Pattern Memory and Analysis in Bat-Inspired Echolocation Systems

Patrick B. Ellis, Tarek Massoud *Student Member, IEEE*, Timothy Horiuchi, *Member, IEEE*

***Abstract*—Bats are well known for navigating the world using a sense that is completely foreign to human beings – echolocation. Investigating echolocation may provide rich opportunities to influence the technologies of today and the future. We are using a sonar-based system modeled on bat echolocation to demonstrate a pattern recognition neural network on a field-programmable gate array (FPGA) board. The system learns to use sonar signals to identify several specific objects presented at different distances. The object identification provided by this system will be used in a neural model of animal navigation currently being investigated.**

***Index Terms*—Echolocation, Neural Networks, Interfacing, VHDL**

I. Introduction

NATURE often finds elegant solutions to the problems scientists and engineers face in building artificial neural systems. The use of acoustic echoes to detect objects (i.e., sonar) is one remarkable example found in the echolocating bat. By observing the temporal pattern of echoes made by different objects, an object recognition system can be designed for detecting landmarks in the environment [4].

II. Background

A. The Set-Up Configuration

In this project, a set of simple objects were created that produce distinguishable patterns of echoes. Each object is a connected set of plastic pipes standing perpendicular to the floor. The recognition system creates a template for the patterns these objects create from reflected sonar waves, and can therefore answer which set of plastic pipes is being viewed. "Object A" is a single pole, "Object B" is a group of two poles, "Object C" is a group of three poles in a line, and "Object D" is a group of four poles in a line. The poles are always presented to the system such that the system as a "line of sight" to each pole.

Manuscript received August 2, 2010. This material is based upon work supported by the National Science Foundation under Grant No. 0755224..

P. B. Ellis is with the Department of Electrical and Computer Engineering, Bradley University, Peoria, IL 61606 USA (e-mail: pellis@mail.bradley.edu).

T. Massoud is with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742 USA (e-mail: tarek.massoud@gmail.com).

T. Horiuchi is with the Deptment of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742 USA (e-mail: timmer@umd.edu).

Interpreting an echo can be difficult – many complex echoes can arise from even the simplest situations. For instance, sound waves can bounce any number of ways between the poles before being detected by a receiver, making background echoes and close-proximity reflections difficult to differentiate. Such variability in the echoes makes it difficult to predict the pattern to be detected. A useful general method of addressing this variability is by using an artificial neural network.

B. The Neural Network

A neural network adds a dimension of flexibility that allows patterns that have complex and numerous relationships between its variables to be recognized [3]. In the interpretation of sound amplitude patterns, datasets can look completely different when looking at the same object at different angles, placing the object in a different environment, or administering minute changes like placing the object farther away. By *learning* the distinguishing features of the different objects (presented at different distances) the neural network can provide robust object recognition. A neural network can offer a comparatively simpler, faster implementation. Most importantly, it discovers the complex relationships between the variables through a procedure called "training."

The neural network is a single layer, feed-forward network that uses "supervised learning" [2]. Figure 1 depicts the network graphically.

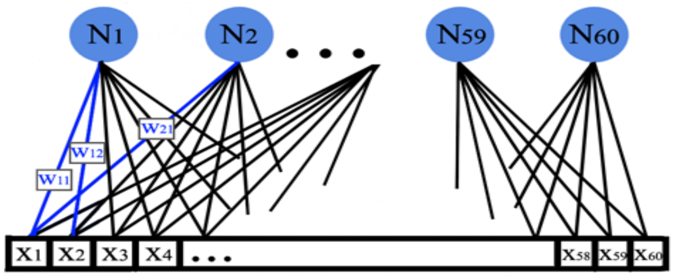

Fig. 1 The Neural Network. The blue circles are the 60 neurons (Ni), the the inputs (Xj) are multiplied by the weight matrix (Wij).

There are 60 neurons in total, organized as four groups of 15 neurons. Each group is detects one of the four objects across a span of 140.97 centimeters. Object A is detected by the first 15 neurons, Object B is detected by the next 15 neurons, and so on. Each neuron is responsible for detecting its assigned object within a specific (9.398 centimeter) detection zone.

The training requires the acquisition of data. For the 15 neuron zones 3 sonar samples were taken for each of the objects for a total of 180 datasets taken. Care was taken to be sure that the samples were picked randomly within each neuron zone. Randomness adds to the effectiveness of the network by enhancing the robustness of detection.

The neural network was trained using a supervised learning rule, meaning that the desired output pattern for the network was explicitly provided [3]. 85% of the data taken were used as training data and the remaining 15% were left as testing data ("validation set") for when the training is done. The training "learns" the complex relationships of variables when discovering the patterns of the objects in the sonar data.

The learning algorithm is called the Perceptron Learning Rule [2]. The algorithm only works if the solution the network is searching for is linearly separable, meaning there must exist a line, plane, or hyperplane that separates the samples of the different objects [3]. A simple example would be the two input system shown in Figure 2. The two colors represent the two categories to be differentiated. The original line is the randomly picked line that separates the two objects. Through adjustments to the weights and the bias, the line shifts to where it divides the two categories. This visualization is simplistic but can extend into much higher numbers of dimensions. The basis of this algorithm lies in the fact that the coefficients of the terms in the equations separating the objects directly correlate with the weights in the Weight Matrix [2].

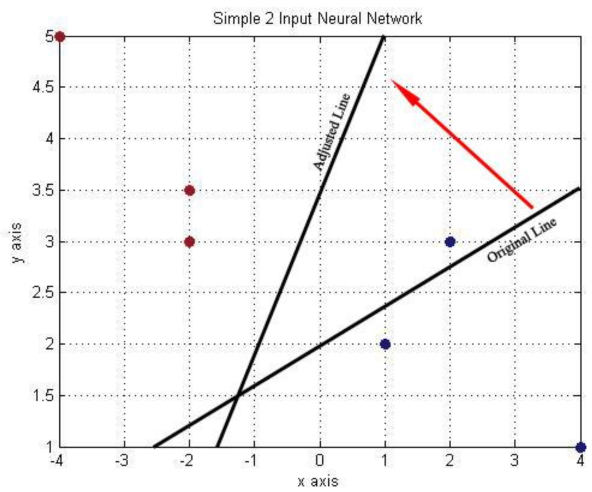

Fig 2. Simple Example: Two Input Neural Network. There are two classes of data – red and blue. The Perceptron Learning Rule moves the dividing line to a location in the two-dimensional space where it separates the two classes.

Specifically, the training proceeds by first creating a random weight and bias matrix and picking one of the training inputs and applying it to the network. The network output is then compared to the correct output and small adjustments (Equation 1.1) are made to the weights and biases to reduce the difference of the network output and desired output. If this is done enough times and a solution exists, the weights and biases will converge [2]. Gamma is a user-defined learning rate that controls the magnitude of changes made to the weight matrix and bias matrix with each input example.

The network output, or answer matrix, is acquired by first using Equation 1.2 and then taking the hyperbolic tangent of that answer. This saturates the neuron outputs at around -1 or +1, and as more training is done the outputs of the neurons should converge towards these numbers.

$$W_{new} = W_{old} + \Gamma \times X \quad (1.1)$$

$$M_{answer} = M_{weights} \times M_{input} + M_{bias} \quad (1.2)$$

A neuron fires in the following manner: (Figure 1) each neuron reads the normalized values of the input matrix (X_1, X_2, …) that comes from the sonar and then multiplies this matrix by weight matrix (W_{1n}, W_{2n}, …) that has been previously trained. The result is a 60x1 matrix whose values each correspond to one of the neurons. This result is then added with a bias matrix of the same dimension, also trained with the weight matrix. The position of the highest value in the answer matrix is the neuron that "wins," which then indicates the object being viewed. The bias matrix is an optional feature of adding preciseness and maneuverability in the learning of the pattern classes. M_{answer} is a 60x1 matrix, M_{weight} is a 60x256 matrix, M_{input} is a 256x1 matrix, and M_{bias} is a 60x1 matrix. The '256' number correlates to the number of amplitude measurements the sonar takes for each input vector of amplitudes taken.

III. IMPLEMENTATION ONTO MATLAB®

MATLAB® was used to communicate with the sonar (triggering and data acquisition) and graphed the incoming amplitude signal of the sonar.

The first task before writing the neural network was to determine whether the solution the network was searching for was linearly separable. To reiterate, for this single layer neural network to work there must exist a hyperplane that separates the samples of the different objects. In two or three dimensional problems this is easily portrayed, but in higher dimensions it poses a tricky problem.

Figure 3 portrays how it was solved. A program in MATLAB® was written profiling three neurons' outputted power in three different ranges of equal length within the 140.97 centimeter range. The dots represent the data taken and the power of that neuron in a particular range. Clearly there exist planes that can separate the three clumps of data. This then indicated that a solution could be found.

The neural network code calculates all the neuron outputs and the highest value found indicates the neuron that "fires," thereby telling

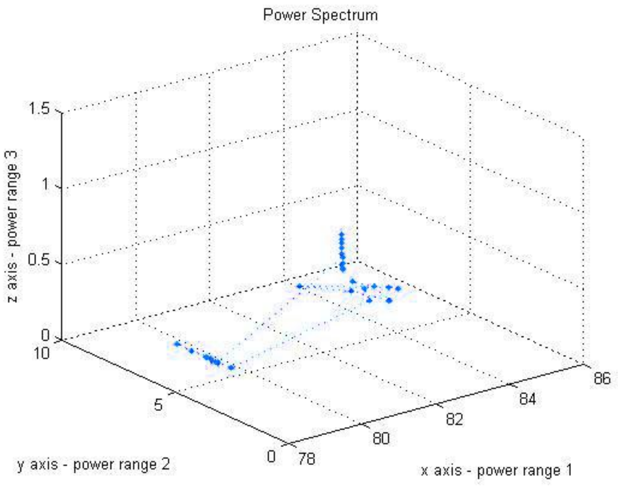

Fig. 3 Linear Separability. The 3 axis represent 3 different power ranges and the 3 clumps of data represent the 3 different neurons tested. The ability to put planes to separate the clumps makes this neural network's solution a linear separable one.

the system which object is present. The system was tested after 1 million iterations of training on the reserved 15% of the data ("validation set") and live readings from the sonar itself. Successful identification rates were around 75%. Figure 4 shows a sample screen output from MATLAB detailing the neuron outputs and the detected object. It can clearly be seen that Object A's first neuron is "firing."

Further improvements for better identification were postponed to allow time to implement the neural network on an FPGA.

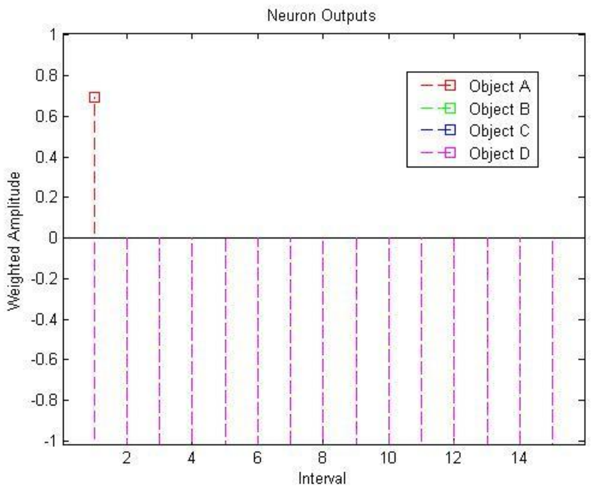

Fig. 4 MATLAB Screen for Object Identification. The plot shows all of the Objects neurons clearly in the negative except for Object A Neuron 1 which is "firing" to around 0.7.

The learned weight matrix from the sonar data was is shown in Figure 5. It can be seen that as the number of poles increases, the width of the positive values increase with the negative values surrounding them at all times. The increasing distance is indicative of the given neuron's allocation to a particular range. Further iterations and tweaking of the learning rates of both the weight and bias matrices would lean the positive values more towards 1 and the negative values to -1.

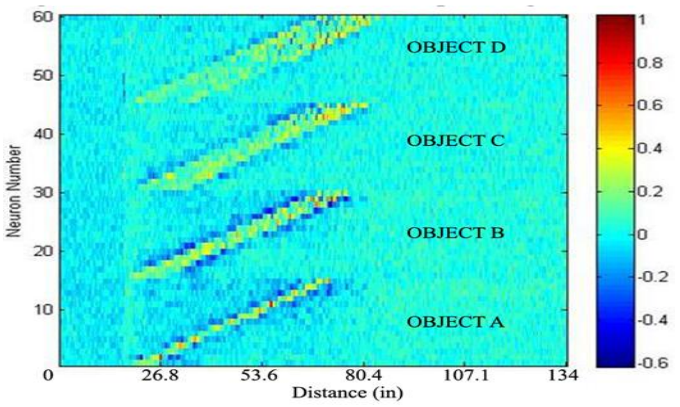

Fig. 5 Colormap of the computed weight matrix. The rows of the weight matrix correspond to neurons covering a range of distances for each of the four objects.

IV. Implementation onto an FPGA

A. *Choosing the FPGA*

Neural network literature notes an interesting connection between parallel processing and the neurons in the human brain. This system can be realized as parallel processing of the 60 neurons acting as small processors performing the same simple calculations. The human brain can be seen as "a parallel system of about 10^{11} processors" [2]. This propensity for parallel computing lends a logical intuition to implement the neural network on an FPGA.

The FPGA chosen was the Xilinx Spartan-3E XEM3005. It comes with a USB 2.0 interface allowing fast FPGA-PC communication as well as the software FrontPanel™, which aids in interfacing and provides controllability of the design.

Figure 6 shows the desired system diagram. Initialization of the system would be the loading of the weight and bias matrix calculated in the MATLAB® neural network to the FPGA. The sonar data would then be collected, normalized, and sent to the FPGA by MATLAB® when desired. The FPGA would then perform the calculations and perform real-time object detection.

XILINX is the compiler for the VHDL program written and FrontPanel™ is the software that allows communication to implement or access the program design on the FPGA.

B. *FrontPanel™*

FrontPanel™ comes with a variety of features that need to be accessed in order for communication with the XEM3005 FPGA to commence.

One FrontPanel™ feature are "FrontPanel HDL" modules that are designed to be put within the FPGA hardware that

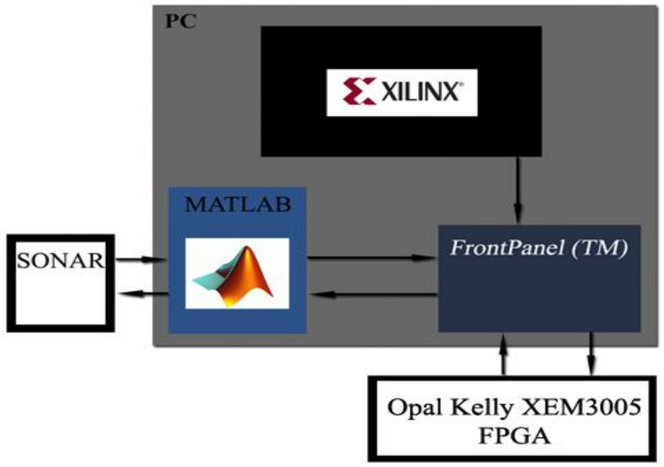

Fig. 6 Interfacing Diagram. Communication to the board must go through FrontPanel. Xilinx is the complier and MATLAB® is the API that controls the sonar.

allows the communication with the FPGA. There are entities within the HDL called "HDL Endpoints" that connect FrontPanel™ components to signals within the design and function as external pins [5]. One of these endpoints is called a "wire," which serves the purpose of transferring a signal state into or out of the design [5].

"FrontPanel API" includes basic libraries that are provided to give a programmer's interface that ultimately allows certain PC applications access to FrontPanel™, and therefore the FPGA. MATLAB® functions as the API [5].

In this system the requirement of a header file, library file, and an extension file provided by FrontPanel must be placed in the directory of the MATLAB® m-file calling the .bit file produced by the Xilinx Compiler. The Xilinx compiler must have okWireIn.ngc, okWireOut.ngc, as well as the okHostInterfaceCore.ngc included in the directory in which the .vhdl file is being saved. Without these included, the compilation of the code will be successful but the mapping and communication to the FPGA will not.

C. *The Transfer of Matrix Values from MATLAB® to FPGA*

Preliminary programs were written as variants off of a sample program "first" given by FrontPanel™ that tested simple functionalities such as signed numbers, fractional binary numbers, LED manipulation, and simple arithmetic. The next step was controlling the FPGA through the API, MATLAB®. Values were sent and retrieved through the wires set up within the HDL of the program compiled in Xilinx. The abilities to send decimal integer values from MATLAB® to the FPGA through FrontPanel™ and retrieve binary numbers from the FPGA were discovered. However, sending values through wires were found to have certain boundaries. MATLAB® was found unable to send negative or fractional numbers, as they will only be processed as positive whole numbers. On the FrontPanel side, sent and received signals through wires must be Standard Logic Vectors, and therefore unsigned and whole numbers.

To complete the system in Figure 6 the first programming step to be addressed was transferring the values calculated for the weight matrix and bias matrix to the FPGA. Two approaches were taken to resolve three complications. The first complication was the calculated weight and bias matrix values were occasionally negative and fractional, and wires were found to be able to pass only positive whole integers. The second was the difficulty of coordinating the transfer of values in an iterative sequence with the parallel processing nature of the FPGA and the von Neumann method of processing in MATLAB®. The third was the method of allocating the values within VHDL in a manner that kept iterative value, which is needed for future calculations.

The first approach kept the fractional values. MATLAB® interpreted the weight or bias matrix value as positive or negative and sent a '0' or '1' through a wire to the FPGA. The decimal portion of number was taken and converted to binary with four bit values designated for the decimal portion in MATLAB® and sent over in four wires to the FPGA. The FPGA then concatenated the wires in a 2s complement fashion and would store the values within 'for-loops' that would wait for particular signals/wires set in MATLAB® that would decide whether the next iteration was ready to be saved.

The FPGA had three processes running for the weight, bias matrix, and the input matrix. The 'for-loops' would save values into arrays of arrays functioning as two dimensional arrays. This approach worked, but was found to be too impractical and lost too much accuracy on the conversion of the fractional numbers. The lack of support of fractional number arithmetic within the libraries needed to be included in the compilation of a FrontPanel™ coordinated program also deterred this approach. Furthermore, the use of 'for-loops' with the use of arrays of arrays allocating the values made synthesizing extremely slow and consuming a lot of space. Synthesis could take up to an hour and a half when only synthesizing array of arrays that were cut down from their true size for quicker testing [1]. Synthesis is a stage in Xilinx that generates and optimizes the logical or digital form of the HDL code [1].

The second approach used one dimensional arrays without 'for-loops' to store the values. Instead of 'for-loops', the iteration number was controlled by a 'for-loop' in MATLAB® which sequentially sent values to the FPGA. The iteration number in the 'for-loop' within MATLAB® was sent in a wire to the FPGA, converted to an integer, and simply used as the iterative integer to assign the correct place in the one dimensional arrays. The weight matrix, previously a 60x256 matrix was converted into a 1x15360 one dimensional array, "15360" being the product of 60 and 256. Fractional numbers were replaced by multiplying the numbers by 1000. This approach was successful, and the matrices were successfully loaded. The design was also significantly more conducive to synthesizing digital logic as array of arrays are difficult to synthesize [1]. As a result compilation time drastically improved and more accuracy than the fractional storing system was implemented.

For all versions, when values come into the FPGA, they come in from MATLAB® as Standard Logic Vectors and are immediately converted to 2s complement if the sign flag (a wire) has been set as negative. They are then concatenated and stored in their appropriate places.

C. Calculating the Neuron Outputs on the FPGA

A fourth process is delegated the role of doing the calculation of the answer matrix. The input matrix has been normalized by MATLAB® to be 0s and 1s, in contrast to the 1s and -1s the input matrix was normalized to in the MATLAB® implementation. This allowed the ability to simply add the portions of the weight vector together when its corresponding Input Matrix iteration was a ‘1’ and then add the bias value. This method was tested with a known input vector and only two rows of the Weight Matrix. The outputs, when retrieved by MATLAB® were correct therefore demonstrating full ability of the FPGA to host the neural network. However, when making the Weight Matrix its true 60x256 size the FPGA was unable to compile as the program exceeded its memory capabilities.

V. Conclusion

A successful MATLAB® neural network was completed in its entirety. In fulfilling the ultimate goal of achieving the same success on an FPGA, significant strides were made. The interfacing between the sonar, software, and hardware depicted in Figure 6 was established and well documented, thereby establishing solid groundwork for future work. The weight, input, and bias matrices were all loaded successfully onto the FPGA and the full functionality of the neural network was demonstrated on a small scale.

The size of the current FPGA chip (1,200,000 gates) is, however, insufficient to implement the whole system with all the neurons in place.

VI. Future Work

The 60 neurons and the 256 samples per input vector are excessive in number and can be reduced. This will help alleviate the storage problem, but an in-depth look at the way arrays and ‘for-loops’ are synthesized in the Xilinx compiler is needed. The functionality needed to complete the neural network on the FPGA has been demonstrated, but optimization of how the Xilinx compiler assigns hardware to the VHDL code still remains.

Acknowledgment

I would like to thank all the assistance and guidance I have received from my mentors Professor Horiuchi and Tarek Massoud, the MERIT program, the Computational Sensormotor Lab, and the National Science Foundation.

References

[1] J. Pick, *VHDL Technique, Experiments, and Caveats*. New York: McGraw-Hill, Inc, 1995, pgs 187, 225-226, 234-233, 367-369, 375

[2] J. Hertz, A. Krogh, R. G. Palmer, *Introduction to the Theory of Neural Computation*. Redwood, CA: Addison-Wesley Publishing Co, 1991, pgs 1-20, 89-111.

[3] K. Mehrotra, C. K. Mohan, S. Ranka, *Elements of Artificial Neural Networks*. Cambridge, MA: Bradford Books, 2000, pgs 1-62

[4] A. N. Popper, R. R. Fay, *Hearing by Bats.* New York: Springer-Verlag, 1995, 1-37, 481-494.

[5] Opal Kelly Admins. “tutorial[Opal Kelly Wiki],” Feb 28, 2007. [online] Available: *http://wiki.opalkelly.com/tutorial/* [July 3, 2010].