Dakota Sliter
Austin Auckerman
Noah Betz

AI Project 2 Writeup

Project Brief:

We will design an AI agent whose outputs are determined by a neural network trained with a genetic algorithm. The agent will be trained to play a python version of the SnakeGame, in which a "snake" navigates a grid world to seek "food". Upon successful consumption, the length of the snake's body grows by one grid segment. The game ends when the snake collides with a wall boundary, eats a segment of its own body, or reaches the maximum length possible. Score is simply the number of food the agent collects before dying.

Our project will use the snake game implementation found here, with some modifications: https://github.com/rajatdiptabiswas/snake-pygame. We will make use of the Python library Keras to create the neural network, and we will create our own implementation of a genetic algorithm to train the weights of said network.

Phase I: Preparation

Before we can start on our application, we need to develop an understanding of the neural network library we will be using. Additionally, we need to review our notes on genetic algorithms and brainstorm ways to integrate one with our game code. The game needs to be modified to run with no GUI for speed of iteration, and we need to run multiple concurrent instances of it with different genomes to simulate a generation of a certain population size. We then need to take the results of those runs and design an evaluation function that will appropriately advance our agent towards our goals. Since the start of iteration results in semi-random agent moves, the score would be 0 for many generations and this would take far too much time to train. To address, we brainstormed the possibility of introducing a time limit for each generation, and if the time limit is exceeded with a score of 0, instead evaluating performance for those early generations by tracking how close an agent got to the food at any given point. In addition, we also wanted to put some weight towards survival time because an agent that is staying alive for longer is better than one that received the same score.

We also need to decide the inputs for the neural network. Initially we thought about using the location of the food and the coordinates of the head and tail of the snake body. However, since the snake grows as it eats more food, this presents a unique challenge. How do we account for the increasing length of the snake? Our neural network has a fixed number of inputs, so we cannot simply add more to it as the game progresses. Instead, we attempted to track three points on the snake's body: the head, midsection, and tail. We also brainstormed introducing a length input as well. Unfortunately, these inputs did not work, so we decided to instead use the change in distance between a move and the current position in relation to the food, whether or not each move is safe, and whether or not there is a piece of snake body immediately in the way of the food.

Another challenge is determining what size neural network is adequate. The size of our neural net remains constant, but the weights between nodes are evolved, so we must ensure that our starting point is sufficiently complex enough to result in our desired behavior, but not too complex as to be computationally inefficient for our consumer-grade hardware. There is some question on whether or not a GA will be able to evolve the agent behavior sufficiently to play the game. We believed that it would be able to, especially because we found that there are some implementations of snake running on a neural network already, even if these implementations take completely different approaches to the game and to network training.

Phase II: Coding the Network/Game

We started coding the neural network by first experimenting with Keras in a test file. Since none of us had any experience with Keras for neural networks, we read up on documentation, watched some tutorials, and made an untrained model of a network. It took awhile, but we figured out how to properly get and set the weights of the network manually, and then collapse those weights into a one-dimensional array to represent an agent's genome. Finally, we figured out how to grab the outputs of the model and return the index of the highest output value, which we will take to be the action for the agent. Now that we had some practice with Keras, we moved on to integrating it in our game file.

To do so posed some challenges. The game file we are using is set up to take in user input (WASD keys) to control the agent, but we instead needed to replace those inputs with the chosen action output of our neural network. We also decided we needed to run the neural network for each game loop, so that the agent has the chance to make an action decision in every frame, although we were unsure if the network would be too slow for that. However, after modification and testing, our current net turned out to be fast enough to run in the game loop without issue.

To make the network, we used Keras' Sequential model to add layers to a model object. We set the input shape, as well as the number of nodes in each layer, and used a rectilinear activation function. We coded the entire file in a class, so the game class can construct a neural network and then run the model with the designated input values for each game loop. In order to determine the exact size and shape of the network, we will need to test it with the genetic algorithm to see if training progress is being made. If not, we may need to increase complexity.

Next, we created a version of the game that runs without a GUI to speed up training the neural network weights. We also modified the game to be an instantiable class, so our genetic algorithm can create different instances of the game object and call the runGame() method on each. Additionally, the game object is responsible for creating the neural network class by passing in a genome, which is a one-dimensional array containing all the weights of the network in order. This way, whenever the GA loop finishes, it can update the genomes for the population and pass them into the game object constructor in the next loop. Once the GA has trained the neural nets for a significant period of time, we will take the resulting genome and run it in the GUI version of the game to observe agent behavior and judge if more training is needed.

To further speed up runs, we implemented the python multiprocessing module's Pool method. This allowed us to run each generation approximately 8x faster, depending on how many CPU cores the running machine has. We also decided to run the game multiple times for each run so that we could confirm that our results were not due to the randomness of food placement, and we made sure to normalize all of our inputs so that the magnitude of the inputs would be the same..

Phase III: Coding the Genetic Algorithm

To code the genetic algorithm, we referenced the general approach provided in class notes. We then brainstormed metrics for the fitness function, deciding to account for score and time. Higher score in a shorter period of time results in higher fitness values. Next, we implemented elitism to keep the best-performing individual each generation. We programmed a tournament selection method and our mutation function that chose new random values for a gene. Now we were ready to test out our solution.

After running for many generations, we discovered our first algorithm and neural net inputs were unsatisfactory. Evolutionary progress stagnated after 100 generations, and fitness even started to decline. We decided we needed to revisit our fitness function and neural net inputs. For the inputs, we changed from inputting simple coordinates to instead distances. Inputs now include: food delta up/down/left/right which represent if that move is closer to the food than the current position, a metric representing how frequently the agent has eaten, the last choice move's X/Y coordinates, and four parameters representing if a direction to move is safe. In our fitness function, we added a component to the fitness score that tracked an agent's distance from the food upon death, and weighted it to prioritize higher fitness for closeness. We now ran the game again with these changes, and found an improvement over the last version, but after hundreds of generations, we seem to be getting stuck in a local maximum. We hypothesize we either need a much more complex net structure, much longer evolution times than we can execute on our machines, or different inputs.

We tried another iteration of the neural net that had three equal-sized hidden layers instead of just one. After many more generations, performance seemed the same or worse than the simpler neural network, despite having more genes to encode information in. A hypothesis is that our crossover or mutation functions are not as effective as needed. For example, we are using single-point crossover by choosing a random point in the parent genome to splice into the child. Another method we could implement would be to instead loop through each gene of the parents, and if a random number is greater than 0.5, place the gene of parent 1 into the offspring, or parent 2 otherwise.

We also tried both the previous implementations using a 3-game average instead of single runs for each individual in the populations, but this did not get us any further. In addition to this, we removed the last_choice inputs and the "time since last eaten" metric, and we replaced these with an input that tells the network if there is a body piece nearby that is blocking the direct path

to the food. These changes still stopped improving the network after 100 generations and they weren't progressing past eating 1 or two foods. This made us try using deeplearning and a set of inputs derived from manually playing the game and saving every input that the neural net would receive, but this still wasn't able to work. We were able to get the network to claim a 85% accuracy training on thousands of inputs over hundreds of epochs, but unfortunately the networked refused to do anything but go right, and wasn't useful at all.

Since our progress with training the genetic algorithm with a static neural network wasn't progressing as we'd hoped, we instead opted to test out a different topology for our network: NEAT. We found a simple python library, neato, and implemented a neural network with it took the following inputs: food delta up/down/left/right, a boolean representing if the body is in the way of the agent and food (up to path length of 4), and booleans representing the safety of a directional move. The NEAT library allowed us to easily initialize the starting population using inputs we already had. It also supports simply adding fitness values to genomes within the Ecosystem object, and has a built-in get_best_genome() function. Using this new neural network structure, we ran more generations to see if our results would improve. Our fitness function remains the same. We found better results than before, but note that higher crossover rates result in lower fitness over time. Using NEAT, the network randomly attempted to add new nodes and connections between the nodes, and then scored the network based on how it played. We then killed off the worst 50% of the population, and rebuilt it using mutated versions of the best individual. This worked considerably better and we ended up with some networks averaging 3-8 food eaten in a run after a few hundred generations, but after that we saw the same dropoff that we saw before.

Phase IV: Training the Weights

For training the weights, we started off with two main fitness weights: food score and time score. We wanted to weigh getting food heavier so we multiplied the fitness by 2000 when calculating. The time score was simply how long the agent would survive in the game. When training these weights didn't get us very far we brainstormed another weight we could use. This new weight was named death score and was weighed extremely high with multiplying it by 80000. After having these three weights in our fitness function the training began to show progress with continuous generations getting 3-5 food scores, which is better than hundreds of generations of only the previous two weights only.

We spent the most part of a week testing different weights and inputs to get better results. For the first half of training we only saw a few 2s for food score, but after implementing the new weights in the fitness function and finding the ideal crossover and mutation rate we started to see higher and higher food scores. It was soon common for our generations to score double digit food scores, and even a score of over 30. At the beginning of the training process we saw fitness values of under a thousand mostly lower hundreds that slowly started to grow throughout the generations. Having it weighed mainly to avoid death and pick up food once it figured out at

least a few one scores in a generation it started to grow and within 50 generations it was scoring all numbers between one and ten.

Using NEAT, the edge weights and nodes were randomly created, so we didn't have any focus or control on the structure of the network. Instead, the algorithm randomly created these connections and also randomly assigned edge weights. These weights were all between -1 and 1. After hundreds of generations, the networks created by this algorithm were all significantly improved, and our leading network was able to eat 7 food on average for every game it played, and it was able to survive for way longer than any other run. Ultimately, NEAT was able to do a better job with the inputs we gave it, but it still fell short of our goal of consistently playing like a good player.

Phase IV: Why things didn't work

Things didn't turn out how we wanted, and this is for a few reasons. The first reason is we spent too much time changing things in the genetic algorithm when we should have traded it for NEAT earlier. We didn't have any clue how many nodes to use, how many layers to use, or what weights to use, so we took a guess for all of this. We also spent a significant amount of time running tests for dozens of generations then changing the parameters to try and make it better. These cases work better for NEAT because it tries randomly adding nodes and weights and edges so that all of this is handled by the genetic algorithm. We also spent too much time trying new things before settling on a set of inputs. This meant constantly trying to train, then throwing it out when we decided it wasn't good enough, even though we didn't know what the specific cause of the network not being good enough was. We thought that training using deep learning from recordings of our gameplay would let us jumpstart the genetic algorithm training, but this failed too because there were too many inconsistencies in our play to train the network. With data training, garbage in equals garbage out, and if we didn't act perfectly we would be training the network to act bad too. Additionally, it seemed to take into account move frequency for its training, so it usually just tried whatever direction happened to be the most common output in the training data. Finally, the two constants between all our implementations that failed was the inputs and training time. With our machines it took as many as 12 hours to train 100 generations, and our inputs are the biggest constant, so we believe that maybe different inputs and more training time would have improved the NEAT implementation if not all the implementations.