Visualization Tool for a Self-Splitting Modular Neural Network

V. Scott Gordon, Michael Daniels, James Boheman II, Marcus Watstein, Derek Goering, and Brandon Urban

***Abstract*—We describe and implement a visualization tool for a Self-Splitting Neural Network (SSNN). The SSNN is a modular neural network that partitions the input domain during training through the identification of solved chunks and a divide-and-conquer strategy. The visualization tool shows a 2D projection of the input domain as partitioning proceeds, highlighting the boundaries of trained regions. Greyscale can be used to contrast the ranges of outputs so that generalization can be visually assessed. The tool is useful for illustrating how the SSNN works and for comparing different learning and splitting strategies.**

I. Introduction

THE Self-Splitting Neural Network (SSNN) is a particular version of a *modular* neural network, and as such divides the input domain into partitions where a separate network is assigned to each partition.

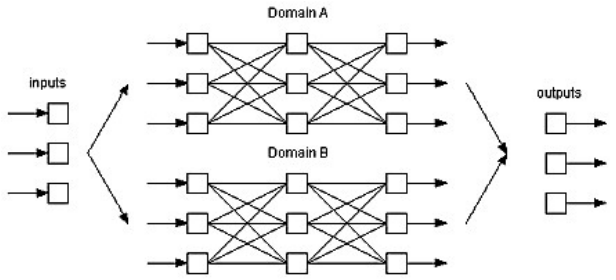

Fig. 1. Modular Network Architecture

In Figure 1, the inputs are routed to one of the networks (here shown as either Domain A or Domain B), depending in which portion of the input space those input values are contained. Whenever one network is assigned to those inputs, it and it alone produces the corresponding output values.

Manuscript received December 14, 2008.

V. Scott Gordon, Michael Daniels, Marcus Watstein, and Brandon Urban are with the Computer Science Department at California State University, Sacramento, CA 95819 USA (phone: 916-278-7946, e-mail: gordonvs@ecs.csus.edu, mdaniels@csus.edu, watsteim@ecs.csus.edu, brandon@urban-code.edu).

James Boheman II is with the California Department of Technical Services (e-mail: James.Boheman@dts.ca.gov).

Derek Goering is with Accenture, C-IV (e-mail: Goering@c-iv.net).

Modular neural networks were first described in the early 1990s by various researchers [1], [2] and explored further by Sharkey [3], [4]. Chen et al. used divide-and-conquer to assign one or more networks to each case in a sort of hybrid modular/ensemble approach [5]. Olsen-Darter and Gordon used a modular configuration of 120 neural networks for a vehicle control application, but the division of the input domain was determined by experimentation [6].

II. Self-Splitting Neural Network

The self-splitting modular approach was described in a previous study [7], and attempts to divide the problem by examining the results of training. It starts by attempting to train a single network. If it fails to learn its entire training set, that does not mean that it learned nothing. In fact, it may have learned a substantial portion of the training set. It is possible, in many cases, to extract a trained portion, and split the domain along the solved partition boundaries. Algorithm 1 illustrates the process.

Algorithm 1 – Self splitting framework

1. attempt to solve the training set
2. if success, the input domain is solved ... stop.
3. if failure:
   (a) select a variable and value(s) for splitting
   (b) assign new network(s) to each sub-domain
   (c) distribute training cases by sub-domain
   (d) recursively apply algorithm 1 to each network

Apply recursively until all networks learn their subset of the training cases.

Splitting can be described as a method of finding well-defined clusters of solved cases, and characterizing the remaining (unsolved or only partially solved) regions. The weights used to solve the trained portion would be stored, along with a description of the sub-domain it solves.

Our first version splits in a simple manner, by finding ranges of values within one of its inputs for which every case is solved. Of course it is possible that trained regions can be identified for more than one variable. As well, there may be more than one solved region for a particular variable. Our initial approach is to find the *largest solved*

978-1-4244-3553-1/09/$25.00 ©2009 IEEE

region amongst all of the inputs. In that way, we try to favor networks that generalize rather than networks that specialize. Once the variable with the largest solved chunk is identified, the problem domain is split along the boundaries of that chunk, producing *one solved partition* and *two unsolved partitions*, as shown in Figure 2.

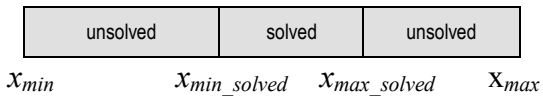

Fig. 2. Splitting by trained region for input variable *x*

The entire process of identifying solved regions and the resulting splitting is shown in Algorithm 2. Examples of the splitting process were provided in [7].

For each trained region, the ranges of input values and the network weights are stored. Note that the regions as designed *overlap*. This is to ensure that the entire range of possible values is sufficiently modeled. Overlapping does give rise to certain details that must be considered to avoid a *runaway* splitting, which can happen when one of the new sub-domains is identical to the original domain. These problems are avoided by verifying the cardinality of training cases before and after splitting.

Algorithm 2 – Identifying region boundaries

1. For each input variable *s* of the current network:
   (a) sort the training pairs on *s*
   (b) scan the pairs from lowest to highest value
   (c) find the subrange $min_s...max_s$ for which:
   (i) all pairs within $min_s...max_s$ are solved, and
   (ii) no other subrange contains more such pairs
   (d) if the size of the subrange is greater than any found so far (for other inputs), copy *s* and $min_s...max_s$ into *i* and $min_i...max_i$.
2. If an input (*i*) with a trained region was found, split on variable *i* and subrange $min_i...max_i$
   else
   split by centroid on the variable with the largest domain.

It is possible that no trained regions are found. This can happen if for all solved cases there happen to exist other unsolved cases that have duplicate input values. When no trained regions are found, or if all trained regions would give rise to a runaway split (one that duplicates an untrained region), we split the domain in half on the variable with the most values, dubbed splitting *by centroid*.

After all neural networks have been successfully trained on their respective sub-domains, they can be tested on other (untrained) cases, as described in Algorithm 3. Here, the trained multi-net acts as a modular network, described earlier in Section I. The set of inputs is routed to the particular network which was trained on values belonging to the same sub-domain as the new test case.

Algorithm 3 – Running the multi net

1. retrieve a set of new (untrained) inputs
2. for each network, if the inputs are in its sub-domain, compute the output of the network (there is usually *only one* such network).
3. if only one network responded to step 2, the output of that network is the output of the multi-net.
   if more than one network responded, the final output is averaged from the outputs of those networks.

The SSNN was tested on a variety of hard problems and was shown to be effective at learning large training sets while displaying good generalization [7]. That version utilized standard backpropagation, although of course other training methods could be used.

III. Visualization Tool

The self-splitting model described in the previous sections lends itself to 2-D visualization, particularly when the SSNN is used to solve a problem with two inputs. The input domain is portrayed as a rectangular space, with one variable extending across the X-axis, and the other along the Y-axis. When a trained subdomain is identified and splitting occurs, the region can be illustrated as a smaller rectangle within the larger one. Thus, as training proceeds, the user can watch the input domain being partitioned.

The tool was implemented, and will run on any platform with Java version 1.5 or higher and Java Web Start version 1.5 or higher. An internet connection is required when launching the program the first time, but subsequent program launches can be done without one. The system is accessible on the web at:

`http://ecs.csus.edu/~ssnn`

Upon launching the application, the main screen appears, as shown in Figure 3. The main screen is divided into several regions, allowing the user control over the neural network configuration, training and testing data, training settings, as well as what items are included on the visualization pane. The user can also specify a background image to overlay over the visualization pane if that helps in visualizing how the target problem is being solved. There is also the option for the user to save entire predefined scenarios containing all of the settings, training, and testing data for a particular problem. A first-time user would probably want to start with a predefined scenario in order to become acquainted with the tool.

For example, consider the *two-spiral problem* in which the neural network attempts to learn to differentiate between data points that exist on one of two interleaved spirals [8] (the website is configured with this problem already included as a predefined scenario). Neural network settings would include topology information such as number of inputs (`2`), number of outputs (`1`), number of hidden layers, number of nodes per layer, and whether or not the user wishes to include unity bias units. Training parameters include learning rate, momentum rate, success criteria (`0.4`), and the number of iterations to run before splitting. Training and testing datasets are also specified. The input values of the training set are shown as red dots on the visualization pane.

As training proceeds, solved regions (or *chunks*) are shown as rectangular areas in the center pane. Chunks colored green are regions that have been solved. The chunk currently being trained is shown in yellow. A text log showing details for each solved region is included in the rightmost pane.

When training completes, the network is run against the provided test data. Summary information for training and testing results is produced, which includes total training iterations, number of networks produced, number of chunk splits, number of centroid splits, the duration that was required for training (in seconds), and the level of generalization expressed as a percentage of (untrained) test cases correctly solved by the modular network.

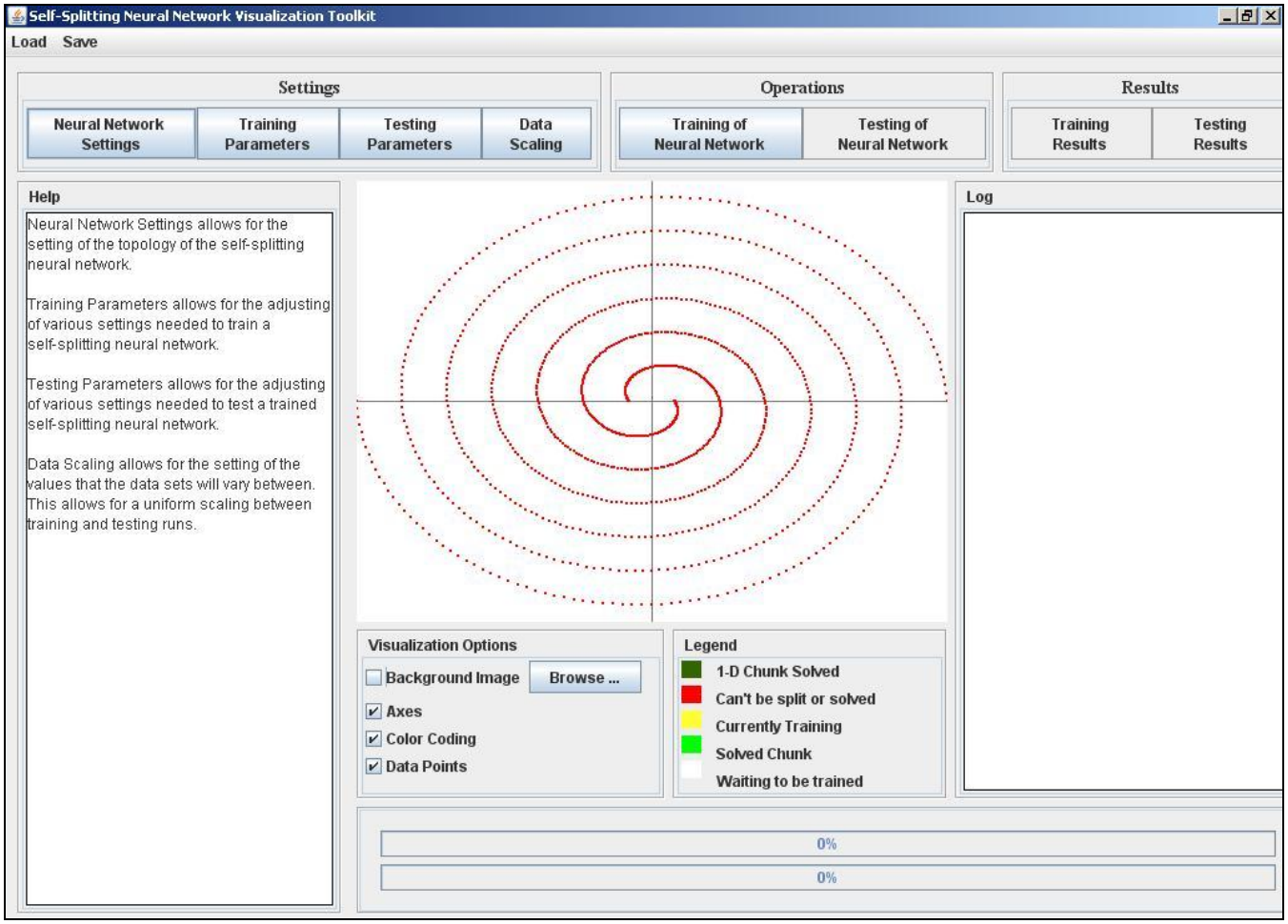

Fig. 3. SSNN Visualization Tool – *main screen*

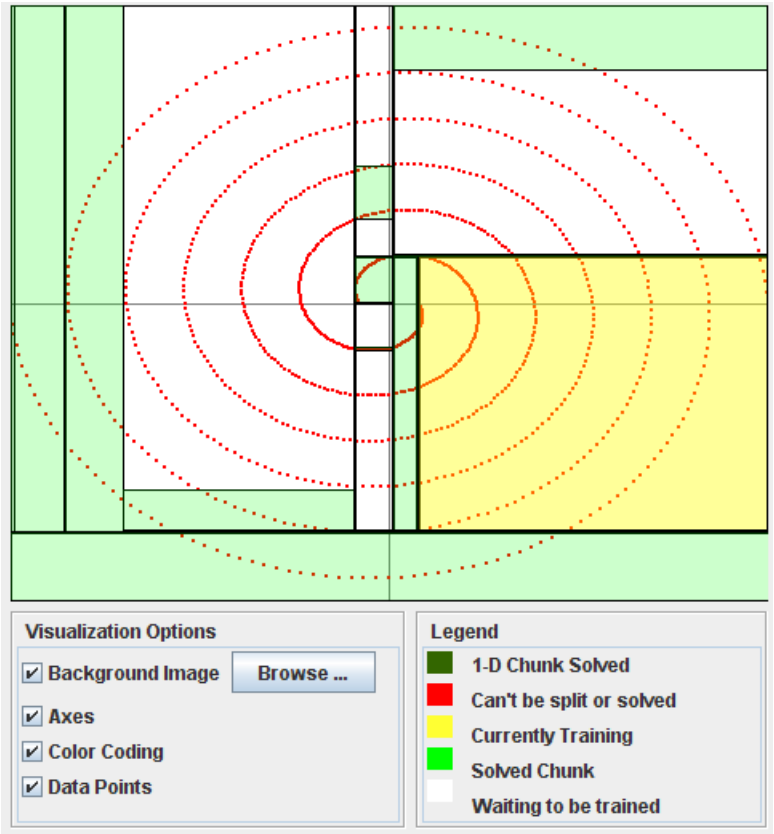

Fig. 4. SSNN Visualization Tool - *visualization pane* during training

Figure 4 shows an example of the visualization pane during training on the two-spiral problem. Since the application is two-dimensional, the entire domain can be illustrated. The red dots represent training points. Green areas have completed training, and there is thus a neural network assigned to each green rectangle. The yellow rectangle is currently training. When training is complete, if it is successful, the entire pane will be covered with green rectangles, and a summary of generalization results on the user-provided test data set also becomes available.

IV. First Results

We expect the full utility of the visualization tool to reveal itself as more problems are added and a greater variety of learning and splitting modes are incorporated. However, even in its early stages it has helped us to find ways of improving the SSNN.

For example, one of the things that we observed during execution is the rather numerous occurrances of small chunks, resulting in a three-way split rather than what might be a more efficient two-way centroid split. This led us to some additional tests to determine whether a larger minimum chunk size would work better than our typical setting (which was 3).

Watching the SSNN proceed, we found it impossible to detect any noticeable pattern to the production of solved chunks. In fact, solved chunks are an arbitrary byproduct of the learning. Backpropagation's goal is to minimize the aggregate sum-squared error – it does nothing to proactively try and find large solved chunks.

Furthermore, we noticed that during training, more often than not, the size of the largest solved chunk actually decreased during the early stages. This is because one way that backpropagation initially reduces sum-squared error is to simply drive the outputs of every training case to 0.5. (assuming output is normalized to 0.0-1.0). Although the number of cases solved would be worse than for the initial random network, the aggregate sum-squared error is better.

This has led us to start exploring other learning methods that can be configured with other goals – such as trying to maximize solved chunk size. For example, one of our early attempts has been to utilize simulated annealing, which can incorporate a fitness function. In this case, fitness is measured not by sum-squared error, but instead maximizing the largest solved chunk size [9]. After incorporating simulated annealing, we noticed a surprising number of times the SSNN failed to solve very small chunks. The visualization tool makes it easy to note the nature of chunks that fail to solve. In this particular

case, chunks that failed to solve were those that involved small ranges of values that happened to be close to 0.5. It turned out we were using an inferior method of scaling in which we scaled the entire data set prior to training. A better method is to scale each chunk independently. After incorporating the improved scaling method, training became much more effective – to the point that the number of chunks required to solve the two-spiral problem has been reduced from around 65 down to 15, and with it we are observing improved generalization. Detailed results will appear in a future paper.

In another application of the visualization tool, Bender and Gordon have used it to identify more effective splitting strategies, also reducing the number of networks generated and improving generalization [10]. The development of their improved *area-based binary splitting* is attributable to insights gained from the visualization tool.

V. Future Work

The SSNN visualization tool has exposed other weaknesses of the SSNN that are being addressed. Clearly, the splitting algorithm currently being used is primitive, working in only one dimension. We are working on more sophisticated multi-dimensional splitting techniques that hopefully can find not only larger chunks, but more accurately reflect the clustering of solutions being produced during training.

One interesting extension that we are developing is a stronger visualization of the functions implemented by the networks in each solved chunk. Figure 5 shows an experimental version of the visualization pane which illustrates each rectangular chunk as an image where the range 0..1 is modeled in greyscale. Although this has not yet been incorporated into our online version, it improves the information conveyed in the pane.

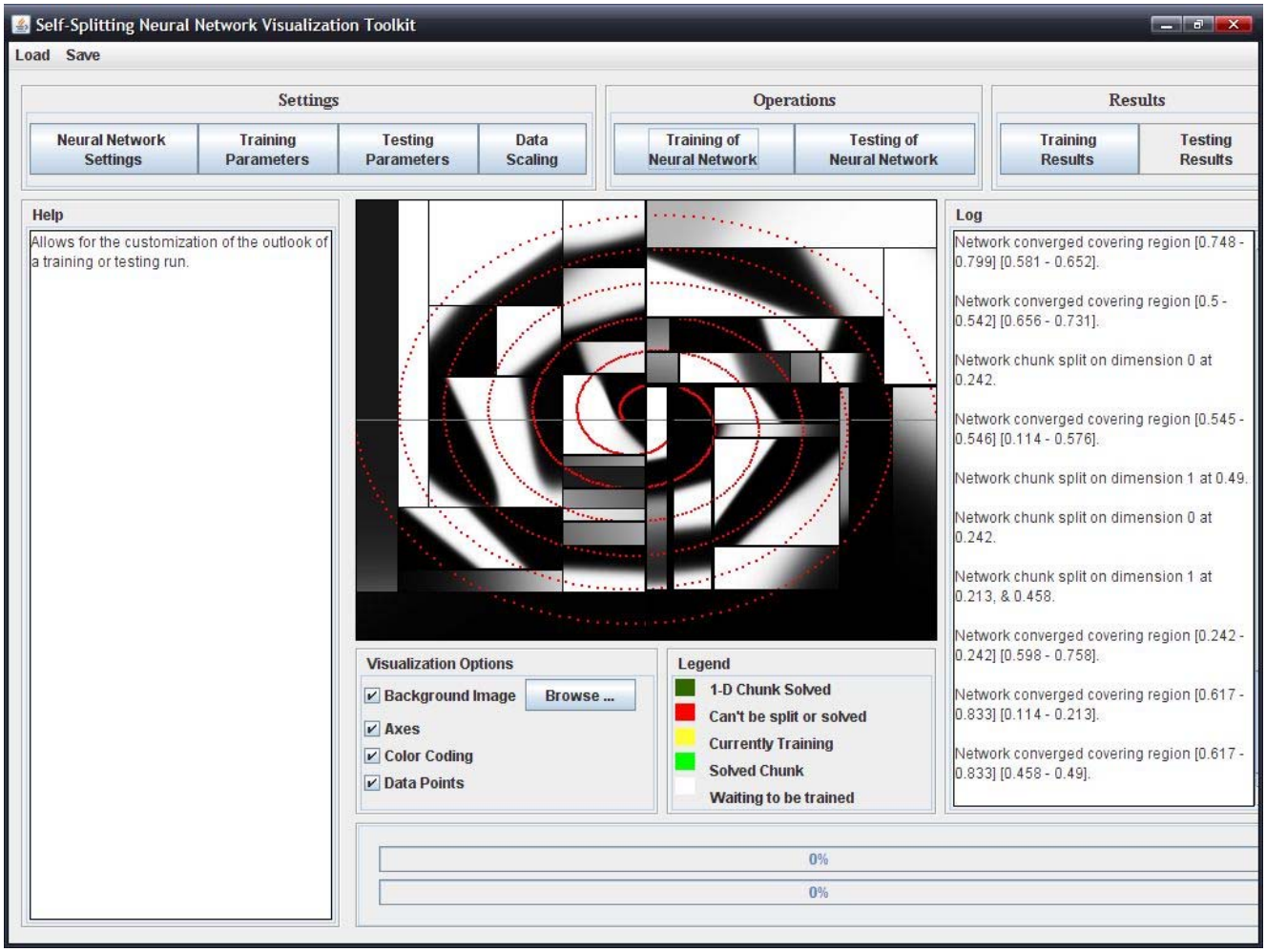

Fig. 5. SSNN Visualization Tool – *greyscale representations of chunk functionality*

We are also pursuing more robust training methods. Although simulated annealing has been effective and has shown the efficacy of fitness-based approaches for the SSNN, other methods for neural network training are believed to be more promising. In particular, Particle Swarm Optimization has been shown highly effective for neural network training [11], and enjoys the same benefits as simulated annealing for our needs.

VI. Conclusions

We described a visualization tool for a self-splitting modular neural network (SSNN), and implemented it as a Java application which can be run from a browser with Java Web Start. The tool is useful for illustrating how the SSNN works, and more importantly how it can be improved. The tool has already pointed us in new and better directions, and we expect that as we implement further improvements, it will become even more useful.

Acknowledgment

The visualization tool described in this paper was developed as part of a capstone team project at California State University, Sacramento. We thank Bob Buckley for accepting the project, and Dr. William Mitchell for his supervisory assistance.

References

[1] R. A. Jacobs and M. I. Jordan, *A Modular Connectionist Architecture for Learning Piecewise Control Strategies*. Proc. of the American Control Conference, pp 343-351, 1991.

[2] S. D. Whitehead, J. Karlsson, and J. Tenenberg., *Learning Multiple Goal Behavior via Task Decomposition and Dynamic Policy Merging.* Robot Learning, MIT Press, 1992.

[3] A. Sharkey, *On Combining Neural Networks*, Connection Science, 8(3/4): 299-314, 1996.

[4] *Combining Artificial Neural Networks – Ensemble and Modular Multi-Net Systems.* ed. A. Sharkey, Springer-Verlag, 1999.

[5] K. Chen, L. Yang, X. Yu, and H. Chi, *A Self-Generating Modular Neural Network Architecture for Supervised Learning*, Neurocomputing 16: 33-48, 1997.

[6] M. Olsen-Darter and V. Gordon, *Vehicle Steering Control Using Modular Neural Networks*, IEEE International Conference on Information Reuse and Integration (IRI), pp 374-379, 2005.

[7] V. Gordon and J. Crouson, *Self-Splitting Modular Neural Network – Domain Partitioning at Boundaries of Trained Regions*, 2008 International Joint Conference on Neural Networks (IJCNN 2008).

[8] M. White, *Two-Spirals Benchmark Data Set Generator*, 26 April 2006: http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu/bench.tgz

[9] V. Gordon, *Neighbor Annealing for Neural Network Training*, 2008 International Joint Conference on Neural Networks (IJCNN 2008).

[10] T. Bender, V. Gordon, and M. Daniels, *Splitting Strategies for Modular Neural Networks*, 2009 International Joint Conference on Neural Networks (IJCNN 2009).

[11] V. Gudise and G. Venayagamoorthy, *Comparison of Particle Swarm Optimization and Backpropagation as Training Algorithms for Neural Networks*, Proceedings of the 2003 IEEE Swarm Intelligence Symposium (SIS'03) pp 110-117.