

SSIE 519 Final Project Report:

Neural Network Hyperparameter Optimization Using Genetic Algorithms

Christopher Woloshyn
cwolosh1@binghamton.edu

December 2019

Contents

1	Introduction	4
2	Background on the Application	4
2.1	NP Problems	5
2.2	Introduction to Hyperparameters	5
2.3	What To Do?	7
3	Background on the Approach	8
3.1	Background on the Neural Network Architectures	9
3.1.1	Perceptron	9
3.1.2	Multi-Layer Perceptron	11
3.2	Background on the Genetic Algorithm	13
4	Details of Implementation	14
4.1	Neural Network Classes	14
4.2	Chromosome Class	17
4.3	Generation Class	19
4.4	Controller and Main Classes	22
5	Description of Results	24
6	Conclusions	28
6.1	Limitations	28
6.2	Refinement	29
6.3	Future Work	29
A	Appendix: Source Code	31

List of Figures

1	Any specific hyperparameter's domain.	6
2	A two dimensional hyperparameter space.	7
3	A three dimensional hyperparameter space.	7
4	A perceptron with 4 input nodes and 3 output nodes.	9
5	A multi-layer perceptron with 4 input nodes, 5 hidden nodes, and 3 output nodes.	12
6	Python Terminal during the algorithm's evolution phase.	24
7	Python Terminal after the program finished evolving.	25
8	Plot of the sum of all fitness scores in a given generation	25
9	Plot of the best, average, and worst fitness in a given generation.	26
10	Plot of the best and worst scaled fitness scores in a given generation.	27
11	\hat{E} value for each epoch of the best network configuration.	27

Listings

1	The constructor for the perceptron and multi-layer perceptron.	14
2	Main training method that forward and backward propagates.	15
3	Method that plots the error over the total number of epochs.	17
4	The constructor for the chromosome class.	17
5	Method that instances a network from the decoded DNA.	18
6	Method that trains the organism's network and extracts the average fitness from the last epoch.	19
7	The constructor for the generation class.	19
8	Method that has each organism train to obtain its fitness.	20
9	Method that creates the next generation of organisms and replaces the old generation.	20
10	Several methods used for plotting various results.	21
11	Constructor for the controller class; runs the entire GA from neural network training to organism breeding.	22
12	Main program that instances the controller.	23

1 Introduction

Artificial Neural Networks (ANNs) and Genetic Algorithms (GAs) are two substantial components of applied soft computing. Both are based on digitally mimicking the phenomena of nature in order to solve problems that traditional algorithmic thinking cannot. To name a few applications, neural networks are remarkable for uncovering powerful relationships in large amounts of data, and genetic algorithms can efficiently search or optimize a set of parameters to a function. Yet, as helpful as they may be, there are still some limitations to what can be done with both neural networks and genetic algorithms. For instance, neural networks can have a large number of *hyperparameters* influencing their overall performance; searching for these hyperparameters can be a computationally burdensome task, but more detail about this in section 2.

The goal of this project is address the task of searching for optimal hyperparameters through the use of a genetic algorithm. In other words, the neural network will learn from a given set of data, and the genetic algorithm will search for the best hyperparameters that help the network learn the fastest. Additionally, this project aims to demonstrate expertise in the core concepts of the course, such as the perceptron and multi-layer perceptron neural network architectures, and Dr. Goldberg's Simple Genetic Algorithm (SGA) [1] implementation of a genetic algorithm. The demonstration from this report should showcase a thorough knowledge of the subject, as well as establish a strong foundation for future work that can be done.

To overview the report in more detail, section 2 will provide an in depth derivation and explanation of the problem to be solved. Section 3 will overview the necessary mathematical frameworks for implementing both the neural networks and the genetic algorithm, with section 4 walking through some of the key algorithms as part of the implementation, with commentary. Following this, section 5 will present the project's results and associated figures. Finally, section 6 will regard the project's limitations, areas for improvement, and plans for future work to be done.

2 Background on the Application

This section will establish the necessary background for understanding the principles of soft computing that will be applied. Moreover, this section will establish the core problem that the work of

this problem addresses.

2.1 NP Problems

The world is full of what we call 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” [2]. 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 $O(n^2 2^n)$ ” [2]. 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 heat death of the universe.

In order to tackle these types of problems, people needed to think in an entirely new way. Inspired by biology, heuristics, and holism, the sub-field of “Soft Computing” was born. Ever since the 1980’s, artificial neural network technology has grown tremendously. The design and construction of faster computers has drastically increased the accessibility of this technology, and has thus caused a surge in its use over the past decade in particular. Artificial Neural Networks lie at the heart of data science and machine learning; their implementation permeates throughout the practices of the biggest tech companies in existence, and their rapid development shows how powerful of a tool they are for solving modern, data driven problems.

2.2 Introduction to Hyperparameters

When constructing an artificial neural network, there are a set of parameters that must be decided upon when creating the actual network. These parameters are called *hyperparameters*. Since the network architectures used for this project are only a single-layer and multi-layer perceptron (SLP, MLP), the scope of the discussion about hyperparameters will be reduced to these architectures. For example, some hyperparameters could include the learning rate, the number of hidden layer nodes, where 0 nodes means we are using a SLP, the ratio between training data and validation

data, etc. 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.

As it turns out, finding the optimal hyperparameters for a neural network is an NP problem. So ironically enough, neural networks, which aimed to solve NP problems nearly optimally in P time, have reintroduced a new NP problem. 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 is a fairly mathematical think about hyperparameters, but it illustrates very nicely why trying to search for the best hyperparameters is an NP problem. “Let $\theta_1, \dots, \theta_n$ denote the hyperparameters of a machine learning algorithm, and let $\Theta_1, \dots, \Theta_n$ denote their respective domains. The algorithm’s hyperparameter space is then defined as $\Theta = \Theta_1 \times \dots \times \Theta_n$ ” [3]. Figures 1, 2, and 3 all illustrate an example of what a hyperparameter space would be in one, two, and three dimensions respectively. From these examples, it is easier to understand what is meant by “high dimensional” hyperparameter spaces. Moreover, by thinking about a hyperparameter 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 $2^{|\Theta_1| \times \dots \times |\Theta_n|}$. So, a brute force search of hyperparameters would be an algorithm with running time $O(2^n)$.



Figure 1: Any specific hyperparameter’s domain.

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” [4] [5]. 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.

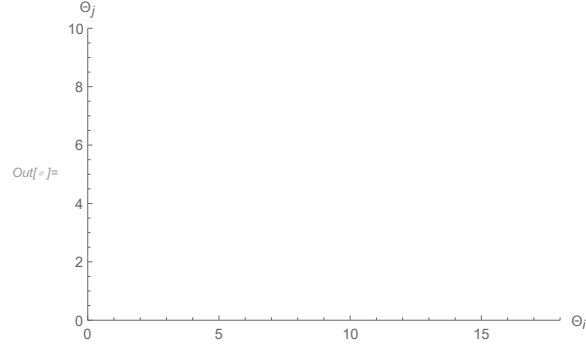


Figure 2: A two dimensional hyperparameter space.

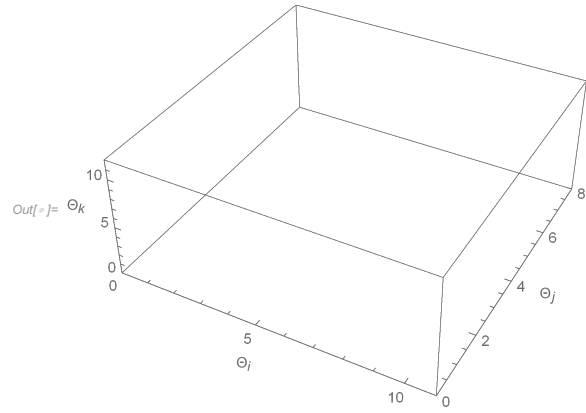


Figure 3: A three dimensional hyperparameter space.

Grid search is essentially 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” [4]. And random search can eliminate the problem of wasting resources in unnecessary dimensions of the hyperparameter space, but at, again, a loss in optimality.

2.3 What To Do?

If the methods of applied soft computing aim to solve NP problems nearly optimally in polynomial time, then by extension, the methods of applied soft computing can optimize the hyperparameters of a neural network. In addition to artificial neural networks, one of the core foci of SSIE 519: Applied Soft Computing is genetic algorithms. And one of the primary applications of a GA is heuristic optimization, or search methods.

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” [6]. Therefore, using an evolutionary algorithms approach to optimize the hyperparameters of a neural network seems like a viable approach to solving the problem.

3 Background on the Approach

The goal of this project is to create a framework for various neural network architectures such that they can be instanced and trained very easily. With a framework that creates and trains neural networks in an organized and simple fashion, it will be very easy to implement a genetic algorithm on top of this network architecture.

It is important to note that there is one fundamental difference between this approach and the approach developed in class. The implementation of this project treats each layer of the neural network as a single vector, as opposed to a series of individual inputs. This allows for two very beneficial things: First, the mathematical representations of the networks become significantly simplified, assuming a non-trivial background in linear algebra [7]. This, of course, translates to an equally significant simplification of the code. Second, it allows for the use of a Python resource called Numpy, which significantly optimizes linear algebra operations making the network train faster, which will be extremely important for the evolution of the genetic algorithm to occur more quickly.

The background of this different approach is outlined in the following subsection along with the background on the genetic algorithm approach, which follows very closely to the methods covered in class akin to Dr. Goldberg’s Simple Genetic Algorithm. Thus, the components for discussion are separated into two major parts. These parts will provide a foundation when considering the details of implementation in section 4.

3.1 Background on the Neural Network Architectures

For this project, two different, very basic neural network architectures have been created for the genetic algorithm. The first network is a simple perceptron, using the linear basis function, and the sigmoid activation function, and the second, is a more complicated multi-layer perceptron. First consider the perceptron architecture. This will then be expanded upon for discussing the multi-layer perceptron.

3.1.1 Perceptron

It is composed of an input vector with dimension n , and an output vector with dimension m . The connections between every node in the input layer with every node in the output layer can thus be represented with a weights matrix \mathbf{W} , where each weight $w_{i,j}$ indicates the specific weight of a connection from node i to node j . Also note that $0 \leq i \leq m$ and $0 \leq j \leq n$. Moreover, the biases will be added separately as a vector \vec{b} after the matrix vector product. This is essentially the same as having a zeroth column as the bias in the weights matrix [8]. By representing the neural network from a linear algebra perspective, it intuitively follows that the linear basis function is simply a matrix-vector product producing a vector with dimension m . This architecture is represented visually in figure 4.

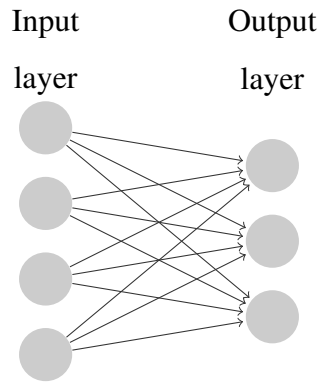


Figure 4: A perceptron with 4 input nodes and 3 output nodes.

To illustrate this mathematically, let $\vec{a}^{(0)}$ be the input vector of a specific training example, and let $\vec{a}^{(1)}$ be the output vector after forward propagation. Additionally, let $\vec{s}^{(1)}$ be the output vector *before* the sigmoid activation function has been applied. So, we see that the linear basis function

of our input vector is simply:

$$s^{(1)} = \mathbf{W}a^{(0)} + \vec{b}$$

Moreover, recall that the sigmoid activation function is

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

It's important to realize that a scalar function such as $\sigma(x)$ 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:

$$a^{(1)} = \sigma(\mathbf{W}a^{(0)} + \vec{b})$$

[7]. This is an amazing simplification for representing the entire process of forward propagation in one simple line, and it hopefully isn't too inaccessible for those a little weaker with mathematics.

Similarly, calculating error and the partial derivatives are very similar to the previous method, but are also cleaned up considerably through the vector notation. Given the previous output $a^{(1)}$, and let \vec{y} be the vector for the expected output.

$$\vec{\epsilon} = a^{(1)} - \vec{y}$$

So, the squared error is

$$\hat{E} = \sum_i \vec{\epsilon}_i \cdot \vec{\epsilon}_i$$

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

The formula for the partial derivatives can be expressed in the vector notation using a special type of matrix operator known as a Hadamard Product. This operation takes two matrices (or vectors) 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.

$$\vec{d} = \vec{\epsilon} \odot a^{(1)} \odot (1 - a^{(1)})$$

Lastly, the formula for the partial derivatives matrix can be constructed by taking the outer product of the partial derivatives vector \vec{d} with the input vector. Since the dimensionality of \vec{d} will be m ,

and the dimensionality of the input is n . Hence, their outer product produces an $m \times n$ matrix representing the partial derivatives for one training example. This formula for one particular training example is

$$\mathbf{C} = \vec{d} \otimes a^{(0)}$$

and the final weights matrix accumulated from every training example, for the purposes of batch updating, is

$$\mathcal{C} = \sum_i \mathbf{C}_i,$$

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.

$$\vec{B} = \sum_i \vec{d}$$

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

3.1.2 Multi-Layer Perceptron

The second network architecture is a modification of the first, with the addition of a hidden layer between the input layer and the output layer. Thus, the mathematics of the multi-layer perceptron is quite similar to the perceptron, with a few additions. First, there is now a hidden layer vector comprised of h nodes, and two weights matrices $\mathbf{W}^{(0)}$ and $\mathbf{W}^{(1)}$ where $\mathbf{W}^{(0)}$ is an $h \times n$ matrix, and $\mathbf{W}^{(1)}$ is an $m \times h$ matrix. Figure 5 illustrates the multi-layer perceptron with 1 hidden layer as discussed.

The modification for the forward propagation will be as follows:

$$a^{(1)} = \sigma(\mathbf{W}^{(0)}a^{(0)} + b^{(0)})$$

which goes from the inputs to the hidden layer, and

$$a^{(2)} = \sigma(\mathbf{W}^{(1)}a^{(1)} + b^{(1)})$$

which goes from the hidden layer to the output layer.

Additionally, calculating the error vector and the squared error \hat{E} is exactly the same, the only difference being the use of $a^{(2)}$ as the output layer activation. However, back propagation

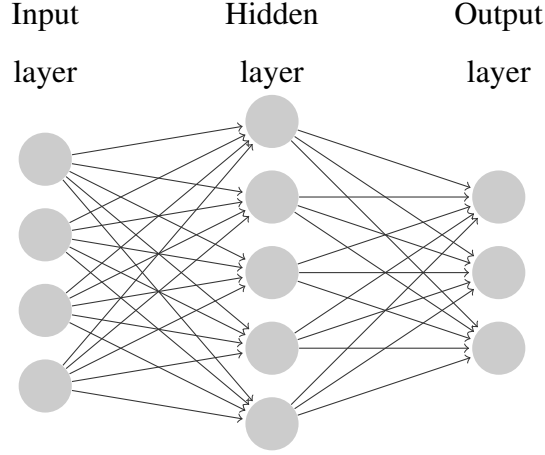


Figure 5: A multi-layer perceptron with 4 input nodes, 5 hidden nodes, and 3 output nodes.

accumulates an additional layer of complexity since there is a hidden layer to move through. Using the same formula to calculate the partial derivatives for the transition from hidden layer to output layer,

$$\vec{d}^{(2)} = \vec{\epsilon} \odot \vec{a}^{(2)} \odot (1 - \vec{a}^{(2)})$$

we have $\vec{d}^{(1)}$. Note that this vector has dimension h . So, we apply the same formula to obtain the partial derivatives matrix exactly as in the previous architecture,

$$\mathbf{C}^{(1)} = \vec{d}^{(1)} \otimes \vec{a}^{(1)}$$

and accumulation of the result over all training examples:

$$\mathcal{C}^{(1)} = \sum_i \mathbf{C}_i^{(1)}$$

$$\vec{B}^{(1)} = \sum_i \vec{d}^{(1)}$$

Now, in order to collect the partial derivatives vector for the hidden layer. This is done by considering each connection between the hidden layer and the output layer. Hence, the new partial derivative vector can be calculated by

$$\vec{d}^{(0)} = \mathbf{W}^{(1)} \vec{d}^{(1)}$$

where $\vec{d}^{(0)}$ has dimension n . Then, the partial derivatives matrix for a certain training example will be

$$\mathbf{C}^{(0)} = \vec{d}^{(0)} \otimes \vec{a}^{(0)}$$

and the accumulated partial derivatives for the weights and biases can be calculated by

$$\mathcal{C}^{(0)} = \sum_i \mathbf{C}_i^{(0)}$$

$$\vec{B}^{(0)} = \sum_i \vec{d}^{(0)}$$

Finally, after all that, the only thing left to do is use these new matrices to update the weights and biases according to the learning factor at the end of the epoch. Overall, it's important to note the usefulness of the matrix-vector representation of the neural network.

3.2 Background on the Genetic Algorithm

With the neural network framework in place, it is easy to more clearly outline the important hyperparameters being applied by the genetic algorithm. This will outline how the genetic algorithm will be built to optimize these parameters. There are only three hyperparameters that this project considers. The first is the architecture—whether to use a perceptron or multi-layer perceptron. The second is the learning factor; as mentioned before, the learning factor determines how much the weights and biases are changed each epoch. And the third is the number of nodes in the hidden layer. As mentioned in class, it is important to tinker with the overall neural network structure to find what is best for the current data set. So, the genetic algorithm will be doing all of this automatically, which will be a much faster and more exhaustive search of the hyperparameter space.

This genetic algorithm will follow the conventions of Dr. Goldberg's Simple Genetic Algorithm. This is represented in the class structure of the project software with a chromosome class, generation class, and controller class. The chromosome class represents the individual organism of the SGA, and contains the "DNA", the actual neural network, and a fitness value corresponding to its performance after training. The generation class instances a population of chromosomes, accounts their fitness scores, and implements the reproduction phase. The reproduction phase implements methods from Goldberg's SGA such as elitism, hermaphroditic sexual reproduction based on the roulette wheel, and fitness scaling. Lastly, the controller class is the actual program that pre-processes the data, instances the generation, and evolves the organisms. The generation class is instanced based on a specific population size and evolves over a specified number of generations. At the end of the evolution all the results are presented with visually intuitive graphics

produces from attribute of the various classes included by design. Many of which are plots of fitness or error over time. The next section will address the specifics of the code that powers this process, analyzing the predominant methods of each class.

4 Details of Implementation

The implementation of this project is done in Python 3, using an object oriented approach, the PEP-8 style guide, and the google documentation conventions. The main additional package used in this project is the Numpy package. Numpy adds its own array like objects and incorporates highly optimized linear algebra operations such as matrix multiplication, dot product, and outer product into their framework. This package is purely 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 sub-methods that are nested in these main methods.

4.1 Neural Network Classes

The neural network architecture is the most important component of the project. It needs to be written extremely cleanly for its implementation in the genetic algorithm to be simple. For both the perceptron and multi-layer perceptron classes, there are three main methods of the class that are called by the SGA. The first is the constructor, which instances the network.

Listing 1: The constructor for the perceptron and multi-layer perceptron.

```
59     def __init__(self, data, expected, rate, split=0.7, batch=1., \
60                 epochs=0, seed=False, DEBUG=False):
61         """ Instantiates a Perceptron neural network."""
62         self.DEBUG = DEBUG
63         self.seed = seed
64
65         if self.seed:
66             np.random.seed(1)
67
68         self.rate = rate
69         self.split_data(data, expected, split, batch)
```

```

70     self.set_layer_dims()
71     self.init_weights()
72     self.init_error(epochs)

```

The “data” and “expected” parameters are pre-processed input data representing the input and expected output respectively. The “rate parameter” corresponds to the learning rate. “split”, “batch”, and “epochs” are all keyword arguments corresponding to the ratio between training and validation data, proportion of the data used in one epoch’s batch (although this currently is not implemented), and the specified number of epochs, where zero refers to training until the validation error function begins to increase, respectively. Lastly, the “seed” parameter refers to creating a random seed for determining the weights, and the “DEBUG” determines whether or not the debug settings are turned on or not. This includes various statements that print to terminal etc.

After setting some initial attributes, there are four sub-methods that set most of the remaining attributes. “split_data()” sets the training and validation data and expected values, as well as the batch size for stochastic gradient descent, which again, at the time of writing is currently not implemented. “set_layer_dims()” sets integer values for the dimension of the input layer, output layer, and hidden layer in the case of the MLP. “init_weights()” sets the weights matrices and bias vectors as a random value between $\pm \frac{2}{3}$. And lastly, “init_error(epochs)” initializes all of the training and validation errors as 0 and sets empty lists to be used for keeping track of all these values over the number of epochs. Moreover, if the keyword argument for “epochs” is > 0 , then it sets the predetermined number of epochs. If no keyword argument is given, then this will initialize the over-fitting prevention instead, although this feature has not been implemented yet.

After the network is initialized, it needs to be trained. The second major method is the “train” method, which is fairly clear cut in its functionality.

Listing 2: Main training method that forward and backward propagates.

```

137 def train(self):
138     """ Adjusts the weights iteratively through gradient descent.
139
140     Args:
141         self [Perceptron]:
142         """
143     epoch_count = 1
144     while self.epochs:
145         if len(self.valid_data) > 0:

```

```

146         self.calc_valid_error()
147
148         for i in range(self.batch_size):
149             self.forward_prop(i)
150             self.calc_train_error(i)
151             self.back_prop()
152
153         if self.DEBUG:
154             self.print_progress(epoch_count)
155
156         self.update()
157         epoch_count += 1
158
159     if self.DEBUG:
160         self.print_results()

```

All sub-methods names describe exactly what those sub-methods do, so there is no need for further explanation. Moreover, any sub-method referring to error calculation or propagation follow directly from the mathematics outlined in section 3.

One exception to this is the “average_valid_error” and “average_train_error” values. These average values are rather important attributes because they will eventually be the metric of fitness for the genetic algorithm later. The average error value is calculated by taking the sum of sum of the error from each value in the error vector divided by the dimension of the output vector, divided by the number of training or validation data points considered.

$$\frac{1}{Tm} \sum_{i=1}^T \sum_{j=1}^m \epsilon_{ij}$$

where j is the index of the error vector, m is the dimension of the output layer, i is the index of the training example, and T is the total number of training examples. This formula for average error gives an estimate for how accurate the network is averaged across all output nodes and averaged across all training examples. Also, this value will always be between 0 and 1 like all other output values.

Finishing up the discussion of sub-methods, the “print_progress(epoch_count)” is a debug method that prints the error on training data, error on validation data, and the average error on validation data to the terminal after each epoch, and “print_results” is a debug method that prints the network’s final weights matrices, bias vectors, and average error on the validation data. Finally, the “update()” method adjusts the weights and biases, appends the error histories to their

corresponding arrays, and resets the partial derivatives and error values to zero in preparation for the next epoch.

The third and final method that called by the genetic algorithm is one for visuals and human interpretation. The “plot_error()” method plots the training and validation error across all epochs from the training process. There is also a “plot_avg_error()” method that does the same thing, but with the average errors.

Listing 3: Method that plots the error over the total number of epochs.

```
236 def plot_error(self):
237     """ Plots the training and validation error with
238         respect to the number of epochs."""
239     plt.style.use('seaborn')
240     plt.plot(self.train_error_history, color="red")
241     plt.plot(self.valid_error_history, color="blue")
242     plt.legend(["Training_Error", "Validation_Error"])
243     plt.xlabel("No._of_Epochs")
244     plt.ylabel("Error")
```

This method is fairly simple and straightforward, but an important method called by the SGA, therefore included in this explanation.

4.2 Chromosome Class

Now that the details of the neural networks have been covered, the next step is to look at the details of the genetic algorithm classes. Of the two main components of the GA, the chromosome class is the first, and lower level class. 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()” It will be immediately apparent that the doc-strings for this class are incomplete. This will also be the case for the generation and controller classes as well. One of the main areas of refinement for this project is with completing the documentation, but this is addressed more thoroughly in section 6.2

Listing 4: The constructor for the chomosome class.

```
32 def __init__(self, data, expected, dna=''):
33     """ YOUR TEXT HERE """
34     self.data = data
35     self.expected = expected
```

```

36
37     if dna == '':
38         self.dna = self.random_dna()
39     else:
40         self.dna = dna
41
42     self.genes = self.get_genes()

```

The input parameters are the “data” and “expected” arrays, again. This data is needed to instance the neural networks, so it has to follow all the way down from the controller to the generation, and from the generation to the chromosome, to eventually be used for 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: network architecture, learning rate, and number of hidden layer nodes. If an organism is instanced with the default kwarg 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 in the middle of listing 4. Lastly, the “get_genes()” method simply separates the dna into their various components and decodes them as three attributes representing the three hyperparameters: “network_type,” “learn_rate,” and “hidden_nodes.”

Once the organism has been instanced, it is ready for the decoded bit-string to instance a neural network.

Listing 5: Method that instances a network from the decoded DNA.

```

71     def init_network(self):
72         """ YOUR TEXT HERE """
73         self.preprocess()
74         if self.network_type:
75             self.network = mlp.MultiLayerPerceptron(\
76                 self.data, self.expected, self.learn_rate,\
77                 self.hidden_nodes, epochs=GLOBAL_EPOCH, seed=True)
78         else:
79             self.network = perceptron.Perceptron(\
80                 self.data, self.expected, self.learn_rate,\
81                 epochs=GLOBAL_EPOCH, seed=True)

```

This method is very simple, because it instances a neural network from one of the two network classes, passing in “data” and “expected” while using the decoded bit-string for “rate” and other keyword arguments.” Now, the “network” attribute of the chromosome corresponds to either a

“perceptron” object, or a “multilayerperceptron” object.

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

Listing 6: Method that trains the organism’s network and extracts the average fitness from the last epoch.

```
88     def get_fitness(self):  
89         """ YOUR TEXT HERE """  
90         self.network.train()  
91         self.fitness = 1 - self.network.avg_valid_error_history[-1]
```

This method calls the aforementioned “train” method from the network classes. So, this is when the GA trains the networks it has instanced. Then, the average 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.

4.3 Generation Class

The generation class is the object that packages everything 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 7: The constructor for the generation class.

```
42     def __init__(self, data, expected, size):  
43         """ Instantiates generation 0 of the genetic algorithm. """  
44         self.ID = 0  
45         self.data = data  
46         self.expected = expected  
47         self.size = size  
48  
49         self.init_fitness_history()  
50         self.init_generation()
```

This method sets the generation’s ID number, passes in the “data” and “expected” from the controller, and then initializes the various fitness histories. The fitness histories are various 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 instantiating P chromosomes, all with random DNA, in a list, where P is the specified population size.

After generation 0 has been created, the controller will evolve the network. This is done over a specified number of generations, and calls two additional methods in the generation class: “calc_fitness()” and “next_generation().”

Listing 8: Method that has each organism train to obtain its fitness.

```
67     def calc_fitness(self):
68         """ YOUR TEXT HERE """
69         self.gen_init_text()
70
71         for i in range(self.size):
72             chromosome = self.generation[i]
73             chromosome.init_network()
74             chromosome.get_fitness()
75             self.print_train_info(chromosome, i)
76
77         self.organize()
```

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 average error value 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 [1].

When the “calc_fitness()” method has completed, the “next_generation” method is called next. This method calls three sub-methods.

Listing 9: Method that creates the next generation of organisms and replaces the old generation.

```
153     def next_generation(self):
154         """ YOUR TEXT HERE """
155         self.new_generation = []
156         self.elitism()
157         self.repopulate()
158         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 section 5, but below is the source code for some of the graphs that will appear.

Listing 10: Several methods used for plotting various results.

```

215 def plot_total_fitness(self):
216     """ YOUR TEXT HERE """
217     plt.style.use('seaborn')
218     plt.plot(self.total_history, color="black")
219     plt.xlabel("Number_of_Generations")
220     plt.ylabel("Fitness")
221     plt.show()
222
223 def plot_fitness(self):
224     """ YOUR TEXT HERE """
225     plt.style.use('seaborn')
226     plt.plot(self.best_history, color="red")
227     plt.plot(self.avg_history, color="green")
228     plt.plot(self.worst_history, color="blue")
229     plt.legend(["Best_of_Generation", "Average_Fitness", \
230               "Worst_of_Generation"])
231     plt.xlabel("Number_of_Generations")
232     plt.ylabel("Fitness")
233     plt.show()
234
235 def plot_scaled_fitness(self):
236     """ YOUR TEXT HERE """
237     plt.style.use('seaborn')
238     plt.plot(self.best_scaled_history, color="red")
239     plt.plot(self.worst_scaled_history, color="blue")
240     plt.legend(["Best_of_Generation_(scaled)", \
241               "Worst_of_Generation_(scaled)"])
242     plt.xlabel("Number_of_Generations")

```

```
243 plt.ylabel("Fitness")
244 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. The third and final graph plotted from the generation class is the “plot_scaled_fitness()” method, which plots the best scaled fitness score of the generation, and the worst scales fitness score of the generation. The average is left out since fitness scaling will cause this to stay constant over every generation.

4.4 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 and learned from 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 11: Contructor for the controller class; runs the entire GA from neural network training to organism breeding.

```
23 def __init__(self):
24     """ Preprocesses the data and expected values for
25         the networks and runs the main evolution function."""
26     self.data = self.get_data()
27     self.exp = self.get_exp()
28     self.size = GENERATION_SIZE # Population size of generation.
29     self.gens = NUMBER_OF_GENS # Total number of generations.
30     self.main()
31
32 def main(self):
33     """ Instantiates generation 0, evolves the population,
34         and plots fitness values."""
35     self.pop = generation.Generation(self.data,\
36                                     self.exp, self.size)
37     self.evolve()
38     self.print_best_net()
```

The controller has other methods called “get_data()” and “get_exp()” which are the methods that produce Numpy arrays containing the pre-processed data for training the neural networks. For this project, the data used represents a simple exclusive OR (XOR) relationship, which is presented in table 1.

Table 1: Truth table for exclusive or (XOR)

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

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, 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 instantiated in its own program called “main.py,” where all the code from that program is listed below.

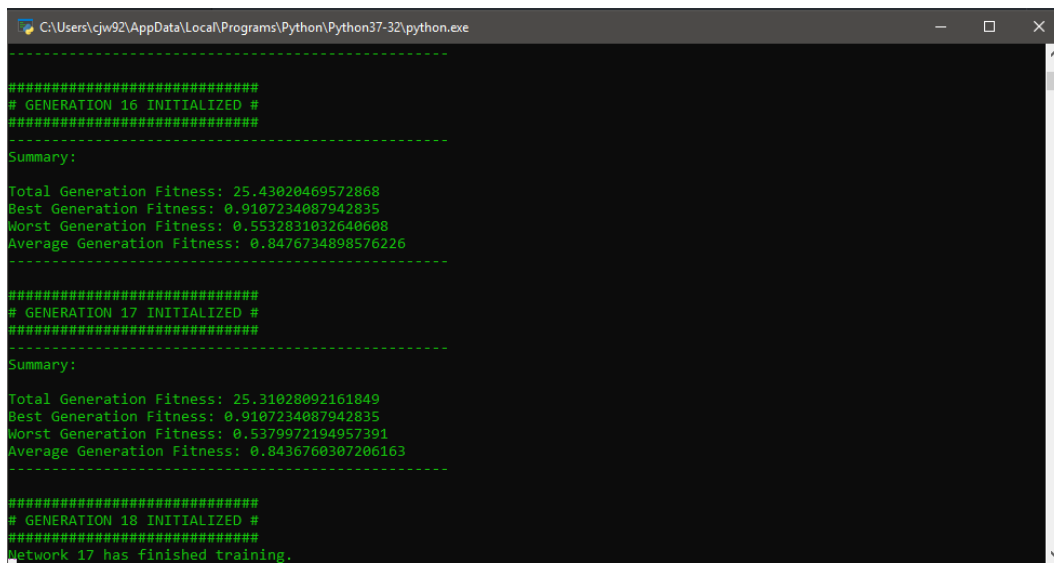
Listing 12: Main program that instances the controller.

```
1 import controller
2
3 def main():
4     controller.Controller()
5 main()
```

Again, this is the result of convention and clean code practices. Hence, that 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.

5 Description of Results

With the framework properly established, the entire program can finally be executed in full. The parameters chosen for the genetic algorithm are 30 organisms per generation, and 100 generations of evolution. This section will cover the results of the evolution, showcasing the plots from the previously discussed plotting methods of the various classes. Throughout the programs' evolution phase various bits of information get printed to the terminal after each generation. Figure 6 shows a screenshot from the Python terminal showing the terminal in the middle of the evolution phase. Figure 7 shows the Python terminal after the last generation has finished training its neural networks. At the end of the evolution, the best network configuration is printed in the terminal.



```

C:\Users\cjlw92\AppData\Local\Programs\Python\Python37-32\python.exe

#####
# GENERATION 16 INITIALIZED #
#####
Summary:
Total Generation Fitness: 25.43020469572868
Best Generation Fitness: 0.9107234087942835
Worst Generation Fitness: 0.5532831032640608
Average Generation Fitness: 0.8476734898576226
-----
#####
# GENERATION 17 INITIALIZED #
#####
Summary:
Total Generation Fitness: 25.31028092161849
Best Generation Fitness: 0.9107234087942835
Worst Generation Fitness: 0.5379972194957391
Average Generation Fitness: 0.8436760307206163
-----
#####
# GENERATION 18 INITIALIZED #
#####
Network 17 has finished training.
```

Figure 6: Python Terminal during the algorithm's evolution phase.

In this case, the DNA bit-string was “111111101,” which decodes to mean the architecture type is a multi-layer perceptron, the learning rate is 0.4, and there are 14 hidden layer nodes present. 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 a multi-layer perceptron, but this is expected because the training data was intentionally chosen to see if a multi-layer perceptron would emerge from the evolution process. Another interesting property from running the evolution multiple times is that the optimal network that evolves always tends to have a larger number of hidden layer nodes. In particular, the number of hidden nodes always evolves to be greater than 10, and the maximum


```
C:\Users\cjw92\AppData\Local\Programs\Python\Python37-32\python.exe
# GENERATION 100 INITIALIZED #
#####
Summary:
Total Generation Fitness: 26.02695868830729
Best Generation Fitness: 0.9107234087942835
Worst Generation Fitness: 0.5532831032640608
Average Generation Fitness: 0.8675652896102429
#####
Summary:
Total Generation Fitness: 26.02695868830729
Best Generation Fitness: 0.9107234087942835
Worst Generation Fitness: 0.5532831032640608
Average Generation Fitness: 0.8675652896102429
#####
# Best Network Configuration DNA: 111111101 #
#####
Network Type: Multi-Layer Perceptron
Learning Rate: 0.4
Number of Hidden Layer Nodes: 14
#####
```

Figure 7: Python Terminal after the program finished evolving.

number of hidden layers possible is 16. It was very surprising that the network found that many hidden layer nodes useful when there is only two input nodes and one output node.

So, without further adieu, here are the plots produced by the program after the evolution took place! In figure 8, we see that the overall fitness reaches its best values only about twenty gen-

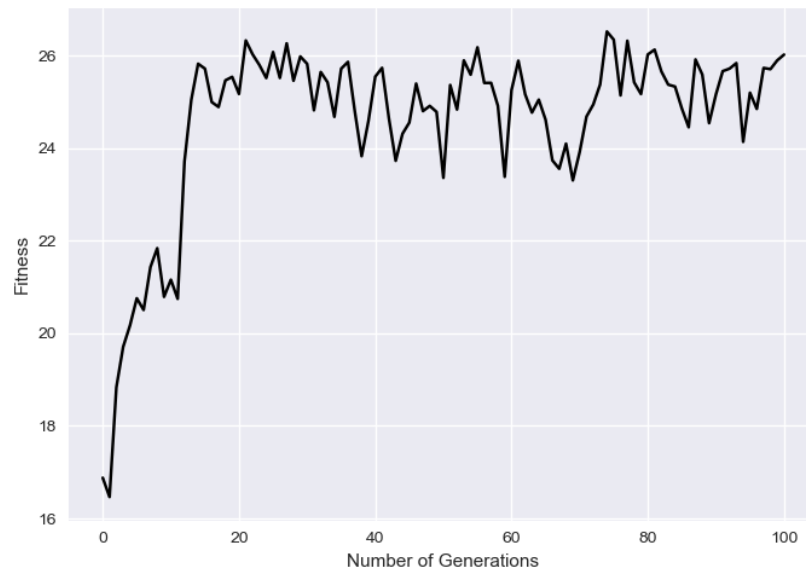


Figure 8: Plot of the sum of all fitness scores in a given generation

erations 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.

This Idea is further reinforced in figure 9. It appears that the optimal network configuration

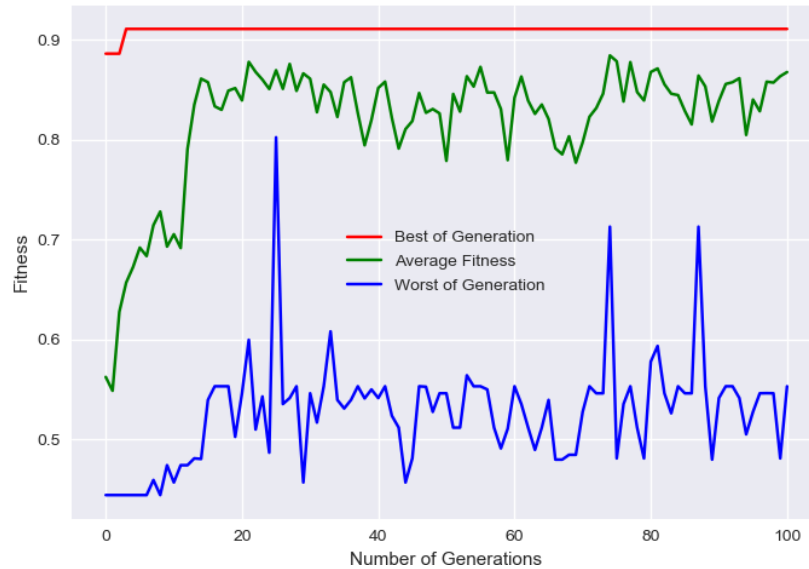


Figure 9: Plot of the best, average, and worst fitness in a given generation.

is found only about five or so generations into the evolution, since it is at this point that the best fitness score of the generation remains static for all following generations. There is also a lot of variance in the worst of the generation. This likely stems from the random variation that can occur after two organisms reproduce. Lastly, the average fitness reaches its peak state around fifteen to twenty generations into the evolution akin to the total fitness graph from figure 8. This makes sense because the average fitness plot is just a scaled down version of the total fitness plot.

Figure 10 offers insights into the overall fitness into all the organisms in the network. Once again, at around the fifteen to twenty generation mark, the scaled fitness value of the worst organism in the generation drops down to essentially zero, indicating the overall fitness of the network is high. Recall, the average scaled fitness for every organism in a given generation is always 1, so for the worst organism of a generation to have a scaled fitness implies there are more organisms that have relatively higher scaled fitness scores.

The final plot produced by the program is figure 11: the plot of the training and validation of the best network configuration from the final generation. Every network trained for 500 epochs.

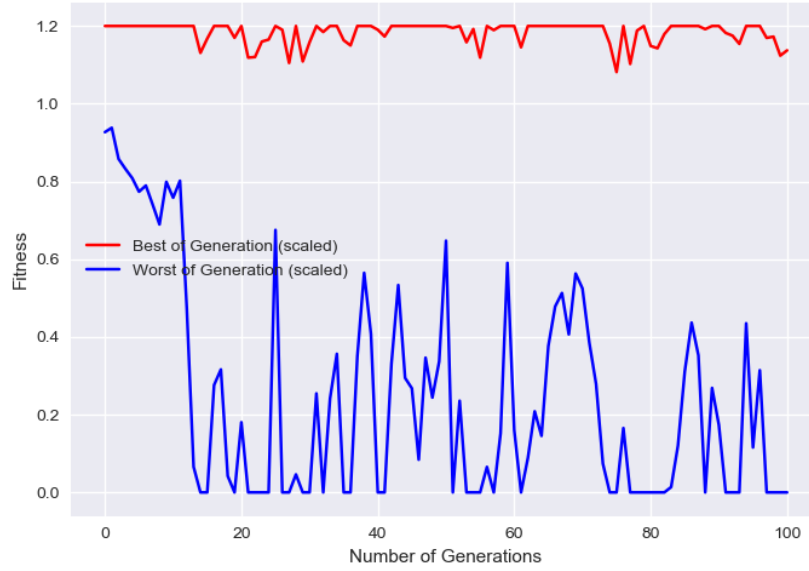


Figure 10: Plot of the best and worst scaled fitness scores in a given generation.

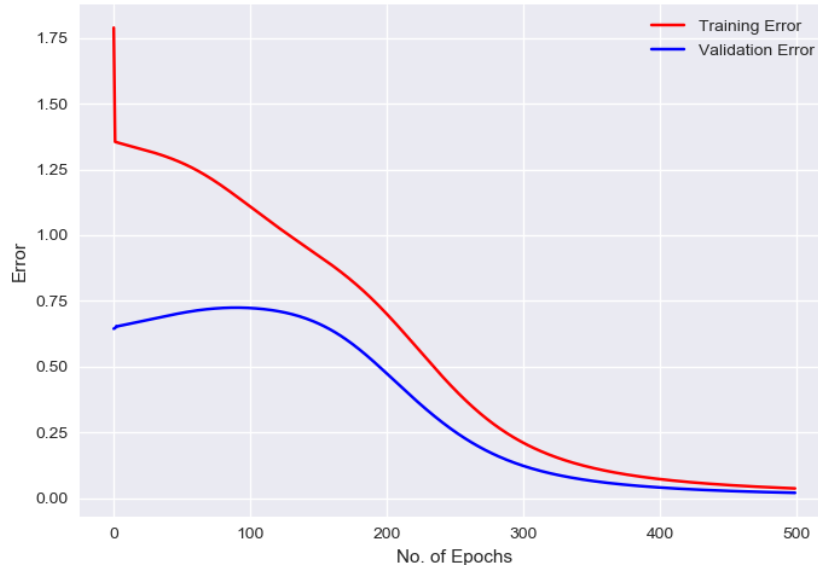


Figure 11: \hat{E} value for each epoch of the best network configuration.

The sheer 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 how the validation error is actually increasing for the first 100 epochs before joining the training error in its gradual descent. The network still reaches its final trained state, but it is strange how the validation error rises slightly before properly decreasing.

This concludes the presentation of the results. Conveniently, the program for this project was written such that the results present themselves, and only a moderate amount of commentary is needed to fill in the gaps. The next and final section will cover the limitations of the project, areas for refinement of the project, and areas for future work on the project.

6 Conclusions

In this section, some of the shortcomings of this project will be examined. It is important to discuss the potential flaws of any project so there is better understanding of which issues to address in the future, and how to make the best improvements moving forward. Following this discussion, some potential refinements of the current project will be covered. There are many incomplete parts to the program that need to be implemented before the project can be considered “completed,” and the next project can be started. These are mainly relatively small components that don’t have a large effect on overall project outcomes. The last topic of this section is a discussion centered on the future work extending beyond this project. These are large scale additions that would have a non-trivial effect on this project’s outcomes.

6.1 Limitations

The largest shortcoming of this project is the lack of “real world” data in its implementation. The data used was to simulate an XOR gate, so the resulting network topology is very simple. Moreover, the data contains no noise, and is very easy to process. At one point, there was a plan to use some education data to have the network attempt to predict grade outcomes for each semester, as well as overall grades. This idea was scrapped, due to other limitations in the overall implementation in the program.

The other main limitations stem from the incomplete features of the neural network architectures. For example, there is not yet a protection in place against over-fitting the weights to the training data; the user can only specify a number of epochs beforehand when instantiating the network. Hence, there is no guaranteed way to prevent over-fitting. Additionally, the feature allowing for a specified batch size for stochastic gradient descent has not been implemented either. This option would allow for much larger input spaces, and be more computationally forgiving with larger

data sets. It would also allow for a more practical data set such as the MNIST handwritten digits data set. This is a widely used data set for testing the performance of various network architectures.

6.2 Refinement

Some of the most immediate and obvious refinements that must be made follow from the limitations of the project, and are even listed in the document strings of various classes in the source code. For instance, the aforementioned small features should be implemented into the neural network classes. The ability to control for over-fitting is a huge gap in the functionality of the neural network at the moment, and hence is a priority for being implemented quickly. Furthermore, the option for using stochastic gradient descent on a random subset of training data each epoch is an important and useful addition to make the neural network more applicable to any data.

Another crucial addition is the completion of the document strings. Doc-strings are essential to having clean, bug-free code. This project's doc-strings are about 65% complete, but they will soon be completed to make this project more presentable in the near future.

Third, it should be noted that the hyperparameter space is very small. In fact, it is so small that it does not justify the construction of a genetic algorithm to search for its hyperparameters. There are only three variables, so a three dimensional hyperparameter space—one of which is binary, and two of which have sixteen possibilities each. Hence, a theoretical brute force search of the hyperparameters would only involve instancing and training 512 networks as opposed to the 3030 networks from the GA instancing 30 organisms across 100 generations. *However*, the principle behind this project is as a proof of concept, a demonstration of ability, and prepares for larger work in the long term. Future additions to this project will certainly increase the complexity of the hyperparameter space, making the work done not in vein.

6.3 Future Work

To end, here are some proposals for future, larger scale expansions to this project. Below are a few ideas of how the network framework that has been built can be expanded over the next five months. These additions concern, in particular, modifying the current neural network classes, or creating new, even more generalized network classes.

First, adding more activation functions. Right now, the only activation function that is used is the sigmoid activation function. Its derivative is embedded directly into the code for back propagation, which is convenient and easy, but severely limiting. In the future, a library of activation functions and their derivatives that can be called separately by each layer of a network offers a lot of potential. A given instance of a network could specify a certain activation function for a certain layer, and so on. Moreover, this could also give many more hyperparameter options for the genetic algorithm to search, making it a more interesting optimization problem.

Second, adding more network architectures. The matrix-vector approach to the mathematics of neural networks allows for a tremendously powerful representation of the concepts. By using recursion, it would be very feasible to write another network object that can instances a network that is arbitrarily deep. In conjunction with a large library of activation functions, each layer of this arbitrarily deep network could have its own activation function allowing for a very wide range of possibilities.

Third, creating a network architecture that uses a genetic algorithm to minimize the error function instead of gradient descent. This architecture, or set of architectures could then be compared to the current architectures that have already been written. It would be interesting to see which network architecture trains faster, is more accurate, etc. Moreover, a GA powered network architecture can also allow for implementing more complicated neural network topologies, while also bypassing having to learn about the increasingly more convoluted methods for back propagation (at least until a later time).

A Appendix: Source Code

Unfortunately, it is not practical to include the source codes for this project in the report, since it would roughly double the length of the entire paper. Instead, all of the source codes for this project can be accessed on GitHub with the link:

<https://github.com/cwolosh1/ssie-519-final-project>

References

- [1] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
- [2] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, *Algorithms*. McGraw-Hill Higher Education, 2008.
- [3] M. Feurer, J. T. Springenberg, and F. Hutter, “Initializing bayesian hyperparameter optimization via meta-learning,” in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
- [4] 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.
- [5] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.
- [6] J. H. Holland, “Genetic algorithms,” *Scientific American*, vol. 267, no. 1, pp. 66–73, 1992. [Online]. Available: <http://www.jstor.org/stable/24939139>
- [7] G. Sanderson, “But what is a neural network? — deep learning, chapter 1,” *YouTube*, Oct 2017.
- [8] 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>