jGeneticNeuralNet: Multithreaded Training of Neural Networks by a Genetic Algorithm in Java

Eric Wimberley

July 7, 2018

Abstract

Genetic neural networks, first characterized in 1989, have generally taken a back seat to training algorithms with a better time complexity. However, the more complex neural networks being used for image, video, and audio processing today use varying structures. Gradient descent algorithms are good at quantitative error reduction, but qualitative properties such as network structure must be determined with alternative algorithms.

Genetic algorithms are capable of evolving both quantitative properties of networks such as edge weights and bias, as well as quantitative properies such as which activation function to use and network structure. jGeneticNeuralNet is a Java implementation of a neural network and associated training algorithms for classification and regression that uses such a genetic algorithm to train networks.

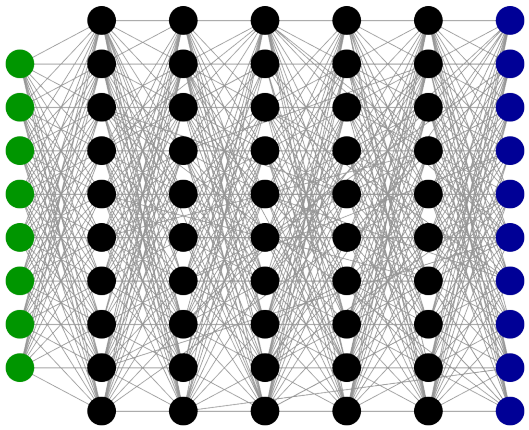

Figure 1: A visualization of a classification network with 8 input variables (green) and 10 output classes (blue). There are 5 hidden layers of 10 nodes each (black);

1 Introduction

Genetic neural networks (GNNs) are learning algorithms that use neural networks to associate an input with an output. Miller et al were some of the first researchers to train neural networks with genetic algorithms [1]. Rather than backpropagation, which determines the weights and biases within the network using partial derivatives, GNNs mutate the weights and biases randomly in a population of networks over many generations.

Genetic algorithms (GAs) are algorithms that use random mutations of solutions to optimimize some function of a system. They mimic genetic evolution of organisms in nature. While a random search of the problem space may seem less than optimal, progress is saved as a set of the best solutions so far, and the process can be easily parallelized [2].

More recent implementations allow for the structure of the network to mutate [3]. This allows networks to form more optimal structures for a particular problem space, as well as to optimize networks for size and complexity. For example, a GNN could automatically form a Convolutional Neural Network

(CNN), which is optimal for problems such as handwritten character recognition [4], and classification on other complex data.

2 Network Model

For extensibility and ease of implementation, neurons are implemented as objects. InputNeurons, HiddenNeurons, and OutputNeurons extend the neuron class. Each neuron has the following fields:

- A unique ID
- A bias
- The activation function to use
- A map from IDs to input neurons
- A map from IDs to output neurons
- A map from IDs to weights (IDs correspond to output map above)

Input neurons also include a field to contain the input feature. The activation function is overridden to return this input feature without any calculation. A neuron calculates its output by summing the outputs of all input neurons multiplied by their respective weights, adding the bias, and passing the result to an activation function [5].

$$\varphi((\sum_{n=1}^{N} n \times w_n) + bias)$$

A number of different activation functions are implemented. Functions with a range between 0.0 and 1.0 are particularly useful for probability regressions (the predicted probability of a class). Output neurons for the classification problem use these activation functions.

$$\varphi_{sin}(x) = (sin(x * \pi - \pi/2) + 1)/2$$

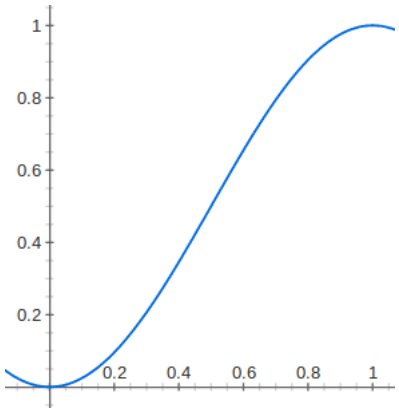

Figure 2: A plot of the $\varphi_{sin}(x)$ activation function.

The arctan activation function can also be used as a bounded output function for classification or probability regressions.

$$\varphi_{arctan}(x) = arctan(x)/\pi + 0.5$$

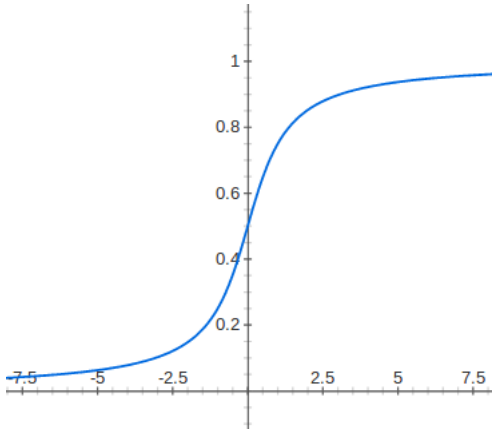

Figure 3: A plot of the $\varphi_{arctan}(x)$ activation function.

A simple step function is implemented as follows.

$$\varphi_{step}(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

For regressions, some functions with a larger range are implemented, such as the linear and squred activations below.

$$\varphi_{linear}(x) = x$$

$$\varphi_{squared}(x) = x^2$$

3 Genetic Algorithm

Networks are trained with a genetic algorith, which first produces a large number of random networks. Each network is tested for fitness by determining its average error rate on the training data, and the population of networks is mutated to generate networks with improved fitness. By selecting only the networks with the lowest error at each step (the most fit networks), the population of networks slowly moves towards a low error rate.

3.1 Mutation

Mutation can affect edge weight, node bias, node activation function, and network structure. Numeric values are changed by a random fraction of the learning rate, which is set as a hyperparameter.

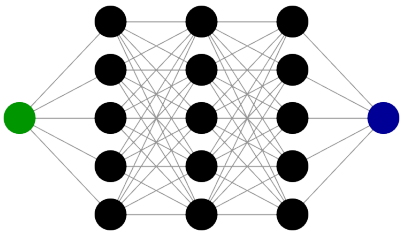

Figure 4: A simple regression network that is fully connected.

Figure 4 above shows a fully connected network that could be used for a regression model. After a few generations of random mutation and fitness selection, the network is no longer fully connected (Figure 5).

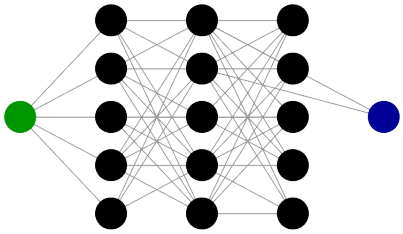

Figure 5: A mutated network with a different connection structure.

The visualization in Figure 6 below shows a network that was trained on the Iris dataset [6]. It shows the variety of edge weights, biases and activation functions that have been selected by the genetic algorithm.

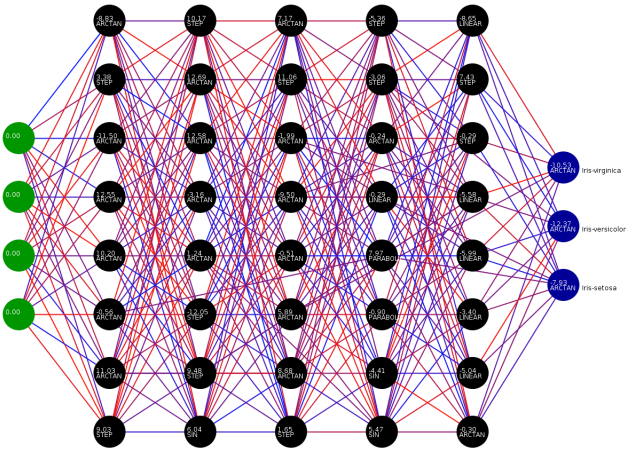

Figure 6: A visualization with lowest edge weights in blue and highest edge weights in red. Biases

3.2 Fitness

While it is common for genetic algorithms to maximize a fitness variable, the GNN training algorithm minimizes error instead. This means that the "fitness" function is really the inverse of what is normally considered fitness.

Algorithm 1 Fitness Algorithm

```
1: function FITNESS(training)
2:     avgError = 0.0
3:     for datum in training do
4:         e = abs(datum.out − predict(datum.in))
5:         avgError = avgError + e
6:     end for
7:     return averageError/len(training)
8: end function
```

Algorithm 2 Training Algorithm

```
1: function TRAIN(c)
2:     pool = fitness computing thread pool
3:     pop = network queue ordered by fitness
4:     offspring = job queue ordered by fitness
5:     survivors = network queue ordered by fitness
6:     for generation < c.maxGens do
7:         for nework in population do
8:             pool.execute(newJob(network))
9:         end for
10:        pool.shutdown()
11:        while pool.isRunning() do
12:            sleep(10)
13:        end while
14:        i = 0
15:        while !pool.isEmpty() do
16:            offspring.add(pool.getJob(i))
17:            i = i + 1
18:        end while
19:        while len(survivors) < c.population do
20:            survivors.add(offpring.poll().network)
21:        end while
22:        population = survivors
23:    end for
24: end function
```

4 Optimizations

Training genetic neural networks is computationally expensive. Thousands of networks must be evaluated during selection of the final model. Therefore, the network training algorithm employs several important optimizations:

- Each network is trained in a seperate thread
- Memoization of neuron output
- Memoization of average network error
- Most fit networks produce more offspring
- Random sub-sampling of the training data during training error estimation

4.1 Multithreaded Training

Networks are judged for fitness by computing the average error on training samples. This can be computed in an embarassingly parellel fashion by running one thread per network. As shown in the pseudocode below, a job to compute the fitness of a network is executed in a thread pool, which can be configured with a maximum number of threads depending on hardware.

Inside each thread the original network is cloned, mutated, and then evaluated for average error. This minimizes the amount of work that must be performed in the main thread.

4.2 Activation Function Memoization

Highly connected networks use the output from the same neuron for multiple neurons in the next layer. Rather than recalculating the output of these neurons, the activation function output is memoized. Each neuron only needs to store two variables in order to accomplish this: the previous activation output and a "memoized" flag. Note that this optimization is important for both training and production use of the network.

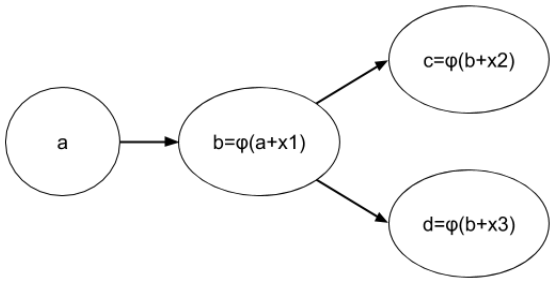

Figure 7: Two neurons use the output of neuron b. Instead of recomputing b each time, the output of b can be memoized. Note that x_i refers to the bias for that neuron.

The structure from Figure 7 occurs a large number of times in layered neural networks. While this optimization cannot compete with matrix multiplication with specialized hardware, it does mitigate some of the performance issues associated with object oriented neural networks. The instructions to compute the output of this network with and without memoization are shown below.

Without memoization:
Step 1: Compute $\varphi(a + x1)$
Step 2: Compute $\varphi(b + x2)$
Step 3: Compute $\varphi(a + x1)$
Step 4: Compute $\varphi(b + x3)$

With memoization:
Step 1: Compute $\varphi(a + x1)$
Step 2: Compute $\varphi(b + x2)$
Step 3: Compute $\varphi(b + x3)$

For a fully connected neural network with N layers and M neurons per layer, memoization reduces predicion complexity from $O(M \times M \times N)$ to $O(M \times N)$. In other words, instead of computing the neuronal output for each conection for each layer, neuronal output is computed once per neuron. Not all networks are fully connected, but this is the worst case scenario for a non-memoized implementation.

4.3 Fitness-Based Offspring Rate

More fit networks produce more offspring than less fit networks. This is based on the idea that a mutant of a more fit network is more likely to produce a better network than a less fit network. Not only should this decrease the number of generations required to reach a sufficiently trained model, but it should reduce the number of total networks tested for fitness as well. This results in a shorter training time.

5 Methods

5.1 Classification

The Iris dataset [6] was used to train classifiers. Future implementaions may use 10-fold cross validation, however the current algorithm simply leaves out 1/5th of the input data for testing. Models were judged based on their 3-class accuracy. Two-hundred networks with 3 hidden layers of 8 neurons each were trained for 15, 30, 60, 120, 240 and 480 generations respectively. The confusion matrix and accuracy for one such model is shown below.

Expected	versicolor	virginica	setosa
versicolor	8	1	0
virginica	0	10	0
setosa	0	0	12

Accuracy: 0.967741935483871

5.2 Regression

The x^2 function was used to generate training data for a regression network. Each network had 2 hidden layers and 6 neurons per layer. Like the classification benchmark, 1/5th of the input data was set aside for testing, and two-hundred networks were trained for 15, 30, 60, 120, 240 and 480 generations. Some expected outputs and predicted outputs are compared in the table below, along with the accompanying mean squared error.

Expected	Predicted
0.0	1.20346
1.0	2.06567
4.0	5.16131
9.0	10.0320
16.0	16.9270

Mean squared error: 0.3971038018343635

6 Results

Classification accuracy for the 3 class Iris problem is shown in Figure 8 below. Accuracy increases as exepected with generation number until generation 240. However there was no noticable difference between 240 and 480 generations in terms of average or variance. Some training attempts appear to get stuck in local minima. The average accuracy seems to stabalize at around 0.9.

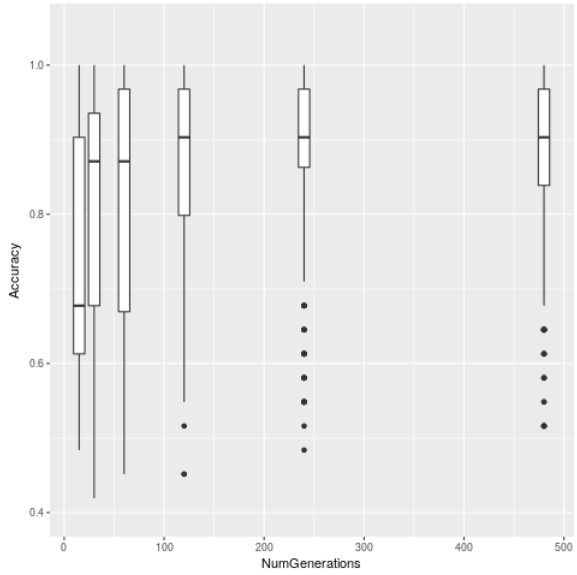

Figure 8: Classification accuracy after 15, 30, 60, 120, 240, and 480 generations.

Mean squared error for a regression of the x^2 function (Figure 9) resulted in many more outliers than classification. However, the distribution of model error improves with increased generations.

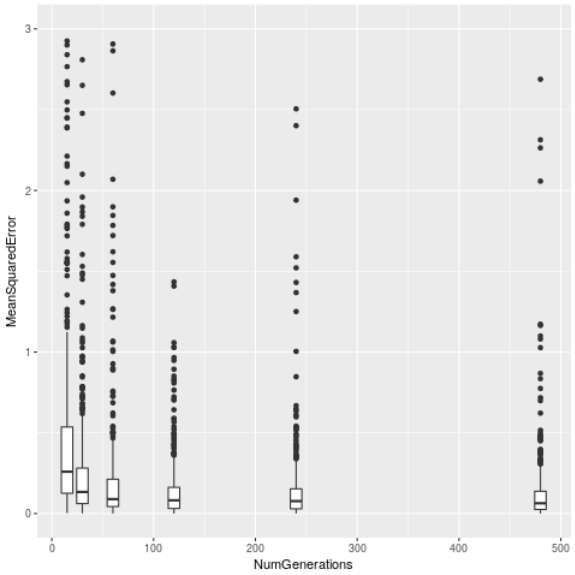

Figure 9: Regression mean squared error after 15, 30, 60, 120, 240, and 480 generations.

7 Conclusion

While a genetic algorithm is unlikely to beat backpropagation in terms of time complexity, it has the advantage of optimizing network structure and other qualitative network properties. The algorith was able to train high quality models on known good data sets. Future improvements could include cross-fold validation, computation of AUC ROC, and area under the precision/recall curve to give a more in depth evaluation of models.

References

[1] G. Miller, P. Todd, and S. Hedge, "Designing neural networks using genetic algorithms," 1989.

[2] R. Tanese, "Distributed genetic algorithms for function optimization," AAI9001722, PhD thesis, Ann Arbor, MI, USA, 1989.

[3] H. Lam, S. Ling, F. Leung, and P. Tam, "Tuning of the structure and parameters of neural networks using an improved genetic algorithm," 2003.

[4] D. Cireşan, U. Meier, L. Gambardella, and J. Schmidhuber, "Convolutional neural network committees for handwritten character classification," 2011.

[5] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003, ISBN: 0137903952.

[6] R. Fisher, "The use of multiple measurements in taxonomic problems," 1936.