Compensation of Sensors Nonlinearity with Neural Networks

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

***Abstract*—This paper describes a method of linearizing the nonlinear characteristics of many sensors using an embedded neural network. The proposed method allows for complex neural networks with very powerful architectures to be embedded on a very inexpensive 8-bit microcontroller. In order to accomplish this unique training software was developed as well as a cross compiler. The Neuron by Neuron process was as developed in assembly language to allow the fastest and shortest code on the embedded system. The embedded neural network also required an accurate approximation for hyperbolic tangent to be used as the neuron activation function. This process was then demonstrated on a robotic arm kinematics problem.**

Keywords-component; Neural Networks, Embedded, Nonlinear Sensor Compenstatoin, Microcontroller.

I. INTRODUCTION

One common cause of nonlinearity in otherwise linear control systems is the sensors. By linearizing the sensors the system as a whole becomes easier to control and often a simple PID controller is adequate. The nonlinear compensations can be performed using neural networks imbedded in inexpensive microcontrollers. Another advantage of the proposed approach is to convert the sensor outputs to a digital format that can easily be transmitted relatively long distances without distortion.

Sensors can be divided into three categories:

1. Linear sensors where output signal is proportional to measured value.
2. Nonlinear Sensors where output is nonlinear function of the measured value. Examples of such nonlinear sensor are a thermostat or capacitive sensors measuring distance between plates.
3. So called sensorless measurement where measured vales are estimated indirectly. One example of such sensorless sensors measurement estimation of torque and position of the rotor in an inductive motor by measuring electrical parameters on the output terminals of the motor [6,7]. A more complex example of this approach is the measurement of the parameters of the "Oil Well Diagnosis by Sensing Terminal Characteristics of the Induction Motor" by measuring the characteristics of the electrical motor driving the oil well [5].

The sensors the second and third category require relatively advanced signal conversion. In the case of the second type only nonlinear transformations of one parameter are usually required. This linearization has been accomplished using neural networks for multiple applications [8,9]

In the case of the "sensorless" approach a complex nonlinear transformation of several variables are needed. Such complex transformations cannot be done with look-up-tables (LUT) because for multi dimensional transformations the size of the LUT would be too large to be practical. Also fuzzy systems have difficulties to transform several variables and transformations are not smooth enough to be useful. Such nonlinear transformations can be done efficiently using neural networks, but their practical implementation face another challenge. Until now neural networks are mostly implemented on computers with significant computational abilities to solve many types of real world problems[1-4]. Many people have put neural networks on FPGAs, DSPS, or high end embedded processors such as the ARM cores[2][4].

In this paper it is shown that it is possible to implement relatively complex neural networks on one of the simplest microcontrollers available the PIC microcontroller made by Microchip. Such implementation was possible because of several improvements. . In order to fully utilize the power of neural networks, particularly powerful architectures were used with arbitrarily connected neurons. In order to automate the process a new Training tool and cross compiler was developed for fast and efficient assembly code generation. Assembly language implementation of the Neuron by Neuron approach which allows for faster and shorter code. Next is the pseudo floating point calculations which allow for integer computation complexity to be used for high accuracy computation. Also a new implementation of the activation function which allows for fast and accurate methods of calculations of hyperbolic tangent (*tanh*) was produced. Finally, an example calculation of the position of a robotic arm based on simulated sensor data.

II. ABITRARILY CONNECTED NETWORKS

Neural networks are most powerful in certain configurations. It has been shown that fully connected networks are easier to train and produce better results with smaller networks [13-16].

Fully connected networks are extremely powerful compared to the most common multi layer perceptron (MLP) networks. A great example of this is where a double spiral problem was solved using MLP networks with 35 to 38 neurons [12]. The same problem has been solved with as few as 8 neurons with a fully connected architecture shown in Figures 1 and 2. This is why fully connected networks are

1550-445X/10 $26.00 © 2010 IEEE
DOI 10.1109/AINA.2010.170

chosen over MLP networks. Most all sensor linearization problems are far less complicated than this double spiral problem. This particular problem is considered a very difficult problem to solve with neural networks [10,11].

In order to effectively train arbitrarily connected neural networks a new training software was developed in Matlab. It allows the user to train networks with any feed forward architecture. It trains using the Neuron By Neuron method as described previously. Other training tools such as Matlab's Neural Network Toolbox does not allow connections across the layers. This Neuron by Neuron method uniquely allows networks to be trained more efficiently. This has resulted in the ability to solve very difficult problems such as the double spiral problem on inexpensive 8-bit microcontrollers. These microcontrollers cost less than two dollars and do not even have a divide function yet they are able to handle complex neural networks.

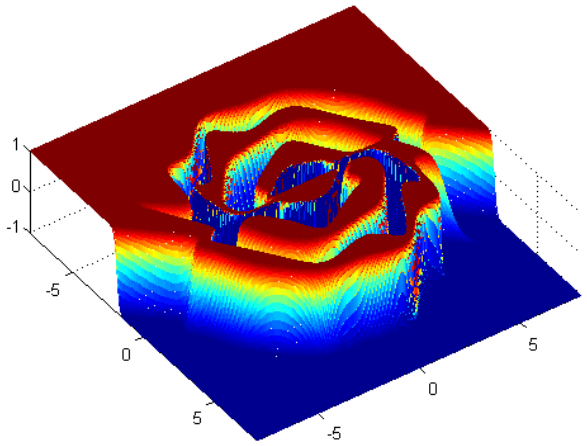

Figure 1. Double Spiral problem trained with 8 neurons fully connected architecture shown in figure 2.

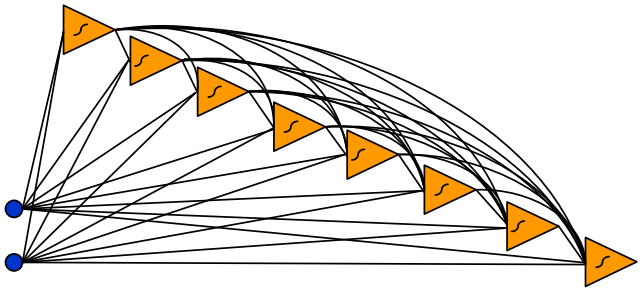

Figure 2. Fully connected eight neuron architecture for solving double spiral problem.

III. Neuron By Neuron Process in Assembly

Assembly language was chosen for the embedded neural network to optimize for faster and more memory efficient code. In order to automate the process of converting the neural network architectures from a text file used for training to assembly language a Matlab Cross compiler was created. The networks are trained in Matlab using the Neural Network Trainer (NNT) as described in section XX. NNT was modified to incorporate a cross compiler that generates assembly and C text files to for easy programming of the Microcontroller. This process allows any neural network architecture to be trained and implemented in the hardware system in a matter of minutes with no room for human error. The actual assembly calculations only need to follow the forward calculation process. The training and initial floating point values are calculated in Matlab prior to being programmed to the microcontroller to speed up the process.

This process of forward calculations is a unique method compared to most neural network implementations because it is uses the Neuron By Neuron method described in [15]. 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 3. 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 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.

Topology Array [8-bits]
[Input Scale; Number of Neurons; Weight Scale (1); Number of Connections(1), Output Node(1), Connection(1a), Connection(1b),…Connection(1n); Weight Scale (2), Number of Connections(2), Output Node(2), Connection(2a), Connection(2b),… Connection(2n);… Weight Scale (n), Number of Connections(n), Output Node(n), Connection(na), Connection(nb),…Connection(nn);]

Weight Array [16 bits]
[Number of Weights; Weight(1);Weight(2);…Weight(n)];

The arrays automatically generated by the NNT as described in Section **Error! Bookmark not defined.**. 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 allowing the user to make as many cross layer connections as desired only limited by the total number of weights.

As seen in Figure the network starts with an initialization block that configures the microcontroller by setting up the hardware for inputs and outs. 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 loop 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 process of the individual calculation can be seen in more detail in Section 4.2. The Activation Function details can be seen in Section 5.

After the Activation Function is finished the Network loop determines when all neurons have been calculated. Once they are finished it removes the scale factor and sends the output. The process is then repeated indefinitely. The details of the pseudo floating point arithmetic is shown in Figure 3.

IV. Pseudo Floating Arithmetic

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. This 16 bits consisted 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 entire 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 around 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 allow 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 to the microcontroller. This process has been automated for ease of use. The Neural Network Trainer [15] was modified to automatically scale the weights and inputs after it trains the network. 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 all 16-bits of the mantissa.

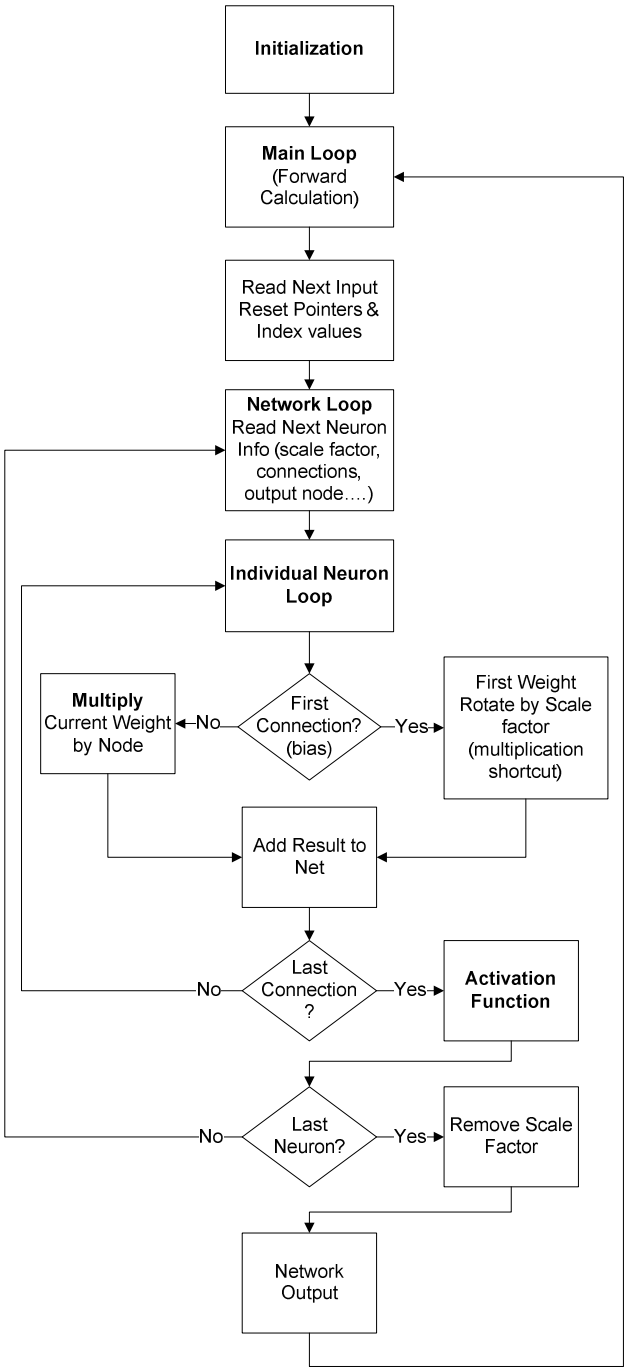

Figure 3 Block diagram of Neural Network forward calculations using the nested loop structure for cross layer connected networks.

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 .

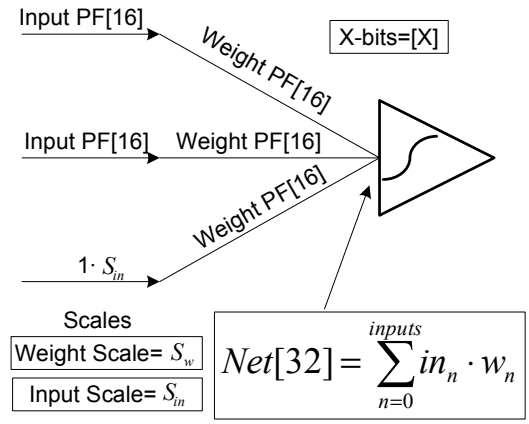

Figure 4. PF stands for Pseudo Floating point number. The Numbers in brackets refer to the number of bits that represent that particular value.

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 present 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 the figure 5.

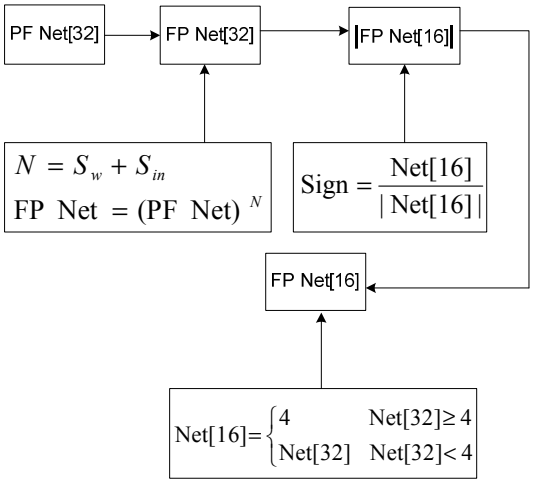

Figure 5. Pre Activation Function Routine. The transformation between a pseudo floating point number to a fixed point number that the activation function can use.

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 Figure 5 is used. This raised to the *N* power is always the same as shift by *N* because the way the scale factors are calculated as described in Section **Error! Reference source not found.**. This makes the scaling process very fast opposed to having to actually execute the multiplication instructions. Next, the sign of the net value is stored and the absolute value of 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 in the following section.

V. ACTIVATION FUNCTION

A Soft activation function was needed for the neural network. The most common activation function is *tanh* and the definition is shown below.

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

The pure definition *tanh* was not a reasonable solution for several reasons. Mainly the exponents would be very difficult to calculate accurately with the limited hardware in a timely fashion. Also the floating point division would also be very time consuming without any hardware such hardware. The next possible activation function was to use Elliott's function shown in Equation 3.

$$\frac{n}{|n|+1} \tag{2}$$

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 will be too slow of a process for the final solution. The solution chosen was a second order approximation of *tanh*.

Several features were added to the activation function besides simply calculating a second order approximation of *tanh*. One of these features analyze the inputs to the activation function and convert negative numbers to positive to make the internal calculations faster and reducing the number of values that must be stored in the lookup table. The sign is then restored at the end of the activation function. Another feature that is added is a check for numbers that when calculated will round to either extreme. In this case the second order approximation is skipped and the neuron is put into saturation. These features that incase the second order approximation can be seen in better detail in Figure .

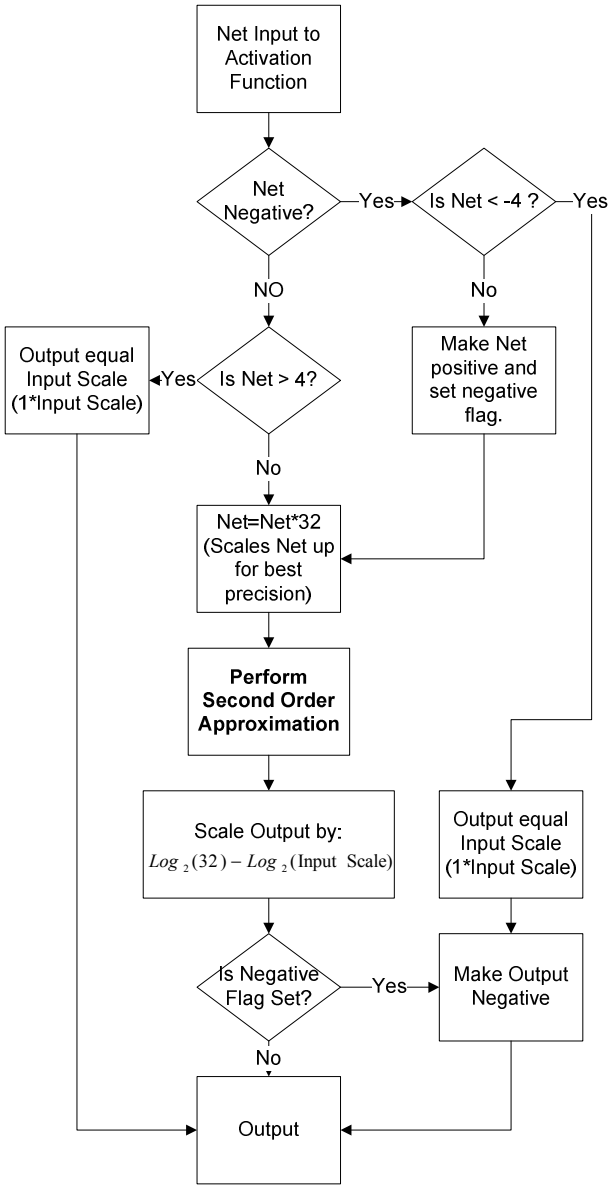

Figure 6. Logical block diagram of the activation function.

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.

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

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.

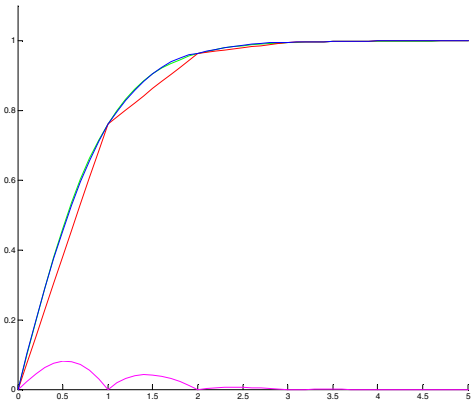

Figure 7. Example of linear approximations and parabolas between 0 and 4. Only 4 divisions were used for demonstration purposes.

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 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.

VI. Application

The 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. This system we will assume that the sensors are linear potentiometers. The *x* and *y* position of the arm is very nonlinear. The position can be calculated by equations XX and XX. In other words we will have 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 8. Two arm planar manipulator with variables shown.Figure 8.

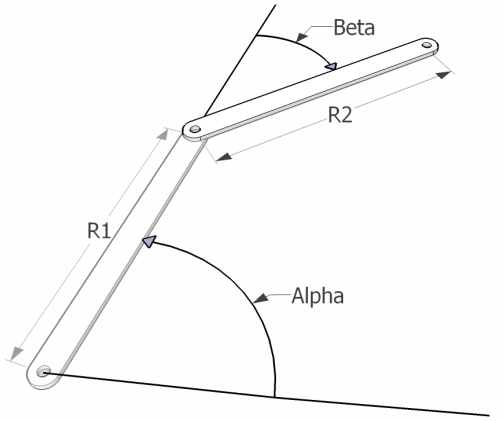

Figure 8. Two arm planar manipulator with variables shown.

The process is tested with 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 arm position and transmits this data via the serial port back to Matlab. The reason for this is to give a more accurate test of the results. In this test system the accuracy of the sensors can be avoided. Also the position of the arm would have to be measured by hand and this measurement would also introduce error into the final results. The error produced by the system is less than the predicted by many sensors and measuring techniques.

The first step of the process was to generate neural network training data. The following equation was used to calculate the *x* and *y* position based on *alpha* and *beta.*

$$x = R1 \cdot \cos(alpha) + R2 \cdot \cos(alpha + Beta)$$
$$y = R1 \cdot \sin(alpha) + R2 \cdot \sin(alpha + Beta) \quad (4)$$

The neural network was then trained using this data. The trained network was ran in Matlab to confirm that it functions correctly. 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 error. This process will introduce error in two places. First there will be the error created by using a neural network approximation opposed to the original equations. Then there is the error introduced between the ideal neural network and the network on the microcontroller. The sum of these two errors has a max value of less than two percent at any single point for the given surfaces.

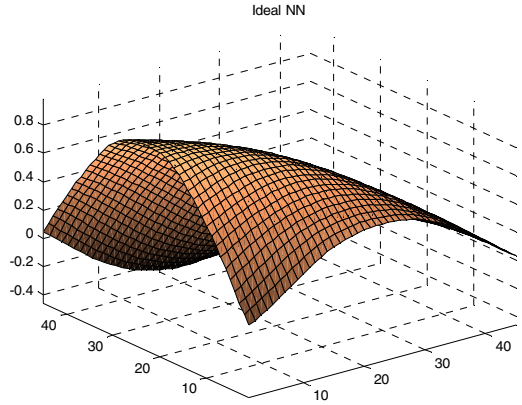

Figure 9. Output *x* of two output system generated by ideal neural network.

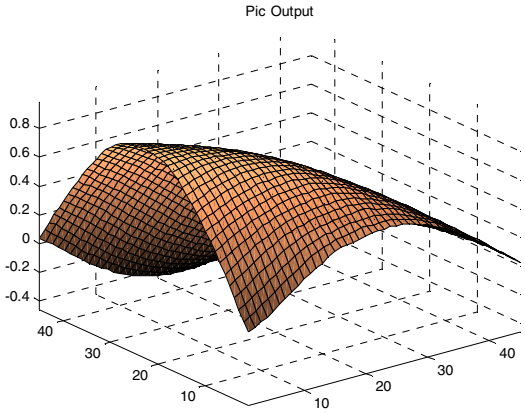

Figure 10. Output *x* of two output system generated by embedded neural network.

Figures 11 and 12 show the ideal neural network and the output of the microcontroller for the *y* component of the forward kinematics problem.

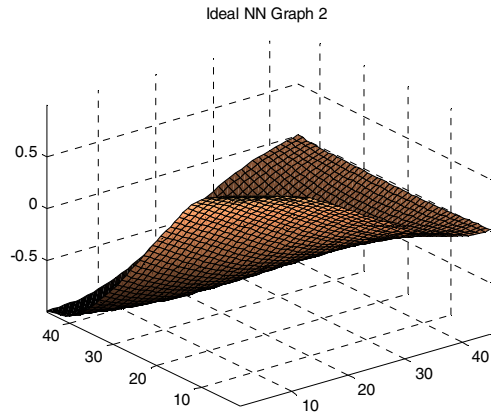

Figure 11. Output *y* of two output system generated by ideal neural network.

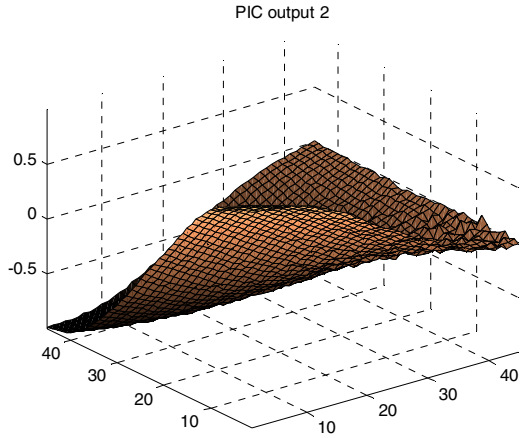

Figure 22. Error between the embedded neural network and ideal neural network surface of output *y*.

VII. Conclusion

In this paper, a novel method of linearizing sensor data for nonlinear control problems using neural networks at the embedded level. It has been shown with the correct neural network architectures even very difficult problems can be solved with a just a few neurons. When using the NBN training method these networks can be easily trained. Then by using the NBN forward calculation method networks with any architecture can be used at the embedded level. For very inexpensive and low end microcontrollers a novel floating point algorithm has been developed and optimized for neural networks. The second order approximation of *tanh* in conjunction with the pseudo floating point routines allow almost any neural network to be embedded in a simple low cost microcontroller.

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] Bhim Singh; Vishal Verma; Jitendra Solanki, "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

[5] Wilamowski, B.M.; Kaynak, O., "Oil well diagnosis by sensing terminal characteristics of the induction motor," *Industrial Electronics, IEEE Transactions on* , vol.47, no.5, pp.1100-1107, Oct 2000

[6] Abu-Rub, H.; Schmirgel, H.; Holtz, J., "Sensorless Control of Induction Motors for Maximum Steady-State Torque and Fast Dynamics at Field Weakening," *Industry Applications Conference, 2006. 41st IAS Annual Meeting. Conference Record of the 2006 IEEE* , vol.1, no., pp.96-103, 8-12 Oct. 2006

[7] Holtz, J., "Initial Rotor Polarity Detection and Sensorless Control of PM Synchronous Machines," *Industry Applications Conference, 2006. 41st IAS Annual Meeting. Conference Record of the 2006 IEEE* , vol.4, no., pp.2040-2047, 8-12 Oct. 2006

[8] Dempsey, G.L.; Alt, N.L.; Olson, B.A.; Alig, J.S., "Control sensor linearization using a microcontroller-based neural network," *Systems, Man, and Cybernetics, 1997. 'Computational Cybernetics and Simulation'., 1997 IEEE International Conference on* , vol.4, no., pp.3078-3083 vol.4, 12-15 Oct 1997

[9] Bashyal, S.; Venayagamoorthy, G.K.; Paudel, B., "Embedded neural network for fire classification using an array of gas sensors," *Sensors Applications Symposium, 2008. SAS 2008. IEEE* , vol., no., pp.146-148, 12-14 Feb. 2008

[10] Chen, S.; Wu, Y.; Alkadhimi, K., "A two-layer learning method for radial basis function networks using combined genetic and regularised OLS algorithms," *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)* , vol., no., pp.245-249, 12-14 Sep 1995

[11] Ampazis, N.; Perantonis, S.J., "Two highly efficient second-order algorithms for training feedforward networks," *Neural Networks, IEEE Transactions on* , vol.13, no.5, pp. 1064-1074, Sep 2002

[12] Jian-Xun Peng; Kang Li; Irwin, G.W., "A New Jacobian Matrix for Optimal Learning of Single-Layer Neural Networks," *Neural Networks, IEEE Transactions on* , vol.19, no.1, pp.119-129, Jan. 2008

[13] Wilamowski, B.M., "Neural network architectures and learning algorithms," *Industrial Electronics Magazine, IEEE* , vol.3, no.4, pp.56-63, Dec. 2009

[14] 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.

[15] 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.

[16] 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