesian Modeling

This method of object detection computes probability models of the foreground objects and the background and the two compete over ownership of the pixels. The models are each 5 dimensional (x, y, red, green, blue) and consist of every pixel in the image. Instead of modelling each pixel independently, it models the entire background as one image. This is to show the dependability of pixels on each other, as pixels nearby to a high probability background pixel are going to have a higher probability of being a background pixel, likewise for foreground pixels. For every frame the probability of a pixel belong to the background or the foreground is calculated for every pixel. A sliding window of frames is kept for both the foreground and the background, which corresponds to the learning rate of the system.

The background of an image should have a set structure. It is either stationary or it moves in some sort of a pattern, so pixel regions that follow this structure have a higher probability of being in the background. The foreground calculates its probabilities based on the idea of temporal persistence [SS05]. Basically what this means is that objects tend to stay in the same general area, and stay the same general color between frames. Changes have to occur over time and aren't instant. At first, the probability of any pixel belonging to an object is uniform across all pixels and frames. Then, pixel regions that vary from other dominant pixel regions have a higher probability of belonging to a foreground object. Once a pixel is found to have a high probability of belong to a foreground object, nearby pixels have a higher probability of also belonging to an object than those further away. At the next frame, there is a high probability of a pixel region to be in the foreground if, in the previous frame, there was a pixel region in the foreground in the same general area. This strategy of object detection had an extremely high detection rate at 99.708%. In each scene nearly every object that was detected by a human observer was also detected by the algorithm.

2.1.4 CAMSHIFT

Continuously Adaptive Mean Shift (CAMSHIFT) is another probability-based approach to object detection and tracking [Bra98]. The original ”mean shift” algorithm's use is to find peaks in probability distributions, and has been used previously to divide static images into regions. CAMSHIFT modifies the mean shift strategy to find these peaks in probability distributions that change in time along with a sequence of video frames, such that it instead tracks dynamic image regions. The algorithm chooses an arbitrary section of the image to search. It then calculates the mean pixel values in that region, and chooses a new subregion that contains those values, and repeats on the new region. It does this until is converges on what it decides is an image region. CAMSHIFT applies this strategy over a sequence of video frames to track the movement of image regions. CAMSHIFT ends up being a simple, efficient algorithm, but is found to still be able to handle noisy images, and objects moving in irregular ways.

2.1.5 Comparison

Object detection and tracking doesn't have a definite solution yet. As such all available strategies for solving the problem are based in probability and learning, neither of which are 100% accurate. Of these methods CAMSHIFT provides high accuracy with high efficiency, and Bayesian Modeling provides near-perfect accuracy, but both are beyond our abilities to implement. Using only a genetic algorithm to solve our problem probably would take a lot of generations to accomplish, which would take a lot of time that we do not have. If we were to try this again using one of these methods, we would use a supervised neural network, as it is within our abilities to design, and can be done in a reasonable time frame while also likely providing a higher accuracy than our current method.

2.2 Refining / Structuring a Neural Network

2.2.1 Genetic Algorithm to Refine a Neural Network

[WL93] provides the basic strategy that we elected to use to solve our problem, a genetic algorithm used to optimize a neural network. More specifically the genetic algorithm decides the optimal amount of hidden layers, the number of neurons in each hidden layer, and the weights and biases connecting each. We use a simpler version that only optimizes the weights and biases, and our hidden layers maintain the same size. The algorithm, dubbed "GANNet", runs the problem on randomly generated neural networks. The highest scoring networks are then selected to breed together to create a new network using values from both parents. This is done by selecting values from one parent and overwriting the values in the other parent with them. These new individuals are then mutated by randomly changing randomly selected values in the neural network. This is done to provide diversity in order to search more of the possible options. The new generation that the problem is then run on is a selection of the parents and the new children. This strategy is always guaranteed to converge on a network that is better than a starting network, but it has the problem that it is possible for it to converge on a solution that is not a global solution. This is a problem that is common to most genetic algorithms, and is one that we are prepared to accept.

2.2.2 Genetic Algorithm Optimal Population Size

Finding what size of population to use with a genetic algorithm is tricky. It largely depends on the complexity of the problem [Ala92]. Low populations have a much more rapid mutation from the original, which has been proven in nature, while larger populations resist to mutation more. Time of execution is also a factor to consider, which is tied to the complexity of the problem. Seeing as our neural network has quite a high complexity, in order to reduce the time of execution we have limited our population size down to five. This also helps with a more rapid mutation which we predict to lead to faster results. This does run the risk of mutating quickly away from a potentially successful neural network structure, but it is a calculated risk we are willing to take.

2.2.3 Mutation in Genetic Algorithms

Looking at nature we can look and see there are various ways to create the next generation [SF96]. There is self mutation, crossover, crossover mutation to name a few, and most genetic algorithms only take advantage of self mutation. Using different or multiple types of mutation can be hugely beneficial to the algorithm to achieve an optimal solution more quickly. That is why we utilize all three of the methods mentioned above in our genetic mutation. This provides us the possibility of converging on an optimized solution more quickly. The optimized mutation rate for each of these strategies may vary in the problem but also depends on the problem. The one studied by Smith and Fogarty has significant differences from ours so we decided to go with a fixed mutation rate as their findings for the population size we have chosen didn't have too significant of an improvement between adaptive mutation rates and fixed.

2.2.4 Neural Network Ensembles

Using an ensemble of neural networks to improve training and performance can be very effective [HS90]. Using cross validation, Hansen and Salamon optimize using cross-validation in order to get a more generalized neural network that can work across multiple data sets instead of just keying in on the training data set. On top of that they use an ensemble because of the issue of optimizing weights being tied to local minima, not global, so multiple runs of the algorithm can give different results. We took the idea for an ensemble for training purposes and decided that since we only care about a single data set, that a genetic algorithm would be better suited on the ensemble of neural networks to try to optimize the best solution.

2.3 A Different Approach to the Problem

2.3.1 A Minimax Algorithm

Find a maximizing method for player 1 when player 2 isn't necessarily trying to minimize player 1's payoff in a a zero sum game. Creating a finite-state machine to mimic a nonminimax player is the problem with finding an algorithm to maximize player 1's payout accurately. They use an evolutionary programming technique to try to achieve this. It takes the finite-state machine used to mimic the maximizing player and randomly creates a mutation from one of four choices [Bur69]. The resulting child finite-machine's average payoff is then analyzed and compared with the parent's average payoff. If the child has a worse average payoff than the parent then it is discarded and another mutation is chosen. They use this technique with several children which are evolved in parallel. They conclude that the experiments they run indicate success with this method.

This approach is quite comparable to the method we chose to use. They use a genetic algorithm very similar in how we use ours, the subject of their genetic algorithm however is different. While we are trying to find an optimized neural network to play the game, they are trying to find an optimized finite-state machine to imitate the player. While it seems this would create the same result, we decided a neural network better covers our specific problem. With the minimax algorithm we would have to treat the game itself as a player not as the environment it is.

2.3.2 Alpha-Beta Pruning

The concept of alpha-beta pruning is similar to an algorithm called "branch and bound." The main difference between the two algorithms is that "branch and bound" doesn't have both a lower and upper bound so it cannot make "deep-root cutoffs"[KM75] like alpha-beta pruning can. The most notable possible applications of alpha-beta pruning is chess. It analyzes the best case with a proof that they are dealing with the optimal case. It then deals with a non-optimal case with no "deep cutoffs" before finishing talking about the model of the algorithm.

Alpha-Beta pruning would've been something to try to add to the minimax algorithm above. The specifics of how to accurately apply pruning to a scenario where one player in a non-minimax player proved very difficult which is ultimately why we decided against using a minimax algorithm with or without pruning.

2.3.3 State Space Search

State space search (SSS*) [Sto79] proposes a better strategy for the minimax algorithm than alpha-beta pruning. They achieve this by traversing the tree in parallel instead of a left-to-right strategy that alpha-beta is forced to do. They then delve into the definition of state space search in general and explain how it applies to create SSS*. Following with a proof of correctness to back their claim. Then they compare the efficiency of SSS* to alpha-beta by running an extensive simulation in which every test they made SSS* performed strictly better.

This strategy is something that could have been used in tandem with the minimax algorithm above. We ran into similar struggles as we did with the Alpha-Beta pruning, how do we apply it to the minimax algorithm with a non-minimax player? This proved to be beyond our scope of understanding so we chose not to go this way with our approach.

3 The Approach

Our approach to the problem can be broken down into two main parts: the neural network, and the genetic algorithm acting on the neural network. We set it up so that our AI will take as input a snapshot of the game and read every pixel individually. It then evaluates those pixels to decide one of five moves: look left, look right, look up, look down, or snap a photo.

3.1 Neural Network

We set up the neural network to take as input each pixel of the screen, 256 x 256 pixels x 3 color channels for a total of 196,608 input values. These input values are then passed through two hidden layers of the network, each with 20 nodes, before reaching the output layer which has five nodes corresponding to the five possible actions. The output that those nodes receive are of the confidence the AI has in each of the actions and ultimately picks the action with the highest confidence. The weights and biases originally designated to each node is in a range from a negative constant, to a positive constant. That constant differed as we experimented.

Between each layer of the neural network we applied an activation function called the Sigmoid function to shrink the value in each neuron to a value between zero and one. The sigmoid function is: $f(x)=\frac{1}{1+e^{-x}}$. This is done so that no one node of the network can become a large outlier that can skew the system the wrong way. Positive values will be closer to one while negative values will be closer to zero.

3.2 Genetic Algorithm

To refine our neural network, we apply a genetic algorithm to it. We start with five different neural networks and run them to see each of their results. The results are determined using a scoring function that we designed to influence certain behavior that we want the neural network to do. We set up the scoring function with the intent of having the AI maximize taking photos while also trying to reach the end of the level. This is so it has a chance to see all the Pokemon in the level and have as many opportunities to take good photos. The scoring function also includes what the game gives as a score for the photos. The scoring system we implemented is discussed in further detail below in 4.2.

After determining the scores of each of the five networks, the genetic algorithm will then keep the top two scoring networks, toss the bottom three and breed three children to replace the ones that were tossed. This is done so that we can take the good parts of the top scoring networks and try to create improved networks that succeed in doing what we want them to do. The breeding consists of taking some parts from top scorer 1 and some parts from top scorer 2, while also adding in a chance of mutation. The mutation is necessary so the chance of the neural networks reaching a local maxima instead of the global maxima is minimized. The breeding process is discussed in further detail below in 4.1.

4 Experiment

To test the effectiveness of our genetic algorithm we chose to run several different versions of it in the actual game and run the results through our scoring functions. We hoped to see which of these versions would produce the highest scoring neural network. They were all run on the game's first level, from the very beginning. Since there is no randomness in the game, starting them all at the same point would result in an even playing field amongst all of the resulting neural nets.
A "run" for a neural net goes from the start of the level until it reaches the end, or until after it has run out of film after 60 pictures have been taken. After a neural net is run, it is scored using one of our scoring functions, and after 5 have been run and scored, a new generation is formed using one of our breeding strategies. We ended up running the algorithm with different combinations of two breeding algorithms, three scoring functions and several network topographies.

4.1 Breeding

We had two versions of a breeding strategy. In both versions, the top two highest scoring networks would survive to the next generation. The third child would be

a cross-breed with the top scoring network as the parent, and the second as the mate. The fourth would be another cross-breed with the second as the parent, and the top as the mate. There is a distinction here between the mate and the parent, because in our algorithm, the parent's network has a higher probability of passing on its nodes than the mate's network.
The fifth child is where the two breeding strategies differ. In breeding strategy 1 (B1), for the first five generations, child 5 would be a completely new, random neural network. After the fifth generation, the fifth child would be another cross-breed with the top as the parent and the third as the mate. The reasoning behind this strategy was to increase the chances of including a high-scoring network in the initial batch of children, while in the later generations basing more networks on the highest scoring children.
In breeding strategy 2 (B2), adding randomness is still the main idea with the fifth child. The fifth child is always a cross-breed with a new, random neural network as the parent and the mate is the top scoring network. This is to allow the algorithm to explore more of the state space while still promoting the top scoring network's values.

4.2 Scoring

We started the experiments with two scoring functions (S1 and S2) planned, but after some experimentation, we decided to add a third one (S3). A strict, specific scoring function doesn't allow for more free, creative networks to form, and is is closer to just writing the AI ourselves rather than letting it form. This describes our third scoring function, while our first is gives much more freedom, and our second is somewhere in between. All of our scoring functions have four parts.
The first part is the progress of the child. In the first and second functions, the network scores 1500 points for completing the level, and 0 points for not. The third scoring function gives points based on making it to key milestones throughout the level, more points the further it makes it through. A child still earns 1500 points for completing the level, but loses points for barely making out of the start. This part of the scoring function also completely throws the child away if all it does is take 60 pictures all at once. The reasoning behind that decision will be discussed in section 5.
The second part is the number of pictures taken. In the first scoring function, a child earns 30 points for every picture taken, for a maximum of 1800 points. In the second and third functions, the points per picture is based on a curve. A child taking a large amount of pictures is unlikely to take a good picture except by chance. A child can earn 40 points per picture if it takes around 40 pictures, but as the number drifts away from 40 pictures, the score value lowers per picture.
The third and fourth parts are the same for all three scoring functions, and are intended to encourage taking good pictures. The third part gives 300 points for every picture that will be sent to Oak for scoring. In order to be eligible for this,

the picture needs to be of a Pokemon, and only one picture per Pokemon can be sent. We chose these pictures ourselves based on what we believed would be the highest scoring pictures, but we also sent every picture that we could. So, if a picture was really bad, but it was the only one for a Pokemon, it would be sent. The fourth part adds the score Oak gives for each picture to the child's score. The game scores pictures based on how much of the screen the Pokemon takes up, whether the Pokemon is facing the camera or doing an interesting pose, and whether or not the Pokemon is in the middle of the frame.

4.3 Network Topography

We ran the experiments on neural networks with several different topographies. We always used the same general structure: one input layer, one output layer, and two hidden layers. What differed between trials was the number of nodes in the hidden layers, and the maximum values for weights and biases. For the numbers of nodes we used 10, 20 and 50 nodes for the first hidden layer, and 5, 20, and 25 nodes for the second hidden layer. For maximum weight values we used 2, 5, and 20. Finally, for maximum bias values we used 0, 2, and 20. These maximums for weights and biases were maximums for the starting values in the neural network, and for how much a weight or bias could be mutated by. A mutation could push a value beyond its maximum.

4.4 Results

Seven different trials were run. Each was run until no significant changes occurred in the last 4 generations. Figure 1 shows the first four trials. Three of the trials are using scoring function 1 and one is using scoring function 2.

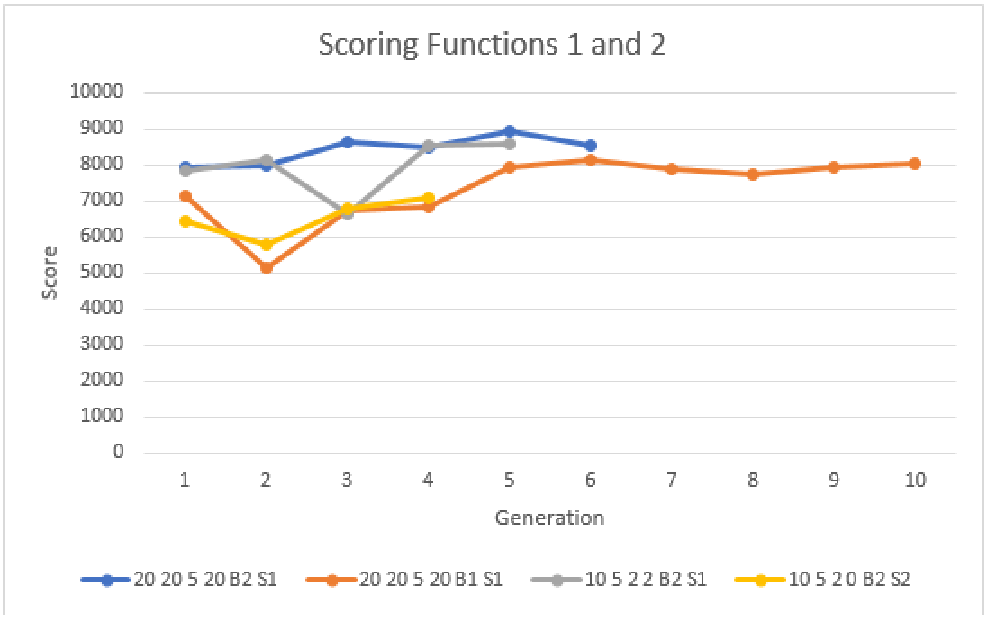

Figure 1: Results of Running Scoring Functions 1 and 2. Labels are in the format (HLayer 1 Size, HLayer 2 Size, Max Weight, Max Bias, Breeding Strategy, Scoring Function)

After those four trials were run, scoring function 3 was introduced. The results from the three resultant runs are shown in figure 2. The third trial was forced to be cut short due to a lack of time, causing the abrupt lack of data from the trial.

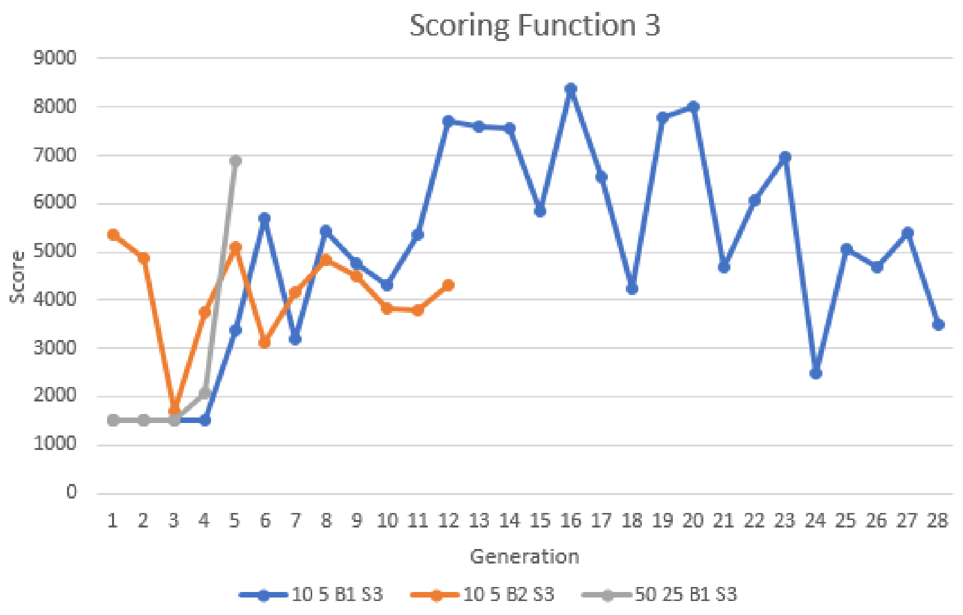

Figure 2: Results of Running Scoring Function 3. Labels are in the format (HLayer 1 Size, HLayer 2 Size, Breeding Strategy, Scoring Function). All were run with maximum weight 2 and maximum bias 2.

5 Analysis

With the first two "free" scoring algorithms, every trial run converged on one solution within a few generations. The solution they all found was to press the snap button 60 times in a row. The luckiest few would earn points in the high 8000's and would move on to produce children that also pressed the snap button 60 times in a row. As shown in figure 1, there would be no improvement after this convergence. They had found a local maximum that they would not stray from. Our idea of a global maximum involved taking good pictures of every unique Pokemon in the level. These managed to get 4 unique Pokemon pictures while the level contains around 10. To move beyond this local maximum, we introduced scoring function 3, which would throw away any child that pressed the snap button more than 90% of the time.

After the introduction of scoring function three, scores became much more volatile. Children were found to make it to the end of the level much more often, but the quality of pictures dropped significantly. Improvements in strategy also came much more infrequently. In our one trial that did manage to stabilize, (10 5 B2 S3) in figure 2, the strategy it came up with was not an expected one. What it would do is immediately look straight up into the sky, and position itself perfectly, such that when the one Pokemon that flies over the player does so, it would take a picture of it. Figure 3 shows an example of what such a picture would look like.

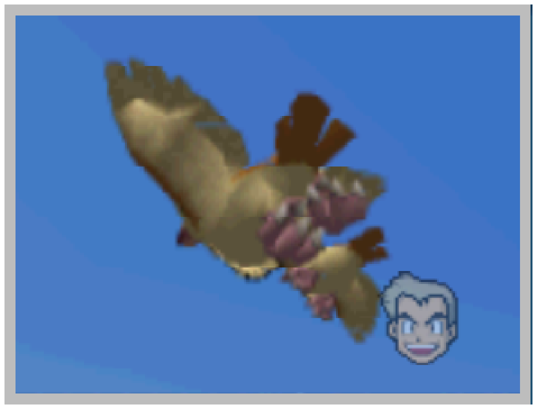

Figure 3: Example Pidgey Snap

Perhaps if given many more generations to run, these would eventually converge on a strategy similar to our ideal, but with what we have, maybe a genetic algorithm isn't an ideal method of finding such a strategy.

6 Conclusion

The results we achieved weren't exactly what we were looking for. While we simplified the problem as much as we could, it proved more difficult than initially anticipated. Our neural networks did a fantastic job of finding local maxima and not being able to deviate from them. This was unexpected from our initial expectations and proved difficult to find a suitable work-around.

These results can't be pegged to one or two different aspects. It could be how we scored the networks, how we bred them, the number of networks in each generation, the activation function used. The scoring was set up in a way that we thought would be best for maximizing score, and although we made some adjustments to it, our approach to scoring might not have been conducive to the learning we were hoping for. As for breeding, more mutation could have enough effect so that the networks would deviate from the local maxima to get closer to the global maxima we are looking for. If the number of networks per generation were increased, while it would severely slow down testing, it might overall arrive at the global maxima we are looking for in less generations. Any or all of these things may have the impact to change our results to something closer to what were striving to achieve.

6.1 Future Work

Going forward, there are many avenues that can be pursued. The first would be to look at the number of networks per generation, it would inherently also change our breeding algorithm and would be the best place to start to try to get results better than what we got. Second is to look at our scoring functions. Find a way to score that puts more emphasis on the score the game gives, instead of scoring heavily towards trying to reach getting a score the game gives in the first place. This would slow down the progress significantly in the beginning of the generations but could have the effect we are looking for in avoiding the local maximums that we ran into. Thirdly would be to look at the activation function, the Sigmoid Function, that we used. Another function that is used commonly as well is called the ReLU function, standing for Rectified Linear Unit function, f(x)=$max(0, x)$. This gives all inputs less than 0 the value of 0 and keeps all values greater than 0 as the value they are. This could speed up the process of arriving at a successful neural network. A completely different route to go would be to use a supervised neural network instead. This way the network could learn from us playing the game instead of learning from the scoring function we provided. This would speed up the learning process significantly and provide a way for us to avoid the local maximums we were road-blocked with.

References

[Bur69] G Burgin. "On playing two-person zero-sum games against nonminimax players". In: *IEEE Transactions on Systems Science and Cybernetics* 5.4 (1969), pp. 369–370.

[KM75] Donald E Knuth and Ronald W Moore. "An analysis of alpha-beta pruning". In: *Artificial intelligence* 6.4 (1975), pp. 293–326.

[Sto79] George C. Stockman. "A minimax algorithm better than alpha-beta?" In: *Artificial Intelligence* 12.2 (1979), pp. 179–196.

[HS90] Lars Kai Hansen and Peter Salamon. "Neural network ensembles". In: *IEEE transactions on pattern analysis and machine intelligence* 12.10 (1990), pp. 993–1001.

[Ala92] Jarmo T Alander. "On optimal population size of genetic algorithms". In: *CompEuro'92.'Computer Systems and Software Engineering', Proceedings.* IEEE. 1992, pp. 65–70.

[WL93] David White and Panos Ligomenides. "GANNet: A genetic algorithm for optimizing topology and weights in neural network design". In: *International Workshop on Artificial Neural Networks.* Springer. 1993, pp. 322–327.

[BZP94] Suchendra M Bhandarkar, Yiqing Zhang, and Walter D Potter. "An edge detection technique using genetic algorithm-based optimization". In: *Pattern Recognition* 27.9 (1994), pp. 1159–1180.

[SF96] Jim Smith and Terence C Fogarty. "Self adaptation of mutation rates in a steady state genetic algorithm". In: *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on.* IEEE. 1996, pp. 318–323.

[Bra98] Gary R Bradski. "Real time face and object tracking as a component of a perceptual user interface". In: *Applications of Computer Vision, 1998. WACV'98. Proceedings., Fourth IEEE Workshop on.* IEEE. 1998, pp. 214–219.

[MFM04] David R Martin, Charless C Fowlkes, and Jitendra Malik. "Learning to detect natural image boundaries using local brightness, color, and texture cues". In: *IEEE transactions on pattern analysis and machine intelligence* 26.5 (2004), pp. 530–549.

[SS05] Yaser Sheikh and Mubarak Shah. "Bayesian modeling of dynamic scenes for object detection". In: *IEEE transactions on pattern analysis and machine intelligence* 27.11 (2005), pp. 1778–1792.

[Giu+16] Alessandro Giusti et al. "A machine learning approach to visual perception of forest trails for mobile robots". In: *IEEE Robotics and Automation Letters* 1.2 (2016), pp. 661–667.

7 Group Contributions

7.1 Kieran Kelly

- Half of Section 2
- Abstract
- Sections 1, 3, 6

7.2 Wyatt Kormick

- Programming
- Experimentation
- Half of Section 2
- Sections 4, and 5