Parametric Optimization of Artificial Neural Networks for Signal Approximation Applications

J. Lane Thames
Georgia Institute of Technology
210 Technology Circle
Savannah, GA 31407
001-912-966-7210
lane.thames@gatech.edu

Randal Abler
Georgia Institute of Technology
210 Technology Circle
Savannah, GA 31407
001-912-966-7210
randal.abler@gatech.edu

Dirk Schaefer
Georgia Institute of Technology
210 Technology Circle
Savannah, GA 31407
001-912-966-7210
dirk.schaefer@me.gatech.edu

ABSTRACT

Artificial neural networks are used to solve diverse sets of problems. However, the accuracy of the network's output for a given problem domain depends on appropriate selection of training data as well as various design parameters that define the structure of the network before it is trained. Genetic algorithms have been used successfully for many types of optimization problems. In this paper, we describe a methodology that uses genetic algorithms to find an optimal set of configuration parameters for artificial neural networks such that the network's approximation error for signal approximation problems is minimized.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning – *Parameter Learning*

General Terms

Algorithms, Design, Theory

Keywords

Genetic Algorithms, Artificial Neural Networks, Parameter Optimization

1. INTRODUCTION

Artificial neural networks (ANN or just "network" if no confusion arises) are computational systems that can be used to solve problems for a large number of application domains. These application areas include signal (or function) approximation, data classification, pattern or sequence recognition (speech, text, imaging, etc), data clustering, gaming systems, control (robotic control, process control, vehicular control, etc), and many more. When using ANNs, one needs to have domain knowledge of the input-output space being presented to the network in order to determine appropriate configuration parameters for the network before its training. Incorrect selection of configuration parameters can lead, in many cases, to a resultant network that does not perform well after training. In these cases, poor classification results from mapping input features to incorrect output classes, which leads to networks with high rates of error. Neuroevolution [1], [6], [7], [8] has been used as a method to generate ANNs via simulated evolution with genetic algorithms (GA). Traditionally, neuroevolution is used as a method to evolve the ANN's topology and synaptic weight matrices such that traditional network learning (training) operations are bypassed, and the goal is to find an optimal topology with its associated synaptic weights for some problem instance. In this paper, we investigate a methodology that uses GA to find optimal networks for signal approximation applications such that the network produces minimal error. However, instead of using the GA to evolve the ANN's topology and synaptic weights as with neuroevolution, we use the GA to find the best choice of configuration parameters that define its topology and activation (transfer) functions while allowing the ANN to train itself based on the parameters provided by the GA.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of artificial neural networks, and section 3 gives a brief overview of genetic algorithms. In section 4, we describe how the genetic algorithm is used to optimize networks and give experimental results demonstrating its efficacy for 3 types of signal approximation scenarios. We provide closure and references in sections 5 and 6, respectively.

2. OVERVIEW OF ARTIFICIAL NEURAL NETWORKS

The underlying theory of ANNs was originally inspired by biological processes. Specifically, ANNs are modeled after the human central nervous system, which consists of a very sophisticated interconnection of neurons and their associated axons, dendrites, and synapses. At the core of an ANN is the neural unit (NU) as shown in Figure 1. The ANN is created by interconnecting many neural units across several layers to form a highly connected neural network. An NU takes as its input a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Associated with each input connection x_i is a "synaptic" weight w_i, and these weights form the weight vector $\mathbf{w}$. The output of an NU represents its activation level for a particular set of inputs where the output is denoted by $u = T(z)$. $T(z)$ is the transfer function of the NU (sometimes T is referred to as the activation function). Several forms exist for the transfer function. The work described in this paper will only use three types, which are given by Equations 1, 2, and 3. Equation 1 is the logistic sigmoidal function or the logsig fuction, Equation 2 is the hyperbolic tangent sigmoidal function or the tansig function, and Equation 3 is the linear function or purelin function.

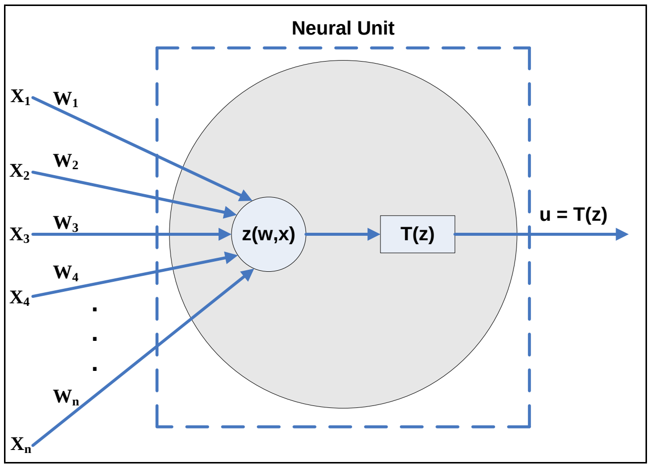

Figure 1. A conceptual diagram of the neural unit.

We will refer to logsig and tansig units collectively as sigmoidal units.

$$T(z) = \frac{1}{1+e^{-z}} \tag{1}$$

$$T(z) = \frac{e^{2z}-1}{e^{2z}+1} \tag{2}$$

$$T(z) = z \tag{3}$$

The transfer function's input, z, is the dot product of the input vector with the weight vector:

$$z = \sum_{i=1}^{n} w_i x_i \tag{4}$$

In this investigation, we study applications that use ANNs for signal approximation. It is well known in the intelligent systems community that a multilayer ANN can approximate any bounded continuous function within some arbitrary amount of error [5]. Specifically, an ANN computes an approximation function where:

$$f_a : X \rightarrow U \tag{5}$$

$$0 \leq |f - f_a| \leq \delta \tag{6}$$

In other words, the ANN can produce some approximation function f_a for the target function f such that the magnitude of the difference is within some bounded error δ. In this paper, the function f_a represents a signal approximation (or a signal function approximation).

Networks are created by interconnecting neural units to other neural units formed by one or more hidden layers, where each layer has some prescribed number of units. Networks learn how to map values in the input space to values in the output space via training, and training is provided by a learning algorithm, of which many different forms exist. Common types of ANN learning algorithms are based on the gradient decent algorithm. The basic idea is as follows. Training data is provided to the ANN in the form of (x, f(x)) tuples where x is the input data and f(x) is the target function. The learning stage takes the training tuple and sends the input value(s) into the ANN. Then, the output f_a(x) is compared to the target value and an error is calculated. This error is used to evolve the weights such that over time (training epochs) the error of the ANN is minimized to some preferably global but possibly local minimum error. For this investigation, we utilized networks based on the back-propagation algorithm, whose weight update rule is give by Equation 7:

$$w_i(t+1) \leftarrow w_i(t) + \alpha[f(x_i) - f_a(x_i)]x_i \tag{7}$$

During training, each weight vector component for each NU in the ANN is updated similar to Equation 7. As seen in Equation 7, as the approximation function approaches the actual function over increasing training epochs, the *change* in weight value w_i approaches zero such that at convergence $w_i(t+1) \approx w_i(t)$. The value α represents the learning rate, and this value determines how fast the weights evolve.

The performance of a ANN is sensitive to the selection of parameters that define its overall configuration. These parameters include, just to name a few, the type of transfer function to use in each layer, the total number of layers, the total number of units per layer, the learning rate's value, the type of training algorithm to use, and the number of training epochs to use. Furthermore, these parameters are not generalized to any given network, and in many cases, they depend on the underlying data's input-output space. If an experienced network designer has a good understanding of the input-output space, then the designer's domain knowledge and expertise allows him/her to select respectable parameter values. However, this is normally a trial-and-error process even for experienced designers. Further, the problem is more challenging when working with high-dimensional input-output spaces where underlying patterns that drive the selection of parameters are not known. Hence, methods for automated selection of optimized parameters using other computational optimizations sounds promising.

3. OVERVIEW OF GENETIC ALGORITHMS

The creation of genetic algorithms (GAs) was inspired by the biological evolutionary process. The primary inspiration is due to the fact that biological systems can adapt over time (evolve) within changing environments. Further, this adaptation can propagate through successor generations within the biological system. This adaptation-propagation scheme leads to the idea of survival of the fittest—individuals that can adapt well to changing environments have a higher probability of survival. Goldberg [3] provides a detailed treatment of GA theory and its application.

The primary operations performed by GA include chromosome representation, genetic selection, genetic crossover, genetic mutation, and population fitness evaluation. In GAs, problem domains are encoded via chromosomes in a population P(t). This chromosome encoding is usually in the form of a bit string or some numerical representation, i.e., one is required to map population members to a binary or numerical form. The population represents a particular state space of hypotheses at evolution time epoch t, where a hypothesis is a possible solution to a given problem. At each time epoch, the fitness of each individual of the population is evaluated. The fitness is evaluated with a fitness function F(h_i) where h_i is the hypothesis represented by the ith member (chromosome) of the population and the fitness F represents how well a particular hypothesis represents the solution of the given problem. In general, the GA's fitness function must be an increasing function with respect to a candidate hypothesis's response to the problem such that good solutions have higher fitness and poor solutions have low fitness.

F is computed for each member, and the next population P(t+1) is created by probabilistically selecting the most fit members of the current population. Some of the members will be part of P(t+1) in their current form, and some are selected for genetic modifications such as crossover and mutation. Crossover produces offspring from two parents whereas mutation is the act of randomly modifying the encoding features of a selected set of individuals. There are two important design issues when using a GA: 1) one must define a mapping from the input-output space of the problem into an encoding that can be used by the GA, i.e., a binary or numerical mapping, and 2) one must design a fitness function for the problem domain. The power of the GA is in its ability to encode a very large set of possible solution spaces for a given problem. They are often used successfully for optimization problems, but they have also been used for function approximation, complex circuit layout, and scheduling. As stated earlier, they are also used in neuroevolution to evolve neural networks. In this work, the GA will be used to optimize a certain set of ANN design parameters.

4. SYSTEM DESIGN AND EVALUATION

The goal of this study was to evaluate a methodology for signal approximation applications whereby a GA computes an optimal set of ANN configuration parameters via simulated evolution such that the resultant configuration produces a trained ANN whose error function is minimized. In this section, we will first present the generic process of our system design. Then, we will describe the detailed description of options used during our experimental evaluations along with experimental results.

The system design we evaluated for ANN parameter optimization consists of a two-phase evolutionary process. During phase 1, the GA has a population of chromosomes that are numerical representations of ANN configuration parameters. At each evolution time epoch, **t**, the chromosome for each population member is submitted to the ANN. The ANN maps the chromosome's numerical values to their respective parameter types, implements a self-configuration based on these values, and then learns from a training set of the problem's input-output space. Once the ANN has been trained, a set of labeled validation data from the input-output space is used to evaluate the ANN's post-training error response. This error response is then used to evaluate the fitness of the population member whose chromosome was submitted to the ANN for configuration and training. Since the goal is to find an ANN with minimal error, the error response, which is given by Equation 8, is used as input to the GA's fitness function:

$$E_i = \sqrt{\sum_{j=1}^{N}(f(x_j) - f_a(x_j))^2} \qquad (8)$$

In Equation 8, E_i is the error of the ANN configured and trained based on the chromosome of the ith population member, and the error is calculated over a total of N evaluations from the validation dataset. The fitness function for the system is given by Equation 9:

$$F(h_i; E_i) = \frac{1}{E_i} \qquad (9)$$

The fitness function is inversely proportional to the error of the ANN configured by parameters represented by the ith chromosome (i.e., the ith hypothesis). Hence, a decrease in error provides an increase in fitness, which is our objective.

The steps described above are performed for each member in the GA's population. Once each member in the population has been evaluated for fitness, the GA performs its selection, crossover, and mutation operations and then proceeds to the next evolution epoch, **t+1**. This entire process proceeds until the evolution process terminates. Note that during this phase, the training and validation data used by the ANN are extracted from a dataset from the target input-output space, and the training set is separate from the validation set.

During phase 2, which proceeds after the simulated evolution process terminates, the GA submits the chromosome from the terminal population's best fit individual to the ANN. The ANN uses this chromosome to configure its parameters and then trains from a set of phase-2 training data. Once this training is complete, the ANN is ready to be deployed for its target application.

For the experiments performed during this study, the chromosomes encode three ANN parameters, which include: 1) the number of hidden units in a single hidden layer, 2) the transfer function to use at the hidden layer, and 3) the transfer function to use at the output layer. We used numerical chromosome encodings in the GA, and the mapping from numerical to transfer function types is shown in Table 1.

Table 1. Symbolic to numeric mapping scheme for the experimentation.

Transfer Function	Mapping
logsig	$0.0 \leq y_j < 0.5$
tansig	$0.5 \leq y_j < 1.5$
purelin	$1.5 \leq y_j < 2$

The chromosome encoding was defined by $h_i = (y_1, y_2, y_3)$ where y_1 represented the number of nodes in the hidden layer, y_2 represented the transfer function for the hidden layer, and y_3 represented the transfer function of the output layer. So, j = 2 or j = 3 in Table 1. For example, if the GA produced a chromosome for $y_2 = 0.2$, then the ANN self-configured its hidden layer to use the logsig transfer function. For the number of hidden units, the ANN rounded the numerical value to the nearest integer. For both experiments, we use a back propagation ANN with a learning rate of 0.1. The GA used numerically encoded chromosomes, as stated above, with a fixed population size of 20 chromosomes, and we used arithmetic crossover, uniform mutation, and normalized geometric selection. (Refer to [4] for a description of these options).

4.1 Evaluation with the Boolean XOR Benchmark

The Boolean XOR function falls into a class known as non-linearly separable functions [5]. Because of this, the XOR is commonly used as a benchmark to test machine learning and artificial intelligence algorithms. For this experiment, a brute force technique was implemented over the entire set of possible configurations under study. The results from these ANN configurations were compared to the best fit (optimal) configuration produced by the GA. All possible combinations of the transfer function set {logsig, tansig, purelin} for both the hidden and output layer were evaluated, giving a total of 9 combinations. Further, each possible combination was evaluated over a number of hidden units ranging from 1 to 20. Overall, 9*20 = 180 configurations were compared with respect to their error response. Figure 2 shows the ANN XOR approximation error over the 180 configurations. The following notation is used to describe the data in the graph: LS = logsig, TS = tansig, and PL = purelin. The notation X-Y represents X as the hidden layer transfer function and Y as the output layer transfer function, i.e. LS-TS is an ANN with logsig for the hidden layer transfer function and tansig for the output layer transfer function.

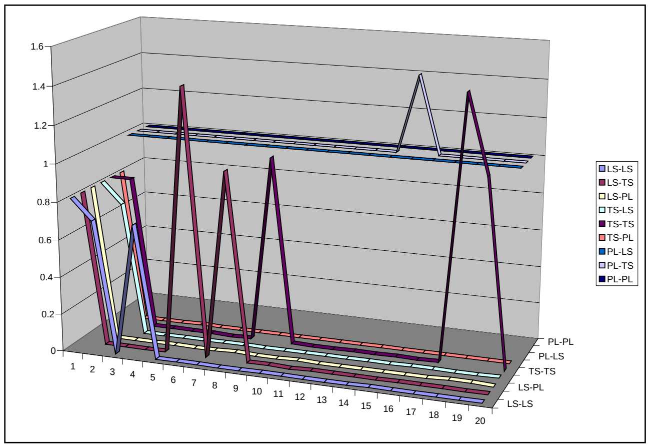

Figure 2. ANN error versus the number of hidden units for the nine different T(z) combinations.

As seen in Figure 2, the ANNs with purelin used at the hidden layer do not approximate XOR very well—the error does not go below 1 for any number of hidden units. The LS-PL and TS-PL gave the best results in terms of convergence and stability (they both converged to small error at two hidden units and retained a small error through 20 units). LS-LS, LS-TS, TS-LS, and TS-TS did well, but the error fluctuated over the different number of hidden nodes. Table 2 shows the average and standard deviation of the error for the nine combinations across the hidden unit range.

Table 2. Average and standard deviation of ANN XOR approximation error using the brute force evaluation.

	Average	Std Dev
LS-LS	0.115	0.271
LS-TS	0.165	0.405
LS-PL	0.042	0.182
TS-LS	0.080	0.234
TS-TS	0.255	0.461
TS-PL	0.041	0.183
PL-LS	1.000	0.000
PL-TS	1.020	0.093
PL-PL	1.000	0.000

For the GA approach, we ran simulated evolution over 60 generations with GA options as described above. The population size of the GA was fixed at 20 members. Figure 3 shows the average and best population fitness over the 60 generations.

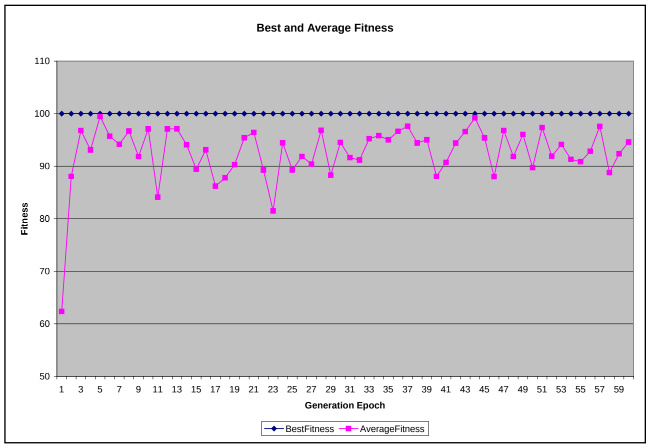

Figure 3. Best and average fitness over the 60 generations.

The fitness values shown in Figure 3 are computed by the GA with Equation 9. The near-constant line in blue is the fitness of the best member during each epoch. Notice that the GA quickly finds a solution with high fitness, i.e., finding a best fit ANN to approximate the XOR function. Table 3 provides a sample of the best fit candidate of the population as the GA evolved.

Table 3. Best fit candidate moving towards convergence.

Epoch	Number of Hidden Units	Hidden Layer Transfer Function	Output Layer Transfer Function
1	15.8916	0.2475	1.5243
2	20.0000	0.1921	1.8668
4	20.0000	0.0000	1.8429
60	20.0000	0.0000	1.8429

Table 1 is used to map the hidden and output layer transfer function numerical values to their symbolic representations. The GA's final result for the best fit member to encode this problem is in the last row of Table 3. Specifically, the number of hidden units is 20, the hidden layer transfer function is logsig, and the output layer transfer function is purelin.

4.2 Evaluation with a Sinusoidal Signal

This section explores the GA optimization methodology when using an ANN to approximate a sinusoidal signal. For this experiment, the signal (or target function) to approximate will be *f*(x) = sin(x). The experimental setup was similar to the XOR evaluation. First, a set of brute force configuration evaluations were performed for each possible transfer function configuration and for the number of hidden units ranging from 1 to 20. Figure 4 shows the approximation error for the nine configurations.

Similar to the XOR evaluation, the configurations with PL hidden layer transfer functions do not perform well. Table 4 gives the average error along with its standard deviation and minimum values for all nine configurations over the range of hidden nodes. Using the minimum error and the minimum average error as a decision factor, the ANN configuration with TS-TS or TS-PL as the transfer functions appear to provide the best approximations to the sine function. The GA used for this approximation problem was similar to the XOR evaluation except that the number of generations was set to 100 instead of 60. Figure 5 shows the best and average fitness over the 100 generations and Table 5 provides a sample of the best fit candidate over the 100 generations. Observe that the GA converged to a best fit candidate after 26 generations.

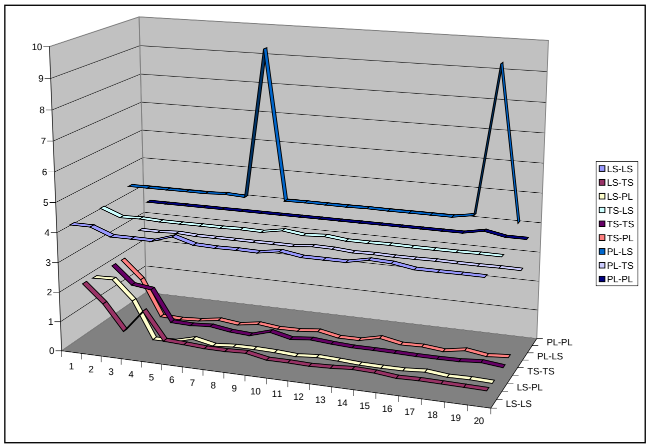

Figure 4. ANN approximation error for the nine T(z) configurations.

Table 4. Average, standard deviation, and minimum of error for the nine configurations.

	Average	Std. Dev.	Minimum
LS-LS	4.026	0.094	3.964
LS-TS	0.473	0.513	0.155
LS-PL	0.433	0.611	0.128
TS-LS	3.996	0.064	3.964
TS-TS	0.404	0.530	0.130
TS-PL	0.359	0.474	0.121
PL-LS	4.972	1.551	4.457
PL-TS	2.699	0.019	2.689
PL-PL	3.540	0.030	3.533

The best configuration produced by the GA is an ANN with 20 hidden layer units, tansig hidden layer transfer function, and tansig output layer transfer function. Comparing this result with the brute force technique shows that the GA can indeed produce good configurations in the optimal sense of minimal error for ANN configurations.

4.3 Description of a Real-World System

Part of our research deals with autonomic computing applications. We are currently investigating the idea of self-preservation controllers (SPC) as an enabler of self-configuration and self-healing for Enterprise-class servers and high-performance computing systems. A complete description of SPC is beyond the scope of this paper, but in this sub-section, we provide some preliminary results of the ANN parameter optimization techniques described in this paper that have been employed in our SPC designs.

In our SPC design, a set of applications are deployed on server systems, and these applications measure particular types of system activities related to the server's function. Activity measurements are sampled at a constant rate, and sliding windows of the resultant time series data are delivered to a wavelet transformation engine. The wavelet transformation engine produces series of wavelet coefficient profiles over different scales. We then extract a subset of the coefficient profiles and feed them into an ANN whose job is to approximate the coefficient profiles. The goal is to determine when the server experiences a significant "profile shift" induced by changing workloads. When the SPC detects these changes, it invokes higher-level applications that perform self-healing and self-configuration operations that allow the server to adapt to the changing workload environment.

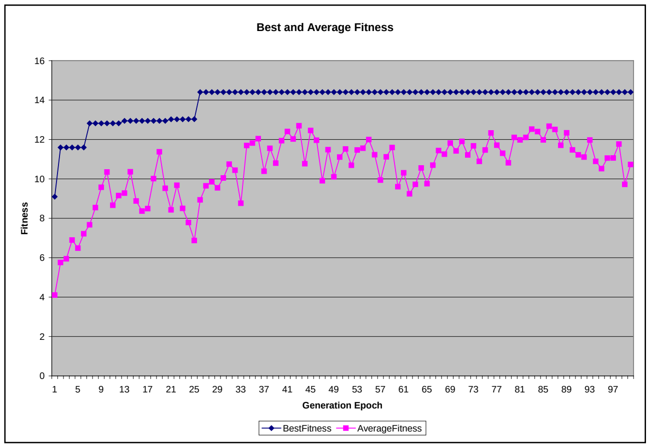

Figure 5. Best and average fitness over the 100 generations.

Table 5. Best fit candidate moving towards convergence.

Epoch	Number of Hidden Units	Hidden Layer Transfer Function	Output Layer Transfer Function
1	15.2914	1.2639	0.9166
2	19.9994	0.6405	1.3945
7	19.1401	0.5817	0.9340
13	20.0000	1.4569	1.0444
26	20.0000	0.7054	0.9339
100	20.0000	0.7054	0.9339

For this application, we require an ANN with significantly low error during normal signal approximation because the ANN's error function is used as an indicator function in the SPC. We are exploiting the fact that an ANN can produce reasonable approximation to a "continuous and smooth" function. However, an ANN will produce increased approximation error when a signal discontinuity occurs (a signal rupture). Therefore, we need an approximation system with extremely low error except for points around the signal's discontinuities. The following figure shows a sample of a web server's activity measurements (the top plot), which in this case is a measurement of file system activity, its scaled time-frequency coefficient plot in the middle, and the coefficient line from the mid-point scale on the bottom. The noticeable "spike" from the top plot represents a flash-crowding event that caused a significant shift in the resources required for the server to continue functioning. The increased demand in server resources is represented by a signal rupture in the coefficient plots. Notice that the profile shift is observable in all 3 plots.

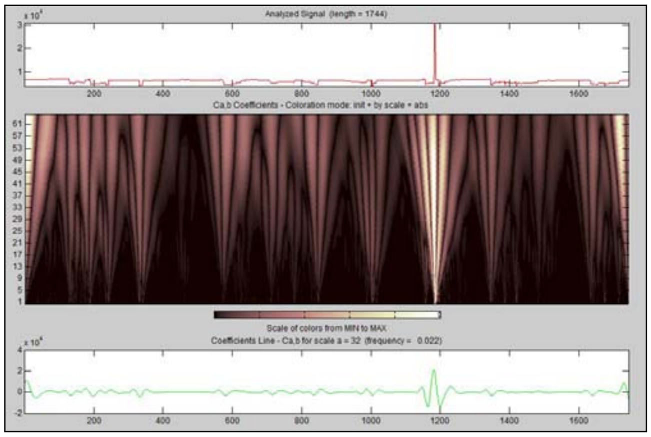

Figure 6. Wavelet analysis plots of activity measurements.

Figure 7 shows the error of the system's ANN signal approximation that has been configured using the GA optimization method described in this paper.

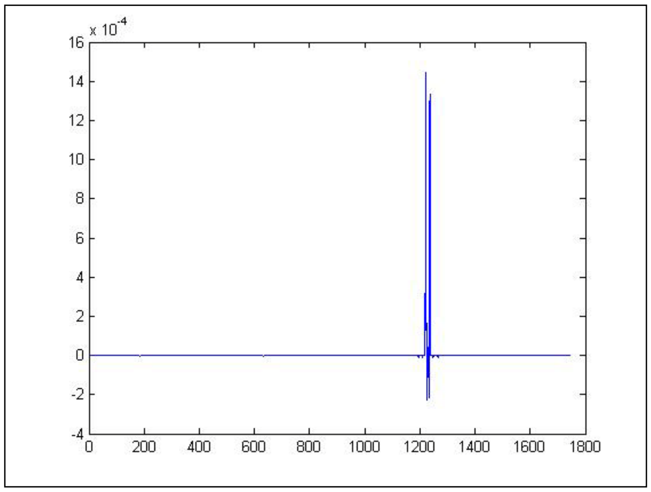

Figure 7. Error response of GA optimized ANN in the SPC.

Observe two things from Figure 7. First, the magnitude of the approximation error is extremely small, i.e., less than $16*10^{-4}$. Second, a significantly observable "spike" is seen at the point in time represented by the signal rupture. This type of behavior, extremely small error except at times when signal ruptures occur, is a fundamental requirement of our SPC designs. Current investigations have shown that optimized ANNs can achieve these requirements.

5. CONCLUSION

This paper described a method for using genetic algorithms to optimize the configuration parameters of artificial neural networks. Experimental results revealed that the method performs well in terms of finding the configuration parameters of artificial neural networks such that the error in signal approximation applications can be reduced without the need for human-assisted trial and error selection of configuration parameters. The methodology described in this paper appears to be promising for applications that require significantly small approximation error when using artificial neural networks for signal approximation. Even though the investigation reported in this paper only considered a few of the possible network configuration parameters, the method can be generalized to support optimization of virtually any configuration parameter that is traditionally provided by the neural network designer such as learning rates, number of hidden layers, and training algorithms, just to name a few. Further, the method can be seamlessly integrated into an automated, software-based neural network configuration framework.

6. REFERENCES

[1] Carrano, E., Takahashi, R., Caminhas, W., Neto, O. 2008. A genetic algorithm for multiobjective training of ANFIS fuzzy networks. In Proceedings of the 2008 IEEE Congress on Evolutionary Computation (CEC'08)

[2] The GAOT Toolbox. http://www.ise.ncsu.edu/mirage/GAToolBox/gaot/

[3] Goldberg, D. 1989. Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, MA

[4] Houck, C., Joines, J., Kay, M. 1995. A genetic algorithm for function optimization: A MATLAB implementation. Technical Report 95-09. North Carolina State University.

[5] Mitchell, T. 1997. Machine learning. McGraw-Hill. Boston, MA.

[6] Rossi, A., Carvalho, A., Soares, C. 2008. Bio-inspired parameter tuning of MLP networks for gene expression analysis. In Proceedings of the 2008 8th International Conference on Hybrid Intelligent Systems (HIS'08)

[7] Stanley, K., Miikkulainen, R. 2002. Evolving neural networks through augmenting topologies. Evolutionary Computation 10, 2, 99-127

[8] Stanley, K., Bryant, B., Miikkulainen, R. 2003. Evolving adaptive neural networks with and without adaptive synapses. In Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC'03)