FPGA NEURAL NETWORK

Digital Systems Design - Dr. Reddy, Fall 2009

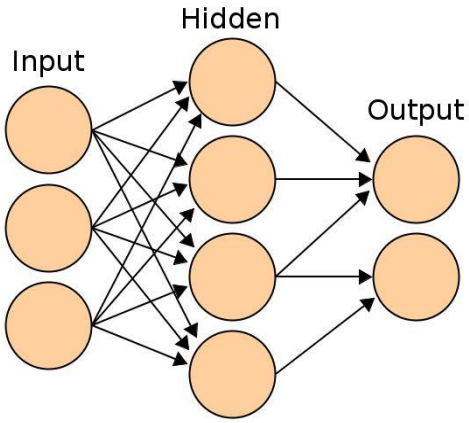
Alex Karantza **Sam Skalicky**

Artificial Neural Networks are an intriguing application of digital electronics. Using digital systems to approximate natural analog behaviors opens up new possibilities for solving problems generally considered ill-conditioned or too vague for ordinary algorithms. In this project, a VHDL implementation of a small artificial neural network on a Spartan 3E-100 FPGA is shown, and its implications for problem solving and performance are discussed.

ABOUT NEURAL NETWORKS

The history of computer science is filled with attempts to mimic biological systems using technology. Artificial intelligence was - and to many still is - the Holy Grail of computing theory. The mathematical models of many kinds of biological systems, including intelligence, were discovered and improved throughout the past hundred years. Genetic algorithms model the process of genetic mutation and evolution to optimize some kind of function. Neural networks are another kind of optimization function, modeled after the functional understanding of the cells (neurons) that make up the animal nervous system.

The actual nervous system is a very complex electro-chemical system, operating by accumulating ions and transmitting them through cell membranes and manipulating the density of neurotransmitters. Artificial neural networks simplify this system.



An artificial neural network of the kind presented here (feed-forward with back-propagation) is a graph consisting of three layers of nodes, called "neurons," each of which may be fully connected to the next layer. Each of these neurons acts as a multiply-and-accumulate operator, generating a weighted sum of the outputs of the nodes in previous layers. Each connection between these nodes represents a weight. Once this weighted sum is calculated, the neuron's output is determined by a nonlinear function - large positive sum values produce outputs close to one, large negative sum values result in outputs close to zero. This kind of function is known as a "sigmoid". Once these values have propagated through all three layers, the output of the final layer of neurons is the final output of the whole network.

This fundamental operation, however, does not explain the allure of using neural networks for problem solving. The key is that, like a human brain, this network can learn. By adjusting the connection weights properly, the output neurons can describe *any* function of the inputs, including functions specified only incompletely, or by

example. The algorithm for adjusting these weights can be thought of as running the network backwards - propagating the final error back through the weighted connections. By considering the error as a function of the weights, one can use calculus to minimize this and train the network.

MATHEMATICS OF THE ALGORITHM

As said previously, a neural network is a composition of nonlinear functions. By adjusting the weights in this composition, and performing sufficiently many compositions, any smooth n-dimensional function can be approximated. This can be described mathematically as follows.

Suppose a neuron *m* is in the hidden or output layer. It is connected to *n* neurons in the previous layer. When the network is run forward - with external inputs determining the "output" of the input layer - the neuron will perform the following weighted sum:

$$s_m = \sum_{i=0}^{n} w_{i,m} o_i$$

After the weighted sum of the previous layer is found, it is passed through a sigmoid function. This takes large values of the sum and clamps them to one, and takes large negative values and clamps them to zero, while remaining continuous and differentiable over the reals.

$$o_m = \begin{cases} i_m, \text{ if m is the input layer} \\ \dfrac{1}{1+e^{-s_m}}, \text{otherwise} \end{cases}$$

This output is then fed forward to the next layer, and so on, until the output layer.

The learning of the network is essentially a gradient descent operation using a kind of Newton's Algorithm to find the minimum of the error with respect to the weights. With the following derivation, we can find how to both propagate the error through the network and how to adjust the weights properly.

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial s_j}\frac{\partial s_j}{\partial w_{i,j}} \to 0$$

The change of the sum with respect to the inputs is simply the output of the previous run.

$$\frac{\partial s_j}{\partial w_{i,j}} = \frac{\partial}{\partial w_{i,j}}\left(\sum_{(i=0)}^{n} w_{i,j} o_j\right) = o_j$$

And the derivative of the sigmoid function is a simple function of itself, which we've already computed.

$$\frac{\partial E}{\partial s_j} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial s_j} = \frac{\partial E}{\partial o_j}\frac{d}{ds_j}\left(\frac{1}{1+e^{-s_m}}\right) = \frac{\partial E}{\partial o_j} o_j(o_j + 1)$$

When we now look at $\frac{\partial E}{\partial o_j}$, we see that it is simply the weighted sum of the errors from the following layer. This is known as the *Delta Rule* in most literature.

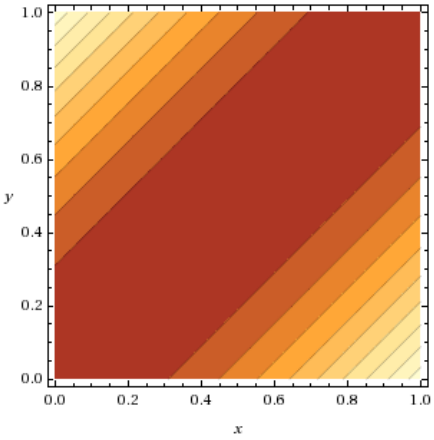
$$\frac{\partial E}{\partial o_j} = \sum_{k=0}^{n} w_{j,k}\frac{\partial E}{\partial o_k}$$

When we combine these functions together into a practical algorithm, we find that one must perform a weighted sum of errors leading from the output neurons (where their error is the difference between their result and their target) up through to the input layers. Then, by modifying the weights a fraction (λ) of $\frac{\partial E}{\partial w_{i,j}}$ we can slowly minimize the overall error.

$$\frac{\partial E}{\partial s_m} = e_m = \begin{cases} t_m - o_m, \text{ if m is in the output layer} \\ o_m (o_m + 1) \sum_{i=0}^{l} w_{m,i} e_i, \text{ otherwise} \end{cases}$$

$$\Delta w_{m,n} = \frac{-\partial E}{\partial w_{i,j}} = -\lambda o_m e_n, \text{ where } \lambda \text{ is the learning rate}$$

Here is a plot of the output of a network trained to process XOR. The boolean conditions of (0,0), (0,1), (1,0), and (1,1) can be seen to have an output of 0, 1, 1, and 0 respectively. What is more interesting is that the network - through attempting to minimize error at those four points - has created a full two-dimensional gradient that gives meaning to the XOR of arbitrary real numbers. For instance, (0.5) $\oplus$(0.5) gives 0, since they're "the same", while (0.8) $\oplus$(0.1) gives approximately (0.7), saying that they're "70% different." This simple example outlines the usefulness of training neural networks by example, and allowing them to learn their own topology and extrapolate to unseen inputs.



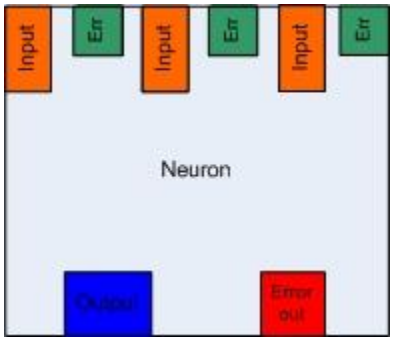
IMPLEMENTATION

This project implemented a neural network capable of learning any two-input function, such as AND, OR, XOR, etc, using two input neurons, three hidden neurons, and one output neuron, with nine weighted connections total. This is the smallest network that is practically useful, but it could be expanded to any number of inputs and outputs, operating on any number of degrees of freedom, simply by increasing the number of nodes; no additional components would need to be designed.

The core of the project is four VHDL entities. They each realize some part of the equations given above, and are arranged in a structural form so the connections between them match up to the common theoretical depictions of feed-forward neural networks.

NEURON

The most critical component is, of course, the neuron. The neuron has signal inputs ($w_{i,m} o_i$), a signal output (o_m), error inputs ($w_{m,i}\ e_i$), and an error output (e_m). When the signal inputs change, the neuron computes the new sum of those inputs. The summed value is then passed into the Sigmoid component (discussed later) which determines the final signal output. A similar



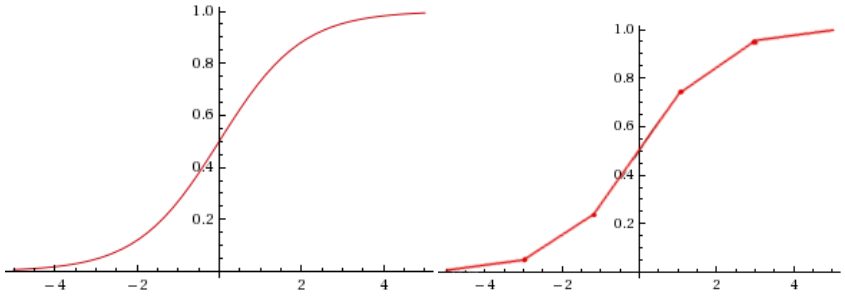
thing happens when the error inputs change; their sum is calculated, and then multiplied by the sigmoid derivative ($o_m\ (o_m + 1)$) to become the error output.

CONNECTION

The Connection components serve as the weights connecting the layers of neurons. They receive the signal and error values from the two neurons they connect, weight those values, and pass them along. Additionally, the Connections may be put into learning mode. When the error value changes, it is not only propagated along but the Connection modifies its internal weight by the equation derived in the previous section ($\Delta w_{m,n} = \gamma\ o_m e_n$).

SIGMOID

The neuron needs a nonlinear activation function to apply to its output. To this end it instantiates the Sigmoid entity. This entity, more than all the others, has undergone the most revisions and is most responsible for the network's performance. The sigmoid, $\frac{1}{1+e^{-sm}}$ is a difficult expression to compute accurately in digital logic. For the gradient-descent approach to learning to be effective, the function used must not only approximate the sigmoid in value, but also in the first derivative. For reasons discussed later, the final implementation of this entity consists of a piecewise linear approximation. This lack of precision, especially far from the origin, has been the culprit of the numerical issues faced in testing.



NETWORK

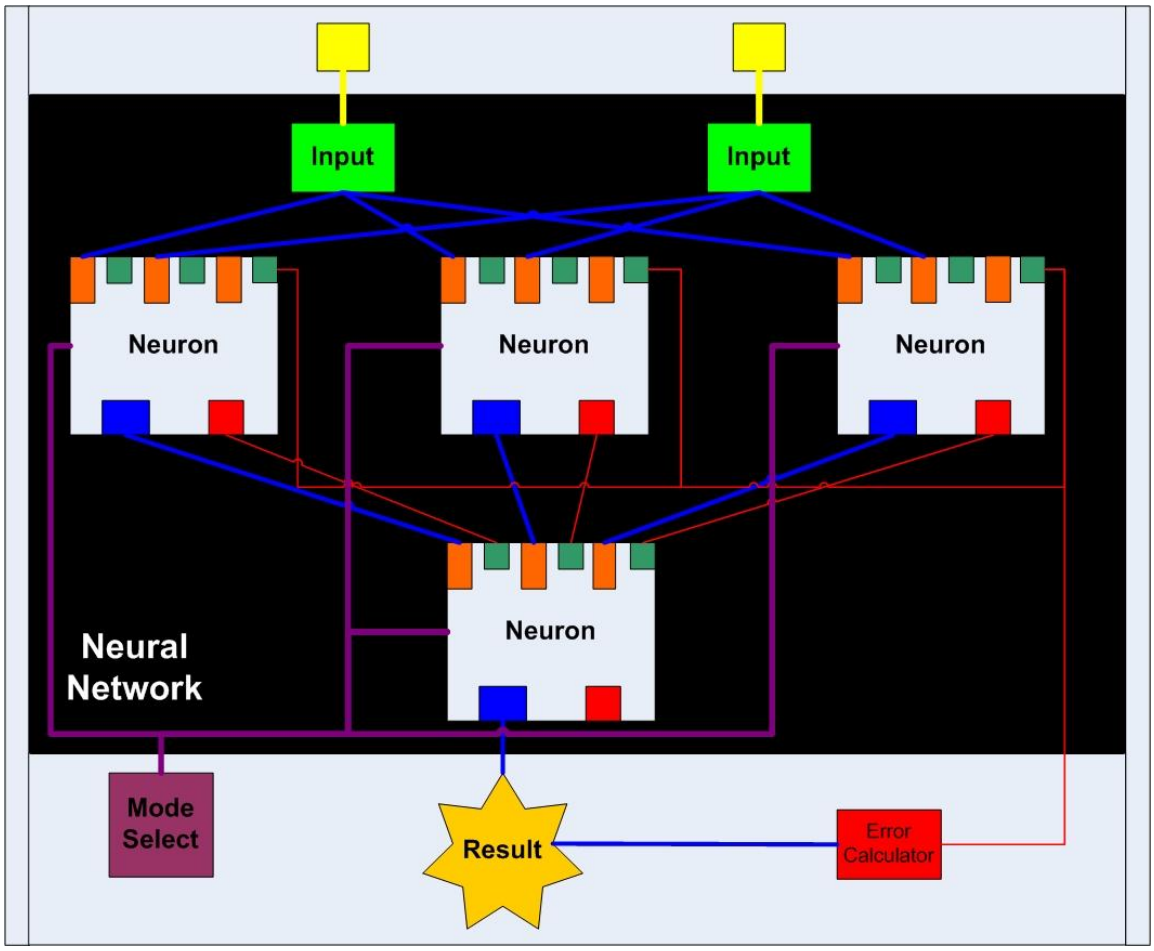
The network entity is what puts together the Neurons and the Connections to decide the topology of the neural net. Numerous internal signals are used to establish the connections between instantiated components. In a less academic implementation, this entity would likely use for-generate statements to build a network of the required size and dimensionality as part of a larger system. The I/O to this entity exposes two values for input, one value for output, a value for entering in the error in the output, and a value for determining if the network is in learning mode.

INTERFACE & TESTBENCH

Other entities such as test benches and generators were created for debugging and simulation purposes. Additionally, an entity was made to interface the Network component with the netlists providing access to the FPGA board's physical I/O. The demo board provides eight LEDs, and eight toggle switches. The interface to the network uses these LEDs and switches. One switch causes the network to retrain. While retraining, a red LED is illuminated to indicate that the error is too high. When the network trains sufficiently, a green LED will light and training will stop. The training set for this two-input binary function is specified by four switches - the target output for (0, 0), (0, 1), (1, 0), (1, 1) respectively. When the training switch is off, two additional switches are used to query the network. The output value - an eight-bit signed number - is displayed on the eight LEDs. Numbers close to 256 indicate a positive response from the network, numbers at or below 128 indicates a zero response.

DESIGN OF SYSTEM

A diagram showing the final design of the neural network is shown below. You can see the data lines between the various components (shown with blue lines) to follow the datapath to the final output result. From here the output gets routed to an error calculator (which, depending on the function) determines the difference between expected output and actual output, and passes this data back into the network (red lines). Lastly is the mode selection lines for choosing between running output, and training of the network (purple). The input from the outside world (switches) are represented as the yellow input blocks.

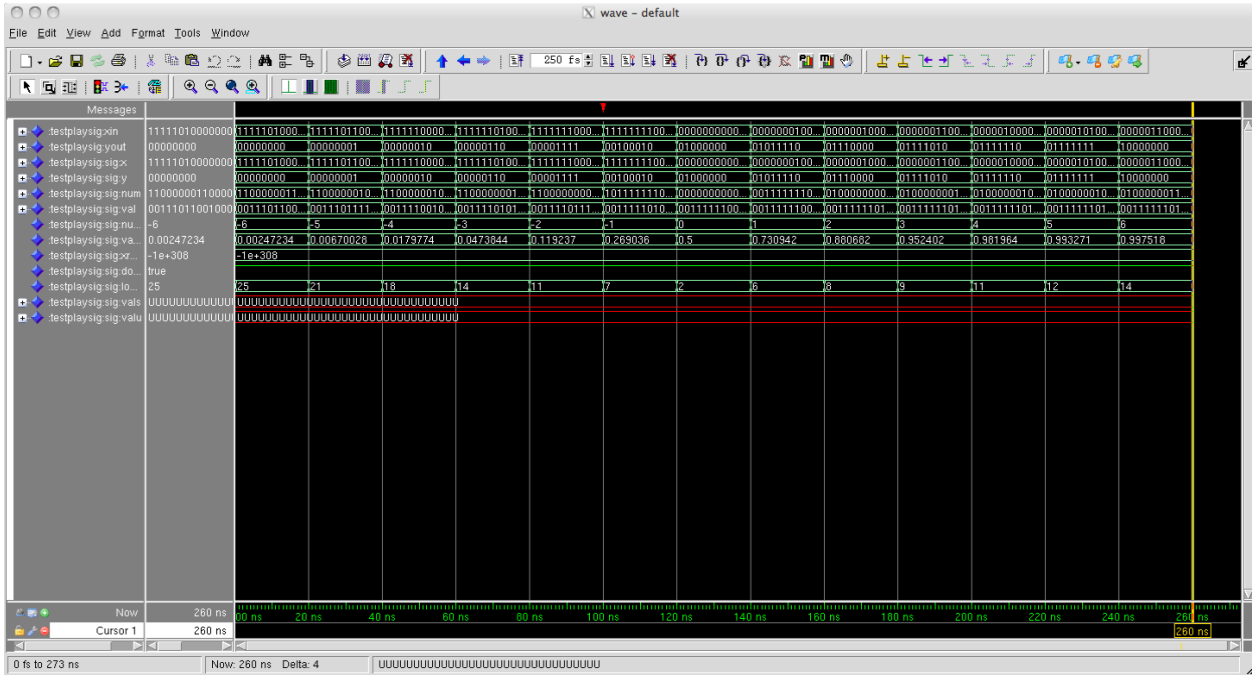


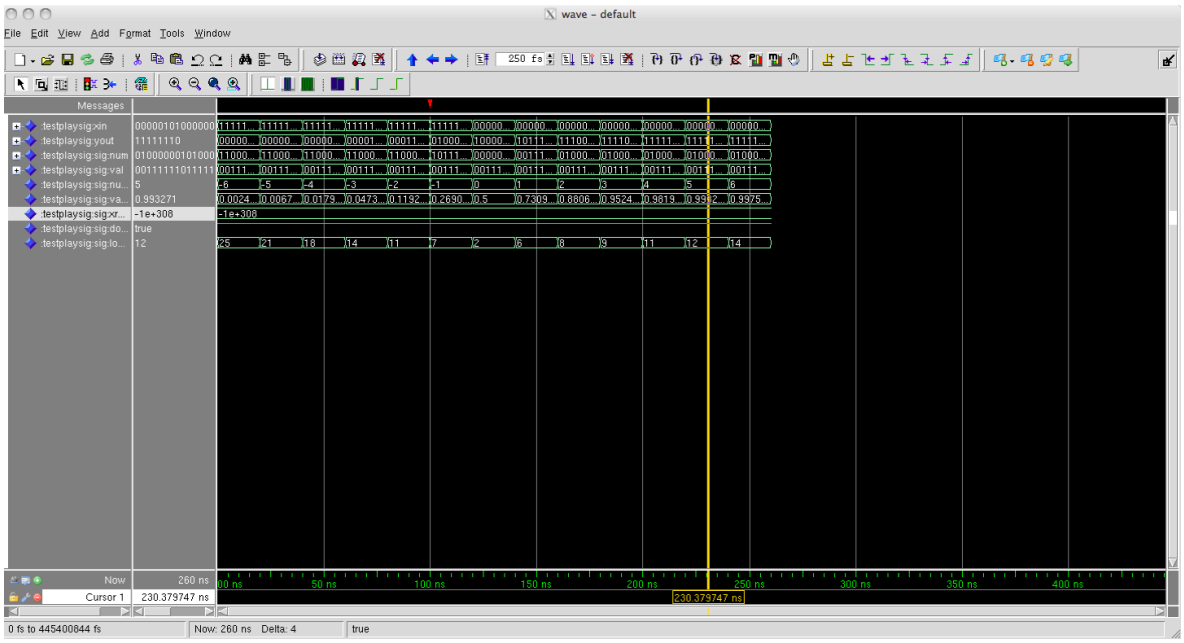
NUMBER REPRESENTATIONS

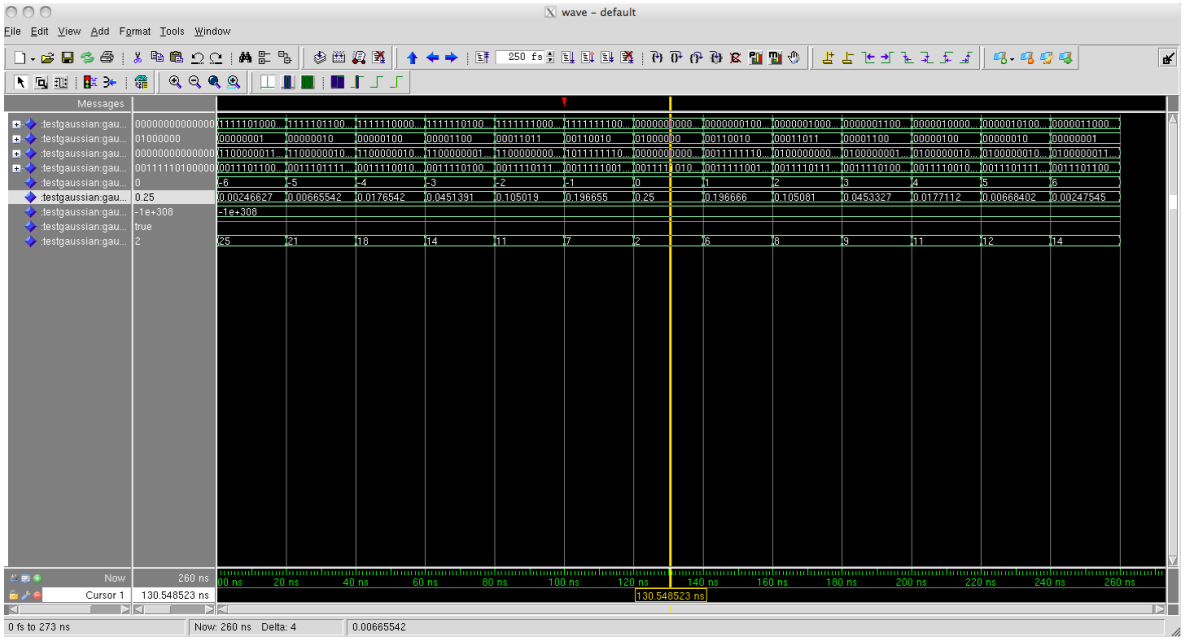
Internally all the calculations are done using custom fixed point arithmetic. This is due to the lack of native floating point support in the IEEE libraries and on the FPGA used. This choice had ramifications that were not entirely expected. The nature of the neural network demands high precision fractions as well as signed values inside the weights and neurons (outputs are always unsigned).

The gradient descent operation on the sigmoid only works if the sigmoid and its derivative are continuous, differentiable, and non-zero at all points in their domain. When using fixed-point representations, there is a point at which the discretization of these functions violates those conditions. This manifests itself as incomplete training, or an inability to converge. Using floating point values would give much greater dynamic range, and indeed the C language prototypes of this project which use floating point values do not exhibit these problems.

TESTING







RESULTS

We initially started out using a C program to verify the logic design of the network. Using this model we were able to design the logic in hardware. The values between VHDL network and C network were almost identical using the floating point library. However when this wasn't synthesizable on the FPGA, we used another version that gave us relatively close numbers (within .01) of each other. We also ran into some problems with logic initially, however once we implemented the self testing in the Network entity, we were able to narrow down the problem very quickly. This ease of debug was thanks to the BIST Generator entity that was implemented and now checks the network at the beginning of every power up sequence.

Total resources used: 19%	Device Utilization Summary		
Logic Utilization	Used	Available	Utilization
Number of Slice Latches	22	1,920	1%
Occupied Slices	197	960	20%
4 input LUTs	360	1,920	18%
	Logic	328	17%
	Route-thru	32	3%
Number of bonded IOBs	10	108	9%
MULT18X18SIOs	4	4	100%

Table 1: Device Utilization table for Spartan 3E-100

Node	Levels	Time
Data In	29	13ns
Data Out	2	5ns

Table 2: Data Speed

The tables above represent the data from the Xilinx ISE software that describes the amount of resources used by this design in the Spartan 3E-100 FPGA. We wondered how accurate these values were, and we found out that these values really depend on the type and size of FPGA being used. As well as the type of specific hardware resources available (such as 18x18 multiplies) and the routing between the different parts of the design on the fabric of the FPGA define these numbers.

FUTURE WORK

The network implemented in this project has a few shortcomings that would need to be addressed if the intent was to use it in a practical or industrial application. The most important of which is the discretization of the numbers used in the network. Floating point computations would solve the overflow and saturation issues.

Included in the VHDL listings, but not strictly included in the project, is an alternative implementation of the sigmoid function that assumes floating point support is available. Instead of approximating the sigmoid with a piecewise linear function, it computes the function properly using first few terms of the Taylor series for the exponential. Clearly, much dynamic range is needed to store these values of the exponential. In simulation, however, this has proven a much more effective solution.

$$Sigmoid(x) = \left(1 + \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!}\right)^{-1}$$

Taylor series approximation to inverse exponential used

From the beginning a generic form of the Network entity was desired. However, this was not achieved. The framework for this generic functionality is still inherent in the design. This is still an area of interest and will be improved upon.

CONCLUSION

The results from the testing show that the network is able to correctly evaluate the weights to produce the correct outputs of the XOR function. This can thusly be tailored for any set of outputs that directly related to respective inputs. Using this design, functions with many inputs such as touch sensitive devices can be used for character recognition, where the correct input may be many possible inputs that are so large they cannot be defined specifically. However doing these calculations takes up a lot of processing power, so being able to run these calculations on hardware versus software speeds up the time to answer and opens the door for this logic to be used in new areas.

This project has demonstrated the feasibility of applying a theoretical understanding of neural networks to the design of a parallel asynchronous digital system, and the speed benefits of such efforts. While the ability of the prototype configured for this demonstration is rather shallow, the principles are applicable to systems of any size, and the nature of the equations lead themselves well to even more parallel systems. The prototype was limited by the computational resources available on the FPGA chosen - the lack of sufficient hardware multipliers or a floating point unit. All these problems could be easily overcome if a practical neural network was desired for quickly solving ill-conditioned problems, for learning based on fragmented examples, or for helping to gain an insight into the function of biological neural networks.