

Comparison of Backpropagation Neural Network in FPGA and Microprocessor



Final Project Report

Siyan Lin

Jie-Kai Yang

12/13/2016

1. Project Introduction

The artificial neural network(ANN) which is inspired by the brain behavior has recently become very popular. Neural network is a type of program that learns how to do things instead of being programmed to do things. Neural network can be trained to solve problems like a human brain which is difficult for traditional computers. Hardware implementation of neural network can be extremely useful for a larger of applications, such as pattern recognition and functional approximation [1]. Field programmable gate array (FPGAs) has the ability to be reconfigurable and massively parallel which is useful for the neural network application. In our brain, neural network exhibits high level of parallelism and makes it suitable for FPGA implementation [2].

A simple example of feedforward and backpropagation neural network is shown below in figure 1 below. It contains input layer, hidden layer and output layer, each of the layer consists of input unit and nodes in the layer has a weight to associate with it. The neural network will take the input values and apply math to calculate the weight with all the input values. Training a neural network involves adjusting all the weight such given inputs, the network will give you correct output values. The feedforward and backpropagation algorithm will go forward through the network from inputs to outputs, then propagate back from outputs to inputs by adjusting the weight in each node using the activation function between 0 and 1. The neural network uses the expected outputs for the input to calculated the errors form the outputs to see how far off our predicted output from the correct output. Then we use that error to go back to the network adjusting the weight by very small amounts, so we can get a smaller error next time that we given that input. We repeat this whole process for all the inputs and output in the training until the error we are getting is very small.



For this final project, we are going to present a feedforward and backpropagation neural network which can be trained to solve any range of Boolean equations in FPGA. We will start with a simple XOR gate with 4 hidden neural as figure 1 shown below. We need to provide a set of inputs and with the correct output, since it is a supervised learning. Once the training is completed, our neural network should able to give you the correct output with the correspond inputs. Our back-propagation neural network has the ability to learn and solve the problem relies in the multi-layer perceptron in a non-linear relationship between inputs and outputs. The supervised backpropagation algorithm will try to find the minimum error between predicted output and correct output by adjusting synaptic weight values in the hidden layers.

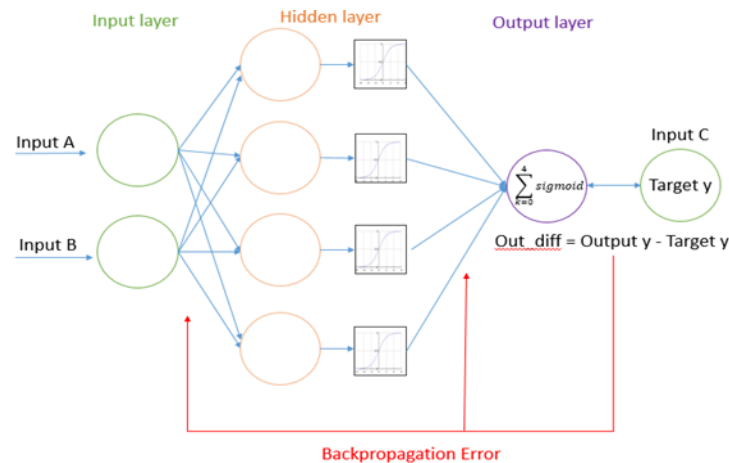


Figure 1. Implementation of feedforward and backpropagation block diagram

2. Discussion of experimentation

We are going to create a neural network to solve any two inputs logic gate Boolean equations using feedforward and backpropagation algorithm implemented into FPGA and JAVA. We are using XOR gate as example, since it is the most interesting one. The neural network consists of two inputs in the input layer represents the two inputs for the XOR gate. We have four neural nodes in the hidden layer to avoid the XOR problem. XOR problem is a simple example of a non-linearly separable problem used to benchmark the learning ability of an ANN [3]. It means at least one hidden layer is need to solve the XOR problem. One output is generated in the output layer shown in figure 1. Here is the pseudocode for the feedforward and backpropagation in Verilog. Basically, our hardware design is that all the wires for the propagation and backpropagation are input driven without latches in the middle, and this design architecture can allow us to finish our one iteration in only two cycles. The values of the propagation results will go to stored registers since that our backpropagation wires are also input driven. We need to use additional registers for storing the propagation results. Moreover, our state machine design is to drive these data of wires, we need to provide data at the backpropagation states for these wires in order to backpropagate. On the other hand, input values are driven by the inputs from the memory. So in our state machine design, there are less wire flow in the propagation state than the backpropagation state.



logic states are three states: propagation, backpropagate, test

(one clock period for propagation, and another clock period for backpropagation)

Assign wire for neuron propagation(multiplication and addition) to the stored registers

Assign wire for backpropagation using stored register values

State machine design for the signal always triggering at **negedge** clock

//-----always block for state machine-----

If current_state equals to propagation

Then

next_state equals to backpropagation

Else //in backpropagation state,

1. update all the weights (hidden weights and output weights)
2. update the iteration register
3. update the outputs, hidden outputs for correction calculation
4. next state equals to propagation


//-----parallelism implementation-----

// -----all the blocks below are in parallel, combinational circuits-----

Call modules fixed_point_multiplier for weights' calculation

Call modules fixed_point_multiplier for correction calculation

Call modules sigmoid function for each neurons

We first design the finite state machine with three states. The first state is propagation state. In the *propagation* state, all the input values will multiply with all the hidden weights and sum up the results send to an activation function, sigmoid function. It will generate an output based on the sum of the activation functions. In the *backpropagation* state,  all the weight in the hidden layers will be updated based on how far the predict value from the target value. A learning rate is used a register variable called iteration is used to keep track when to finish the training process and back to *idle/test* state. The finite state machine is shown below in figure 2.

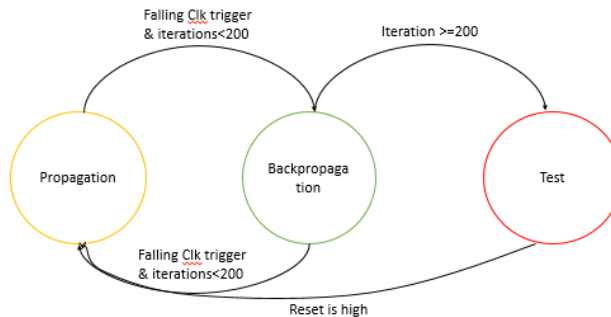


Figure 2. Finite State Machine

A simulation of ModelSim result for the finite state machine is shown in below in figure 3. Three *input_value* are needed because *input_value[0]* and *input_value[1]* are the input values for the XOR gate and the *input_value[3]* is the target output for the XOR gate are shown in yellow color. A new set of data is coming into the FSM at each the falling edge of the backpropagation state. It changes state in every falling edge of the clock and new data coming in the after backpropagation state. The current state and next state are shown in cyan color in the figure. By doing this way, all four combination set of the XOR gate will be train together within the 200 iteration cycles. On each cycle, the combinational logic will be take care all the calculation and the we have a set of registers to store the value at the end of each cycle. The hidden weight of the hidden layer will be changed in order to adapt all this four combination inputs. We will see the *output_diff* will decrease as we iterate more cycle in the process.

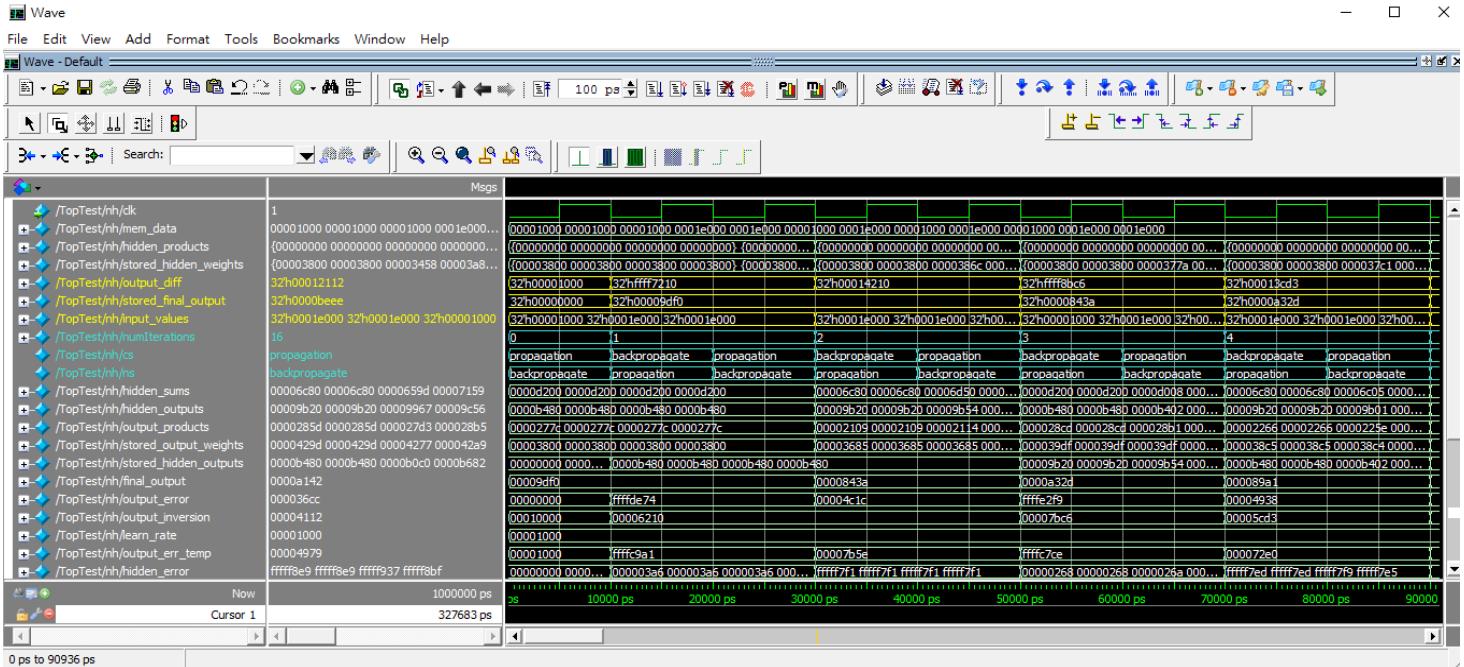


Figure 3. FSM simulation in modelSim

Parallelism

We can see from the pseudocode above, there are a lot of potential parallelism are used in the algorithm. As long as there is no data dependence on the calculation we can put them in parallel. In other words, it means no data should not be waiting for the previous data for next state calculation. For example, in a multilayer neural network, different layers can be processed in parallel. In each node in the layer, each individual neuron can calculate its corresponding weight in parallel in the feed-forward and also the error in the back propagation in parallel. It makes the parallelism in the FPGA be the best candidate for this application. An example of parallelism in RTL viewer for $input * input_weight$ is shown in figure 4. We can see all the data is sent to the multiplier and is being processing in parallel at the same time. This is the reason why FPGA implementation is faster than microprocessor performance which will be discuss later.

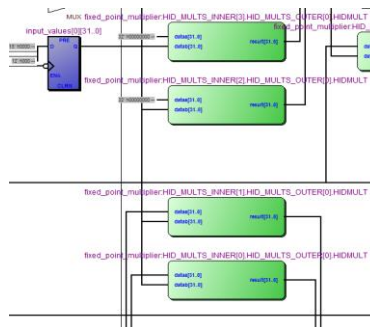


Figure 4. RTL viewer of parallelism

Activation Function / Sigmoid Function

Sigmoid function is implemented by linear approximation. We set our sigmoid function in four segments. If the input is larger than 5, and we assign the slope zero. On the other hand, if the input is between 1 to 2.6 (hexadecimal representation), then the slope is 0.2 as the code shown below. And then input times the slope, and adds the constant value. This figure 5 below is plotted by the decimal values converted from the hexadecimal values, which is not the fixed point representation

Assign wire for val_to_mult equals to

```
If (input < 32'h0001_0000)          32'h0000_4000 // 0.25
  Elseif (input < 32'h0002_6000) 32'h0000_2000 // 0.125
  Elseif (input < 32'h0005_0000) 32'h0000_0800 // 0.03125
  Else 32'h0000_0000; // 0.
```

Assign wire for val_to_add equals to

```
If (input < 32'h0001_0000)          32'h0000_8000 // 0.5
  Elseif (input < 32'h0002_6000) 32'h0000_A000 // 0.625
  Elseif (input < 32'h0005_0000) 32'h0000_0800 // 0.84375
  Else 32'h0001_0000 // 1.0
```

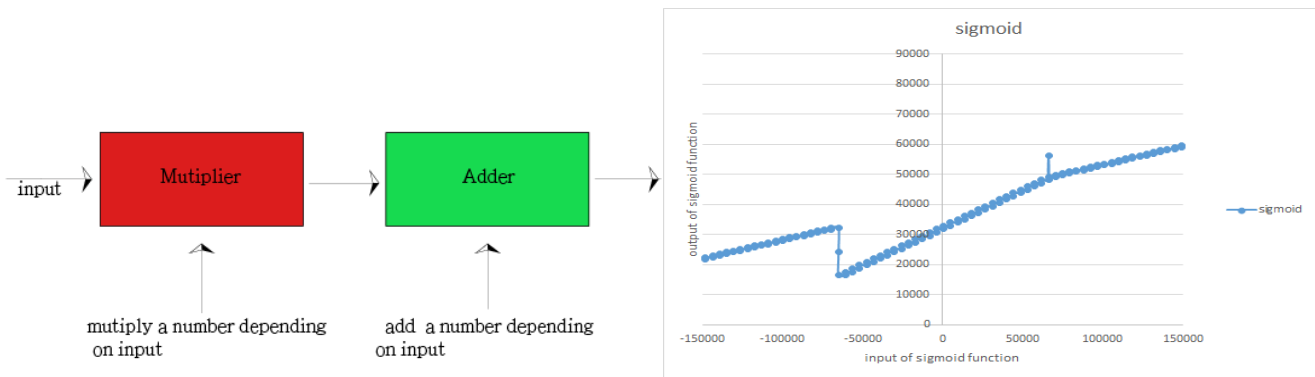


Figure 5. Sigmoid function.



Multiplier

We implement our multiplier by the IP core in QUARTUS with some modification. Since we use the fixed point representation, we implement this function using the technique which truncates the 32 bits in the middle of the result 64 bits as the code shown below. This technique can fix our radix point at the middle of the 32 bits. For example, 1.2 times 3.6 is 4.32. But the computer does not know exactly the radix point is. It only does two digits' multiplication and derive the result of 0432. If we truncate the middle digits from 0432, it will get 43, which is the same result of 4.3 if neglecting the 0.02. Therefore, we can use this technique to set our radix point at the middle of the 32 bits, which means that the fraction bits are 16 bits and integer bits are 16 bits. The sign bit is 1 when the number is negative, else it is positive.

Assign wire for result equals to *original_result*[47:16]

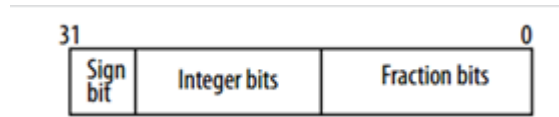


Figure 6. IEEE-754 single precision fixed point representation

Storage and update of weight

As mentioned before, we update our weights at the backpropagation state. In the propagation state, we use the registers to store the values which has been calculated after the propagation. These values will be assigned to the register for the backpropagation state for the weight correction. And we use the gradient descent algorithm to calculate our weights, which basically will assign a large number of correction for the weight update if the outputs are far off the targets. The correction value is proportional to the negative of the gradient of the sigmoid function output. The learning rate for the design is adjustable, and it is between 0 to 1. We found out that when the learning rate is close to one, the training process will be faster for the reason that this term, learning rate, is actually times our correction terms. So the larger the learning rate, the more correction the system will perform during each iteration. We found out that since this algorithm is easily trapped into the local minimum, which means that we cannot solve the XOR problem for our neural network design. We can solve this problem by choosing our inputs randomly from the memory into the neural network.

3. Discussion of results

Training result (JAVA)

The key of this training process lies in the set of weights given to the input values. Neural networks train these weight based on all four combination sequence of the logic gate with known outputs. The weight calculations will be done in using the backpropagation algorithm. The algorithm essentially iterates through the input dataset and change the weight until we reach a predefined iteration count. The figure below shows the error between the target value and predicted value in JAVA simulation with the learning rate of 0.25 and iteration of 500 times in log scale. The lower the error value we get, the more accurate the value the network will predict. It is interesting to see that OR gate is the easiest logic gate to train and XOR gate is the most difficulties one. It is not possible to set up a single neuron to perform the XOR gate operation due to the XOR problem which will described in more detail in the problem encountered section.

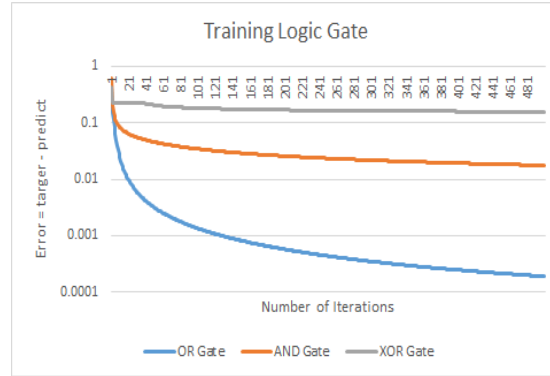
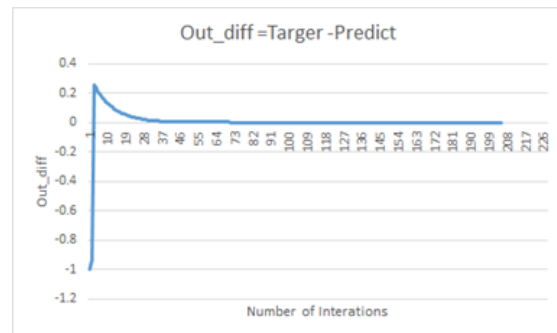


Figure 7. Neural network trains logic gate(JAVA)

Output of the neural network (Verilog)

Our project result shows that network training is effective. And it converges the target we set. As the graph below, the input values [2] is the target we set, we want the neural network to distinguished the logic '1' and logic '0', which can be represented by a large number against a small number. In the example below, we use 00001000 to represent logic '0' and 0001e000 to represent logic '1' as input value.

Data shows that there are two range of output values compared to our target. 00006bf9 represent a logic '0' and 0000e920 represent a logic '1', which indicates that our network converges. And the logic XOR is presented in the below. When the inputs are 0001e00 (1), 0001e00 (1), we get the result of 00006bf9 (0). If the inputs are 0001e00 (1), 00001000 (0), the output is 0000e920 (1). And the output of the neuron can be separated by the two states, which are 0 and 1.



ps	delta	/TopTest/nh/cs	/TopTest/nh/numIterations	/TopTest/nh/input_values	/TopTest/nh/stored_final_output
50000000	+1	propagation	73'h0000000000000009c4	{32'h0001e000 32'h0001e000 32'h00001000}	32'h0000e900
50010000	+1	backpropagate	73'h0000000000000009c5	{32'h0001e000 32'h0001e000 32'h00001000}	32'h0000e900
50010000	+2	backpropagate	73'h0000000000000009c5	{32'h00001000 32'h0001e000 32'h0001e000}	32'h00006bf9
50020000	+1	propagation	73'h0000000000000009c5	{32'h00001000 32'h0001e000 32'h0001e000}	32'h00006bf9
50030000	+1	backpropagate	73'h0000000000000009c6	{32'h00001000 32'h0001e000 32'h0001e000}	32'h00006bf9
50030000	+2	backpropagate	73'h0000000000000009c6	{32'h0001e000 32'h0001e000 32'h00001000}	32'h0000e920
50040000	+1	propagation	73'h0000000000000009c6	{32'h0001e000 32'h0001e000 32'h00001000}	32'h0000e920
50050000	+1	backpropagate	73'h0000000000000009c7	{32'h00001000 32'h0001e000 32'h00001000}	32'h0000e920
50050000	+2	backpropagate	73'h0000000000000009c7	{32'h00001000 32'h0001e000 32'h0001e000}	32'h00006bf9
50060000	+1	propagation	73'h0000000000000009c8	{32'h00001000 32'h0001e000 32'h0001e000}	32'h00006bf9
50070000	+1	backpropagate	73'h0000000000000009c8	{32'h00001000 32'h0001e000 32'h0001e000}	32'h00006bf9

input value[2]=target

output

Figure 8. Neural network of XOR gate(Verilog)

Performance

Performance of Backpropagation Neural Network in FPGA and Microprocessor is shown in the figure 9 in log scale of based 10. We variance with the number of iteration for both FPGA implementation and Java implementation. We use *currentTimeMillis* method of the System class in JAVA to measure the elapsed time of the

program. This data could be different even in the same machine, since that processes shared use of CPU and memory system. The elapsed time will depend on other processes are running on the computer when a test is performed. Running the feedforward and backpropagation algorithm for a single training example has a total of 2 cycles.

1. Feedforward, data come in, calculating predict output
2. Backpropagation, calculating error and update weight, pass value into registers

In total, the training should take $2(\text{states}) * 200(\text{iteration}) + 1(\text{test state}) = 401$ clock cycle to complete on the FPGA. A clock speed of 1000Mhz puts the theoretical running time at around $1 / (1000 \text{ MHz}) * 401 \text{ cycle} = 0.4$ Microseconds which shown in the figure 9 below. However, in the real case, the clock of FPGA may exist slower clock rate due to jitter and clock skew. As result, FPGA may slower than the ideal case of 1000 MHz and require time more than 0.4 microseconds to complete 200 iterations. The data shows that FPGA is about 20 times faster than the microprocessor.

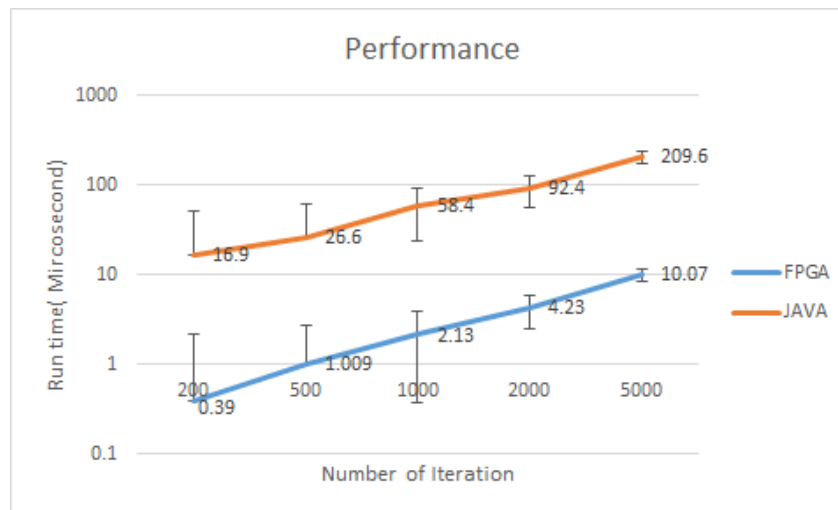


Figure 9. Performance of Backpropagation Neural Network in FPGA and Microprocessor



Area Report

The figure 10 below shows the analysis and synthesis resource usage summary on our neural network on 200 iterations. We had use about 3915 LUTs in the project which we missed this information in the presentation. We believe that the number of LUTs can be further reduced if we pipeline the stages. More block will be used in parallel and more effective way to use the limited resource. We also believe that the area will be increased if we use floating point representation instead of fixed point representation. The result we got matches with what we learned in the lecture that most of the LUTs will has small number of inputs. Most of them are 3 input LUTs and a few with 4 input LUTs. We choose to use the Altera DE2-115 (EP4CE115F29C8) [5] development and education board that we borrow from M5. We also can see the number of I/O pins almost reaches to the limit but we only use 3% of the total logic of element due to rent's rule. Modern FPGA have a lot of logic element which can be used to implement a much larger network, like convolutional neural network. About 500 registers is being used to store the data that we generated during the process.

Analysis & Synthesis Resource Usage Summary				
	Resource	Usage		
1	Estimated Total logic elements	3,924		
2				
3	Total combinational functions	3915		
4	Logic element usage by number of LUT inputs			
1	-- 4 input functions	618		
2	-- 3 input functions	2889		
3	-- <=2 input functions	408		
5				
6	Logic elements by mode			
1	-- normal mode	1098		
2	-- arithmetic mode	2817		
7				
8	Total registers	518		
1	-- Dedicated logic registers	518		
2	-- I/O registers	0		
9				
10	I/O pins	417		
11	Embedded Multiplier 9-bit elements	280		
12	Maximum fan-out node	cs.backpropagate		
13	Maximum fan-out	609		
14	Total fan-out	17905		
15	Average fan-out	3.23		

Flow Summary	
Flow Status	Successful - Mon Dec 12 16:41:17 2016
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	XOR
Top-level Entity Name	NeuralHookup
Family	Cyclone IV E
Device	EP4CE115F29C8
Timing Models	Final
Total logic elements	3,963 / 114,480 (3 %)
Total combinational functions	3,915 / 114,480 (3 %)
Dedicated logic registers	454 / 114,480 (< 1 %)
Total registers	454
Total pins	417 / 529 (79 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	280 / 532 (53 %)
Total PLLs	0 / 4 (0 %)

Figure 10. Area resource usage summary

4. Problems encountered.

We implemented the code in JAVA first, since it is easier to test and debug. Once we get the correct outputs from the JAVA implementation, we can trust the algorithm and to show our implemented is accurate. The biggest problem for us is that we need to transfer the JAVA code into Verilog Code. There are a lot of functions are the JAVA which does not support in Verilog, for example, while loop, sigmoid function and fixed point multiplier. Instead using while loop in JAVA, we break down the code into finite state machine to represent the current states. By this way, we make it easier to synthesis and less resource are being used. Separate the code into FSM also help to improve parallelism by sending data parallel to multiplier in a module. Instead of using sigmoid function of $1/(1+e^{(\text{sum})})$, we use linear approximation to separate the input data into four ranges of data points. Other problems we encountered with the sigmoid function is that we need to set the data into the sigmoid function in the linear range. For example, if the data is not in the linear range of the activation function either is too big or too small, the error between target value and the predict value will not decrease. As result, we will not get the corer predict value or the result is far off.

Other main problem we encountered is the XOR problem of neural network. In the middle stage of the development, we seem ran into the XOR gate problem because we don't get the XOR function working, but other logic gate AND and OR gate work very well with our algorithm implementation. XOR gate problem states that XOR function is a hybrid involving multiple units [6]. In order to solve this problem, we need to implement at least two neural in the hidden layer to store the intermediate value. It is because that $A \text{ XOR } B$ is the same as $A \text{ OR } B$ AND NOT $(A \text{ AND } B)$ write as $(A + B) \cdot \overline{(A \cdot B)}$ [7]. We can think about the XOR gate is the combination of AND, OR and NOT. Therefore, it required multi-layer network or with two or more neural in the hidden layer. It solved problem when we implement more than two neural in the network. But, we also need to carefully picking the initial value in order to avoid the error value stuck in the local minimum.

5. Personal contribution

Siyan Lin

1. Design and Implement the system using the JAVA code
2. Write the documents and the charts
3. Simulate the results
4. Using MATLAB to verify the result
5. Debug the system with MODELSIM
6. Synthesis the design in QUARTUS
7. Read papers and conceive the system
8. Tuning the parameters of the system in JAVA
9. Debug and involve with the system design in VERILOG
10. Format the document and schedule planning

The software side implementation of this project is less challenging than Verilog implementation. I am trying to design and implement the JAVA code as much as similarly to the Verilog implementation, so we can do a fair comparison at the end on performance. I created a fixed neural network 2 input nodes, 4 hidden nodes and 1 output node in JAVA. These go through 200 iterations of the feedforward and backpropagation training algorithm and test all the possible output after training. Applying sigmoid function and fixed point presentation multiplication is easier done in JAVA than Verilog. I did verify the neural network to train a simple network to recognize the AND, OR and XOR gate.

I also designed the finite state machine and figure out how data pass to registers in Verilog. I try to keep it as simple as much by implementing all the combinational logic at the beginning of the cycle and pass the store data before the state changes. It also was my first time experience with SystemVerilog in such larger scale of project. Procedural block *always_comb* is being used to process the combination logic and *always_ff* is used to infer synchronous logic instead of the general purpose always block in Verilog. Those blocks help me to immediately distinguish block which implementing different kind of logic. Googling the error message in Quatus helps me understand how each function works in more depth way. Sometimes, the error message is confusing if you have less solid background is synthesized the Verilog code.

I also synthesized and debugged the system in ModelSim to make sure we are getting the correct predicted value on each cycles. I compared with the hand calculation with the simulation values we get from ModelSim to ensure the algorithm is working correctly. If the value is not right, then I went back to the Verlog code and figured out where does the error occurred since we have the JAVA code for comparison. After getting the correct value, I calculated the cycle number in order to come out with the runtime in ModelSim and JAVA. Use those vales to make the final result and write the document for the final report.

The trickiest issue that I encounter is that our simple neural network may not able to produce accurate results sometimes deepens on the range of the input. During the process of tuning the parameters, I realized that we have chosen an inappropriate dataset parameter. For example, we cannot use zero as input since all number multiple with zero is zero. We also need to find the linear range of the sigmoid function. If you choose a too big or too small value, the value is not going to lay on the situation region. As result, we don't see a lot of reduction on the error. In order to

solve those problem, we end up using a big number to represent as logic '1' and a small number to represent as logic '0'. The error between target value and predict value will never end up to exactly 0 but very close to 0 due to limitation on floating point representation. This required a fine tuning for the initial weight and learning rate for the network in both JAVA and Verilog implementation.



Jie-Kai Yang

1. Design and implement the system using the VERILOG code
2. Debug the and adjust the parameters of the design
3. Synthesize the results with QUARTUS
4. Generate the results and examine the design with MODELSIM
5. Read papers and conceive the system
6. Implement parallelism and state machine of the neural network
7. Debug and involve with the system design in JAVA
8. Write the documents and the charts
9. Technical support for the QUARTUS and MODELSIM

Firstly, we try to implement the neural network in MATLAB to see how its data flow. And I examine through the code, and find out that there are many loops in the NN design. And I try to convert the matlab code directly to the Verilog code. Due to my previous project experience, it is almost impossible to code the long while loop to the Verilog code, since that the compiler of the QUARTUS will directly unloop all these loops and synthesize. If there are lots of data dependencies in the loops, it will cause synthesis to fail. Since it is the hardware design, I know that the possible approach to replace the loop design is finite state machine. I design all the wires in the Verilog to propagate in only one cycle, and the data will backpropagate in the next cycle. By doing so, there will not have latches in the propagation and backpropagation routes. It saves me a lot of time trying to examine through cycles at the waveform file in the MODELSIM to verify the correctness of the design if there are many latches in the design. We use the SystemVerilog to design for it supports the high-level syntax as generate, always_ff, which is easier than Verilog write testbenches in just one .sv file. The reason why we use the negative clock in the design is that I try to do all the things (one iteration) in one cycle at the beginning. I find out that it is impossible for the QUARTUS to synthesis the design with both positive and negative clock, so I leave out half of the positive edge cycle. I design the four nodes in the neural network since that the neural network will only have linear separation for one node, which basically means that the neural network cannot distinguish the inputs of XOR for 00,01,10,11, in the solution graph because it only has one straight line to separate these four data. If we increase the nodes, the classification graph will be a oval, which means that the nodes can be separated into two parts, which corresponds to the outputs 0 and 1. Sigmoid function design in Verilog seems easy, which only has four linear lines to approximate the sigmoid function. But in our neural network design, I have to make sure that all the inputs are at the adequate regions in order to classify the outputs successfully. Moreover, it is hard for the Modelsim to see the relationship between inputs and outputs of the sigmoid function. I look through the Modelsim user manual to find out the way to output these values and plot the graph. It causes me lots of time waiting for the data manipulation (copy to the excel and plot) because of the large amount of data. So I sample some of the datas generate by the

modelsim to plot. The inputs and parameters of the neural network are also the tricky part; it is hard for FPGA design to distinguish whether the neural network has converged or not. The finite digits in the FPGA design also mean that the number of iteration will have an upper bound, for the reason that we only have limited 32 bits. The correction values will increase the weight to approximate the targets bits by bits during every iteration. When the weights are so close to the ideal values, which might be a large number since that we want to separate the 1 (a large number) and 0 (a small number). The next increment of the weight in binary might cause this binary number to flip to a negative value, for our first bit is designed to be the sign bit. From our results, it verifies this idea. If our iteration number exceeds certain values, the output difference (Target value- output value) will go high abruptly. After I realize this problem, I set our learning rate to a small value. By this modification, our weights of our neural network can have a small increment to our target, which will benefit our outcomes of the experiment. We use the gradient descent method in the system. It indicates that the outputs of the neural network can be easily trapped into the local minimum. This is unlike JAVA implementation which only can implement the algorithm using randomly selected inputs to avoid this problem. In sum, I think we really do a quite good result after considering all the effects on FPGA design and software design. Thanks to my partner for the trial and error and file format.

6. Project summary

In conclusion, we presented the backpropagation neural network on FPGA and microprocessor, which uses gradient descent to adjust the network using feedforward and backpropagation algorithm. we implemented a neural network to solve any logic gate Boolean equation as a model of how our brain works. The network structure chosen contains of 2 input nodes, a single layer with 4 hidden nodes and 1 output node which both successfully implemented in both JAVA and Verilog.

Backpropagation neural network has been parallelized, and each of the nodes in the system is calculate and propagate at the same time with different multiplier units at the same cycle. The value has been stored back to the additional registers for correction of the weights in the backpropagation state. We minimize the cycle to two for speeding up the system. One cycle for the propagation and the other for the backpropagation. We use the supervised learning and set up the target for the neural network to train on chip. Both implementation shows the correct expected output of the XOR, AND and OR gate. It showed that given enough hidden units, any Boolean function can be represented by multi-layer backpropagation network. The primary development was done using ModelSim to simulate to ensure correctness. The arithmetic precision of neural network plays an important role in convergence of the network. We choose to implement fix point representation because we want to save area. As result, there is less precision on the output.

Based on the above result, we can safely say that hardware implementation is much faster than the software implementation for backpropagation neural network. The data shows us that FPGA is about 20 times more effective than JAVA implementation in run time by doing same amount of workload. It is because of the implementation of hardware is assigned wire and sending signal in parallel way. On the other hand, implementation of software is sequential which means the program follows in a logical order in sequence.

7. References

- [1] Andres Perez-Urbe. Structure-Adaptable Digital Neural Networks, Ph.D. Thesis, Logic Systems Laboratory, Computer Science Department, Swiss Federal Institute of Technology-Lausanne, 1999
- [2] A. Suliman and Y. Zhang, "A Review on Back-Propagation Neural Networks in the Application of Remote Sensing Image Classification", Journal of Earth Science and Engineering 5 (2015) 52-65 doi: 10.17265/2159-581X/2015.01.004.
- [3] A. Omondi and J. Rajapakse, *FPGA implementations of neural networks*. New York: Springer, 2006.
- [4] W. Sibanda and P. Pretorius, "Artificial Neural Networks - A Review of Applications of Neural Networks in the Modeling of HIV Epidemic", International Journal of Computer Applications (0975 – 8887) Volume 44 – No1 6 , April 2012.
- [5] "DE2i-150 FPGA System User Manual", Altera,
- [6] Floyd, Thomas L. (2003). Digital fundamentals. (8th ed.). New Jersey: Prentice Hall.
- [7] H. Kareem and E. Mohammed, "Design Artificial Neural Network Using FPGA", IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.8, August 2010