TAY
A Neural Network

Jit Kanetkar, Stephan Greto-McGrath, Stephen Carter and Sheng Hao Liu

Abstract—The purpose of this project is to create a neural network capable of machine learning using LabVIEW. The design goal was to teach the MyRio device to accurately and consistently recognize its spatial orientation following a training stage. Applying the principles of machine learning through the implementation of various LabVIEW components, we were able to create a LabVIEW virtual instrument that can self-train using input data from the on-board accelerometer. Exploring the design space, we settled on implementing the neural network with fixed point data precision rather than floating-point precision with all components on the processor as opposed to the FPGA.

Index Terms—Neural Network, LabVIEW, MyRio, Embedded Systems

1 Introduction

The goal of this project was to develop a neural network capable of detecting its current orientation. We were required to design a neuron and then use several of these to create a neural network. This network then needed to be trained using backpropagation to ensure accurate results. We were able to design a virtual instrument in LabVIEW that used a neural network to determine its current orientation. The neuron we used can be seen in figure 1, which was then used to design the network seen in figure 2. The myRIO device is able to train itself to recognize its current position. Once training is complete the device is able to recognize the orientations that the device was trained for. The tutorials on neural networks and backpropagation provided us with a strong base to successfully implement the neural network and its components.

2 Design

Our design approach began with a review of tutorials on how feedforward and backpropagation work. We began by creating a two-neuron-per-layer network which could be easily tested with Matt Mazur's Tutorial on backpropagation. Once verified we were able to extend our network to a three-neuron-per-layer design in order to accommodate the X, Y and Z spacial orientation that the device must learn.

Our first implementation used only one hidden layer and based on the accuracy of our results with only this layer, we decided not to pursue a deeper network. The diagram for this network can be seen in figure 2.

We ran analysis on several different designs, but ultimately were only able to use three due to the inconsistencies in timing with LabVIEW. We noticed that the number of front panels open negatively impacted timing data. Due to time to submission constraints we were unable to re-run the analysis on several designs.

2.1 Design and Pareto Optimal Choices

We set out metrics that would serve to evaluate the performance of our design. Three main metrics were specified: speed (time to train), hardware utilization (FPGA resource efficiency) and confidence (accuracy of output values).

Three designs were made in order to explore each of these design spaces. The first was implemented using all floating point data precision. The second was made with mostly all fixed point precision save for a few exceptions, most notably the sigmoid function used in the neuron as well as the output of the accelerometer. The third, simply placed an arithmetic module (computation of $\frac{\partial E}{\partial Net}$) used in our backpropagation VI on the FPGA to see if it would offer a performance improvement with regards to the time to train the device.

2.1.1 Exploration

Upon testing these three designs, we found the most optimal choice was the design that used mostly all fixed point data representation and had no portion on the FPGA. This gave us very accurate results with respect to our X, Y and Z orientation, more accurate than using floating point, and equally accurate as the FPGA implementation. Therefore, we can say that this design was equally accurate and used less hardware. We chose this design over the floating point implementation despite a small speed improvement of about $200ms$ using the floating point representation because of its significantly higher accuracy, which we believe to outweigh the small improvement in training time. These trade-offs can be seen in the bar graphs in figures 5, 4, and 3.

We expect that the reason fixed point operations were slower than floating point is caused by the conversions. In our design, we constantly convert from the double precision values to fixed point from the accelerometer and then fixed to double precision while performing exponentiation. These repeated conversions likely caused the lack of improvement with fixed point arithmetic on the FPGA.

2.1.2 Learning Time

Our design takes 3 samples from the accelerometer, one for each direction and allows the device to self-train using these values, repeatedly correcting itself using backpropagation in

a loop. In order to find an ideal number of loop iterations, we performed a binary search, testing out different values and starting with 15000 passes through the loop. We found that with a minimum of 3750 loop iterations the device was able to reliable and accurately know its orientation.

3 Implementation

In the end, our implementation was very test focused. While we didn't follow the test driven mantra of setting up tests and then creating our modules, we primarily ensured our VIs were functioning using LabVIEW's Unit Test Framework.

Thanks to Matt Mazur's Tutorial on Back Propagation, we were able to leverage LabVIEW's unit testing framework in order to use a systematic method to ensure valid results. We had 3 main tests, 2 for the neuron and 1 for the full backpropagation module. Due to the simplicity of the neuron equation, as seen below,

$$Output = \sum weight_i \cdot input_i + offset \tag{1}$$

we simply made our own arithmetic test case and were able to validate our results. We performed a similar setup for the sigmoid functions. These unit tests were particularly useful for when we were switched from double precision to fixed point arithmetic. We were able to immediately ensure that our neural network was still behaving similarly to the what was expected.

However, our unit test for backpropagation was far more critical. Due to the complexity of the backpropagation stage, with multiple partial derivatives, it was a necessity to ensure it was working properly. For our unit test, we just used Matt Mazur's input values, in dynamically sized arrays, and fed them into the backpropagation stage. First we ensured that our values for $\frac{\partial E}{\partial Net}$ were correct. Knowing the first part of backpropagation was working properly, we were able to start ensuring we were properly calculating the output values. Eventually we were able to finalize a unit test capable of calculating values for both the first and second instances of backpropagation.

Knowing our backpropagation was functional, we began to try to set up the two layer network from Matt Mazur's tutorial. Initially, due to miscommunication between group members, we implemented a backpropagation stage which was using input values from the previous iteration of the loop, instead of the forward facing layer in the network. Upon clearing up this mistake, we were able to run Matt Mazur's example network successfully. We didn't use LabVIEW's unit testing framework for this module, instead setting the loop iterations to execute manually and comparing it to the values from Matt Mazur's tutorial.

Knowing how to make connections, we began to setup our neural network. We followed the various images from the assignment description to make a neural network with 1 hidden layer and 3 nodes in the hidden layer.

4 Conclusion

The neural network we designed was capable of detecting the orientations it had been trained in. The system could be trained for three dimensions and was self-training. LEDs were used to show when training was running, complete, and the current orientation of the device.

4.1 Results

We were able to design a successful network capable of detecting orientation. We explored three design choices as discussed earlier in the design section. Although all three design choices worked, we saw little use in putting our $\frac{\partial E}{\partial Net}$ virtual instrument on the FPGA. The results of training time, confidence, and hardware utilization can be seen in figures 5, 4, 3 respectively. These figures show that using fixed point precision for backpropagation, with no FPGA utilization, has the highest confidence (lowest mean squared error) with zero hardware utilization, and only a slightly higher cost in training time, compared to the other designs.

4.2 Reflections

Overall, there are quite few things we would do differently while implementing this project. In short, we initially built our neural network while ignoring the limitations of the FPGA.

4.2.1 Fixed Point

Our design initially involved building a full neural network using just the double precision floating point values. Since these are given to us by the accelerometer, this was a nice way to avoid type conversions. However, this resulted in a significant amount of time to be spent while converting all double precision wires into fixed point wires.

4.2.2 Sigmoid

The exponentiation VI from LabVIEW only worked with floating point numbers. Because of this, we were unable to move the full Neuron VI to the FPGA. Ideally, we should have used the "Fixed Point Math Library" in order to implement a more efficient exponentiation. However, due to timing constraints, we were unable to do so.

4.2.3 Arrays

Unbeknownst to us, dynamic length arrays are not allowed to be used with the FPGA in LabVIEW. Trying to keep our code similar to Matt Mazur's example (a 2x2 Neural Network), we thought this would be an ideal situation. However, as we moved components to the FPGA, we weren't able to get a significant improvement. Because we weren't able to pass in our arrays directly to the FPGA, we instead moved some math modules. However, since these modules would be inside of a for loop on the processor, we weren't actually performing tasks in parallel. Due to the speed of the MyRio's ALU, we didn't gain a significant advantage.

We should have used clusters instead of arrays for this whole process. Each group of weights and output data was continuously a 1x3 array. If we had instead used a cluster, with 3 values, we would have been able to create a more modular neural network which could have been more easily implemented on the FPGA.

Acknowledgments

The authors would like to thank Professor Brett Meyer, PhD.

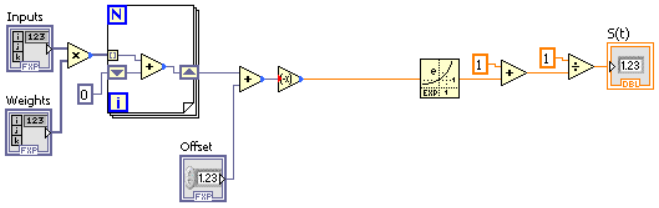

Fig. 1. Neuron design in LabVIEW

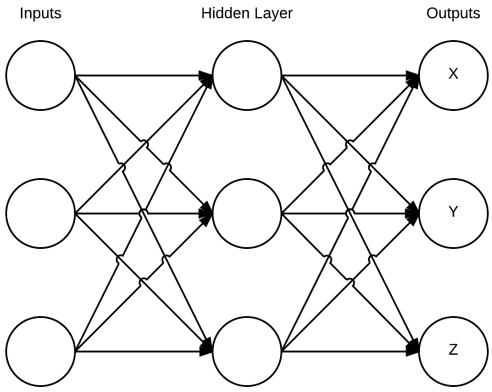

Fig. 2. Layered network design with single hidden layer

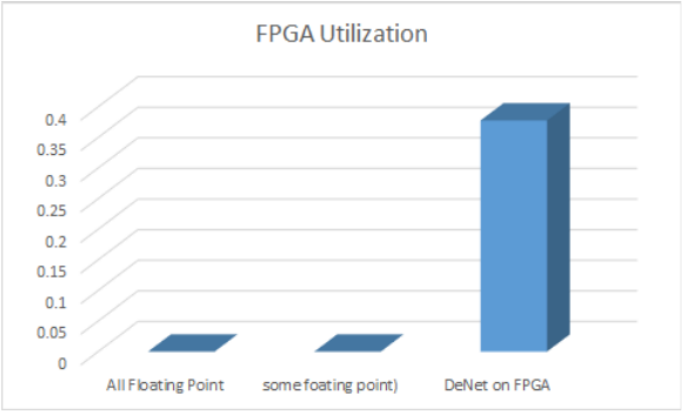

Fig. 3. FPGA utilization of our three designs

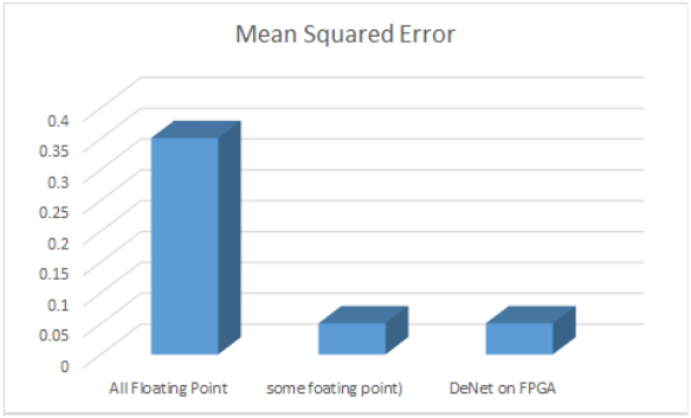

Fig. 4. Mean squared error of our three designs

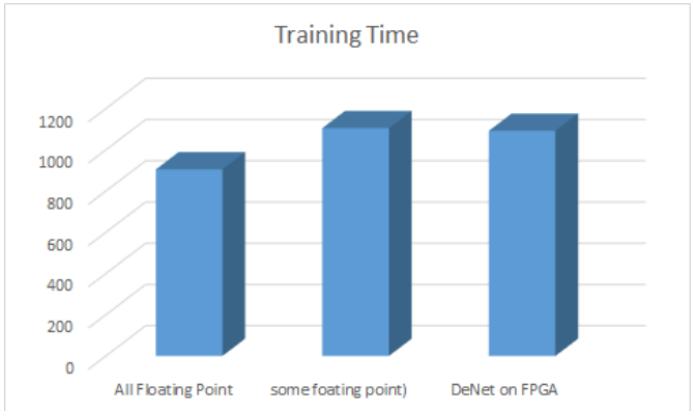

Fig. 5. Time to train of our three designs