A Neural Network Implementation on Embedded Systems

by

Nicholas Jay Cotton

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 9, 2010

Keywords: Neural Network Implementation, Microcontroller, Neural Network Training

Approved by

Bogdan Wilamowski, Chair, Professor of Electrical and Computer Engineering
Thaddeus Roppel, Associate Professor of Electrical and Computer Engineering
Victor Nelson, Professor of Electrical and Computer Engineering
Vitaly J. Vodyanoy, Professor of Physiology

Abstract

This dissertation presents a solution for embedded neural networks across many types of hardware and for many applications. The software package presented here allows the user to develop a neural network for a desired application, train the network, embed it on most platforms, and verify its functionality. This software supports advanced and very powerful types of neural networks including cascade, fully, and arbitrarily connected networks. It also supports several different training algorithms both first and second order. This system automates the process of transforming the trained neural network to an embedded neural network on most microcontrollers with a C compiler. There is also an assembly language neural network highly optimized for speed based on an inexpensive 8-bit PIC microcontroller. Software for testing and verifying functionality of the embedded neural networks is also included. Several neural network examples are also shown being calculated on the embedded system.

Acknowledgments

The author would like to thank Auburn University's Department of Electrical and Computer Engineering at the Samuel Ginn College of Engineering for their support during this research. The author would also like to recognize his advisor and mentor Dr. Bogdan Wilamowski for his professional and personal support over the years. Without Dr. Wilamowski's confidence and commitment to the author, graduate school would not have been possible. A graduate student simply could not have a better advisor. The author would also like to thank the members of his committee; Dr. Thaddeus Roppel, Dr. Victor Nelson, and Dr. Vitaly Vodyanoy. The author also recognize Dr. Roppel for his extra guidance and support through ought the author's undergraduate and graduate curriculums at Auburn. He has been a source of great support several times through the years. The author would lastly, but most importantly, thank his wife Erin for her love, patience, and encouragement that provided the perseverance to continue in the most difficult of times.

Table of Contents

Abstract ii

Acknowledgments iii

List of Tables v

List of Figures vi

List of Abbreviations vii

1. Review of Embedded Neural Networks 1

1.1. Neural Network Critical Components 2

1.2. MLP Versus Arbitrarily Connected Networks 3

1.3. Activation Functions 6

1.4. Analog implementation 11

1.5. Microcontroller Implementations 13

1.5.1. Embedded Neural Network for Fire Classification Using an Array of Gas Sensors 14

1.5.2. Microcontroller Based Neural Network Controlled Low Cost Autonomous Vehicle 17

1.5.3. Control Sensor Linearization Using a Microcontroller-Based Neural Network 20

1.5.4. A Solar-powered Battery Charger with Neural Network Maximum Power Point Tracking Implemented on a Low-Cost PIC-microcontroller. 21

1.6. Current Research Summary 23

2. Neural Network Training 25

2.1. Neural Network Trainer 26

2.1.1. Training Data 27
2.1.2. Input File 28
2.1.3. Training Parameters 30
2.2. NNT Adaptations 34
2.2.1. Neural Network Weight Scaling 35
2.2.2. 8-Bit Neural Network Simulator 36
2.2.3. Generated Files 40
2.3. PIC Simulator Software (PicSim) 43
3. Hardware Implementation 46
3.1. Pseudo Floating Point 46
3.2. Multiplication 48
3.3. Addition and Subtraction 49
3.4. Activation Function 50
3.5. Memory Structures 56
3.6. Neuron By Neuron Computation Process 58
3.6.1. Forward Calculations 58
3.6.2. Individual Neuron Calculations 62
4. Application 65
4.1. Simple Surface 67
4.2. Matlab's Peaks Surface 76
4.3. Two Arm Planar Manipulator 79
4.4. Matlab's Peaks Surface Example In C 87
4.5. Experimental Data Summary 90

5. Conclusion 92

References 95

List of Tables

Table 1: Training data used by Bashyal et al. in "Embedded Neural Network for Fire Classification Using an Array of Gas Sensors". The authors labels the six sensor readings TGS but do not number each sensor. 16

Table 2: Neural network performance comparison. 90

List Of Figures

Figure 1: A single neuron neural network with two inputs and one output. 3

Figure 2: Ten hidden neuron MLP network for solving parity-9. 5

Figure 3: Fully connected cascade neural network with ten hidden neurons for calculating parity-1023. 6

Figure 4: Tangent Hyperbolic function used for neural network activation function. 8

Figure 5: Linear activation function with saturation. 9

Figure 6: Piecewise linear activation function. 9

Figure 7: Sigmoid function non-linear tanh approximation 10

Figure 8: Elliott function non-linear tanh approximation. 10

Figure 9: Sigmoid activation function with an input voltage and output current. 12

Figure 10: Analog sigmoid circuit output. 12

Figure 11: Network used by Bashyal et al. in "Embedded Neural Network for Fire Classification Using an Array of Gas Sensors". 15

Figure 12: Neural network from Farooq's" Microcontroller based Neural Network Controlled Low Cost Autonomous Vehicle". 19

Figure 13: Neural network for "Control Sensor Linearization Using a Microcontroller-Based Neural Network" by Dempsey et al. 20

Figure 14: Neural Network implemented in Petchjatuporn's work on "A Solar-powered Battery Charger with Neural Network Maximum Power Point Tracking Implemented on a Low-Cost PIC-microntroller". 22

Figure 15: Front end of Neural Network Trainer (NNT) 26

Figure 16: Three Neuron architecture for parity-3 problem. 29

Figure 17: PicSim software for simulating and verify embedded neural networks. 43

Figure 18: Implementation of 16-bit fixed point multiplication using 8-bit hardware multiplier. Steps 1-4 are summed with place holders to give the final product on the result line. Abbreviations: Integer (I) Fractional (F) Product (P) Lower-Byte (L) Higher-Byte (H). 48

Figure 19: Logical block diagram of the activation function. 52

Figure 20: Example of linear approximations (red) and parabolas between 0 and 4 (magenta). *Tanh* (green) and the approximation (blue) are also shown on the graph. Only 4 divisions were used for demonstration purposes. 54

Figure 21: Error from *tanh* approximation using 16 divisions from -5 to +5. 56

Figure 22: Memory Allocation table for Pic18F45J10. 58

Figure 23: Block diagram of Neural Network forward calculations using the nested loop structure for cross layer connected networks. 60

Figure 24: PF stands for Pseudo Floating point number. The Numbers in brackets refer to the number of bits that represent that particular value. 62

Figure 25: Pre Activation Function Routine. The transformation between a pseudo floating point number to a fixed point number that the activation function can use. 63

Figure 26: Simple surface training data. 68

Figure 27: Four neuron cascade architecture for solving the simple surface. The inputs are the circles on the left and the output is the last neuron on the right side. 69

Figure 28: Ideal neural network output. 69

Figure 29: Output of the PIC. 70

Figure 30: Error surface showing the difference of the training data and the ideal neural network. 70

Figure 31: Error surface showing the difference of the PIC output and the ideal neural network. 71

Figure 32: Error surface showing the difference of the PIC output and the training data. 71

Figure 33: Histogram of errors between the PIC and the training data. 72

Figure 34: Output of PIC with 196 test patterns. 73

Figure 35: Two neuron architecture for solving simple surface problem. 74

Figure 36: PIC output with 196 points on small two neuron architecture. 74

Figure 37: Histogram of errors between the ideal neural network and the PIC. 75

Figure 38: Error of the PIC and the Training data compared. 75

Figure 39: Training data used for Matlab's peaks surface. 76

Figure 40: Eight neuron network used for solving the Matlab peaks surface. 77

Figure 41: Pic output for Matlab's peaks surface. 78

Figure 42: Histogram of errors between the PIC output and the training data. 78

Figure 43: Histogram of errors between the ideal neural network and the training data. 79

Figure 44: Two arm planar manipulator with variables shown. 80

Figure 45: Ten neuron network for solving forward kinematics problem. 82

Figure 46: Training data output x of the two output system. 83

Figure 47: Output x of two output system generated by embedded neural network. 83

Figure 48: Training data output y of the two output system. 84

Figure 49: Output y of two output system generated by embedded neural network. 84

Figure 50: Error between the embedded neural network and training data of output x. . 85

Figure 51: Error between the embedded neural network and training data of output y. 85

Figure 52: Histogram of Errors between training data and PIC for output x. 86

Figure 53: Histogram of Errors between training data and PIC for output y. 86

Figure 54: Output of the PIC using the C version of the embedded neural network software. 88

Figure 55: Error between the ideal neural network and the PIC output using C. 88

Figure 56: Histogram of errors between Ideal neural network and training data. 89

Figure 57: Histogram of errors between ideal neural network and PIC implemented neural network. 89

List of Abbreviations

NNT	Neural Network Trainer
EBP	Error Back Propagation
FPGA	Field Programmable Gate Array
MLP	Multi Layer Perceptron
PIC	Microcontroller By Microchip Technology Inc.
ACN	Arbitrarily Connected Networks
FCC	Fully Connected Networks

1. Review of Embedded Neural Networks

Neural networks have become a growing area of research over the last few decades and have affected many branches of industry. The concept of neural networks and a few types of their applications in industrial electronics are summarized in [1]. In the field of industrial electronics alone there are several applications for neural networks, some include motor drives [2-9] and power distribution problems dealing with harmonic distortion [10-23]. These papers show how valuable neural networks are becoming in industry. Due to the nonlinear nature of neural networks, they have become an integral part of the field of controls [24-26]. On a parallel note, embedded applications are also becoming exponentially more prevalent [27-33]. However, even though these two independent topics are continually growing there is not a significant amount of research being done on embedded neural networks. This dissertation proposes a solution for implementing neural networks on microcontrollers for many embedded applications.

Many people have a predisposition about neural networks, one being that they require significant computing power. One researcher stated, "most embedded microprocessor cores lack the performance for running neural networks" [34]. Many researchers have implemented neural networks on sophisticated hardware; for example, creating dedicated Application-Specific Integrated Circuits (ASICS) as in [34-39]. Others have used Field Programmable Gate Arrays (FPGAs) to perform the neural network calculations for embedded tasks [40-46]. High-end digital signal processors

(DSPs) are also commonly used to implement neural networks because they are typically designed with floating point hardware. They also have multiply and accumulate registers which are helpful when processing neural network applications. A few examples of DSP implementations are described in [47-54]. Because DSPs and FPGAs have more computing power, they tend to be very expensive. However, this large amount of power is not necessary for implementing neural network tasks that can easily be done on an inexpensive microcontroller.

There is no current solution for implementing neural networks into an embedded environment other than for a few specific applications. The above-mentioned articles validate the utility of neural networks. Technology as a whole is becoming more portable which leaves a need for a portable neural network solution. This dissertation discusses a solution for embedding any neural network on a microcontroller. It also offers methods for implementing Multi Layer Perceptron (MLP) and arbitrarily connected networks (ACN), discussed in the next section, on a microcontroller.

1.1. Neural Network Critical Components

Neural Networks are made up of several critical components. The largest component is the neuron itself which is the triangle component in Figure 1. A neural network is made up of one or more neurons connected in any configuration. The connecting lines represent weights. Selecting these weights determines how the neural network will respond to particular input patterns. Training neural networks is the method of selecting weights to give the desired output with a given set of inputs. Neural

networks gain their nonlinear properties from their activation function which is represented by the signal passing through the neuron. Neural networks can take on many shapes and sizes and be arranged in an infinite number of ways. Some of the most common networks will be discussed in this dissertation.

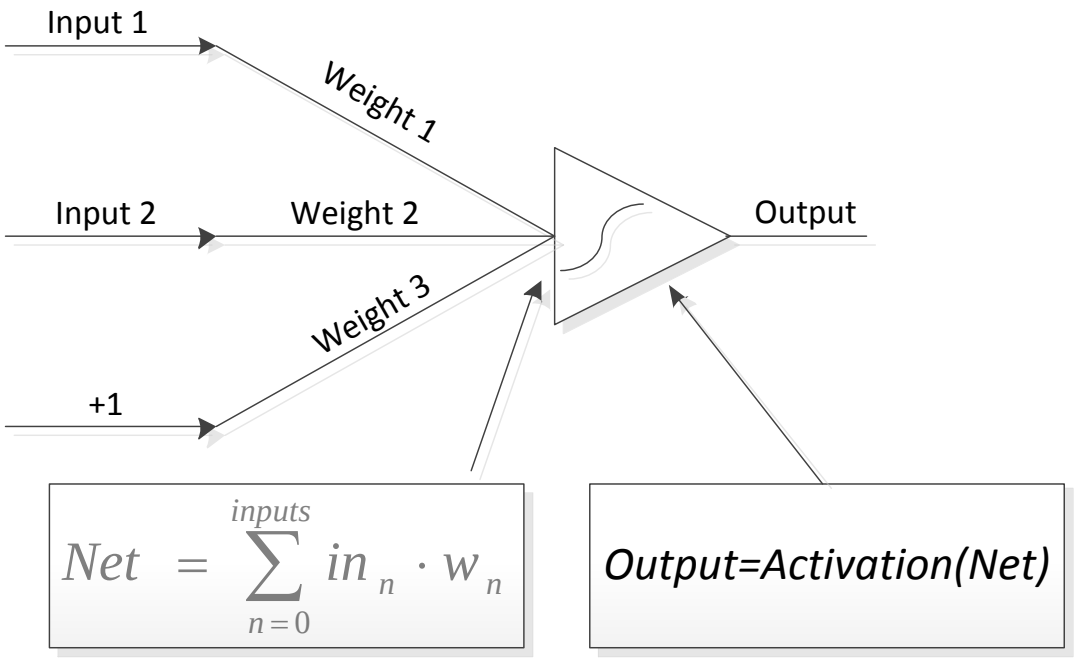

Figure 1: A single neuron neural network with two inputs and one output.

1.2. MLP Versus Arbitrarily Connected Networks

Much of embedded neural network research has involved using Multi Layer Perceptron (MLP) networks. Little research, however, entails neural networks with arbitrarily connected networks on the embedded system level. This is unfortunate because these neural networks are superior to traditional MLP networks in several ways.

These networks are faster, more reliable to train, more efficient because less neurons are needed to solve similar problems, and they can solve more difficult problems that are nearly impossible for MLP networks to solve[55-62]. One common benchmark for training neural networks is the Parity-N problem. For example, if an MLP network is used with one hidden layer with ten neurons then the largest parity problem that can be solved is parity-9. However, if the same ten neurons are used in arbitrarily connected cascade architecture then the network would be capable of solving parity-1023 [63]. The MLP and arbitrarily connected cascade architectures can be seen in Figure 2 and Figure 3 respectively.

Despite the drawbacks of using MLP networks, neural network research is still exclusively done using them. As stated earlier, arbitrarily connected networks are superior. Unfortunately, people seldom utilize the latter because they do not have the software to train them. This dissertation offers a solution to train these networks.

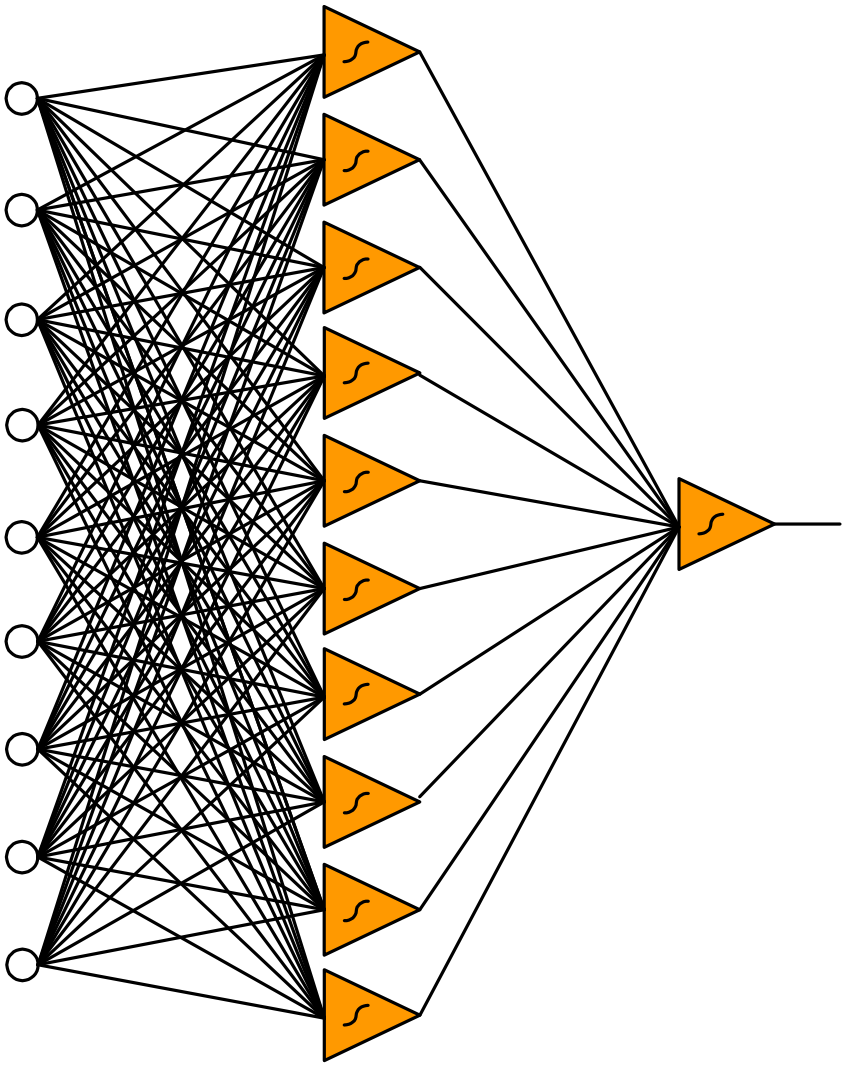

Figure 2: Ten hidden neuron MLP network for solving parity-9.

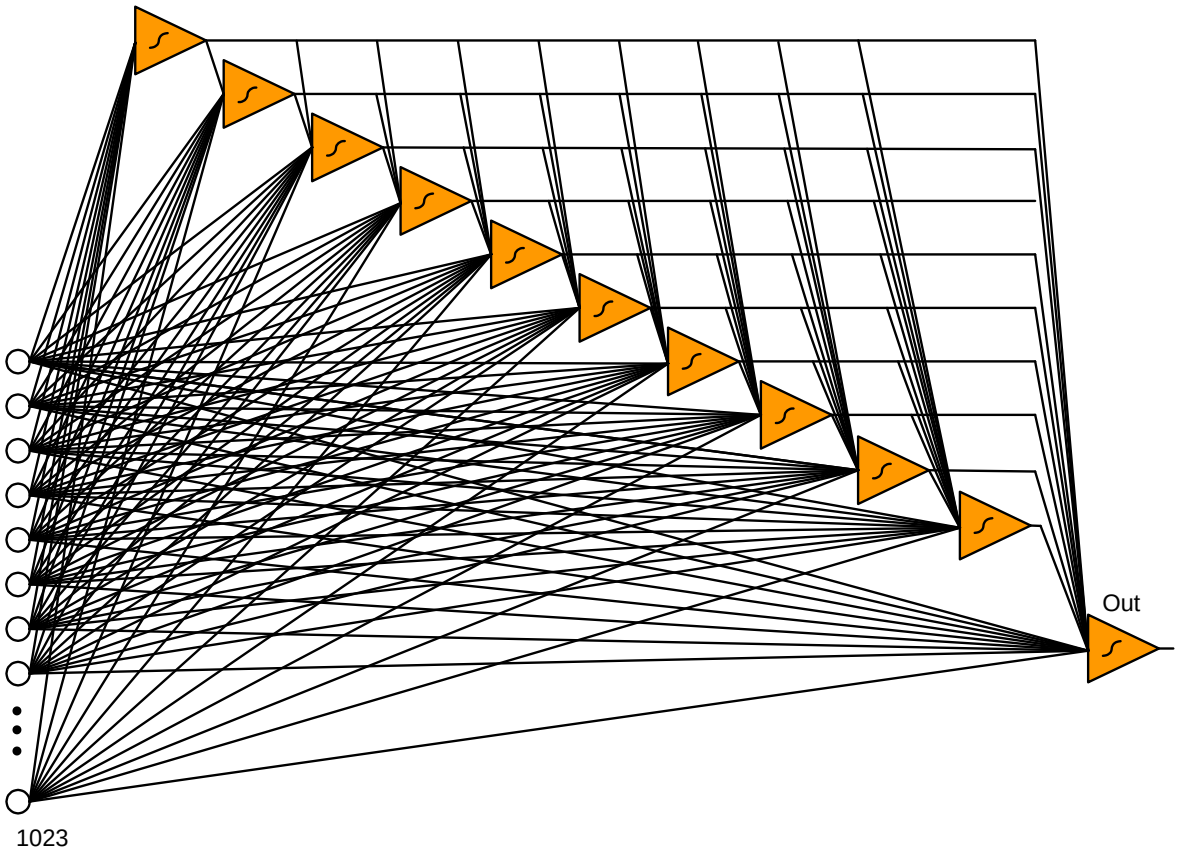

Figure 3: Fully connected cascade neural network with ten hidden neurons for calculating parity-1023.

1.3. Activation Functions

The most common neural network activation function is tangent hyperbolic (*tanh)* which is defined in Equation 1 and shown in Figure 4. Much research on embedded neural networks uses some approximation of *tanh* and most training software trains neural networks based on the activation function *tanh.* One of the simplest approximations is a linear approximation with saturation points as shown in Figure 5 [64]. A piecewise linear activation function approximation of *tanh* is shown in Figure 6 [65]. All of these linear approximations are based on the *tanh* activation function and

allow for quick, simple linear calculations, however at the expense of decreasing accuracy. *Tanh* is defined in Equation 1.

$$tanh = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{1}$$

There are other nonlinear approximations that are easier to calculate than *tanh* but more accurate than the linear versions shown here. One very common activation function is the sigmoid function [66-67]. The sigmoid function is shown in Equation 2 and Figure 7. This sigmoid function has a similar shape as *tanh* but it ranges from zero to one rather than negative one to positive one. This needs to be taken into consideration when the neural network is trained because commercially available neural network training software may not include the sigmoid activation function; thereby, creating difficulty for the user to train the network with the same activation function the hardware is going to use.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

The Elliott function is another nonlinear activation function used by [68]. This function is between negative one and positive one but its shape is not as steep as *tanh* requiring the network to also be trained using the Elliot function. This function can be seen in Equation 3 and Figure 8.

$$f(x) = \frac{x}{1 + |x|} \tag{3}$$

The final common activation function is a simple lookup table. The lookup table can have a variety of different layers of accuracy but this accuracy is exponentially proportional to the number of data points that must be stored. This approach was used in

[69] and stored 256 values, which is not enough resolution. The problem with this method is a vast amount of memory must be dedicated to storing the values.

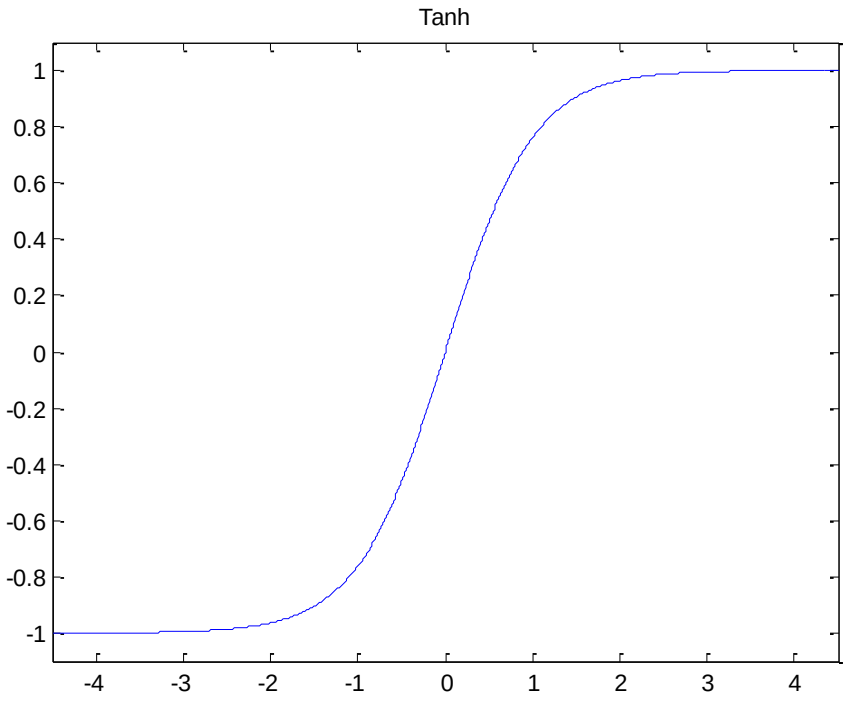

Figure 4: Tangent Hyperbolic function used for neural network activation function.

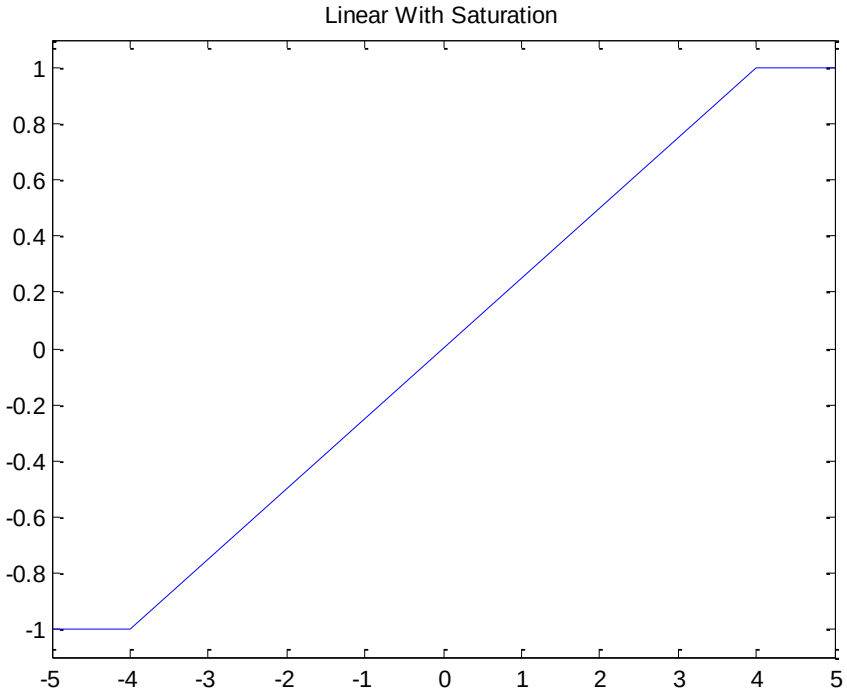

Figure 5: Linear activation function with saturation.

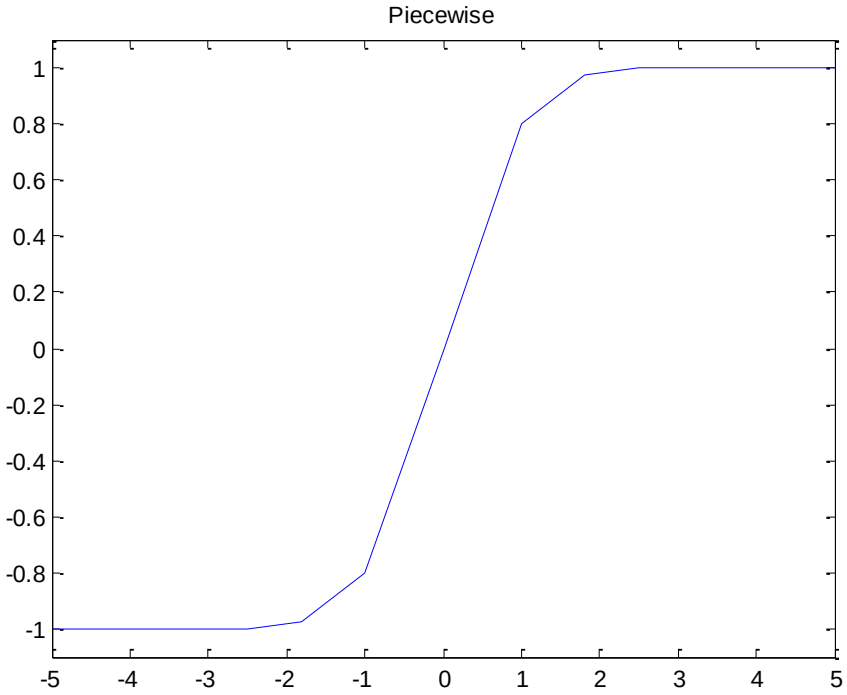

Figure 6: Piecewise linear activation function.

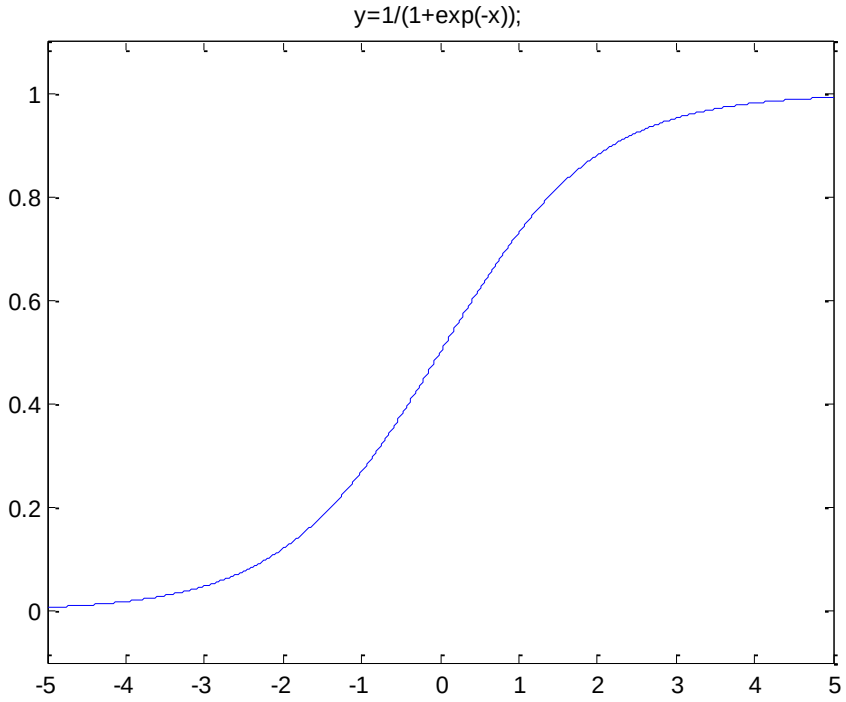

Figure 7: Sigmoid function non-linear tanh approximation

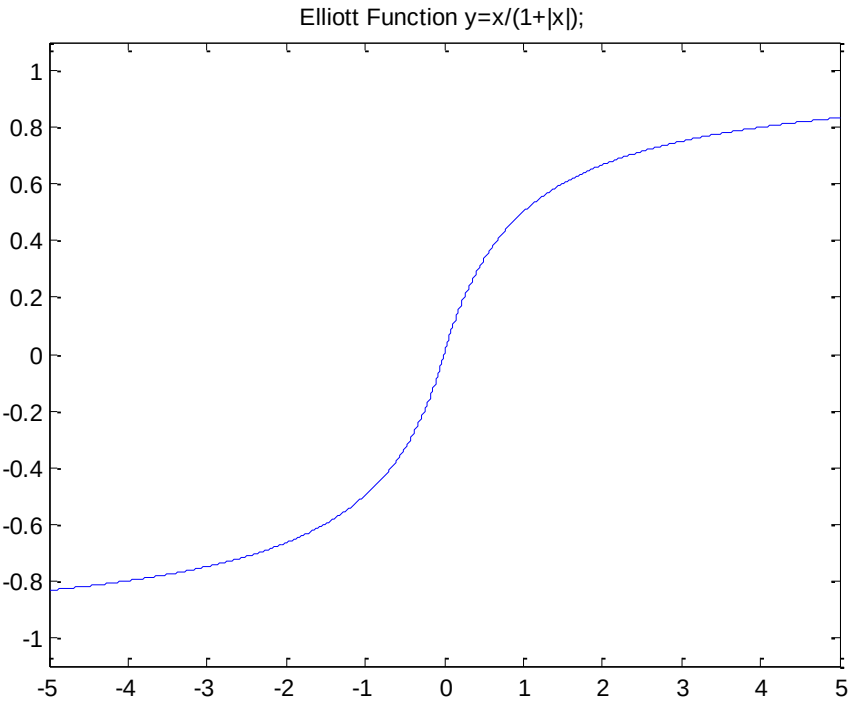

Figure 8: Elliott function non-linear tanh approximation.

1.4. Analog implementation

The analog implementation of neural networks becomes a broad category of methodologies of using neural networks. Most people's renditions work for their specific cases but are not general solutions for analog neural networks. The different ways to implement neural networks in an analog system are as diverse as the applications for neural networks. Every researcher has their own method and approach which is most appropriate for their need and resources. Some researchers make every network on the ASIC level while others have created an ASIC for a single neuron that they can use in different applications [37, 70-73].

Many different activation functions are used from simple threshold functions that are essentially a digital 1 or 0 output to full *tanh* approximations. An activation function that can easily be implemented in VLSI without the need of resistors is demonstrated in Figure 9 [72]. This activation function is made up of two CMOS coupled differential pairs. These circuits are biased by I_H and I_L which also become the higher and lower limits of the sigmoid function. The right half of the circuit is a voltage reference that is biased at ground. This centers the sigmoid around zero. The output of the circuit can be seen in Figure 10.

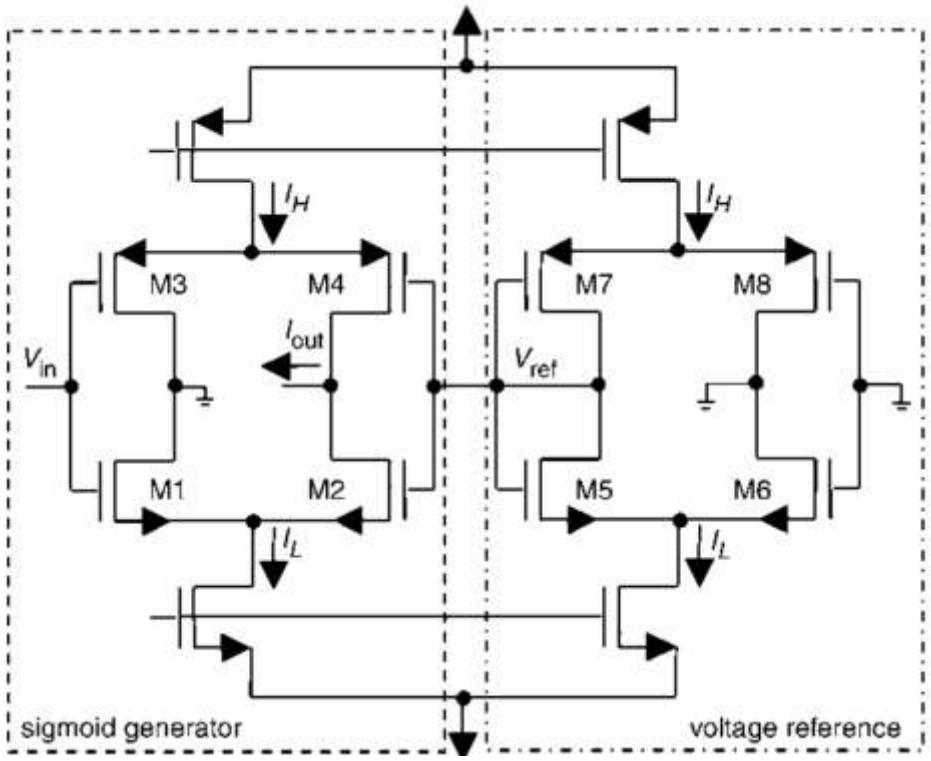

Figure 9: Sigmoid activation function with an input voltage and output current.

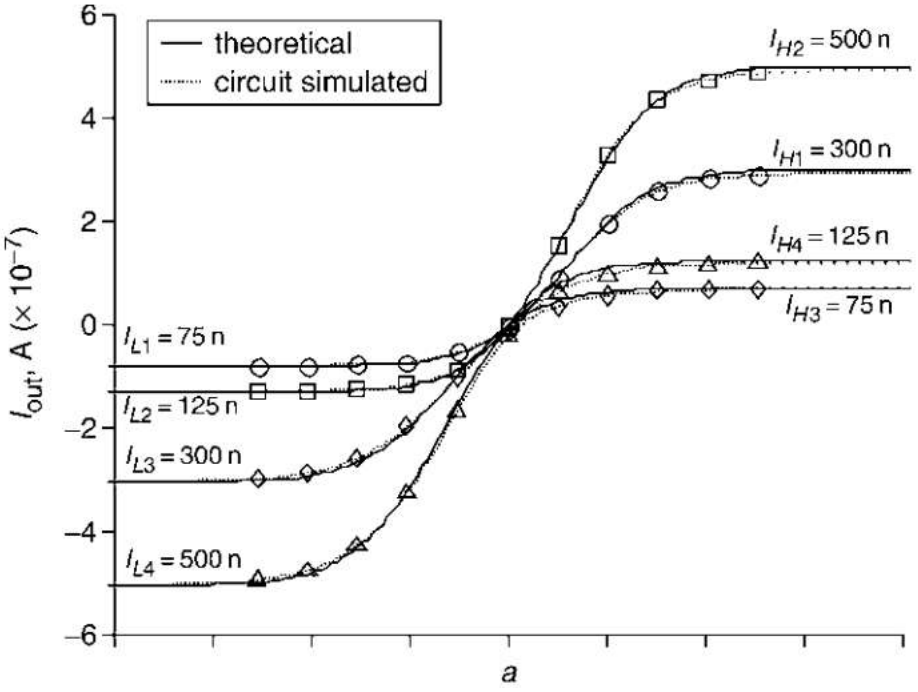

Figure 10: Analog sigmoid circuit output.

Analog neural network implementations offer incredible speed of calculation but not without a price. Probably the single largest deterrent for analog networks is the cost. Application Specific Integrated Circuits (ASICS) are very expensive and time consuming to produce making them not practical for most neural network applications.

Analog neural networks also have other pitfalls such as accuracy of the fabrication process affecting the accuracy of the network output. It is hard to produce very accurate resistive material which is typically used for the weights. It is also difficult to account for circuit variation from temperature dependant components. Moreover, it is very difficult to create accurate analog neural networks.

Finally, with integrated analog circuits it is not possible to modify the network once it is finished. This prevents the network from being retrained or even slightly tuned to be more accurate. Overall, analog neural networks simply are not a feasible solution for the majority of neural network applications.

1.5. Microcontroller Implementations

There have been few papers published [66, 74-79] on microcontroller-based neural networks on a comparable level to the one used in this dissertation. These papers will be discussed in the following section. However, they are lacking in several categories which is the main motivation for this work. These categories are neural network architectures, activation function, and training algorithms. Neural networks are great for approximating systems with limited training points but are exposed to a large

variety of input patterns. Neural networks do a great job of predicting the outputs between the data points. In a large portion of the research, neural networks are used to solve digital type problems with a limited number of inputs and outputs. These problems are not well suited for neural networks but could be solved simply using logic gates. In these applications all possible scenarios are specifically trained. This situation does not utilize the full power of the neural network.

1.5.1. Embedded Neural Network for Fire Classification Using an Array of Gas Sensors

Bashyal et al. at Missouri University of Science and Technology created an embedded neural network for fire classification [66]. This application does not require the system to be able to continuously process data in real-time since once a fire is detected and classified, its work is finished. Even if the calculations required several seconds, this is acceptable for this application. This network is very large relative to the simplicity of the problem to be solved. It has seven inputs and three outputs with the configuration shown in Figure 11.

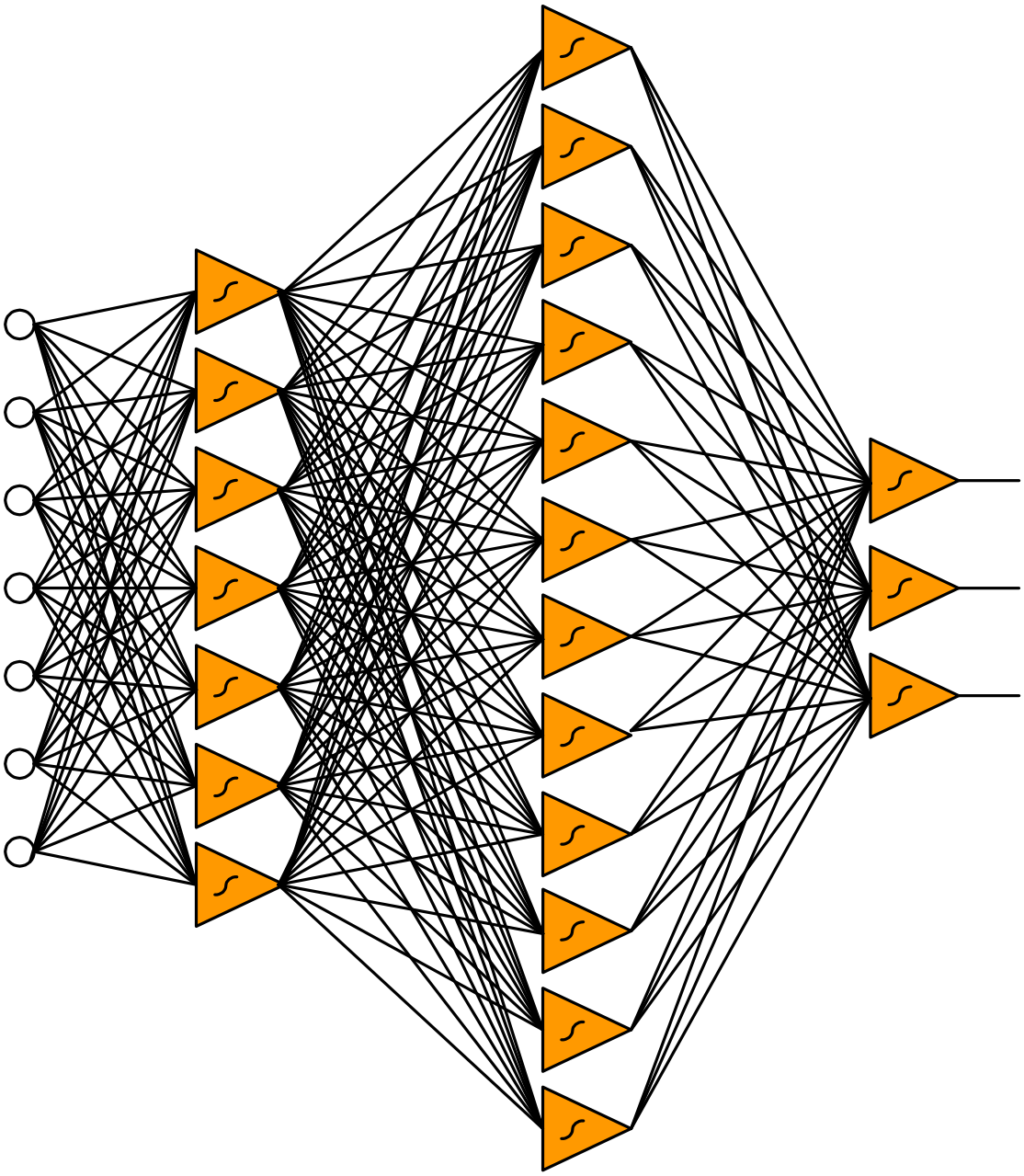

Figure 11: Network used by Bashyal et al. in "Embedded Neural Network for Fire Classification Using an Array of Gas Sensors".

Each of the seven inputs is one individual sensor and the three outputs represent the three types of fires to be classified: No Fire, Class A, and Class B. This work appears

to questionable due to the data used for training the network. The table containing the training data displays the seven sensor readings for different types of materials burning. Of the seven sensors, however, only one sensor is needed to classify the difference between Class A and Class B fires. The other six inputs become irrelevant. Based on the this data, a neural network of this size is not needed to analyze the data. The sensor in the far right column accurately distinguishes the differences between the items burnt once the temperature level is elevated. The training set here is shown in Table 1.

NORMALIZED INPUT READINGS USED FOR ENN TRAINING

Sample	Temp	TGS	TGS	TGS	TGS	TGS	TGS
Paper	0.72	0.14	0.72	0.63	0.52	0.20	0.12
Paper	0.67	0.06	0.66	0.60	0.55	0.20	0.30
Petrol	0.69	0.10	0.69	0.52	0.69	0.18	0.09
Petrol	0.56	0.24	0.85	0.72	0.74	0.40	0.07
Plastic	0.67	0.09	0.38	0.39	0.35	0.09	0.13
Plastic	0.67	0.01	0.43	0.56	0.50	0.13	0.16
Kerosene	0.60	0.13	0.74	0.67	0.70	0.13	0.09
Kerosene	0.63	0.19	0.78	0.66	0.63	0.14	0.08
Normal	0.58	0.02	0.10	0.12	0.09	0.05	0.08
Wood	0.73	0.13	0.60	0.60	0.66	0.23	0.29
Wood	0.69	0.13	0.55	0.70	0.73	0.28	0.39

Table 1: Training data used by Bashyal et al. in "Embedded Neural Network for Fire Classification Using an Array of Gas Sensors". The authors labels the six sensor readings TGS but do not number each sensor.

Looking at the training data of Bashyal et al., it can be assumed that most of the neurons in the network shown in Figure 11 are not actually contributing to the correct classification of the fires. It seems that the designer is unaware of their system, so it can

be very difficult to detect whether the entire network is working or a small number of neurons are carrying the load. The network configuration [66] is inefficient for this dataset because a neural network will train to the simplest distinguishing characteristic; in this case it was the last sensor and the temperature sensor and the other data need not be used.

Bashyal et al. also only used integers and not fractional math which is a limiting factor of the work. The authors used Error Back Propagation (EBP) for training, which they admit is another limitation of the system. The authors also mention that the embedded network was manually converted from the computer based network to an embedded network and that it would be optimal to have an automated system for this implementation. This dissertation addresses all unsolved issues of [66]: arbitrary architectures, training algorithms, and completely automated embedded implementation.

1.5.2. Microcontroller Based Neural Network Controlled Low Cost Autonomous Vehicle

Farooq et al. from the University of The PunJAb Lahore in Pakistan developed a neural network to control a model remote control car [79]. The neural network had three sensor inputs and four outputs for the motor control. The inputs have only two bits of precision for a sonar sensor. The network is shown in Figure 12 and again this network is much larger than needed for this problem. This network should not require as many neurons in the hidden layer. Since the author trained with EBP, more neurons were required to solve the same problem because of the limitations of the first order gradient approach for minimizing the error. If the authors had used arbitrarily connected networks

and trained with a second order algorithm, this network could have been greatly reduced in size possibly to a point of completely removing the hidden layer neurons entirely.

The authors of [79] are not using the neural network to its full potential as discussed previously because all possible input and output patterns are trained. In this case, the model car can only do one of four possible directions: Back, Forward, Right, or Left. This system could be redesigned so that there are only two output neurons, one for each motor. This way, the car's output can be analog, based on simple inputs. Ideally, the input sensors should be analog or at least high resolution digital inputs; for example 8-bits instead of two. The network could then be trained with sample patterns like the digital ones used for their example and allow the network to extrapolate the data between points. This would create a smooth turning car opposed to very rigid all-or-nothing movement.

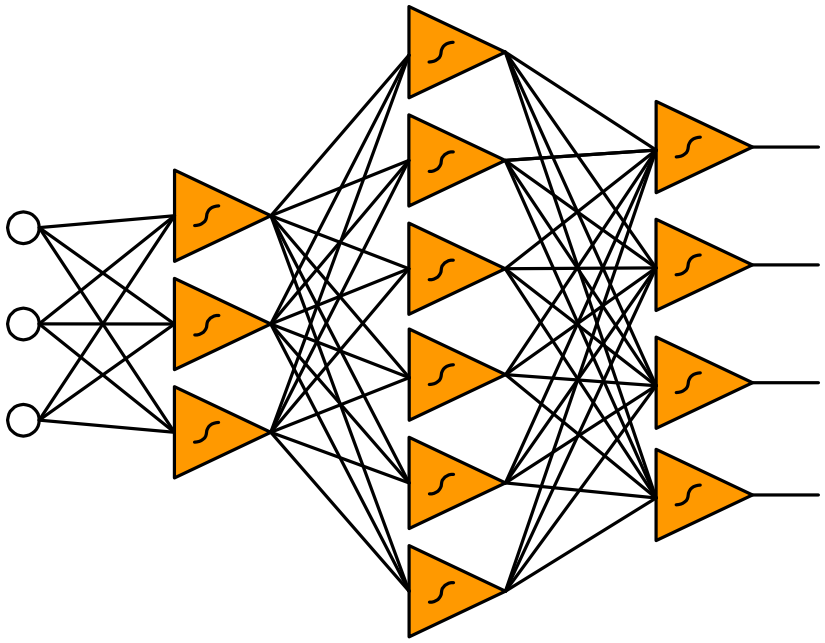

Figure 12: Neural network from Farooq's" Microcontroller based Neural Network Controlled Low Cost Autonomous Vehicle".

Farooq et al. also used a piecewise linear approximation of *tanh* for his activation function. A similar activation function can be seen in Figure 6. This activation function is very simple to calculate but it degrades the accuracy of the neural network, especially with more layers because each error is amplified by the next layer.

Overall their work did not implement a reasonable application of a neural network that is used to its fullest potential. The outputs are rounded to such a broad 4-bit result that it would not be difficult to achieve with many types of control systems.

1.5.3. Control Sensor Linearization Using a Microcontroller-Based Neural Network

Dempsey et al. from Bradley University in Peoria, IL created an embedded neural network used for sensor linearization [77]. This application is a much more practical use of neural networks because it utilizes the nonlinear properties of the network. However, it is a very simple MLP network trained with EBP. The author optimized the architecture and eliminated weights that were approximately zero to decrease the number of calculations for the processor. The network is shown in Figure 13. The authors of [77] even specify that the entire network is simplified to five multiplications and three additions. The activation function used is a lookup table so the calculations on the microcontroller are very limited and are acceptable for this specific network and application. However, this work cannot be translated to general applications because the network is customized and simplified for this application. This work is also done on an 8-bit microcontroller, the Intel 87C51, which is slightly more sophisticated than the one used in this dissertation because it contains a hardware divider.

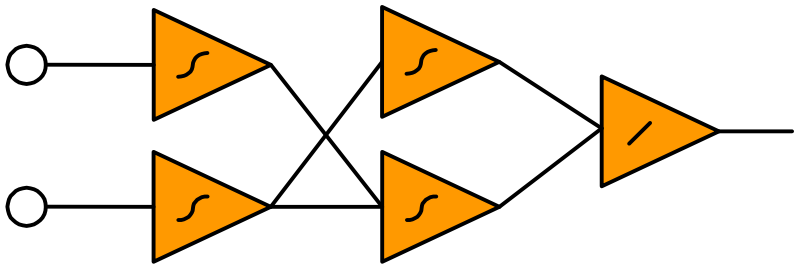

Figure 13: Neural network for "Control Sensor Linearization Using a Microcontroller-Based Neural Network" by Dempsey et al.

Dempsey et al. discusses how the neural network degrades some of the characteristics of the controller they were using. They attributes these errors to several things. First, the errors are rounded to one decimal place which makes the network a crude approximation of a potentially very accurate control device. Secondly, they contribute to the error by using a lookup table for the activation function. The final significant loss of data is the truncation of intermediate calculations of the neural network.

All of the errors that Dempsey et al. note in [77] are addressed in this dissertation. The second order *tanh* activation approximation is more accurate than the lookup table. The use of 16-bit and 32-bit pseudo floating point numbers to prevent truncation at intermediate calculations increase accuracy. Also, the storing of 16-bit pseudo floating point weights drastically reduces the round-off error created by their design. This will come at a cost of slightly more calculations, but in a very similar calculation time based on a slightly faster clock speed.

1.5.4. A Solar-powered Battery Charger with Neural Network Maximum Power Point Tracking Implemented on a Low-Cost PIC-microcontroller

Petchjatuporn et al. from the University of Technology in Thailand used a small neural network to control the power point on a solar charger[64]. This is a great application of an embedded neural network because it is a simple but very nonlinear control problem. However, the author's approach can be improved in a few ways. First, the activation function is a piecewise function with two saturation points at the limits but

then in the "linear" region, as the author describes, the output is simply the input. The activation function does not require calculations between 0 and 1. This heavily limits the neural network because the network gets its nonlinearity from the activation function, and, in this case, that nonlinearity is removed. The next problem with this network is the architecture and training of the network. The network can be seen in Figure 14.

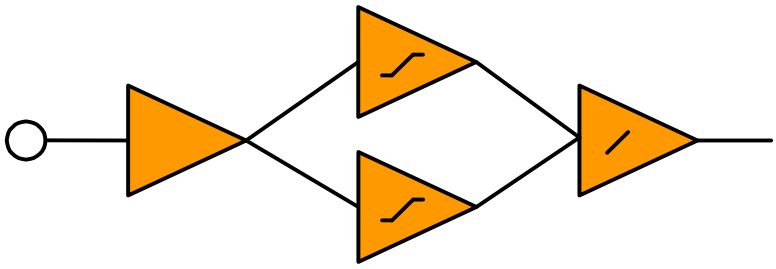

Figure 14: Neural Network implemented in Petchjatuporn's work on "A Solar-powered Battery Charger with Neural Network Maximum Power Point Tracking Implemented on a Low-Cost PIC-microntroller".

The input neuron really is not a neuron at all; it is a specific power calculation based on duty cycle shown in Equation 4.

$$f_i = Wi1 \frac{\Delta \mathrm{P(n)}}{\Delta D(n-1)} \tag{4}$$

The weights for the entire network are designed and actually remove the nonlinear properties of the network. In the first layer, they are +1 and -1. In the second layer, the weights are chosen to make the middle layer linear between -1 and +1 without the use of an activation function. This network could be more powerful if it were trained. The purpose of neural networks is to use nonlinearities to the designer's advantage and train them to solve the application needed. In this case, the author simply linearized the

network and designed it to do what was desired. However, in doing so, this defeats the purpose of using a neural network. This application could also benefit from using an arbitrarily connected network and removing at least one neuron if not more while obtaining better results. The application warrants a neural network but the implementation used prevents the network from operating at its full potential.

1.6. Current Research Summary

The current research on embedded neural networks in low end microcontrollers does not fully utilize the neural networks to their potential. Most of the work modifies the neural networks to simplify calculations to make them easier to implement, or simplifies the data to make their training work easier. The networks are all converted manually from a PC based network to the embedded network. None of the current work uses arbitrarily connected neural networks but instead only MLP networks. All of the previously published networks that are trained are trained using EBP which greatly limits the ability to train the networks. These neural network implementations used integer math or single digit fixed point math to achieve their results.

This dissertation addresses most of these shortcomings of the listed work. It first offers a wide variety of training algorithms for training neural networks, including first and second order algorithms. An automated system was created to convert the newly trained network to an embedded network. The embedded networks are all configured using any feed-forward architecture. The embedded neural network also uses a pseudo floating point algorithm for exceptional accuracy on a limited system. In addition to the

pseudo floating point mathematics is a second order approximation of *tanh.* This is a very accurate and nonlinear approximation to maintain the integrity of the nonlinear neural network. In addition to creating this specialized neural network for the PIC 18F series microcontroller, the software also generates a C version of the network that can be easily transferred to any microcontroller with a C compiler. This will only require minor changes to such things as headers and initial addressing.

2. Neural Network Training

The Neural Network Trainer (NNT) was originally developed as a tool for training neural networks for use on a PC or comparable computing machine. NNT originally produced for the user an array of weights that corresponded to the weights in a neural network architecture designed by that user. From this point, it is was the user's responsibility to create a neural network that could utilize these weights [80]. This dissertation transforms this original tool into a complete neural network implementation package for microcontrollers. This software package includes the trainer, an assembly language based generic neural network for the PIC 18 series microcontroller, 8-bit neural network simulator, a microcontroller communication interface for testing embedded neural networks, and a C implemented neural network for any microcontroller with a C compiler. These features work together to create a total package that can be used not only to implement a neural network for one case but a very wide variety of neural network applications. This software is the only published embedded neural network software that uses arbitrarily connected neural networks.

In addition, this software allows the user to create, train, test, and implement a neural network on a microcontroller for his purpose in an automated process. The tools, steps, and details of the process will be discussed in the sections following.

2.1. Neural Network Trainer

The user interface for the Neural Network Trainer is shown in Figure 15. The user will first notice there is an empty plot on the left side of the trainer where the iterations versus means squared error are displayed as well as training parameters on the right hand side. The user must follow a few simple steps before training a network. He must prepare an input file that contains the training data and an architecture file that describes the network connections, and then set the training parameters.

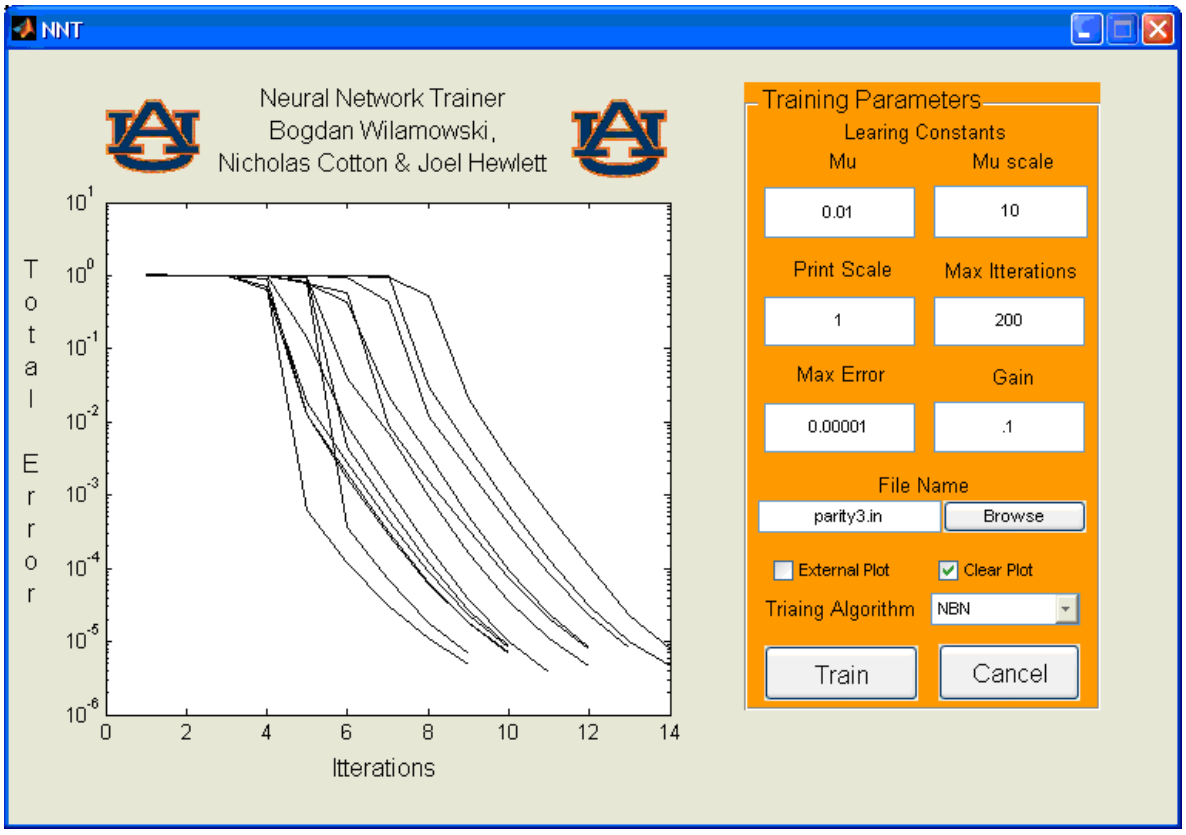

Figure 15: Front end of Neural Network Trainer (NNT)

2.1.1. Training Data

The user must create a training file with all the data sets required to train the neural network. This data may be created in various ways such as by hand, spreadsheet, or directly through Matlab. A simple parity-3 problem will be used for demonstration purposes. This demonstration will use bipolar neurons so the extremes for data will be +1 and -1. The training data for parity-3 is represented by the following matrix:

In_1	In_2	In_3	Out_1
-1	-1	-1	-1
-1	-1	1	1
-1	1	-1	1
1	-1	-1	-1
1	-1	1	1
1	1	-1	-1
1	1	1	1

As with any parity-N problem there are 2^N possible outcomes. As the top row indicates the first three columns are the inputs and the last column is the output for that row. The top row of the matrix is for demonstration purposes only but is not needed in the actual data file. This data is then copied to a text file and saved with the file extension *.dat*. Delimiters other than white space are not required. Once the data file is finished it can be referenced by numerous architecture input files.

2.1.2. Input File

The input file contains the network architecture, neuron models, data file reference, and optional initial weights. Each input file will be unique to a specific architecture but not necessarily to each data set. In other words, the same data set can be used for several different architectures simply by creating a new input file. The input file contains 3 sections: the architecture, model parameters, and data file definition. The following is an example of an input file for the parity-3 problem discussed in the previous section.

```
\\ Parity-3 input file (parity3.in)
n 4 mbip 1 2 3
n 5 mbip 1 2 3
n 6 mbip 1 2 3 4 5

W   5.17  20.08 -10.01   -4.23
W   1.0   10.81   2.20   19.84

.model mbip fun=bip, der=0.01
.model mu   fun=uni, der=0.01
.model mlin fun=lin, der=0.05

datafile=parity3.dat
```

The first line is a comment. Either a double backslash, as in C, or a percent sign, as in Matlab, is acceptable as a comment delimiter. After the comment comes the network architecture for a 3-neuron fully-connected network as shown in Figure 16.

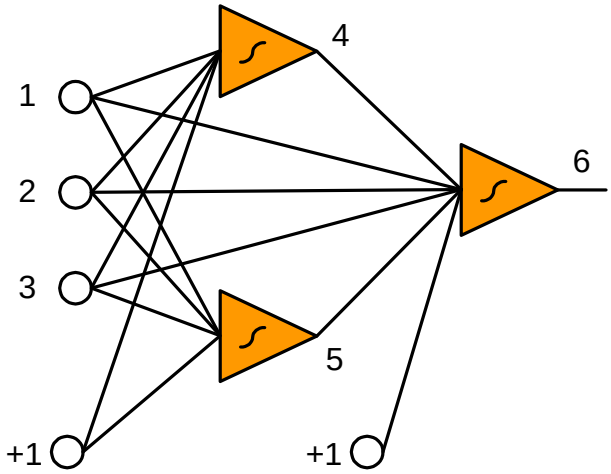

Figure 16: Three Neuron architecture for parity-3 problem.

The neurons are listed in a net list type of layout that is very similar to a SPICE program. This way of listing the layout is node based. The first nodes are reserved for the input nodes. The first character of the line is an *N* to signify that this line describes a neuron. The *N* is followed by the neuron output node number. Looking at Figure 16, the first neuron is neuron 4 because it is the first available number after the three inputs, and it is connected to nodes 1, 2, and 3, which are inputs. The same is true for neuron 5 or the second neuron, which is also connected to all three inputs. The output node is slightly different but it follows the same concept. It is connected to all three inputs as well as to the output of the first two neurons. Based on this it should be straightforward to see the connection between the input file listed above and Figure 16.

Also, listed on the line of each neuron is the model of the neuron, which allows the user to specify a unique model for each neuron. This network is designed to solve a parity-3 problem using three bipolar neurons. This is not the minimal architecture for

this problem, but it serves as a good demonstration of the tool.

Following the architecture of the network are the optional starting weights. If no starting weights are given the trainer will choose random weights. The weights need to be listed in the same format as the architecture. Each line of weights starts with the capital letter *W*. The biasing weight goes in place of the output node of the neuron. In other words, the first weight listed for a particular neuron is the biasing weight followed by the remaining input weights in their respective order. See the input file for an example.

The user specifies a model for each neuron and these models are defined on a single line. The user has the ability to specify the activation function and neuron type (unipolar, bipolar or linear) for each model. The user may include neurons with different activation functions in the same network. The final line of the input file includes a reference to the data file. This line simply needs to read *datafile* followed by the file name. In this example, it is *parity3.dat* which can be seen on the last line of the example input file.

2.1.3. Training Parameters

Once the network architecture has been decided and the input files created the next step is to select the training algorithm and parameters. When NNT is loaded there is an orange panel full of adjustable parameters on the right side of the window. These parameters change for each algorithm so they will be addressed accordingly in the following section. There are several independent algorithms that can be used for training neural networks. The algorithms themselves are explained in more detail in[80], but first the user should select the input file just created.

Implemented algorithms

The algorithm is chosen from the pull down menu in the training parameters. Four of the parameters are the same for all algorithms. They are: Print Scale, Max. Iterations, Max. Error, and Gain. The Print Scale refers to how often the mean squared error is printed to the Matlab command window. This can be important because in certain situations the longest calculation time is that of displaying the data, so increasing this number can significantly decrease training time. Max iterations is the number of times the algorithm will attempt to solve the problem before it is considered a failure. An iteration is defined as one adjustment of the weights, which includes calculating the error for every training pattern and adjusting at the end. The Max Error is the mean squared error that the user considers to be an acceptable value. When this number is reached the algorithm stops calculating and displays the final weights.

Error Back Propagation (EBP)

This algorithm is the traditional EBP with the ability to handle fully connected neural networks. The Alpha parameter is the learning constant. This value is a multiplier that acts as the numerical value of the step size in the direction of the gradient. If alpha is too big the algorithm can oscillate instead of reducing the error. However, if alpha is too small the algorithm can move toward the solution too slowly and prematurely level off. This parameter should be adjusted by the user until an optimal value is found which has some oscillation that diminishes while the error continues to decrease.

Neuron By Neuron (NBN)

NBN is a modified Levenberg-Marquardt algorithm [81] for arbitrarily connected neural networks. The NBN algorithm is briefly described in [58]. It has two training

parameters, μ and μ Scale. The learning parameter of the LM algorithm is μ. Its use can be seen in Equation 5. In Equation 5 the *w* describes the weights, the *J* is the Jacobian matrix, and the *I* is the identity matrix.

$$w_{k+1} = W_k - \left(J_k^T J_k + \mu\, I\right)^{-1} J_k^T e \tag{5}$$

If $\mu = 0$ then the algorithm becomes the Gauss-Newton method. For very large values of μ the algorithm becomes the steepest descent method or EBP. The μ parameter is automatically adjusted at each iteration to insure convergence. The amount it is adjusted each time is μ Scale which is the last parameter for the NBN algorithm.

Self Aware (SA)

The SA algorithm is a modification of NBN. It evaluates the progression of the algorithm's training and determines if the algorithm is failing to converge. If the algorithm begins to fail, the weights are reset and another trial is attempted. In this situation the program displays its progress to the user as a dotted red line on the display and begins again. The algorithm continues to attempt to solve the problem until either it is successful or the user cancels the process. The SA algorithm uses the same training parameters as NBN.

Enhanced Self Aware algorithm (ESA)

ESA is also a modification of the NBN algorithm and is used in order to increase chances for convergence. The modification was made to the Jacobian matrix in order to allow the algorithm to be much more successful in solving very difficult problems with

deep local minima. The algorithm also is aware of its current solving status and will reset when necessary.

The ESA algorithm uses a fixed value of 10 for the μ Scale parameter and allows the user to adjust the LM parameter. The LM parameter is essentially a scale factor applied to the Jacobian matrix before it is used in calculating the weight adjustment. This scale factor is typically a positive number between 1 and 10 or possibly greater. The more local minima the problem has, the larger the LM factor should be.

Forward-Enhanced Self Aware (F-ESA)

F-ESA is another modification of the NBN algorithm developed by J. Hewlett [57] where an alternative method for calculating the Jacobian matrix is used. The calculation of Jacobian is unique in the sense that only feed-forward calculations are needed. This approach is then paired with the Enhanced Self-Aware LM algorithm. The F-ESA algorithm uses the same training parameters as the ESA algorithm.

Evolutionary Gradient

Evolutionary Gradient is a newly developed algorithm, which evaluates gradients from randomly generated weight sets and uses gradient information to generate a new population of weights. This is a hybrid algorithm which combines the use of random populations with an approximated gradient approach. Like standard methods of evolutionary computation, the algorithm is better suited for avoiding local minima when compared to common gradient methods such as EBP. What sets the method apart is the use of an approximated gradient which is calculated with each population. By generating successive populations in the gradient direction, the algorithm is able to converge much

faster than other forms of evolutionary computation. This combination of gradient and evolutionary methods essentially offers the best of both worlds. The training parameters are very different than the LM based algorithms previously discussed. They include Alpha, Beta, Min. Radius, Max Radius, and Population. This algorithm was written by Joel Hewlett and details regarding these parameters may be found in [82].

Training

Once the user selects the appropriate training algorithm the parameter boxes will change to the corresponding parameters and default values will fill the boxes. After the user sets the parameters, there are two other boxes that can be selected. The clear plot box when checked will overwrite any existing plot with the new one, but if it is left unchecked then the subsequent plots will be drawn on the axis with all of the previous drawings. The last option is the external plot which draws the plots inside NNT and in a separate figure allowing for easy printing or modifying the plot. The train button begins the training process, which prints the error to the Matlab Command Window as it is training. At any time the process can be halted and the results plotted by pressing the cancel button.

2.2. NNT Adaptations

The trainer was adapted to aid in the process of creating neural networks on the embedded level. NNT trains the neural network as it would any neural network and then the embedded network verifications begins. The trainer then makes a forward calculation on the network using the 8-bit neural network simulator, which will be described in more

detail in section 2.2.2. It essentially does all of the arithmetic that the neural network would do for one pattern. At every step of the way it rounds all of the digits in the same way the 8-bit microcontroller does. This calculation is performed as a sanity check and debugging step for the system. Step by step results from beginning to end of the network calculation are stored in hex and decimal format in an organized text file for the user.

After the training process, the trainer generates the weights, the architecture, and other parameters into assembly and C files for microcontroller implementation. These files can be directly copied and pasted into the microcontroller IDE and then immediately assembled or compiled, respectively. These files will be discussed in Section 2.2.3.

The trained and verified network can then be further tested on the embedded level using the neural network communication software. This software is used to communicate via a serial port with the microcontroller. This allows the user to simulate data that the neural network would receive from an external source, like an analog to digital converter. The microcontroller then performs the network forward calculation and sends the data back through the serial port for verification, simulating a network output such as a value for a pulse width modulation module. At this step the user can test as many test patterns as necessary to validate proper performance with hardware in the loop simulation. This software's features will be discussed in Section 2.2.2.

2.2.1. Neural Network Weight Scaling

The assembly language version of the neural network implementation uses a custom pseudo floating point algorithm. This algorithm requires a weight scaling process to be performed off chip. This process allows for the largest number of significant digits

to be used for each neuron, and this process scales all the weights for a particular neuron to use the maximum number of digits possible. This scale factor is then saved as an attribute for the particular neuron. However this process is completed as another automated section before generating the assembly file. The details of this scaling process are shown in the following code.

```
scale=ones(1,nn)
for i=1:nn    % Weights for each neuron.
    w=ww(iw(i):iw(i+1)-1);
    max_value=max(w);
    if max_value>=128
        while max(w)>127
            w=w/2;
            scale(i)=scale(i)/2;
        end
    else
        while (max(w))<63.5
            w=w*2;
            scale(i)=scale(i)*2;
        end
    end
```

The previous code shows that the largest weight is scaled to be as close to, but not exceeding 127 which is the largest positive number that can be represented using this protocol. As a consequence of the scaling the largest weight uses almost all of the 16-bits of the mantissa. These scaled weights are then the weights used for generating the assembly file.

2.2.2. 8-Bit Neural Network Simulator

The simulator is written in Matlab to operate in the same fashion as an 8-bit microcontroller. The simulator introduces rounding errors in the appropriate places to function in the same manner as the microcontroller. This was accomplished by creating a

set of functions that operate identically to the PIC microcontroller. These instructions include commands that round, multiply, add, subtract, and perform the *tanh* approximation, all using the pseudo floating point arithmetic. There are also special instructions to detect any overflows.

One example, *rnd8bit* function, takes any decimal number and rounds it to 8-bits of fractional data. The Matlab code can be seen below.

```
function y=rnd8bit(x)
y=fix(256*x)/256;
return
```

The *rnd8bit* function operates by shifting the fraction portion of the number into the integer portion by 8-bits, truncating the fractional part, and then shifting the bits back into place. This step is done any time a Matlab command is used that could possibly generate more decimal digits.

Another example of how the simulator works is a routine setup to multiply in the same fashion as the microcontroller. This function is called *mul* and is shown below.

```
function p=mul(x,y)

x1=x;
y1=y;

if x>=128 | y>=128 ;
   disp(x);
   disp(y);
   error('x=%d and y= %d Overflow!!!',x,y);
end ;

if x<0 & x>-128 ; x1=256-abs(x); end
if y<0 & y>-128 ; y1=256-abs(y); end

x1=rnd8bit(x1);
y1=rnd8bit(y1);
```

```
x1;
y1;

a=floor(x1);       %whole number
b=x1-a  ;          %fractional portion
b=b*256;

c=floor(y1);
d=y1-c;
d=d*256;

p=(a*c*256^2+256*(a*d+b*c)+b*d)/256^2;
p=floor(p*256^2)/256^2;

 if x<0
     p=p-c*256-d;
 end
 if y<0
     p=p-a*256-b;
 end
 if p<-16384
     p=p+65536;
 end
 p=floor(256^2*p)/256^2;
 return
```

This function, first of all, checks to make sure that neither of the two parameters to be multiplied is outside the range of valid numbers. This check is redundant and probably not necessary but the extra layer of protection against overflow errors was desired. If either number is outside the range the function displays the values and halts the process. Next the 8-bit multiplication is completed in the same manner as on the microcontroller. This process only operates on positive numbers and then converts a negative result back to two's-complement at the end. The final and intermediate results are rounded to ensure accurate introduction of error. Several other custom functions were necessary for the simulator as well as the creating of the assembly and C files.

Matlab does not have a way of converting decimal numbers to hexadecimal. In order to work with both hex and base ten numbers, functions were required to go between them. One of these functions is the *frac2hex* function that can be seen below.

```
function out=frac2hex (x)

    if x==0; out='0000'; return; end;

    if x>0
        conv=Fr_dec2bin(x);
        conv=num2str(conv);
        [whole,frac]=strtok(conv,'.');
        whole=dec2hex(bin2dec(whole));
        frac=strcat(frac,'00000000');
        frac=dec2hex(bin2dec(frac(2:9)));
    else
        x=abs(x);
        x=256-x;
        conv=Fr_dec2bin(x);
        conv=num2str(conv);
        [whole,frac]=strtok(conv,'.');
        whole=dec2hex(bin2dec(whole));

        frac=strcat(frac,'00000000');
        frac=dec2hex(bin2dec(frac(2:9)));

    end

    if size(whole,2)==1
        whole(2)=whole;
        whole(1)='0';
    end

    if size(frac,2)==1
        frac(2)=frac;
        frac(1)='0';
    end

    out=strcat(whole,frac);
```

This process requires several steps because the input is a decimal number that is first converted to an integer and a fraction. Next, the two pieces are operated on separately by using some of Matlab's built in functions. It was effective to convert to binary and then to hex from binary. This function has to take into account positive as well as negative numbers. The output is a four character string of hexadecimal numbers and letters to represent the 16-bit fractional number.

These custom instructions are just a few of many required to make the simulator function properly. These functions were written and verified before the assembly code for the microcontroller was started. The assembly code was written step by step to follow each process of the Matlab code. The two processes were then tested and debugged until their results matched for all test cases. This was the only feasible way to construct a system of this size. The assembly code is approximately 1500 lines of code or 30 pages. The code had to be written in sections that could be tested individually and based on something that could verify each piece independent of the rest of the system.

2.2.3. Generated Files

The trainer was adapted to automatically generate an assembly and C file for the user to use for implementing the system. This was motivated by two main factors. Automating the process removed the ability to add human error when converting the weights to hexadecimal numbers and also the process simply took too long to do by hand. An example of the output file for the network previously discussed in Figure 16 can be seen below.

```
;IO data: #Inputs, #Outputs
IO Data 0x0003, 0x0001
Weights Data 0x000B
Data  0x000A, 0x9E6A, 0x6196, 0x9E6A, 0xFFEF, 0x9F4B,
0x60B5, 0x9F4B, 0x001A, 0xC969, 0x5B66

;Inputs Data 0x03, 0x4000, 0xC000, 0xC000, 0x06,
;The output for this particular input should be 1 and
0x0100

Neurons Data 0x03
Data  0x05, 0x03, 0x04, 0x01, 0x02, 0x03
Data  0x08, 0x03, 0x05, 0x01, 0x02, 0x03
Data  0x03, 0x02, 0x06, 0x04, 0x05
```

The assembly file is made up of three parts. The first part is a few general network parameters and the scaled weights. The first line is a comment to clarify for the user what is shown. The second line is the number of inputs followed by the number of outputs for the network. The next line is the number of weights followed by the weights themselves. The weights are stored in their scaled form. This means all the weights for a particular neuron are scaled up or down appropriately to use the maximum number of bits possible. This scale factor is then stored in the topography section for each neuron. The next two lines are comments and are simply for the user as a sanity check. They show random inputs to the system and the correct output for those particular inputs. This way the user always has a valid network input and output pair available. The last section of the file is the neural network architecture and is actually read from program memory by the system as needed. These values act as indexes of which location in the architecture array to use next.

This file is specifically designed to be placed at the end of the microcontroller code and is automatically read using indirect addressing. This file will change based on the

network being used but it will always follow this basic format. This file is all that is needed to change neural network architectures or weights. A similar file is generated in C and can be seen below.

```
rom far float const ww[11] = { 5.0967435373381891e+000,
4.1913570939474143e+000, -3.3389689751411735e+000,
3.3500760230887412e+000,
 -1.3919747593993528e+000, -3.1527033887270779e+000,
3.0337819562494110e+000, -3.1133953328418693e+000,
1.7645099960474910e+000,
 -3.2695061817728570e+000, -2.0549280802763907e+000 };

unsigned char const ni = 3;

unsigned char const topo[14]={ 3, 4, 1, 2, 3,
3, 5, 1, 2, 3,
2, 6, 4, 5};

float nodes[7];

unsigned char nn=3;
```

The C file contains very similar data as the assembly file including the network inputs, outputs, weights and architecture. The C file goes at the top of the code and basically is the initialization for the neural network. It prepares the arrays and sets all of the indexes and allows everything to be read as needed. This C code is mostly platform independent except for the very first line. Depending on the amount of available ram, the weights will most likely need to be stored in program memory and accessed one at a time. This memory initialization line may vary from platform to platform.

2.3. PIC Simulator Software (PicSim)

The simulator and verification tool designed was just as important to this overall project as the microcontroller implementation itself. The simulator engine was previously described but this section will discuss how that engine is interfaced with the user as well as the microcontroller. The PicSim software was generated before the assembly version to verify that is was possible to use the 8-bit math and obtain useable results. The software user interface is shown in Figure 17.

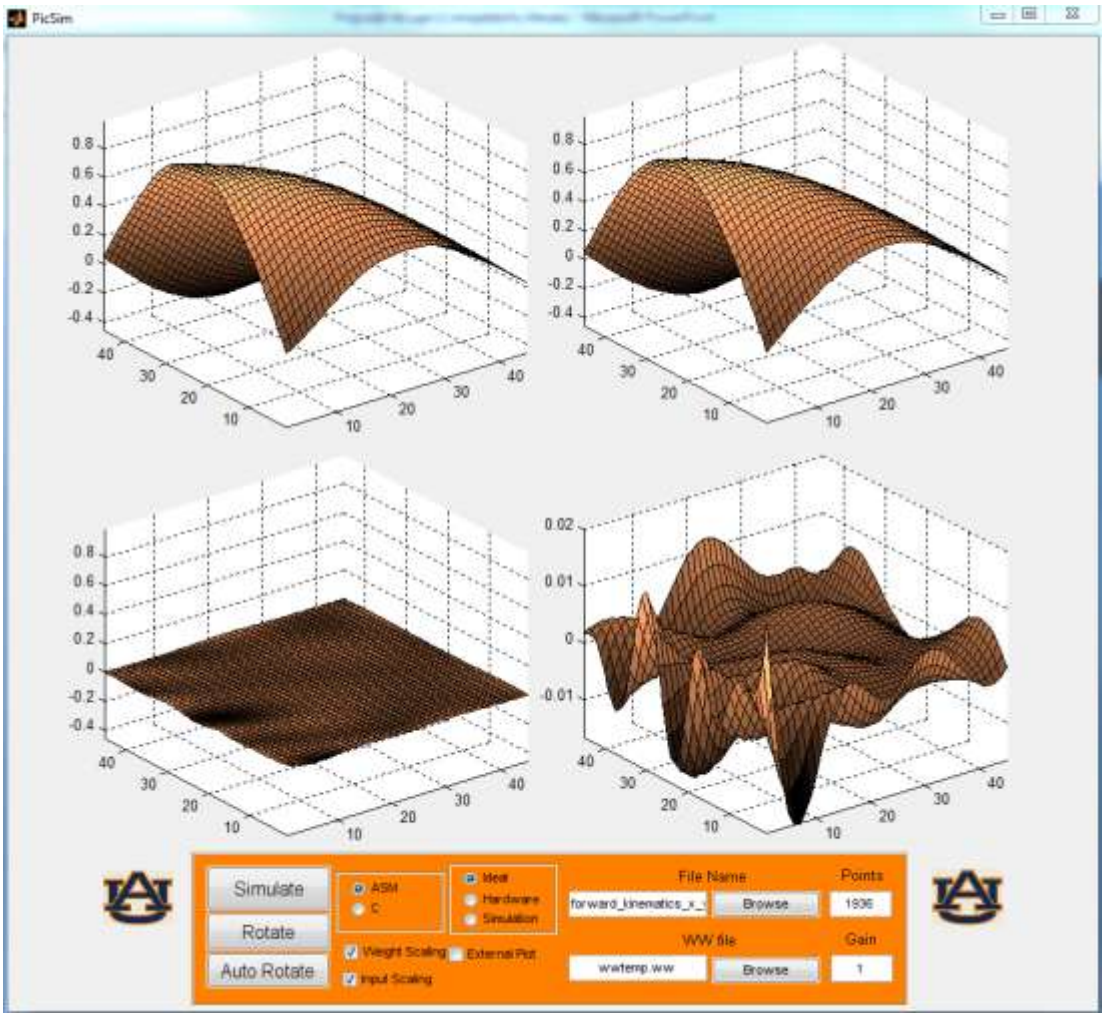

Figure 17: PicSim software for simulating and verify embedded neural networks.

The software is designed to plot a two-input system with one or more outputs. PicSim has two main input requirements: a network architecture and a weight array text file. To simplify the process for the user PicSim reads the same input file used by NNT for the training process and the weight file generated by NNT. The user simply has to point the software to these files. There are four graphs displayed in the user interface. The figures in the top left corner is always the network being used as the reference and the top right is the network being calculated. The bottom two figures are the differences between the top two surfaces. The one on the left is on the same scale as the top two and the one on the right is a tighter axis to show the more specific location of the error.

The user has a few other options as far as what type of network to simulate. The first option is the ideal neural network, which is a neural network on a PC using standard IEEE 754 floating point precision. This allows the user to compare the quality of the trained network to that of the training patterns before any error from the microcontroller is introduced. The user can simulate the error produced by the microcontroller by selecting the simulation button. This then compares the simulated network to the ideal network. The simulator engine discussed previously uses a configurable number of patters for testing. At this point any possible overflows or other errors should be caught before hardware is introduced.

The last option is the hardware in the loop setup. This option is the final stage of testing for the embedded neural network. It allows the user to program the microcontroller and test it but still use the features of Matlab for verifying the data.

Matlab still produces the test patterns and then sends them via the serial port to the microcontroller. This simulates data the microcontroller would gain from another source like the analog to digital converter. Once the embedded network has all of the inputs it needs it then does the neural network forward calculations and produces one or more outputs. These outputs would typically drive an external source such as a pulse width modulator, but in this instance are transmitted back to the PC via the serial port. This allows the user to send and receive data from the microcontroller in real time. In addition the user can verify the hardware calculations and the amount of time being required. The data can easily be verified by using Matlab's graphing tools. This mode can be used for embedded networks using assembly or C. The difference is the format of the test data being sent and received, but both operate under the same principle. This is the final step before the microcontroller is configured to operate in its embedded application with real inputs and outputs but at this point the neural network operation has been thoroughly verified.

The tools created to build the neural network on the microcontroller resulted in an equally challenging project as the embedded network. However, creating and debugging the assembly version of the neural network would never have been possible without the tools. Now with the automated system almost any trained network can be implemented on the microcontroller in a matter of seconds.

3. Hardware Implementation

Implementing neural networks on an 8-bit microcontroller with limited computing power presents several programming challenges. In order for the network to perform as quickly as possible, creating the software at the assembly level was chosen. Writing the software in assembly allows a level of customization that cannot be achieved with C. However, the need for hardware portability was also a motivating factor and a more generic C implementation was also created. It was also very important to manually manage the very limited amount of data memory. Several assembly routines were created with this purpose in mind. A pseudo floating point arithmetic protocol was created exclusively for neural network calculations along with a multiplication routine for multiplying large numbers. A *tanh* compatible activation function was also needed. The final procedure is capable of implementing any neural network architecture on a single operating platform. This robust base removes the need to modify the structure of the software to make network architecture changes.

3.1. Pseudo Floating Point

The first method was to use 16 bits to represent the weights, nodes, and inputs for the neural network. These 16-bits are all significant digits in this pseudo floating point protocol. The 16 bits consist of an 8-bit signed integer and an 8-bit fraction fractional part. The nonconventional part of this floating point routine is the way the exponent and

mantissa are stored. Essentially all sixteen bits are the mantissa and the exponent for the neuron is stored elsewhere. This has several advantages. It allows more significant digits for every weight using less memory. This pseudo floating point protocol is tailored directly to the needs of the neural network forward calculations. This solution requires the analysis of the weights of each neuron and scales them accordingly and assigns an exponent for the entire neuron. A similar process is used for the inputs so the entire range will share a single scale factor. This scaling is done off chip before programming in order to save valuable processing time on each and every forward calculation.

Scaling does two things, first it prevents overflow by keeping the numbers within operating regions, and secondly automatically filters out inactive weights. For example, if a neuron has weights that are several orders of magnitudes larger than others it will automatically round the smallest weights to zero. These weights being zero allow the calculations to be optimized, unlike using traditional floating point arithmetic. However, if all of the weights are the same magnitude they are all scaled to values that preserve maximum precision and significant digits. In other words, the weights are stored in a manner that minimizes error on a system with limited accuracy. Thus far, all of these decisions for scaling the weights are made before the network is programmed on the microcontroller. This process has been automated for ease of use. The Neural Network Trainer [8] was modified to automatically scale the weights and inputs after it trains the network. This is done in Matlab and an example of the scaling process can be seen below.

3.2. Multiplication

The Pic18F45J10 microcontroller has an 8-bit by 8-bit unsigned hardware multiplier. Considering that the hardware multiplier cannot handle floating point values or negative numbers, a routine was needed to allow fast multiplication of fractional values. The multiply routine is passed two sixteen bit numbers, consisting of an eight-bit integer and an eight bit fraction portion. The routine returns a 32-bit product. The result of the multiplication routine is a 32-bit fixed point result shown in Figure 18.

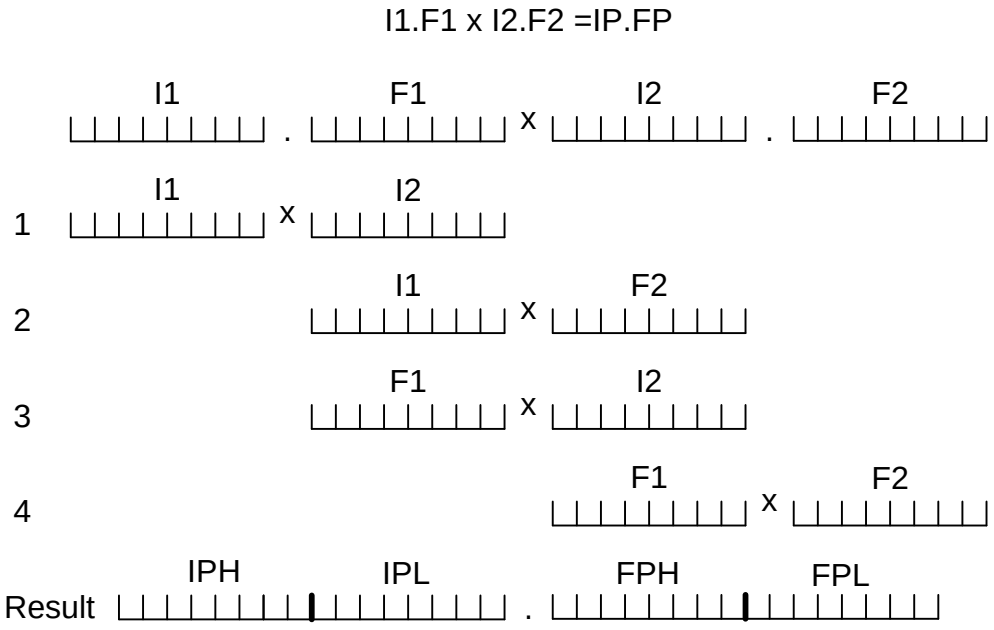

Figure 18: Implementation of 16-bit fixed point multiplication using 8-bit hardware multiplier. Steps 1-4 are summed with place holders to give the final product on the result line. Abbreviations: Integer (I) Fractional (F) Product (P) Lower-Byte (L) Higher-Byte (H).

Equation 6 shows the method of using a single 8-bit multiplier to implement 16-bit fixed point multiplication. The hardware multiplier does the multiplication between

bytes in a single instruction. The 16-bit result of I1 and I2 is placed in IPH and IPL, see Figure 18. Next I1 and I2 are multiplied and the lower byte is placed in IFL the higher byte of the product is added to IPL. F1 and I2 are then multiplied and the product is added to IPL and FPH respectively. Finally F1 and F2 are multiplied and the 16-bit product is summed with the current contents of FPH and FPL. At each step of this process when any of these 8-bit numbers are summed the carry bit must be added to the next most significant byte to maintain accuracy.

This method does not require any shifts or division. This simple process allows each neuron to quickly multiply the weights by the inputs and then use the 32-bit result as an accumulator for all inputs of the neuron. Once the net value is calculated only IPL and FPH are required for the activation function. If IPH is not zero or all ones (signifying a negative number) then the neuron is in saturation and the activation function immediately outputs a one or negative one.

$$\frac{A \cdot C \cdot 256^2 + 256 \cdot (A \cdot D + B \cdot C) + B \cdot D}{256^2} \tag{6}$$

3.3. Addition and Subtraction

Another issue arises when two numbers need to be added but they have different exponents; they must be first be converted to a common scale. This, however, is not necessary when using the proposed pseudo floating point protocol. The only summation that is required is for the calculation of the net value of each neuron. Even though there are two independent exponents these values will not change within one neuron, therefore

allowing all of the products to be summed in a single step. After the summation process is complete the reverse scaling can be done at the final stage. The details of this process will be discussed in more detail in section 3.6.

3.4. Activation Function

A soft activation function was needed for the neural network. The most common activation function is *tanh* and the definition is shown back in Equation 1. The pure definition *tanh* was not a reasonable solution for several reasons. Specifically, the exponents would be very difficult to calculate accurately with the limited hardware in a timely fashion. Likewise, the floating point division would have been far too time consuming without dedicated divide hardware. The next possible choice for an activation function was Elliott's function shown in Equation 3. This activation function was also rejected. The Elliot function does approach one hyperbolically but not at the same rate as *tanh* and therefore is not interchangeable. Networks with the Elliot approach are less powerful that those with *tanh*. This means the networks would have to be trained using the Elliott function, which was not desirable. The other pitfall with the Elliott function is that it requires division. Without dedicated hardware, division would be too slow of a process for the final solution.

A second order approximation of *tanh* was chosen for its accuracy as well as its simple arithmetic calculations. Several features were added to the activation function besides simply calculating a second order approximation of *tanh.* One of these features analyzes the inputs to the activation function and converts negative numbers to positive

numbers to make the internal calculations faster and reduce the number of values that must be stored in the lookup table. The sign is restored at the end of the activation function. Another feature is a check to see if the neuron is in saturation. In other words, make sure the net value is within a given range. In this case the second order approximation is skipped and the neuron is put into saturation. These features of the second order approximation can be seen in better detail in Figure 19.

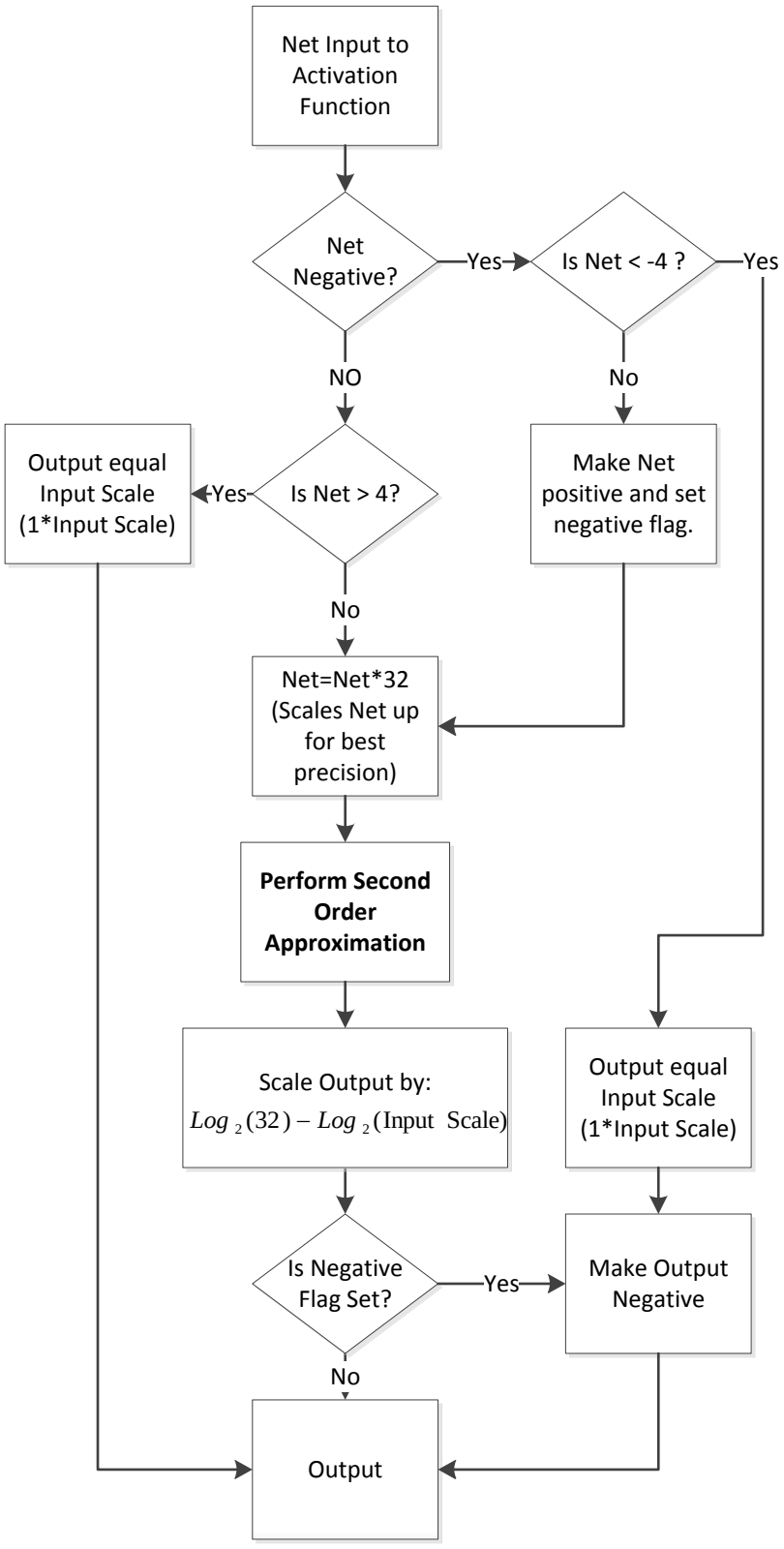

Figure 19: Logical block diagram of the activation function.

The routine requires that 30 values be stored in program memory. This is not simply a lookup table for *tanh* because a much more precise value is required. The *tanh* equivalent of 25 numbers between zero and four are stored. These numbers, which are the end points of the linear approximation, are rounded off to 16-bits of accuracy. Then a point between each pair from the linear approximation is stored. These points are the peaks of a second-order polynomial that crosses at the same points as the linear approximations. Based on the four most significant bits that are input into the activation function, a linear approximation of tangent hyperbolic is selected. The remaining bits of the number are used in the second-order polynomial. The coefficients for this polynomial were previously indexed by the integer value in the first step.

The approximation of *tanh* is calculated by reading the values of y_A, y_B and Δy from memory and then the first linear approximation is calculated using y_A and y_B.

$$y_1(x) = y_A + \frac{(y_B - y_A)\cdot x}{2\Delta x} \tag{7}$$

The next step is the second-order function that corrects most of the error that was introduced by the linearization of the tangent hyperbolic function.

$$y_2(x) = \frac{\Delta y}{\Delta x^2}\left(\Delta x^2 - (x - \Delta x)^2\right) \tag{8}$$

or

$$y_2(x) = \frac{\Delta y \cdot x \cdot (2\Delta x - x)}{\Delta x^2} \tag{9}$$

In order to utilize 8-bit hardware multiplication, the size of Δx was selected as 128. This way the division operation in both equations can be replaced by the right shift

operation. Calculation of y_1 requires one subtraction, one 8-bit multiplication, one shift right by 7 bits, and one addition. Calculation of y_2 requires one 8-bit subtraction, two 8-bit multiplications and shift right by 14-bits.

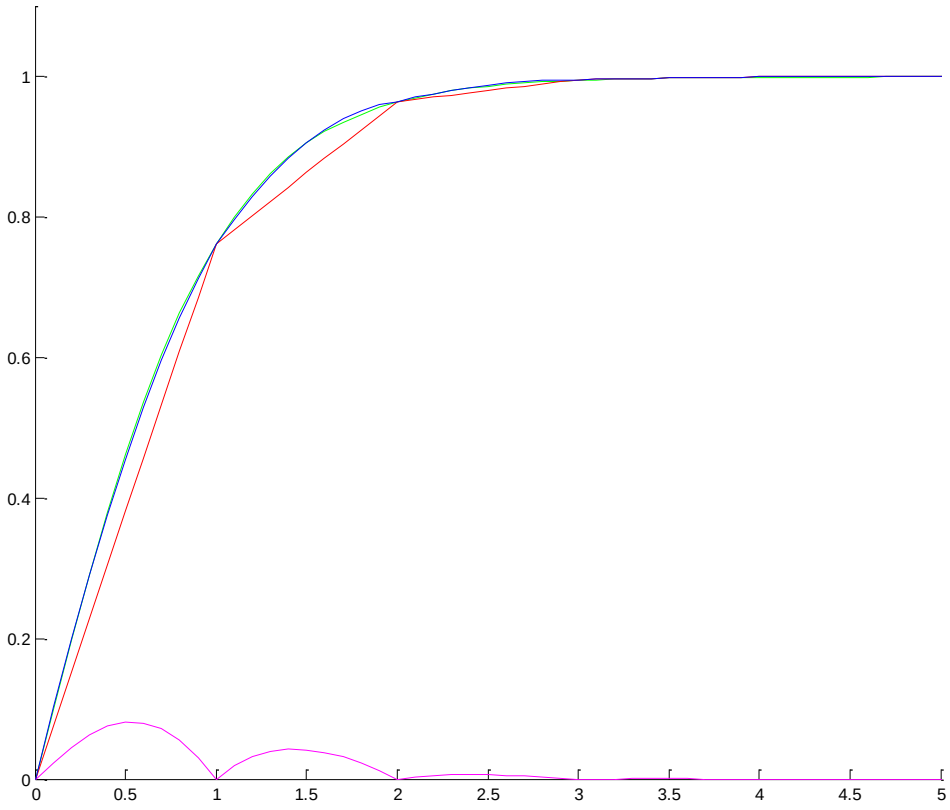

Figure 20: Example of linear approximations (red) and parabolas between 0 and 4 (magenta). *Tanh* (green) and the approximation (blue) are also shown on the graph. Only 4 divisions were used for demonstration purposes.

Ideally this activation function would work without any modification, but when the neurons are operating in the linear region (when the net values are between -1 and 1)

the activation function is not making full use of the available bits for calculating the outputs. This generates significant error. Similarly to the weights and the inputs, a work-around is used for the activation function. Pseudo floating point arithmetic is then incorporated. When the numbers are stored in the lookup table they are scaled by 32 because the largest number stored is 4. The net value is also scaled by 32 and if its magnitude is greater than 4, the activation function is skipped and a 1 or -1 is output. After multiplying two numbers that have been scaled, the product is shifted to remove the square of the scale. Once the activation function is finished the numbers are scaled back to the same factor that was used to scale the inputs.

The activation function was tested in hardware by sending a set of numbers from -5 to +5 and comparing them to the output of the *tanh* function. The difference between the sets of numbers can be seen in Figure 21.

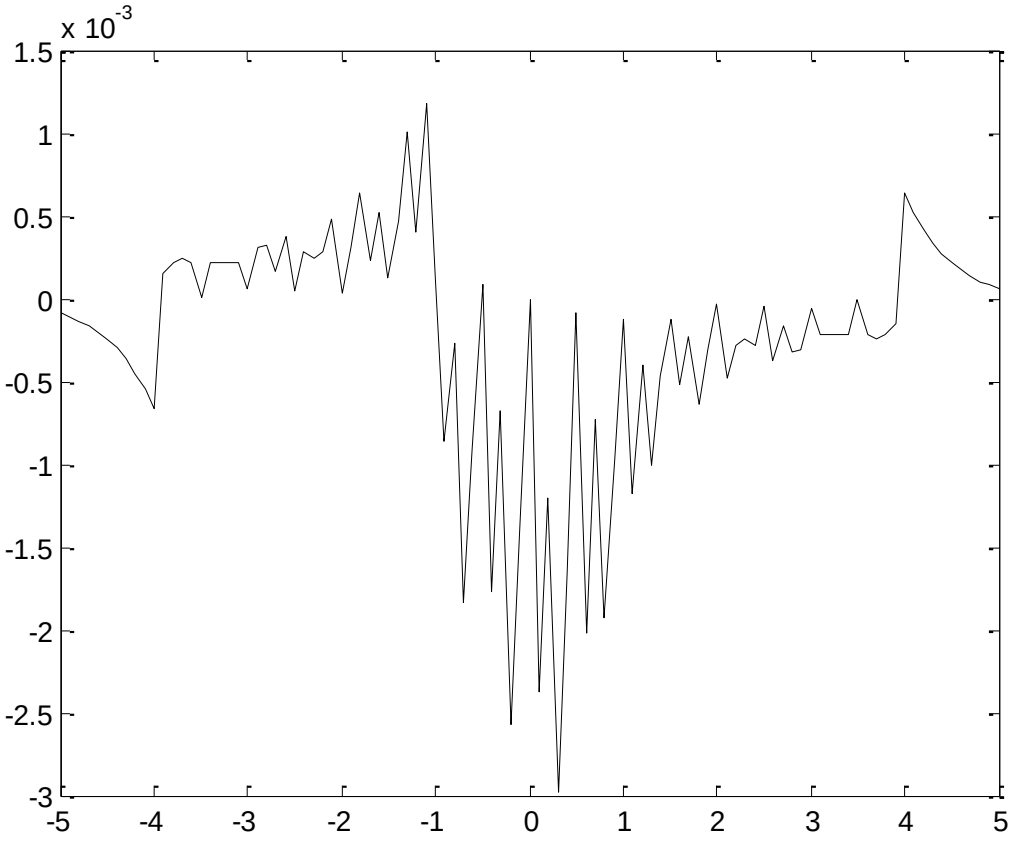

Figure 21. Error from *tanh* approximation using 16 divisions from -5 to +5.

3.5. Memory Structures

The Microchip PIC 18F45J10 microcontroller was used to implement the neural network. The microcontroller has only one true register that can be used for holding data, passing data, and ALU calculations. It has 1 kbyte of ram memory and when the neural network has 255 weights this memory is nearly all utilized. This memory is divided into four 256 byte banks. Only one of these banks can be accessed directly without the use of extra addressing instructions. This one bank has 128 bytes of general purpose memory and 128 bytes of processor configuration memory. This general purpose memory is used

as global and temporary variables for calculations. The other three banks are used for the weights and the individual nodes of the neural network.

The weights are stored as 16-bit numbers, which consist of an 8-bit integer and an 8-bit fractional part. Two banks are used to store the high and low byte of each weight. This allows for 255 weights to be stored. The zero location is not used for indexing reasons. Figure 22 shows the memory mapping.

As the output of the neural network is calculated the output of each neuron and the inputs need to be stored throughout the entire calculation to allow multi-layer connections. These node values are also 16-bit values. This poses a problem because there is only one ram bank left and two banks are needed. This problem is solved by splitting this bank into two separate banks; the low bank and high bank hold the low byte and high byte respectively. Notice this adds an additional limitation to the neural network size. The network may only have 127 total inputs and nodes. This limitation will most likely not be the dominant factor in many cases. Typically the weight limitation would be met prior to approaching the node limit.

This memory limitation is only relevant to this microcontroller. This concept could be extended to other microcontrollers or systems with extended ram. More ram could easily allow for even larger networks with greater numbers of neurons and weights. The C version of the software stores all weights and architecture values in program memory not in ram. There simply is not enough ram for the C version to function if these values are in ram.

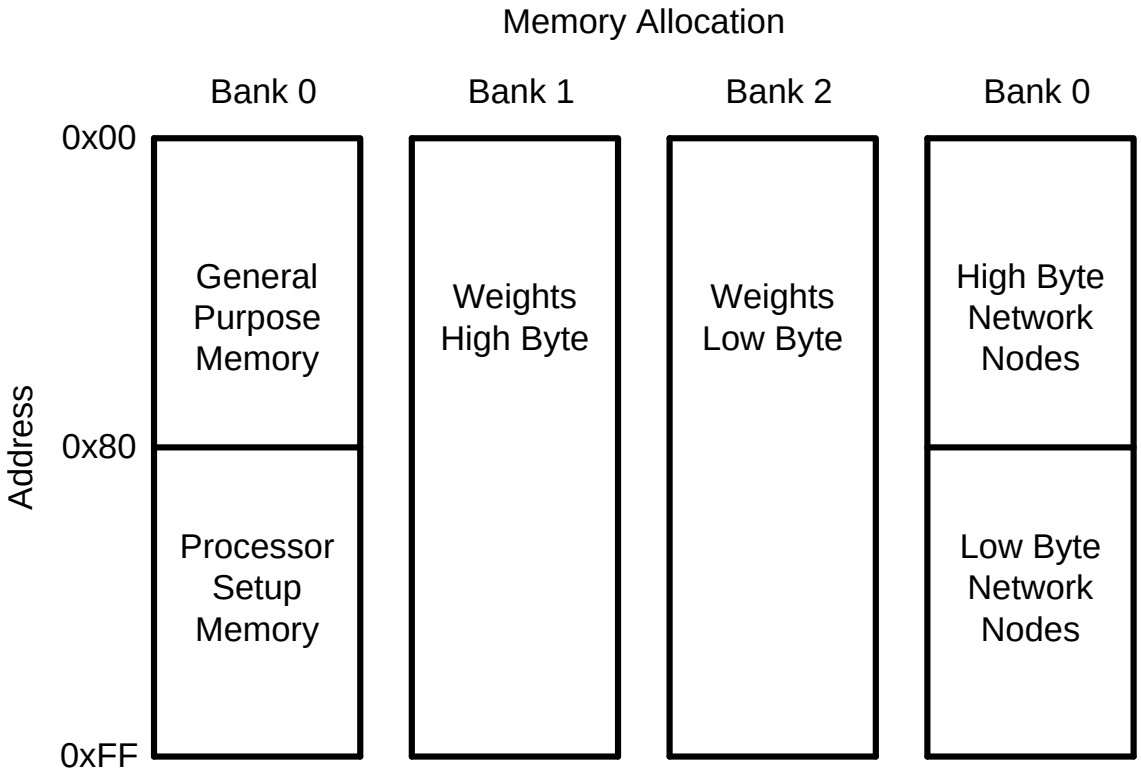

Figure 22. Memory Allocation table for Pic18F45J10.

3.6. Neuron By Neuron Computation Process

3.6.1. Forward Calculations

This process of forward calculations is a unique method compared to most neural network implementations because it uses the Neuron By Neuron method described in [57]. This method requires special modifications due to the fact that assembly language is used with very limited memory resources. The process is written so that each neuron is calculated individually in a series of nested loops; see Figure 23. The number of calculations for each loop and values for each node are all stored in two simple arrays in memory. The assembly language code does not require any modification to change the

network's architecture. The only change that is required is to update these two arrays that are loaded into program memory. These arrays contain the architecture and the weights of the network and are generated by NNT.

```
//Weights
Number of inputs; Number of outputs; (8-Bit)
Number of Weights; (8-bit)
Weight(1), Weight(2), Weight(3)....Weight(N); (16-bit)

//Architecture (8-bit)
Number of Neurons;
//Neuron 1
Neuron Scale, Number of Inputs, Output Node, Inputs(1-N)
//Neuron 2
Neuron Scale, Number of Inputs, Output Node, Inputs(1-N)
.
.
.
//Neuron N
Neuron Scale, Number of Inputs, Output Node, Inputs(1-N)
```

The arrays are automatically generated by the NNT, as described in Section 2.2.3. The forward calculation steps through each node of network without regard for the complexity of the network. Similar to a netlist in Spice, the topology array has the running list of connections and allows the user to make as many cross layer connections as desired, only limited by the total number of weights.

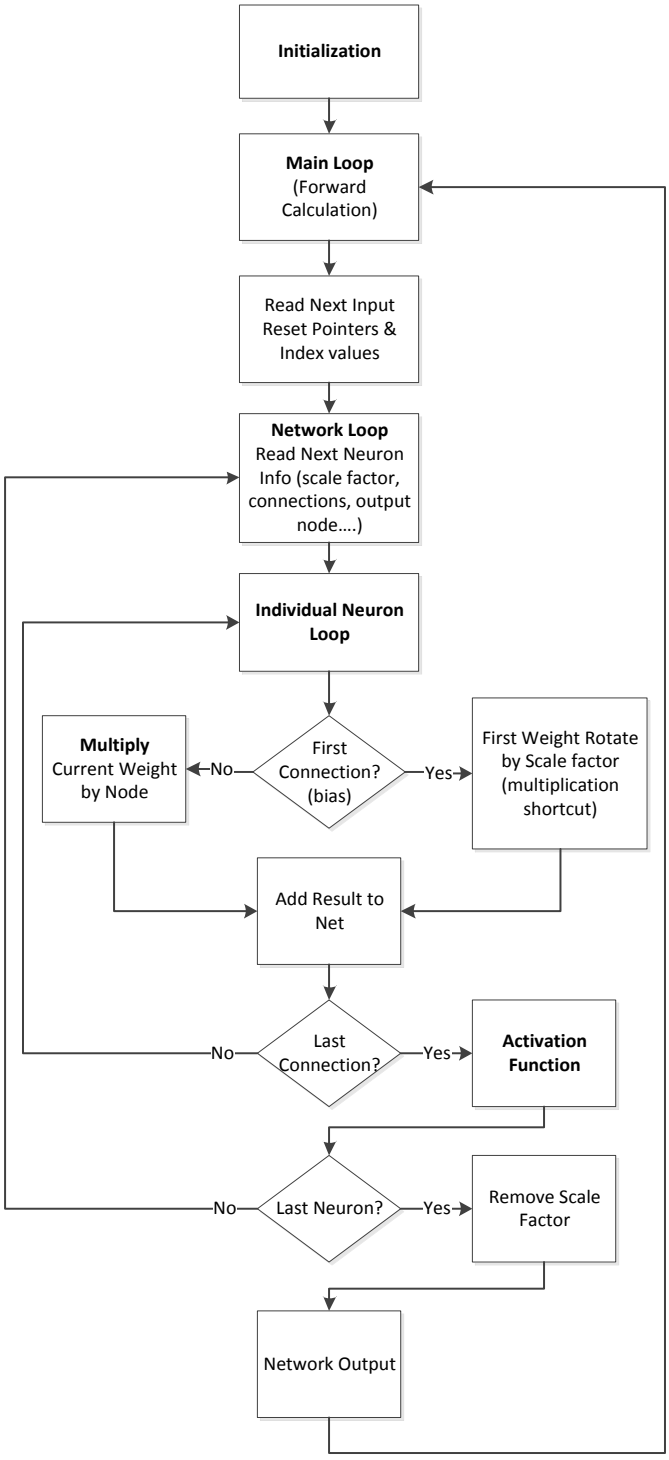

Figure 23 Block diagram of Neural Network forward calculations using the nested loop structure for cross layer connected networks.

As seen in Figure 23, the network starts with an initialization block that configures the microcontroller by setting up the hardware for inputs and outputs. Next the tables for the network are initialized. The weights are stored in ROM or off chip and are loaded into RAM for faster calculations. Finally there are numerous constants that are configured such as scale values and saturated neuron values.

After the initialization block, the Main Loop begins. This is an infinite loop that keeps the network sampling new inputs and then starting the forward calculations. With the next input sampled the network resets pointers and index values and enters the Network Loop.

The Network Loop is essentially a *for* loop that executes the number of times as the number of neurons. The Network is responsible for the architecture of the network as well as the output of the network. It reads the scale factors and neuron connections and sets the corresponding values for the Neuron loop.

The neuron loop begins with all of its indexes and pointers correctly initialized and it simply begins calculations. This loop is only responsible for calculating the output of a single neuron without information about the rest of the network. It begins by checking to see if the current connection is the bias connection or a standard input connection. Once the Net Value is calculated it passes the information to the Activation function. The individual calculation process are presented in more detail in Section 3.6.2. The Activation Function details are presented in Section 3.6.3.

After the Activation Function is finished the Network Loop determines when all neurons have been calculated. The final step is to remove the scale factor and send the output. The process is then repeated indefinitely.

3.6.2. Individual Neuron Calculations

The Neuron calculations go through several steps in order to process the pseudo floating point arithmetic. The first step is the net value calculation which is shown in Figure 24.

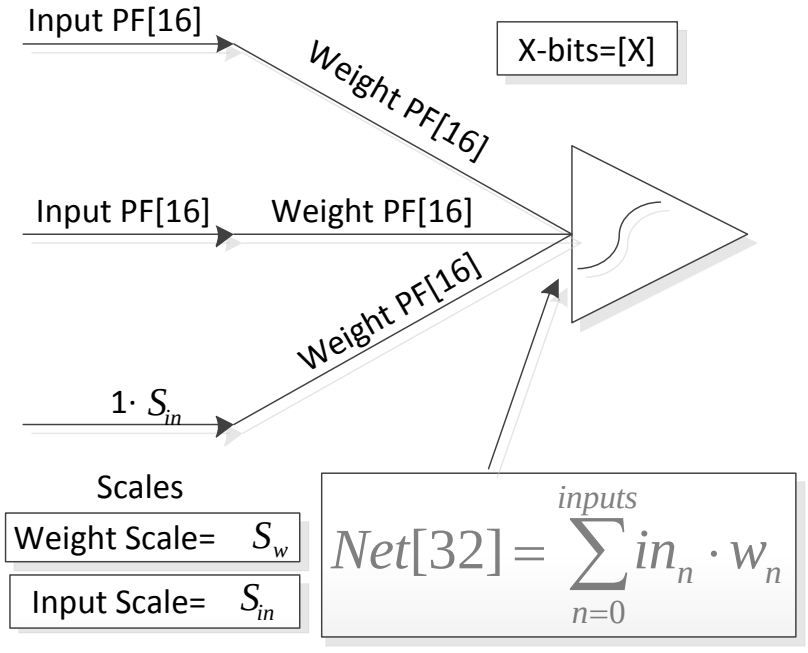

Figure 24. PF stands for Pseudo Floating point number. The Numbers in brackets refer to the number of bits that represent that particular value.

The inputs are multiplied by the corresponding weights and the result is stored in the 32-bit Net register. This is essentially a multiply and accumulate register designed for this particular stage. It is very important to keep all 32 bits in this stage for adding and subtracting. Without the 32 bits of precision at this step it would be very easy for an overflow to occur during the summing process that would not be reflected in the final net value.

The next stage is to turn the pseudo floating point number into a fixed point number. This process can be seen in Figure 25.

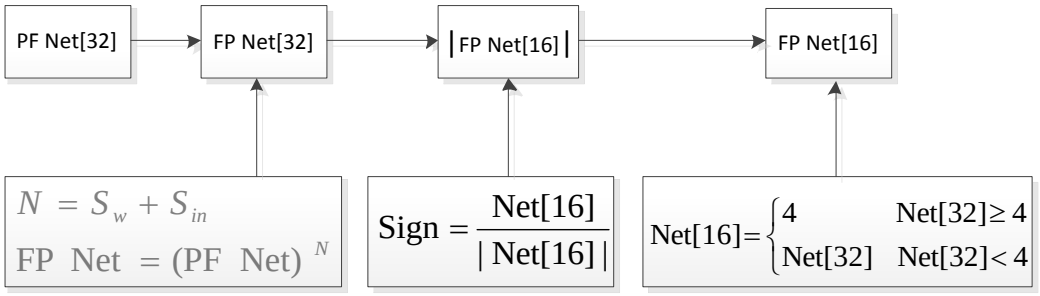

Figure 25. Pre Activation Function Routine. The transformation between a pseudo floating point number to a fixed point number that the activation function can use.

The next step is to convert the pseudo floating point number into a fixed point number that the activation function can correctly handle. First, the weight scale and input scale are summed. If the two factors exactly cancel then there is no scaling needed; however, if not, the formula shown in the figure is used. This raised to the *N* power is always the same as shift by *N*, because of the way the scale factors are calculated as described in Section 3.3. This makes the scaling process very fast as opposed to having to actually execute the multiplication instructions. Next, the sign of the net value is

stored and the absolute value of the net is used for the next steps. The net value is then examined and a decision is made. If the net value is too large then the *tanh* is approximately saturated and the appropriate output is assigned. However if the now fixed point number is within the operating range it is clipped to 16 bits and sent to the activation function. The activation function is detailed Section 3.4.

4. Application

In order to demonstrate that the microcontroller neural network is performing correctly several example control problems were tested. Neural networks have the unique ability to solve multi-dimensional problems with many inputs and many outputs, however these types of problems are not easy to test and verify visually. For this reason the network was tested mainly with two input and one output problems in order to plot the output as a function of the input on a three dimensional surface. This is not the only type of problem that can be solved, it is just to demonstrate. A two input and two output system is also shown by graphing the outputs separately to demonstrate that other types of networks will work as well.

The process is tested with the microcontroller hardware in the loop. In other words, the sensor data is transmitted via the serial port from Matlab to the microcontroller. The microcontroller then calculates the results and transmits this data via the serial port back to Matlab. The reason for this simulation is to isolate the errors in the system to those produced by the microcontroller calculations. In this test system any inaccuracy of the sensors can be avoided. This also removes any possibility of errors entering the system from external measurement tools.

To demonstrate the quality of the approximation several figures have been produced. The following examples will have some or all of the images that are described:

Training Data --- The training data is the data used to train the neural network. The number of points will vary with the application.

Ideal Neural Network -- This refers to a neural network running on a computer or a system using the IEEE floating point standard. The word *ideal* refers to most practical applications where there is no significant data loss due to the precision of the calculations. However, this is still a neural network approximation of the training data and not an identical representation.

PIC Based Neural Network -- This is the output of the neural network running on the PIC hardware. This approximation will not be identical to the ideal neural network because of the approximations that are made on the microcontroller.

Error Surfaces -- The error surfaces are differences between two of the previously shown surfaces. The surfaces will give a visual description of differences between surfaces shown on the same scale as the original surface. This comparison separates the error of using an ideal neural network and using a neural network with on the PIC.

Error Surfaces Tight -- These surfaces are the same as the error surfaces except on a much narrower scale to show what shape the errors have taken. This allows the user to identify problem areas or to confirm the error is evenly distributed.

Histograms -- The histograms show the errors of different surfaces in a numerical manner. This shows the distributions of the errors, in order to identify the

distribution of the errors. The X-axis is the errors and the Y-axis is the number of data points within the corresponding error range.

4.1. Simple Surface

The following example is of a simple three dimensional control surface. This surface was used in a few examples to demonstrate multiple aspects of implementing the neural network on the microcontroller. The training data can be seen in Figure 26. This surface is 16 data points from a smooth surface and the neural network will learn to produce a better, smoother surface than the data given. This shows one of the fundamental advantages of neural networks opposed to other control methods. It is not necessary to have *perfect* training data to obtain very good results. The neural network inherently approximates the points in between the data points in a very smooth fashion.

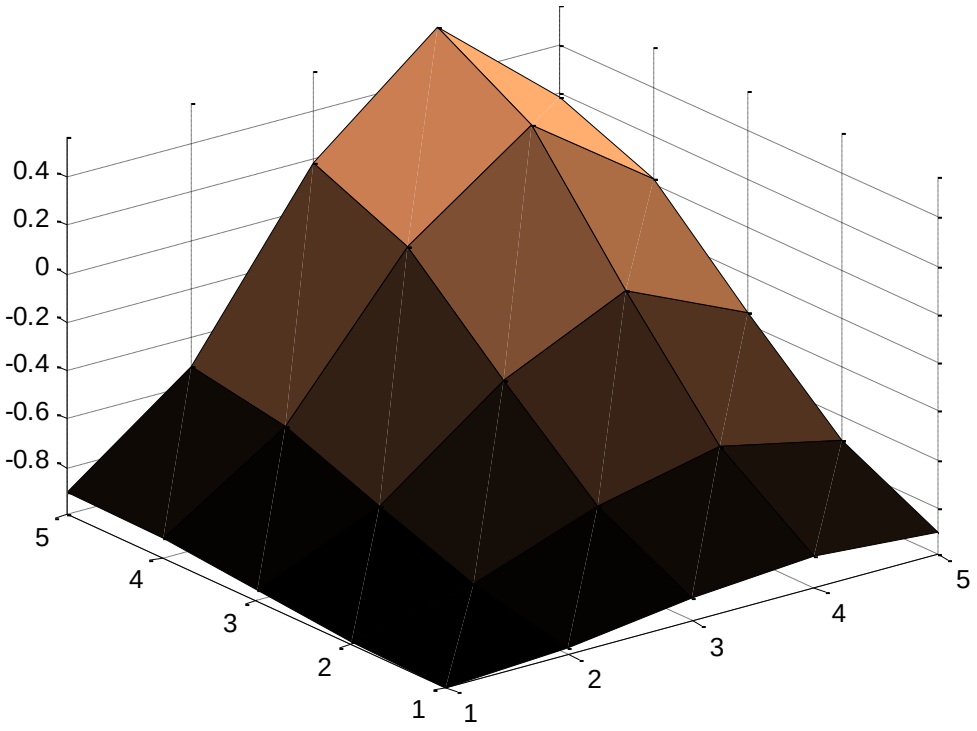

Figure 26: Simple surface training data.

This surface can be solved very effectively using a network with four neurons, which is shown in Figure 27. This architecture approximates the surface very well with minimal error. The output of the ideal network and the PIC can be seen in Figure 28 and Figure 29 respectively. To the naked eye there is no visual difference between the surfaces. Following the surfaces is the tight error surfaces in Figure 30 and Figure 31. These surfaces have a much smaller scale and they show the shapes and offset of the errors.

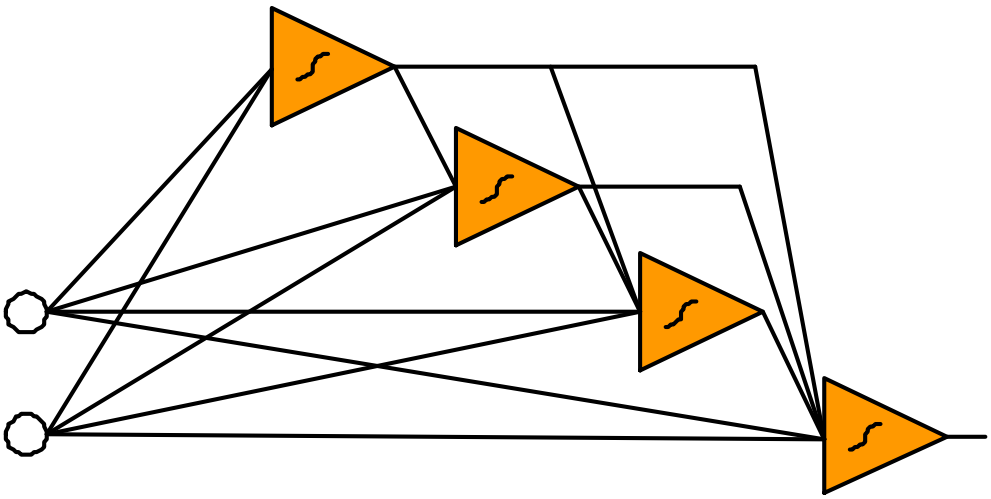

Figure 27: Four neuron cascade architecture for solving the simple surface. The inputs are the circles on the left and the output is the last neuron on the right side.

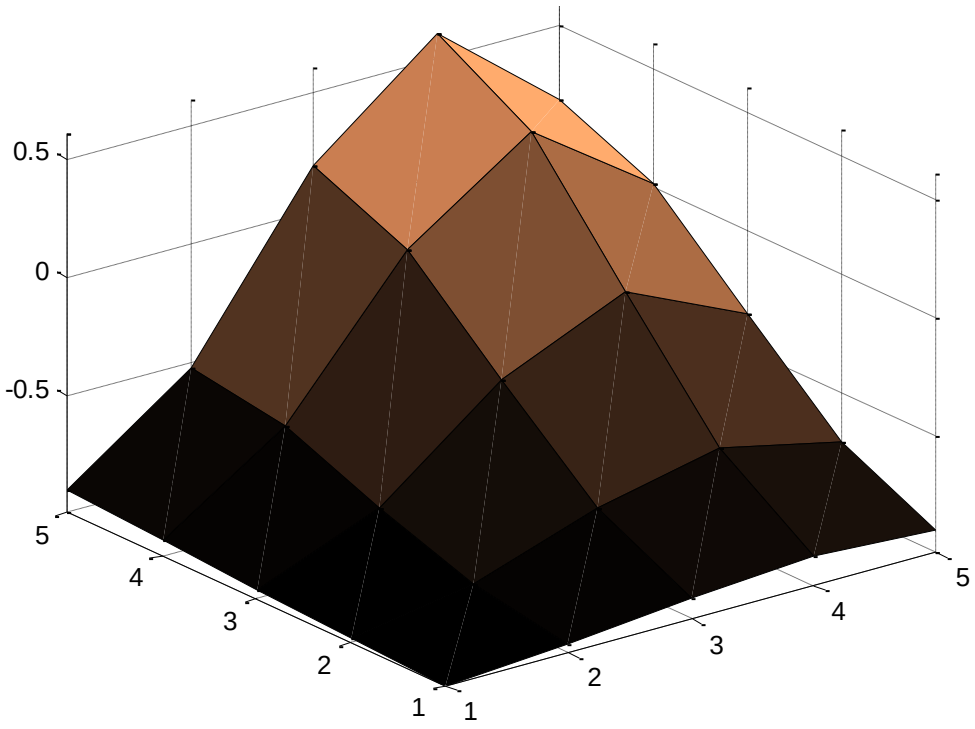

Figure 28: Ideal neural network output.

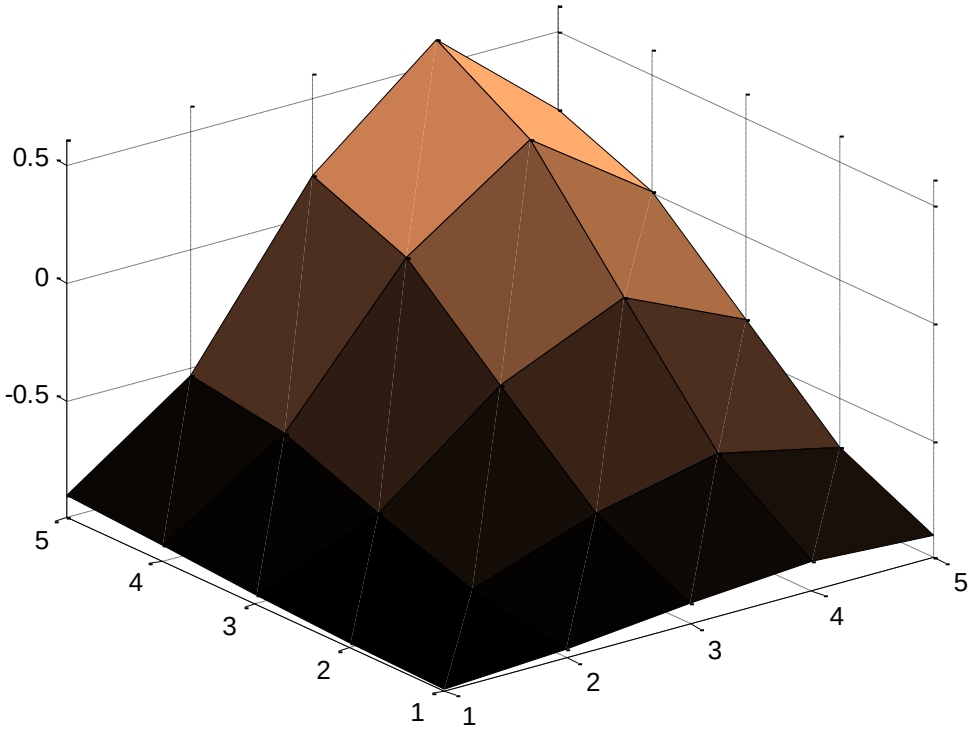

Figure 29: Output of the PIC.

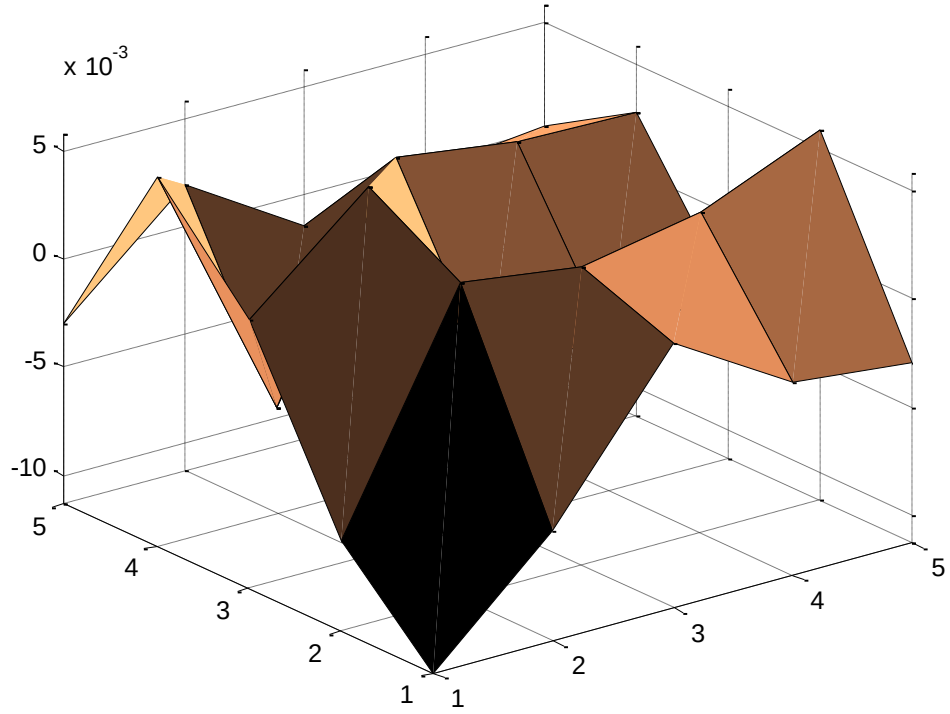

Figure 30: Error surface showing the difference of the training data and the ideal neural network.

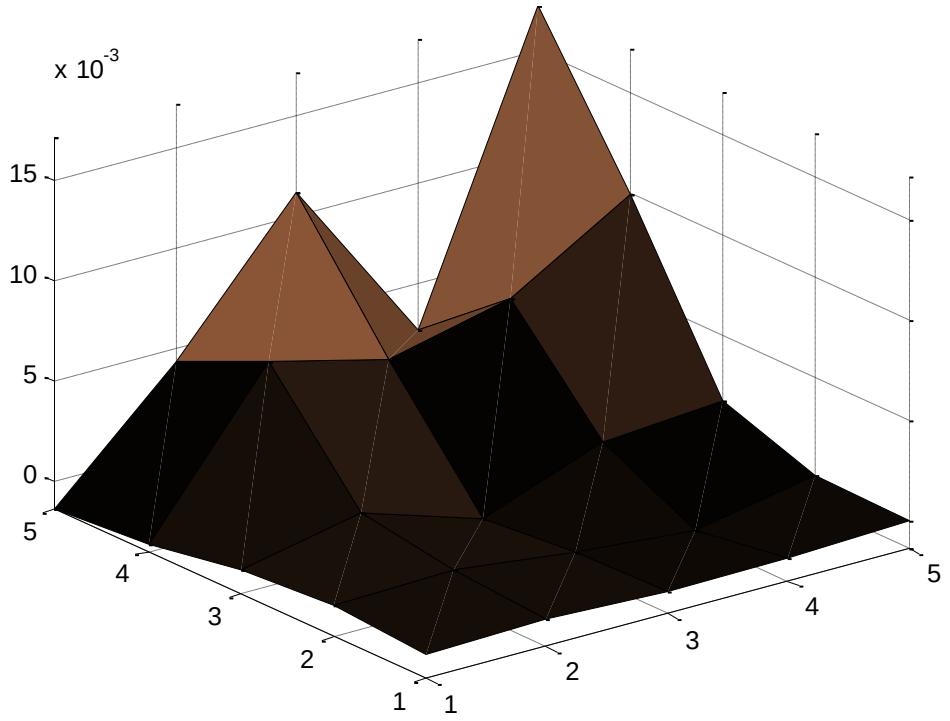

Figure 31: Error surface showing the difference of the PIC output and the ideal neural network.

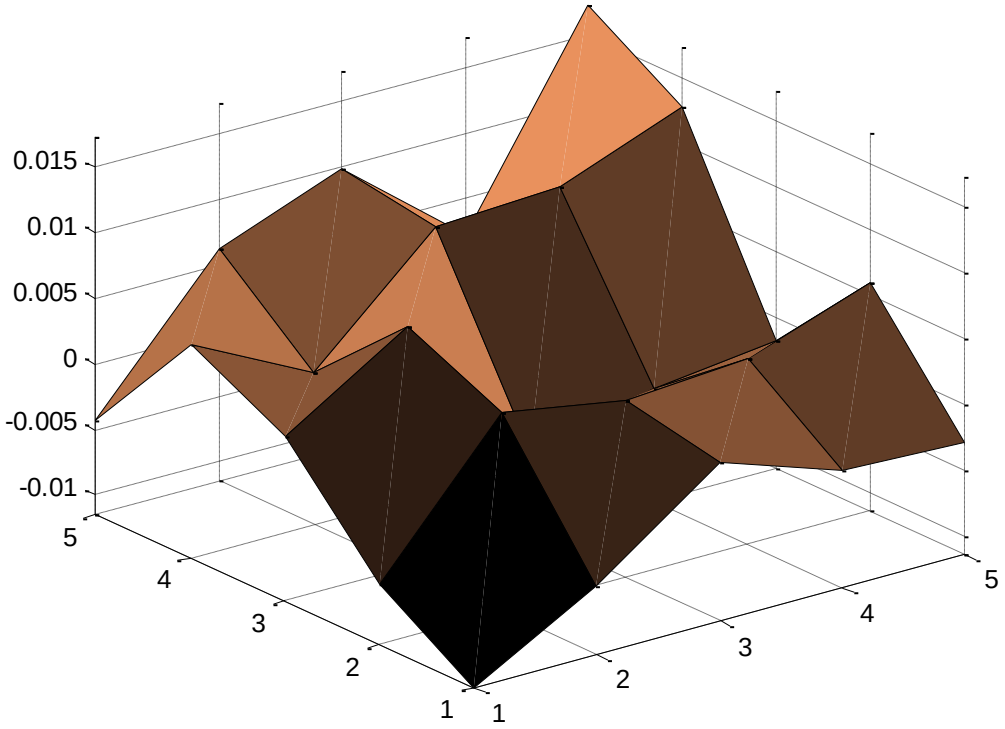

Figure 32: Error surface showing the difference of the PIC output and the training data.

These images show the error introduced by using the neural network of Figure 27 and the show the variation from the ideal neural network to the network calculated on the PIC. Figure 32 shows how the output of the PIC is a very close approximation of the original training data. The majority of the points are centered around zero and have less than 1% error. This data is also verified in the histograms shown in Figure 33.

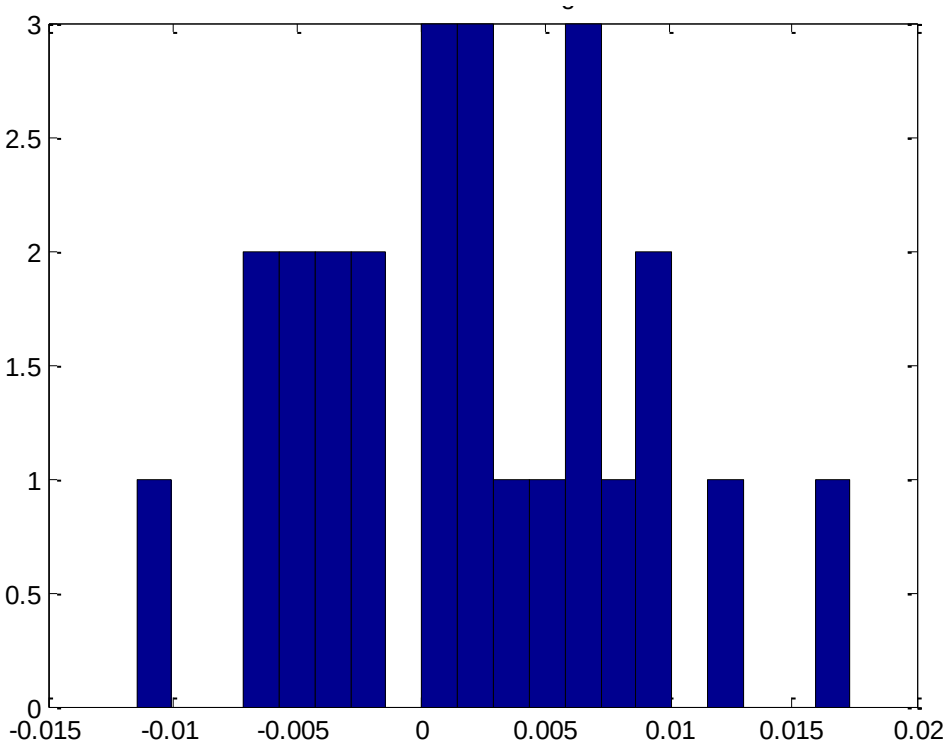

Figure 33: Histogram of errors between the PIC and the training data. The X-axis is the error and the Y-axis is the number of samples.

The same neural network was tested with patterns that were in between the training patterns. A total of 196 test patterns were used to generate the image shown in

Figure 34. This shows the neural network's ability to approximate points for which it was never trained. From this figure it is obvious that the network produces a very reasonable nonlinear approximation between the training points. The error surfaces with more points were omitted because they did not show any significant differences that were not shown in the previous figures.

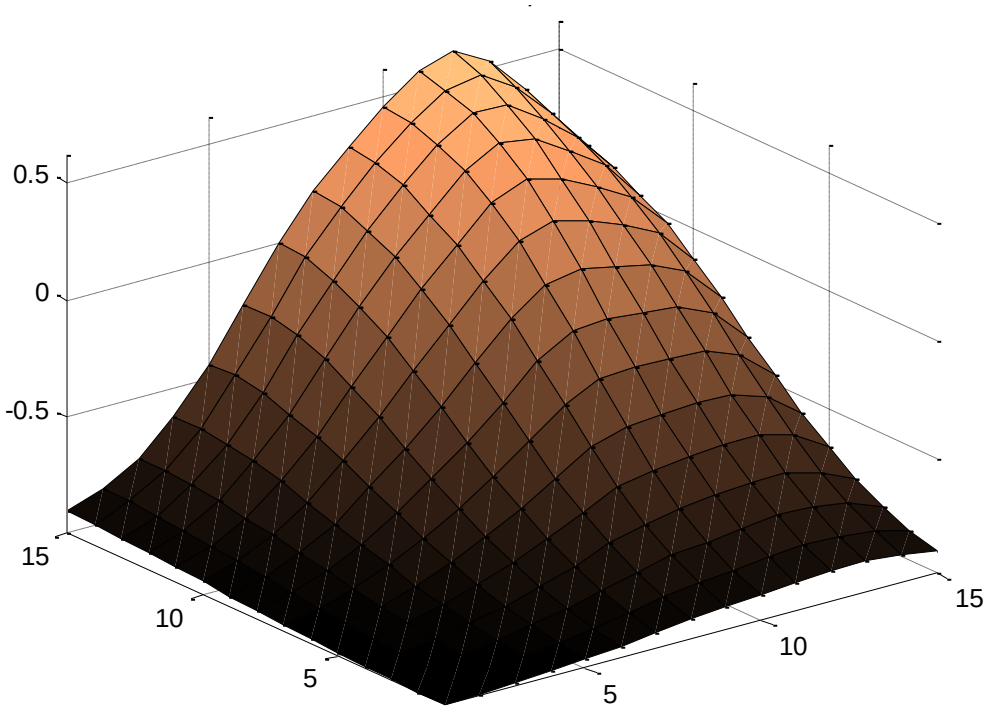

Figure 34: Output of PIC with 196 test patterns.

This same surface was tested once again but with a much smaller network to show that the quality of the surface being produced is dependent on the number of neurons. The ideal neural network is not as close of an approximation of the training surface but the neural network on the PIC is still very close to the ideal neural network. This

network has only two neurons and produces errors typically under 10% for the entire system as shown in Figure 36. The surface is not quite as nonlinear due to the number of neurons being reduced. The histograms in Figure 37 and Figure 38 verify that the total error is larger and mostly introduced by the network and not by the PIC calculations.

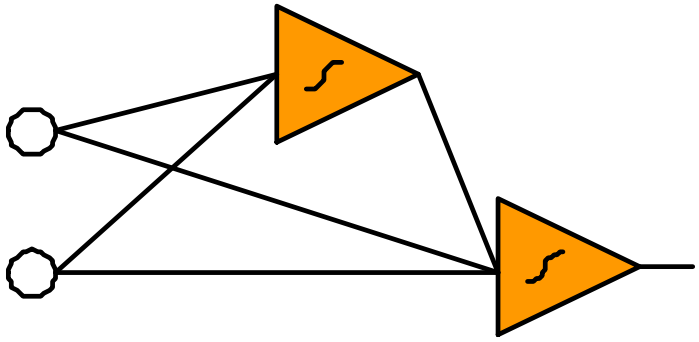

Figure 35: Two neuron architecture for solving simple surface problem.

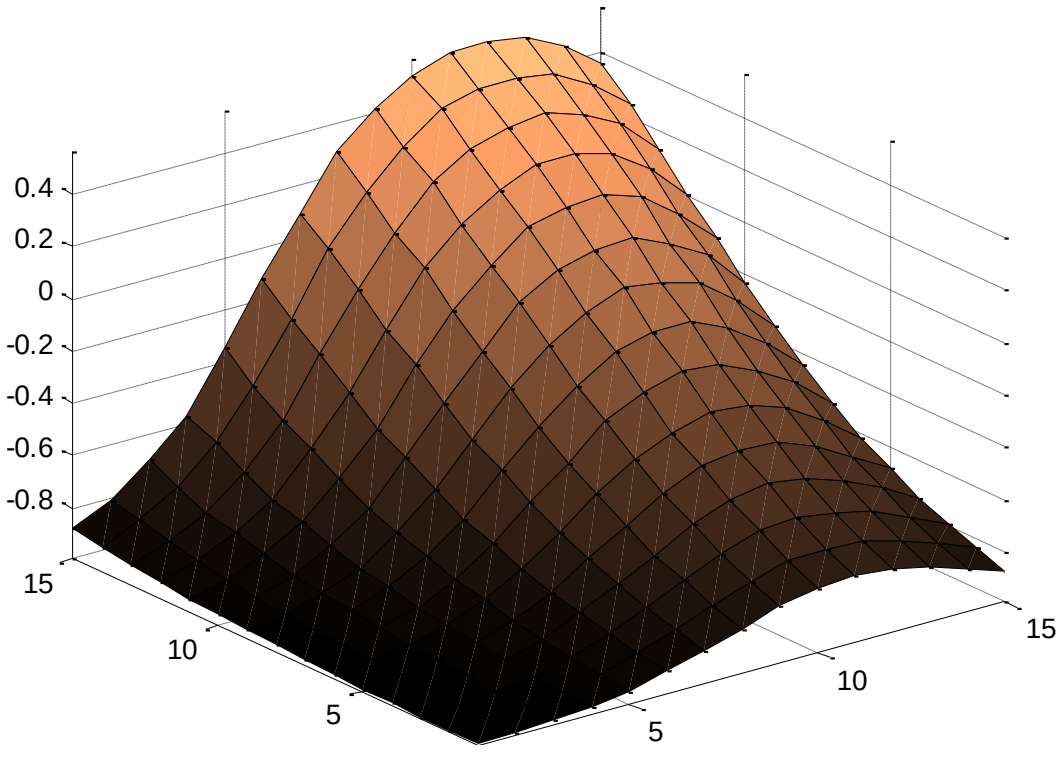

Figure 36: PIC output with 196 points on small two neuron architecture.

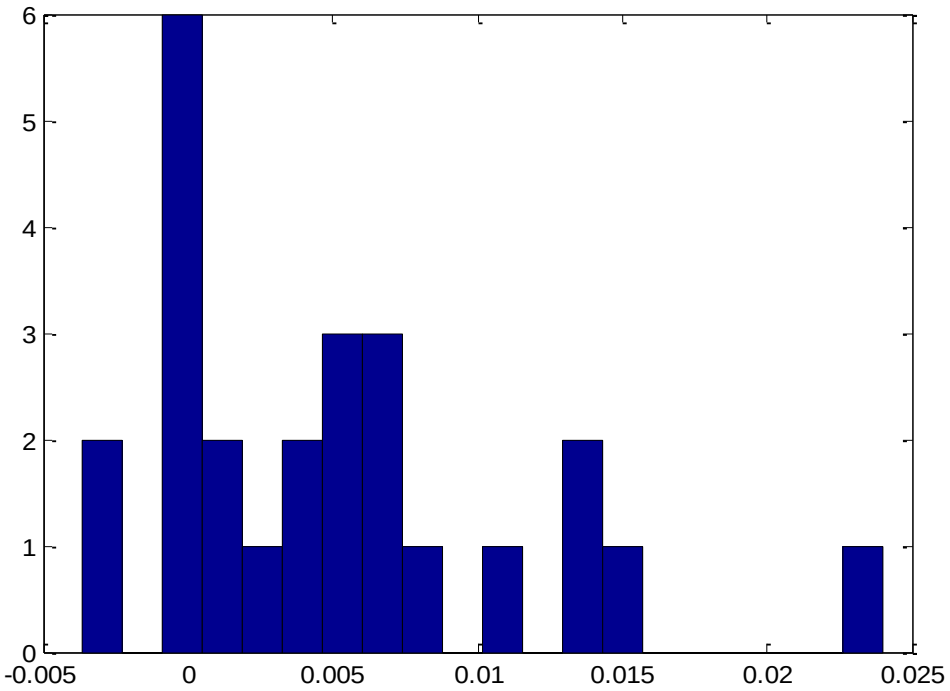

Figure 37: Histogram of errors between the ideal neural network and the PIC. The X-axis is the error and the Y-axis is the number of samples.

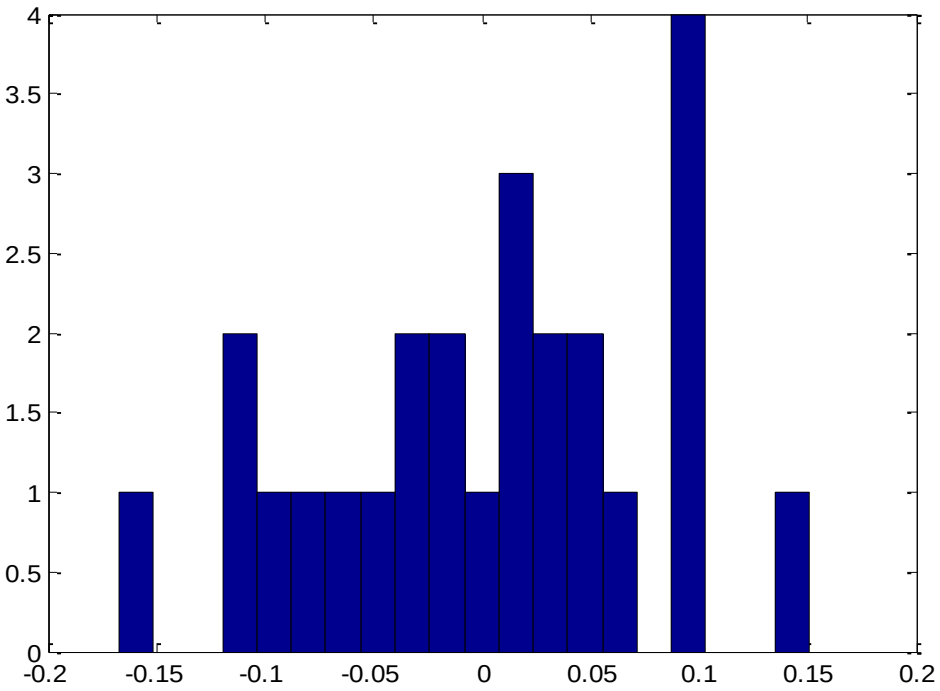

Figure 38: Error of the PIC and the Training data compared. The X-axis is the error and the Y-axis is the number of samples.

4.2. Matlab's Peaks Surface

The next example is generated by the common Matlab function *peaks*. The surface has several peaks and valleys and is a rather complicated nonlinear control surface. This complicated surface requires significantly more neurons to solve to a comparable accuracy. The training surface is shown in Figure 39 and the architecture in Figure 40. The network architecture is somewhat of a hybrid between the common MLP networks and the cascade network shown in the last example. The architecture has two hidden layers but all neurons are connected directly to the inputs and all preceding layers.

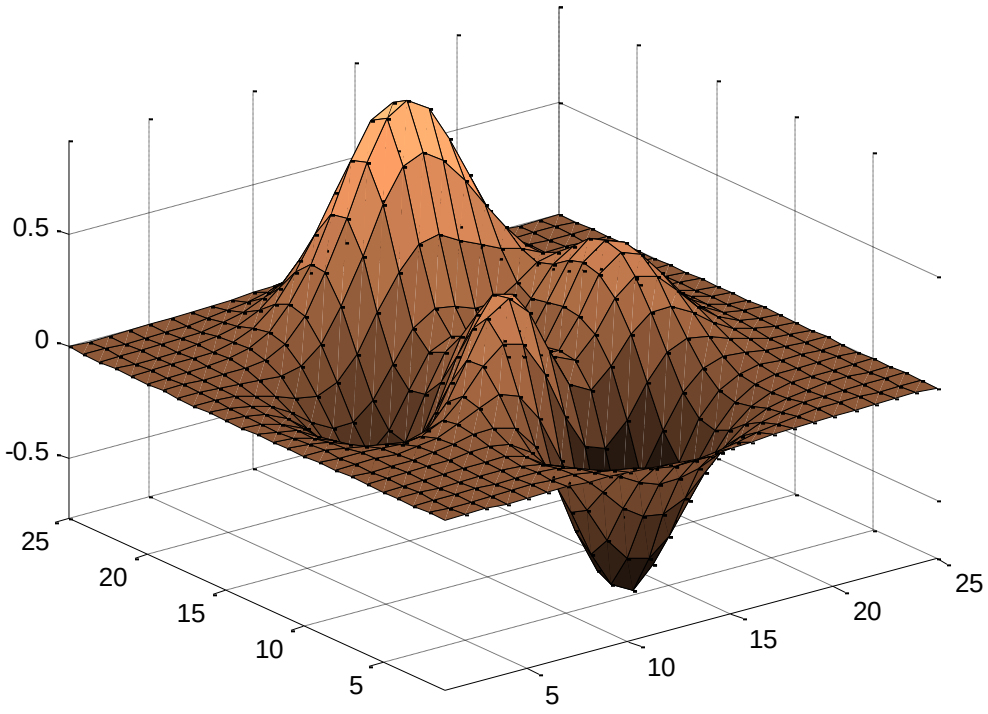

Figure 39: Training data used for Matlab's peaks surface.

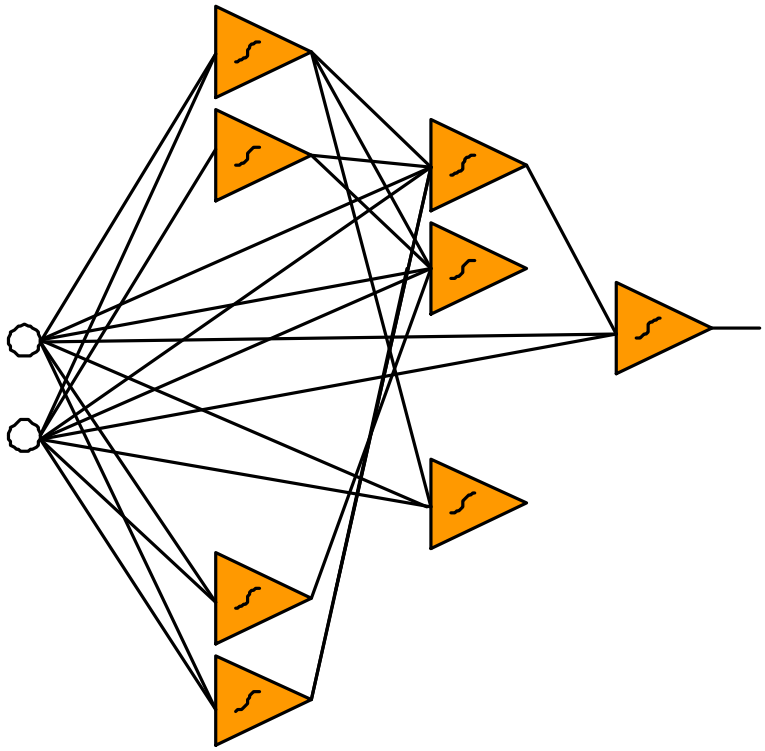

Figure 40: Eight neuron network used for solving the Matlab peaks surface.

This network was able to solve the peaks problem quite well and was implemented in the PIC with reasonable error. The PIC output, shown in Figure 41, had some rippling artifacts that are most likely attributed to rounding errors, with so many calculations for this network. However, the error is still very tolerable with no single point having more than 8% error and a very large percentage of points with 2% error centered around 0%. This can be seen in detail in the histogram in Figure 42. Looking at Figure 43 which is the error analysis of the ideal neural network it can be seen that about half of the large outlier errors are produced by the neural network itself and not the microcontroller calculations.

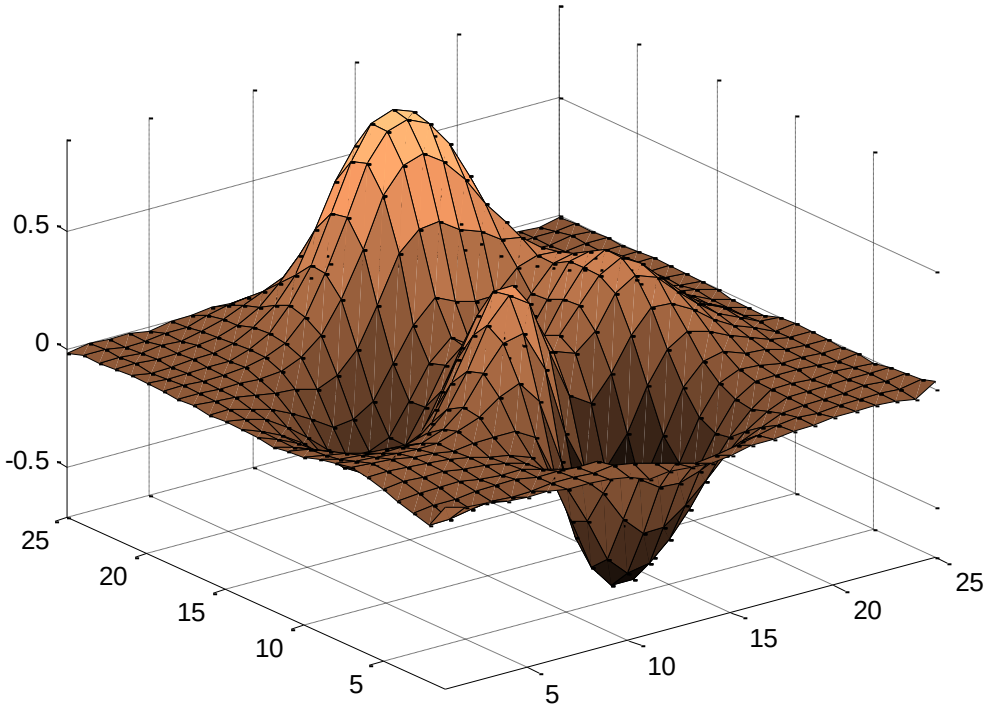

Figure 41: Pic output for Matlab's peaks surface.

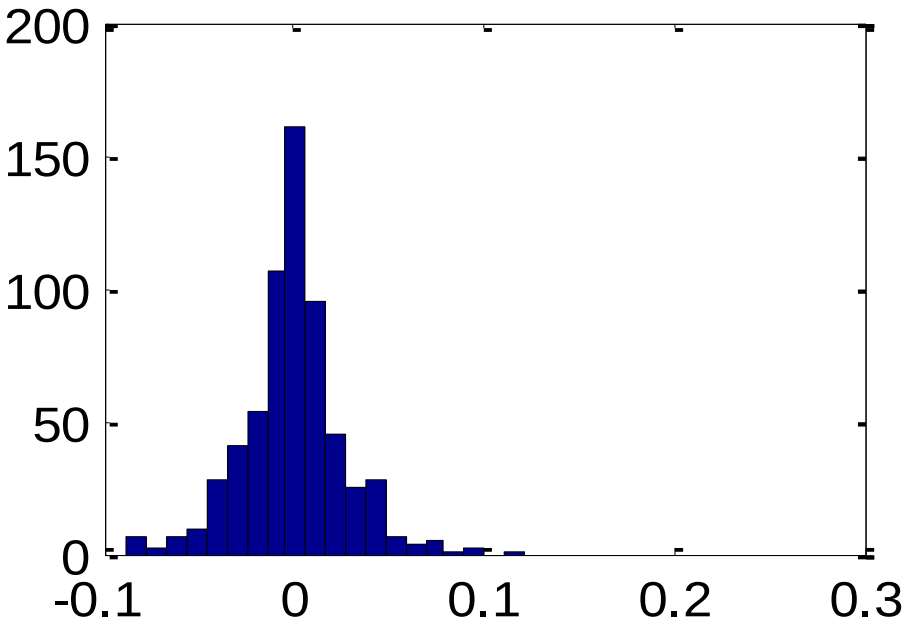

Figure 42: Histogram of errors between the PIC output and the training data. The X-axis is the error and the Y-axis is the number of samples.

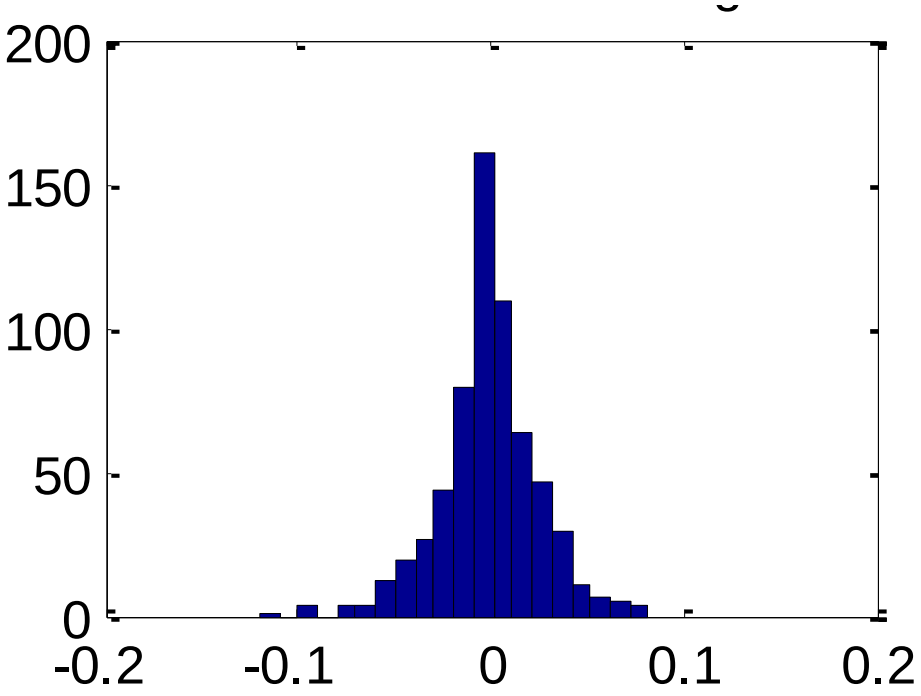

Figure 43: Histogram of errors between the ideal neural network and the training data. The X-axis is the error and the Y-axis is the number of samples.

4.3. Two Arm Planar Manipulator

A two link planar manipulator was used as a practical application for this embedded neural network. The particular aspect is shown for sensing the position of a robotic arm, given sensor data of the joints. In this example the embedded neural network will calculate the *x* and *y* position of the arm based on the data read from sensors at the joints. This is known as forward kinematics. With this system it is assumed that the sensors are linear potentiometers. The *x* and *y* position of the arm is very nonlinear. The position can be calculated by Equation 13. In other words, this is a two input and two output nonlinear system. For this experiment we will assume that R1 and R2 are

fixed length arms. However, this same procedure could be adopted for varying length arms by simply retraining the neural network with four inputs rather than two. The robotic arm simulated can be seen in Figure 44.

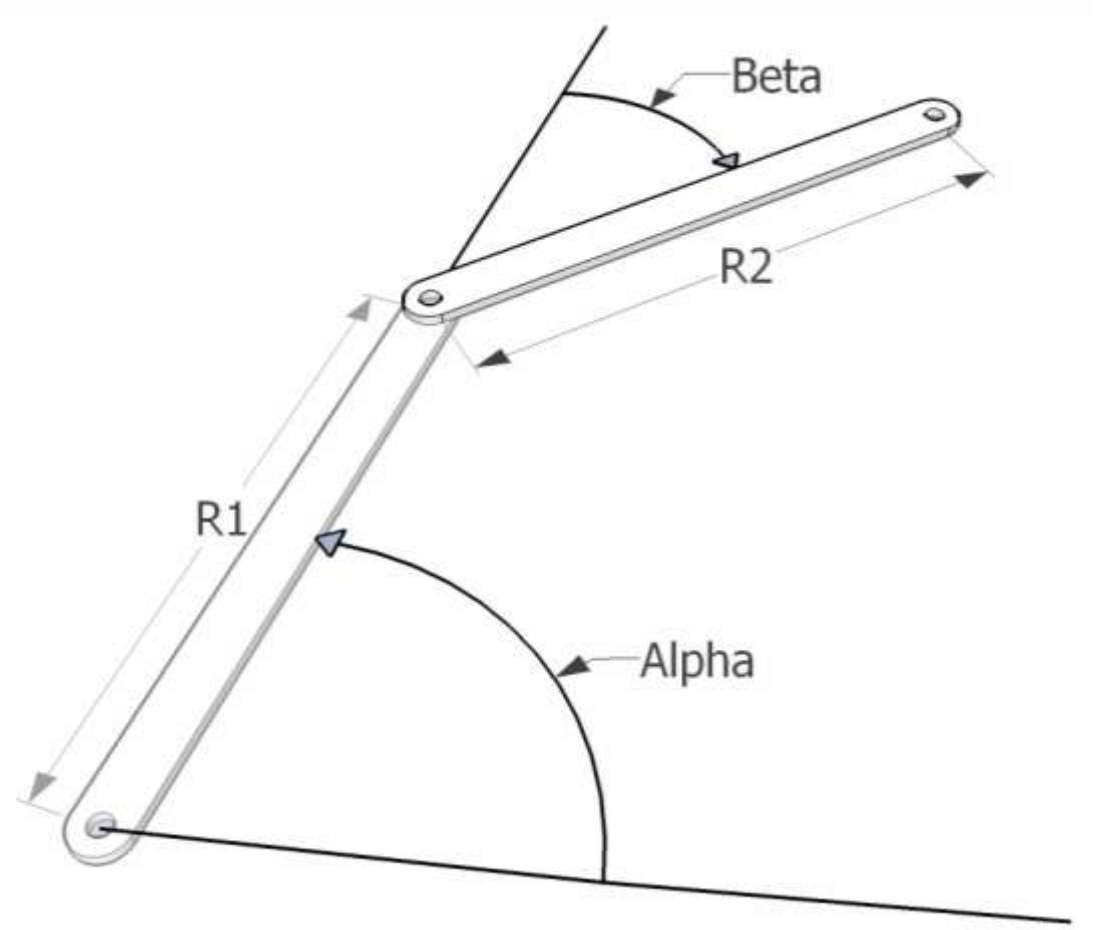

Figure 44: Two arm planar manipulator with variables shown.

The first step of the process was to generate neural network training data. The following equations were used to calculate the *x* and *y* position based on *alpha* and *beta* where *alpha* and *beta* are the angles shown in Figure 44.

$$x = R1 \cdot \cos(alpha) + R2 \cdot \cos(alpha + Beta) \quad (10)$$
$$y = R1 \cdot \sin(alpha) + R2 \cdot \sin(alpha + Beta)$$

The neural network was then trained using this data. The trained network was tested in Matlab to confirm that it functioned correctly and can be seen in Figure 45. Matlab generates a set of test patterns of a user selectable size and transmits these values to the microcontroller via the serial port and reads the results. Matlab is then used to test the output patterns and calculate the errors. This process will introduce errors in two places. First there will be the error created by using a neural network approximation rather than the original equations. Then there is the error introduced between the ideal neural network and the network on the microcontroller. The training data for outputs x and y can be seen in Figure 46 and Figure 48, respectively. Figure 47 and Figure 49 show the outputs of the microcontroller neural network also for both outputs. The error for each output is the difference between the training data and the microcontroller output. These errors can be seen in Figure 50 and Figure 51. Histograms of these errors were also generated and can be seen in Figure 52 and Figure 53. These results are very reasonable and are expected to be less than that of the physical system. In other words, the error generated by the potentiometers or by measuring the position of the arm manually would be comparable to the error generated by the neural network.

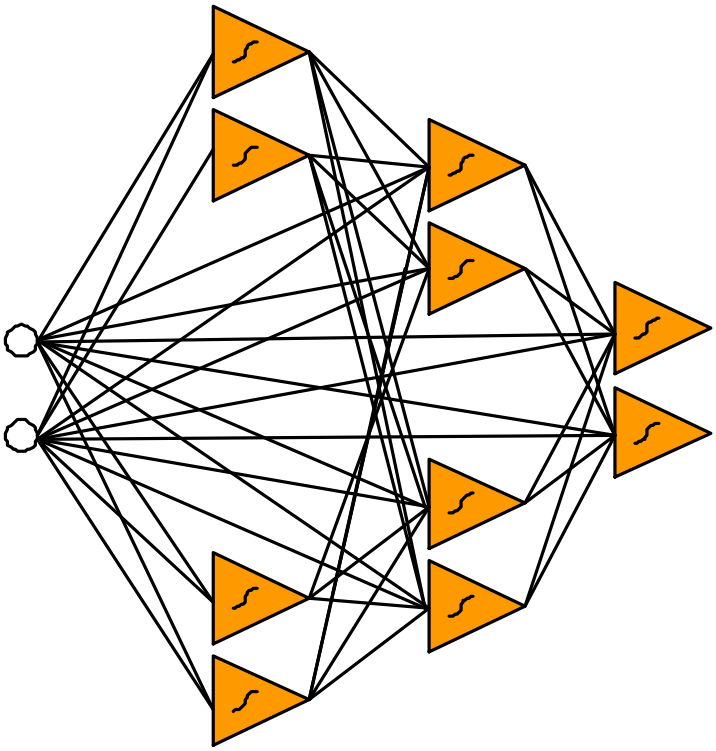

Figure 45: Ten neuron network for solving forward kinematics problem.

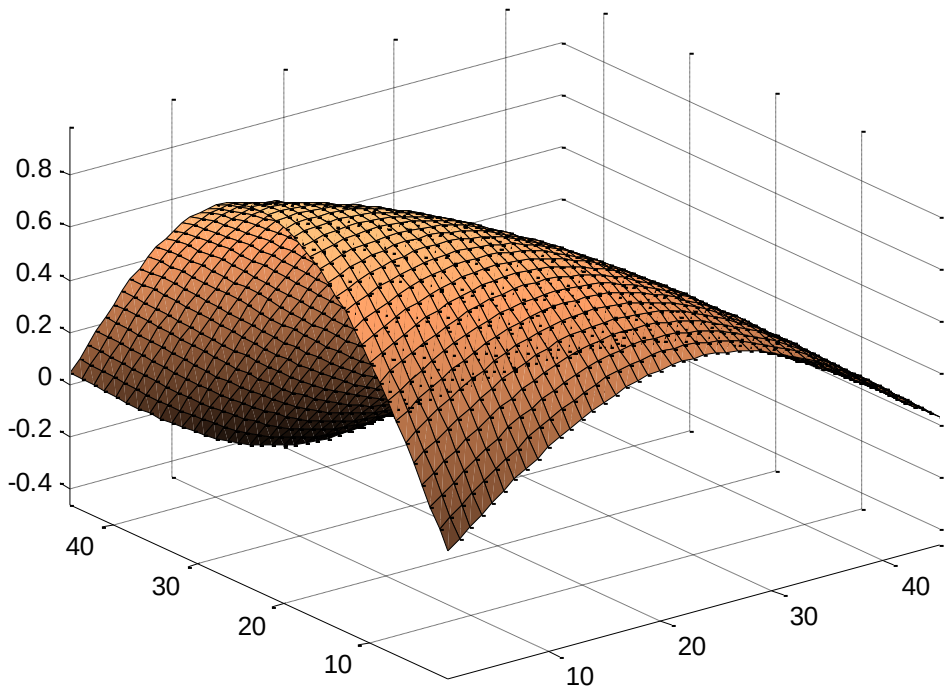

Figure 46: Training data output x of the two output system.

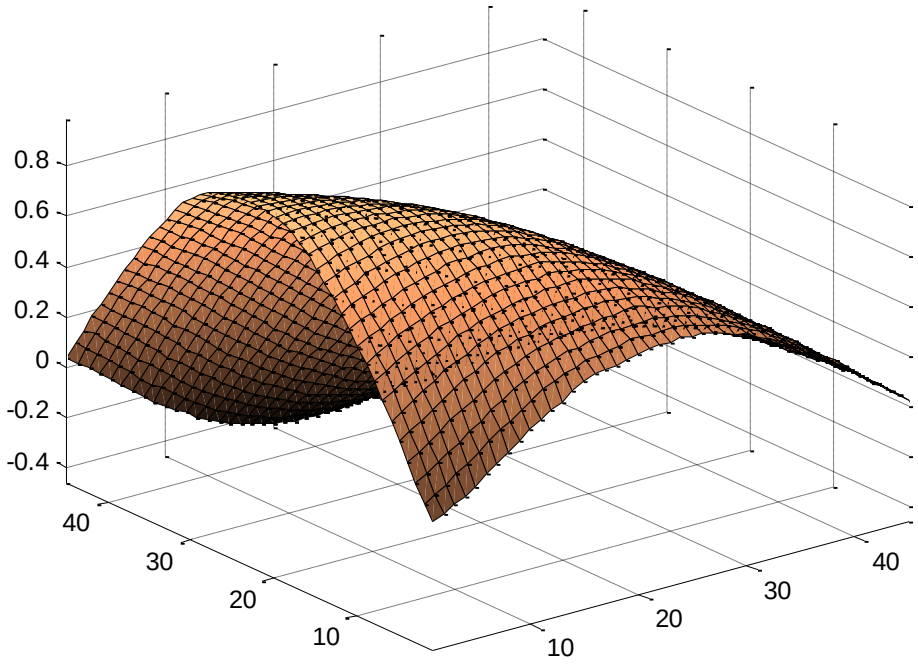

Figure 47: Output x of two output system generated by embedded neural network.

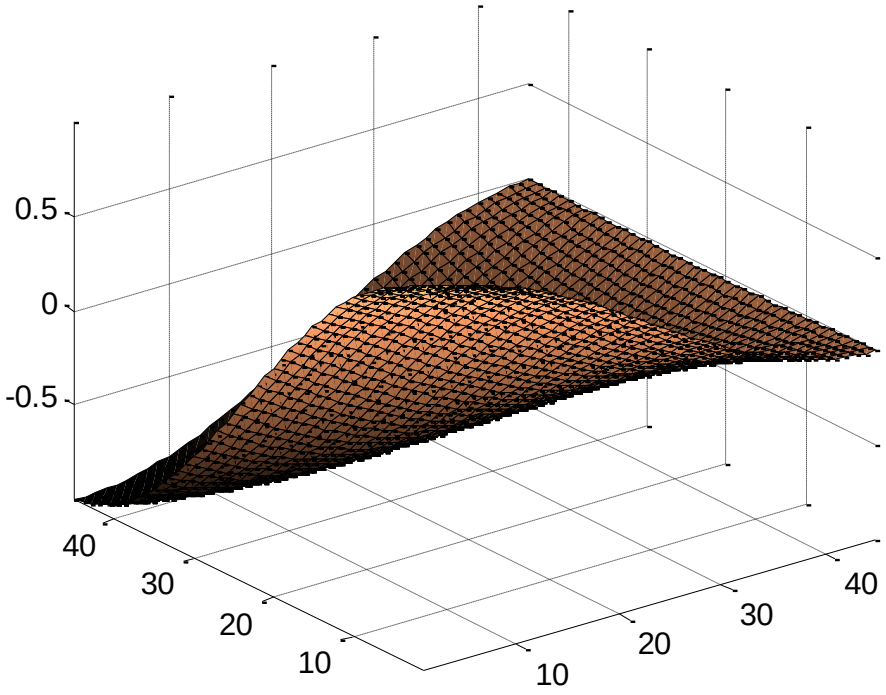

Figure 48: Training data output y of the two output system.

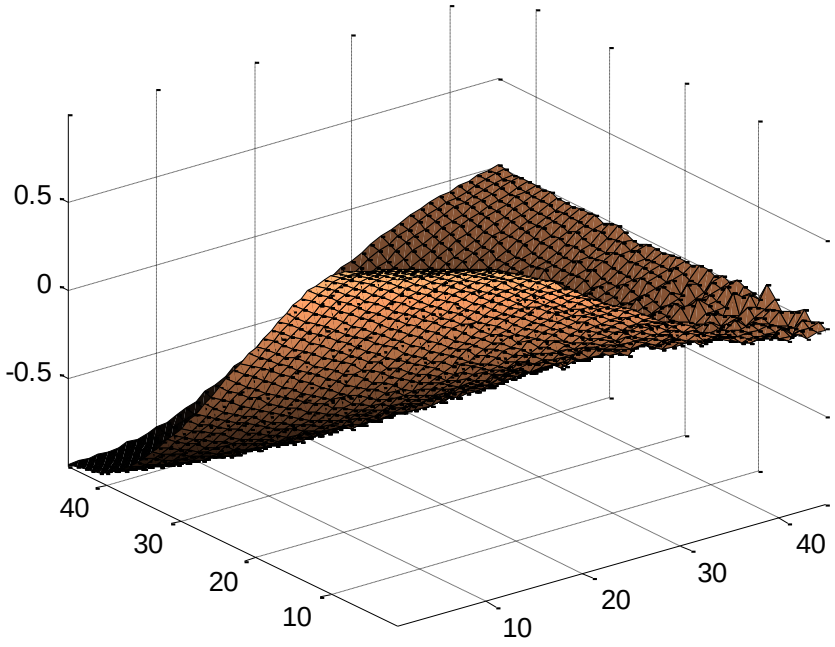

Figure 49: Output y of two output system generated by embedded neural network.

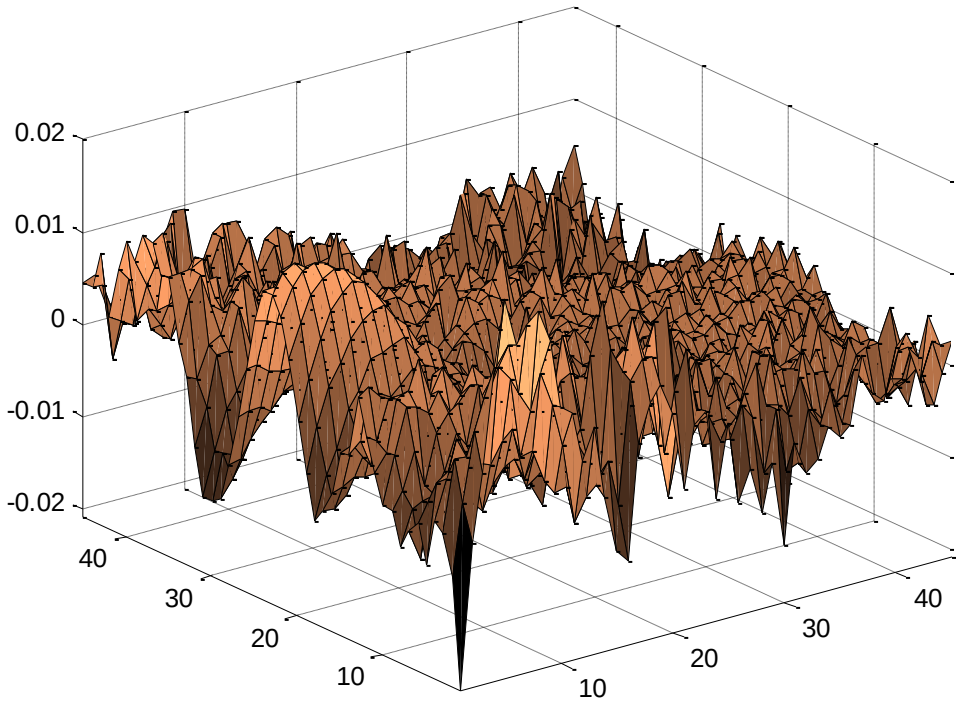

Figure 50: Error between the embedded neural network and training data of output x.

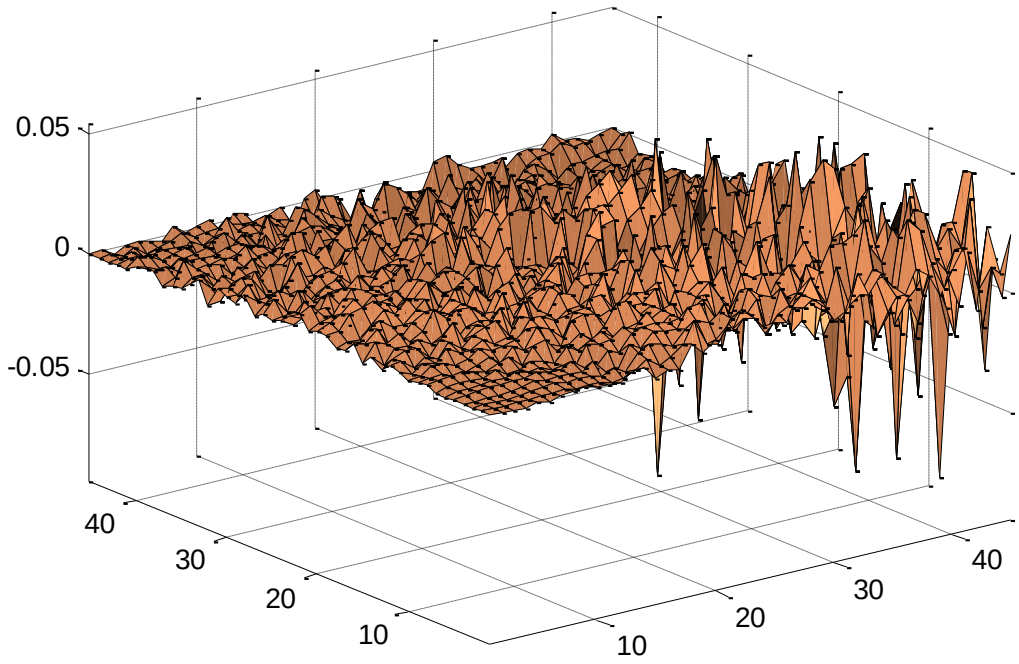

Figure 51: Error between the embedded neural network and training data of output y.

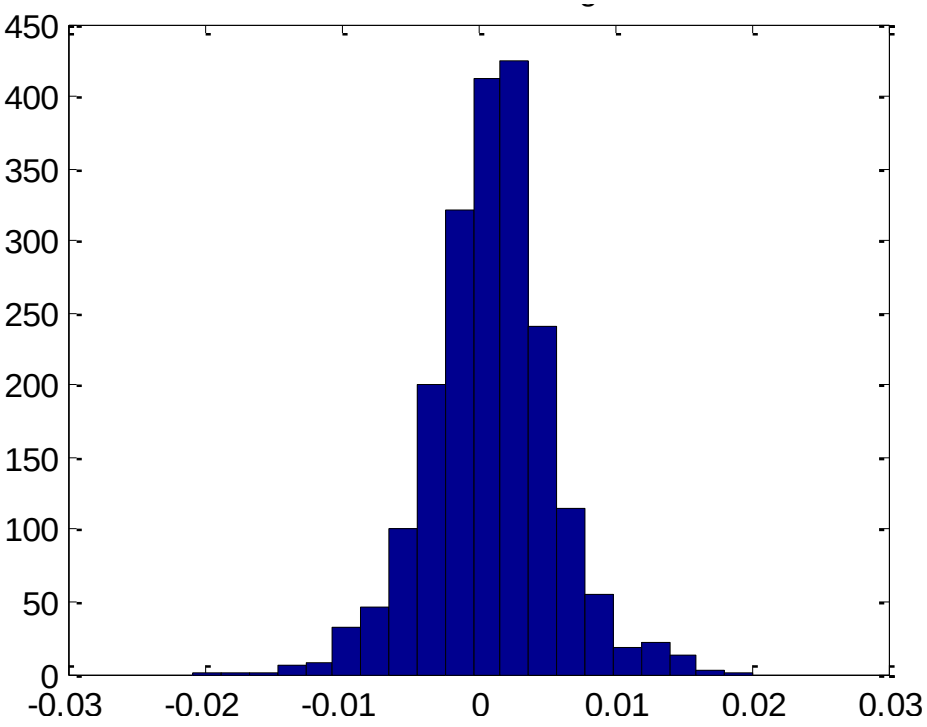

Figure 52: Histogram of Errors between training data and PIC for output x. The X-axis is the error and the Y-axis is the number of samples.

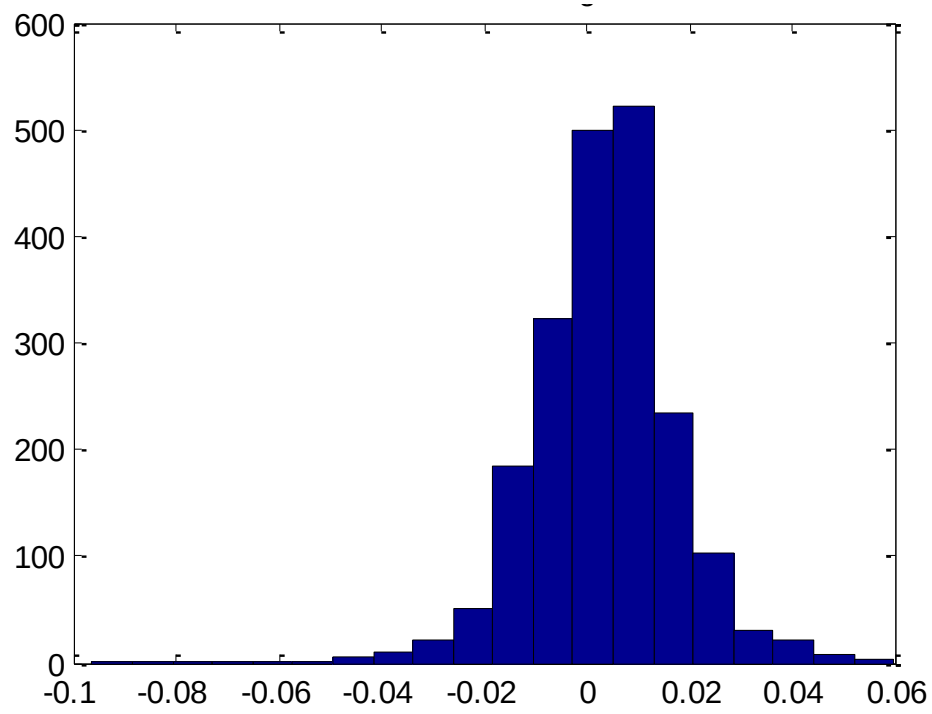

Figure 53: Histogram of Errors between training data and PIC for output y. The X-axis is the error and the Y-axis is the number of samples.

4.4. Matlab's Peaks Surface Example In C

The example shown previously was repeated, except this time using a C generated file instead of the assembly. The output of the PIC using C introduces an insignificant amount of error when compared with the error introduced by the neural network. The same training data and ideal network was used from the previous example. The original training data is shown in Figure 39 and Figure 55 shows the output of the network, written in C and implemented on the PIC. The difference between the ideal network and the PIC implemented network is insignificant when compared to the error added by the neural network itself. The error from the microcontroller is two orders of magnitude smaller than that generated by the microcontroller itself. More details of this can be seen in the histograms describing the errors in Figure 55 and Figure 56.

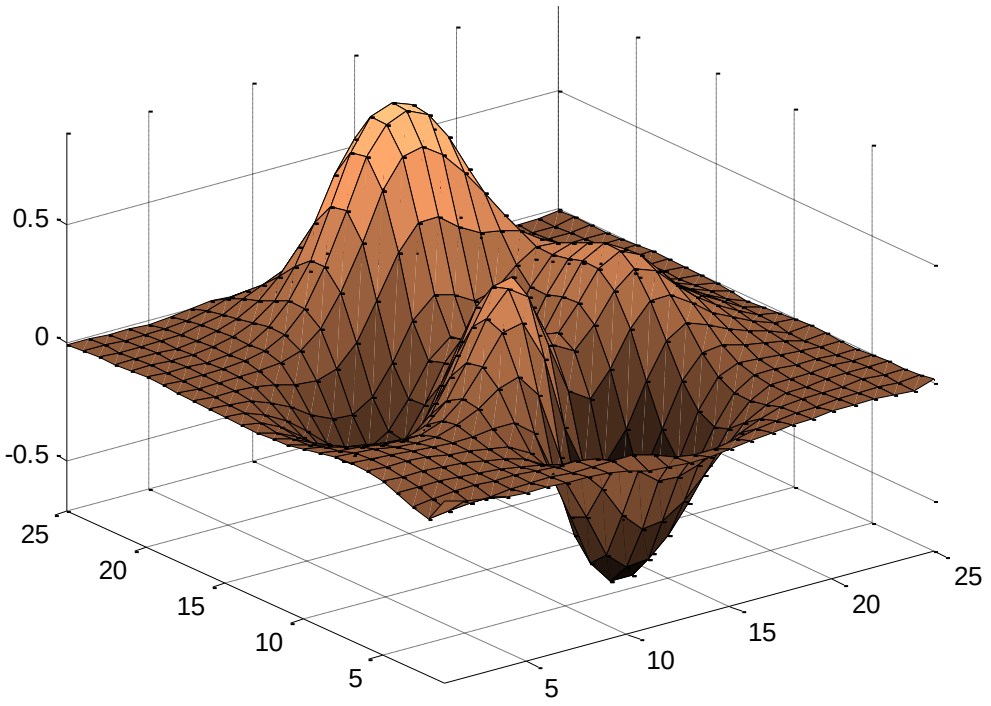

Figure 54: Output of the PIC using the C version of the embedded neural network software.

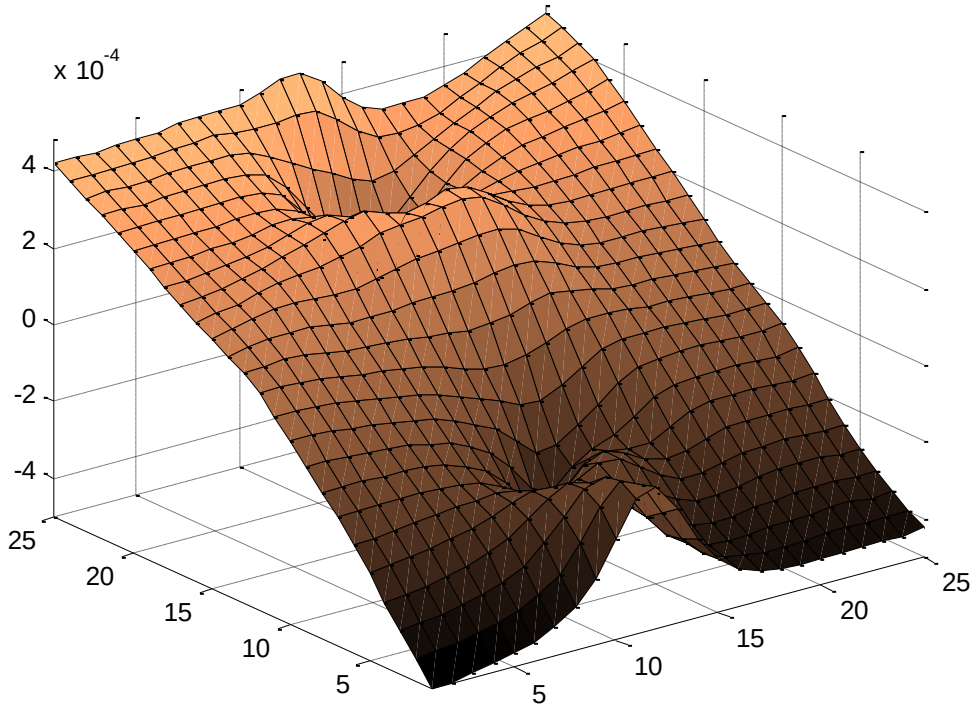

Figure 55: Error between the ideal neural network and the PIC output using C.

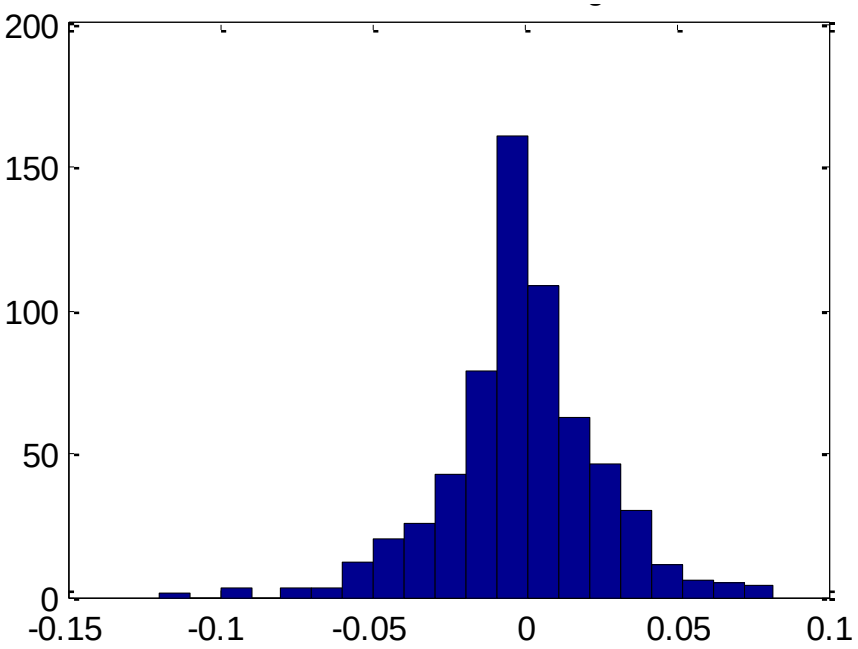

Figure 56: Histogram of errors between Ideal neural network and training data. The X-axis is the error and the Y-axis is the number of samples.

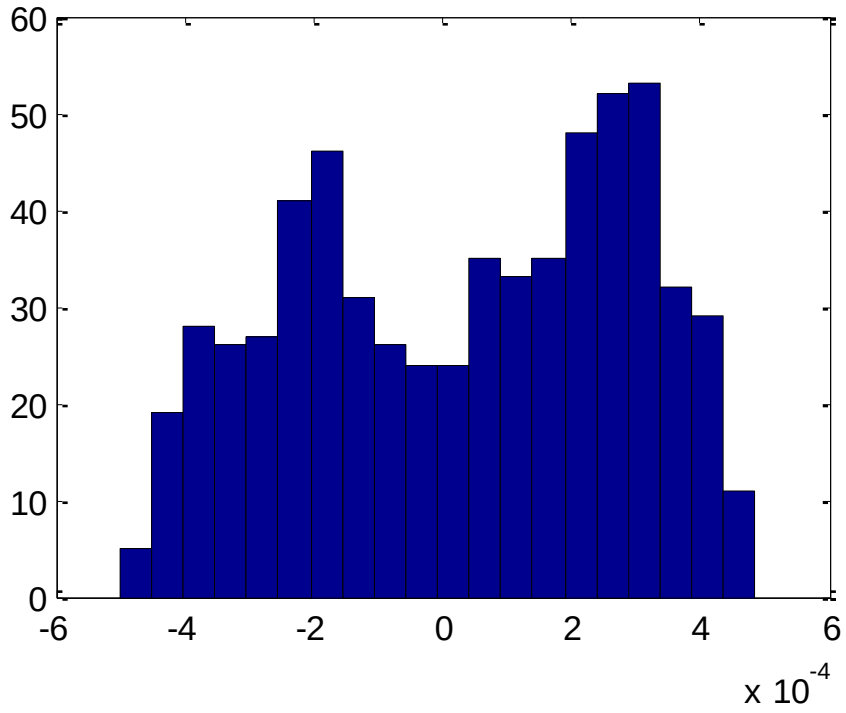

Figure 57: Histogram of errors between ideal neural network and PIC implemented neural network. The X-axis is the error and the Y-axis is the number of samples.

4.5. Experimental Data Summary

After comparing the results of the two different implementations of neural networks, it was obvious that the C version is much more accurate; however this accuracy does not come without a decrease in performance. The C version was significantly slower due to its complexity of calculations and its necessity to store all weights and nodes in program memory because they are too large to put in RAM. The accuracy and speeds can be seen in Table 2: Neural network performance comparison..

Surface	Neurons	Training Error	Ideal & PIC RMS	Ideal & Training RMS	Training & Pic RMS	Time ms
Peaks	19	0.01	0.015005	0.010594	0.018257	1.6
Peaks	8	0.01	0.007292	0.0253	0.026012	0.752
Simple	4	0.001	0.005151	0.0044663	0.0068827	0.317
Simple	2	0.001	0.0080934	0.07516	0.076336	0.163
Peaks C	19	0.01	0.00012772	0.0089456	0.0089523	2.95
Peaks C	8	0.01	0.00025928	0.0253	0.025301	5.87
Simple C	4	0.001	0.000041934	0.0044663	0.0044666	2.2

Table 2: Neural network performance comparison.

Table 2 shows the performance of the neural network in several categories. The first column is the surface name as discussed in the previous sections. The neuron column is the number of neurons used in the network for the given example. The training error is the total error for all points summed. This parameter is used to decide when the training has been completed. The root mean squared (RMS) errors are taken for the difference between two of the three surfaces and then labeled in each column. The time

in milliseconds is the time required for one forward calculation of the neural network. This time does not include the time for acquiring a sample input or using the output.

The C implementation time requirements are very difficult to analyze and virtually impossible to predict because of the sophisticated C compiler that is used. A few preliminary tests were done on the C program, such as taking into account how long it takes to manipulate the floating point numbers. Times were collected for memory reads and writes, multiplication, and the *tanh* calculation. Based on these individual component times, estimates were made to predict how long the system would take to process a given number of neurons based on the number and type of calculations. However, this data is a reasonable approximation for small neurons with a small number of calculations as in the simple surface shown in the last line in Table 2. The C version took approximately ten times longer than the assembly version, which was predicted based on number of calculations. Due to the optimization of the compiler this does not hold true for larger networks as in the peaks surface. The original estimate was significantly slower than it actually is. This allows the C version to operate faster than anticipated and still be very accurately which makes it very valuable even on such a low end microcontroller.

5. Conclusion

This dissertation presents a solution for embedded neural networks across many types of hardware and for many applications. The software package presented here allows the user to develop a neural network for a desired application, train the network, embed it in any platform, and verify its functionality. This software package is a complete embedded neural network solution.

This package offers the user the ability to use far superior neural network architectures than in other training software. The user has the freedom to customize his network for his application. He can use traditional multi layer perceptron networks or the superior arbitrarily connected networks including fully connected and cascade networks.

Most other software and research only trains with error back propagation or other first order algorithms. This dissertation gives the user their choice of traditional EBP as well as the faster and more efficient second order algorithms such as the Neuron by Neuron algorithm and the Enhanced Self Aware algorithm.

The software offers the user the option of installing the network on a Microchip's 18Fxxxx series microcontroller using custom made neural network software written in assembly language and optimized for both the microcontroller and the neural network application. This version offers a very fast and accurate solution on a very inexpensive microcontroller.

If the user prefers to use a different platform then the C code generated can be used to implement the trained network on any C capable platform. This can be used on other microcontrollers as well as PC based neural networks.

This accomplishment demonstrates that neural networks can be used to solve problems that in the past would require custom software programs to be written for each problem. In other words, if three separate microcontrollers were needed to control three different processes for a single project then three unique programs would need to be written. This solution offers one standard solution for controlling all three. The user simply needs to train three separate networks, which is an automated process. Then the user has the solutions for unique problems without having to write code for the mathematics. The neural network may not be the absolute best solution for every problem but it is a very acceptable and easy to implement solution for an extremely large variety of problems.

Many times neural networks are not used because of lack of software for training and implementing. This dissertation removes that burden and allows neural networks to be used in more main stream applications by allowing users to implement them in their applications with ease.

The same concepts presented here could be used to produce similar custom optimized assembly language hardware for other networks. In microcontrollers with greater computing power this becomes easier. The neuron by neuron approach using the arrays for weights and nodes can be taken to any platform and implemented in the same manner.

The C output file could also be used to run the neural network on a computer as well as microcontrollers. The only modification that would be required is the two type definitions which specifying that they should be stored in program memory. This would not be applicable to a personal computer based network and this line would need to be removed. Otherwise the files are platform independent.

This dissertation cannot stress enough that the proof of concept shown here opens the door for neural networks to be used on any platform for problems of virtually any kind. The complexity of the problems can range from a simple one neuron one input one output system to dozens of neurons and many inputs and outputs. This solution has endless problems it is capable of solving.

The next step in this research is to extend the C version to other microcontrollers and compare calculation times. The final step would be to train the network on data collected for a physical system. This would demonstrate and verify the speed of the neural network in a hardware application.

References

[1] B. K. Bose, "Neural Network Applications in Power Electronics and Motor Drives—An Introduction and Perspective," *IEEE Transactions on Industrial Electronics,* vol. 54, pp. 14-33, 2007.

[2] M. A. El-Sharkawi, "Neural network application to high performance electric drives systems," in *Proc. IEEE IECON 21st Int Industrial Electronics, Control, and Instrumentation Conf,* 1995, pp. 44-49.

[3] L. M. Grzesiak and B. Ufnalski, "Neural stator flux estimator with dynamical signal preprocessing," in *Proc. 7th AFRICON Conf AFRICON in Africa,* 2004, pp. 1137-1142.

[4] Y. Yusof and A. H. M. Yatim, "Simulation and modeling of stator flux estimator for induction motor using artificial neural network technique," in *Proc. National Power Engineering Conf. PECon 2003,* 2003, pp. 11-15.

[5] A. Ba-Razzouk, A. Cheriti, G. Olivier, and P. Sicard, "Field-oriented control of induction motors using neural-network decouplers," *Proc. IEEE IECON 21st Int Industrial Electronics, Control, and Instrumentation Conf,* vol. 12, pp. 752-763, 1997.

[6] S. M. Gadoue, D. Giaouris, and J. W. Finch, "Sensorless Control of Induction Motor Drives at Very Low and Zero Speeds Using Neural Network Flux Observers," *IEEE Transactions on Industrial Electronics,* vol. 56, pp. 3029-3039, 2009.

[7] C. Hudson, N. S. Lobo, and R. Krishnan, "Sensorless control of single switch based switched reluctance motor drive using neural network," in *Industrial Electronics Society, 2004. IECON 2004. 30th Annual Conference of IEEE,* 2004, pp. 2349-2354 Vol. 3.

[8] J. F. Martins, P. J. Santos, A. J. Pires, L. E. B. da Silva, and R. V. Mendes, "Entropy-Based Choice of a Neural Network Drive Model," *IEEE Transactions on Industrial Electronics,* vol. 54, pp. 110-116, 2007.

[9] H. Zhuang, K.-S. Low, and W.-Y. Yau, "A Pulsed Neural Network With On-Chip Learning and Its Practical Applications," *IEEE Transactions on Industrial Electronics,* vol. 54, pp. 34-42, 2007.

[10] J. Mazumdar and R. G. Harley, "Recurrent Neural Networks Trained With Backpropagation Through Time Algorithm to Estimate Nonlinear Load Harmonic Currents," *Industrial Electronics, IEEE Transactions on,* vol. 55, pp. 3484-3491, 2008.

[11] N. Pecharanin, H. Mitsui, and M. Sone, "Harmonic detection by using neural network," in *Proc. Conf. IEEE Int Neural Networks,* 1995, pp. 923-926.

[12] N. Pecharanin, M. Sone, and H. Mitsui, "An application of neural network for harmonic detection in active filter," in *Proc. IEEE Int Neural Networks IEEE World Congress Computational Intelligence. Conf,* 1994, pp. 3756-3760.

[13] M. Rukonuzzaman, A. A. M. Zin, H. Shaibon, and K. L. Lo, "An application of neural network in power system harmonic detection," in *Proc. IEEE Int. Joint Conf. IEEE World Congress Computational Intelligence Neural Networks,* 1998, pp. 74-78.

[14] A. A. M. Zin, M. Rukonuzzaman, H. Shaibon, and K. I. Lo, "Neural network approach of harmonics detection," in *Proc. Int. Conf. Energy Management and Power Delivery EMPD '98,* 1998, pp. 467-472.

[15] H. C. Lin, "Dynamic power system harmonic detection using neural network," in *Proc. IEEE Conf. Cybernetics and Intelligent Systems,* 2004, pp. 757-762.

[16] S. Osowski, "Neural network for estimation of harmonic components in a power system," *IEE Proceedings C Generation, Transmission and Distribution,* vol. 139, pp. 129-135, 1992.

[17] Z. Jin and B. K. Bose, "Neural-network-based waveform Processing and Delayless filtering in power electronics and AC drives," *Industrial Electronics, IEEE Transactions on,* vol. 51, pp. 981-991, 2004.

[18] M. J. Embrechts and S. Benedek, "Hybrid identification of nuclear power plant transients with artificial neural networks," *Industrial Electronics, IEEE Transactions on,* vol. 51, pp. 686-693, 2004.

[19] L. Hsiung Cheng, "Intelligent Neural Network-Based Fast Power System Harmonic Detection," *Industrial Electronics, IEEE Transactions on,* vol. 54, pp. 43-52, 2007.

[20] S. Chakraborty, M. D. Weiss, and M. G. Simoes, "Distributed Intelligent Energy Management System for a Single-Phase High-Frequency AC Microgrid," *IEEE Transactions on Industrial Electronics,* vol. 54, pp. 97-109, 2007.

[21] H. C. Lin, "Intelligent Neural Network-Based Fast Power System Harmonic Detection," *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS,* vol. 54, pp. 43-52, 2007.

[22] W. Qiao and R. G. Harley, "Indirect Adaptive External Neuro-Control for a Series Capacitive Reactance Compensator Based on a Voltage Source PWM Converter in Damping Power Oscillations," *IEEE Transactions on Industrial Electronics,* vol. 54, pp. 77-85, 2007.

[23] B. Singh, V. Verma, and J. Solanki, "Neural Network-Based Selective Compensation of Current Quality Problems in Distribution System," *IEEE Transactions on Industrial Electronics,* vol. 54, pp. 53-60, 2007.

[24] S. S. Ge and W. Cong, "Adaptive neural control of uncertain MIMO nonlinear systems," *Neural Networks, IEEE Transactions on,* vol. 15, pp. 674-692, 2004.

[25] E. B. Kosmatopoulos, M. M. Polycarpou, M. A. Christodoulou, and P. A. Ioannou, "High-order neural network structures for identification of dynamical systems," *Neural Networks, IEEE Transactions on,* vol. 6, pp. 422-431, 1995.

[26] F. L. Lewis, A. Yegildirek, and L. Kai, "Multilayer neural-net robot controller with guaranteed tracking performance," *Neural Networks, IEEE Transactions on,* vol. 7, pp. 388-399, 1996.

[27] U. D. Deep, B. R. Petersen, and J. Meng, "A Smart Microcontroller-Based Iridium Satellite-Communication Architecture for a Remote Renewable Energy Source," *IEEE Transactions on Power Delivery,* vol. 24, pp. 1869-1875, 2009.

[28] F. Ferreyre, R. Goyet, G. Clerc, and T. Bouscasse, "Sensorless Slowdown Detection Method for Single-Phase Induction Motors," *IEEE Transactions on Energy Conversion,* vol. 24, pp. 60-67, 2009.

[29] J. Kwong, Y. K. Ramadass, N. Verma, and A. P. Chandrakasan, "A 65 nm Sub- V Microcontroller With Integrated SRAM and Switched Capacitor DC-DC Converter," *IEEE Journal of Solid-State Circuits,* vol. 44, pp. 115-126, 2009.

[30] C. Labussiere-Dorgan, S. Bendhia, E. Sicard, J. Tao, H. J. Quaresma, C. Lochot, and B. Vrignon, "Modeling the Electromagnetic Emission of a Microcontroller Using a Single Model," *IEEE Transactions on Electromagnetic Compatibility,* vol. 50, pp. 22-34, 2008.

[31] D. G. Lamar, A. Fernandez, M. Arias, M. Rodriguez, J. Sebastian, and M. M. Hernando, "A Unity Power Factor Correction Preregulator With Fast Dynamic Response Based on a Low-Cost Microcontroller," *Proc. IEEE IECON 21st Int Industrial Electronics, Control, and Instrumentation Conf,* vol. 23, pp. 635-642, 2008.

[32] J. H. Lee, H. S. Bae, and B. H. Cho, "Resistive Control for a Photovoltaic Battery Charging System Using a Microcontroller," *IEEE Transactions on Industrial Electronics* vol. 55, pp. 2767-2775, 2008.

[33] S.-Y. Oh, Y.-G. Jung, S.-H. Yang, and Y.-C. Lim, "Harmonic-Spectrum Spreading Effects of Two-Phase Random Centered Distribution PWM (DZRCD) Scheme With Dual Zero Vectors," *IEEE Transactions on Industrial Electronics,* vol. 56, pp. 3013-3020, 2009.

[34] H. Esmaeilzadeh, P. Saeedi, B. N. Araabi, C. Lucas, and S. M. Fakhraie, "Neural network stream processing core (NnSP) for embedded systems," in *Proc. IEEE Int. Symp. Circuits and Systems ISCAS 2006*, 2006.

[35] G. Dundar and K. Rose, "Analog neural network circuits for ASIC fabrication," in *Proc. Fifth Annual IEEE Int. ASIC Conf. and Exhibit*, 1992, pp. 419-422.

[36] L. Gatet, H. Tap-Beteille, and M. Lescure, "Analog Neural Network Implementation for a Real-Time Surface Classification Application," *Sensors Journal, IEEE,* vol. 8, pp. 1413-1421, 2008.

[37] S. Xiao, M. H. L. Chow, F. H. F. Leung, X. Dehong, W. Yousheng, and L. Yim-Shu, "Analogue implementation of a neural network controller for UPS inverter applications," *IEEE Transactions on Power Electronics,* vol. 17, pp. 305-313, 2002.

[38] A. Rajah and M. Khalil Hani, "ASIC design of a Kohonen neural network microchip," in *Proc. IEEE Int. Conf. Semiconductor Electronics ICSE 2004*, 2004.

[39] A. Konig, P. Windirsch, M. Gasteier, and M. Glesner, "Visual inspection in industrial manufacturing," *IEEE Micro,* vol. 15, pp. 26-31, 1995.

[40] S. Himavathi, D. Anitha, and A. Muthuramalingam, "Feedforward Neural Network Implementation in FPGA Using Layer Multiplexing for Effective Resource Utilization," *IEEE Transactions on Neural Networks*, vol. 18, pp. 880-888, 2007.

[41] F.-J. Lin, S.-Y. Chen, and Y.-C. Hung, "Field-programmable gate array-based recurrent wavelet neural network control system for linear ultrasonic motor," *IET Electric Power Applications,* vol. 3, pp. 298-312, 2009.

[42] F.-J. Lin, Y.-C. Hung, and S.-Y. Chen, "FPGA-Based Computed Force Control System Using Elman Neural Network for Linear Ultrasonic Motor," *IEEE Transactions on Industrial Electronics,* vol. 56, pp. 1238-1253, 2009.

[43] F.-J. Lin and Y.-C. Hung, "FPGA-based elman neural network control system for linear ultrasonic motor," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 56, pp. 101-113, 2009.

[44] N. Funabiki, M. Yoda, J. Kitamichi, and S. Nishikawa, "A gradual neural network approach for FPGA segmented channel routing problems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 29, pp. 481-489, 1999.

[45] Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its FPGA implementation," *IEEE Transactions on Neural Networks*, vol. 16, pp. 1664-1672, 2005.

[46] D. Zhang and H. Li, "A Stochastic-Based FPGA Controller for an Induction Motor Drive With Integrated Neural Network Algorithms," *IEEE Transactions on Industrial Electronics*, vol. 55, pp. 551-561, 2008.

[47] M. Mohamadian, E. P. Nowicki, A. Chu, F. Ashrafzadeh, and J. C. Salmon, "DSP implementation of an artificial neural network for induction motor control," in *Proc. IEEE 1997 Canadian Conf. Electrical and Computer Engineering*, 1997, pp. 435-437.

[48] D. Dong, W. N. White, and H. Luo, "Investigation of kinematics and inverse dynamics algorithm with a DSP implementation of a neural network," in *Proc. American Control Conf*, 1994, pp. 2460-2464.

[49] F. Ashrafzadeh, R. Sachdeva, and A. Chu, "A novel neural network controller and its efficient DSP implementation for vector controlled induction motor drives," in *Proc. 37th IAS Annual Meeting Industry Applications Conf. Conf. Record of the*, 2002, pp. 1455-1462.

[50] M. Mohamadian, E. Nowicki, F. Ashrafzadeh, A. Chu, R. Sachdeva, and E. Evanik, "A novel neural network controller and its efficient DSP implementation for vector-controlled induction motor drives," *#IEEE_J_IA#*, vol. 39, pp. 1622-1629, 2003.

[51] T. Xu and G. Ni, "Research on Algorithms of Neural Network Ensemble with Multi-dsp Mixture Structure," in *Proc. Third Int. Conf. Natural Computation ICNC 2007*, 2007, pp. 162-166.

[52] F. F. M. El-Sousy, "Robust adaptive H∞ position control via a wavelet-neural-network for a DSP-based permanent-magnet synchronous motor servo drive system," *IET Electric Power Applications*, vol. 4, pp. 333-347, 2010.

[53] F.-J. Lin, P.-H. Chou, and Y.-S. Kung, "Robust fuzzy neural network controller with nonlinear disturbance observer for two-axis motion control system," *IET Control Theory & Applications*, vol. 2, pp. 151-167, 2008.

[54] E. Echenique, J. Dixon, R. Cardenas, and R. Pena, "Sensorless Control for a Switched Reluctance Wind Generator, Based on Current Slopes and Neural Networks," *IEEE Transactions on Industrial Electronics,* vol. 56, pp. 817-825, 2009.

[55] B. M. Wilamowski, "Neural network architectures and learning algorithms," *IEEE Industrial Electronics Magazine,* vol. 3, pp. 56-63, 2009.

[56] B. M. Wilamowski, "Special neural network architectures for easy electronic implementations," in *Proc. Int. Conf. Power Engineering, Energy and Electrical Drives POWERENG '09*, 2009, pp. 17-22.

[57] B. M. Wilamowski, N. Cotton, J. Hewlett, and O. Kaynak, "Neural Network Trainer with Second Order Learning Algorithms," in *Proc. 11th Int. Conf. Intelligent Engineering Systems INES 2007*, 2007, pp. 127-132.

[58] B. M. Wilamowski, N. J. Cotton, O. Kaynak, and G. Dundar, "Method of computing gradient vector and Jacobean matrix in arbitrarily connected neural networks," in *Proc. IEEE Int. Symp. Industrial Electronics ISIE 2007*, 2007, pp. 3298-3303.

[59] B. M. Wilamowski, N. J. Cotton, O. Kaynak, and G. Dundar, "Computing Gradient Vector and Jacobian Matrix in Arbitrarily Connected Neural Networks," *IEEE Transactions on Industrial Electronics,* vol. 55, pp. 3784-3790, 2008.

[60] B. M. Wilmowski, "New Trends in Neural and Fuzzy Systems," in *Proc. Int. Conf. Intelligent Engineering Systems INES 2008*, 2008, p. 9.

[61] H. Yu and B. M. Wilamowski, "C++ implementation of neural networks trainer," in *Proc. Int. Conf. Intelligent Engineering Systems INES 2009*, 2009, pp. 257-262.

[62] H. Yu and B. M. Wilamowski, "Efficient and reliable training of neural networks," in *Proc. 2nd Conf. Human System Interactions HSI '09*, 2009, pp. 109-115.

[63] B. M. Wilamowski, D. Hunter, and A. Malinowski, "Solving parity-N problems with feedforward neural networks," in *Proc. Int Neural Networks Joint Conf*, 2003, pp. 2546-2551.

[64] P. Petchjatuporn, W. Ngamkham, N. Khaehintung, P. Sirisuk, W. Kiranon, and A. Kunakorn, "A Solar-powered Battery Charger with Neural Network Maximum Power Point Tracking Implemented on a Low-Cost PIC-microcontroller," in *TENCON 2005 2005 IEEE Region 10*, 2005, pp. 1-4.

[65] U. Farooq, M. Amar, K. M. Hasan, M. Khalil Akhtar, M. U. Asad, and A. Iqbal, "A low cost microcontroller implementation of neural network based hurdle avoidance controller for a car-like robot," in *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*, 2010, pp. 592-597.

[66] S. Bashyal, G. K. Venayagamoorthy, and B. Paudel, "Embedded neural network for fire classification using an array of gas sensors," in *Sensors Applications Symposium, 2008. SAS 2008. IEEE*, 2008, pp. 146-148.

[67] M. Xiaoying, T. Yuening, and H. Jia, "A Fuzzy Neural Network Control System Based on Embedded System," in *Mechatronics and Automation, 2007. ICMA 2007. International Conference on*, 2007, pp. 2394-2398.

[68] J. Binfet and B. M. Wilamowski, "Microprocessor implementation of fuzzy systems and neural networks," in *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on*, 2001, pp. 234-239 vol.1.

[69] G. L. Dempsey, N. L. Alt, B. A. Olson, and J. S. Alig, "Control sensor linearization using a microcontroller-based neural network," in *Systems, Man, and Cybernetics, 1997. 'Computational Cybernetics and Simulation'., 1997 IEEE International Conference on*, 1997, pp. 3078-3083 vol.4.

[70] M. Mestari, "An analog neural network implementation in fixed time of adjustable-order statistic filters and applications," *Neural Networks, IEEE Transactions on*, vol. 15, pp. 766-785, 2004.

[71] J. R. Stack, G. J. Dobeck, X. Liao, and L. Carin, "Kernel-Matching Pursuits With Arbitrary Loss Functions," *IEEE Transactions on Neural Networks*, vol. 20, pp. 395-405, 2009.

[72] S. Tabarce, V. G. Tavares, and P. G. de Oliveira, "Programmable analogue VLSI implementation for asymmetric sigmoid neural activation function and its derivative," *Electronics Letters*, vol. 41, pp. 863-864, 2005.

[73] N. Zhang and D. C. Wunsch II, "A Switched-Resistor Approach to Hardware Implementation of Neural Networks," in *Proc. 14th IEEE Int. Conf. Fuzzy Systems FUZZ '05*, 2005, pp. 336-340.

[74] B. M. Wilamowski and J. Binfet, "Do Fuzzy Controllers Have Advantages over Neural Controllers in Microprocessor Implementation," presented at the Proc of.2-nd International Conference on Recent Advances in Mechatronics, Istanbul, Turkey, 1999.

[75] J. Binfet and B. M. Wilamowski, "Microprocessor implementation of fuzzy systems and neural networks," in *Proc. Int. Joint Conf. Neural Networks IJCNN '01*, 2001, pp. 234-239.

[76] N. J. Cotton, B. M. Wilamowski, and G. Dundar, "A Neural Network Implementation on an Inexpensive Eight Bit Microcontroller," in *Proc. Int. Conf. Intelligent Engineering Systems INES 2008*, 2008, pp. 109-114.

[77] G. L. Dempsey, N. L. Alt, B. A. Olson, and J. S. Alig, "Control sensor linearization using a microcontroller-based neural network," in *Proc. IEEE Int Systems, Man, and Cybernetics 'Computational Cybernetics and Simulation'. Conf*, 1997, pp. 3078-3083.

[78] U. Farooq, M. Amar, K. M. Hasan, M. Khalil Akhtar, M. U. Asad, and A. Iqbal, "A low cost microcontroller implementation of neural network based hurdle avoidance controller for a car-like robot," in *Proc. 2nd Int Computer and Automation Engineering (ICCAE) Conf*, 2010, pp. 592-597.

[79] U. Farooq, M. Amar, E. ul Haq, M. U. Asad, and H. M. Atiq, "Microcontroller Based Neural Network Controlled Low Cost Autonomous Vehicle," in *Proc. Second Int Machine Learning and Computing (ICMLC) Conf*, 2010, pp. 96-100.

[80] N. Cotton, "Training Arbitrarily Connected Neural Networks with Second Order Algorithms," Masters of Science, Electrical and Computer Engineering, Auburn University, Auburn, 2008.

[81] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, pp. 989-993, 1994.

[82] J. Hewlett, B. Wilamowski, and G. Dundar, "Merge of Evolutionary Computation with Gradient Based Method for Optimization Problems," in *Proc. IEEE Int. Symp. Industrial Electronics ISIE 2007*, 2007, pp. 3304-3309.