Neural Networks:

Optimizing Hyperparameters with Evolutionary Algorithms

Christopher Woloshyn

cwolosh1@binghamton.edu

May 2020

Contents

Listings

Abstract

Artificial Neural Networks lie at the heart of data science and machine learning; their implementation permeates throughout the biggest tech companies in existence, and their rapid development shows how powerful of a tool they are for solving modern, data driven problems. The goal of this project is to first, outline the fundamentals of neural networks, second, explain what hyperparameters are and why optimizing for them is such a difficult problem, and third, showcase my solution to the problem. By the end of this report, we will see why an evolutionary algorithm is the best choice over every other contemporary approach to picking hyperparameters.

Chapter 1

Introduction

There are two goals for this report. The the first goal is to provide a clear, and accessible outline of some essential topics in modern machine learning and soft computing. The second is to report my attempt to address the problem behind searching for optimal hyperparameters for a deep neural network through the use of an evolutionary algorithm. Hence, project features a combination of biologically inspired learning solutions in order to produce the best possible solution to a particular problem. I hope to outline these methods in a way that is educational and digestible who are unfamiliar with the concepts, and an excellent demonstration of my knowledge on the subject for those who are familiar. However, before I can begin with the exposition of the topic and the motivation of the project, it is important to establish the notation conventions that will be used throughout the report, as well as review some fundamental concepts of linear algebra that are essential for understanding how Neural Networks function.

1.1 Notation

Having a clear and consistent notation established right in the beginning is worth the extra effort. Many scientific papers or text books do not do this which can be very confusing for a reader, especially if they are unfamiliar with the content to begin with. In this section, I will outline all notation that will be used throughout the rest of this report, for clarity and efficiency. This section can be skipped if you are comfortable with the mathematics this report will cover.

1.1.1 Linear Algebra Notation

- Let any boldface lowercase letter be a vector. E.g. $\mathbf{v}$ is a vector of dimension m.
- Let any boldface uppercase letter be a matrix. E.g. $\mathbf{M}$ is a matrix of dimension $m \times n$.
- Let any normal lowercase letter be a scalar. E.g. x is a scalar.
- Let the letters i and j be indexing variables. Hence, $x_{ij} \in \mathbf{M}$, $\forall\ 1 \leq i \leq m$, $1 \leq j \leq n$.
- Let $\mathbf{A} \odot \mathbf{B}$ represent the Hadamard product between $\mathbf{A}$ and $\mathbf{B}$.
- Let $\mathbf{v} \otimes \mathbf{w}$ represent the outer product between $\mathbf{v}$ and $\mathbf{w}$.

1.1.2 Activation Functions

The specific details of the following functions will be covered later in Chapter 2.5. However, the notation for these functions will be established here. Note that applying any activation function to a vector $\mathbf{x}$ is equivalent to applying the function element-wise to each x_i, the i^{th} element of $\mathbf{x}$.

- Let $\sigma(x)$ represent the Sigmoid Activation Function.

$$\sigma(x) := \frac{1}{1 + e^{-x}}$$

- Let $\text{ReLU}(x)$ represent the Rectified Linear Unit.

$$\text{ReLU}(x) := \max(0, x)$$

- Let $\mathrm{ReLU}_L(x)$ represent the "Leaky" Rectified Linear Unit.

$$\mathrm{ReLU}_L(x) := \begin{cases} ax & x < 0 \\ x & x \geq 0 \end{cases}$$

- Let $\mathrm{S}^+(x)$ represent the Softplus Activation Function.

$$\mathrm{S}^+(x) := \ln(1 + e^x)$$

- Let $\mathrm{Sw}(x)$ represent the Swish Activation Function.

$$\mathrm{Sw}(x) := x\sigma(\beta x)$$

- Let Sw^e represent the E-swish Activation Function.

$$\mathrm{Sw}_e(x) := \beta x\sigma(x)$$

1.2 Linear Algebra Review

Linear Algebra is an essential topic to all higher level mathematics, and it is especially prevalent in Machine Learning architectures. Being able to work with values at the scale of vectors instead of individual scalars saves a lot of time, both from the perspective of learning, and from the perspective of calculating. This section will outline some very basic concepts and operations from linear algebra that will be used by the neural network for the actual project. It will also make covering how Neural networks work in our crash course much, much easier.

1.2.1 Vectors and Matrices

Vectors and matrices are the building blocks of linear algebra; they are the mathematical objects that define the entire topic. A vector is a 1 dimensional array of numbers. In our case, each x_i in a vector **x** is a real number. Hence, a vector with

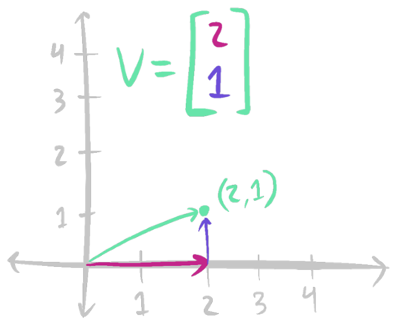
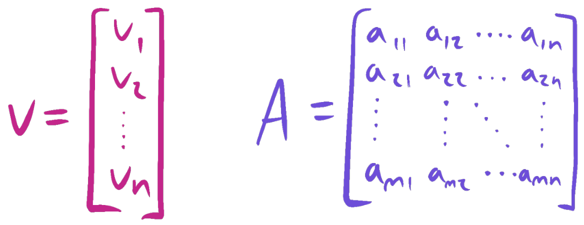


Figure 1.1: Representing a 2-D vector graphically.

2 entries is said to have *2 Dimensions*, and belongs to $\mathbb{R}^2$. This can be represented graphically by drawing an arrow pointing to the coordinate pair the entries in the vector correspond to, as is done in Figure 1.1.

Matrices are the second fundamental component of linear algebra. These are *two* dimensional arrays of numbers. Each row and column of a matrix can be treated as their own vectors, called row vectors and column vectors respectively. Additionally, vectors can be abstracted in terms of a matrix. For example, a 2-D vector could also be called a 2×1 matrix. We will later see why representing



$$V = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \qquad A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Figure 1.2: An $n \times 1$ vector $\mathbf{v}$, and an $m \times n$ matrix $\mathbf{A}$.

values with vectors and matrices is so useful and compact. Vectors and matrices come with their own sets of operations that can compress most of the mathematics we are dealing with.

1.2.2 Matrix Multiplication

The first operation we will discuss is called the *dot product*. The dot product is a operation that "multiplies" two vectors together to produce a scalar value. In order for this to be possible, both vectors must be the same size. If there are two vectors **v** and **w**, each with two elements, then the dot product would be $\mathbf{v} \cdot \mathbf{w} = v_1 w_1 + v_2 w_2$.

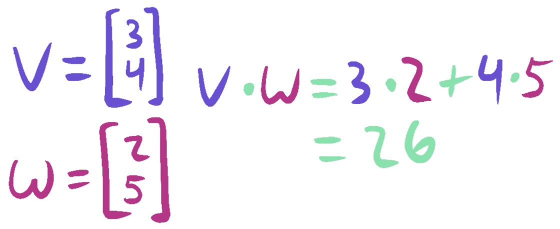

$v = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$
$w = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$
$v \cdot w = 3 \cdot 2 + 4 \cdot 5$
$= 26$

Figure 1.3: An example of the dot product of two 2-D vectors.

With the dot product established, we can now address the second matrix operation, the *matrix product*. The matrix product is a massive driving force behind neural networks. Like regular numbers or vectors, matrices can also be multiplied together, however, this process is much more complex. The matrix product considers every combination between the row vectors of the first matrix, and the column vectors of the second matrix, and produces a new matrix comprised of the dot products of those combinations. It can be quite a confusing concept, so hopefully the example from Figure 1.4 is a little more clear.

Similar to how a vector can be thought of as a special case of a matrix, the dot product can be thought of in terms of a special case of matrix multiplication, where a $1 \times n$ matrix, and an $n \times 1$ matrix are multiplied to produce a 1×1 matrix, which is the same as a scalar.

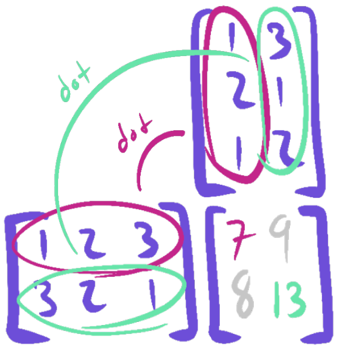


Figure 1.4: An example of a 2×3 matrix being multiplied with a 3×2 matrix to produce a 2×2 matrix.

1.2.3 Special Operations

Lastly, I want to cover two additional types of matrix and vector operations that are not typically covered in an introductory linear algebra class, but are also important for understanding how a neural network functions. The first operations is called the *Hadamard Product*. The Hadamard Product can best be described as an element wise multiplication between two matrices of the same size. This operation takes two matrices of the same dimension and outputs another matrix of the same dimension whose entries are the product of every corresponding entry of the two input matrices.

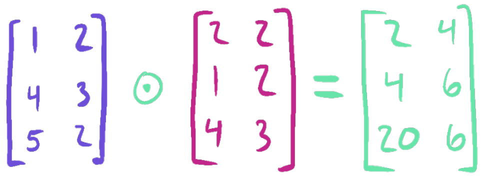
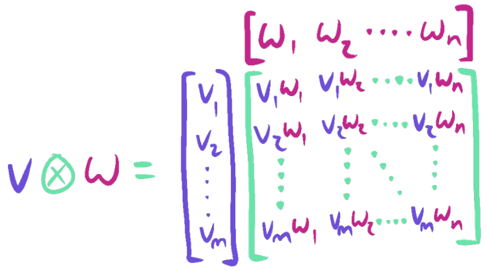


Figure 1.5: An example of two 2×3 matrices being multiplied via the Hadamard Product.

The final operation we will cover is the *outer product*. The outer product is an operation between two vectors, but instead of producing a scalar like the dot product does, it produces a matrix. In other words, the outer product between an $m \times 1$ vector **v**, and a $1 \times n$ vector **w**, $\mathbf{v} \otimes \mathbf{w}$ will produce an $m \times n$ matrix. Like matrix multiplication, this can be confusing, and Figure 1.6 will display the process more visually.



$v \otimes w =$
$[w_1 \; w_2 \cdots w_n]$
v_1
v_2
v_m
$v_1w_1 \; v_1w_2 \cdots v_1w_n$
$v_2w_1 \; v_2w_2 \cdots v_2w_n$
$v_mw_1 \; v_mw_2 \cdots v_mw_n$

Figure 1.6: An example of two 2×3 matrices being multiplied via the Hadamard Product.

Finally, all of the building blocks are in place and we are ready to discuss the topics that motivated the project, such as Neural Networks, evolutionary algorithms. The next section will outline the remainder of the report including the topics of each chapter, and their importance in the context of the full project.

1.3 Outline of Contents

The purpose of this section is to briefly outline what is to come in the following chapters. Since this report also serves to explain the most important background concepts of the topic, roughly half of the report is dedicated to background information of neural networks, hyperparameters,and genetic algorithms, and the other half is dedicated to the details of implementation for the project as well as the final results and reflections.

Chapter 2 will briefly cover the details of neural networks. Obviously, this is not an in depth explanation of every concept, since that would require a book's worth of information. Instead, Chapter 2 is meant to serve as an explanation to those already familiar, or lightly familiar with the topic already. Knowledge of calculus is assumed, and knowledge of linear algebra makes understanding easier. Topics covered in this chapter are artificial neurons, different network architectures, training and the back propagation algorithm, and a discussion of various activation functions, since this is an important piece of the project.

Chapter 3 introduces the problem, and thus the motivation for the project. It discusses what hyperparameters are and why they can make optimizing the performance of a neural network an very challenging problem. Then there is a discussion of solutions in the form of evolutionary algorithms, hence the motivation of the project. There is also a brief discussion of how evolutionary algorithms work, and the particular type chosen for this project.

Chapter 4 discusses the details of implementation. We will walk through each class of the code in detail and discuss how everything functions. This discussion is meant to elevate the understanding of someone who is not as comfortable or experienced with programming. The classes discussed are the Deep Neural Network architecture, the two classes that make up the genetic algorithm, and the class and script that run the evolution applied to the provided data.

Lastly, Chapter 5 will showcase the results and my reflections on those results. This is where the key pieces of project, and hence the final argument from the Abstract are solidified by the evidence of the results. Then, Concluding remarks and a discussion of future steps and ways to improve the project even more will follow.

Apendices and references will be provided at the end. Please enjoy reading!

Chapter 2

A Crash Course in Neural Networks

Neural networks are a hot subject. Thousands of books have been written on the subject over the past few decades, so attempting to teach neural networks from scratch would require its own book. Instead, the purpose of this chapter is to outline and walk through the fundamentals of neural networks, and the most important aspects about them, that are relevant for the main research question. To start the discussion on neural networks, the best place to begin in my opinion is with the modern, real world applications that we encounter every day.

2.1 Real World Applications

Neural networks are a pretty common buzz word throughout tech company rhetoric and the media. But even if we look past the hype they possess we discover a huge set of possibilities. My favorite example of a problem neural networks can solve is hand written digit recognition. That is, given a hand written input, classify which digit, 0 through 9, it represents. With this, the research question seems fairly simple, but when you sit down and think about how to solve this problem in a traditional computer science fashion, the problem then becomes very daunting. How can all possible cases be accounted for with such a large input? Moreover, how can one programmer account for every possible rendition of what a “2” or a

"4" is, since there are so many ways that they can be written?

As it turns out, neural networks solve this problem by allowing the engineer to circumvent hard coding every possible case. It is through this concept where the power of neural networks, and machine learning tactics like it, come to fruition! The process of a neural network "Learning" can even be thought of as a process that automatically and heuristically accounts for every such case. It does this through data and approximation. The data is needed to provide enough examples for the architecture to learn what each integer that a hand written digit corresponds to is. The approximation occurs via the mathematics that powers the network. It is impossible to account for every possible input, and therefore every possible case, even with a computer, because the typical problem suited for a neural network can have upwards of 10^{100} possible inputs. However, if enough data can be provided, the network can "learn" an approximate set of possible classifications for each hand written digit, regardless of the extremely large input space.

For example, imagine there is a calculus test that needs to be passed. There is no reason for the student to study European history or sports medicine to do well the test. The best way for the student to prepare for the calculus test is to complete practice problems that have been on past tests. When problems similar to the practice problems appear on the test, the student will have learned the proper way to solve them, and likely do well on the test.

2.2 Neurons

Let's first break apart what a neural network actually is. As the name suggests, a neural network is a collection of neurons, in a brain, connected together to process data [1]. In biology, neurons are comprised of three major parts: the input, which is made up of dendrites and synapses, the processing, which is made up of the cell body and nucleus, and the output which contains the axon and terminals. See Figure 2.1 for an illustration. The input part of a neuron receives information from all of the neurons that feed into it. The main body of the neuron processes this

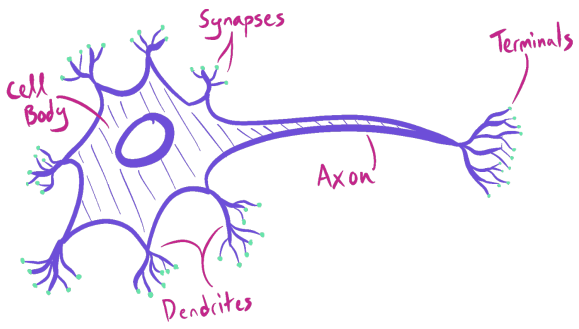


Figure 2.1: A schematic image of a neuron [1], showing the inputs, processing area, and outputs. These are the building blocks of the human brain.

information and, as a response, sends this new information to all of the neurons it feeds into via the axon and terminals. This is an extreme simplification, but it outlines the three important components that we try to replicate with an artificial neuron.

Compare this, to an artificial neuron, which adopts the same structure as a real neuron, but instead uses mathematics to handle processing tasks. Artificial

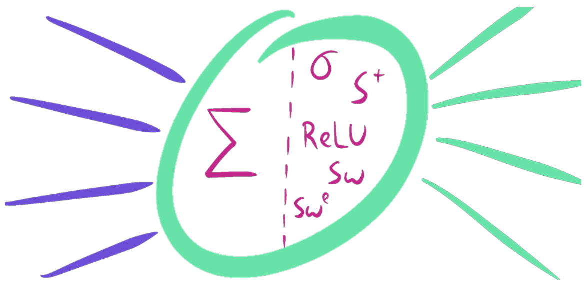


Figure 2.2: A model of an individual artificial neuron, showing the inputs, processing area, and outputs. These are the building blocks of the neural networks.

neurons are just complex functions that take a vector as an input and return a scalar value to all neurons it feeds into. As a result, an entire network of artificial neurons is really just an even more complex function that takes a vector as an input and produces a new vector as its output.

Each neuron contains two functions: the first one considers all the inputs from the nodes leading into it. This is called a basis function. The second function is called the activation function, it's purpose is to replicate the "firing" of a neuron that occurs in an actual brain. There are several different types of activation functions, but there will be more detail on this in Chapter 2.5. In the next section, we will cover a few of the many different neural network architectures, especially the one used for the project.

2.3 Architectures

There are dozens of different types of neural network architectures. They all have their specific use cases, and many are complex expansions of older, more traditional architectures. The most important architecture for this project is the Multi-layer Perceptron, which most of our time will be dedicated to. However, we will also spend some time introducing more modern architectures at a high level, mainly for the sake of revealing the depth of the subject.

2.3.1 Multi-Layer Perceptron (MLP)

The Multi-layer Perceptron is a fundamental architecture that many of the more widely utilized architectures today are built upon. It is a generalization of the simple perceptron, which was introduced as early as 1958 [2]. At this time, computers weren't nearly as capable as they are today, so their viability as computational allies were extremely limited. Moreover, the mathematics that allow the neural network to actually "learn" from the training data weren't yet concrete. I wasn't until several decades later in the mid 1980s when the back propagation algorithm was first published, reawakening the viability of neural networks [3]. Computers were significantly quicker, and used on a much larger scale that the usefulness of the perceptron, multi-layer perceptron, and deep MLP were thrust to the forefront of machine learning. For this project, a deep MLP architecture has been created with the capability of creating an MLP of any depth.

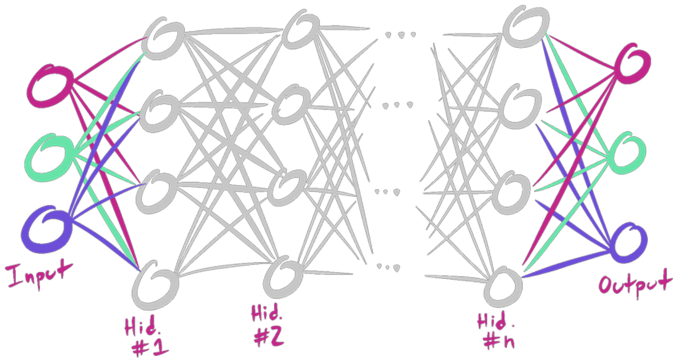


Figure 2.3: An MLP with input dimension 3, output dimension 3, and n hidden layers each with dimension 4.

Consider an MLP with $n + 2$ total layers. The first layer is the input layer, suppose it has dimension p_0. The last layer is the output layer; suppose it has dimension p_{n+1}. Thus there are a total of n hidden layers each with dimension p_i where $0 \leq i \leq n$. Figure 2.3 is an illustration of one particular example of an MLP with arbitrary depth. We will look at the details of what the connections between nodes on the network means and how the network actually learns in Chapter 2.4. Next, we will very briefly go through a few more modern neural network architectures.

2.3.2 Modern Architectures

Now we will briefly look into two of the many more modern artificial neural network architectures. This first is the Convolutional Neural Network (CNN), and the Recurrent Neural Network (RNN).

Convolutional Neural Network (CNN)

The CNN was first introduced in the late nineties by Dr. LeCun and colleagues [4]. This architecture has proven to significantly and consistently outperform the

MLP and as a result is the its modern successor. CNNs are used at the forefront of machine vision, image recognition, self driving cars, etc. The reason for its success is due to the way it extracts features from the input data before feeding into a deep MLP. We can see in Figure 2.4 that this architecture has several pooling layers to reduce the input into a smaller set of features that are then used to train an MLP attached at the end of it.

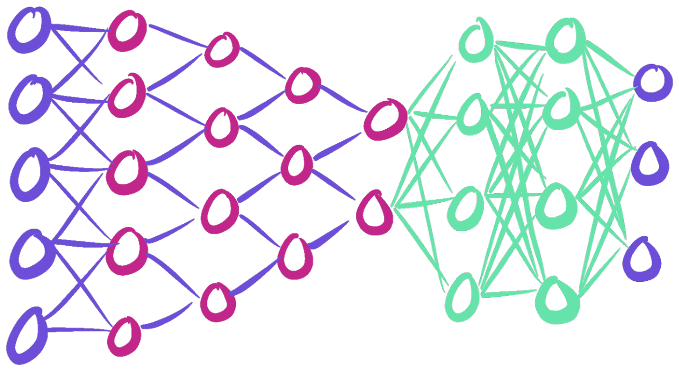


Figure 2.4: A basic diagram of a CNN.

Recurrent Neural Network (RNN)

Similarly to the CNN, the RNN was also first introduced in the nineties by Dr. Jeffery Elman [5]. An RNN is very similar in structure to a regular MLP, but it has the feature that the hidden layer nodes possess recurrent connections. In other words, there are edges of the network that direct back into the node itself rather than just forwards or backwards. This allows the network to hold information through time. Essentially, recurrent connections give a node information about itself from the previous pass of training, while also taking inputs from the previous layer. Firgure 2.5 represents a basic diagram of a recurrent neural networks' topology.

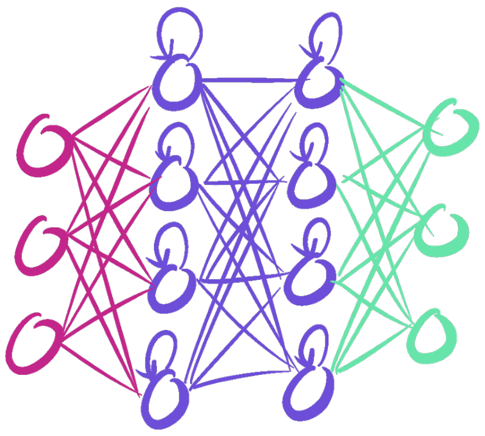


Figure 2.5: A basic diagram of a RNN.

2.4 Training

Given what a neural network's structure actually looks like, and what we have covered about linear algebra and neural network basics thus far, we are ready to move on to the mathematical "meat and potatoes" of what actually makes them work, and what it means for a Neural Network to "Learn." As mentioned before, we will be discussing in detail the mathematics behind the MLP architecture. The entire process can be decomposed into two major parts: *forward propagation* and *backward propagation*(back propagation).

2.4.1 Forward Propagation

Recall, an MLP takes a vector as its input, where the dimension of that input vector corresponds with the total number of features being accounted for. For example, if the goal is to predict the price of gas in your area, then the time of year, price of oil, average distance from other gas stations could all be features used to make this prediction.

To illustrate this mathematically, assume an MLP like the one from Chapter 2.3.1. Let $\mathbf{a}_0$ be the input vector with dimension p_0 of one particular training example. Furthermore, let $\mathbf{a}_{n+1}$ be the output vector of dimension $\mathbf{a}_{n+1}$ after forward propagation. We need to "feed" this information forwards through the network in a way that produces a meaningful result as the output. In the case of our example, some value that represents the price of gas per gallon given the input.

Recall Figure 2.2. There are two functions applied within each neuron. The first function, called the linear basis function, sums the input from all nodes in the previous layer multiplied by some weight w. Second, the neuron outputs some "activation" based on the linear basis function. This activation is the output that serves as the input to the next layer until the final layer is reached.

Now consider the first hidden layer, $\mathbf{a}_1$, with dimension p_1. Since each of the connections between every node in $\mathbf{a}_0$ and every node in $\mathbf{a}_1$ have a corresponding weight, we can represent all of them at once with a weights matrix $\mathbf{W}_0$, where each weight $w_{i,j}$ indicates the specific weight of a connection from node $i \in \mathbf{a}_0$ to node $j \in \mathbf{a}_1$. Also note that $0 \leq i \leq p_0$ and $0 \leq j \leq p_1$. There is also a bias value that must be added separately to each of the nodes in the next layer. This can be done with a vector $\mathbf{b}_0$. This is essentially the same as having a zeroth column as the bias in the weights matrix [6].

So, beginning the forward propagation, let $\mathbf{s}_1$ be the resulting vector from applying the linear basis function to each node from $\mathbf{a}_0$, which can be completed in one step as a matrix multiplication. So, we see that the linear basis function of our input vector is simply:

$$\mathbf{s}_1 = \mathbf{W}_0\mathbf{a}_0 + \mathbf{b}_0$$

Now, we apply some arbitrary activation function $f(x)$ to the vector $\mathbf{s}_1$ to get the activation for our next layer: $\mathbf{a}_1$. It's important to note that a scalar function can take in vector input and produce a vector as its output, applying the function to each individual entry of the vector. This allows for the forward propagation of an entire input to be compacted down to one simple formula [7]:

$$\mathbf{a}_1 = f(\mathbf{W}_0\mathbf{a}_0 + \mathbf{b}_0)$$

Now we have successfully moved from the input layer to the first hidden layer, and the next step is to continue forward. We see that given $\mathbf{W}_0, ..., \mathbf{W}_{n+1}$ and $\mathbf{b}_0, ..., \mathbf{b}_{n+1}$ we can represent forward propagation recursively as follows:

$$\mathbf{a}_j = f(\mathbf{W}_{j-1}\mathbf{a}_{j-1} + \mathbf{b}_{j-1})$$

where $1 \leq j \leq n+1$.

2.4.2 Error

Once the output activation has been reached the output can be compared with target values that represent the expected output of the network. The target values allow for the calculation of error, which is important for two main reasons. First, the error is a metric for measuring the performance of the network given the set of weights it has. Initially, the network's performance is likely to be very poor because it has not yet "learned" from the data. However, over time, with enough training examples, the network's weights will eventually be adjusted to minimize this error. This happens during the back propagation phase which is the second main reason error is important. Calculating error is extremely simple, we just subtract the target values from the output values, which we will call **y**. Let's call the error vector $\boldsymbol{\epsilon}_0$, therefore we have

$$\boldsymbol{\epsilon}_0 = \mathbf{a}_{n+1} - \mathbf{y}$$

The overall error for the entire training set is the sum of the squares of every error value. Taking this sum, we get a value that we'll call $\hat{E}$ that represents the overall performance of the network across all training examples. So, the squared error is

$$\hat{E} = \sum_i \boldsymbol{\epsilon}_{0i} \cdot \boldsymbol{\epsilon}_{0i}$$

where $i \in I$ is the index set of all the training examples.

2.4.3 Back Propagation

The final phase of training is to adjust the weights matrices in order to minimize the squared error. The best way to do this is by calculating the small changes, or partial derivatives between every node from one layer to the next. This is not as intimidating as it seems, though, because the partial derivatives can also be expressed in the vector notation, and the relationship between one layer and the last can be established recursively! The error $\boldsymbol{\epsilon}_0$ represents the error from $\mathbf{a}_{n+1}$. So, we can essentially calculate an "error value" for each layer in the network. Let the vector containing the partial derivatives for $\boldsymbol{\epsilon}_0$ be $\mathbf{d}$. The partial derivatives are the Hadamard product of the error values with the *derivative* of the activation function on the previous layer:

$$\mathbf{d}_0 = \boldsymbol{\epsilon}_0 \odot f'(\mathbf{a}_n)$$

Hence, the error value for $\mathbf{a}_n$ can be calculated with $\mathbf{W}_n$ multiplied by the previous composition

$$\boldsymbol{\epsilon}_1 = \mathbf{W}_n \mathbf{d}_0$$

$$\boldsymbol{\epsilon}_1 = \mathbf{W}_n(\boldsymbol{\epsilon}_0 \odot f'(\mathbf{a}_n))$$

Again, we can now expand this recursive relationship until we reach the input layer.

$$\boldsymbol{\epsilon}_j = \mathbf{W}_{n+1-j}(\boldsymbol{\epsilon}_{j-1} \odot f'(\mathbf{a}_{n+1-j}))$$

again where $1 \leq j \leq n+1$.

Lastly, the formula for the partial derivatives matrix, used for adjusting the weights, can be constructed by taking the outer product of the partial derivatives vector $\mathbf{d}_0$ with the previous activation vector $\mathbf{a}_n$. Since the dimensionality of $\vec{d}$ will be p_{n+1}, and the dimensionality of $\mathbf{a}_n$ is p_n, their outer product produces an $p_{n+1} \times p_n$ matrix representing the partial derivatives of $\mathbf{W}_n$ for one training example. We will call this matrix $\mathbf{C}_n$ and the formula for one particular training example is

$$\mathbf{C}_0 = \mathbf{d}_0 \otimes \mathbf{a}_n$$

Recursively, we get

$$\mathbf{C}_j = \mathbf{d}_j \otimes \mathbf{a}_{n-j}$$

with $0 \leq j \leq n$. This process is repeated for every training example, and after every training example has been assessed a final matrix of the accumulated values is used to update the weights all at once. Let $\mathscr{C}_j$, hence we have

$$\mathscr{C}_j = \sum_i \mathbf{C}_i$$

again where $i \in I$ represents the index set of all training examples. The bias vector also needs to be updated, but since the "activation" for the biases is always one, its the same as taking the sum across all training examples of the partial derivatives vector.

$$\boldsymbol{\beta} = \sum_i \mathbf{d}_j$$

A small proportion of the changes in the weights and biases are then subtracted from the network's actual weights and biases so that there will be less error when training on the next pass of the data. These passes are known as *epochs*, and the small proportion's value is called the *learning rate*. A learning rate that is too high may not be able to converge to a minimum on the error function's surface.

2.5 Activation Functions

In the previous section, we covered the details of what it means when a neural network is training and learning. In these explanations, however, we used $f(x)$ to represent an arbitrary activation function for the neurons. In this section, we will cover a series of well known, and actively used activation functions, and discuss why some are used more predominantly than others.

2.5.1 Sigmoid

The first activation function the classic and original activation function called the *Sigmoid* Activation Function. The Sigmoid function is defined as

$$\sigma(x) := \frac{1}{1 + e^{-x}}$$

and is graphed in Figure 2.6. The purpose of this function is to take the value of

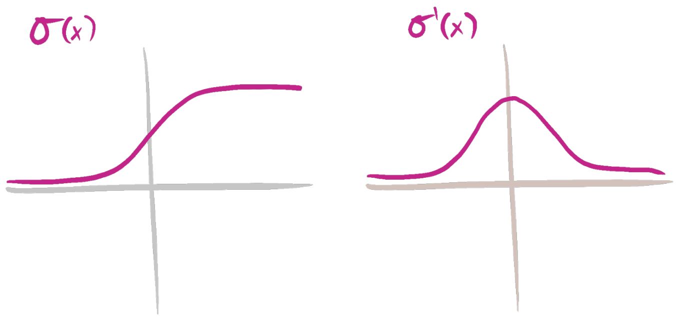


Figure 2.6: Sigmoid activation function (right) and its derivative (left).

each node after applying the linear basis function and "squish" that value into a new value between 0 and 1, which represents the new activation of the next layer. This function is continuous everywhere and has the derivative

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

However, for as reliable and consistent as this function is in neural networks, it also has its own problems. The derivative for very large, or very negative inputs is very close to 0 which can cause the partial derivative matrices to converge to zero, thus not significantly updating the weights and biases ever. This is known as the disappearing gradient problem. However, since its conception there have been dozens of new activation functions introduced into the literature that have taken the place of sigmoid because they are faster, better performing, and are actively being used today.

2.5.2 ReLU

The main successor to the Sigmoid function is the *Rectified Linear Unit*. This is a much simpler activation function defined as

$$\text{ReLU}(x) := \max(0, x)$$

Unlike the sigmoid activation function, the activation of a neuron is unbounded in the positive direction, and if the result from the linear basis function is negative, then there is no firing of the neuron at all. The function graph can be seen in Figure 2.7. This activation function helped spark what some call the end of the second

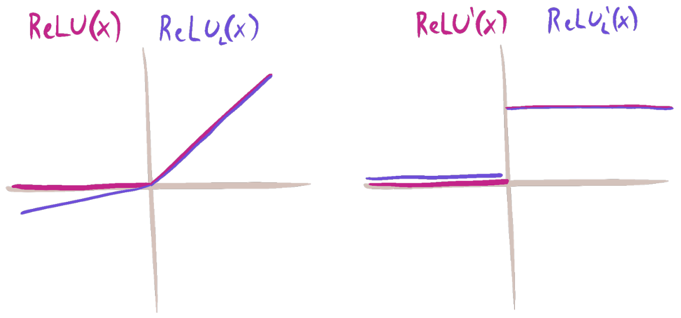


Figure 2.7: ReLU activation functions (right) and their derivatives (left).

A.I. winter. I think its simplicity and performance also make neural networks more accessible as a whole, since the derivative can be represented as

$$\mathrm{ReLU}'(x) := \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

In addition to the standard ReLU function, there is a variant activation function called the Leaky ReLU function, also in Figure 2.7, defined as

$$\mathrm{ReLU}_L(x) := \begin{cases} ax & x < 0 \\ x & x \geq 0 \end{cases}$$

where most often, $0 \leq a \leq 1$. This function has derivative

$$\mathrm{ReLU}_L'(x) := \begin{cases} a & x < 0 \\ 1 & x \geq 0 \end{cases}$$

The key difference is that that there is an activation for values that are less than zero, it's just at a significantly reduced proportion. There is evidence to suggest that this is a slightly better performing activation than the original ReLU, but this claim is mostly observational [8].

2.5.3 Swish

The last category of activation functions are very similar to rectified linear units, but they are continuous functions instead of piece-wise functions. The first represents a sort of "interpolation" of ReLU, and it is called the *Softplus* Activation function.

$$\mathrm{S}^{+}(x) := \ln(1 + e^{x})$$

Interestingly, this function is the anti-derivative of the Sigmoid function! So, it's not too difficult to keep track of because it's part of the Sigmoid family. Figure 2.8 shows the graph of this function.

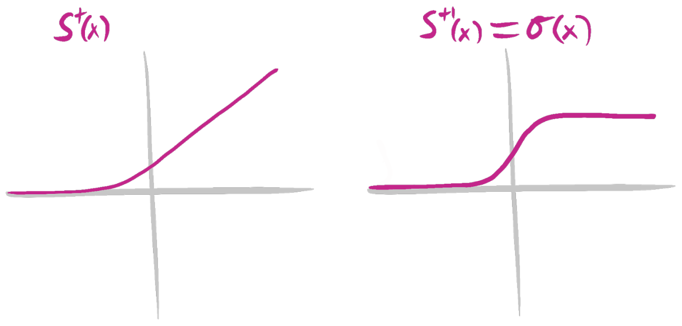


Figure 2.8: Softplus activation function (right) and its derivative, the sigmoid function (left).

However, the next two functions are very interesting, because they are both similar to one another, but much better continuous approximations of the ReLU functions. The first one is called the *Swish* Activation function [9]. The formula for the swish activation function is given partial in terms of the Sigmoid activation function as

$$\mathrm{Sw}(x) := x\sigma(\beta x)$$

and can be seen in Figure 2.9. The derivative is a bit more complicated than sigmoid, though. Because of the chain rule we get the derivative to be

$$\mathrm{Sw}'(x) = \sigma(\beta x) + \beta\mathrm{Sw}(x)(1 - \sigma(\beta x))$$

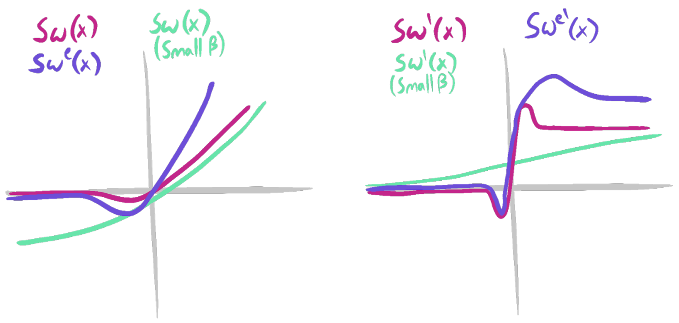


Figure 2.9: Swish and E-swish activation functions (right) and their derivatives (left).

In their paper, this function possesses a parameter β that essentially controls the "smoothness" of the function. What makes the swish activation function really interesting is that as $\beta \to \infty$, the function Sw $\to$ ReLU.

Finally, we will briefly cover another variant of Swish called the *E-swish* Activation Function [10]. The function formula is

$$\text{Sw}_e(x) := \beta x \sigma(x)$$

and it is also represented graphically in Figure 2.9. This function is very similar to the regular swish function, but it adds an extra parameter on the outside, β, which is claimed to have a slightly better result than swish; typically $1 \leq \beta \leq 2$. The derivative is slightly different from that of the Swish function:

$$\text{Sw}_e'(x) = \beta\sigma(x) + \text{Sw}_e(x)(1 - \sigma(x))$$

This concludes the discussion of different activation functions and their importance. Next we will discuss how in why picking the right combination of attributes for a specific neural network can be such a complex task.

Chapter 3

Hyperparameters and the Problems They Cause

Many machine learning algorithms require their own set of parameters to function, and neural networks are certainly no exception! As we will see in this chapter, understanding how to pick the right parameters can create a lot of unforeseen problems.

3.1 Introduction to Hyperparameters

As we have explored during Chapter 2, there are many ways to construct a neural network. It can have arbitrary depth, with an arbitrary number of nodes in each layer. The activation function can vary, etc. Every neural network has a set of parameters like this that must be decided upon when creating the actual network. These are called *hyperparameters*. Hyperparameters are the components of the neural network that are decided upon by the creators, given the context of the problem they are trying to solve. These decisions could be based on prior experience, trial and error, or even arbitrarily. And while there are standard conventions that have become accepted from the thousands of trial and errors over the decades, it is still the case that these conventions are just general guidelines. As it turns out, finding the optimal set of hyperparameters to use on a specific network for a specific problem is an extremely difficult task.

To begin addressing this, let's first think of a way to represent hyperparameters. There is a very nice paper written by Matthias Feurer and colleagues that explains the idea of hyperparameters as part of a *hyperparameter space*. This

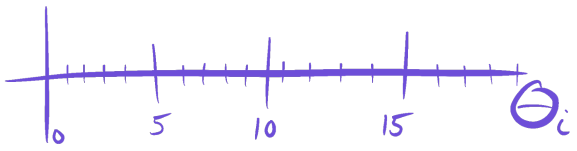


Figure 3.1: Any specific hyperparameter domain.

is a fairly mathematical way to conceptualize hyperparameters, but it illustrates very nicely why trying to search for the best hyperparameters can be so difficult. "Let $\theta_1, ..., \theta_n$ denote the hyperparameters of a machine learning algorithm, and let $\Theta_1, ..., \Theta_n$ denote their respective domains. The algorithm's hyperparameter space is then defined as $\mathbf{\Theta} = \mathbf{\Theta_1} \times \cdots \times \mathbf{\Theta_n}$" [11]. Figures 3.1, 3.2, an 3.3 all illustrate an example of what a hyperparameter space would be in one, two, and three dimensions respectively.

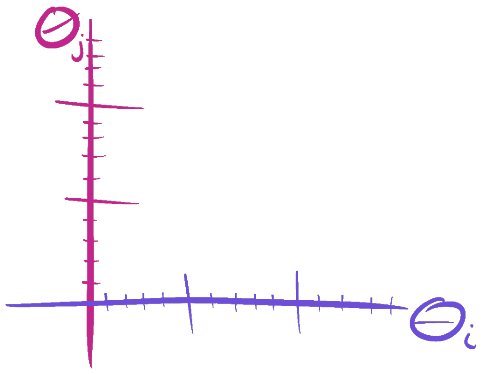


Figure 3.2: A two dimensional hyperparameter space.

From these examples, it is easier to understand what is meant by "high dimensional" hyperparameter spaces. Moreover, by thinking about a hyperparameter

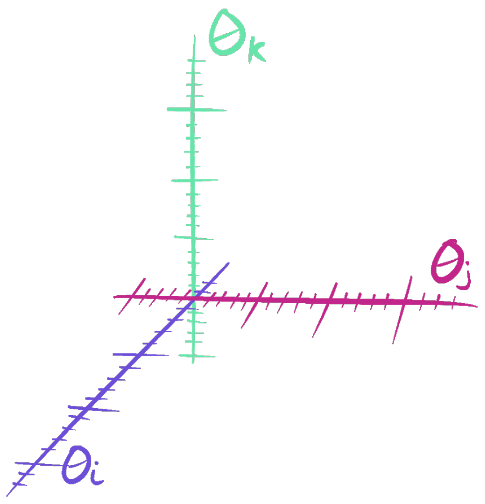


Figure 3.3: A three dimensional hyperparameter space.

in terms of a position in Euclidean Space, it is much easier to think about what it means to pick a set of hyperparameters as a vector being represented in that space. In other words, the number of possible hyperparameters is equivalent to $2^{|\Theta_1| \times \cdots \times |\Theta_n|}$. Hence, there exists at least one point in this space such that it produces optimal results for our neural network. All we need to do is find such a point.

3.2 NP Problems

The problem with finding such a set of hyperparameters is it can be extremely computationally expensive to search for that magic point in the hyperparameter space. Notice that a brute force search for these hyperparameters would result in an algorithm with running time $O(2^n)$. Essentially, as the number of different hyperparameters increases, the more likely it will be impossible for a computer to finish the search before the heat death of the universe. This is what's known as an NP problem.

The world is full of NP problems. These are problems that can only be solved in non-deterministic polynomial time. One of the major issues with this kind of problem is that given a certain, not necessarily large, input size, the computation time becomes unpractical very quickly. An example of this is the “Traveling Salesperson” problem. The idea is, given n cities, the traveling salesperson “will conduct a journey in which each of his target cities is visited exactly once before he returns home” [12]. By analyzing the time complexity of such a problem, we see that “there are at most 2^n sub-problems, and each one takes linear time to solve. The total running time is therefore $O(n^2 2^n)$” [12]. So, as the number of cities n increases, the total calculation time diverges a mind-bending amount. It diverges so much so that the problem become impossible for a computer to calculate before the end of time.

3.3 Shortcuts for solving These Problems

While the brute force approach to solving these problems, there are alternative approaches that don’t involve testing every single possibility. There are several methods currently in use to approximate non-deterministic polynomial problems with polynomial solutions, and this section will be dedicated to explaining a few of those techniques.

3.3.1 Manual, Random, and Grid Search

Currently, there are three predominant ways of searching for hyperparameters, all of which have benefits and limitations. As phrased by Steven R. Young and colleagues: “The three most widely used methods for hyper parameter selection in deep learning are (1) manual search, (2) grid search, and (3) random search” [13] [14]. Manual search is the aforementioned “Trial and Error” approach, which is the least time consuming, but also the least likely to be optimal or near optimal. Grid search is essential a brute force, where an algorithm scans across the hyperparameter space fixing $n-1$ parameters and searching one parameter at a time.

"Grid search wastes resources exploring what could be unimportant dimensions of the hyper-parameter space while holding all other values constant" [13]. And random search can eliminate the problem of wasting resources in unnecessary dimensions of the hyperparameter space, but at, again, a loss in optimality.

3.3.2 Heuristic Search and Genetic Algorithms

In order to tackle these types of problems, people needed to think in an entirely new way. Inspired by biology, heuristics, and holism, Genetic Algorithms (GAs) are a biologically inspired heuristic search or optimization algorithm. And while heuristic solutions are usually not optimal, they are near optimal for a completely negligible proportion of the computational effort.

In an excerpt from Scientific American, Dr. John Holland metaphorically explains why GAs ability to search in high dimensional space is superior to brute force. "As the number of dimensions of the problem space increases, the countryside may contain tunnels, bridges and even more convoluted topological features. Finding the right hill or even determining which way is up becomes increasingly difficult...Genetic algorithms cast a net over this landscape. The multitude of strings in an evolving population samples it in many regions simultaneously" [15]. Therefore, using an evolutionary algorithms approach to optimize the hyperparameters of a neural network seems like a viable approach to solving the problem.

The genetic algorithm for this project will be based on the conventions of Dr. Goldberg's Simple Genetic Algorithm (SGA). The reproduction phase implements methods from Goldberg's SGA such as elitism, hermaphroditic sexual reproduction based on the roulette wheel, and fitness scaling. At the end of the evolution all the results are are expected to converge on hyperparameters that train the networks more quickly and with less error than would be expected from a manual search, grid search, or random search. The next chapter will cover, in detail, the specifics of the implementation from the organization of classess to the specific methods in each class.

Chapter 4

Implementing a Solution

The implementation of this project is done in Python 3, using an object oriented approach, the PEP-8 style guide, and Google documentation conventions. The main additional package used in this project is the Numpy package. Numpy adds its own array like objects on top of python and incorporates highly optimized linear algebra operations such as matrix multiplication, dot product, and outer product into their framework. This package is an expansion of linear algebra for python, and its inclusion allows for optimized linear algebra operations to be the workhorse of the neural networks. The main detailed discussion of the code will center around the most important methods of each class. As a result, there will only be a brief description of each of the supplementary methods that make up the main methods.

4.1 Class Structure

This project is comprised of four main classes nested together, each one instances multiple copies of the class directly below it. The first class is the DeepMLP class. This is the neural network architecture I wrote to serve as the foundation for the entire project. Everything else is built upon this class. It needs to be fast, and well written.

Above this are the two classes that comprise the genetic algorithm: the first is the phenotype class, and the second is the generation class. The phenotype class contains the “genetic” material that encodes all of the hyperparameters for the network. It represents the individual of within a population of a specific generation, and contains a network object and a few methods that allow it to reproduce. The generation class contains the population of phenotypes that exist throughout the evolutionary simulation. It calculates the fitness of all agents of the population and performs methods inspired by, and similar to natural selection in order to converge on an optimal set of hyperparameters.

The entire evolution process is handled by the controller class. This is a class that contains all elements and parameters of the evolutionary algorithm so that the only thing to do to run the simulation is to instance that class. Thus, this lastly takes us to the main script, of which there are two: one for each data set being used. The main script generates its data set and instances the controller class.

A more detailed explanation of the specifics of each class, how it was mad, and some insight into how it functions will be covered throughout the remained of this chapter. This is the implementation of the project, and the main body of work towards this project comes from here.

4.2 DeepMLP Class

The neural network architecture is the most important component of the project. It needs to be written extremely cleanly, and extremely generalized for its implementation in the genetic algorithm to be simple. The DeepMLP class is a generalized MLP object that can be instanced with any number of hidden layers, and any number of nodes in each hidden layer. There are four main methods of this class: two are functional, and two are cosmetic. The first is the constructor, which instances the network.

Listing 4.1: The constructor for the perceptron and multi-layer perceptron.

```
    def __init__(self, X, y, layers, seed=False, DEBUG=False):
        """ Instances the Deep Multi-layer Perceptron object."""
        self.DEBUG = DEBUG

        if seed:
            np.random.seed(1)

        if self.DEBUG:
            np.random.seed(1)

        self.data = [X, y]
        self.depth = len(layers)
        self.layers = layers
        self.w, self.b = self.init_weights()
        self.nab_w, self.nab_b = self.init_weights()

        self.train_errs = []
        self.valid_errs = []

        self.functions = {
        "sigmoid": sigmoid,
        "ReLU": ReLU,
        "leaky": leaky_ReLU,
        "softplus": softplus,
        "swish": swish,
        "e_swish": e_swish,
        }

        plt.style.use('ggplot')
```

The “X” and “y” parameters are the pre-processed data representing the input and expected output respectively. The “layers” parameter corresponds to a list representing the topology of the network. The length of this list corresponds to how many layers the network has, and the value in each index of the list represents the number of nodes in that layer. For example, if layers was the list $[3, 4, 4, 3]$, the network will look like the one in Figure 4.1. Lastly, the “seed” parameter

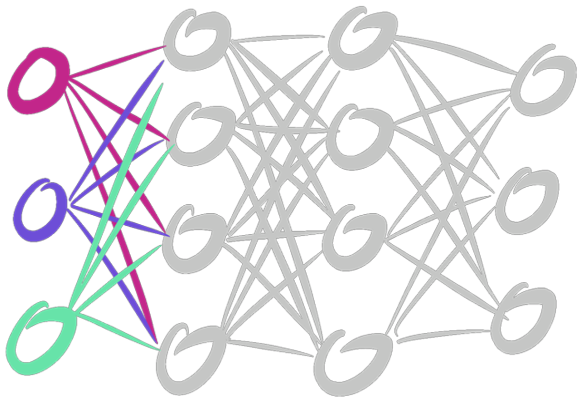


Figure 4.1: A Deep MLP with 3 input nodes, 2 hidden layers with 4 nodes each, and 3 output nodes.

refers to creating a random seed for determining the weights, and the "DEBUG" parameter determines whether or not the debug settings will be turned on or not. This includes various statements that print to terminal for development purposes.

After setting some initial attributes, such as the input and expected output values, the list of all weights and biases are instanced as a random value between $\pm\frac{2}{3}$. There are also two lists that are used to store the training and validation errors throughout the training process. These data are used for visualization, but can also be used to prevent over fitting the data. Lastly, a dictionary of activation functions is stored. This is so that an activation function can be specified and called during forward/back propagation.

After the network is initialized, it needs to be trained. The second major method is the "train" method, which runs a loop of epochs until a specified number is reached.

Listing 4.2: The "train()" method is a loop that calls the "calc_err()" method repeatedly.

```
        while epoch < epochs: #self.error_change():38,
            E_train = self.calc_err(train, train_true, function,
    back_prop=True)
```

```
            E_valid = self.calc_err(valid, valid_true, function)
            self.update(learn_rate, E_train, E_valid)
            epoch += 1
```

This method has several important keyword arguments. “function” is a tring that specifies the activation function being used in the network. “epochs” is a specified number of loops that the train method will execute. Using a fixed number of epochs is a useful constant for experimenting with the GA. Lastly, there is a ”learn_rate” argument and a “split” argument which specify the learning rate to for gradient descent and the split between training and validation data respectively. Inside this loop is the “calc_err()” method which is where all of the mathematics are embedded.

Listing 4.3: “calc_err()” method that forward or backward propagates recursively.

```
    def calc_err(self, X, y, function, back_prop=False):
        """ Calculates activation error and will back prop if
   specified."""
        E = 0
        for i in range(len(X)):
            act = self.forward_prop(X[i], function)
            err = act[-1] - y[i]
            E += np.dot(err, err) / 2

            if back_prop:
                self.back_prop(err, act, function)

        return E
```

The “forward_prop()” and “back_prop()” methods are simply recursive implementations of the mathematics described in Chapter 2.4. Moreover, back propagation applies the derivative to the appropriate activation function when training. Lastly, at the end of the training method are a few DEBUG methods that will print a status along with the weights matrices to the terminal after each epoch. And at the end of training the final weights and biases are displayed to the terminal.

The final two methods provide visuals for human interpretation. The "plot_error()" method plots the training and validation error across all epochs from the training process. This visual is useful for seeing how fast the network trained, and/or any complication during the training process. There is also a "adjacency_matrix()" method that visualizes all wights matrices as a block matrix. This visualization could be useful for performing a topological analysis on the trained network.

4.3 Phenotype Class

Now that the details of the neural networks have been covered, the next step is to look at the details of the GA classes. Of the two main components of the GA, the phenotype class is the first, and represents the lowest level. Like the previous classes, there are three main methods that are called by the generation class that will be covered in detail: the constructor, "init_network()," and "get_fitness()."

Listing 4.4: The constructor for the phenotype class.

```
    def __init__(self, X, y, dna='', max_depth=4, global_epochs
=200, ):
        """ Instances one individual of the population."""
        self.X = X
        self.y = y
        self.dna = dna
        self.max_depth = max_depth
        self.epochs = global_epochs
        self.genes = self.get_genes()

    def get_genes(self):
        """ Gets genes from DNA, either offspring DNA or random
DNA."""
        if self.dna == '':
            self.generate_dna()

        return self.separate_dna()
```

The input parameters are the “X” and “y” arrays, again. This data is needed to instance the neural networks, so it has to travel all the way down from the main script to the controller class to the generation class, and from the generation to the phenotype, all to eventually be used to instance the neural network. The keyword argument “dna” is the encoding for the network’s hyperparameters. It is bit-string that encodes for the aforementioned set of hyperparameters: number of hidden nodes in each hidden layer, learning rate, and the activation function. If an organism is instanced with the default keyword argument for “dna,” then a string of DNA will be randomly created. This is the case when creating generation 0. Otherwise, the “dna” kwarg that is passed into the constructor becomes the DNA bit-string for that organism. This is all handled by the if statement of the “get_genes()” method at the end of the constructor. If no DNA was passed in, then DNA will be generated randomly. Then the DNA is separated into its various smaller components and decoded into the attributes representing the hyperparameters.

Once the organism has been instanced, it is ready for the genes to instance a new neural network.

Listing 4.5: Method that instances a network from the genes that have been decoded from the DNA.

```
    def init_network(self):
        """ Initializes the network based on phenotype DNA."""
        self.layers = self.get_layers()
        self.network = mlp.DeepMLP(self.X, self.y, self.layers,
    seed=True)
```

This method is very simple, because it instances a neural network from the deepMLP class, passing in “X” and “y,” while using the genes for “layers” and eventually the other keyword arguments for the training.

The third method is also very simple, but essential for the generation class.

Listing 4.6: Method that trains the organism’s network and extracts the average

fitness from the last epoch.

```
    def get_fitness(self):
        """ Trains the network for a fixed number of epochs, and
     gets error."""
        self.lr = self.get_learn_rate()
        self.funct = self.get_activation()
        self.network.train(function=self.funct,
                            epochs=self.epochs, learn_rate=self.
    lr)
        self.fitness = self.network.valid_errs[-1]
```

This method calls the aforementioned "train" method from the deepMLP class. So, this is when the GA trains the networks it has instanced. Then, the validation error from the final epoch becomes the fitness for that organism. The idea behind this is that the lower the error of a network after a fixed number of epochs, then the better that network has performed. Moreover, this fitness value contributes to the reproduction and re-population phase in the generation class.

4.4 Generation Class

The generation class is the object that packages the phenotypes together so the controller class can operate everything smoothly. There are 4 important methods in this class called by the controller class. The first one again is the constructor.

Listing 4.7: The constructor for the generation class.

```
    def __init__(self, X, y, pop_size=30):
        """ Initializes Generation 0, and lal data tracking
    attributes."""
        self.ID = 0
        self.X = X
        self.y = y
        self.size = pop_size
        self.generation = self.init_generation()
        self.init_fitness_history()
        plt.style.use('ggplot')
```

The constructor first sets the generation's ID number, then passes in the "X" and "y" arrays from the controller. Afterwards the fixed population size is set followed by the various fitness history arrays. The fitness histories are lists for storing the fitness data across every generation. This is used later for visualizing the evolution process. After the fitness history attributes are created, the last thing the constructor does is actually build generation 0. This is done by instancing n chromosomes, all with random DNA, where n is the specified population size.

After generation 0 has been created, the controller will evolve each phenotype's fitness. Afterwards, the fitness values of each individual is assessed for the reproduction phase. This is repeated over a specified number of generations, and calls two additional methods in the generation class: "calc_fitness()" and "next_generation()."

Listing 4.8: Method that has each phenotype's network train to obtain its fitness.

```
    def calc_fitness(self):
        """ Instances and trains all networks and gets fitness
    from error."""
        self.gen_init_text()

        for i in range(self.size):
            phenotype = self.generation[i]
            phenotype.init_network()
            phenotype.get_fitness()
            self.print_train_info(i)
```

The "calc_fitness()" method instances the neural networks for each chromosome from the chromosome's DNA, and then gets the fitness value for that chromosome. Recall, this involves training the network and getting the validation error from the final epoch. After this process is complete for every organism in the generation, the "organize()" method sorts the generation list to prepare for the re-population phase. Also nested in the "organize()" method is the fitness re-scaling algorithm which re-scales the fitness scores to better distribute the organisms by fitness. This is the same fitness scaling algorithm as the one used in Dr. Goldberg's SGA [16].

When the "calc_fitness()" method has completed, the "next_generation" method is called next. This method calls three additional nested methods.

Listing 4.9: Method that creates the next generation of organisms and replaces the old generation.

```
    def next_generation(self):
        """ Creates a new generation based on elitism and
reproduction."""
        self.new_generation = []
        self.elitism()
        self.repopulate()
        self.update_generation()
```

The elitism method picks the first two organisms from the generation, since the generation has been sorted by fitness, the first two are the most elite organism. Then, they are put together to reproduce two unique organisms for the next generation. These four organisms are then appended for the next generation.

Following this process, the "repopulate()" method applies the same reproduction algorithm, but to random pairs of organisms using the roulette wheel method, and the re-scaled fitness values. This process is repeated until the next generation reaches the same population as the previous generation.

Then, the "update_generation()" method replaces the old generation with the new one, and accumulates the generation ID by one. At the end of each generation some data about the fitness of the generation is printed to the terminal. This whole process repeats until the specified number of generations is reached.

At the end of evolution, several different plotting methods are called to produce visuals of the results. These visuals will be presented in Chapter 5, but below is the source code for some of the graphs that will appear.

Listing 4.10: Several methods used for plotting various results.

```
    def plot_total_fitness(self):
        """ Plot the fitness totals over all generations."""
        plt.title("Total Fitness for each Generation")
        plt.plot(self.total_history, color="black")
        plt.xlabel("Number of Generations")
        plt.ylabel("Fitness")
        plt.show()

    def plot_fitness(self):
        """ Plot the best, average, and worst fitnesses over all
    generations."""
        plt.plot(self.best_history, color="red")
        plt.plot(self.avg_history, color="green")
        plt.plot(self.worst_history, color="blue")
        plt.title("Other Fitness histories for each Generation")
        plt.legend(["Best Fitness", "Average Fitness", "Worst
    Fitness"])
        plt.xlabel("Number of Generations")
        plt.ylabel("Fitness")
        plt.show()
```

The first graph, from “plot_total_fitness()” is a simple graph that plots the total fitness (across every organism in the generation) over the total number of generations. The second graph, from “plot_fitness():” plots the three other fitness metrics that are measured each generation: best of the generation, worst of the generation, and generation average.

4.5 Controller and Main Classes

Finally, at the top of the class totem pole, we have the controller. This class' constructor is the main function that runs the entire program. This style follows closely with object oriented conventions taught to me at Binghamton University, and allows for an overall cleaner and more organized presentation and execution of the source code; the cleaner the code, the easier it is to locate and eliminate bugs!

Listing 4.11: Contructor for the controller class. This runs the entire GA from the neural network training all the way to organism breeding.

```
    def __init__(self, X, y, pop_size=30, num_gens=100):
        """ Gets the data for the networks and runs the evolution methods."""
        self.X = X
        self.y = y
        self.size = pop_size # Population size for each generation.
        self.gens = num_gens # Total number of generations.
        self.main()

    def main(self):
        """ Instances generation 0, evolves the population, plots values."""
        self.pop = generation.Generation(self.X, self.y, pop_size=self.size)
        self.evolve()
        self.print_best_net()
        self.plot_results()
```

The controller passes in “X” and “y” data for training the neural networks. Additionally, the keyword arguments assign the parameters for the GA, namely the population size of each generation and the total number of generations in the whole simulation. For this project, there are two “toy” data sets for the GA to evolve parameters for. The first one represents a simple exclusive OR (XOR) relationship, which is presented in table 4.1.

The second data set is just a simple 4-D input replication. Table 4.2 illustrates a few examples of the input replication relationship.

Finally, the “main()” method of the controller is called at the end of the constructor. This is where everything happens; the first generation, generation 0, is instantiated, that generation is evolved over the specified number of generations,

Table 4.1: Truth table for exclusive or (XOR)

input	**input**	**output**
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.2: Truth table for 4-D input replication.

input	**input**	**input**	**input**	**output**	**output**	**output**	**output**
0	0	0	0	0	0	0	0
0	1	1	0	0	1	1	0
1	1	1	0	1	1	1	0
1	0	1	0	1	0	1	0

and then the results are displayed through the visuals that the lower classes produce all while printing the best network, and its hyperparameters to the terminal.

The controller class is instanced in its own program called "main.py." The main sript generates the toy data and passes it into the controller along with the keyword arguments.

Listing 4.12: Main program that instances the controller and passes in the data for training.

```python
def main():
    X, y = get_data(), get_exp()
    controller.Controller(X, y, pop_size=50, num_gens=100)

main()
```

Again, this is the result of convention and clean code practices. This concludes the detailed description of the implementation of the project. Obviously there is a lot going on, but once everything is unpacked and presented in digestible pieces, the process becomes much easier to understand.

Chapter 5

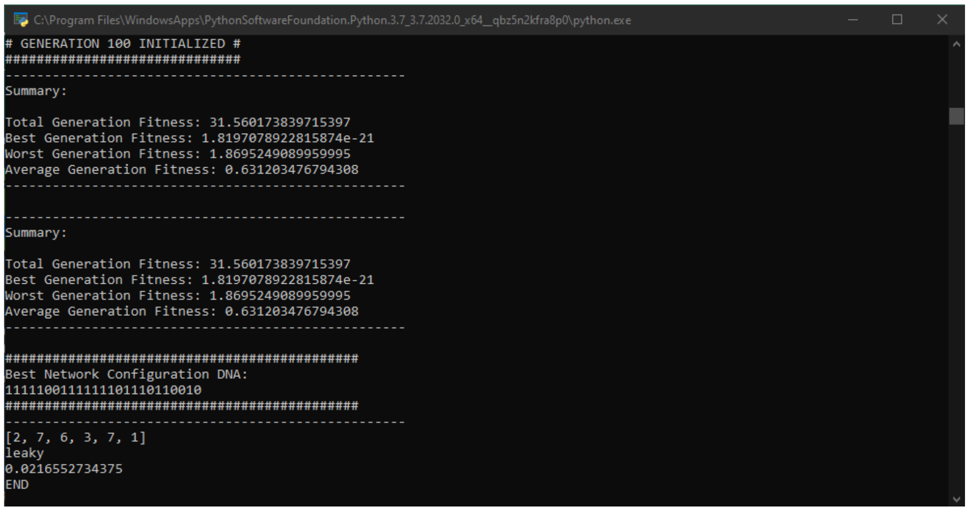
Results from Experiments

In this chapter, we will first cover the results from experiments run on the two sample data sets introduced in the previous chapter. We will first cover the results from the XOR gate data, and then cover the results from the input replication data. After analyzing the results, I will discuss my reflections and interpretations of the results. Finally, we will cover some limitations of this project and introduce future work that can combat these limitations and pave a direction for more work to be done. The parameters chosen for the genetic algorithm in both experiments are 50 organisms per generation, and 100 generations of evolution in the entire simulation.

5.1 XOR Gate

Throughout the programs' evolution phase various bits of information get printed to the terminal after each generation. Figure 5.1 shows the Python terminal after the last generation has finished training its neural networks. At the end of the evolution, the best performing phenotype prints its DNA, and decoded network configuration to the terminal.

In this case, the DNA bit-string was "11111001111111101110110010" which decodes to mean the layers list is [2, 7, 6, 3, 7, 1], the activation function is ReLU_L,



```
C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.7_3.7.2032.0_x64__qbz5n2kfra8p0\python.exe
# GENERATION 100 INITIALIZED #
##############################
---------------------------------------------------
Summary:

Total Generation Fitness: 31.560173839715397
Best Generation Fitness: 1.8197078922815874e-21
Worst Generation Fitness: 1.8695249089959995
Average Generation Fitness: 0.631203476794308
---------------------------------------------------

---------------------------------------------------
Summary:

Total Generation Fitness: 31.560173839715397
Best Generation Fitness: 1.8197078922815874e-21
Worst Generation Fitness: 1.8695249089959995
Average Generation Fitness: 0.631203476794308
---------------------------------------------------

#############################################
Best Network Configuration DNA:
1111100111111101110110010
#############################################
---------------------------------------------------
[2, 7, 6, 3, 7, 1]
leaky
0.0216552734375
END
```

Figure 5.1: Python Terminal after the program finished evolving (XOR).

and the learning rate is about 0.0217. In general, with other trials of evolving neural networks, similar results have been reached. That is, in all cases the best network configuration is always has multiple hidden layers. That is, we don't see the network evolving to have less hidden layers because it doesn't converge on a solution faster. This makes sense because XOR logic gates will not work if the network has no hidden layers, so the GA avoiding evolving less hidden layers makes sense. Another interesting property from running the evolution multiple times is that the optimal network tends to settle on ReLU type functions. The most common result was Leaky ReLU or ReLU. This result further reinforces the out performance of Sigmoid that has been historically documented.

Next, we will look at the plots produced by the program after the evolution took place. In Figure 5.2, we see that the overall fitness reaches its best values only about twenty generations into the evolution. After this point it is likely that the best network configurations have already been discovered, and any further variances of the fitness value is due to the new organisms of the generation. Since they are new, there is no guarantee that they will be better than their parents for instance.

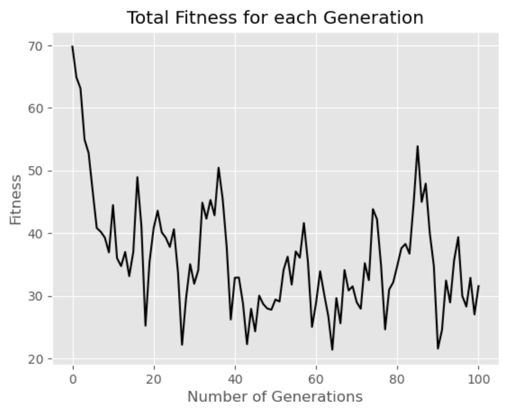


Figure 5.2: Plot of the sum of all fitness scores for each generation (XOR).

This Idea is further reinforced in Figure 5.3. It appears that the near opti-

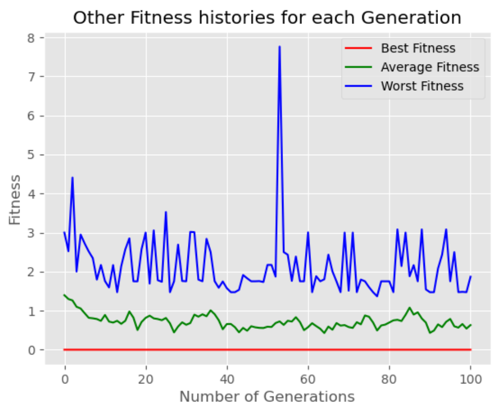


Figure 5.3: Plot of the best, average, and worst fitness in each generation (XOR).

mal network configuration is found immediately at the beginning of the evolution, since it is at this point that the best fitness score appears to remain static for the entire evolution; however, each generation is slightly optimizing the network negligible amounts each generation.

There is also a lot of variance in the worst of the generation. This is likely because of the random variations that can occur after two organisms reproduce.

We see around generation 55 or so that there was some sort of mutation or “birth defect” that cause one of the networks to perform horribly that generation. Lastly, the average fitness settles into a stable minimum state around twenty generations into the evolution akin to the total fitness graph from figure 5.2. This makes sense because the average fitness plot is just a scaled down version of the total fitness plot.

The final plot produced by the program is Figure 5.4: the plot of the training and validation of the best network configuration from the final generation. Every

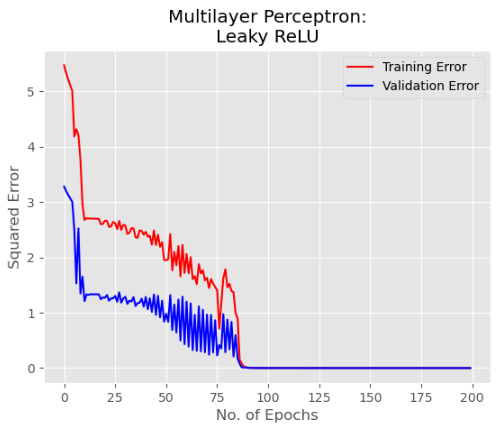


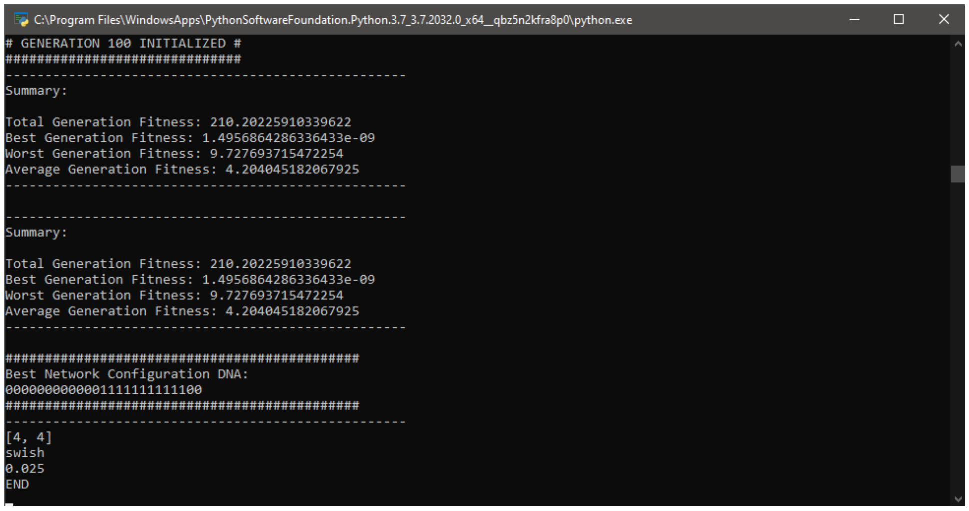
Figure 5.4: $\hat{E}$ value for each epoch of the best network configuration at the end of the evolution process (XOR).

network trained for 200 epochs. The shear drop off in the training error in the first couple dozen epochs is interesting. After this point the decent to its final state is more gradual. Another interesting structure to note is series of fluctuations in the middle of the training process before finally converging on the proper wights and biases at around epoch 85.

5.2 Input Replication

Now we will look at the results from scenario 2, the input replication. While the results from scenario 1 are interesting and insightful, I believe the results from scenario 2 truly illuminate the power of Genetic Algorithms for solving these types of problems. Recall, the input replication data is trying to train the network to replicate the input as its output; reference Table 4.2 as the truth table for this input.

Let's begin with Figure 5.5, which is a terminal screenshot of the end of the evolution, with the best performing phenotype's DNA, and decoded network configuration.



```
C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.7_3.7.2032.0_x64__qbz5n2kfra8p0\python.exe
# GENERATION 100 INITIALIZED #
##############################
---------------------------------------------------
Summary:

Total Generation Fitness: 210.20225910339622
Best Generation Fitness: 1.4956864286336433e-09
Worst Generation Fitness: 9.727693715472254
Average Generation Fitness: 4.204045182067925
---------------------------------------------------

---------------------------------------------------
Summary:

Total Generation Fitness: 210.20225910339622
Best Generation Fitness: 1.4956864286336433e-09
Worst Generation Fitness: 9.727693715472254
Average Generation Fitness: 4.204045182067925
---------------------------------------------------

#############################################
Best Network Configuration DNA:
0000000000001111111111100
#############################################
---------------------------------------------------
[4, 4]
swish
0.025
END
```

Figure 5.5: Python Terminal after the program finished evolving (Input Replication).

The resulting DNA bit-string for this example was "0000000000001111111111100." This decodes to a simple [4, 4] layer structure, the Swish activation function Sw, and the learning rate is 0.025. This is a powerful result because it shows that the network evolved to have the highest possible learning rate, and the least possible number of hidden layers. This, intuitively, would be the best performing result because the network's task is so simple. If there are no hidden layers, the weights matrix will simply adjust to all be ones, and the biases will adjust to all be zeros. The genetic algorithm was able to find this solution from training a total of 5000 networks. A grid search of the same hyperparameter space is comprised of 16,777,216 network configurations, which would have taken my computer *months* to run completely, instead of the roughly 1 hour that this took.

Now let's examine Figure 5.6: the plot of the total fitness from every generation. This plot is very similar to the fitness over time of the XOR data. We see

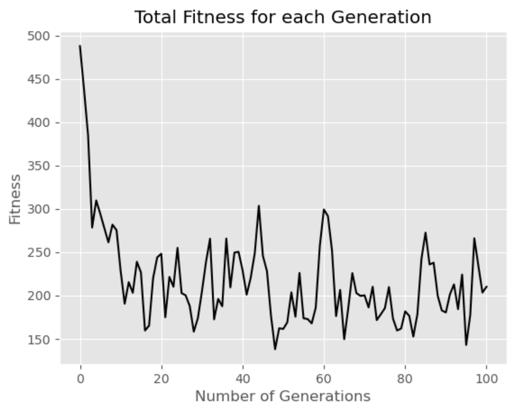


Figure 5.6: Plot of the sum of all fitness scores for each generation (Input Replication).

it converges towards a set of near optimal solutions after about 20 epochs, and then continues searching to make minuscule improvements thereafter. Figure 5.7 is also very similar to the fitness plots from the XOR data. We see that near optimal solutions are found only after about 5 generations, however, in this plot, there seems to be many more fluctuations of the worst performing organisms in a given generation. This is most likely because the best network configuration involves no hidden layers, so if an organism mutates it's hidden layer genes it could go having from no hidden layer to having 1 or two, which we would expect to not do very well.

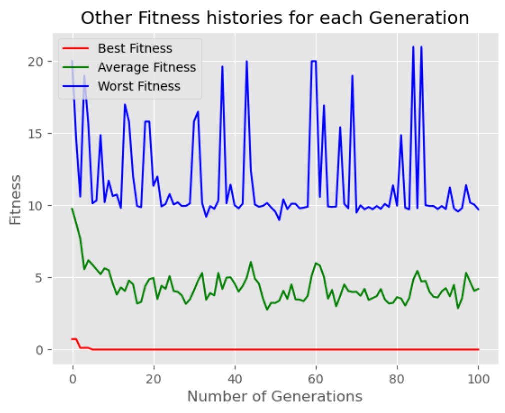


Figure 5.7: Plot of the best, average, and worst fitness in each generation (Input Replication).

Lastly, we will look at Figure 5.8. This plot is very interesting because it shows how fast the error converges. It only takes about 25 epochs for the network to essentially be done training. Overall, for this example I do not think the activation

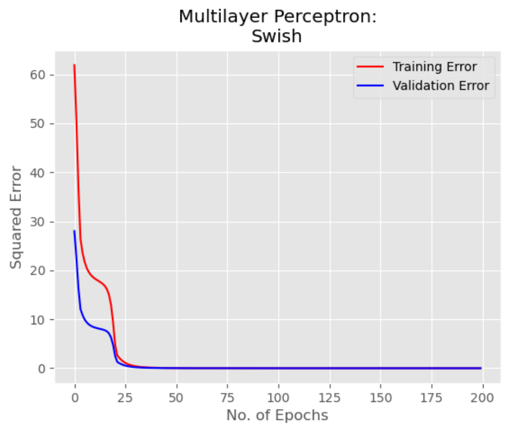


Figure 5.8: $\hat{E}$ value for each epoch of the best network configuration at the end of the evolution process (Input Replication).

function is nearly as important as the network's topology, although most results end with the Sigmoid function *NOT* being used. In this particular example it happened to be the Swish function, but in other runs of the evolution, ReLU, Leaky ReLU, and E-swish were all emergent results.

5.3 Conclusions

Evolutionary Algorithms are an essential component to driving machine learning progress forwards. Manual search, random search, and Grid search are outdated methodologies that should be replaced by faster, and more optimal searching solutions, like evolutionary algorithms.

The input replication result shows a network that was predisposed to having several hidden layer evolve to have none. It was able to find this solution from training a total of 5000 networks, which implies it searched less than 5000 possible parameter combinations. By contrast, A grid search of the same set of hyperparameter would have to train 16,777,216 networks. Although the grid search would most likely produce a more optimal configuration, it would be more optimal only negligibly so. On larger scales, the benefits to using a evolutionary algorithms are astronomical to the efficiency and optimization to our machine learning infrastructure.

5.4 Limitations and Future Work

The most obvious next step of exploration is with data. The use of the two tow data sets for this project are perfect for discovering the capabilities of the Genetic Algorithm approach to searching for Hyperparameters. However, without the validation of a data set that mimics the real world more accurately, there is no good way to emphasize the application potential in real world machine learning. It should be noted that the computational power required to execute this would also need to be greatly increased, hence why this project deals with the toy data sets.

The best way to approach this problem is to implement a well developed, stable, and consistent neural networks package from python such as TensorFlow. This would allow for the main focus of the project, or research, to be about exploring different large scale data sets, and different types of evolutionary algorithms. Additional features could be added to this framework such as something to signal the evolution process to stop, so a fixed number of generations is not needed to govern the process. This would save tremendously on computational effort, and thus potentially be good for implementing in the everyday model training process.

Appendix A

Source Code

All of the source code for this project can be accessed on GitHub with the following link:

https://github.com/cwolosh1/hyperparameter-optimization

Bibliography

[1] B. Mehlig, “Artificial neural networks,” 2019.

[2] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[5] J. L. Elman, “Finding structure in time,” *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[6] B. Kröse and P. van der Smagt, *An Introduction to Neural Networks*. University of Amsterdam, 1996. [Online]. Available: https://books.google.com/books?id=M7FTNQAACAAJ

[7] G. Sanderson, “But what is a neural network? — deep learning, chapter 1,” *YouTube*, Oct 2017.

[8] B. Xu, N. Wang, T. Chen, and M. Li, “Empirical evaluation of rectified activations in convolutional network,” *arXiv preprint arXiv:1505.00853*, 2015.

[9] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[10] E. Alcaide, "E-swish: Adjusting activations to different network depths," *arXiv preprint arXiv:1801.07145*, 2018.

[11] M. Feurer, J. T. Springenberg, and F. Hutter, "Initializing bayesian hyperparameter optimization via meta-learning," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[12] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, *Algorithms*. McGraw-Hill Higher Education, 2008.

[13] S. R. Young, D. C. Rose, T. P. Karnowski, S.-H. Lim, and R. M. Patton, "Optimizing deep learning hyper-parameters through an evolutionary algorithm," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 2015, p. 4.

[14] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[15] J. H. Holland, "Genetic algorithms," *Scientific American*, vol. 267, no. 1, pp. 66–73, 1992. [Online]. Available: http://www.jstor.org/stable/24939139

[16] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.