Model Development and Extraction from Neural Networks Final Report

Brian G. Maddox
Ryanne Dolan

Open-File Report 2005-

U.S. Department of the Interior

U.S. Geological Survey

CONTENTS

ILLUSTRATIONS

KEY WORDS

Neural networks model development extraction

ABSTRACT

Developing mathematical models of physical phenomena can become a complex task. More and more data are being created with improved sensing devices, and more demands are being placed on creating highly accurate models. Models with large numbers of variables can be difficult for a researcher to develop due to the extreme complexity involved. One way of trying to simplify this process is to use a neural network to automatically train from the variables and deliver a mathematical model to the researcher.

INTRODUCTION

One problem with developing mathematical models of physical phenomena is the complexity involved. These models can be comprised of large numbers of variables. While there are techniques to determine which variables are dependent and independent, this can be difficult and time consuming as the number of variables increases. Additionally, a large number of variables might be left that have to be used to create the final mathematical model of the phenomena. Dealing with these variables can make it difficult for a researcher to develop a comprehensive model to be used for modeling and predictive purposes.

To address this issue, it was theorized that a neural network could be used to train itself on datasets from a specific phenomena and then extract a mathematical model based on how the network trained. As neural networks normally are “black boxes”, this would require researching how to change a neural network, and how to create a mathematical model from it. Doing this would require a change to the traditional functionality of a neural network to fully track the data flow. Once this is done, it was theorized that, at a minimum, mathematical step functions could be generated that describe the model that the network trained on.

To test this hypothesis, work was done with data from the Centers of Disease Control to apply the theory to modeling the spread of tularemia, a disease caused by the bacterium *Francisella tularensis*. This organism is a known pathogen and is listed as a possible agent for use in a biological attack. The idea was to use the network to model the spread of the disease to establish a baseline model. Non-natural occurrences of the disease could then be identified and investigated as possible biological threats.

While the project was cut short due to funding issues, this final report will discuss some of the discoveries made, the work that has been done, and future work that is needed to complete the project. It is hoped that this project will someday be resurrected to complete this research and evaluate the possible impact on the development of models for physical phenomena.

BACKGROUND

The study of artificial intelligence uses computer technology to model thought, knowledge, and intelligent behavior. The traditional models of intelligence follow two basic paradigms, *symbolism* and *connectionism*. The former approach involves manipulating symbols following a set of rules, and the latter involves forming connections of various strengths between nodes. The trend in the past few decades has been toward connectionism, as neurological research suggests that connectionist systems more closely mimic the human brain.

The connectionist paradigm comes from two fundamental assumptions: that the best way to achieve human-like intelligence is by mimicking the living brain, and that the brain consists of millions of highly interconnected processing units called *neurons*. By studying the interplay of neurons in living brains, scientists seeking *strong artificial intelligence* hope to construct artificial systems that exhibit behavior similar to that of a human brain.

An early product of this research was the *neural network*, a type of artificial neuron system. Various neural network models have emerged since the conception of the archetypal perceptron networks pioneered by Hebb (1949). In his book *The Organization of Behavior*, he proposed Hebb's rule which states "*When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*" (Hebb 1949). In other words, the connection between two neurons that fire together is strengthened, and this strengthening is one of the core operations of learning and memory. This biological understanding helped artificial neuron researchers develop the perceptron, an artificial neuron that could learn by the strengths of several weighted inputs. All neural networks fundamentally are similar in form and function.

The individual processing units that make up a neural network are called *nodes*. A neural network consists of a weighted, directional graph representing a set of nodes and the connections between them. Unlike regular computational systems that are given a predetermined set of instructions, neural networks cannot be preprogrammed to solve a particular task. Instead, they must be trained on sample data that are chosen to represent a particular problem. This training strengthens pathways between the various nodes inside the network. The more data that follow a certain pattern, the more certain pathways will be strengthened. Once trained, a neural network operates as a "black box" function that accepts inputs comparable to the sample data, and returns outputs that will be similar for corresponding inputs.

There are two ways to train a neural network. Data need to be converted to numeric form with either method before it can be input into the network. In supervised learning, the user must first collect the training data to be used. These data contain input variables with known outputs. These are run through the network, and it then trains itself on the relationship between the inputs and outputs. The other method, known as unsupervised learning, uses only the input variables. This type of network is a bit different from a supervised one in that it attempts to sort out the structure of the data on its own. Of the two, supervised learning is the most common use of a neural network.

The ability to associate an input set with a learned output set makes neural networks especially adept at control applications such as robotics. For example, a neural network

can be mapped with sensors as inputs and motors as outputs. A slightly modified neural network can be trained on a sequence of data, such as the stock market, and used as a predictor.

The strengths of using neural networks are also weaknesses in some ways. Neural networks are used to create models of large and complex functions with large numbers of variables where traditional linear modeling would fail. Linear models mainly apply where a relationship is already known and the function is somewhat less complex and can be optimized. A neural network is used where there is a suspected relationship between inputs and outputs, but that relationship is not known and must be deduced inside the network. The primary downside to using a neural network is that the network learns by example, and the model it learns is not in any human usable format. The input variables are converted into signals inside the network that then go through the various pathways to create the output.

The neural network is often considered a “black box” in that it will produce an output, but provides no explanation for the result and no guarantee that the output is correct. Though neural networks are adept at modeling patterns in input sets, extracting any information about the pattern is nearly impossible. For example, a neural network designed to predict the weather would offer no explanation for its forecast. Furthermore, when a prediction is incorrect, it is difficult to fix the inaccuracy, except by further training the neural network. Neural networks are most often employed in systems involving innumerable input sets, so testing a network to determine its accuracy is usually impossible.

The goal of this project was to resolve these issues by extracting more information from a neural network than is provided by its outputs. Conceivably, an algorithm could examine a neural network and extract the function encoded in the network. This function should return the same output vector as the neural network when both are presented with the same input vector. The function should be easier to implement than the neural network encoding, and should provide a model for analyzing the patterns within the input sets. This information could then be used to generate mathematical relationships to model the physical phenomena. While these may not be linear functions, they should at least allow researchers to better understand the model generated by the neural network.

METHOD AND TESTING

Development Tools

To keep new development to a minimum, the Stuttgart Neural Network Simulator (SNNS) was chosen as the base platform. This software was developed at the Institute for Parallel and Distributed High Performance Systems at the University of Stuttgart for

simulating neural networks and has been in use for many years. Over time it has built up a following and has had the code reviewed numerous times. This system provided a stable base for the research work.

The software development platform was Mandrake Linux using the gcc compiler suite. Some initial work had to be done to get SNNS working properly on the Linux distribution. The software was written in an older style of the C programming language and contained a few constructs that are no longer considered valid by modern compilers.

Data Collection

The first task was to provide climate data for Arkansas and Missouri to determine possible causes for the spread of tularemia. The data needed were temperature, rain, and snow data from 1990 to 2000. After receiving the data from NOAA, it was necessary to convert the data from exact daily readings to a monthly average. This average was necessary on a county-by-county basis as some counties had multiple NOAA reporting stations, while others had none. County level tularemia data were also the finest granularity in which the tularemia data were available.

Modifications to SNNS

The SNNS neural network model was first modified to send data packets instead of digital signals. These data packets recorded various pieces of information as they run from the inputs to the outputs of the neural network. For each current node, the data packets would record the current node, what other nodes were connected to the current node, the firing thresholds of each node based on connections to other nodes, and the signal levels of each connected node when the current node fires. While these packets added considerable overhead, they also allowed an in-depth study of the network while it is in operation. These packets also helped to connect outputs to the inputs, so that given input levels would be known when the network generated an output.

As SNNS provided a stable and well tested neural network system, work began by making modifications to SNNS to test some of the initial theories about extracting information from the network. First, however, the base version of SNNS had to be regression tested to ensure that porting to Mandrake Linux did not result in any software errors. For the first test, a simple neural network example, an OR gate, was created and then used pre-existing files to train and test it. The results were as expected, so SNNS was working with small networks.

Next, a larger network was tested. This network had a two-dimensional array of inputs designed to represent each character of the alphabet, and output neurons for each letter.

Following an example file, the network was created, and test files were developed. After training, the network worked as expected and the output matched the known outputs. Since the first two tests were recreating previous examples, it was then decided to try a modified example. To do this, the numbers zero through nine were added to the alphabet network and the corresponding test cases were written. Training went well, with the resulting network distinguishing between the alphabet and the numbers. Since all three tests worked without a problem, SNNS was working correctly.

After verifying that SNNS was functioning correctly under Linux, work then began to modify it for testing purposes. After studying SNNS, the areas of the source were identified that needed to be modified: the low-level functions of SNNS – Test functions, Learning functions, and Update functions. This was actually quite difficult as SNNS is a very large code base and does not tend to follow modern software engineering practices. The sheer number of functions that would have to be modified, however, forced a look at other methods to modify the code. Instead of using the low level functions, the function that calls them was modified. This greatly reduced the amount of code that would have to be written, as well as having a lower overall impact on the operation of SNNS.

In implementing the data packet system, a linked list structure was written as well as the structure that stores the packet data. The structures and the related functions are in the source files r_tracking.h and r_tracking.c. Also, a global variable called master_list is defined in r_tracking.h. From there, the SNNS files were modified. A new function called create_list was added to kernel.c, which is called in kr_callNetworkFunctionSTD. Also, the Unit structure, defined in kr_typ.h, was modified by adding unit_number, which stores the ID number of the unit.

All these files function together as follows:

1. The network is modified or updated in some way, whether through learning, testing, or adding/removing units or links.
2. SNNS calls the current update/learning/test function that is needed.
3. After the function finishes updating all the activations and outputs on the units in the network, create_list is called.
4. create_list erases the current list, and rebuilds it using the new data. This list can then be displayed in the packet list window mentioned below, as well as saved to disk.
5. Finally, when the program exits, the list is erased completely.

Note that the list is created after the network is updated. Originally, it was thought that the list needed to be built as the network was being updated, which is why originally it seemed necessary to edit the lower level functions.

Once the packet tracking was working, more functionality was added to the SNNS graphical user interface, since a way to view the packet list was needed. A new button was added to the control window that brings up the packet list window. The packet list window has a save button that allows the packet list to be saved to disk for further analysis. There are still some issues with the packet list window, in that it cannot display a list of packets over a certain size correctly. This is due to the nature of the X Window System widget tool set used by SNNS.

After this work was complete, Phase II of the project began. This phase was to research ways to automate data input into SNNS. While SNNS, like other neural networks, already supports unsupervised training, unsupervised training alone is still a difficult process and would require users to have some knowledge of how neural networks work. Since one of the goals of this project was to find ways to automate the process and make it easier for non-computer scientists, Phase II involved trying to find easier ways to do unsupervised training.

For Phase II of the project, we only need the algorithms that deal with unsupervised training. There are two popular algorithms, Kohonen and Art2, and they are already implemented in SNNS. However, the software does not automatically output the result of the clusters. SNNS had to be modified to extract the data that is necessary for analysis.

To train data using SNNS, it has to be fed a “pattern file” ending in .pat. There is an example of the pattern file in “SNNSv4.1/examples/” directory. There are two kinds of examples, one for supervised training and one for unsupervised training. If we look for unsupervised training, the example is some_cute.pat with input but no output. The test pattern file has to be generated in exactly the same format as some_cute.pat, otherwise it will not work. Also, in “SNNSv4.1/examples/” there are files ending with net which are the “network” files. On SNNS Manager Panel there is a button named *BIGNET*. This allows selection of the preferred unsupervised training method: art1, art2, kohonen, and so on. This selection means that the network file does not have to be manually generated and loaded into SNNS. To train the data, the pattern file has to be loaded into the network (land cover and climate data converted into pattern file format). After the network has been created, pattern file loaded, and selected appropriate parameters and functions, we are ready to train our data.

The software was modified to output data in a specific file named “winner_list.txt”. This file contains a list of winners that are learned by some patterns. For the ART2 method, patterns learned from the same winner have similar characteristics and so are considered to belong to one group (cluster). For the Kohonen method, not only do the patterns that are learned from the same winner have similar characteristics, but the patterns that are learned from the winners around the neighborhood also have similar characteristics. The

DISPLAY button in SNNS Manager Panel allows us to view the screen display of each training method.

After the data are clustered, we then examine their relationship to understand what factors affect the spread of tularemia. The result data from Phase I can then be used for supervised training. For this, we can use back_propagation. This time the patterns contain both input and output. Input is land cover / weather data, and output in this case is the number of tularemia cases. After several supervised trainings, the network will form patterns and construct a model. The network file then has to be saved so that it can then be used for the next step. The saved network file is loaded and fed in the data (pattern file) with patterns that contain only "inputs" where we want to predict the output. The important step is to cluster data and find attributes (water, grass, etc.) that play a role in spreading tularemia.

SNNS was also modified to include a scripting engine so that the neural network interface could be controlled from a text file. Over fifty SNNS interface commands were implemented in the scripting language, allowing the user to automate the creation, training, and testing of neural networks. This modification greatly expedited development because functionality could be implemented within the scripting system rather than at the GUI and kernel level within the SNNS source. Also, algorithms under development could be tested much more efficiently, since a script could be written to create, test, and destroy thousands of neural networks without any interaction from the user. The scripting system also provides the potential to simplify end-user tasks by automating the training of networks with minimal user interaction.

The Lua scripting language was chosen as the base language for the SNNS scripting system after a speed and usability comparison between many different candidates, including Perl, Ruby, and Python. Lua was judged to be the fastest and easiest to use because of its efficient and simple open-source interpreter written in C. Lua is based on Pascal, making it easy to learn, even for novice programmers.

Using the scripting system as a development framework, a novel genetic algorithm was implemented to automatically produce neural networks with an optimal topography for a given problem. This new algorithm is an improvement on the "cell division and migration" method, which emulates the way natural nervous systems grow during the early development of an organism (Cangelosi and others, 1994). One weakness in the original cell division and migration algorithm is that it stops growing only after a predetermined number of artificial neurons have spawned. The new algorithm implemented in SNNS resolves this issue by allowing developing neurons to alter their virtual environment via artificial chemical signals which tell neighboring neurons when to migrate and reproduce. In this way, the algorithm can converge on a stable topography automatically. In other words, the size, density, and topography of the generated network is determined by the

algorithm without any information from the user.

The algorithm attempts to simulate the biological process of nervous system growth and development. The developing neurons migrate and reproduce on a large grid such that each cell can contain at most one neuron. Neurons can move, reproduce, or release chemical signals into adjacent cells. Each neuron maintains its own strand of artificial DNA, which encodes when the neuron will move or release chemical signals. Some of these chemicals act as artificial hormones that dictate when a neuron reproduces. By consuming energy and releasing reproductive hormones into the proximate environment, individual neurons can communicate and affect the overall growth of the network. This novel algorithm more closely simulates natural neuron growth and produces an incredibly diverse pool of convergent and stable networks. Furthermore, the algorithm will always produce the same network from a given strand of DNA and so provides an excellent method for encoding a neural network in a compact binary representation.

The following images represent a few networks evolved using the algorithm. Notice that the algorithm is capable of producing feed-forward multilayer networks as well as densely interconnected recurrent networks, depending on the problem set and fitness evaluation function.

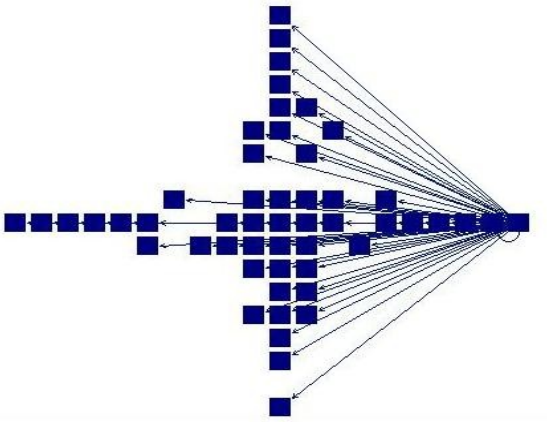

Figure 1: Two Dimensional Representation

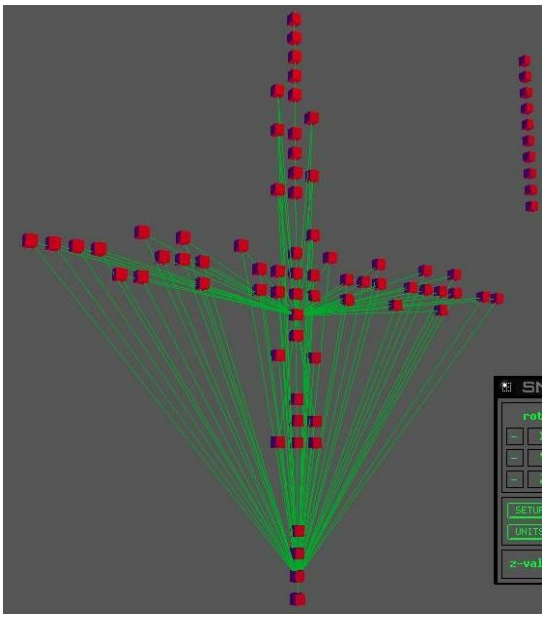

Figure 2: Three Dimensional Representation

The genetic algorithm can be modified via the scripting system to evolve feed-forward or recurrent networks by enabling or disabling a built-in pruning algorithm. This allows a single algorithm to grow optimized networks for most network architectures supported by SNNS. While comparable genetic algorithms can take many hours to produce results,

the algorithm added to SNNS takes only a few seconds to evolve a population of several hundred networks. Once a network has been evolved by the genetic algorithm, it can then be trained, tested, and stored either manually or via the scripting language. By programming a script to use the genetic algorithm and SNNS interface, thousands of neural networks can be evolved, trained, tested, and analyzed autonomously.

Results

The following graphs illustrate the results from a sample run of the Kohonen unsupervised clustering program. The input set comprised geographical data of various counties that was normalized and clustered automatically by the algorithm. Several groups were returned, two of which are compared below. Note that the algorithm made an accurate distinction between these two groups despite their impressive similarity.

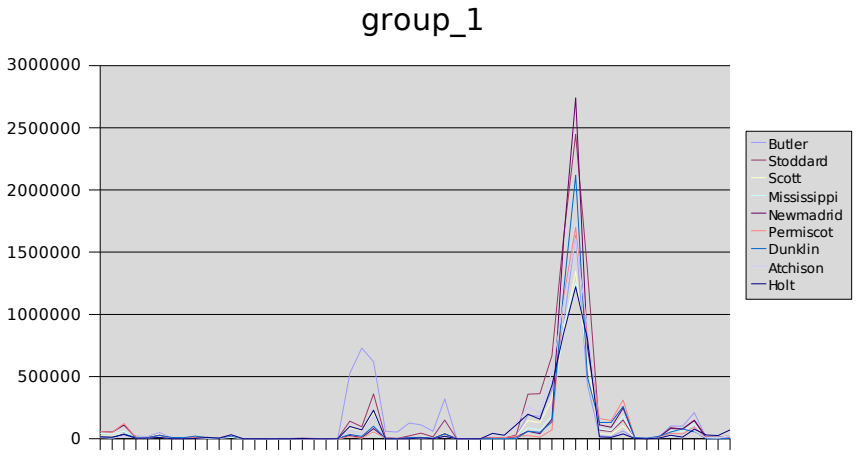

Figure 3: Sample climate and tularemia clustering.

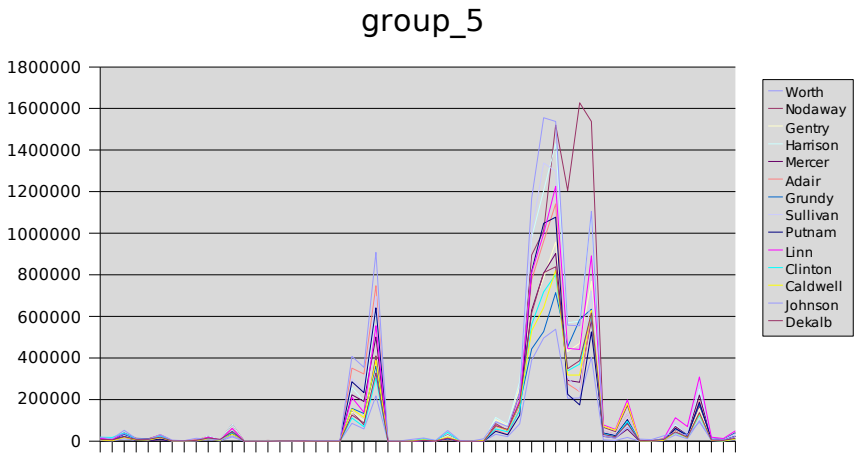

Figure 4: Second group clustering.

While these results are encouraging, they do not provide much insight into the inherent relationships present in the data set. These data must be further processed to discover any underlying patterns and rules. To do this automatically, a script could be written that would evolve an optimized neural network and train it on the clustered data. Finally, the relationships inherent in the network would be extracted using the packet tracing system or some other data extraction algorithm.

DISCUSSION

Data collection for this project posed an interesting problem. Climate data from the NOAA, Land Cover data from the USGS, and county-based disease data from the CDC were easy to obtain. The spatial resolution of the disease data proved to be a problem when making associations between variables. Privacy laws, however, prevented the use of higher-resolution disease data.

Towards the end of this project it was decided to switch the data to modeling land use change for another project in the Geography discipline. These data were more readily available as it was being created by USGS scientists, and a member of the Land Cover team was in the same section as the principal investigator. This project is model agnostic in that it does not matter what is being modeled to develop the theory as “a model is a model”.

During the research and development of this project, several alternative neural network data extraction methods were studied, and the feasibility of each method was considered. In addition to the packet tracking idea, one such method was the discretized interpretable multilayer perceptron (Bologna). This modified neural network was designed to simplify the data extraction process by ensuring that the domain is divided into discrete regions

during the training process. This allows the algorithm to extract *if..then* implications that can be used in lieu of the network. Bologna showed that his technique proved effective for some data sets, but did not necessarily discover useful information about the domain.

If Bologna's modified neural network was implemented in SNNS, then the tools developed during this project would become more powerful. Scripts could be written to genetically evolve a discretized network, which could then be analyzed using the packet tracking system. Raw geographic data could be normalized and clustered automatically and fed to the evolved discretized network. Bologna's algorithm could be used to extract interpretable *if..then* implications that realize the inherent rules governing the data set.

This approach raises some important questions, however: Are simplified rules-based systems better models of a natural phenomenon then a black-box neural network? Is it practical to expect well-defined rules to emerge from a neural network trained on a large and noisy data set?

The first question arises from the fact that the rules extracted from a neural network are usually simplified to eliminate irrelevant implications. In fact, an important part of Bologna's network is a pruning algorithm. Bologna has shown that even after pruning, his algorithm can produce rules that perfectly match the results of the trained discretized network, but whether or not these rules are as accurate as a more traditional network remains to be seen. It is possible that the discrete nature of Bologna's networks makes them less efficient at learning large and complex geographic data sets, and it is likely that the algorithm would produce too many rules for human interpretation if given an overly-complicated data set.

The second question involves large and imperfect data sets where an underlying pattern may or may not actually exist. Because the data set is not usually well understood, it is nearly impossible to determine the accuracy of the rules extracted from a trained neural network without testing the rules in a real-world environment. This assumes that such underlying rules truly exist in the data set. For situations where little or no patterns can be found, the algorithm should have a mechanism for rating its own results, and for giving up when no results can be determined. Until these mechanisms are in place, it is probably not practical to expect a discretized network to provide sensible interpretations from a real-world data set.

FUTURE WORK

This project was closed at the end of FY2005 due to budget constraints, but there are several areas of work that could be continued in the future. The first area is in combining

genetic algorithms and neural networks. Some work is currently(2005) being done to try to get as much research completed as possible before the projects ends. Another method for combining neural networks and genetic algorithms could be achieved by having the genetic algorithm actually modify the individual nodes within a network while it is running. This would be extremely difficult, but it might provide a better optimization of the neural network. It might also make totally unsupervised training easier as the genetic algorithm could continually adjust until outputs match known values.

Converting the output from the modified network into mathematical functions could also be explored. Initially, work should focus on converting the data into step functions, as this would be easier and would provide a better understanding of how to go from neural pathways to functions. Once this is done, the step functions could be converted into complex functions instead of simple linear ones.

Another task for future work could be looking at different neural network packages in existence. When this project first began, SNNS was the only full featured package that could be used for research purposes. Different offerings have recently (2005) become available that might make a better fit for this work. SNNS follows a “kitchen sink” approach in that it includes everything one might need while using a neural network. SNNS is also “old code” that is becoming increasingly harder to maintain as new generations of programmers are not familiar with older methods of writing code. A smaller more targeted neural network system might make a better choice for this research in the future as it would be easier to maintain and debug.

CONCLUSION

The primary focus of this project was to find ways to help scientists develop complex models of physical phenomena via technology. Modern need and problems are creating a demand for more complex and exact models of various phenomena. As our understanding of the world grows, so does the number of variables that are considered when developing a model. Automating the process can help researchers generate these models much more easily than if done manually. Automating this process can also help to speed model generation. This can be significant, as a phenomena might need to be quickly modeled so that disaster planning and mitigation can be performed quickly.

Modeling the physical world requires a lot of data, and these data usually need to be modified so that they will work together in a neural network. Experience from this project has shown that to be accurate, data need to match the lowest common denominators of items such as resolution, scale, accuracy, and so on. This is important as it will help ensure that the output from the network will be accurate and make sense. Higher resolution versions of one variable might skew results when combined with lower resolution versions of another. This work can actually be fairly difficult, as it requires a lot

of reformatting and averaging of input variables.

Neural networks can be modified so that data can be more easily extracted from them. These modifications include tracking what data travel to each node, firing levels of the various nodes, connection between nodes, and so on. This allows data to be fully tracked inside a network instead of the traditional “black box” model. Ideally, this information can then be used to automate the generation of a mathematical model that describes the relationships between various variables and what variables actually contribute to the final result.

The most important part of this work is in automating the training process of the neural network. While there are some preexisting methods of self training, new techniques such as genetic algorithms must be added to allow users who are not technically trained in using neural networks to benefit from them. This training allows users to input large sets of variables and tune the operation of the network so that the output matches known values. Computer science techniques such as genetic algorithms can be used to find the global best network that matches said outputs without much user intervention. This is an important step to bring the power of such computational techniques to non-computer science researchers.

The work done for this project resulted in several useful extensions to SNNS. A packet tracking system was implemented, enabling researchers to trace the flow of information through a neural network in real-time. The Kohonen algorithm was extended with support utilities for automatically clustering data sets into groups for further study. A novel genetic algorithm was implemented to grow optimized neural networks. SNNS was extended with an embedded scripting language for automation of tasks. With these tools in place, the goal of automatic data extraction from trained neural networks is closer than ever before.

REFERENCES

Bologna,Guido. "Rule Extraction from a Multi Layer Perceptron with Staircase Activation Functions." IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN'00) Volume 3. 2000.

Cangelosi, Angelo, Domenico Parisi and Stefano Nolfi. Cell Division and Migration in a 'Genotype' for Neural Networks. Network. 1994.

Connected Graph. <http://dictionary.reference.com/search?q=connected%20graph>. Dictionary.com. 2005.

Hebb, D.O. *The Organization of Behavior*. Wiley, New York. 1949.

Stergiou, Christos and Dimitrios Siganos. *Neural Networks*. <http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html>. Imperial College of London Department of Computing. 2005.

University of Tübingen. *SNNS - Stuttgart Neural Network Simulator.* <*http://www-ra.informatik.uni-tuebingen.de/SNNS/*>. University of Tübingen. 2005.