Distributed Neural Network: Dynamic Learning via Backpropagation with Hardware Neurons using Arduino Chips

Gary Parker and Mohammad Khan
Department of Computer Science
Connecticut College
New London, CT, USA
parker@conncoll.edu, mkhan4@conncoll.edu

Abstract—**In this paper we present an implementation of and a proposed algorithm for an easily expandable hardware Artificial Neural Network (ANN) capable of learning using inexpensive, off-the-shelf microprocessors. While significant work has been done in hardware ANN implementations, this research offers a unique, general use, unspecialized, and inexpensive model with a flexible architecture representation. Using Arduino Pro Mini microprocessors and a flexible data communication framework that makes use of the built-in circuit bus called the Inter-Integrated Circuit, this implementation involves the programming of one neuron per microchip. This one to one ratio allows for the computational parallelism inherent in neural networks and provides for the flexibility of building various ANN architectures. The prototype that was developed consists of an input layer element microchip, two hidden layer neuron microchips, and an output layer neuron microchip. Learning happens completely on hardware via backpropagation without a data connection to a computer. Tests showed that the prototype can learn the logical operations OR, AND, XOR, and XNOR, and that the system can accommodate dynamic changes in learning between logical operations.**

Keywords—Artificial Neural Networks; backpropagation; dynamic learning; hardware; Arduino; Inter-Integrated Circuit;

I. INTRODUCTION

An Artificial Neural Network (ANN) models a biological neural network, which can have billions of neurons with trillions of interconnections. Most ANNs are implemented solely in software simulations, and those implemented in hardware usually have the whole ANN loaded on one chip. This research takes a step towards a new ANN implemented by connecting a number of microchips such that each microchip represents a single neuron. While the commercial market does have a host of hardware ANNs, no general use, unspecialized, and inexpensive model exists such that it has a clear one chip to one neuron representation. Generally, extra modules and processing parts are added to have the chip function as a single neuron and more commonly multiple neurons are implemented on one chip. Under-the-hood, these representations also generally require a good grasp of electrical engineering to fully understand how the inputs and outputs are mapped as they use voltage, current, and similar elements to model data. In order to manipulate these elements, specialized and expensive parts are often needed. Moreover, many of the architectures are built for specific tasks, and they are not easily incorporated into bigger systems, and even fewer have learning capabilities. This paper presents a more accessible approach for creating general hardware ANNs capable of learning.

In this research we develop a hardware ANN utilizing Arduino Pro Mini (APM) microchips, which are popular, low-priced microcontrollers, coupled with open-source software, to function as neurons that are able to learn using the backpropagation algorithm. In this implementation, each of the APM chips act as a single neuron with learning capabilities, which is a step towards a more genuine imitation of a biological neural network. The architecture of the ANN that we developed for this initial trial is such that one APM chip handles the input, two APM chips are each neurons in the hidden layer taking input from the previous APM chip, and one APM chip is a neuron in the output layer. The communication between the chips is supported by the I2C (Inter-Integrated Circuit) capability of Arduino which allows a Master-Slave connection between the output chip, the hidden layer chips, and the input chip. This works into the theme of a more genuine model because the output neuron fires only when it receives input from the hidden layer neurons. This design was implemented on a breadboard to show the dynamic nature of the system whereby chips/neurons can be moved to grow or shrink the ANN. The finished network has the capability of learning four different logic operations: AND, OR, XOR, and XNOR. Learning happens on the hardware APMs themselves without a data connection to a computer. The learning can be controlled by two physical switches, which allow the user to change in real time from one input/output combination correlating to one of the logic operations to another.

The key to the implementation was an understanding of the intricacies of transferring data across I2C and modularizing the overall ANN across multiple Arduinos. A protocol for transferring the information from chip to chip was developed to allow backpropagation in the learning phase.

II. RELATED WORKS

Most commonly ANNs are implemented as software and trained on computers. This technique is inherently non-parallel due to the sequential von-Neumann architecture of computers. In contrast, the past two decades have shown strides being made in the development of dedicated hardware to develop faster and more genuine networks that are processed in parallel [1]. However, there is much room for improvement. Dias, Antunes, and Mota note in a review of commercial ANN hardware that much of this hardware is specialized and expensive in development time and resources, and not much is known about its commercial implementations [2]. Goser notes in a paper on limitations of hardware ANN that one problem in developing hardware models of ANNs is dedicated, complex wiring due to specialized hardware [3]. It is important to consider how the architecture of an ANN is represented in hardware. Generally, multiple neurons are implemented on one chip and often the full ANN is put on one chip. Liao notes in a survey of Neural Networks hardware that almost all hardware neuron designs are such that they include an activation block, weights block, transfer function block, and other processing elements [4]. The activation block is always on the chip, but the other blocks need not be. In other words, much of these implementations do not focus on building a hardware neuron in a single chip. There have also been highly specialized products that explore learning through implementation of the backpropagation algorithm on several different hardware structures [5]. These products, however, deal more with hardware manipulation in parallel computers than representation on the microcontroller level.

One method for creating hardware ANNs is the use of Field Programmable Gate Arrays (FPGAs). FPGAs are integrated circuits that are physically configurable and serve as programmable logic. Significant work has been done in using FPGAs for ANN development [6, 8, 9]. Sahin, Becerikli and Yazici implemented a multiple layer ANN such that an FPGA was configured to link neurons together with multipliers and adders were representative of connections between neurons [7]. This framework allowed for manipulation of the ANN architecture by having a dynamic number of layers and neurons. The weights were learned beforehand and mapped onto the FPGA. Although no clear learning method was explored on the hardware itself, this work took an accessible product commonly used for prototyping to develop a hardware ANN. It is important to note that even though the ANN architecture can be changed in these frameworks, it does require physical configuration and rededication of parts within the hardware itself.

A general theme with development of hardware ANNs has been an architectural representation such that multiple neurons exist on one chip. Ienne and Kuhn give an analysis of multiple commercial chips [10]. For example, the Philips' Lneuro-1 consists of 16 neurons while the Philips' Lneuro-2.3 consists of 12 neurons. The Ricoh RN-200 also consists of a 16 neurons in a multiple layered ANN and includes backpropagation learning. Some implementations add extra processing parts to a chip to make it a faster system. For example, Ienne and Kuhn also describe the Intel's Ni1000 which consists of three parts where the microcontroller is combined with a classifier and an external interface that does conversion calculation to decrease the load on the chip itself.

Other specialized designs have been considered, and many intricate methods for modeling neurons have been explored. Joubert, Belhadj, Temam, and Héliot give an analysis of the Leaky Integrate-and-Fire neuron which although is a single neuron representation, has a specific architecture split among three main parts: the synapse, the neuron core, and the comparator [11]. A number of these implementations are based on electrical properties such as current, capacitance, voltage, and similar elements. Although this may be a more genuine representation of a biological neuron it does take a good amount of electrical engineering expertise to implement and reproduce, and the parts are specialized and expensive.

In summary, there has been much research in the development of novel hardware ANNs that are efficient and fast. However, most of these hardware implementations are built with specialized parts, the resources are expensive, and there does not exist a clear general, unspecialized, and inexpensive "one neuron to one chip" implementation. Hence, the ultimate goal of the research reported in this paper is to provide an alternative, to develop a hardware ANN system where each neuron is implemented on a single inexpensive, readily available chip. In addition, each neuron/chip will have learning capability (initially through backpropagation) and the implementation will be accessible to typical users. In this research we used the APM microchip because it is inexpensive, widely available, small enough for quick prototyping of ANN architectures down to the level of the neuron, and allows for a communication framework for learning through backpropagation.

III. ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks (ANNs) consist of single neurons that are linked together by weighted connections via which information is transferred. Every neuron receives a set of inputs with corresponding weights and produces only a single output. The output of one neuron becomes the input for neurons in a subsequent layer. For example, Figure 1 includes an input layer (consists of no neurons - it just provides input), a hidden layer with two neurons, and an output layer with one neuron. A threshold value (θ) symbolically represents the activation potential of a biological neuron. Normally a weighted sum of all the inputs is compared to this value to determine an output. We treat the threshold as a constant input value of -1 and give it a corresponding weight. The weighted sum (X) of all the inputs is fed into an activation function to determine an output (Y). The production of an output is called activation. Here, we employ the sigmoid activation function as such:

$$Y = \frac{1}{1 + e^{-X}} \quad (1)$$

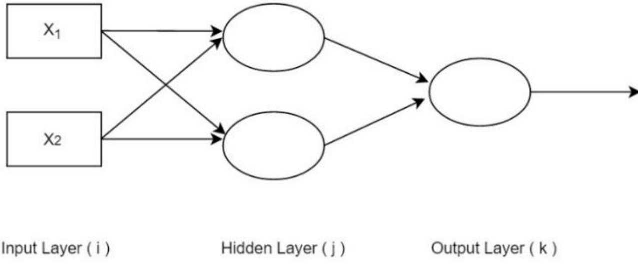

Figure 1. Visual representation of a multiple layered ANN with two inputs, two hidden layer neurons, and an output layer neuron.

The standard learning mechanism for feed-forward ANNs consists of two stages, forward propagation and backpropagation. Learning happens when these stages are continually repeated with a set of desired input/output pairs. The learning process starts by initially setting all of the weights in each neuron to random values within a small range. The input layer provides the input to each neuron in the hidden layer. The hidden layer neurons produce an activation, which is passed to the next layer. Activations continue to be output in each subsequent layer until the output layer neurons are reached. An actual output is produced after activation at the output layer, and forward propagation ends. This value is compared to the desired output corresponding to the current input and an error value is produced as the difference between the two values. Using this error, the process of backpropagation begins as described below. Error gradients are used to adjust weights in the case of multi-layered ANNs because hidden layers do not have a clear desired output value, and thus cannot have an error by the definition above. Below, the equation formatting is similar to that which is used in the Neural Networks chapter of Artificial Intelligence: A Guide to Intelligent Systems by Negnevitsky [12]. The backpropagation algorithm was utilized for the learning portion of this ANN. The algorithm follows as such:

1) The error gradient (δ) for the neuron in the output layer (k) at the current iteration (t) is calculated.

$$\delta_k(t) = Y_k(t) \times [1 - Y_k(t)] \times e_k(t) \tag{2}$$

Where Y_k is the actual output at the output layer and
e_k is the error at the output layer neuron such that:

$$e_k(t) = Y_{d_k}(t) - Y_k(t) \tag{3}$$

Where Y_{d_k} is the desired output at the output layer neuron.

2) The weights (W_{j_k}) between the hidden layer (j) and output layer (k) are updated for the next iteration (t+1) using this error gradient.

$$W_{j_k}(t+1) = W_{j_k}(t) + [\, \alpha \times Y_j(t) \times \delta_k(t) \,] \tag{4}$$

Where Y_j is the actual output at the hidden layer and α is the constant learning rate

3) The error gradient (δ) for the neurons in the hidden layer (j) is calculated.

$$\delta_j(t) = Y_j(t) \times [\, 1 - Y_j(t) \,] \times [\, \delta_k(t) \times W_{j_k}(t) \,] \tag{5}$$

This is the case when there is a single output neuron.

4) The weights (W_{i_j}) between the input layer (i) and hidden layer (j) are updated using this error gradient.

$$W_{i_j}(t+1) = W_{i_j}(t) + [\, \alpha \times x_i(t) \times \delta_j(t) \,] \tag{6}$$

Where $x_i(t)$ is the input to the hidden layer neuron at iteration t

5) Once the weights are updated, a new iteration runs through forward propagation and backpropagation. This process loops continuously.

IV. Implementation of Hardware Neural Network

The Arduino Pro Mini (APM) was selected for this implementation of a hardware ANN because it is widely available, small enough for quick prototyping of ANN architectures, and inexpensive (the pricing for the chip runs $3 to $10 depending on distributer). The 5V and 16MHz model was chosen for this research. The APM has 14 digital input/output pins. A 6 pin header connects via an FTDI cable to the computer to upload programs or provide USB power. The size of the microchip (.7'' x 1.3'') allows linkage of many microchips together without taking up much space.

The Inter-Integrated Circuit (I2C) bus, which is a built in part of the APM, was used to control communication of data between neurons. Each neuron was given its own memory address through assignments within its program. The I2C works such that a single bus allows communication via a Serial Clock Line and a Serial Data Line through Master-Slave relationships between APM microchips. The Master microchip initiates the clock and demands communication from the Slave microchips. The Slave microchips synchronize with and respond to Master commands. On the APM, pin A4 provides access to the Data Line and pin A5 provides access to the Clock Line. Thus, these two lines come together in simplified wiring along one simple I2C bus that connects all APMs within the hardware ANN.

In order to embed a ANN framework within the APMs using the I2C bus, the Arduino Wire library was needed. This library comes with the Arduino Integrated Development Environment and is preset with functions that allow communication between microchips connected together with the I2C bus. The two main commands used were read() and write() which allow data flow to happen in two directions. Other commands were also used, such as: requestFrom() which allows a microchip to ask for data from another microchip and available() which is a Boolean functions that returns true if data is available from a microchip.

A. I2C Data Transfer

The transfer of data in the Arduino I2C is limited to bytes or characters. This means that a double precision value - which consists of 4 bytes in Arduino - cannot easily be transferred over wire. Precision is vital for the weights learning portion of the Neural Network. We worked around this limitation by using a Union - a data type in the C programming language that allows storing of different data types in the same memory location:

```
union T { byte b[4]; double d; }  T;
```

Here the same memory location refers to a double and a byte. Hence, a double can be encoded as an array of bytes, packaged and sent over Wire to an APM neuron address:

```
int dataSize = 4;

byte packet[dataSize];

for (byte i = 0; i < dataSize; i++)

    packet[i] = T.b[i];

Wire.write(packet, dataSize);
```

The receiving end utilizes another Union that reverses this masking to obtain the double precision and store it into a variable:

```
for (byte i = 0; i < 4; i++)

    T.b[i] = Wire.read();

double value = T.d;
```

B. Configuration of Individual Microchips

Each microchip was configured with a program written in the Arduino language which is based on C/C++ functions. A single neuron class was developed for a hidden layer neuron and an output layer neuron. Similarly, a class was developed for the input element. Within the context of the I2C bus framework, each APM program included a memory address to be used while probing and communicating through the bus. This is very beneficial to making the hardware ANN a dynamic system whereby the architecture can be changed easily with minimal adjustment.

Weights and inputs are each stored in different arrays of type double within each hidden and output neuron. The learning rate is set at .35 for all neurons - this can be easily adjusted based on observation of learning performance.

Below, software components of each type of microchip are outlined.

a) Input Element Microchip (Slave)

A desired output set is defined as a 2D array matrix such that each element is a chain of desired outputs to the corresponding logic operation input values, ordered as XOR, OR, AND, and XNOR:

{{0,1,1,0}, {1,1,1,0},{1,0,0,0}, {1,0,0,1}}

Similarly a 2D array matrix defines the training set of inputs:

{{1,1},{1,0},{0,1},{0,0}}

A requestEvent() function waits continuously for a request from the output APM Master neuron for a desired output value. Then, this value is packaged via the I2C process above and sent over wire. Simultaneously, the next input pair is chosen to be sent to each hidden neuron for the next forward propagation iteration.

b) Hidden Neuron Microchip (Slave)

There are two main functions in the Slave hidden neuron class: requestEvent() and receiveEvent(). requestEvent() waits continuously for a request from the output APM Master neuron, and upon triggering the function, an output value after activation is packaged and sent over wire as part of the forward propagation phase. receiveEvent() is triggered upon the arrival of data from the output neuron in the backpropagation phase. If the correct amount of data, 4 bytes, arrives then this means that the output neuron sent an error gradient value back for learning. Using this value, the weights are updated.

c) Output Neuron Microchip (Master)

There are two main functions that allow data to flow into the Master output neuron: readHidden() and readOutput(). readHidden() requests a neuron output from each hidden layer neuron via I2C and reads these as inputs for the output neuron itself. Once the inputs are received, an actual output is produced for forward propagation. readOutput() requests a desired output value for the current logic operation row from the input element via I2C and stores this value to be compared to the actual output. This comparison allows the calculation of an error. Using this new value backpropagation is initiated. The weights are updated, and an error gradient is calculated and sent back to each hidden layer neuron via I2C.

C. Neural Network Circuit Design

For this proof-of-concept there are 4 APM microchips. As shown on Figure 2, there is one input element, two hidden layer neurons, and one output layer neuron. This ANN architecture mirrors Figure 1 except a single APM input element provides both inputs. The communication functions such that the output neuron in the ANN is the labeled Master APM and the hidden neurons and input layer in the ANN are the labeled Slave APMs. The I2C bus connects the A4 and A5 pins of all the Slave APMs to the A4 and A5 pins of the Master APM. Two wires from each APM, one for Data and one for Clock, allow this connection. Meanwhile, the APM input element provides constant logic operation row input through two wires (can be established via any input/output Arduino pins) to the two respective pins on the hidden APM neurons as values of 0 or 1 - respectively LOW (0V) and HIGH (5V). Two physical switches are connected to the input layer element to allow the user to dynamically change between logic operations to be learned; a switch state of 00 learns XOR, 01 learns AND, 10 learns OR, and 11 learns XOR. Moreover, the current desired output for the corresponding logic operation row at the output neuron is obtained through an I2C request to

the input layer element. Power is provided through the VCC pin for each APM, and all APMs are grounded through the GND pin.

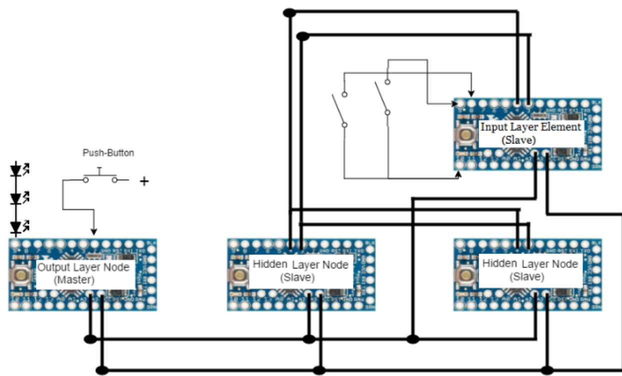

Figure 2. A schematic of the APM circuit is shown. Each neuron is labeled as a node. The two switches are attached to the input element. A pushbutton to toggle between learning and testing, and 3 LEDs to visualize output are attached to the output layer neuron.

For presentation purposes, 2 LEDs are added to two input layer element output pins to help visualize the current inputs (OFF and ON correlate to an input of 0 and 1 respectively). An array of 3 LEDs is provided at the output neuron to visualize the current output. Thus, the spectrum of LED states provide a visualization of the range of outputs (see Table 1). A button is also connected to a pin on the output APM neuron to provide a means to switch between learning and testing. Upon testing (button is pressed down), the backpropagation stops and there is a 1.5 second delay at each APM data transfer to slow down changes between the different inputs to allow the visualization of the LED brightness states. Refer to Figure 3 for visual of finished prototype on breadboard.

TABLE I. THE NUMBER OF LEDS ILLUMINATED SHOWS THE RANGE OF THE OUTPUT. IN THE THREE DIGIT NUMBER, 0 AND 1 CORRESPOND TO OFF AND ON, RESPECTIVELY, FOR EACH OF THE 3 DIODS.

OUTPUT LED STATE	**OUTPUT RANGE**
000	0 - .3
001	.31 - .5
011	.51 - .7
111	.71 - 1

D. Procedure

The following algorithm works with the forward propagation phase and the backpropagation phase to allow for data transfer between APM chips utilizing the I2C Master-Slave framework:

1) Upon being powered, all APM neurons have a threshold input variable set to -1 within their programs.

2) Initially, random weights between -1 and 1 are set in arrays within each hidden and output neuron program.

3) Initiation of the forward propagation process starts at the output (Master) neuron. It requests 4 bytes from each hidden node. The hidden neurons each forward propagate using inputs from the input APM element and their corresponding weights, and send an output to the Master neuron upon request via I2C. The output neuron takes these values as inputs.

4) The Master neuron sends a request to the input APM element to send a 4 byte desired output double value.

5) The input APM element sends the desired output to the Master neuron via I2C.

6) Activation happens at the output layer, and the Master neuron produces an actual output.

7) The backpropagation algorithm starts here:

8) Using the actual output and the desired output, an error is calculated at the Master neuron. Using this, an output layer error gradient is also calculated. This value is used to calculate a hidden layer error gradient.

9) This hidden layer error gradient is sent to each hidden layer neuron.

10) The weights of the output layer are updated using the output layer error gradient.

11) The weights of the hidden layer are updated using the hidden layer error gradient.

12) Loop back to step 3. This time, the input element changes the current input/desired output pair to the next row in the current logical operation.

When the testing button is pressed, the backpropagation steps 7-11 are skipped and the data flow is slowed down by 1.5 seconds to visually show via LEDs the corresponding output to the current input.

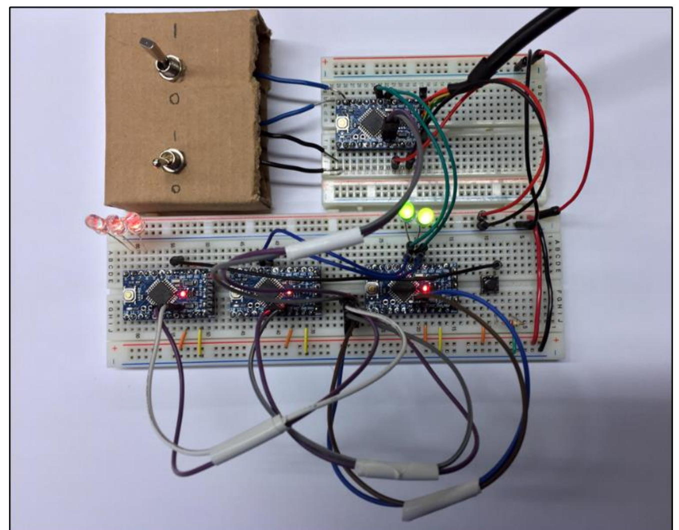

Figure 3. This is a photo of the implemented circuit design (compare to Figure 2). Here the green LEDs show the current input, and the red LEDs show the corresponding output.

V. Results

The hardware ANN successfully learned the logic operations OR, AND, XOR, and XNOR. Refer to Table 2 for the results of 5 trials. The times and average times it takes to learn each logic operation within 49% and 30% error are shown. These benchmarks mean, for example, that if a desired output is 1 and an error of 49% is reached, then the actual output of the ANN would be .51, and if the desired output is 0 and an error of 49% is reached then the actual output of the ANN would be .49. This error benchmark was selected because at this point the ANN would always yield the correct output as long as the actual output was rounded to the nearest integer (0 or 1). A similar test was done at 30% error. At this point all outputs definitely correspond to the desired output with the actual output more fine-tuned. Recorded trials were run 5 times from a random weights start, and time was taken when the error was reached for the two benchmarks. In context of the 30% benchmark, XOR took the least time to learn while AND took the most, respectively around 43 seconds and 2 minutes, 36 seconds. With further training all logical operations were learned with 2% error or less for each corresponding input/output. The Neural Network was also able to learn dynamically upon switching to other logic operations. This means, for example, that after having learned a logical operation such as XOR to within a low percentage error, the switches were used to change to another logical operation such as AND. After the switch, the AND logical operation was also learned within a low percentage error. Every combination of change between logic operations successfully worked to learn the logical operation changed to within 30% error. Tests of changing the architecture - adding more hidden layer nodes - were also done with positive results.

VI. Conclusion

A hardware implementation of artificial Neural Networks with individual neurons on individual chips and backpropagation can be accomplished using inexpensive, off the shelf hardware. Tests showed that the constructed ANN can learn different logic operations dynamically. This system of individual neurons constructed provides a great framework for adjusting the ANN architecture. Simple one-line changes in the code to add unique memory addresses are needed upon addition of APM chips. This is an inexpensive, quick to implement ANN system that can be deployed within minutes, which provides a strong foundation for the ability to build more complex Neural Networks.

VII. Future Work

Research is being conducted in expanding robustness of the ANN system such that while learning, APM neurons will be removed to simulate failure of chip in a real world environment and to show that the learning still is successful without a shutdown of the system. Furthermore, instead of inputting binary values and learning logic operations, a future stage of this research is to explore obstacle avoidance using a remote controlled car and taking input as sensor values using this hardware ANN prototype. The flexible I2C framework and procedure that was developed will be further built upon to make more complex ANN architectures. Specifically, more layers will be added, and more neurons per each added layer along with multiple output neurons will also be explored.

Acknowledgments

We would like to thank Tony Knapp (University of Edinburgh) for exploring possible microprocessors for the task and suggesting the Arduino. We would also like to thank James Meyers (United States Coast Guard) and Jonathan Ray (United States Coast Guard) for their work on earlier versions of Arduino communication and hardware prototyping.

References

[1] Misra, J., & Saha, I. (2010). Artificial neural networks in hardware: A survey of two decades of progress. Neurocomputing, 74(1-3), 239-255. doi:10.1016/j.neucom.2010.03.021

[2] Dias, F. M., Antunes, A., & Mota, A. M. (2004). Artificial neural networks: A review of commercial hardware. Engineering Applications of Artificial Intelligence, IFAC, 17(8), 945-952.

[3] Goser, K. (1996). Implementation of artificial neural networks into hardware: Concepts and limitations. Mathematics and Computers in Simulation, 41(1-2), 161-171.

[4] Liao, Y. (2001). Neural networks in hardware: A survey. (). Davis, CA: Department of Computer Science, University of California, Davis.

[5] Kumar, V., Shekhar, S., & Amin, M. B. (1994). A scalable parallel formulation of the backpropagation algorithm for hypercubes and related architectures. IEEE Transactions on Parallel and Distributed Systems, 4(10), 1073-1090.

[6] Omondi, A. R., & Rajapakse, J. C. (Eds.). (2006). FPGA implementations of neural networks. Netherlands: Springer.

[7] Sahin, S., Becerikli, Y. & Yazici, S. (2006). Neural network implementation in hardware using FPGAs. NIP, Neural Information Processing, 4234(3), 1105–1112.

[8] Maguire, L. P., McGinnity, T. M., Glackin, B., Ghani, A., Belatreche, A., & Harkin, J. (2007). Challenges for large-scale implementations of spiking neural networks on FPGAs. Neurocomputing, 71(1-3), 13-29. doi:10.1016/j.neucom.2006.11.029

[9] Zhu, J., & Sutton, P. (2003). FPGA implementations of neural networks–a survey of a decade of progress. In Field Programmable Logic and Application (pp. 1062-1066). Springer Berlin Heidelberg.

[10] Ienne, P., & Kuhn, G. (1995). Digital systems for neural networks. Digital Signal Processing Technology, 57, 311-45.

[11] Joubert, A., Belhadj, B., Temam, O., & Héliot, R. (2012, June). Hardware spiking neurons design: Analog or digital?. In Neural Networks (IJCNN), The 2012 International Joint Conference on (pp. 1-5). IEEE.

[12] Negnevitsky, M. (2005). Artificial intelligence: A guide to intelligent systems. (2nd ed., pp. 165-216). England: Pearson Education Limited.

TABLE II. FIVE TRIALS FROM RANDOM START WEIGHTS WERE RUN. THE TIMES BELOW ARE ROUNDED IN SECONDS FOR THE CORRESPONDING ERROR BENCHMARKS.

	Trial 1		Trial 2		Trial 3		Trial 4		Trial 5		Average	
Logical Operation	49%	30%	49%	30%	49%	30%	49%	30%	49%	30%	**49%**	**30%**
OR	1:23	2:17	1:24	2:16	1:23	2:16	1:23	2:16	1:24	2:16	**1:23**	**2:16**
AND	1:56	2:37	1:53	2:35	1:54	2:36	1:53	2:35	1:53	2:35	**1:54**	**2:36**
XOR	0:02	0:44	0:02	0:43	0:02	0:43	0:02	0:43	0:02	0:43	**0:02**	**0:43**
XNOR	13:45	1:24	15:16	1:27	14:93	1:25	16:38	1:27	16:78	1:27	**0:15**	**1:26**