INDEPENDENT STUDY GROUP REPORT

ARDEN RASMUSSEN, SEAN RICHARDSON, ADRIANA ROGERS

1. Introduction

In our independent study, we aimed to study the mathematical foundations of neural networks, and other machine learning techniques. We began by following Stanford's lecture videos from their course, "CS231n: Convolutional Neural Networks for Visual Recognition" [Li]. This course provided a foundation for neural networks and convolution, from which we continued our studies. We also used Goodfellow's book, "Deep Learning" [GBC16].

We quickly realized that analogies to biological neural networks and the brain, which seem to be unavoidable in many cursory instruction on neural networks, are misleading. They give rise to the mischaracterization of neural networks as black boxes. Rather, neural networks "learn" by optimizing for some loss function. At the heart of this method, as we will describe below, are techniques that we find from linear algebra and calculus.

Another grievance we had with our study of neural networks, besides the biological interpretation, was the arbitrary selection of hyper-parameters. Most importantly, parameters such as the step size of gradient descent and the learning rate, which we will describe below, are typically set using guess-and-check, or cross-validation. To ease these frustrations, we began studying genetic algorithms, as a way to search through the space of possible hyperparameters to be used to train a neural network to play the game snake.

This report will begin in Section 2 by framing our studies as an optimization problem, giving a general background to optimization. Next, in Section 3, we will give an overview of the architecture of neural networks, propagation, and convolutional layers for image classification. In Section 4, we will outline the process of a genetic algorithm to solve optimization problems. Finally, in Section 5, we will explain how we used these concepts to train a neural network to play the game snake.

2. Optimization

3. Neural Networks

3.1. **Architecture.** The structure of a neural network can be broken down into smaller building blocks. Every network is a connection of layers. Each layer has a different type, and does a different action. Some of the common layer types are described here.

Input: The input layer takes the input from the user and provides it to the rest of the network. This is one of two ways the network can interact with the user.

Date: Spring 2019.

Dense: A dense layer is the core of the neural networks that we will consider. These layers are what makes the neural network able to make decisions, and process the inputs. We will discuss more about these layers later.

Convolutional: The convolutional layer is a method for extracting the important information from the input to the layer. These layers do little of the actual processing, but can be very important to distill the input into the important components. We discuss the specifics of the convolutional layer in section 3.3.

Pooling: The Pooling layer is used to remove superfluous information form the input. It is only used to reduce the size of the input. It is very frequently used in close conjunction with convolutional layers.

Output: The output layer, takes what the neural network has processed, and provides it to the user. If the network is training, then this layer determines the error of the prediction of the network to the actual of the data set.

As mentioned the Dense layer is where the important computation occurs. We will go into more detail here, as to what is actually happening in the dense layers. A dense layer can be through of as a collection of neurons. Each neuron takes the value of the all the neurons on the previous layer, and outputs a values of its own. Each of the values from the previous layer are multiplied by some weight, and then they are added together. It is also necessary to have a bias term which is also added to the sum.

We can now construct the expression for a single neuron (y_j). We begin by considering the values of the previous layer to be stored in x_i, and the weights for the jth neuron to be stored in w_{ji}, where i corresponds to the i of x_i. Then the bias term will be b_j. Using this notation we can express the computation of a single neuron to be

$$y_j = \sum_{i=1}^{N} w_{ji}x_i + b_j.$$

Generalizing this for all the neurons in the layer, it becomes clear that the expression for a dense layer will be

$$\mathbf{y} = W\mathbf{x} + \mathbf{b}.$$

Where $\mathbf{x}$ is the vector of values from the previous layer, $\mathbf{b}$ is the vector of bias terms, and W is the matrix of weights.

This is the general structure of a dense layer, however there is one issue. If we combine multiple dense layers in a row, we see that each is linear, so multiple layers in a row will also be linear. This means that multiple layers, could be equally powerful as a single layer. To prevent this, we introduce an activation function. This changes our expression for the single neuron to become

$$y_j = f\left(\sum_{i=1}^{N} w_{ji}x_i + b_j\right),$$

where f is our activation function.

The reason for this activation function is to introduce a source of non linearity into the system. There are three main choices for activation functions.

Sigmoid: The sigmoid is a standard activation function, and works well in most cases, but has somewhat expensive computations. One of the main

issues is that it does not center the output around zero.

$$\frac{1}{1+e^{-x}}$$

TanH: The tanh function is also frequently used, as it is similar to the sigmoid function, but centers the outputs around zero. It also has some more intensive computation cost.

$$\tanh(x) = \frac{1}{1+e^{-2x}} - 1$$

ReLU: The ReLU (Rectified Linear Unit) function has been found to be the best activation function to use. It has very little computation cost, and provides suitable enough output.

$$\max(x, 0.0) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

By connecting different combinations of layers, with different numbers of neurons in each layer, we are able to construct neural networks that are able to recognize, interpret, and act on complex sets of data. However, we must first be able to train a network, so that it acts correctly. We will explain this process of how networks lean and act on the data they are provided.

3.2. **Propagation.**

3.3. **Convolutional Layers.**

4. Genetic Algorithms

A genetic algorithm is a method for solving optimization problems, often analogized to the process of natural selection. Genetic algorithms are useful for solving optimization problems that are not well suited for standard optimization algorithms, such as problems where the objective function is discontinuous, nondifferentiable, or highly nonlinear.

For our study of genetic algorithms, we relied heavily on Sastry and Goldberg's chapter on genetic algorithms in *Search Methodologies* [SGK05].

The basic set up of a genetic algorithm is to "evolve" a population of candidate solutions. Each candidate solution in a search population, referred to as a chromosome, is represented by a binary string. To evolve a solution, we optimize the fitness of every individual in the population, which is the value of the objective function being solved in the optimization problem. Once our problem is structured in terms of a population, candidates, and fitness, we evolve with the following steps [SGK05].

4.1. **Initialization.**

The initial population of candidate solutions is usually randomly generated in the search space. The size of the population is a parameter that must be pre-determined.

4.2. **Evaluation.**

The fitness values of the candidate solutions are evaluated. This occurs once the population is initialized, and after each offspring population is created. Steps 2-6 are repeated until a terminating condition is met, usually if the fitness value is within some tolerance level. If such a condition is met, the process is terminated during this phase.

4.3. **Selection.** Based on the fitness values, a portion of the existing population is selected to breed a new generation. Selection will create copies of solutions with higher fitness values to prefer better solutions to worse ones. In order to select these chromosomes, there are many methods available. Commonly, roulette-wheel selection is used. With this method, the selected candidates are chosen randomly with probability proportionate to their relative fitness [SGK05].

4.4. **Recombination.** In this phase, a new population generation is created from the parent generation. For each new solution, two or more parental solutions are combined to create new offspring. Recombination methods typically perform crossover operators. As shown in figure 1, in one-point crossover, a crossover site is selected at random, and binary digits on one side are exchanged between the chromosomes. More generally, in k-point crossover, some k crossover points are selected, and digits are exchanged *k-1* times. Alternatively, in uniform crossover, every binary digit is exchanged with some swapping probability. These provide examples for the most basic crossover operators.

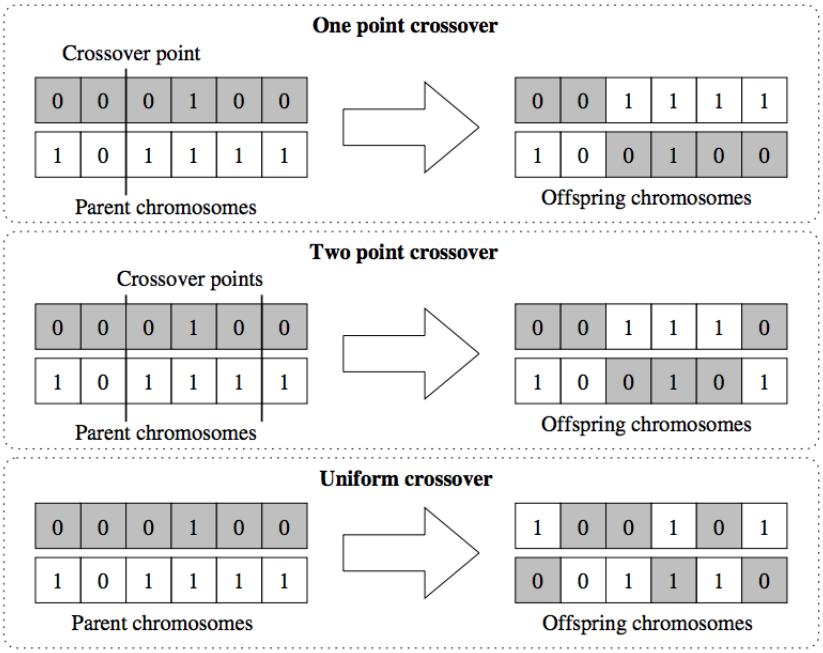

FIGURE 1. Crossover methods [SGK05].

4.5. **Mutation.** Mutation randomly modifies a single candidate solution according to some mutation probability. Essentially, this can be viewed as performing a random walk in the vicinity of the individual chromosome solution.

4.6. **Replacement.** The offspring population replaces the original parent population. The offspring population can replace *all* of the parent population, or we can use steady-state replacement to only replace n new offspring.

5. Snake Project

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.

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.

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 3.

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 3, 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.

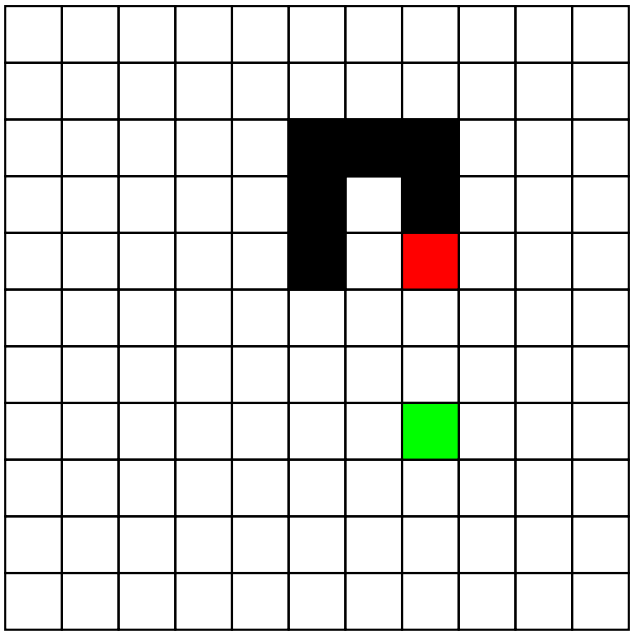

FIGURE 2. Example game state for a snake game, here black denotes the snake body, green denotes the apple, and red denotes the snake head.

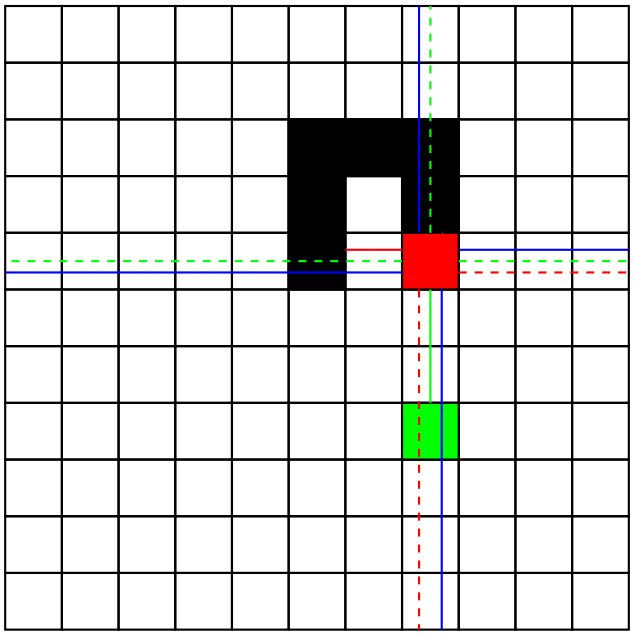

FIGURE 3. Rays cast from the snake head. Green rays are the apple rays, red rays are the snake rays, and blue rays are the wall rays. If a ray is dashed, then it did not hit anything.

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 4.

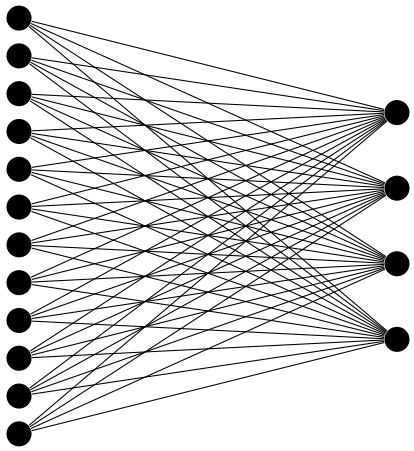

FIGURE 4. Architecture of the snake AI neural network.

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.

References

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[Li] Fei-Fei Li. Cs231n: Convolutional neural networks for visual recognition.

[SGK05] Kumara Sastry, David Goldberg, and Graham Kendall. Genetic algorithms. In *Search methodologies*, pages 97–125. Springer, 2005.