A Combined Metaheuristic Algorithmic Approach for Dynamic Optimal Training of Feedforward Neural Networks

Abstract:

In this paper, we proposed an optimal combined algorithmic approach involving both heuristic and nonlinear-predicting algorithms to address the issues in overfitting and underfitting while training supervised artificial neural networks. To obtain this algorithm, we researched and implemented multiple pre-existing algorithms to our neural network, before combining them into a single algorithm. This algorithm avoids these issues through training a single neural network object from "general" to "specific"; In other words, this neural network object was initially trained using algorithms highly dependent on random variability, and slowly began to delve into an algorithm less dependent on the ideas of randomness. This algorithmic approach proved to be one of the more optimal techniques compared to the earlier researched algorithms. The proposed algorithm demonstrates its capabilities by detecting multiple complex functions dynamically using a fixed neural network architecture and implementation. This paper discusses this algorithm's superior learning rates and generalization capabilities compared to any of the constituent well-known algorithms that it combines.

Introduction:

This paper considers studying artificial neural networks (hereby referred to as ANNs) and their training algorithms using various optimization formulas. ANNs are powerful machine learning models often used in pattern recognition, which can be used in a multitude of applications, especially as universal approximators. These neural networks are based off of how the human brain functions, taking the intricately interconnected neurons and attempting to create a model which learns and recognizes patterns the way that our brains do. ANNs have a throng of specific optimization algorithms at hand, but these optimization algorithms are not consistently accurate in all situations. The lack of simple, efficient and thorough ANN optimization algorithms hamper progress in the field of artificial intelligence, as ANNs cannot be utilized to their fullest potential.

ANNs consist of serious issues, such as the stability versus plasticity dilemma. The stability versus plasticity dilemma refers to the enormous problem that all computer data scientists face– finding a balance between overfitting and underfitting. Underfitting in ANNs is when an ANN has not undergone enough training to predict and understand the patterns. In a fervent attempt to avoid underfitting, programmers accidentally veer towards overfitting, or overtraining the ANN so that it can "correctly" recognize patterns. However, overtraining can actually lead to a more disastrous effect. Overtraining can cause the ANN to be so acutely attuned to the initial pattern that it won't be able to generalize to unseen data for a given pattern, deeming it to be almost useless for predictive purposes. In an alternate situation, overtraining can also train better for new patterns, but cannot train for the old patterns at all. It is also very hard to avoid overfitting issues for ANNs [1]. Either way, without a balance between underfitting and overfitting, the ANNs cannot reach their full potential in pattern recognition.

In response to this gap in ANN training capabilities, we studied various algorithms and created a singular combined metaheuristic algorithm for optimizing ANNs to solve any generic problem requiring pattern recognition; we approached this through function approximation by avoiding overfitting and underfitting problems. There have been several research papers on how overfitting and underfitting present problems for ANNs [2, 3]. Our approach attempts to solve this problem in a simple fashion by using heuristic based algorithms along with traditional nonlinear optimization algorithms for training an ANN.

This process took six stages. First, we developed a feedforward neural network, which is an ANN where the information flows merely in one direction, and implemented four different algorithms (backpropagation, genetic, simulated annealing, and nelder mead), to get a proper grasp as to how each of the optimization algorithms work in different situations. Next, we created a multi-algorithmic approach, which applies three of the earlier algorithms (genetic, simulated annealing, and nelder mead) to three different neural network models. Then, using a comparison model, the program uses a scoring method to determine the best algorithm for the situation. The program implements this algorithm to output the results based on the selected neural network's training. After unsatisfactory results from the previous stages, we used a multi-algorithmic approach, which trained an ANN object with the three algorithms used in the previous step in a specific sequence, which resulted in fairly consistent accuracies.

Based on our preliminary results, if this algorithmic approach undergoes more maturity, it could become extremely useful in the field of artificial intelligence, as the ANNs could then potentially be utilized for a large number of scenarios requiring pattern recognition. For example, this metaheuristic algorithm could be implemented to accurately predict the behavior of physical processes, with the assumption that these processes can all be represented mathematically. Hopefully, optimally and dynamically predicting ANNs can be implemented in replacement of harder-to-write algorithmic approaches.

Materials and Methods:

(Note: all programming was performed in Java in the Eclipse Indigo environment, using the TerminalIO and Encog libraries [5]. We also implemented our own modified version of the NormUtil class from the Encog wiki webpage [4]. All other Java classes or methods that do not belong to Encog have been written by the authors of this paper.)

Our hypothesis validation consisted of the following stages:

Preliminary Stage: High-Level Solution Design

For a broader idea of the picture, we began with a high-level design, which streamlined our approach to the problem. For simplicity, we chose to create an optimization function which can predict the behavior of a two-dimensional function approximation. The solution is comprised of two stages: the initialization stage, and the execution stage, as illustrated in Figure A. The solution module in the first stage of the high-level design accepts multiple user inputs and their corresponding outputs. This module utilizes these (x,y) cartesian coordinates to train the ANN for the remaining data for interpolation and extrapolation. In the second stage, the solution module uses its thorough training to predict the output for any given extraneous input.

In this scenario, we implemented a simple feedforward ANN, because the information only flows in one direction. All feedforward ANNs consist of an input layer, one or more hidden layers, and an output layer. Since, the feedforward ANN is a supervised neural network, our intention was to ensure that the supervised training stage only requires a minimal number of known inputs and outputs, to increase the ease of trainability for the proposed ANN in all plausible and realistic scenarios. After the function detector is initialized, the test program will invoke the function detector again to determine the output for the "unknown" inputs. Our goal was to keep the fundamental solution design the same, regardless of which training algorithm is implemented, as the different algorithms merely increase efficiency and accuracy.

Stage 1: Feedforward and Backpropagation Algorithm

For simplicity, we began with an evaluation of feedforward ANNs using a backpropagation algorithm. The backpropagation algorithm is the standard method for training ANNs, which tries to find network weights such that the network error (difference between ideal output and actual output) is minimized [7]. However, we found this approach works for only linear functions, which could lead to issues later on when detecting second degree or larger degree polynomials. Next, we progressed towards more complex nonlinear optimization algorithms, to increase adaptability.

Stage 2: Feedforward and Genetic Algorithm

In an attempt to increase adaptability and percentage of accuracy, we changed our neural network training algorithm to use a genetic algorithm (hereby referred to as GA) [6]. GAs are based on the theory of Darwinism, also known as natural selection. The ANN training using GA for our project is done as follows.

The initial step is to create a large population of neural networks, a network of "chromosomes", using random initial weights. Next, encode these "genomes" for each network in the form of an array of network weights. Then, define the "fitness" criteria for the GA through the overall network error (Root Mean Squared Error or RMSE) for any given network using all known training data. The GA makes it its goal to minimize the network error. Genetic operators, such as crossover, are completed by simply splitting the network weights with a preset probability (p_c); mutators, however, are implemented by randomly changing a few weights from the original gene with a probability of (p_m). The preset probabilities were initialized at the start of the GA process. At the end of the genetic evolution process, the best solution available within the population is selected.

We believed that a GA would be ideal for our problem, mainly due to the fact that it has the ability to predict nonlinear functions. GAs, however, do provide other benefits, seeing as their dependence on randomness can be useful in select situations, and we do not need to understand the behavior of the data beforehand, which increases its adaptability. However, due to somewhat unsatisfactory levels of consistency, we moved to nonlinear optimization algorithms less influenced by randomness.

Stage 3: Feedforward and Simulated Annealing Algorithm

The simulated annealing algorithm (hereby referred to as SA) was another algorithm which could potentially make our function detection/approximation program more accurate. The SA simulates the physical annealing process available in metallurgy which solidifies a metal into a uniform crystalline structure. SA also involves simulating a cooling process, temperature schedule, from a high temperature to low temperature gradually. SA starts with an initial solution (i.e network weights) and then in each iteration, it tries to choose a set of random weights as solutions. The random perturbation is high in the beginning due to high temperature and as the temperature cools down, random perturbation becomes much smaller [6]. In order for SA to validate the quality of a solution in each iteration, we use a cost function which is essentially the overall network error or score (RMSE) as defined in the section for GA. SA's objective is to find a set of network weights which minimizes the network error.

In each iteration, if the randomly chosen weights gives a smaller network error, then SA chooses this solution as the current solution. However, SA sometimes accepts a worse solution via Metropolis probability [6]. Typically, at the beginning of the SA process, we choose a solution of lower accuracy in comparison to the current solution.

SA is an ideal algorithm in this situation, due to its ability to predict for nonlinear functions, and in part due to its randomness. However, this algorithm is not nearly as dependent on randomness as GA. SA works in a more organized fashion, as it narrows into a smaller search space, whereas with GA, we are forced to depend only on the occasionally unreliable fitness function. Unfortunately, similar to the GA, the SA can also get stuck on local minima, which don't give us the highest percentage of accuracy, leading us to search for another, more reliable algorithm.

Stage 4: Feedforward and Nelder-Mead Algorithm

In order to combat this variability in accuracies for our results, we used the Nelder-Mead algorithm (hereby referred to as NM), which is essentially used to find the global minima of an objective function in n-dimensions, which doesn't have as much dependence on randomness as GA and SA. This is a heuristic search method, using simplexes which can converge to local minima [8].

With one to one mapping of variables, a simplex is noted to be a triangle. This algorithm compares all of the function values at the three vertices of the triangle. Whichever vertex is the largest (and thus most inaccurate) is replaced by a new vertex, which decreases by the user-defined step size, and so on and so forth. This process results in a sequence of simplexes, until the triangles converge to find the global minima, also known as the optimization point.

We had to hone in on a small step size, which determines the decrement from one vertex of the simplex to a new one. Although NM as a general trend had a high percentage of accuracy, a lot of the time, SA and GA could predict with more accuracy than NM due to their variability, so we then worked our way towards our initial multi-algorithmic approach.

Stage 5: Initial Multi-algorithmic Approach

We initially decided to combine our three best algorithmic approaches– SA, GA, and NM, creating a class which selected the optimal ANN algorithm for each situation. As we learned from our initial four attempts, adaptability is key for ANNs, due to the amount of random fluctuations. Therefore, we declared three ANN objects, and trained each of them using a different algorithm; the first was trained using SA, the second using GA, and finally, the third using NM.

Next, we calculated an average score (or network error) for each of the ANN objects after their training, selecting the ANN with the lowest score (or network error) for the given situation. We also checked to see which trained ANN had the greatest accuracy by re-entering some of the given inputs, and using that as a deciding factor for which ANN we should select for the situation. However, although more consistent than the previous stages, due to the occasional low percentage of accuracy, we believed that we could combine the algorithms in a more succinct way that could result in better accuracy.

Stage 6: Ultimate Multi-algorithmic Approach

Ultimately, we chose to train the neural network using multiple algorithms, from increasing to decreasing levels of randomness or in other words all heuristics based algorithms (such as GA and SA) in the beginning. We began by training the ANN with the GA as the first optimizer, as it works the best when both the good and bad genomes influence the ANN weights. Then, we cloned the GA network, and sent the copy to the next algorithm for training. Next, we trained the ANN with the SA algorithm, as the dependence on randomness was smaller than that of GA. Again, we made a copy of this network. Then, we trained this hybrid ANN with the NM algorithm, which really hones in on the global minima optimization point for the function. Finally, we used our initial optimal ANN selector and selected the network which has the lowest score or network error. Once we tested the ANN with the training data set, we found more consistent and higher percentages of accuracy than we had before the initial multi-algorithmic approach. In this approach not only did we combine multiple algorithms in an algorithmic pipeline, but also we conjoined multiple heuristic based algorithms (GA and SA) to create a combined metaheuristic algorithm.

Results and Analysis:

Prestage Results and Analysis:

In an attempt to create a solid structure for our ANN, we delved deeper into the architecture and optimal set-up for the most accurate results. We began by testing how the number of hidden layers from 0 to 6 affected our accuracy while using a feedforward ANN.

Therefore, we came to the conclusion that 1 hidden layer had the peak in percentage of accuracy, and chose to keep the structure of one input layer, one hidden layer, and one output layer for all attempts in modification throughout our research. This finding aligns well with various literature that we've read [7,8,9].

Another important structural viewpoint for this solution design lies in how we chose the minimum number of possible iterations for optimal results. We implemented a multitude of algorithms on the ANN, and tested how the number of iterations (from 0 to 250) influenced the regression score error for each of the algorithms. We found that the score is lowest when it is somewhere between 175 to 225 for each of the algorithms we implemented. Therefore, we chose to use 200 iterations, as it was safely within the range for the optimal number of iterations, yielding the lowest score (see Figure F).

We tested with various activation functions (Sigmoid, TANH, LOG) and we found that LOG was the most effective function for a generic ANN as it accepts values from -1 to 1 and saturate less quickly. Finally, we updated an existing normalization class, NormUtil, which shifted all training data inputs into a certain "normalization range", which served to fulfill several of the parameters for the Activation Functions. This function takes the difference from the value requiring normalization (aka– the input) and the data low and multiplies it by the difference between the normalized high and normalized low. Then, it divides this product by the difference between the data high and data low, and sums that quotient with the normalized low to get a normalized value between the given range.

Stage 1 Results and Analysis:

The backpropagation algorithm, the first algorithm we used, eventually started off with extremely high percentages of accuracy, that were, in general, in the 80s and 90s, when the function was very simplistic and very linear, such as $y = x + 5$. However, we noticed that, the more complex the function, the more inaccurate the predictor would be. In order to get the accuracy percentages even up to the 30s and 40s for nonlinear functions, many training iterations were required. In many cases it would take thousands of iterations *and* training data inputs to yield a percent accuracy above 30, slowing the program down considerably. An increase in the number of iterations caused other ANN structural problems, such as overfitting. Therefore, we reduced the iterations, and only increased the number of training inputs to 1000. However, these extra training inputs didn't really cause any significant difference in the percentage of accuracy. Seeing as 1000 inputs is not realistic in the real world, in terms of data collecting, we needed to find a method of optimization which could predict for nonlinear functions with high percentages of accuracy, and did not require more than 100 training data inputs to run properly.

Stage 2 Results and Analysis:

We immediately noticed the increase in accuracy from the backpropagation to the GA, in regards to predicting the outputs of nonlinear functions. However, the resulting percentage of accuracies were still comparatively lower than what we expected, and were far from consistent, due to the GA's extreme dependence on randomness, which is very similar to that of biological natural selection. The GA has especially low percentages of accuracy when it comes down to the size of search spaces, as the larger the space, the greater the probability is for the ANN to gain incorrect prediction power. The size of the search space is dependent on the complexity of the function, and can also lead to more random fluctuation in the regression error.

We attempted to decrease this random fluctuation by attempting to use the population size to our advantage. We created a model which tested a multitude of population sizes, from 10 to 1000 with increments of 50, and graphed these results (see Figure E). From this model, it is evident that the ideal population size lies within the range of 550 and 650. Once we changed our population size to 600, we got significantly more accurate predictions, and a higher prediction power than that of backpropagation. The percentage of accuracy had increased to typically 60-75% accuracy, whereas backpropagation had led to 30-50% accuracy, even with 1000 inputs.

We also changed the crossover and mutation probabilities. And, we found mutation probability should be very small compared with crossover probability for having a positive impact on accuracy. Eventually, we settled to use crossover probability as 0.9 and mutation probability as 0.1.

Also, when using a GA, we didn't require nearly as many training inputs– we had reduced the training input requirement to about 75 inputs, which is much more feasible than having to collect 1000 data inputs. The extreme randomness of the GA, however, still led to variable and inconsistent results, which led us to search for ANN algorithms which are more independent of the randomness factor.

Stage 3 Results and Analysis:

The SA algorithm also had a wide range of percentage accuracies, due to random fluctuations, as was the case for the GA. SA randomly assigns weights to the neurons in the ANN, before trying to solve the problem, which is the main cause of its dependency on randomness. We cannot have too large of a search space, as we may run into extremely inaccurate percentages of accuracy, due to the randomness of where the perturbation will occur.

However, at the same time, we cannot have too small of a search space, as it may not be enough to provide us with the most accurate prediction. Therefore, we graphed the trend of how our score fluctuates as the initially instantiated high temperature increases, in order to determine the optimal high temperature setting (see Figure D). From our data, we came to the conclusion that the optimal high temperature setting is somewhere between 15 to 20 degrees, as the graph has a score closest to 0 within that range, which managed to increase the consistency of our percent accuracies. The SA appeared to predict complex trigonometric functions the best, but failed once it began attempting to predict cubic, quartic, or other functions of larger degrees. Thus, due to its incapability to adapt to multifarious situations, we reasoned that SA was far from a satisfactory solution.

Stage 4 Results and Analysis:

The NM algorithm, which was implemented next, proved to yield results that were much more consistent and accurate than those of the GA and SA. The NM maintained a higher level of consistency, mostly due to the fact that it doesn't depend nearly as much on randomness as GA and SA. NM only implements some level of randomness as it determines the position of the initial simplex within the search space. The NM is also highly dependent on the step size variable, which determines how far the largest vertex of a simplex moves. This step size can allow for better accuracy, as we can hone in on the global minima more efficiently. Therefore, to be able to choose the optimal step size for this algorithm, we graphed the regression error versus the increasing step size. We found that 0.0055 yielded the most consistent and accurate results, as the curve of the graph dips down to almost 0 for a long period of time between 0.01 and 0.001 (see Figure C).

As can be evidenced in the figure, the NM algorithm, while yielding more consistent and predictable results, still was not as precise and accurate as we wished, only being able to vaguely approximate within the general area of the ideal output. NM typically had a high percentage of accuracy, ranging from about 80 to 90% with the occasional fluctuation, but we weren't satisfied with the results, hoping to increase the consistency and accuracy to about 90-100%.

Stage 5 Results and Analysis:

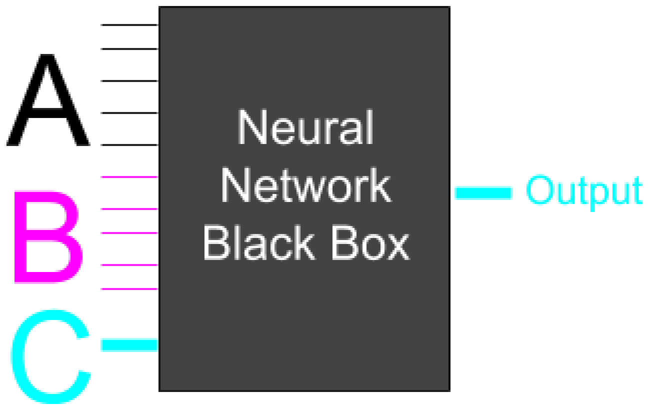
After realizing that certain ANN algorithms worked better in specific situations, closely based on the training set selected, and that this "optimal" ANN differed from time to time, we decided to create a class that selected the optimal ANN algorithm for each situation. We used this optimal neural network selector in an attempt to ensure adaptability, which the lack of, often times, caused the largest percentage of error in the ANNs (see Figure B). As we can see, more often than not, the NM algorithm is the optimal algorithm. However, when the GA and SA algorithms were deemed optimal by the class, the accuracies were much higher than those of NM. This algorithm still maintained high levels of fluctuation in the percentage of accuracy, seeing as the GA, SA, and NM algorithms weren't exactly changed at the core, or implemented differently. One of the reasons for having sometimes comparatively higher inaccuracies is because we found that network scores are not the sole determinator for choosing an optimal algorithm. An algorithm can have minimal score, but can still produce results of higher inaccuracy due to overfitting issues. Though there was less fluctuation when using the NM algorithm than those of the previous stages, and more adaptability, we still weren't satisfied with its meager ability to predict consistently from 80 to 90, with some occasional 95%. Therefore, we attempted to combine the algorithms in a more succinct way that could result in better accuracy, adaptability, consistency and prediction power.

Stage 6 Results and Analysis:

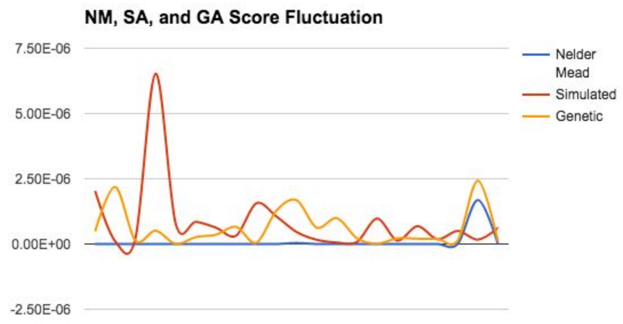
The final stage was the most accurate amongst the approaches that we undertook throughout this research paper. We took on a true metaheuristic approach, and trained only one ANN, from increasing to decreasing levels of randomness (see Figure B again). As illustrated by the diagram, it is evident that the GA had the largest amount of random fluctuation in its score/regression error, and SA was a close second. Seeing as the graph of the score for GA over multiple trials had several curves, we can determine that GA was best suited for the initial training of the ANN, as the goal was to move from general to more specific, because in order for a GA to have a high prediction power, it requires both good and bad genomes; otherwise the GA has nothing to compare the good genomes to, resulting in low accuracy. Without the bad genomes, we wouldn't have crossovers with potentially extremely successful mutations; GA simulated the biological model of natural selection, where it needed more than just the super race to keep improving.

Throughout each of our training stages, we made copies of the ANN object– one after GA, another after GA and SA, and finally one after GA, SA, and NM. Then, as previously mentioned, we implemented our optimal neural network selector class to select which ANN had the most optimal algorithm for the scenario. Through this, we were able to include more adaptability, and decrease error, in case we ran into overtraining in the multi-algorithmic approach. With this algorithm, we also had much more consistent percentages of accuracy, which were usually in the 90s, as we managed to minimize much of the randomness involved within these algorithms by training the ANN with NM at the final stage, which really allowed the algorithm to hone in on what the global minima is for the specified function. For the most part, we found that the most optimal ANN algorithm implemented throughout our research, was in the final stage of the new multi-algorithmic approach.

Illustrations:

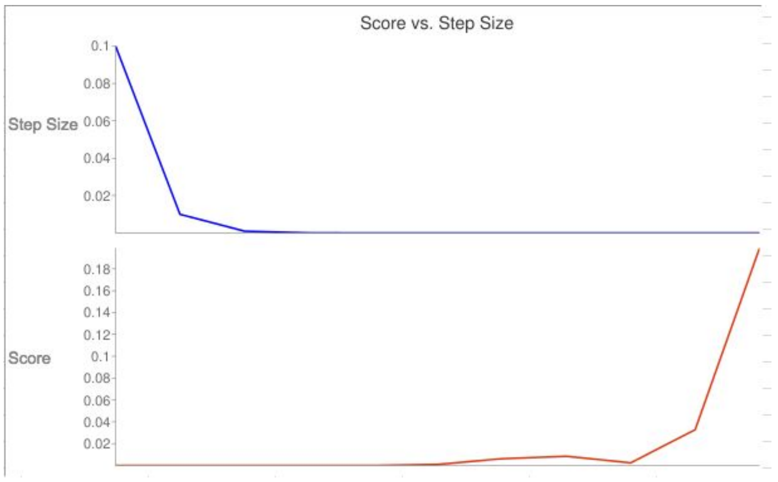


(Figure A- This diagram describes the high-level design of the overall solution prototyped for validating the proposed combined meta-heuristic algorithm. Here A and B are the known inputs and outputs of the function that will be approximated by the solution. C is an extraneous input for which the solution will provide the output)

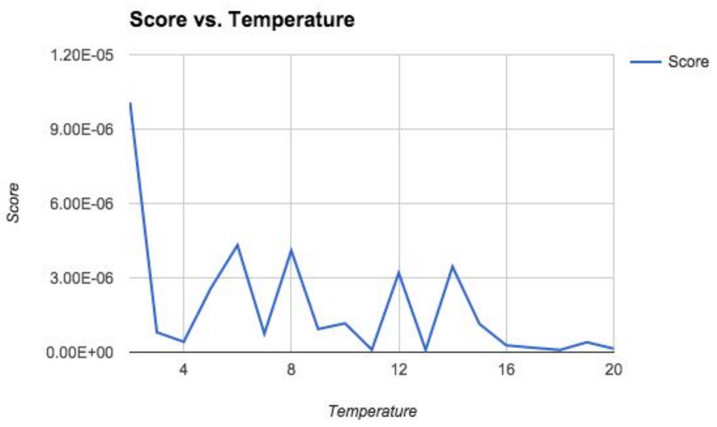


(Figure B- In this graph, which shows the regression error over several trials for each of the three main algorithms, it is evident that NM has the least fluctuation in error, with only a slight upward curve at the last trial. This fits with our expectations, seeing as the NM is the least dependent on randomness, whereas GA and SA have higher fluctuations, due to their dependence on

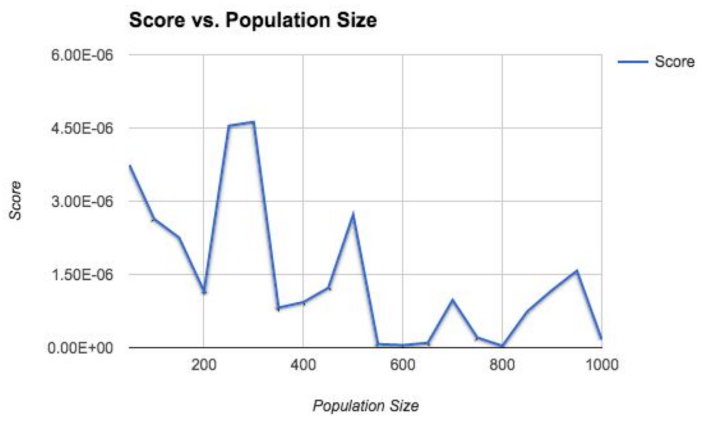
randomness. There is an extremely high outlier in the SA score, but other than that, SA's fluctuation in error is clearly much less than that of GA's.)



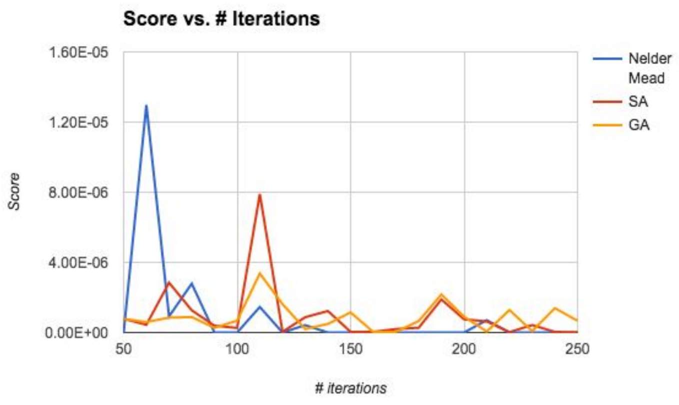
(Figure C- This figure shows that the regression score is the lowest when the step size is between 0.001 and 0.01, therefore allowing us to infer that the ideal step size is 0.0055. One might think that the lower the step size, the lower the score, but after a certain point, the score starts to increase instead of decrease.)



(Figure D- This figure shows that the regression score is the lowest when the highest temperature in the SA algorithm is set to some value between 16 and 20, because the score is closest to 0 at those points. It is evident that the score fluctuates greatly, no matter what the temperature, in part due to SA's dependence on randomness.)



(Figure E- The ideal population size lies between 550 and 650, seeing as this is one of the only ranges where the score is closest to 0. Again, similar to SA, the score fluctuates greatly just in part due to its randomness.)



(Figure F- As the number of iterations increase, after a certain point, the score does not decrease, due to the issue of overfitting, which renders the ANN unable to accurately predict functions with great accuracy. Surprisingly, the ideal number of iterations appears to lie in the same range for all three of the algorithms– somewhere between 160 to 170.)

Conclusions:

Our research successfully found an effective multi-algorithmic approach to ensure a fairly successful optimal answer for almost any functions we tested with our final algorithm. We developed a metaheuristic combined algorithmic approach, which implemented the GA, SA, and NM algorithms for the supervised training of the ANN. Our final new heuristic related to the idea that when an algorithm is trained from general to specific (in other words, most randomness to least randomness), we hope to obtain an optimal ANN for almost every situation. As a result of this approach, we were able to interpolate and extrapolate with a consistently high percentage of accuracy for almost any given function. Unfortunately, this approach tended to work better on complex trigonometric functions, as opposed to cubic, quartic, and other higher degree polynomials, mainly due to the fact that the normalization functions do not accept inputs/outputs beyond a certain range; at the moment, this input range is -200 to 200, and the output range is -1000000 to 1000000, but this can be changed by the user when necessary.

We limited the testing of our proposed algorithm to two dimensional continuous functions at this time. However, we think our proposed algorithm could be applied to solve more complex problems by extending our research in future in the following direction.

As a first step in extending the capability of our algorithm, we plan to make implementation model of all individual training algorithms (i.e. GA, SA and NM) more robust and adaptive to the function it is asked to detect. For example, this includes allowing the fitness function for GA to adaptively change after few generations of evolution. We would also test this algorithm on several other situations, to allow this heuristic to solve any problems applicable to ANNs. Also, we would like to change the overall solution design to handle functions for multiple dimensions. For example, currently, the algorithm can detect functions of two dimensions. If the solution is enhanced to handle n number of arbitrary dimensions, then the proposed algorithm will be capable to solve very complex real world problems.

References

[1] Steve Lawrence , C. Lee Giles , Ah Chung Tsoi (1997) - Lessons in Neural Network Training: Overfitting May be Harder than Expected. *Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI-97, AAAI Press, Menlo Park, California.* From - http://clgiles.ist.psu.edu/papers/AAAI-97.overfitting.hard_to_do.pdf

[2] Yinyin Liu, Janusz A. Starzyk, Senior Member, IEEE, and Zhen Zhu, Member, IEEE (2008) - Optimized Approximation Algorithm in Neural Networks Without Overfitting. *IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 19, NO. 6, JUNE 2008.* From - http://www.ohio.edu/people/starzykj/network/Research/Papers/Overfitting%20TNN06-P1314R%20manuscript.pdf

[3] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov (2014) - Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research 15 (2014) 1929-1958.* From - http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

[4] Encog Machine Learning Framework. From - http://www.heatonresearch.com/wiki/Main_Page

[5] Jeff Heaton (2011) - *Programming Neural Networks with Encog3 in Java, 2nd Edition.* Heaton Research, Incorporated

[6] Mohamad H. Hassoum (2003) - *Fundamentals of Artificial Neural Networks.* A Bradford Book

[7] Simon Haykin (1999) - *Neural Networks, A Comprehensive Foundation 2nd Edition.* Prentice-Hall, Inc.

[8] Edwin K. P. Chong; Stanislaw H. Zak (2013) - *An Introduction to Optimization, 4th Edition.* John Wiley & Sons