MAIA: Modulation Classification with FPGA

Progress Report

Brenda So

Abstract

This paper summarizes the progress that I have made since the beginning of the semester. The goal of my project is to implement a neural net that performs modulation classification on a Field Programmable Gate Array (FPGA). The paper starts with a description of the neural network architecture that we are using and the basic elements of the architecture. It then describes the challenges that we faced when implementing the neural network in software and details the solutions that made the neural net work. Afterwards, the paper explores two of the essential building blocks of the neural net and how they are translated to hardware (my partner, Cory, build the other essential building blocks of the neural net).

The second section of the paper describes the overall hardware design and the implementation progress. The implementations of the first and second convolution stages are discussed extensively and important control mechanisms are highlighted as appropriate. The implementation results and resource utilization are discussed in the section as well.

At the end, the paper describes the implementation of the wireless setup that is going to be used to test the FPGA system, as well as the next steps of the project.

Neural Network Implementation

Our modulation classifier uses a convolutional neural network that was proven to work in literature. The neural network architecture was designed by O'Shea [3], and the high level architecture is shown in Table 1.

	Layer	Output dimension	Stage
1	Input	2 x 128	N/A
2	Convolution + ReLU (128 filters, size 2x8)	128 x 121	1
3	Max Pooling (size 2, stride 2)	128 x 60	
4	Convolution + ReLU (64 filters, 1 x 16)	64 x 45	2
5	Max Pooling (size 2, stride 2)	64 x 22	
6	Flatten	1408	N/A
7	Fully Connected + ReLU	128	3
8	Fully Connected + ReLU	64	4
9	Fully Connected + ReLU	32	5
10	Fully Connected + ArgMax	10	6

Table 1 : Neural Network Architecture of MAIA with respective output dimensions and stages

The five main operations of the CNN in the architecture we used are:

1. **Convolution**

   Convolutional neural networks use convolution to generate another mapping of the inputs. Within each convolutional layer is a series of filters. Each filter will convolve with its input, generating another representation of the input known as a feature map. At a higher level, these filters are feature identifiers. As we use the filter to sample over the larger input, we can apply this filter as a feature detector to other parts of the inputs to identify the existence of common features.
2. **Rectified Linear Unit (ReLU)**

   ReLUs are useful in introducing non-linearity into the model. Since the raw data samples collected by a radio has undergone non-linear transformatoins through the channel, we cannot solely rely on linear models to recover its modulation scheme. ReLU acts as an activation function, where it is “activated” when the input is positive, and deactivated when input is negative.

3. **Maximum Pooling**

   Maximum Pooling effectively downsamples the feature maps we have in the CNN. We can obtain a lot of new features through convolution, yet to process all these features at every step of the neural network is computationally expensive. In this regard, downsampling along each feature map is useful. This downsampling operation is called pooling in CNN. During downsampling, we take a window of a predefined size on the feature map and extract the maximum number from the window. We make a stride of predefined distance between two pooling operations, hence downsampling the feature maps.

4. **Matrix Multiplies**

   Matrix Multiplies are the basic operations of a fully connected (FC) layer in a CNN. In a fully connected layer, all the input values and the weights of the layer are all pairwise connected to one another, each output element is the sum of all the products from a particular weight, and the output dimensions of the FC layer is nx1, where n is the number of weights in the layer.

5. **Argument Maximum (ArgMax)**

   Argument maximum is often used as the last layer of a CNN. In a classification task, we want to know what class the input belongs to. The input of argmax is usually an nx1 matrix, where n is the number of classes in the classification task. The index of the element with the highest value is chosen to be the class of the input.

We used Tensorflow, an open-source deep learning framework, to reconstruct and train the neural network with the radioML dataset [2].

One of the challenges that we faced when implementing the neural network is the lack of details in the paper. It is very common in neural network design that a change of parameters or objective functions can alter the results drastically, or in worse cases, the model would not converge at all. To compensate for the lack of details in the paper, I consulted one of his earlier works [4], where he made the source code openly available to the public [4]. However, his earlier works uses another neural network framework – Caffe, while our implementation uses Tensorflow. Therefore, we need to read through the documentation of another framework to understand what the older version does. The following are the conclusions we drawn from the Caffe framework:

1. The objective function used to optimize the model is the Adam optimizer[1] with a learning rate of 0.01.
2. All training data is passed through the model for 100 times each.
3. Fully connected layers in the Caffe implementation does not use any bias terms. Tensorflow by default sets the bias term to be non-zero, which was fatal to training since the network does not converge if a bias term is used. Hence, we need to disable the bias term.
4. The convolution is unpadded, i.e. the first convolution takes the first n samples from the input, instead of only taking the first input and padding the input with n-1 zeros.

After we make the above changes, the model converges and reaches satisfactory classification accuracy, as shown in Figure 1. As expected, the better the SNR, the higher the classification accuracy. Note that at SNR higher than zero, the accuracy stayed at around 80%.

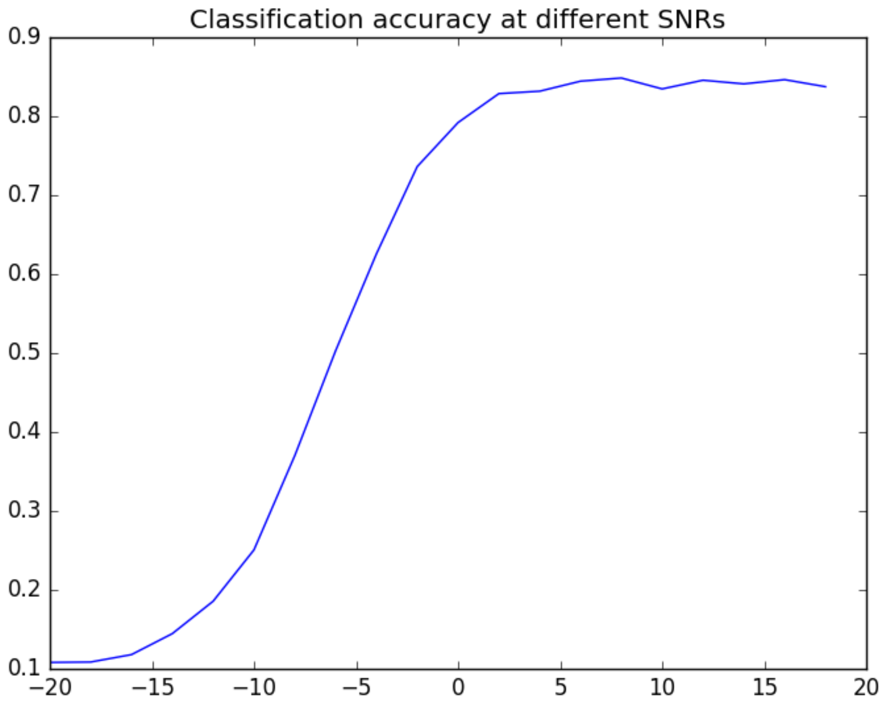

Figure 1 : Graph showing classification accuracy of classifier under different SNR. The x-axis is the SNR, the y axis is the classification accuracy.

Basic Building Blocks

To build a fully dedicated hardware architecture of the neural network, we need to design and build the basic blocks. The basic blocks were then implemented in VHDL on Vivado, an FPGA programmer. I built two of the basic blocks: convolution and ReLU.

1. **Convolution**

   We used a transposed FIR filter to implement convolution. The convolutions that we use in the model is mostly a 1D operation. Although the second layer of the model has 2D convolution, the first dimension is only size 2, hence it is easily calculated by summing the output of two 1D convolutions together. An FIR filter can be realized easily in Vivado with the FIR compiler. The FIR compiler can be used to optimize for different metrics, such as speed and area. There are two realizations of FIR filters – a systolic form and a transposed form. The difference between the two is that the systolic FIR filter is optimized for area whereas the transposed form is optimized for latency. We chose the transposed form since we want to be able to process real-time data samples as fast as possible in our test environment.

2. **ReLU**

   The ReLU functional block is easily realized with a multiplexor. In two's complement, the most significant bit (MSB) indicates whether the the number is negative or not. The address bit of the multiplexor is determined by the MSB of the number, which selects between the input (0) and zero (1). We used a few test inputs to test and verify the functionality of the ReLU unit, which is shown in figure 2

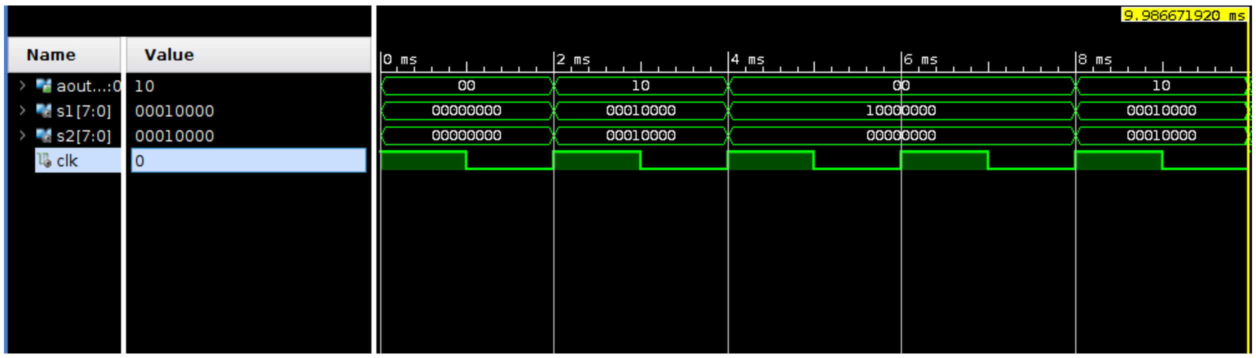

Figure 2: Simulation of ReLU functional unit in Vivado. aout is the output of ReLU, while s1 is the input of ReLU. The result is represented in binary form

High Level Hardware Design

After building the basic blocks, we proceeded to map the software architecture to a hardware architecture. The two design goals that drive our design are:

1. Reusability : Make design patterns that can be reused in other parts of the design
2. Resource Constraints : The Zedboard has very few resources that we can use; we need to be resourceful about using limited resources.

We used FIFOs (First In First Out) to implement IO of each layer because FIFOs are commonly used in real-time DSP systems. We broke down the overall architecture into five stages, as shown in Table 1. The whole neural network can be broken down into two main modules. We observed two patterns in the software architecture that allows us to modularize the whole neural network

1. **Convolution + ReLU + MaxPool (Conv-ReLU-MaxPool)**

   In this module, the inputs are first convolved with a filter compatible in size; the subsequent output is activated through the ReLU and downsampled through the max pool. This pattern shows up from layer 2 to layer 5 in the neural network architecture. A naïve implementation of this pattern would be to make 256 FIR filters for layer 2 and 8192 filters for layer 3; yet it is unfeasible because we only have 220 DSP slices on the Zedboard and one eight-tap FIR filter takes up 8 DSP slices. We would need more than 100M DSP slices to build such a system! To resolve this issue, the DSP slices need to be shared between filters, i.e. multiple coefficient sets need to share the same FIR filter. This is also feasible in the timing aspect, since the sample is produced as 44.1 kHz while we can clock the FPGA up to 100 MHz.

   We also need to multiplex the inputs of the FIFO to accommodate for multiple coefficient sets per FIR filter. Since all data points need to go through each FIR filter once, after clocking in a data sample from the FIFO to the FIR filter, it needs to be loaded back to the back of the FIR filter so that the same data samples are saved for the subequent set of filter coefficients.
2. **Fully Connected + ReLU**

   This pattern shows up from layer 6 to layer 10 of the neural net architecture. This design pattern is implemented with a multiply-accumulator (MAC). There are two inputs to the

MAC – the input FIFO from the previous stage, and the weights, which will be stored in ROM. We cycle through the values of the input FIFO, updating the weights once every cycle, to obtain the output of the Fully Connected + ReLU stage. This output is subsequently fed into another MAC and ReLU in a similar fashion.

Implementation of First Convolution Stage

This first convolution stage emcompasses layers 2-3 in the network architecture. The high level block diagram of this stage is shown in Figure 3. The procedure to implement the first Conv-ReLU-MaxPool module works as follow:

1. Initialization: Load the input into a FIFO until the FIFO fills up
2. Read data samples from FIFO to FIR filter. As each data sample is read, write the sample back to the FIFO.
3. Push the output of the FIR filter to the ReLU and then MaxPool. Note that the first seven outputs of the FIR filter is not valid because the convolution operation that we use is not padded.
4. Read the output of MaxPool once every two clock cycles. (max pool is the architecture has stride 2)
5. After going through all the data samples in the FIFO, change the weights of the FIR filter
6. Repeat steps 2-6 until we have gone through all the weights once.

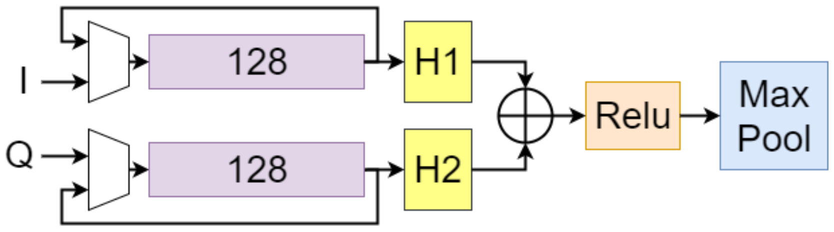

Figure 3: Overall block diagram of first convolution stage. The purple blocks represent FIFOs of length n. (e.g. a purple block with the number 128 in it indicates that it is a 128-byte FIFO)

With this setup, the output of this layer is laid out in a row-wise manner, i.e. the first 121 outputs of the module are the first row of the output matrix, the second 121 outputs of the module are the second row of the output matrix etc. The final output of the MaxPool is stored in an intermediate FIFO that bridges between the first and second convolution stages.

FIFO Implementation Details

The FIFO is implemented with a FIFO Generator in Vivado. The generator produces a 128-byte FIFO with two control pins – read enable (read a value from the FIFO) and write enable (write a value to the FIFO). The control of these two pins depends on whether we are in step 1 (initialization) and whether the FIFO is full or not (in steps 1-6). Table 2 shows the control of the two pins in a Karnaugh Map.

FIFO Full	**Init**	**Read Enable (rd_en)**	**Write Enable (wr_en)**
0	0	0	1
1	0	1	0
0	1	0	0
1	1	1	1

Table 2: Karnaugh map and read and write enable of FIFO in first convolution stage

The state of read enable is the state of FIFO full. To detect whether the FIFO is full, we use the built-in *full* pin. However, the FIFO has a read latency of one clock cycle. This means that at each reading, the state of the full pin oscillates and is unstable. Therefore, we need to use the *almost full* pin of the FIFO to monitor when the FIFO is "full" after we fill up the FIFO initially. Full is defined to be *full* (during initialization) or *almost full* (during processing).

FIFO write is enabled when the FIFO is not full (during initialization) or is one step behind read enabled (during process). This is to compensate for the fact that there is one cycle lost to read latency.

Weight Changing Mechanism

The first convolution stage has 256 sets of 8-tap filters (128 sets for in-phase components, 128 sets for quadrature components). It is unfeasible to place all of them on the FPGA at the same time. As a result, we only use 2 filters (one for In Phase, one for Quadrature) and change the weights as we finish processing the 128 data points in the IQ FIFO. After each weight change, we load in the same 128 data points again. The FIFO, hence, practically works as a circular FIFO.

We only change weights when we have gone through all 128 data points in the input FIFO. To implement this, we have a data counter that counts from 0 to 127. When the counter reaches 127 and data is valid, we increment the weights counter. Note that we cannot increase the weights counter during initialization because during initialization the FIFO is not full yet.

Implementation Results

The first convolution stage is implemented in VHDL. We simulated the hardware on Vivado to verify the results of this stage and it has been successfullly synthesized and implemented as well. The resource usage of this stage is shown in Table 3. We are satisfied with the results since the resource utilization is very low.

Resource	Used	Available	Utilization (%)
Look Up Table (LUT)	999	53200	1.88
LUT RAM	30	17400	0.17
Flip Flops (FF)	809	106400	0.76
Block RAM (BRAM)	1	140	0.71
DSP Slices	16	220	7.27

Table 3: Resource Utilization of FPGA Fabric

Implementation of Second Convolution Stage

The second convolution stage encompasses layers 4-5 of the network architecture. During convolution in the second stage, each input row is convolved with a separate 16-tap filter and the results are added together to produce an elmeent of the output row. We took a drastically different path to implement this convolution stage as compared to the first one. This is because this stage effectively requires 8k FIR filters (64 different convolutions across 128 rows of inputs), which exceeds our DSP slice limits. Therefore, we did not use an FIR filter for this stage, but instead uses a MAC to calculate the results, where each MAC produces one row of the output matrix. The high-level procedure to implement this stage is as follow:

1. Take 16 elements from one row (60 elements) from the input FIFO and the corresponding filter from ROM – these elements are placed in the FIFO, which becomes the input of the MAC. We call the 16-byte FIFO x_fifo and filter FIFO h_fifo.
2. Run the MAC for 16 clock cycle, dropping the first element and reading back the other 15 elements into x_fifo. This act as the “sliding” mechanism during convolution. All the

elements in h_fifo is read back into itself. Note that this output is only a partial sum of an element in the output matrix.

3. Read the output of MAC and push it an output FIFO.
4. Push one value from the input FIFO to x_fifo
5. Repeat steps 2-4 for 45 times. These operations essentially slide the filter through a row and store the values.
6. Repeat steps 1-5 for 128 times. Howoever, instead of reading the output of MAC and push the value directly to the output FIFO, we read the output of the FIFO and add it to the output of the MAC before writing the value back into the output FIFO. This operation sums up the convolution of each row and produce the final output row afterwards.

We are in the process of implementing a state machine to coordinate between all the multiplexors and FIFOs. The high level block diagram of this stage is shown in Figure 4.

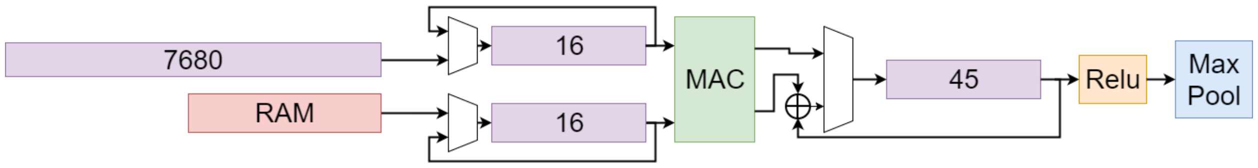

Figure 4: Overall block diagram of second convolution stage

Wireless Setup

We are going to use an SDR to test the FPGA by sending out IQ signals of a selected modulation scheme from the transmitter. RadioML, the dataset that we use to train the neural net, contains ten specific modulation schemes (eight digital and two analog). We implemented a transmitter with the ten schemes on an SDR (USRP N200) with GNU Radio Companion. In addition, we added a noise enable feature the user can enable noise and adjust the amount of noise added to the transmitted signal.

The transmitter has five stages, as shown in Appendix C. The first stage is the source. Since we have both analog and digital modulation schemes, we need both digital and analog sources. Currently, the analog source is a 1k triangular wave with amplitude 1 and the digital soruce is a random source that generates numbers 0 – 256. The second stage is modulation, where the digital and ananlog sources are modulated by ten different modulation schcmes. The third stage is select, where only one stream of symbols (selected by the user on the GUI) is sent to the transmitter. The fourth stage is noise, where the user can add Gaussian noise to the signal. The amplitude of the Gaussian noise ranges from 0.125 to 10 (SNR : -18 to 20 dB). This is implemented with a noise source and a multiplexor to select the input of noise or constant zero source (disable noise). After noise addition, the final signal can be sent over the air through the transmitter.

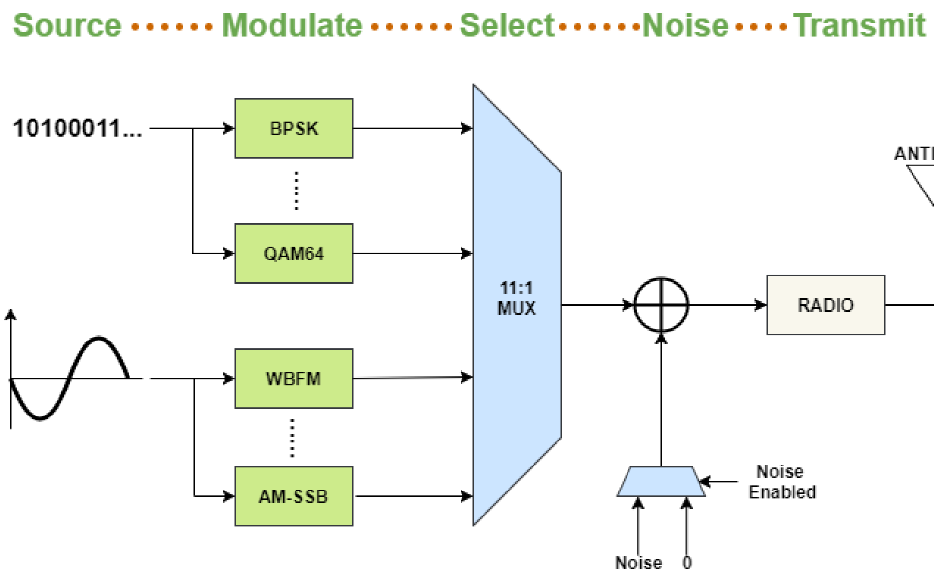

Figure 5: Appendix C : Block Diagram of Wireless Test Environment Setup

The system was implemented and verified with a spectrum analyzer. As the SDR was transmitting a signal at the ISM band (2.4 GHz), the spectrum analyzer showed a peak at 2.4 GHz. We also know that the peak is not due to the presence of Wifi since when the SDR was turned off, the peak disappeared.

Next Steps

1. Finish implementing and verifying second convolution stage
2. Implement the last three matrix multiplication stages
3. Integrate neural network with input and output

Bibliography

[1] Diederik P. Kingma and Jimmy Ba, “Adam: A Method for Stochastic Optimization, ”CoRR, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[2] “RadioML – Advancing Radio Systems that Learn from Data and Approximate the Complexity of the Real World,” 2016, https://radioml.com/.

[3] J. H. Timothy J O’Shea, “An Introduction to Deep Learning for the Physical Layer,” CoRR, vol. abs/1702.00832, 2017. [Online]. Available: http://arxiv.org/abs/1702.00832

[4] T. J. O’Shea and J. Corgan, “Convolutional radio modulation recognition networks,” CoRR, vol. abs/1602.04105, 2016. [Online]. Available: http://arxiv.org/abs/1602.04105