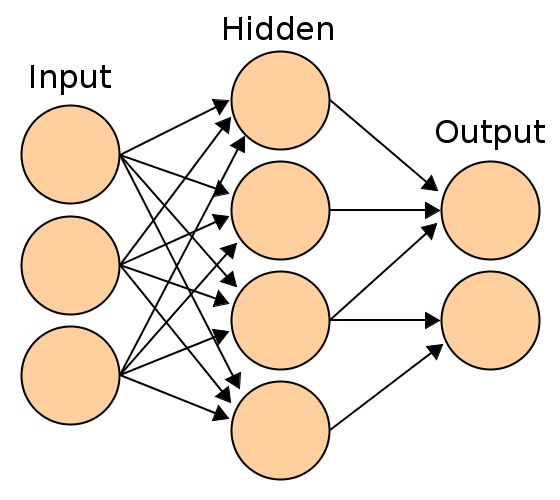
FPGA Neural Network  
Digital Systems Design - Dr. Reddy, Fall 2009

**Alex Karantza**

**Sam Skalicky**

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

## About Neural Networks

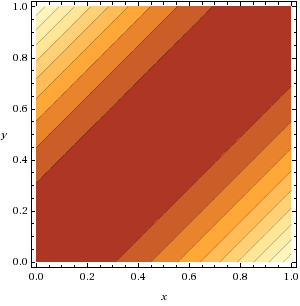
 The history of computer science is filled with attempts to mimic biological systems using technology. Artificial intelligence was - and to many still is - the Holy Grail of computing theory. The mathematical models of many kinds of biological systems, including intelligence, were discovered and improved 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



## 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.

<Structural overview of the entities in the project and how they realize the equations above>

Internally all the calculations are done using custom fixed point arithmetic. <Explain why we did this, and why not floats, and how we could have improved with floats>

<How we use the demo board's IO>

## Testing

<Simulation in ModelSim>

<Sim results>

<Physical testing>

## Results

<Modelsim vs C implementation>

<Testing on the xilinx demo board>

<Device utilization>

|  |  |  |  |
| --- | --- | --- | --- |
| 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 : Device Utilization table for Spartan 3E-100 | | | |

<Performance on the fpga>

|  |  |  |
| --- | --- | --- |
| **Node** | **Levels** | **Time** |
| Data In | 29 | 13ns |
| Data Out | 2 | 5ns |
| Table : Data Speed | | |

## Further Work

<Talk about what would be needed of a board to implement the Taylor series, better IO system to solve problems like robotic motion or pattern recognition>

## Conclusion

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.