

# Compensation of Nonlinearities Using Neural Networks Implemented on Inexpensive Microcontrollers

Nicholas J. Cotton, *Member, IEEE*, and Bogdan M. Wilamowski, *Fellow, IEEE*

**Abstract**—This paper describes a method of linearizing the nonlinear characteristics of many sensors and devices using an embedded neural network. The neuron-by-neuron process was developed in assembly language to allow the fastest and shortest code on the embedded system. The embedded neural network also requires an accurate approximation for hyperbolic tangent to be used as the neuron activation function. The proposed method allows for complex neural networks with very powerful architectures to be embedded on an inexpensive 8-b microcontroller. This process was then demonstrated on several examples, including a robotic arm kinematics problem.

**Index Terms**—Embedded, microcontroller, neural networks, nonlinear sensor compensation.

## I. INTRODUCTION

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 [14], [17] and power distribution problems dealing with harmonic distortion. Also, due to the nonlinear nature of neural networks, they have become an integral part of the field of controls [2]–[4]. The common myth is that neural networks require relatively significant computation power, and usually, advance computing architectures are needed. Primarily, fuzzy systems are implemented on inexpensive microcontrollers [13]–[16], [20], [21]. There has been also an attempt of implementations of neural networks on microcontrollers [5]–[7]. Due to the use of various simplification methods, such as limited bit resolution or piecewise approximations of activation functions, the obtained results are not encouraging. The exception is an implementation of neural networks on a Motorola HC11 microcontroller [8] where it is shown that neural networks can be superior to fuzzy system in almost all aspects: smaller errors, smoother surfaces,

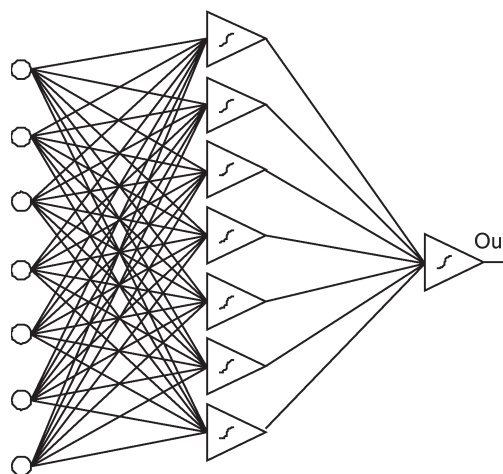


Fig. 1. Eight neurons in MLP architecture for solving parity-7.

shorter code, and faster operation. This was possible by the replacement of the traditional tangent hyperbolic function

$$\tanh(net) = \frac{2}{1 + \exp(-2net)} - 1 \quad (1)$$

with Elliott function

$$\tanh(net) = \frac{net}{1 + |net|} \quad (2)$$

Unfortunately, most simple microcontrollers lack a hardware divider, giving no advantage to using the Elliott function. Also, in most cases, nonlinear approximations of neural networks using the Elliott function (2) require 30%–40% more neurons than in the cases when tangent hyperbolic function (1) is used. In this paper, it is shown that positive results can be obtained if the *tanh* activation function is approximated by piecewise parabolic approximation.

With limited hardware resources, it is important to use efficient neural network architectures [9], [10]. Fig. 1 shows the popular multilayer perceptron (MLP) architecture with one hidden layer, which is limited to solving problems smaller than parity-7, while Fig. 2 shows an example of a fully connected cascade (FCC) architecture which also uses eight neurons and is capable of solving problems as large as parity-255. There are, of course, other possible architectures which are more powerful than the popular MLP architecture [18]. The question is: Why do most researches use the MLP architectures? The simple answer to this question is that there is no software to efficiently train neural network architectures other than MLP. For example,

Manuscript received July 23, 2010; revised October 15, 2010; accepted November 15, 2010. Date of publication December 10, 2010; date of current version February 11, 2011.

N. J. Cotton was with the Department of Electrical and Computer Engineering, Auburn University, Auburn, AL 36849 USA. He is now with the Naval Sea Systems Command, Panama, FL 32407-7001 USA (e-mail: cottonj@ieee.org).

B. M. Wilamowski is with the Alabama Micro/Nano Science and Technology Center, Auburn University, Auburn, AL 36849 USA (e-mail: wilam@ieee.org).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TIE.2010.2098377

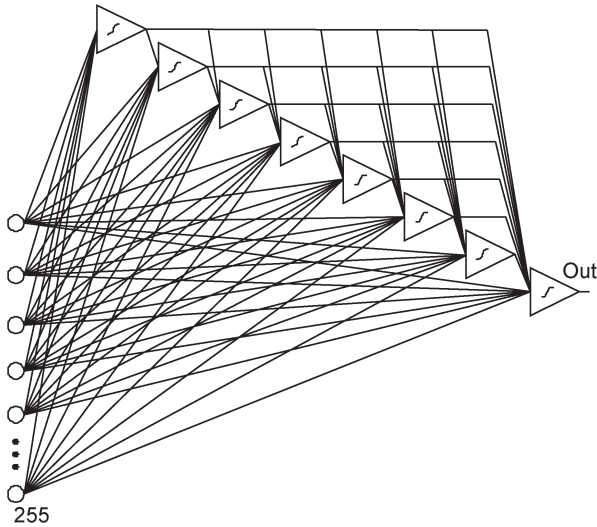


Fig. 2. Eight neurons using FCC architecture capable of solving parity-255.

the most efficient second-order Levenberg–Marquardt algorithm was implemented in popular MATLAB Neural Network Toolbox only for MLP topologies. This situation has been changed recently when second-order algorithms were developed for arbitrarily connected neural networks [11], [12].

The approach presented in this paper is taking advantage of the following:

- 1) usage of more efficient architectures than popular MLP architecture;
- 2) implementation of neuron-by-neuron (NBN) computation algorithm;
- 3) efficient quadratic piecewise approximation of activation function;
- 4) pseudo floating-point arithmetic taking advantage of specific computation process.

In the floating-point computation, each number has its own mantissa and exponent. It turns out that it is possible to simplify the computation process by using the same exponent (read the same scaling factors) for all weights of each neuron. These methods allow for more efficient use of the embedded processor and memory.

Two approaches of microcontroller programming were investigated. The most robust method was to use C with IEEE 754 floating-point arithmetic. This method, however, was very inefficient and led to slow operation. In the second approach, the microcontroller programming was done directly in an assembly language giving more flexibility to the programmer. The code was up to ten times faster. The neural network was embedded in a microchip brand microcontroller PIC18F45J10. 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. In order to shorten the code and to increase the capacity of microcontroller, several special approaches in assembly programming were used. These approaches include the following: efficient neural network architectures and NBN computations discussed in Section II, pseudo floating arithmetic as described in Section III, 16-b multiplications using 8-b hardware multiplier is described in Section IV, and approximate calculation of activation function is described

in Section V. These same techniques could also be translated to the C domain without the overhead of traditional floating point.

## II. TAKING ADVANTAGE OF EFFICIENT NEURAL NETWORK ARCHITECTURES

As mentioned in the introductory section, the power of neural networks strongly depends on the neural network architectures [9]. Popular MLP architectures are not only one of the least powerful architectures but also have other disadvantages. With an increased number of layers, the training of such networks becomes more difficult because the networks become less transparent for error propagation. In other words, the effect of weight changes on the first layers are disturbed by weight changes of the subsequent layers. The second disadvantage of multilayer networks without connections across layers is that inaccuracies in the first layers are magnified by inaccuracies of subsequent layers. The most powerful architecture is the FCC architecture shown in Fig. 2. Due to cross-layer connections, the network is much more transparent for training and is also less sensitive to error accumulations from layer to layer. Even for FCC neural networks with connections across layers, their architectures should have as small number of layers as possible. For example, instead of the FCC shown in Fig. 2 with seven hidden layers, where there are only single neurons in each layer, it would be more desirable to introduce several neurons in each layer and to reduce the number of hidden layers. Such architectures, known as bridged multilayer perceptron (BMLP), are a good compromise between MLP and FCC architectures. For example, the BMLP architecture  $N = 4 = 4 = 1$  with only three hidden layers and four neurons in each layer is capable to solve problems up to parity-249 [9], [10], [18]. In order to be able to implement the neural networks with arbitrary connected architecture, a special method of computation scheme had to be developed. The computation process on microcontroller follows the neuron-by-neuron algorithm [11], [12]. This method requires special modifications due to the fact that assembly language is used with very limited memory resources.

Unfortunately, most of available learning software is not able to train FCC or BMLP architectures. Only very recently, adequate and very efficient algorithms for arbitrarily connected feedforward neural networks have been developed [9], [12], [22]. In order to implement more powerful neural network architectures on microcontrollers, the special NBN computing routine should be used. The NBN routine was described in detail in [11], but its efficient implementation on microcontroller with simplified arithmetic was another challenge.

This method requires special modifications due to the fact that assembly language is used with very limited memory resources.

At first, neurons must be numbered starting from inputs to outputs. Then, the information about neural network topology is stored in subsequent memory locations using the following format:

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

where  $K$  is the number of weights in the network. Notice that the order that the weights are listed follows the network architecture described in the first array.

```
//Architecture (8-bit)
Number of Neurons;
//Neuron 1
Neuron Scale, Number of Inputs, Output Node, Inputs(1 –
M1)
//Neuron 2
Neuron Scale, Number of Inputs, Output Node, Inputs(1 –
M2)
...
//Neuron N
Neuron Scale, Number of Inputs, Output Node, Inputs(1 –
MN)
```

where  $N$  is the number of neurons and  $M_1, M_2, \dots, M_N$  are the number of input weights of each neuron. All weights are stored in linearity addressed memory using the described format.

The process is written so that each neuron is calculated individually in a series of nested loops: main loop, network loop, and neuron loop [15].

### III. PSEUDO FLOATING-POINT ARITHMETIC

The nonconventional part of this floating-point routine is the way the exponent and mantissa are stored. Essentially, all 16-b are the mantissa, and the exponent for the entire neuron is stored in a single location. Such an approach is possible because the same exponent can be used for all weights of every neuron. 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. This scaling is done offline before programming in order to save valuable processing time on each forward calculation.

Scaling does two things: First, it prevents overflow by keeping the numbers within operating regions, and second, it 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 have 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. In the proposed approach on the 8-b microcontroller, the  $net$  values for each neuron are calculated with 32-b accuracy. In order to increase computation, accuracy weights are being scaled separately as shown in Fig. 3. Before the activation function is applied to both scales, input scale and weight scales are summed. This raised to the  $N$  power is always the same as shift by  $N$  because of the way the scale factors are calculated. This makes the

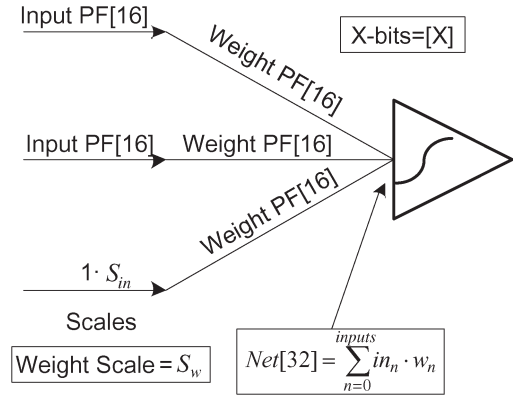


Fig. 3. PF stands for pseudo floating-point number. The numbers in brackets refer to the number of bits that represent that particular value.

scaling process very fast opposed to having to 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.

### IV. MULTIPLICATION

The PIC18F45J10 microcontroller has an 8 b  $\times$  8 b unsigned hardware multiplier. Because 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 has passed two 16-b numbers, consisting of an 8-b integer and an 8-b fraction portion. The routine returns a 32-b product. The result of the multiplication routine is a 32-b fixed-point result shown in (3)

$$\frac{A \cdot C \cdot 256^2 + 256 \cdot (A \cdot D + B \cdot C) + B \cdot D}{256^2}. \quad (3)$$

The hardware multiplier does the multiplication between bytes in a single instruction. 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-b result as an accumulator for all inputs of the neuron.

The inputs are multiplied by the corresponding weights, and the result is stored in the 32-b  $net$  register. This is essentially a multiply and accumulate register designed for this particular stage. It is very important to keep all 32-b in this stage for adding and subtracting. Without the 32-b 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 [19].

### V. ACTIVATION FUNCTION

Several approaches were initially considered, such as lookup tables or linear piecewise approximations, but with these approaches, it was very difficult to obtain proper accuracy. As an alternative, an expansion of  $\tanh$  function into series was also considered, but with this approach, an acceptable accuracy is only possible for very small and very large  $net$  values. The accuracy in the middle range of  $net$  values was unsatisfactory. Finally, the second-order piecewise quadratic approximation was selected as shown in Fig. 4.



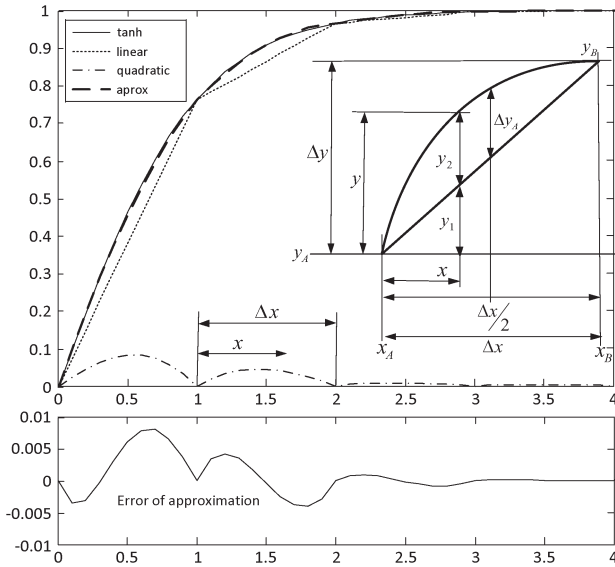


Fig. 4. (Solid line) Approximation of tangent hyperbolic function: (Dotted line) Linear, (dashed line) quadratic, and (dash dotted line) quadratic component only. Lower graph shows approximation error with excessively large segments because only four divisions were used for demonstration purposes. Inside the upper figure, there is a construction illustrating the symbols in (4) and (5).

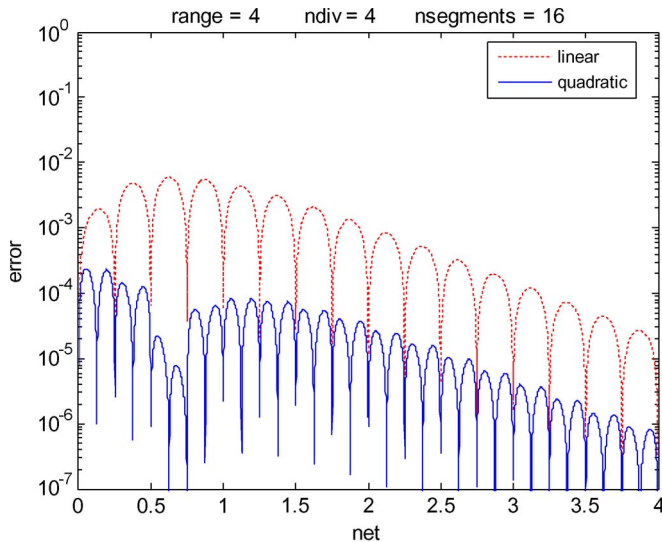


Fig. 5. Error comparison using 16 segments for (dotted lines) piecewise linear and for (dotted lines) piecewise quadratic approximations.

In order to obtain an acceptable accuracy,  $\Delta x = 0.25$  was used, and this implies that, in the range from 0 to 4, there are 16 segments, and four most significant bits of the *net* value identify the segment. Error comparison is shown in Fig. 5. Using piecewise linear approximation, the maximum error is below 1%. As one may see from Fig. 5, with piecewise quadratic approximation with 16 segments, the maximum error was reduced to below 0.03%. If 32 segments are used, then the maximum error is reduced to 0.003%, as it is shown in Fig. 6. One may notice that, with 32 segments, when only piecewise linear approximation is used, the maximum error is about 0.2%, which can be acceptable for many applications.

In order to utilize 8-b hardware multiplication, the  $\Delta x$  was represented by 7 b (having an integer range from 0 to 127). This

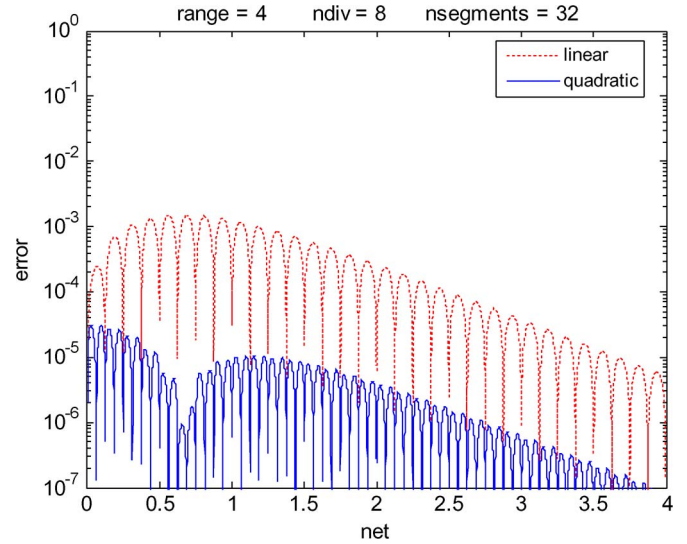


Fig. 6. Error comparison using 32 segments for (dotted lines) piecewise linear and for (dotted lines) piecewise quadratic approximations.

way, the division operation in both equations can be replaced by the right shift operation. Calculation of  $y_1$  requires one subtraction, one 8-b multiplication, one shift right by 7 b, and one addition. Calculation of  $y_2$  requires one 8-b subtraction, two 8-b multiplications, and shift right by 14 b. More information on these calculations can be seen in [19].

The approximate value of  $\tanh(\text{net})$  is found in several steps.

- 1) Using the higher bits of the *net* value, the proper piecewise segment is selected between  $x_A$  and  $x_B$ . Then, the corresponding values of  $y_A$  and  $y_B$  are found from memory. In our implementation, four most significant bits were used to retrieve data for 16 segments.
- 2) The  $\Delta x$  value is obtained from lower bits of *net* value.
- 3) Using  $x_A$ ,  $x_B$ ,  $y_A$ , and  $y_B$ , the first-order linear approximation is computed at first

$$y_1(x) = y_A + \frac{(y_B - y_A) \cdot x}{\Delta x}. \quad (4)$$

- 4) Then, the quadratic component is found as

$$y_2(x) = \frac{4 \cdot \Delta y_A \cdot x \cdot (\Delta x - x)}{\Delta x^2} \quad (5)$$

where  $\Delta y_A$  values are read from memory. Divisions in (4) and (5) can be easily replaced by bit shift operations.

## VI. APPLICATIONS

In order to demonstrate that the microcontroller neural network is performing correctly, several examples of control problems were tested. Neural networks have the unique ability to solve multidimensional 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 inputs.

The process is tested with the microcontroller hardware in the loop. The sensor data is transmitted via the serial port from

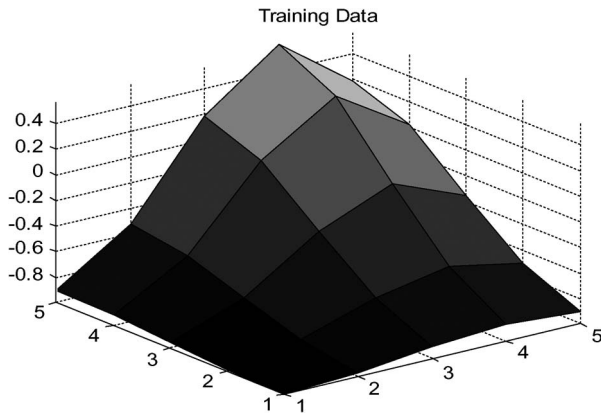


Fig. 7. Control surface of TSK fuzzy controller.

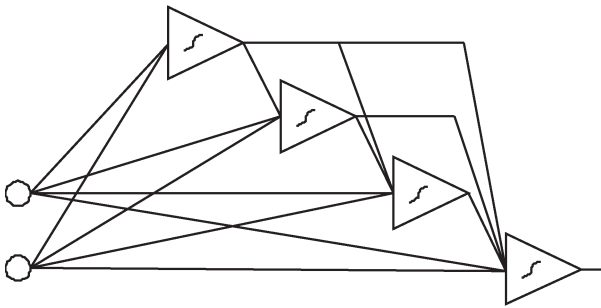


Fig. 8. Four neuron cascade architecture for the surface shown in Fig. 7.

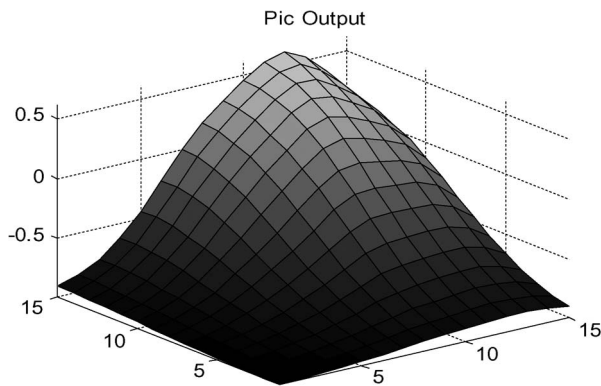


Fig. 9. PIC output with 196 test patterns for the trained neural network in Fig. 8.

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. This way, any inaccuracy of the sensors can be avoided.

#### A. Implementation of TSK Fuzzy System on Neural Networks

The control surface for the Takagi–Sugeno–Kang (TSK) fuzzy system using triangular membership functions is shown in Fig. 7. Twenty-five fuzzy rules were used as the training set. This problem can be solved effectively using a network with four neurons, as shown in Fig. 8.

Fig. 9 shows the control surface obtained with neural network implemented on PIC microcontroller. One may notice the ability of neural network to approximate points which were not

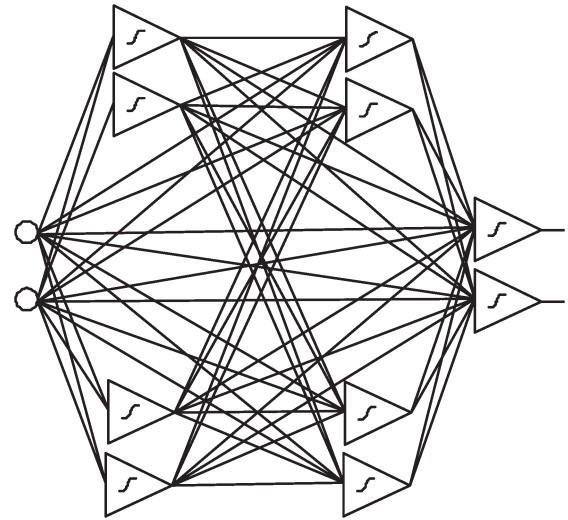


Fig. 10. Ten-neuron network for solving forward kinematics problem with two degrees of freedom.

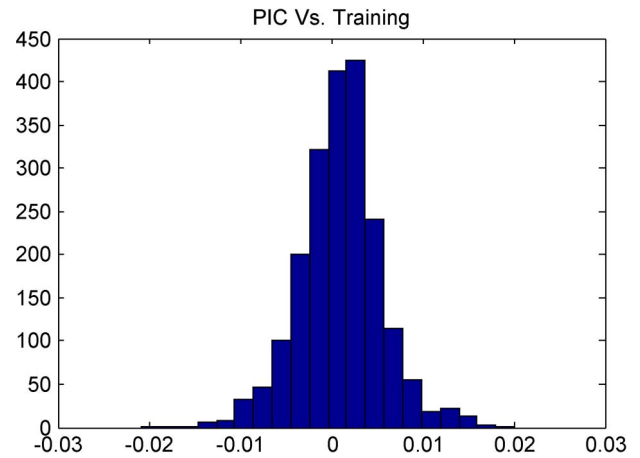


Fig. 11. Error histogram for the network in Fig. 10 implemented on PIC microcontroller.

used for training. It is obvious that the neural network produces much smoother and more reasonable nonlinear approximation than fuzzy system.

#### B. Two-Arm Planar Manipulator

The two-link planar manipulator was used as a practical application for this embedded neural network. In this example, the embedded neural network calculates the  $x$  and  $y$  positions of the arm based on the data read from sensors at the joints. This problem is known as forward kinematics. The solution to this problem was already presented in [23]. What is unique about this approach is that, in order to map for both forward and reverse kinematics problem, we were able to implement on PIC neural networks with multiple outputs. In the case of a 2-D problem, we have used the architecture shown in Fig. 10.

Error histogram for the network in Fig. 10 implemented on PIC microcontroller is shown in Fig. 11. Most of the errors are below 1%. This error is a superposition of an error created during the training process and from an error created by limited accuracies of computation on PIC microcontroller. In this particular case, the influences of both errors are in the same range.

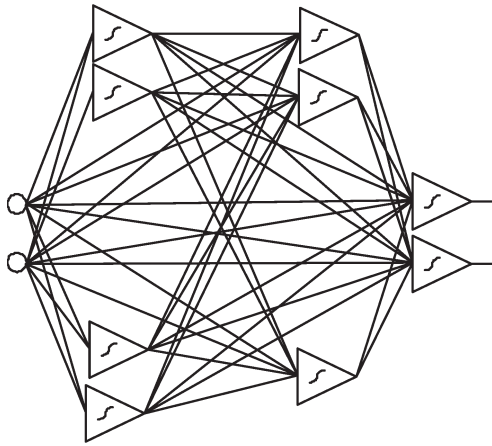


Fig. 12. Eight-neuron network used for solving the Matlab peaks surface.

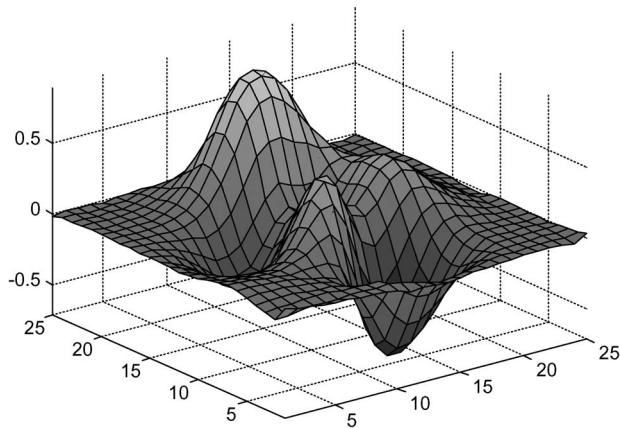


Fig. 13. PIC output for the peaks surface using assembly language programming.

### C. MATLAB Peaks Surface

Programming microcontrollers in assembly is time consuming and has limited portability from one microcontroller to another one. It is much more convenient to use the C language because the code can be transposed between different types of microcontrollers. However, direct implementation of C language led to much longer program memory and significantly slower operation because true floating-point operations are used, and tangent hyperbolic is computed using sophisticated and slow routine which gives much higher accuracy than needed. In order to efficiently use the C language, an advantage of the proposed solutions can be taken such as the following: NBN computation process, pseudo floating-point concept, and quadratic piecewise approximation of activation function. With this approach, it was possible to obtain computation speed comparable to the assembly programming.

Let us use the popular Matlab function peaks in this experiment. Instead of 11 neurons as was used in [23], it was possible to solve this problem with eight neurons using BMLP architecture, which has two hidden layers with all neurons connected directly to the inputs and all preceding layers as shown in Fig. 12.

Fig. 13 shows the surface obtained using assembly language programming, and Fig. 14 shows the surface obtained using the C-programming.

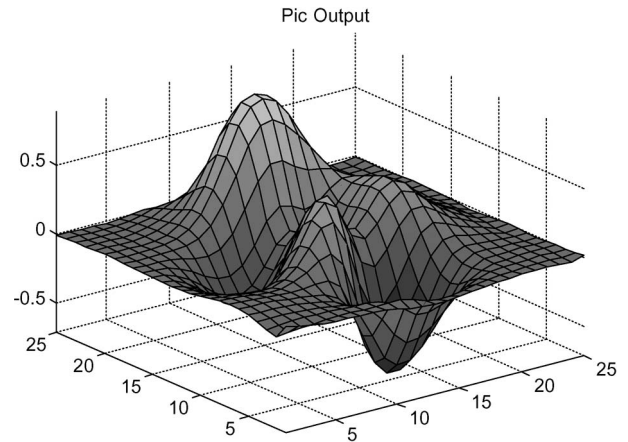


Fig. 14. Output of the microcontroller using the C version of the embedded neural network software.

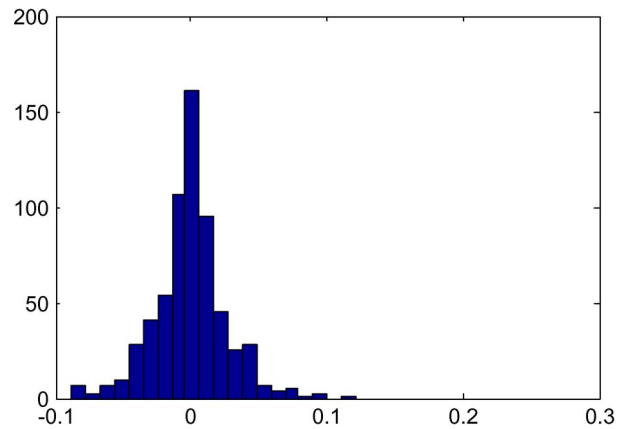


Fig. 15. Histogram of errors between the PIC output and the training data. The X-axis is the error, and the Y-axis is the number of samples.

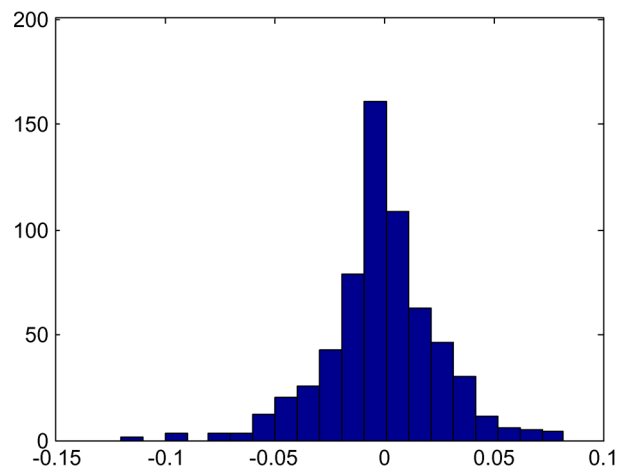


Fig. 16. Histogram of errors between ideal neural network and training data. The X-axis is the error, and the Y-axis is the number of samples.

The difference between the ideal network and the 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. For each point, an error value was calculated as the difference between the training data and the microcontroller output. These errors are shown in the form of histograms in Figs. 15 and 16. Notice that the

TABLE I  
NEURAL NETWORK PERFORMANCE COMPARISON

Surface	Neurons	NN Error	Ideal & PIC RMS	Ideal & Training RMS	Training & Pic RMS	Time ms
Fuzzy	4	0.001	0.005151	0.00446	0.0068827	0.317
Fuzzy	2	0.001	0.008093	0.07516	0.076336	0.163
Fuzzy C	4	0.001	0.000041	0.00446	0.0044666	2.2
Peaks	19	0.01	0.015005	0.01059	0.018257	1.6
Peaks	8	0.01	0.007292	0.0253	0.026012	0.752
Peaks C	19	0.01	0.000127	0.01059	0.0089523	5.87
Peaks C	8	0.01	0.000259	0.0253	0.025301	2.95

assembly version has a few more outliers than the C version, but overall, there errors are comparable and similar in distribution. The C version more accurately represents the ideal neural network; however, the ideal network does not perfectly represent the training data which is the root cause of the errors in both cases. In other words, the system accuracy increase by using a significantly more accurate neural network representation is marginal.

#### D. Experimental Data Summary

After comparing the results of the two different implementations of neural networks, it was obvious that the C version can be very accurate. The C version was significantly slower due to its complexity of calculations created by using IEEE 754 floating-point arithmetic. Due to the increase in variable size, it becomes necessary to store all weights and nodes in program memory rather than RAM. The accuracy and speeds can be seen in Table I.

The C version took approximately ten times longer than the assembly version, which was predicted based on the 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 accurate which makes it very valuable even on such a low-end microcontroller. However, when the rms errors from the final output are compared to the training data, both the C and assembly have very similar errors. The reason for this is that the majority of the errors is introduced into the system by the neural network itself not by the digital implementation of the network.

## VII. CONCLUSION

This paper has presented a method for implementing an arbitrary neural network on an embedded system. This process balances speed with accuracy for systems that do not have floating-point hardware. The operational goal was to create a neural network that was as fast as possible while introducing

an error that is not greater than the error already introduced by using a neural network to approximate the original function. When these very similar errors are combined, they fail to produce a final product that is closer to the original function than that of the ideal double precision neural network. This goal was met and has been a successful compromise between speed and accuracy for many applications.

This particular application method has been implemented using assembly language on an 8-b microcontroller. This method, however, can be easily implemented using C and integer math in the same manner. The key components are the implementation of the superior types of neural network architectures [9], [10], the concept of using an exponent for each neuron, a custom multiply and accumulate method, and a better activation function for embedded neural networks. The combination of these four components creates a functionally practical and efficient embedded neural network solution for a large variety of applications. This implementation, being hardware independent, allows the user to customize the requirements for his application. For example, the saturation points in the activation function could be extended to five from four to produce greater accuracy. These parameters are completely adjustable for the user's application. These concepts have been presented to be implemented on any embedded system without floating-point hardware to optimize the balance of speed and accuracy.

The example showing a method of linearizing nonlinear sensor data for nonlinear control problems using neural networks at the embedded level has been introduced. It has been shown that, with the correct neural network architectures, even very difficult problems can be solved with 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. The second-order approximation of  $\tanh$  in conjunction with the pseudo floating-point routines allows almost any neural network to be embedded in a simple low-cost microcontroller. For very inexpensive and low-end microcontrollers, a floating-point algorithm has been developed and optimized for neural networks. This proof of concept on this simple inexpensive microcontroller can be expanded on any other embedded system.



## REFERENCES

- [1] B. K. Bose, "Neural network applications in power electronics and motor drives—An introduction and perspective," *IEEE Trans. Ind. Electron.*, vol. 54, no. 1, pp. 14–33, Feb. 2007.
- [2] S. S. Ge and C. Wang, "Adaptive neural control of uncertain MIMO nonlinear systems," *IEEE Trans. Neural Netw.*, vol. 15, no. 3, pp. 674–692, May 2004.
- [3] E. B. Kosmatopoulos, M. M. Polycarpou, M. A. Christodoulou, and P. A. Ioannou, "High-order neural network structures for identification of dynamical systems," *IEEE Trans. Neural Netw.*, vol. 6, no. 2, pp. 422–431, Mar. 1995.
- [4] F. L. Lewis, A. Yegildirek, and L. Kai, "Multilayer neural-net robot controller with guaranteed tracking performance," *IEEE Trans. Neural Netw.*, vol. 7, no. 2, pp. 388–399, Mar. 1996.
- [5] S. Bashyal, G. K. Venayagamoorthy, and B. Paudel, "Embedded neural network for fire classification using an array of gas sensors," in *Proc. IEEE SAS*, 2008, pp. 146–148.
- [6] 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. Syst., Man, Cybern. 'Computational Cybernetics and Simulation' Conf.*, 1997, pp. 3078–3083.
- [7] 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 ICCAE*, 2010, pp. 592–597.
- [8] B. M. Wilamowski and J. Binfet, "Do fuzzy controllers have advantages over neural controllers in microprocessor implementation," presented at the 2nd Int. Conf. Recent Advances Mechatronics, Istanbul, Turkey, 1999.
- [9] B. M. Wilamowski, "Neural network architectures and learning algorithms," *IEEE Ind. Electron. Mag.*, vol. 3, no. 4, pp. 56–63, Dec. 2009.
- [10] B. M. Wilamowski, D. Hunter, and A. Malinowski, "Solving parity-N problems with feedforward neural networks," in *Proc. Int. Neural Netw. Joint Conf.*, 2003, pp. 2546–2551.
- [11] B. M. Wilamowski, N. J. Cotton, O. Kaynak, and G. Dundar, "Computing gradient vector and Jacobian matrix in arbitrarily connected neural networks," *IEEE Trans. Ind. Electron.*, vol. 55, no. 10, pp. 3784–3790, Oct. 2008.
- [12] B. M. Wilamowski and H. Yu, "Improved computation for Levenberg-Marquardt training," *IEEE Trans. Neural Netw.*, vol. 21, no. 6, pp. 930–937, Jun. 2010.
- [13] N. J. Cotton and B. M. Wilamowski, "Compensation of sensors non-linearity with neural networks," in *Proc. 24th IEEE Int. Conf. AINA*, Apr. 20–23, 2010, pp. 1210–1217.
- [14] F. Betin, D. Pinchon, and G.-A. Capolino, "Fuzzy logic applied to speed control of a stepping motor drive," *IEEE Trans. Ind. Electron.*, vol. 47, no. 3, pp. 610–622, Jun. 2000.
- [15] W. X. Shen, C. C. Chan, E. W. C. Lo, and K. T. Chau, "Adaptive neuro-fuzzy modeling of battery residual capacity for electric vehicles," *IEEE Trans. Ind. Electron.*, vol. 49, no. 3, pp. 677–684, Jun. 2002.
- [16] C. Cecati, A. Dell'Aquila, A. Lecci, and M. Liserre, "Implementation issues of a fuzzy-logic-based three-phase active rectifier employing only voltage sensors," *IEEE Trans. Ind. Electron.*, vol. 52, no. 2, pp. 378–385, Apr. 2005.
- [17] F. Betin, A. Sivert, A. Yazidi, and G.-A. Capolino, "Determination of scaling factors for fuzzy logic control using the sliding-mode approach: Application to control of a DC machine drive," *IEEE Trans. Ind. Electron.*, vol. 54, no. 1, pp. 296–309, Feb. 2007.
- [18] B. M. Wilamowski, "Challenges in applications of computational intelligence in industrial electronics," in *Proc. IEEE ISIE*, Bari, Italy, Jul. 5–7, 2010, pp. 15–22.
- [19] N. J. Cotton, "A neural network implementation on embedded systems," Ph.D. dissertation, Dept. Elect. Eng., Auburn Univ., Auburn, AL, 2010.
- [20] N. J. Medrano-Marques and B. Martin-del-Brio, "Sensor linearization with neural networks," *IEEE Trans. Ind. Electron.*, vol. 48, no. 6, pp. 1288–1290, Dec. 2001.
- [21] C. A. Hudson, N. S. Lobo, and R. Krishnan, "Sensorless control of single switch-based switched reluctance motor drive using neural network," *IEEE Trans. Ind. Electron.*, vol. 55, no. 1, pp. 321–329, Jan. 2008.
- [22] B. M. Wilamowski and H. Yu, "Neural network learning without back-propagation," *IEEE Trans. Neural Netw.*, vol. 21, no. 11, pp. 1793–1803, Nov. 2010.
- [23] N. J. Cotton, B. M. Wilamowski, and G. Dundar, "A neural network implementation on an inexpensive eight bit microcontroller," in *Proc. 12th Int. Conf. INES*, Miami, FL, Feb. 25–29, 2008, pp. 109–114.



**Nicholas J. Cotton** (S'07–M'10) received the M.S. and Ph.D. degrees in electrical engineering from Auburn University, Auburn, AL, in 2008 and 2010, respectively.

He is currently a Research Scientist with the Naval Sea Systems Command for the Naval Surface Warfare Center Panama City Division, Panama City, FL. He is also with Dynetics, Inc., Huntsville, AL, as an Electrical Engineer. He has taught undergraduate courses and was a Research Assistant with the Department of Electrical and Computer Engineering,

Auburn University. His main interests are computational intelligence, neural networks, embedded systems, and cooperative robotics.

Dr. Cotton is also a Reviewer for the IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS.



**Bogdan M. Wilamowski** (SM'83–F'00) received the M.S. degree in computer engineering in 1966, the Ph.D. degree in neural computing in 1970, and the Dr. habil. degree in integrated circuit design in 1977.

He was with the Gdansk University of Technology, Gdansk, Poland; University of Information Technology and Management, Rzeszow, Russia; Auburn University, Auburn, AL; University of Arizona, Tucson; University of Wyoming, Laramie; and the University of Idaho, Moscow. He is currently

the Director of the Alabama Micro/Nano Science and Technology Center, Auburn University.

Dr. Wilamowski was the Vice President of the IEEE Computational Intelligence Society (2000–2004) and the President of the IEEE Industrial Electronics Society (2004–2005). He served as an Associate Editor in numerous journals. He was the Editor in Chief of IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS from 2007 to 2010, and currently, he serves as the Editor in Chief of the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS.