A Neural Network Implementation on an Inexpensive Eight Bit Microcontroller

Nicholas J. Cotton, and Bogdan M. Wilamowski,
Electrical and Computer Engineering
Auburn University
Auburn, AL 36849 United States
cottonj@auburn.edu, wilam@ieee.org

Günhan Dündar
Electrical and Electronic Engineering
Bogazici University
Istanbul, Turkey
dundar@boun.edu.tr

***Abstract*—The paper describes a neural network implementation on a low end and inexpensive microcontroller. It also describes the method of using a simple hardware multiplier to generate multibyte accurate results. An activation function that is equivalent to tangent hyperbolic is also described. An example is shown using an inexpensive eight bit microcontroller is used for a nonlinear control surface.**

I. Introduction

Neural networks have become a growing area of research over the last few decades and have taken hold many branches of industry. One example in the field of industrial electronics would be Motor Drives according to [1-3]. Neural networks are also helping with power distribution problems such as harmonic distortion [4-7]. These papers show how valuable neural networks are becoming in industry. One common drawback is that most software implemented networks require precise calculations that are very processor intensive. This robust processing equipment can be expensive and rather large.

This paper describes a method implanting a fully connected multi-layer network with multi-layer connections on a very inexpensive and low end microcontroller. Fully connected networks are much more powerful and easier to train when using the Neural Network Trainer [8], [9] than traditional layer by layer networks. Cross layer connections reduce the number of neurons by increasing the number of connections and corresponding weights. This in turn reduces the total number of calculations required for the network.

The single most important part of a neural network is the activation function. This paper describes a method of approximating the tangent hyperbolic (tanh) function with great precision. In fact, it is done with such precision the network may be trained offline using the Neural Network Trainer which incorporates tanh as its activation function.

This paper has the following subsections: Section two describes the fast and simple sixteen bit multiplication. Section three describes how the pseudo floating point multiplication is implemented on the microcontroller. Section four explains the approximation of tanh and its implementation on the microcontroller. In the fifth section an example of a neural network is implemented on the microcontroller and the results are described. A Matlab simulation of the microcontroller is described in section six.

II. Fast Multiplication

The Pic18F45J10 microcontroller has an 8-bit by bit hardware multiplier. The hardware multiplier cannot handle floating point values or negative numbers, therefore a routine was developed to allow fast multiplication of fractional values using this hardware multiplier. The multiply routine is given two sixteen bit numbers. The first eight bits is the integer portion and the last eight bits is the fractional portion. The result of the multiplication routine is a thirty-two bit fixed point result see Figure 1.

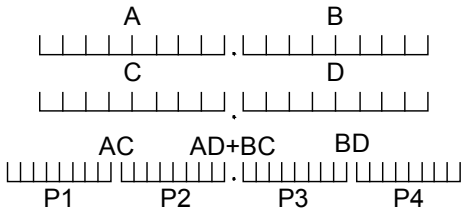

Fig. 1. Result of 16-bit fixed point multiplication using 8-bit hardware multiplier.

Equation 1 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 while the magnitude is adjusted by the results location in the product. The 16-bit result of AC is placed in P1 and P2 see Figure 1. The sum of AD and BC is then added to P2 and P3 and finally BD is added to P3 and P4. Once the product is calculated P1 and P2 contain a 16-bit integer while P3 and P4 contain the 16-bit fractional part. 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 P2 and P3 are required for the activation function. If the absolute value of the net value is greater than 4, the neuron is in saturation and the activation function is skipped resulting in positive one or negative one respectively.

$$\frac{A \cdot C \cdot 256^2 + 256 \cdot (A \cdot D + B \cdot C) + B \cdot D}{256^2} \tag{1}$$

978-1-4244-2083-4/08/$25.00 ©2008 IEEE

III. Pseudo Floating Point

During the neural network training process precise weights are created to generate a series of outputs. Due to the importance of the weights in the output of the neural network, a pseudo floating point system is implemented. The weights of each neuron are scaled in order to use the maximum number of bits available in each calculation. This way unnecessary round-off error is avoided. The weights are scaled as necessary by the user as they are placed into the weight array. In addition to storing the weights the location of the decimal point is also stored by the user. Without this floating point structure when the product of very small numbers is calculated the error is unnecessarily large. In some cases the weights are out of operating range of the microcontroller and an overflow condition can occur. In this case the surface becomes completely unrecognizable. Using the pseudo floating point arithmetics insures that all values remain within the operating range and overflow does not occur.

A similar technique is also used on the inputs in order to help smooth the surface. The inputs are also scaled to maximize the number of bits used for the calculations. The scaling is done by factors of two only so no actual multiplication or division is required but simply shifts. In order to properly take care of the shifts, the inputs can be shifted at the beginning of the calculation and the weights are already scaled. The bias, which is normally one, is replaced with the input scale factor see equation 2.

$$\text{Net} = \text{Act}\left(\frac{(\text{input1}\cdot \text{ww1} + \text{input2}\cdot \text{ww2} + \text{bias}\cdot \text{ww3})}{\text{Weight Scale}\cdot \text{Input Scale}}\right) \quad (2)$$

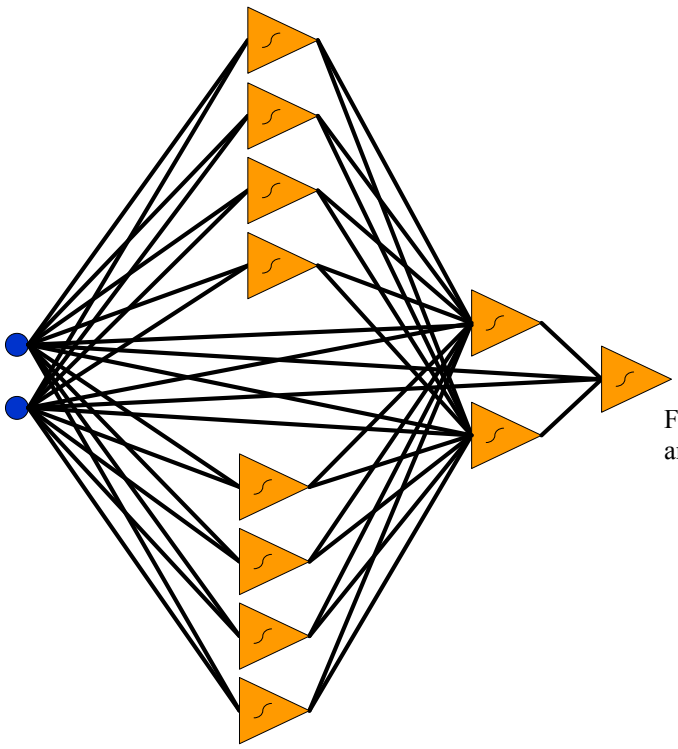

Fig. 2. Fully connected architecture of network used to calculate Figure 3.

An example of a relatively large network with 3 layers is shown in Figure 2. With the three-layer network, a large error occurs without the pseudo floating point calculations. This network was trained to the surface in Figure 3. To help show the importance of these scaling factors Figure 4 and Figure 6 were created. These Figures are an example of a control surface with the pseudo floating point arithmetics disabled and the error created. Notice the distortion around the edges of the surface compared to the ideal surface in Figure 3. The errors between the two surfaces were plotted in Figure 5. Now the surface in Figure 6 shows the same surface, and Figure 7 shows the error when the pseudo floating point arithmetics is enabled.

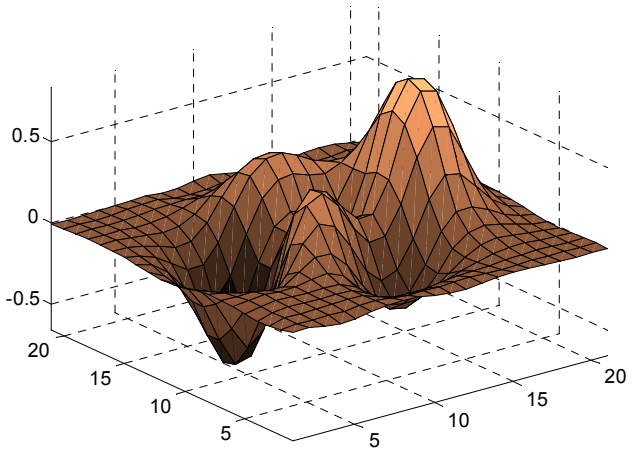

Fig. 3. Required surface.

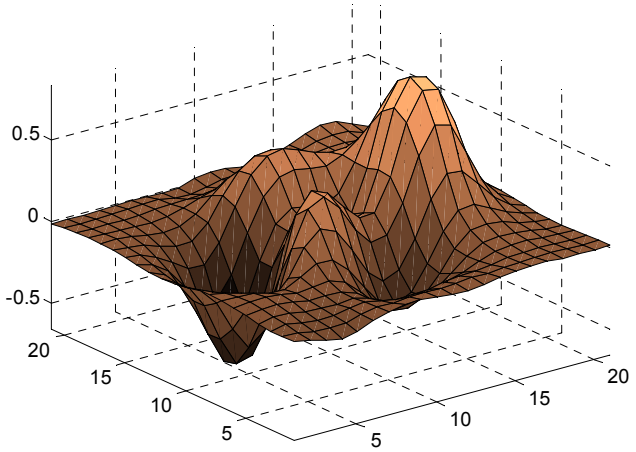

Fig. 4. Neural network surface with pseudo floating point arithmetics disabled.

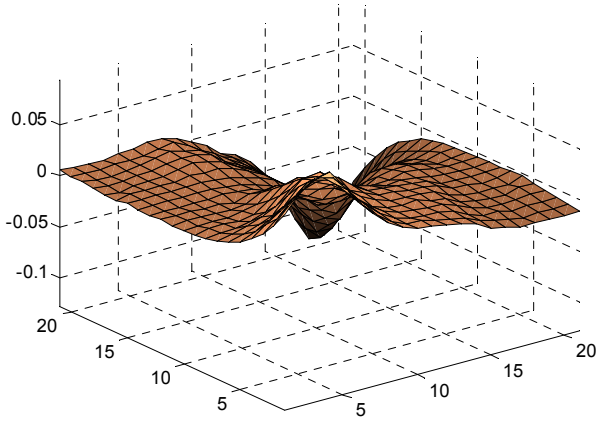

Fig. 5. Difference between Figures 3 and 4.

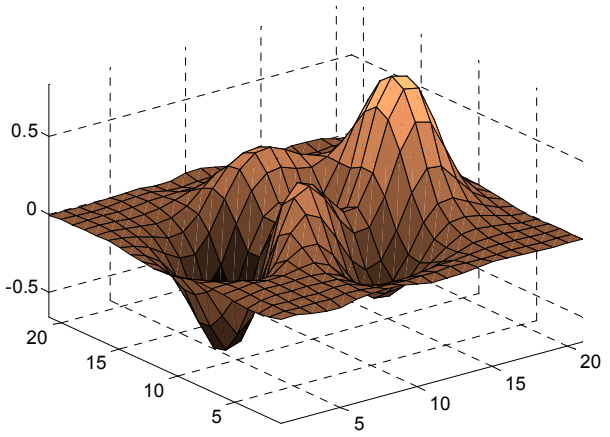

Fig. 6. Surface with pseudo floating point arithmetics enabled.

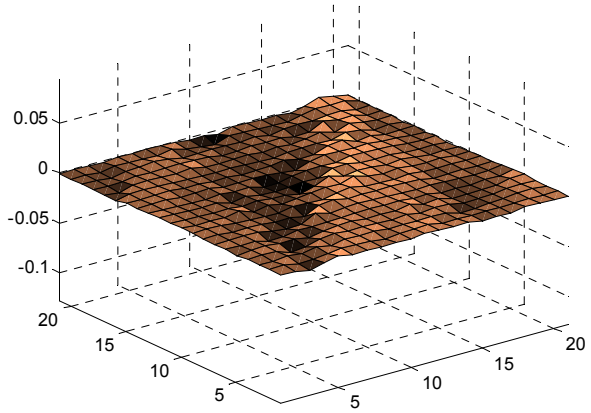

Fig. 7. Difference between Figure 6 and 3.

IV. Activation Fuction

One of the most challenging aspects of implementing a neural network on a microcontroller is creating an activation function. This system was designed using a pseudo tangent hyperbolic activation function. This is also very important because it allows the network to be trained offline using the Neural Network Trainer [8]. The software will generate the needed weights for any arbitrarily connected network which is what the embedded system uses. Creating this activation function is a challenge because of the limitations of the microcontroller. Many other similar designs use techniques such as the Elliot Function but these require the processor to implement division. Another common approach is a fourth-order polynomial, but this approach is very computationally intensive and requires extremely accurate values.

This problem is solved using a simple and fast routine. 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 six 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. Figures 8 and 9 show this procedure in more detail.

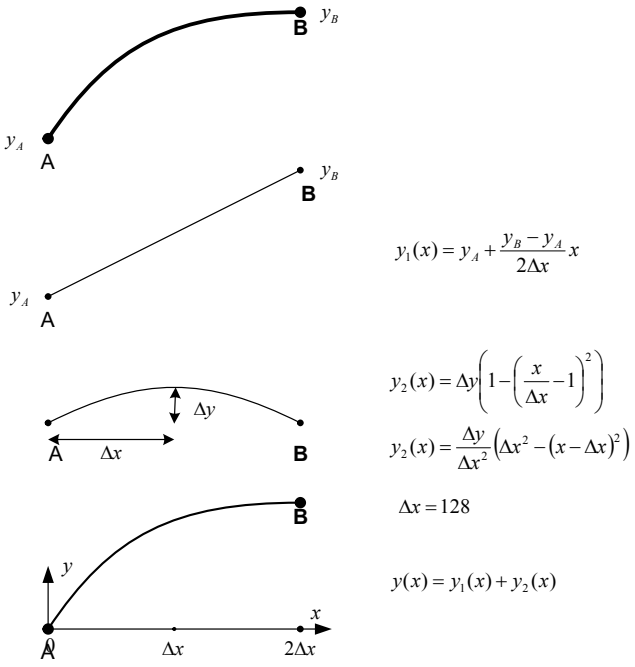

Fig. 8. Tanh approximations.

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.

$$y_1(x) = y_A + \frac{(y_B - y_A)\cdot x}{2\Delta x} \tag{3}$$

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.

$$y_2(x) = \frac{\Delta y}{\Delta x^2}\left(\Delta x^2 - (x - \Delta x)^2\right) \tag{4}$$

or

$$y_2(x) = \frac{\Delta y \cdot x \cdot (2\Delta x - x)}{\Delta x^2} \tag{5}$$

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. Using this activation function and the multiply routine discussed earlier, an accurate error plot was generated for the activation function using the 8- bit arithmetics of the microcontroller in Figure 10

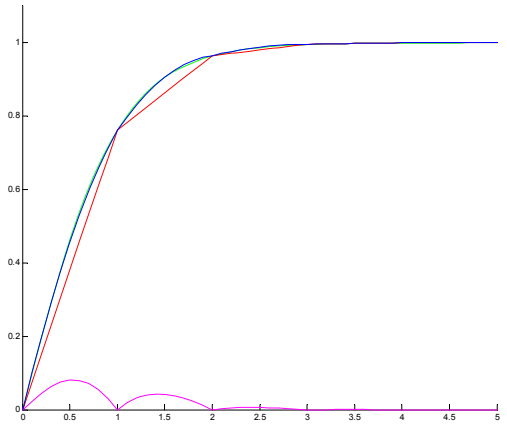

Fig. 9. Example of linear approximations and parabolas between 0 and 4. Only 4 divisions were used for demonstration purposes.

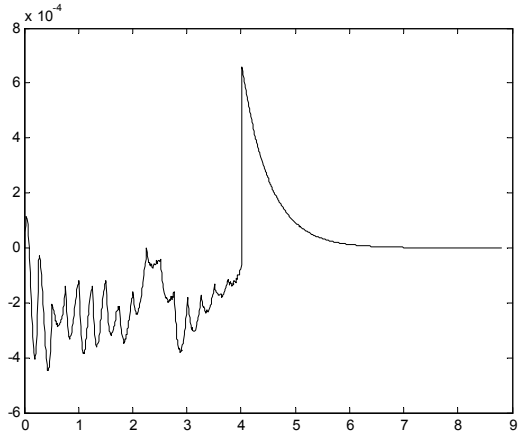

Fig. 10. Error from tanh approximation using 16 divisions from 0 to 4.

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 arithmetics 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.

V. PIC IMPLEMENTATION

The Neural Network implementation is written in assembly language for the Microchip Microcontroller (PIC). The neural network forward calculations are written so that each neuron is calculated individually in a series of nested loops. The number of calculations for each loop and values for each node are all stored in a simple array in memory. The assembly language code does not require any modification to change network's architecture. The only change that is required is to update two arrays that are loaded into memory. These arrays contain the architecture and the weights of the network. Currently, for demonstration purposes, these arrays are preloaded at the time the chip is programmed, but in the final version this would not be necessary. The microcontroller could easily be modified to contain a simple boot loader that makes use of the onboard RS232 serial port which would receive the data for the weights and topography from a remote location. This would allow for the weights or even the entire network to be modified while the chip is in the field.

The input arrays are formatted in a simple way that allows the microcontroller to read the information as is needed. The following example, Figure 11, is a simple three-neuron fully connected network. This network could be used for a three-dimensional nonlinear control surface shown in Figure 12.

In order to load the neural network onto the PIC, the user needs to follow the following steps. First the user should generate input patterns, outputs, and decide on an appropriate architecture. This information is used by the NNT to train the neural network as described in [9]. As an example, we will use the network shown if Figure 11 to generate the surface shown in Figure 12. The weights that are output from the NNT need to be scaled to allow the PIC to use the pseudo floating point arithmetics. Second the user needs to categorize the weight array by neurons. Then the weights are scaled so they are as close to 128 without exceeding it. This scaling uses only factors of two. An example has been given below.

Neuron 1
-1.0340 -1.5983 1.6029 Scale by 64 Neuron 2
0.5453 -1.0880 -0.3181 Scale by 64 Neuron 3
1.4489 -9.390 -1.5794 -9.0460 0.0683 Scale by 8

These new weights are then input directly into the PIC's lookup table in the same order. The next step is to generate an input array that is similar to a Pspice net list. This array contains neuron connections and scaling factors for the PIC. The input arrays are formatted in a simple way that allows the microcontroller to read the information as it is needed. The input array for Figure 11 is generated below.

Generic format
[Input Scale; Number of Neurons; Weight Scale Factor, Number of Connections, Output Node, Connection1, Connection2,…ConnectionN; Next Neuron; Next Neuron…]
The Topography array for Figure 11
[32; 3; 64 3 3 2 1; 64 3 4 2 1; 8 5 5 4 3 2 1]
Note: The input scale connection (bias) is assumed.

The result of the weights and scaling factors given above is the surface shown if Figure 13. The difference between the surfaces is shown in Figure 14.

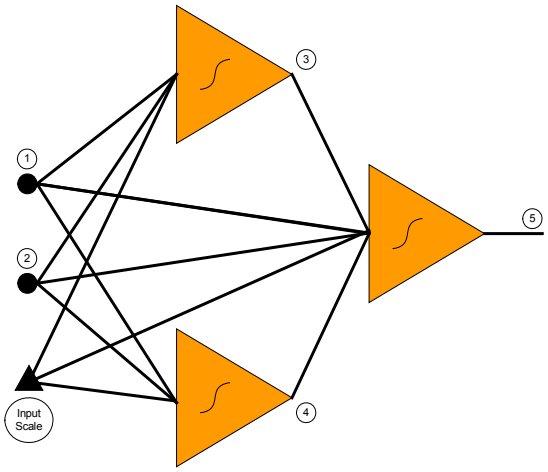

Fig. 11. Network used to generate surfaces shown in Figures 12 and 13.

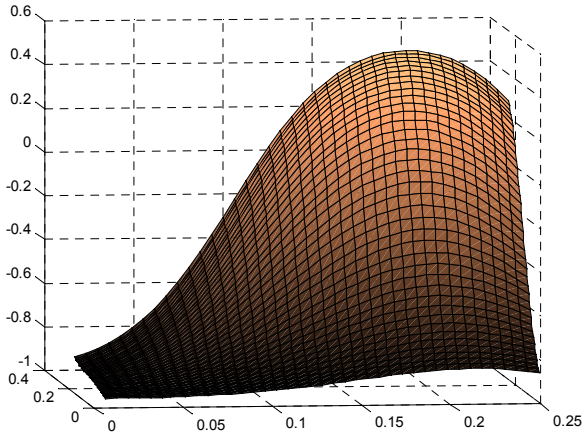

Fig. 12. Required surface using Tanh and IEE 754 floating point calculations

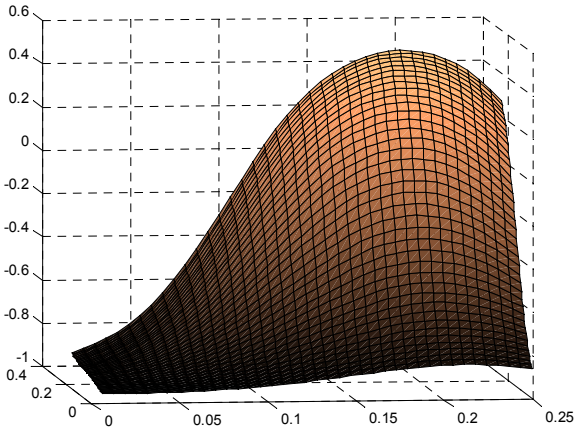

Fig. 13. Surface using approximation of tanh and PIC based mathematics with pseudo floating point.

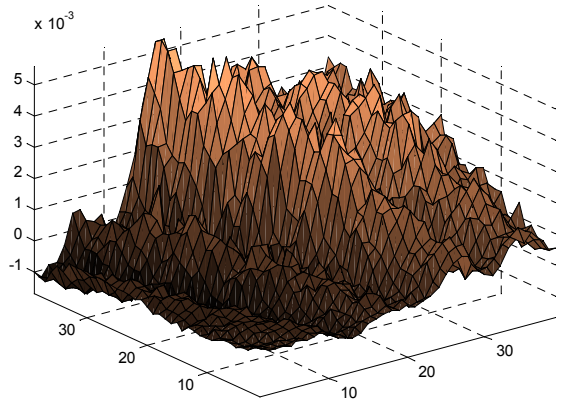

Fig. 14. Difference between Figures 12 and 13.

VI. MATLAB SIMULATION

To help test the multiplication and activation algorithm, a Matlab simulator was built. This was absolutely necessary because the microcontroller introduces round-off errors while computing its 16-bit fixed point multiplication. In order to simulate this round-off error, routines were written that round every calculation to its 8-bit counterpart. The round-off routine is given in equation 6.

$$y = floor(x \cdot 256) \cdot 256 \tag{6}$$

The multiply routine is organized as described above in the Multiply section. Handling negatives needs special care when modeled in Matlab. At the beginning of the Matlab routine the negative numbers are converted to the two's complement, shown in equation 7. Then the multiplication routine is carried out normally. At the end of the routine, the numbers are converted back to standard positive and negative decimals to allow normal addition and subtraction in the other steps. This conversion from two's complement is done with few subtractions. The final result is tested to see if it underflows because when calculated on the microcontroller it would rollover to a positive number. Matlab must simulate this rollover with an extra addition as seen in equation 8.

$$\begin{aligned} &a.b \cdot c.d = p \\ &a.b < 0 \quad \Rightarrow \quad p = p - (c - d) \cdot 256 \\ &c.d < 0 \quad \Rightarrow \quad p = p - (a - b) \cdot 256 \\ &p < -65536 \quad \Rightarrow \quad p = p + 65536 \end{aligned} \tag{7}$$

$$y = 256 - |x| \tag{8}$$

VII. Conclusion

This paper has shown the need for neural networks in many applications especially those where a computer may not be practical. For these applications a microcontroller is typically the solution. A solution has been presented to implement powerful neural networks on a microcontroller that is quite inexpensive. This method uses a pseudo floating point multiplication system and a very accurate tangent hyperbolic approximation. Currently the limitation on the architecture embedded in this microcontroller is limited only by the number of weights needed. The neural network is currently limited to 256 weights. However for most embedded applications this 256 weight should not limit the system.

Acknowledgments

This work was supported by NSF international grant US-Turkey Cooperative research: Silicon implementation of computational intelligence for mechatronics. NSF OISE 0352771

References

[1] Bose, B. K., "Neural Network Applications in Power Electronics and Motor Drives—An Introduction and Perspective," *Industrial Electronics, IEEE Transactions on* , vol.54, no.1, pp.14-33, Feb. 2007

[2] Zhuang, H.; Low, K.; Yau, W., "A Pulsed Neural Network With On-Chip Learning and Its Practical Applications," *Industrial Electronics, IEEE Transactions on* , vol.54, no.1, pp.34-42, Feb. 2007

[3] Martins, J. F.; Santos, P. J.; Pires, A. J.; Luiz Eduardo Borges da Silva; Mendes R. V., "Entropy-Based Choice of a Neural Network Drive Model," *Industrial Electronics, IEEE Transactions on* , vol.54, no.1, pp.110-116, Feb. 2007

[4] Lin H. C., "Intelligent Neural Network-Based Fast Power System Harmonic Detection," *Industrial Electronics, IEEE Transactions on* , vol.54, no.1, pp.43-52, Feb. 2007

[5] Singh, B.; Verma, V.; Solanki, J., "Neural Network-Based Selective Compensation of Current Quality Problems in Distribution System," *Industrial Electronics, IEEE Transactions on* , vol.54, no.1, pp.53-60, Feb. 2007.

[6] Qiao, W.; Harley, R. G., "Indirect Adaptive External Neuro-Control for a Series Capacitive Reactance Compensator Based on a Voltage Source PWM Converter in Damping Power Oscillations," *Industrial Electronics, IEEE Transactions on* , vol.54, no.1, pp.77-85, Feb. 2007.

[7] Chakraborty, S.; Weiss, M. D.; Simoes, M. G., "Distributed Intelligent Energy Management System for a Single-Phase High-Frequency AC Microgrid," *Industrial Electronics, IEEE Transactions on* , vol.54, no.1, pp.97-109, Feb. 2007

[8] Wilamowski, B. M.; Cotton, N.; Hewlett, J.; Kaynak, O., "Neural Network Trainer with Second Order Learning Algorithms," *Intelligent Engineering Systems, 11th International Conference on* , vol., no., pp.127-132, June 29 2007-July 1 2007.

[9] Wilamowski, B. M.; Cotton, N. J.; Kaynak, O.; Dundar, G., "Method of computing gradient vector and Jacobean matrix in arbitrarily connected neural networks," *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on* , vol., no., pp.3298-3303, 4-7 June 2007.

[10] Binfet, J.; Wilamowski, B.M., "Microprocessor implementation of fuzzy systems and neural networks ," *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on* , vol.1, no., pp.234-239 vol.1, 2001

[11] Wilamowski, B.M.; Iplikci, S.; Kaynak, O.; Efe, M.O., "An algorithm for fast convergence in training neural networks," *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on* , vol.3, no., pp.1778-1782 vol.3, 2001