INDEPENDENT STUDY GROUP REPORT

ARDEN RASMUSSEN, SEAN RICHARDSON, ADRIANA ROGERS

We decided to apply the strategies of neural networks and genetic algorithms in order to train an AI to play a game of snake. This sample implementation provides a good testing ground for utilizing these strategies. The search space for an AI to play snake is sufficiently large, that it is possible to see improvements in the performance of the AI, but it is small enough that those improvements do not take long.

Date: Spring 2019.

5.1. **Game Set-Up and Rules.** The first step of the process was to construct the game itself. The game itself is very simple. A player controls a snake on a grid. The snake is trying to eat food (an apple), and every time the snake eats a piece of food, it grows in length by one. If the snake runs into the walls or itself, it dies.

We decided to split the game into four main functions.

(1) *Initialize:* This function initializes the game state. This includes the position of the apple, the position of the snake, and the size of the grid. For most of our testing we defaulted to a grid of size 20.
(2) *Get Move:* This function provides provides some interface with the current game state, and retrieves a direction to move in. This interface can either be a user input, or it could be some AI process.
(3) *Update:* This function updates the game state provided with the direction to move in. This function moves the snake, and if it eats an apple, it extends the snake by one, and spawns a new apple on the grid.
(4) *Valid:* This function checks if the current game state is valid. This is where we test to see if the snake has run into itself, or a wall.

The basic structure of the snake program is to execute function 1 once, and then while the game state is valid execute functions 2-3. The final part of the game is the scoring system. Each update without dying increases the score by 1, and each apple eaten increases the score by 1000.

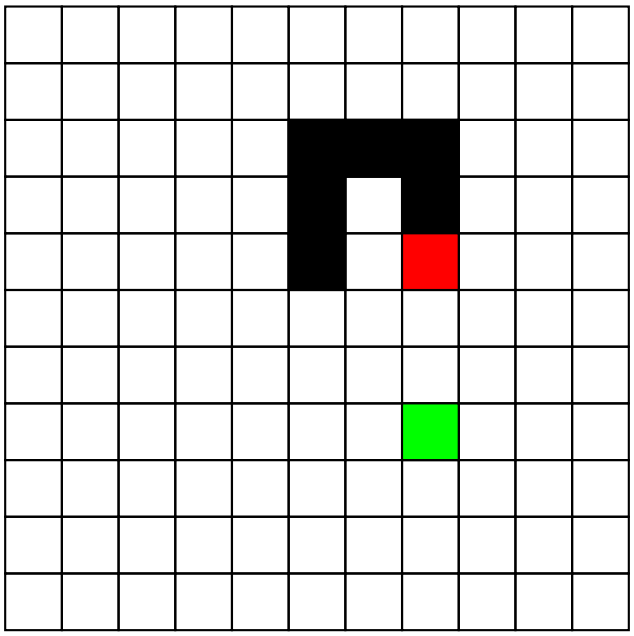

FIGURE 1. Example game state for a snake game, here black denotes the snake body, green denotes the apple, and red denotes the snake head.

Using this basic structure of the snake game, we determine that the optimization problem at hand is to maximize the score of the game. Now we construct the

architecture of a neural network and genetic algorithm to solve this optimization problem.

5.2. **Neural Network Structure.** Since any neural network requires a specific number of input, we decided on utilizes rays. For each of the four directions from the snake, we cast a ray. The ray will tell the neural network how far away something is in that direction. For this we case three different rays in each direction.

(1) Apple Ray. Tells the network how far away the apple is in that direction, if at all.
(2) Snake Ray. Tells the network how far away the snake body is in that direction, if at all.
(3) Wall Ray Tells the network how far away the wall is in that direction.

The casting of these rays can be seen in figures 2.

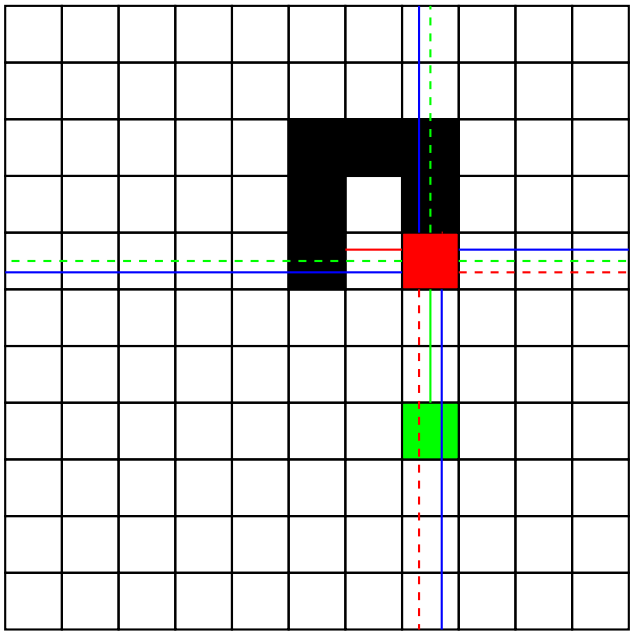

FIGURE 2. Rays cast from the snake head. Green rays are the apple rays, red rays are the snake rays, and blue rays are the wall rays. If a ray is dashed, then it did not hit anything.

Once each ray has been cast, we pass the reciprocal of the distance to the neural network. For example the down apple ray in figure 2, would have a distance of 2 since it travels two spaces before hitting the apple. Thus the network would get a value of $\frac{1}{2}$. This is done so that closer values are larger, and thus can easily be viewed as more important.

We also know that the output of the snake must be one of four values; Up, Down, Left, or Right. Using this we constructed a neural network with 12 input neurons and 4 output neurons. Most of the tests that we made were done without any hidden layers, but it is also possible to insert hidden layers as desired. The architecture of the network is demonstrated in figure 3.

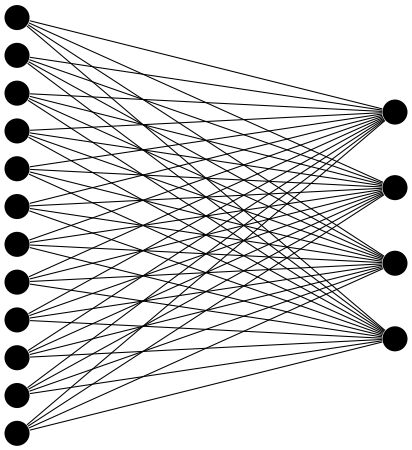

FIGURE 3. Architecture of the snake AI neural network.

Using this network architecture, there will be 48 weights in the weight matrix, and 4 bias values in the bias vector. This means that there will be a total of 52 values that must be optimized.

5.3. **Training Network with Genetic Algorithm.** To optimize weights of a neural network, one must have training data, and the answer key to that data. For situations like a game of snake. It is impossible to know which direction it should go given every game state, so we are unable to optimize a neural network in the classical method. This is why we implement a genetic algorithm to optimize the neural network.

We construct a population of chromosomes, where each chromosomes has 52 genes. We define a method for the conversion from the list of genes in the chromosome to a neural network. The process is outlined below.

$$\text{Chromo} = [x_1, \ldots, x_{52}] \implies W = \begin{bmatrix} x_1 & x_2 & \ldots & x_{12} \\ x_{13} & x_{14} & \ldots & x_{24} \\ x_{25} & x_{26} & \ldots & x_{36} \\ x_{37} & x_{38} & \ldots & x_{48} \end{bmatrix}, \quad B = \begin{bmatrix} x_{49} \\ x_{50} \\ x_{51} \\ x_{52} \end{bmatrix}$$

For the evaluation stage of the genetic algorithm, each chromosome is converted into a neural network, and it the plays the game of snake. The score returned from that game is the fitness of the chromosome. Since the game has some randomness in the generation of the apples. We evaluate each chromosome 5 times, and average the scores to be the fitness. Then the rest of the genetic algorithm proceeds as nominal, selecting the best chromosomes of the population. After 20-50 generations, the network has usually made considerable progress, and the learning curve has begun to plateau.

In the future it would be beneficial to experiment with additional layers, and how having more layers of different sizes would effect the fitness of the neural networks produced by the genetic algorithm.