A Hybrid Particle Swarm and Neural Network Approach for Reactive Power Control

Paulo F. Ribeiro, W. Kyle Schlansker, *Member, IEEE*

Abstract—**Significant research has been done on training Artificial Neural Networks (ANN) with Particle Swarm Optimization (PSO) rather than the standard technique of back-propagation [of errors], yet little work has been done to combine the training method with an optimal network topology and transfer function. This paper will discuss the use of PSO to select optimal inputs, topologies, and transfer functions for ANN. Brief descriptions of ANN, as well as the PSO algorithm itself, will be followed by a concrete application of the proposed marriage of the two. As an example, the Reactive Power Control problem will be solved with the methods presented and the results shown graphically.**

***Index Terms*—Particle Swarm Optimization, Neural Network**

I. Introduction

PARTICLE Swarm Optimization (PSO) has been an increasingly hot topic in the area of computational intelligence. PSO is yet another optimization algorithm that falls under the soft computing umbrella that covers genetic and evolutionary computing algorithms as well. As such, it lends itself as being applicable to a wide variety of optimization problems. One application that PSO has had tremendous success is in the training of Artificial Neural Networks (ANN), a fellow soft computing technique.

Typically, ANN applications of PSO have only been used to find optimal weights for a given network. Networks not only need appropriate weights, but also the appropriate topology and neuron transfer functions. The PSO algorithm does not differentiate between the variables it optimizes, meaning that it is not only capable of optimizing network weights, but also any network parameters that are allowed to be variables. This means that PSO can be used to optimize all parameters of a network: the number of layers, input neurons, hidden neurons, the type of transfer functions etc. This paper will focus on optimizing the weights, transfer function, and topology of an ANN constructed for reactive power control.

A system under optimal power conditions operates with a high power factor. The power factor of a system is composed of two elements, active power and apparent power. Active power is the useful power. Apparent power is the aggregate of active power and reactive power. The power factor ratio is given in (1).

$$\text{PowerFactor} = \frac{\text{ActivePower}}{\text{ApparentPower}} \quad \textbf{(1)}$$

This ratio is also equal to the cosine of the angle between the voltage and the current of the system. This equality may be seen in (2).

$$\text{PowerFactor} = \cos(\phi) \quad \textbf{(2)}$$

As previously mentioned, the higher the power factor, the more the apparent power is being used, whereas a low power factor adds strain on the system. It is desirable to keep the power factor of a system higher rather than lower. Low power factors result in equipment failure and higher transmission costs[1]. To do this, correction measures must be implemented.

It is more common than not for a system to have a lagging power factor. Fortunately this problem is easily remedied by injecting the appropriate amount of reactive power into the system's grid. Without integrating these corrective measures, the supplier of the power would be required to manage the extra reactive power demand leading to increased cost for the customer[2].

Implementation of client-side reactive power control requires a more robust and pseudo-intelligent control system. The control system is now responsible for controlling the reactive power generators to inject the appropriate amount of reactive power at the right time. Power systems are complex, and load demands are constantly changing. There are a vast amount of variables involved in a power system and there is no hard-computing method to determine how much reactive power should be added to a system for a given time. The soft-computing world offers a variety of methods shown to work remarkably well for the reactive power control problem. These methods consist of neuro-fuzzy systems, ANN, and

many combinations of the two such as Adaptive Neuro-Fuzzy Inference Systems[2].

ANN have been shown to be powerful and robust solutions for the control of reactive power. However, no one network is appropriate for all systems. Instead, custom networks are built and trained for each separate system. These networks differ in their weights, neuron transfer functions, topology, and other network parameters. Thus it is appropriate to be able to construct a robust network fairly quickly and with ease when given a list of possible network inputs variables. This paper proposes a method of building an optimal ANN when given sufficient data characterizing the power system dynamics. The data used to build and train an ANN with the proposed method was taken from an automobile industry and represents the voltage, current, and reactive power patterns taken every 10 minutes over a 24 hour period. This data was graciously made available for study by Dr. Ajith Abraham, who used it in a similar paper, "Neuro-Fuzzy Paradigms for Intelligent Energy Management"[1].

Brief introductions to ANN and PSO are given before the proposed method of network construction and training so that when proposed, the reader will have a basic understanding of how the method works.

II. ARTIFICIAL NEURAL NETWORKS

A. Composition

Artificial Neural Networks, heron referred to as ANN, are an attempt at modeling the processing power of the human brain. Humans are able to adapt to new situations and learn quickly when given the correct context. Computers are relatively slow at performing simple human tasks such as recognizing a lizard in a painting of the jungle. ANN work by simulating the structure of the human brain. At their basic level they consist of a network of neurons connected by synapses.

Neurons are the basic element of an ANN. Neurons accept inputs from other connections and produce an output by firing their synapse. Neurons typically perform a weighted sum on all of their input connections and then pass it through a transfer function to produce its output. A simple block diagram showing the process of a neuron applying the transfer function to its inputs before emitting its output may be seen in Fig 1. The traditional ANN is a binary network in which a synapse either fires or doesn't fire. This type of transfer function is a step function in which the neuron compares its weighted sum to a threshold and then either emits a 1 or a 0 (fires or doesn't fire its synapse). While binary networks have their uses, most engineering applications involve the real number system. ANN have thus been adapted to use real numbers. The principles are the same, but rather than only outputting a 1 or 0, a neuron can output a real number on any range, typically [0, 1].

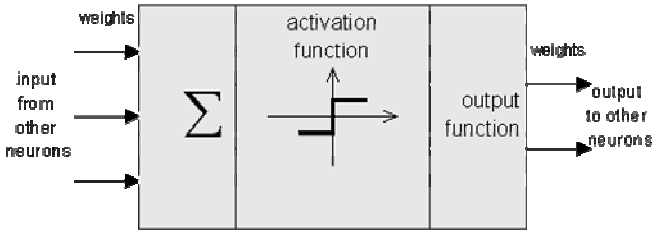

Figure 1. Neuron block diagram

ANN are organized into layers. There is always an input layer, and always an output layer. There may be any number of hidden layers, with the stipulation that there is at least one. The hidden layers are the root of the network. They perform the actual calculations of the network. A three layer feed-forward network, also called the perceptron or the universal approximator, is shown in Fig 2.

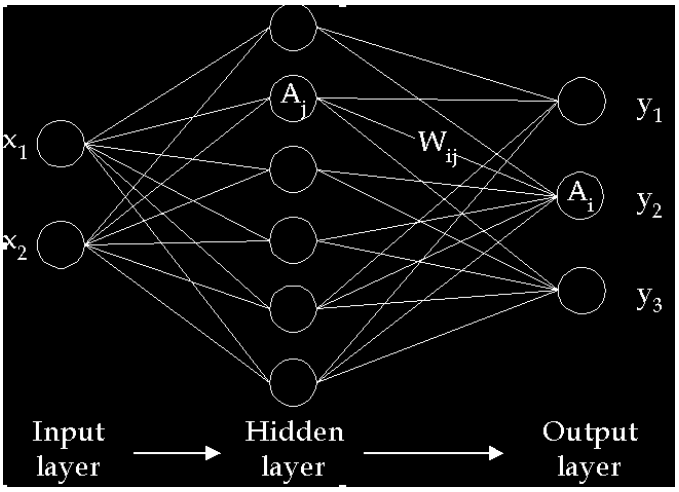

Figure 2. Three-layer neural network

A network is considered operational when it is given a set of input values and the output layer produces the expected result. The result is calculated by the topology of the neurons. Each neuron connection may be weighted differently. Each neuron may have a different transfer function (though usually they are the same). A valid network is one that has arranged itself in such a manner that produces the correct output. To facilitate the proper arrangement of a network, one must train the network.

B. Training Neural Networks

There are several methods of training ANN. Back-propagation is by far the most common. This method is where the network back-propagates its errors when training. An

ANN trains on a set of data which consists of inputs and an expected output. The training process is as follows:

1. Read in the inputs and expected outputs
2. Calculate network result by performing weighted sums and transfer functions
3. Compare network result with expected result
4. Compute and update fitness value based on comparison.
5. Repeat 2-3 until all training points are finished
6. Adjust weights in the appropriate direction to maximize fitness.
7. Repeat 1-6 until acceptable fitness value is found

The Back-propagation method is simply a gradient type adjustment for weight modification. While intuitive and effective it also may take an extremely long time to train a network. This paper suggests the use of PSO as the training algorithm.

C. Topologies

The topology of a network defines the way the neurons are connected to each other. There is a myriad of ways a network can be arranged, but various arrangements fall into two distinct categories, namely recurrent and feed-forward.

Recurrent networks are networks in which internal connections form loops. That is, when the output of one neuron also somehow effects its own input. In computer science terms, a recurrent network occurs when the dependency graph contains one or more cycles. Fig. 3 shows a directed graph representing the neurons of a recurrent network. Recurrent networks inherently contain feedback. Like any system, this feedback can grow and cause problems if it is unbounded. This feedback causes instabilities in the network, and is hard to control when using non standard training methods.

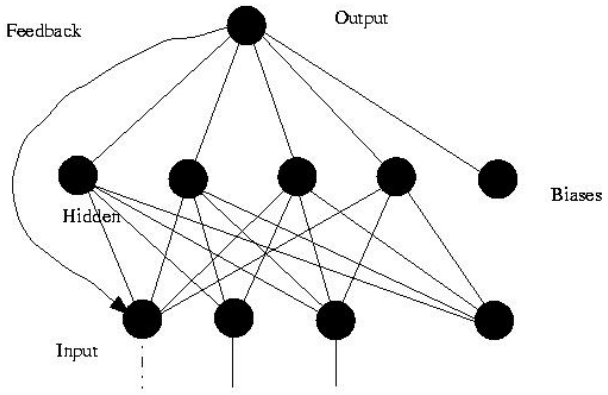

Figure 3. Network with feedback

Feed-forward networks on the other hand are very stable. They do not contain the instabilities that recurrent networks due because they are required to have acyclic dependency graphs as shown in Fig. 4. The researched discussed here only evolves the feed-forward network topologies in order to avoid the instabilities associated with feedback.

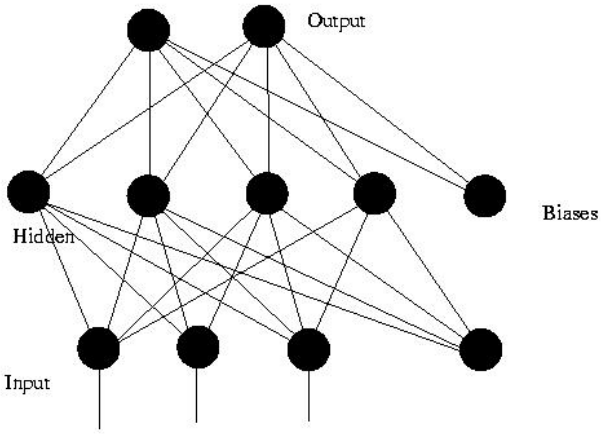

Figure 4. Network without feedback

D. Transfer Functions

Each neuron is associated with a transfer function which operates on its total input. The total input of a neuron is defined as weighted sum of its input connections less some bias. Binary ANN either output a 0 or a 1, so their transfer functions are limited to threshold and step-like transfer functions. Real number ANN are allowed to output any real number value. Transfer functions are typically picked to map their inputs onto a real number range that matches the expected output of the network. For instance, if valid answers for a network lay on the range [0,10] and if there are 10 hidden neurons, then those neurons would each have transfer functions that map to [0,1]. The alternative is to have the transfer function of the output neuron map the range of its inputs into the valid range of outputs. In standard practice, it is good to do both of these. Theoretically one is still left with an infinite amount of solutions, by choosing the range of the transfer functions it cuts down on the number of invalid solutions that a network may attempt during training.

Another danger that needs to be addressed by the transfer function is giving one input more say than another simply by its value rather than its weight. If one input is valid on the domain [0, 50] and another on [0, 2] and the output is on the range [0, 50] then the first input will dominate the second if it is not properly handled. One might assume that connection weights account for the scaling. This is true in part, but is dependent on the range used for connection weights. The more appropriate way to deal with this situation is to use a transfer function which scales all inputs onto an equal range. This ensures that large inputs do not dominate smaller ones since each can contain equal information just on different scales. A good transfer function for this type of scaling is the sigmoid function.

The sigmoid function is given in (3), and its graph depicted in Fig. 5 with alpha set to unity. The sigmoid function maps its input onto the range [0,1]. This is the initial transfer function used in this research, with alpha (the slope of the curve) being a variable.

$$\text{Sigmoid}(x) := \frac{1}{1 + e^{-\alpha \cdot x}} \quad \textbf{(3)}$$

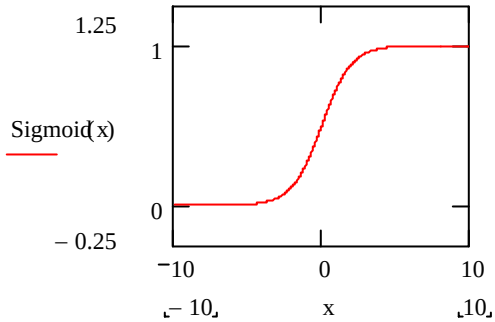

Figure 5. Plot of the sigmoid transfer function

III. PARTICLE SWARM OPTIMIZATION

A. *Description*

Nature itself is the most complex system known, and it works gracefully, almost magically. There are millions of groups or communities of humans and animals that all live in harmony with nature. They have survived for millions of years. How? To what do they attribute their sustainability? There is obviously some valuable knowledge embedded within social systems. By examining the interaction of members within a social system, it is possible to apply the findings to other scientific problems.

James Kennedy and Russell Eberhart did just that when they developed swarm theory. Swarm theory is based on the collective intelligence that arises from the cooperation of (often unintelligent) individual members within a social system[14].

PSO exploits the cooperation aspect and applies it to computer science and engineering optimization problems. It does this by modeling a social system. In this model, the system is populated with individual particles representing possible solutions to the problem. The particles themselves are not intelligent. They do not sit down and think about the best way to solve the problem. They simply follow a predefined set of rules. In this aspect, they are much like Finite Cellular Automata. The PSO algorithm evaluates all particles according to a common fitness function—a function that maps the state of a particle to how good of a solution it is. The particle with the highest fitness is considered the best of the group. All other particles then learn from this best, just as members of a social system learn from those around them. No particle is identical, but they each take on attributes of each other that will help their fitness improve.

B. *PSO Model*

A Particle Swarm is a population of individuals each of which contains the appropriate amount of features or values to place it in a Swarm problem space. The individuals are arranged in neighborhoods in which they can share information. The mathematical definition of a neighborhood is “the set of points surrounding a specified point each of which is within a certain specified distance from the specified point”[dictionary.com]. For instance the bit string “01110” is composed of 5 bits. Letting bit number 3 (the middle bit) be the specified point. A neighborhood of size 3 would include the entire bit string, two to the left and two to the right. Like ANN, these neighborhoods themselves can have different topologies, though these topologies are drastically different from the topologies of ANN. Typical topologies for Particle Swarm neighborhoods are circular or star-shaped.

In a Particle Swarm, each individual is influenced by its closest neighbors. A single particle is a possible solution to the problem. The particle’s position in the problem space defines the solution. Particles fly though the problem space and adjust the trajectory based on influence from their neighbors.

Each particle is randomly initialized to a certain position in the problem space. The number of dimensions in the problem space is equal to the number of components there are to optimize. A particle updates its position according to the Euler integration equation for physical movement given in (4).

$$\vec{x}_i(t) = \vec{x}_i(t-1) + \vec{v}_i(t) \quad \textbf{(4)}$$

The velocity component of the Euler integration equation is what includes the stochastic element of the Particle Swarm. Each particle’s velocity vector is computed based on its current velocity (randomly initialized) and the velocity of the best particle in its neighborhood. Not only that, but two stochastic variables are incorporated as well. One of these variables weights the portion of the velocity vector corresponding to the particle’s previous velocity, while the other weights the portion corresponding to the velocity of the best particle in the neighborhood. The sum of the two particles is generally a constant defined as the random range of a Particle Swarm[14]. The velocity vector is updated prior to the position vector and is given by the (5).

$$\vec{v}_i(t) = \vec{v}_i(t-1) + \rho_1 \cdot \left[\vec{p}_i - \vec{x}_i(t-1)\right] + \rho_2 \cdot \left[\vec{p}_g - \vec{x}_i(t-1)\right] \quad \textbf{(5)}$$

For a given problem, a certain number of particles (typically 20-50) are initialized and let loose to swarm the problem space in search of an optimal or near optimal solution. The particles continue to swarm until the exit criteria has been met.

C. Algorithm

The algorithm for a neighborhood of particles is shown in the pseudocode below presented by AdaptiveView[13].

```
///
/// Begin definitions of control variables
///
numberOfParticles = 40
numberOfNeighbors = 4
maxIterations = 1000

/// set limits for location changes
deltaMin = -4.0
deltaMax =  4.0

/// set individuality and sociality
iWeight = 2.0
iMin = 0.0 /// low  stochastic weight factor
iMax = 1.0 /// high stochastic weight factor
sWeight = 2.0
sMin = 0.0 /// low  stochastic weight factor
sMax = 1.0 /// high stochastic weight factor

///
/// Next 3 variables related to problem solution space.
/// See function "test" for definition of fitness function.
///
initialFitness = -100000
targetFitness = 0
dimensions = 4 /// dim. of solution space

///
/// End of control variable definitions.
///

/// set up particles' next location
 for each particle p do {
  for d = 1 to dimensions do {
    p.next[d] = random(...)
    p.velocity[d] = random(deltaMin,deltaMax)
  }
  p.bestSoFar = initialFitness
}

/// set particles' neighbors
for each particle p do {
  for n = 1 to numberOfNeighbors do {
    p.neighbor[n] = getNeighbor(p,n)
  }
}

/// run Particle Swarm Optimizer
while iterations <= maxIterations do {
  /// Make the "next locations" current and then
  /// test their fitness.
  for each particle p do {
    for d = 1 to dimensions do {
      p.current[d] = p.next[d]
    }
    fitness = test(p)
    if fitness > p.bestSoFar then do {
      p.bestSoFar = fitness
      for d = 1 to dimensions do {
        p.best[d] = p.current[d]
      }
    }

    if fitness = targetFitness then do {
      ... /// e.g., write out solution and quit
    }
  } /// end of: for each particle p

  for each particle p do {
    n = getNeighborWithBestFitness(p)
    for d = 1 to dimensions do {
      iFactor = iWeight * random(iMin,iMax)
      sFactor = sWeight * random(sMin,sMax)
      pDelta[d] = p.best[d] - p.current[d]
      nDelta[d] = n.best[d] - p.current[d]
      delta = (iFactor * pDelta[d]) + (sFactor * nDelta[d])
      delta = p.velocity[d] + delta
      p.velocity[d] = constrict(delta)
      p.next[d] = p.current[d] + p.velocity[d]
    }
  } /// end of: for each particle p
} /// end of: while iterations <= maxIterations
end /// end of main program

///
/// Beginning of function code
///

/// Return neighbor n of particle p
function getNeighbor(p,n) {
  ...
  return neighborParticle
}

/// Return particle in p's neighborhood
/// with the best fitness
function getNeighborWithBestFitness(p) {
  ...
  return neighborParticle
}

/// Limit the change in a particle's
/// dimension value
function constrict(delta) {
```

```
  if delta < deltaMin then
    return deltaMin
  else
  if delta > deltaMax then
    return deltaMax
  else
    return delta
}

/// When given a particle, test() applies its coordinates
/// to the problem and returns a fitness value.
function test(particle) {
  x = particle.current[1] /// dimension 1
  y = particle.current[2] /// dimension 2
  z = particle.current[3] /// dimension 3
  w = particle.current[4] /// dimension 4
  f = 5*(x^2) + 2*(y^3) – (z/w)^2 + 4 /// problem
  if (x*y) = 0 then n = 1 else n = 0 /// favor x and y > 0
  return 0 – abs(f) - n /// fitness value; 0 = optimal
}
```

D. *Training Neural Networks with PSO*

The PSO algorithm is vastly different then any of the traditional methods of training. PSO does not just train one network, but rather trains a network of networks. PSO builds a set number of ANN and initializes all network weights to random values and starts training each one. On each pass through a data set, PSO compares each network's fitness. The network with the highest fitness is considered the global best. The other networks are updated based on the global best network rather than on their personal error or fitness.

Each neuron contains a position and velocity. The position corresponds to the weight of a neuron. The velocity is used to update the weight. The velocity is used to control how much the position is updated. If a neuron is further away (the position is further from the global best position) then it will adjust its weight more than a neuron that is closer to the global best.

The particles in this context are the individual networks rather than the neurons. The dimension of the hyperspace in which the particles reside may be found by the number of neurons in the network. Thus, the positions of each neuron in a network effectively place a network at a certain location in the problem hyperspace.

There are maxima and minima in this hyperspace. Particles fly around the hyperspace, updating their position according to the best position found by their fellow particles. Eventually a particle will come across optima of sorts. When this occurs, it will continue to climb the hill towards the optima. Fellow particles will quickly see this and adjust their positions to swarm towards the optima. What ensues is that a team of these particles cover the optima area. If the associated fitness at this optimum is acceptable, then the network stops training. If it a maximum has been found, but is sub-par, then the positions of the neurons are randomized and the hunt restarts. There are an infinite number of solutions for a real-number ANN. PSO assures that the network will never get stuck trying to converge to a false maxima. Instead, as alluded to previously, PSO takes on two major methods of solution hunting. Namely, exploration and exploitation**.** Exploration is the generalized search for maxima and minima. This occurs with a large number of particles swarming broadly over the entire hyperspace. Exploitation is the convergence on a particular maxima or minima. Generally speaking, particles start by exploration, if a specific optimum looks particularly appealing, it will decide to examine it a bit closer. This is when it switches to exploitation mode. In exploitation mode, the position of a particle does not update as rapidly as in exploration. This has the effect that the particles take a smaller step size, and won't overstep a possible optimal solution. This part of the algorithm is not inherent to PSO but is an addition to it. It is controlled by an annealing factor applied to the positional update equation.

The annealing factor stems from the simulated annealing method often used in evolutionary programming. Basically, as implemented here, the annealing factor starts off near 1. When multiplied against the positional update equation, it has the effect of exploration (it allows the update of the full positional change). As a particle nears an optima, it decreases its annealing factor (either exponentially or linearly) thus taking smaller steps.

IV. PSO MEETS ANN

A. *Constructing a Network Swarm*

This research focuses around training ANN with PSO. To do this, a population of networks must be constructed. In this situation, each individual network is treated as a particle that is located in the problem hyperspace according to the weights of the network elements. Generally, a population of about 20 particles (or 20 networks) has been found to work well.

This population of networks is built and initialized and then formed into PSO neighborhoods. For the example application, a global neighborhood was found to work best (one in which any network is able to teach any other network). This construction is called the topology of the swarm system.

B. *Training the Swarm of Neural Networks*

After the networks have been constructed and arranged topologically, they begin the training algorithm. The training algorithm is similar to the one given in section II, but not the same. The training process is listed below.

1. For each network, iterate over the training data set and keep a running sum of the network error.
2. Compare all of the network errors to find the best network in the neighborhood.
3. If one of the networks has achieved the minimum error required, record its weights and exit the program.
4. Otherwise, for each network, execute the PSO algorithm to update its position and velocity vectors.
5. Loop to 1.

Once a particle has achieved the required fitness, a solution has been found. This particle then changes from being a solution searcher into a production neural network as depicted in Fig. 6.

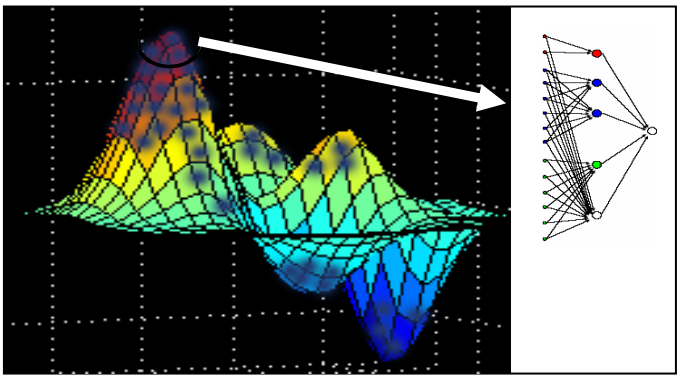

Figure 6. Particle search of the problem hyperspace

V. REACTIVE POWER CONTROL PROBLEM

A. *Description of Problem*

The voltage requirement of power systems swings depending on the load of the system. This causes losses in the system. An effective method to control this is to inject reactive power to compensate for the swing. The problem that arises with this solution is how much reactive power to inject, and when to inject it.

B. *Proposed Solution*

Power Systems are complex systems with a large number of control variables. ANN have proven to be excellent tools for mapping complex systems to a known output. The proposed solution is to construct and train a neural network to predict the reactive power requirements for a given power system.

The training data from the aforementioned automobile plant consisted of the voltage and current for a given time as well as the required reactive power. A network was constructed which used the voltage and current values as inputs to the network. Network fitness, given in (6) was determined to be the mean square of errors for the entire training set, where error is defined to be the recorded reactive power minus the network predicted reactive power.

$$\text{Fitness} = \sum (\text{Recorded\_Value} - \text{Network\_Predicted\_Value})^2 \quad \textbf{(6)}$$

A network is deemed usable once it has met some minimal requirements for performance. The requirement used was the statistical calculation known as the multiple correlation coefficient and is given in (7). This measurement subtracts, from unity, the network fitness divided by the square of the mean subtracted by each output. As the network becomes more accurate, the resulting value approaches one.

$$R^2 = 1 - \frac{\text{Fitness}}{\sum (\text{Recorded\_Value} - \text{Mean\_Recorded\_Value})} \quad \textbf{(7)}$$

C. *Implementation*

A Java program, called Themis, was written to read in the training data, construct a given number of ANN particles, and execute the PSO algorithm on the networks. Themis, a Greek goddess, was chosen as the name because she is said to be "prophecy incarnate; her oracles derive from her sense of order and connection to nature…she personified the social order of law and custom, a reminder that social order is ultimately dependant on the natural order of the earth"[15].

Themis currently only adjusts the network weights for a given topology and transfer function. However, future versions will add the topology and transfer function selection as a PSO variable which will be optimized and selected by the PSO algorithm rather than the user.

The class collaboration diagram for the implementation code is shown in Fig. 7.

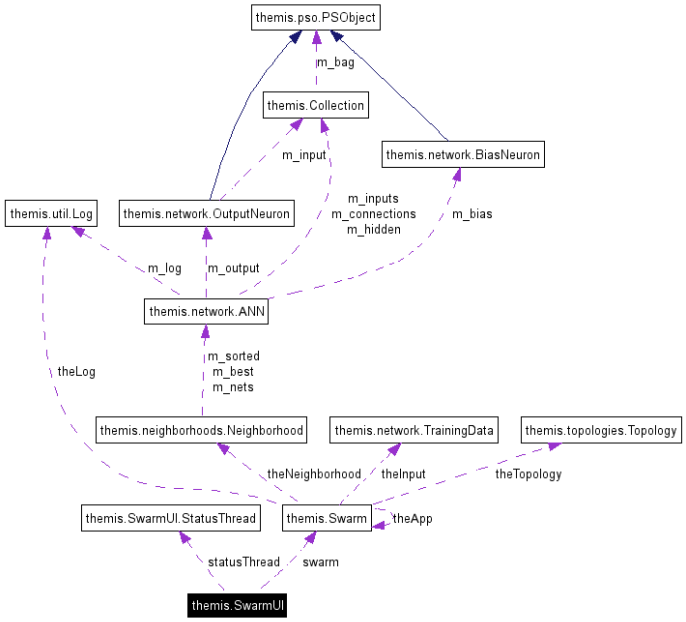

Figure 7. Class collaboration diagram

D. Results

Themis was able to construct a three layer fully connected feed-forward network consisting of two input neurons, three hidden neurons, and a single output neuron. The resulting networks performance was more than sufficient. The correlation coefficient achieved was 0.99873. A plot of recorded values versus predicted values may be seen in Fig. 8 and Fig. 9.

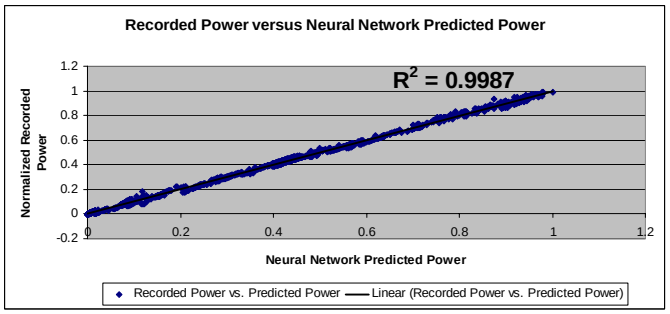

Figure 8. Correlation of recorded and predicted values

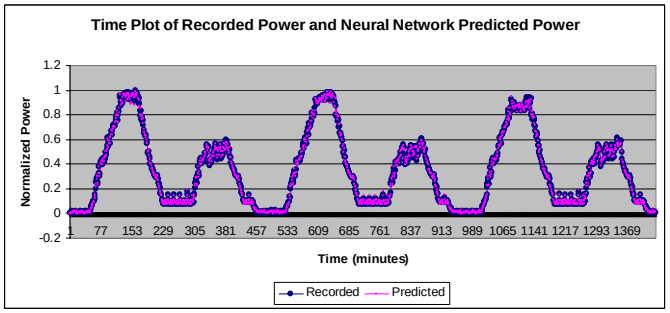

Figure 9. Time overlay of recorded and predicted values

VI. Conclusion

It is clear that Artificial Neural Networks are a very powerful and accurate tool for reactive power dispatch. Due to the very nature of Artificial Neural Networks, and soft-computing in general, there is no one solution equation. This requires networks to be customized for each system. Standard methods require the user to choose a network topology, inputs, and transfer functions for a network before training. Particle Swarm Optimization overcomes these limitations because it is blind to what it is optimizing. Any network parameter may be thrown into the mix along with the network weights to be optimized. To date Themis offers a way of selecting appropriate inputs for the network, and future versions will support the automatic selection of topology and transfer function. Themis is capable of constructing and training a network with a correlation greater than 0.99 in under a minute, placing PSO as a competitor for other top training algorithms.

Acknowledgments

The author thanks the following people for their support and invaluable insight pertaining to this paper:
Dr. Ajith Abraham, Oklahoma State University
Dr. Wayne S. Hill, Foster-Miller Inc, Waltham, MA,
Mr. Thomas Lovell, Foster-Miller Inc, Waltham, MA,
Dr. Paulo F. Ribeiro, Calvin College. Grand Rapids, MI,
Dr. Steven VanderLeest, Calvin College, Grand Rapids, MI,
Dr. Randall Brouwer, Calvin College. Grand Rapids, MI.

References

[1] Ajith Abraham, Muhammad Riaz Kahn, "Neuro-Fuzzy Paradigms for Intelligent Energy Management, Innovations in Intelligent systems: Design, Management and Applications" in *Studies in Fuzziness and Soft Computing*, pp. 285-314, Springer Verlag Germany, 2003.
[2] Ajith Abraham, An Evolving Fuzzy Neural Network Model Based Reactive Power Control, In Proceedings of The Second International Conference on Computers In Industry, ICCI 2000, Majeed A Karim et al (Eds.), Bahrain, pp. 247-253, 2000.
[3] K.H. Abdul-Rahman, S.M. Shahidehpour, M. Daneshdoost, "AI approach to optimal VAr control with fuzzy reactive loads" in *IEEE Transactions on Power Systems*, volume 10, pp 88-97, Feb. 1995.
[4] V. Ajjarapu, Z. Albanna, "Application of genetic based algorithms to optimal capacitor placement" in *Neural Networks to Power Systems*, pp 251-255, Jul. 1991.
[5] G. Cartina, C. Bonciu, M. Musat, Z. Zisman, "AI approach to optimal VAr control with fuzzy reactive loads" in *Electrotechnical Conference*, volume 2, pp 1103-1106, May. 1998.
[6] T.W.S. Chow, Y.F. Yam, "Measurement and evaluation of instantaneous reactive power using neural networks" in *IEEE Transactions on Power Delivery*, volume 9, pp 1253-1256, Jul. 1994.
[7] Yuan-Yih Hsu, Feng-Chang Lu, "A combined artificial neural network-fuzzy dynamic programming approach to reactive power/voltage control in a distribution substation" in *IEEE Transactions on Power Systems*, volume 13, pp 1265-1271, Nov. 1998.
[8] A.A. Sallam, A.M. Khafaga, "Artificial neural network application to alleviate voltage instability problem" in *2002 Large Engineering Systems Conference on Power Engineering*, pp 133-141, June. 2002.
[9] Z.E. Aygen, M. Bagriyanik, S. Seker, F.G. Bagriyanik, "An artificial neural network based application to reactive power dispatch problem" Electrotechnical Conference, volume 2, pp 1080-1083, May 1998.
[10] K. Iba, "Optimal VAr allocation by genetic algorithm" in *Neural Networks to Power Systems*, pp 163-168, Apr. 1993.
[11] Paul Pomeroy, "An Introduction to Particle Swarm Optimization", http://www.adaptiveview.com/articles/ipsop1.html, March 2003.
[12] Russell Eberhart, James Kennedy, *Swarm Intelligence*
[13] Susan Boulet, *Goddess Knowledge Cards*, http://www.daykeeperjournal.com/aarch02/0201/goddess.shtml

W. Kyle Schlansker (M'02) was born in San Antonio, TX in 1982. Since then he has attended school in both Boston MA, and Grand Rapids, MI. In May 2004 he will graduate from Calvin College in Grand Rapids, MI with a Bachelor of Science in Engineering with a concentration in computer and electrical engineering and minors in computer science and mathematics. He then plans to attend the University of Illinois at Urbana-Champaign as an electrical engineering graduate student.

He has worked as a consultant for Foster-Miller, Inc. on projects for both the Navy and NASA. He has also worked as a Java programmer for the Christian Classics Ethereal Library (http://www.ccel.org). His current work is independent research on applications of particle swarm optimization.