Convolutional Neural Networks for FPGAs

May Yee Brenda So and Cory Nezin

May 6, 2018

Abstract

Convolutional neural networks (CNNs) have recently been gaining momentum in the artificial intelligence and machine learning community. They are traditionally implemented on Graphics Processing Units (GPUs), which supersede performance but are physically large and power-consuming. However, for devices that require low latency and power consumption, such as communication electronics, GPUs are impractical and Field Programmable Gate Arrays (FPGAs) offer a better alternative. They are smaller, more power-efficient and are designed for parallel processing. Using wireless modulation classification as an example, we introduce a fully dedicated hardware architecture that implements a CNN on an FPGA. Our contribution comes in two-fold. First, we designed a small CNN that can achieve state-of-the-art accuracy at 10% of the original size. Second, we implemented a hardware version of the proposed CNN that can fit within the hardware constraints of an FPGA with lower latency and power consumption than that of a GPU.

Contents

1 Introduction

Wireless modulation classification has been, and continues to be an important engineering problem. It is the intermediate step between signal detection and demodulation, and is relevant to applications such as cognitive radio research, jammer identification and situational awareness in military/adversarial environments. It was found that CNNs are the most accurate in classifying modulations compared to other methods that perform the same task [1]. However, few of them have been implemented on actual communications devices, which usually have FPGAs in them.

FPGAs are often used to design compact and power-efficient digital circuits. They have built in digital logic such as look up tables (LUTs), registers and connections that allows users to program their own circuits using hardware description language (HDL). They allow users to optimize their circuits for application-specific parallel processing with low power, but it comes at the cost of resource limitation. Thus the biggest challenge in FPGA design involves trading off speed for hardware resources, and vice versa. There have been several CNNs proposed to classify wireless modulations from raw IQ samples. [1] [2] However, the proposed CNNs have more than 300k weights and were tested on GPUs, rendering them inapplicable on communication devices that have limited memory and low power usage.

In this paper, we introduce an implementation of a CNN modulation classifier as a digital circuit on an FPGA prototype board – the Zedboard. Section 3 discusses the design process of our neural network and its performance. Section 4 details our CNN design on an FPGA, and Section 5 discusses the system level implementation and test results.

2 Background

2.1 Modulation Classification

When transmitting wireless signals, it is required that one modulates the original signal with a high frequency periodic carrier signal. the periodic property allows the receiver to demodulate and recover information easily. Some common wireless connections and their corresponding modulation schemes are shown in table 1 Current modulation schemes can be

Wireless Signal	Modulation Scheme Used
WiFi	QPSK
Bluetooth	GFSK
GPS	BPSK
Broadcast Radio	FM or AM

Table 1: Common wireless signals and correspond modulation schemes

classified into two classes:

1. Analog Modulation: Transfer an analog baseband signal over an analog channel at a higher frequency. Examples include AM (Amplitude Modulation) and FM (Frequency Modulation) radio.

2. Digital modulation: Transfer a bitstream signal over an analog channel by encoding the bitstream with an analog carrier. Examples include binary phase shift keying (BPSK) modulation and QAM (Quadrature Amplitude Modulation) schemes.

A generic received signal can be represented by equation 1.

$$r(t) = c(t) * s(t) + n(t) \tag{1}$$

Where $r(t)$ is the received signal, $s(t)$ is the transmitted signal, $n(t)$ is additive noise, and $c(t)$ is the time varying impulse response of the wireless channel. The goal of our classifier is to predict the modulation class that $s(t)$ belong to with $r(t)$ as the given information. The transmitted and received signals are commonly represented in IQ form, where I represents the real part of the signal and Q represents the imaginary part. However, a received signal that takes realistic distortions into account take the form of equation 2.

$$r(t) = \exp(jn_{Lo}(t)) \int_0^{\tau_0} s(n_{clk}(t-\tau))h(t,\tau)d\tau + n(t) \tag{2}$$

In this equation, $\exp(jn_{Lo}(t))$ represents modulation by a residual carrier random walk process, $s(n_{clk}(t-\tau))$ represents resampling by the residual clock random walk process, $h(t,\tau)$ represents convolution with a time-varying channel impulse response and $n(t)$ represents additive noise (that might not be white). The time-varying channel impulse response is usually modeled by two fading models: Rayleigh scattering and Rician scattering. Rayleigh

scattering (a.k.a. Rayleigh fading model) models signal transmission with no direct line of sight and places where there is a lot of scattering, such as in a city. Since there is a lot of scattering, the same signal can arrive at the receiver through different paths. By the Central Limit Theorem, the sum of the signals follows a normal distribution. Hence, the channel is modeled as a zero-mean Gaussian process. Rician scattering (a.k.a. Rician fading model) models signal transmission with direct line of sight and scattering. It is similar to Rayleigh scattering, except that the channel models as non-zero mean Gaussian Process.

2.1.1 State of the Art

On the receiving end of a radio communication system, a matched filter is commonly used to recover the symbol. The received signal is convolved with expert designed filters, forming peaks when the matching symbol is found. [3] However, by using a match filter, we assume that users know what signal and what modulation scheme they are going to receive (hence knowing which expert designed filter to use). Moreover, a matched filter is only optimal in the presence of Gaussian Noise. Other effects, particularly non-linear effects, are not considered by a matched filter. Therefore, a lot of research was done to use machine learning to classify signals. The machine learning algorithms can be generalized into two classes: likelihood based and feature based. Likelihood based methods are based on a pre-defined likelihood function of the received signal. A decision on the type of modulation scheme is reached by performing a likelihood ratio test. [4] Although this method is optimized from a Bayesian point of view, i.e. it can minimize the probability of false classification, it has a very high computational complexity, hence increases the latency of classification results [4]. Feature based methods use expert features and reaches decisions based on observed values. Such features could include normalized signal amplitudes, phase, frequencies, variance of zero-crossing intervals etc [4]. With said features, we could run traditional machine learning algorithms, such as Support Vector Machines (SVM), to detect the modulation schemes. However, expert features that require data collection over long periods of time are hard to obtain due to the short time nature of the collected signals. [1] CNNs have been recently investigated to perform the task of modulation classification. One major advantage of CNN is that it has network designed to handle non-linearities, which makes it outperform other

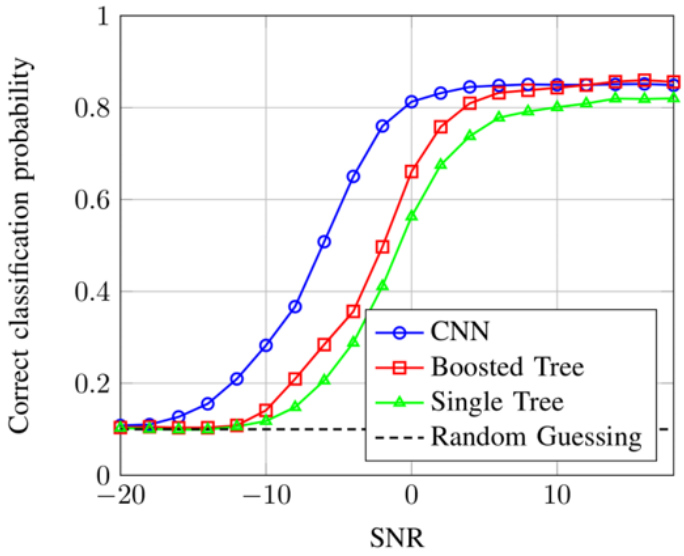

Figure 1: Accuracy of different modulation classifiers at different SNR [1]

algorithms (see Figure 1).

2.2 Neural Networks

Neural networks are a class of machine learning algorithms that are modeled after the development of neurons. A unique neural network architecture is formed the composition of many different functions, or layers, together. A neural network that takes an input x with depth n is described by equation 3, where f_k denotes the k^{th} function or the k^{th} layer of the network.

$$f(x) = f_n(f_{n-1}(\cdots f_1(x)\cdots)) \tag{3}$$

A neural network operates in two modes: training and inference. At the beginning of training, the weights in each layer are initialized to random values. As we feed in training samples, we aim to minimize the cost function of the neural network, which in our case is the cross entropy function (see equation 4). In equation 4, y is the actual probability of the observation o being in class c, while p is the probability predicted by the neural network. During inference, we feed in an observation that has the same dimension as that of the training samples, and the output of the neural network is a vector of probabilities where the

i^{th} element represents the probability of the observation being from class i. The class with the highest probability is chosen to be the class that the observation belongs to.

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c}) \tag{4}$$

In the modulation classification problem, the input to the neural network is a time sequence with 128 IQ samples captured over the air. The range of $f(x)$ is the set of integers $[0, 9]$, which are associated with different modulation classes. More implementation details of our neural network are discussed further in section 3. Our neural network uses the following five functions:

2.2.1 Rectified Linear Unit

The equation of the rectified linear unit (ReLU) function is shown in equation 5

$$f(x) = \max(0, x) \tag{5}$$

ReLUs are useful in introducing non-linearity into the model. Since the raw data samples collected by a radio have undergone non-linear transformations in the channel, we cannot necessarily rely on linear models to recover the modulation scheme. This practice gives the model higher representation power and is common throughout all neural networks. The ReLU has recently come into popularity because it is extremely simple to calculate the function (which is piece-wise linear) and the derivative (which is piece-wise constant).

2.2.2 Fully Connected Layer

A fully connected layer takes an $m \times 1$ input vector and outputs an $n \times 1$ vector through matrix multiplication, as shown in equation 6, where W is an $m \times n$ matrix and b is an $n \times 1$ constant term. Note that all the input values and the weights of the layer are all pairwise "connected" by a weight in the matrix.

$$f(x) = W^T x + b \tag{6}$$

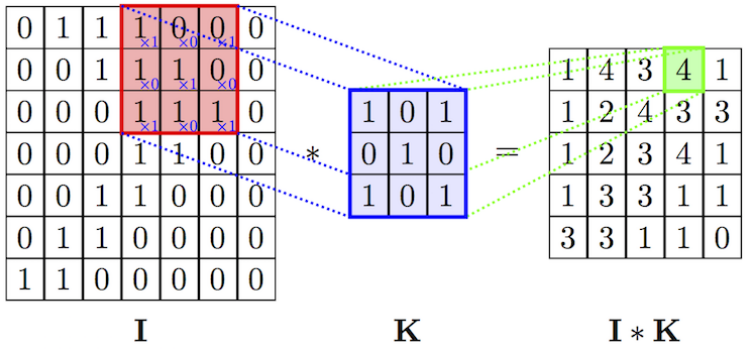

Figure 2: Simple 2D convolutional operation [5]

2.2.3 Convolutional Layer

A convolutional layer uses convolution to generate another mapping of its inputs. Figure 2 shows a simple 2D convolution operation. Within each convolutional layer is a series of filters K. Each filter convolves with its input I, 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 target input, we can apply this filter as a feature detector in other parts of the inputs to identify the existence of common features. A neural network with one or more convolutional layers is called a Convolutional Neural Network (CNN).

2.2.4 Maximum Pooling

Many features can be obtained through convolution, yet to process all of these features at every step of the neural network is computationally expensive. In this regard, down sampling along each feature map is useful. This down sampling operation is called pooling in the neural network community. When down sampling, we take a window of predefined size on the feature map and extract the maximum number from the window, hence the name max pooling. We move the window by a given stride of predefined distance between two pooling operations, hence down sampling the feature maps.

2.2.5 Softmax

At the end of many classifying neural networks, the softmax function (equation 7) is applied. The output of the softmax function is always guaranteed to sum to 1 thus giving the inter-

pretation of a probability to the output. The softmax also acts as a regularizing term during training, ensuring that the gradient of the output does not grow too large and thus cause instability. However it should be noted that despite its simple formula, training software often has a built in softmax function since it is inherent numerically unstable.

$$f(x)_i = \frac{\exp(x_i)}{\sum_i exp(x_i)} \tag{7}$$

The index which achieves the largest value is predicted to be the class that the data originated from. Note that since the softmax function is monotonically increasing, it does not change the outcome of the prediction, and thus is not required during inference.

2.3 Hyperparameters

Hyperparameters describe features of an architecture which are not changed during training. For instance, while we may specify that one of the function in our neural network is a matrix multiply, the size of that matrix is generally fixed throughout the training process. The weights inside the matrix change via gradient descent (discussed in section 2.4) while the number of weights remains constant. In a convolutional layer, the kernel size and number of output filters are both fixed values and thus hyperparameters. It is usually very difficult to find optimal hyperparameters with any intelligent method, and thus a simple brute force search is often used. This search, often referred to as *grid search* trains several models with several different hyperparameters and picks the one which performs best.

2.4 Training

Because neural networks are not convex, obtaining an actual global minimum for neural network loss is a very hard problem. In general, unless it is a very simple network, researchers do not even attempt to solve this. Instead, almost all neural networks are trained by a method called *Gradient Descent*. Gradient descent applies the following update equation at every time step:

$$w_{t+1} \leftarrow w_t - \eta \nabla_w L(f_w, x_t, y_t) \tag{8}$$

To clarify, this equation says that you subtract some constant η (called the learning rate) times the gradient of the loss function L with respect to the weights. Since the gradient points in the direction of maximum increase, this algorithm tends to push the weights in the direction of maximum decrease of loss. This algorithm is very inexact but it turns out that it performs very well over a large amount of different neural net architectures. The loss function L is usually some continuous function which represents how well the output of the neural net, f_w matched the expected output given the weights that were already attached to it.

2.5 Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) have been widely used in a large variety of applications including software defined radio, machine learning, and ASIC prototyping. They are widely used because of their low cost (compared to an ASIC), low power (compared to a GPU), and high speed (compared to a CPU). Essentially, FPGAs offer the advantage of customized hardware without the restriction of a static configuration. This allows for a very important advantage in rapidly evolving fields like machine learning and artificial intelligence. The following discussion pertains primarily to the Zynq and the Zedboard, which is the FPGA chip and prototype board that we used to implement our neural network.

The Zynq series is the newest FPGA chip series offered by Xilinx. Unlike traditional FPGA chips, which only has FPGA fabric in it, the Zynq chip also has a built-in ARM Core and DDR3 memory controller for a smoother development process. The chip can be programmed with VHDL, a programming language used to describe digital circuits in FPGAs. Xilinx offers a proprietary software tool called Vivado to program the Zynq chip. Vivado can manage the whole FPGA development process and provide an environment for VHDL debugging and simulations. After the developer writes the VHDL code, it needs to go through three more processes to flash the FPGA fabric:

1. Synthesis: The VHDL code is synthesized to a netlist of basic components in the FPGA. After this step, Vivado reports an approximation of resource utilization.

2. Implementation: The netlist is mapped to the available components on the FPGA

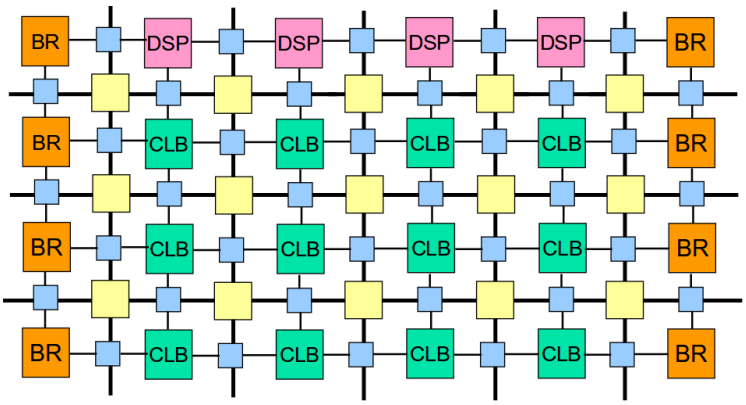

Figure 3: FPGA fabric layout, with Block RAM (BR), Configurable Logic Blocks (CLBs) and DSP Slices connected by routing channels (depicted as blue and yellow boxes)

fabric. Vivado runs its propriety optimization algorithms to reduce the amount of resource used and attempt to meet the system's timing constraints. After this step, Vivado reports timing violations (such as setup time and hold time violations), power consumption and resource utilization.

3. Bitstream Generation: The mapping is generated to a bitstream, which is used to program the FPGA fabric.

In the following sections, we are going to discuss multiple components that make up the FPGA fabric shown in figure 3.

2.5.1 Configurable Logic Blocks

Configurable logic blocks (CLBs) are what make FPGAs programmable. They allow a user to define functions and efficiently implement common logic like memory, shift registers, and muxes. Each CLB contains one look up table (LUT), three muxes, and two D flip-flops.

The majority of logic implemented by an FPGA user will likely be implemented by LUTS. A single LUT is capable of implementing arbitrary functions from 6 boolean variables to 1, two function from the same 5 boolean variables to 2, or two functions from 3 inputs to 1 and 2 inputs to 1 (note that this is just a special case of mapping 5 input variables to 2 outputs). LUTS are implemented using combinations of muxes, where the address pin is driven by a static signal determined automatically when programming the FPGA.

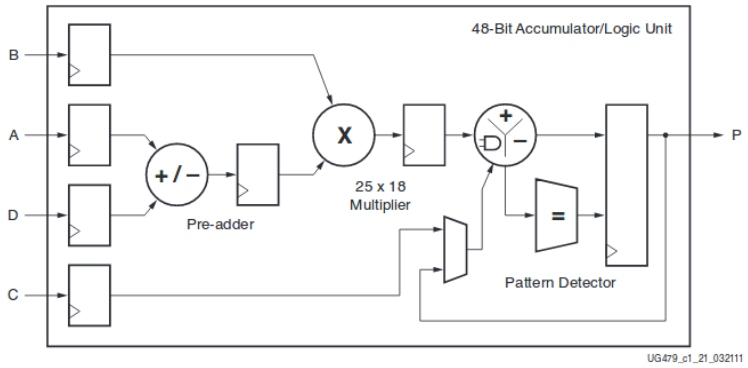

Figure 4: The Xilinx DSP slice.

Besides function generation, LUTs may also be used in combination with D flip-flops in order to create state dependent logic like shift registers and queues. LUTs can also be used to implement "distributed RAM", or LUTRAM, which has the advantage of being close to the logic and therefore very fast and low power.

2.5.2 Block RAM

Block RAM (BRAM) is another type of static memory that makes up the majority of RAM on an FPGA. Unlike LUTRAM, it is dedicated to the purpose of storing precalculated values like neural net coefficients as well as intermediate results. One BRAM slice is capable of storing up to 36kb. All reading and writing must be performed synchronously, with optional pipeline registers at both ends. Each block RAM slices also has additional dedicated hardware for First-In-First-Out (FIFO) buffer logic and error correction.

2.5.3 Digital Signal Processing Slice

The Digital signal processing slice (DSP) is a component specialized for digital signal processing applications, in particular it performs very efficient multiply accumulate operations. The DSP is capable of 25×18 bit multiplication and accumulation of up to 48 bits. All operations are performed in fixed point. The slice may also perform one of 16 basic logic functions such as X XOR Y, X AND Y, X OR Y, etc. A simplified block diagram is shown in figure 4.

3 Software Design

3.1 Training Dataset

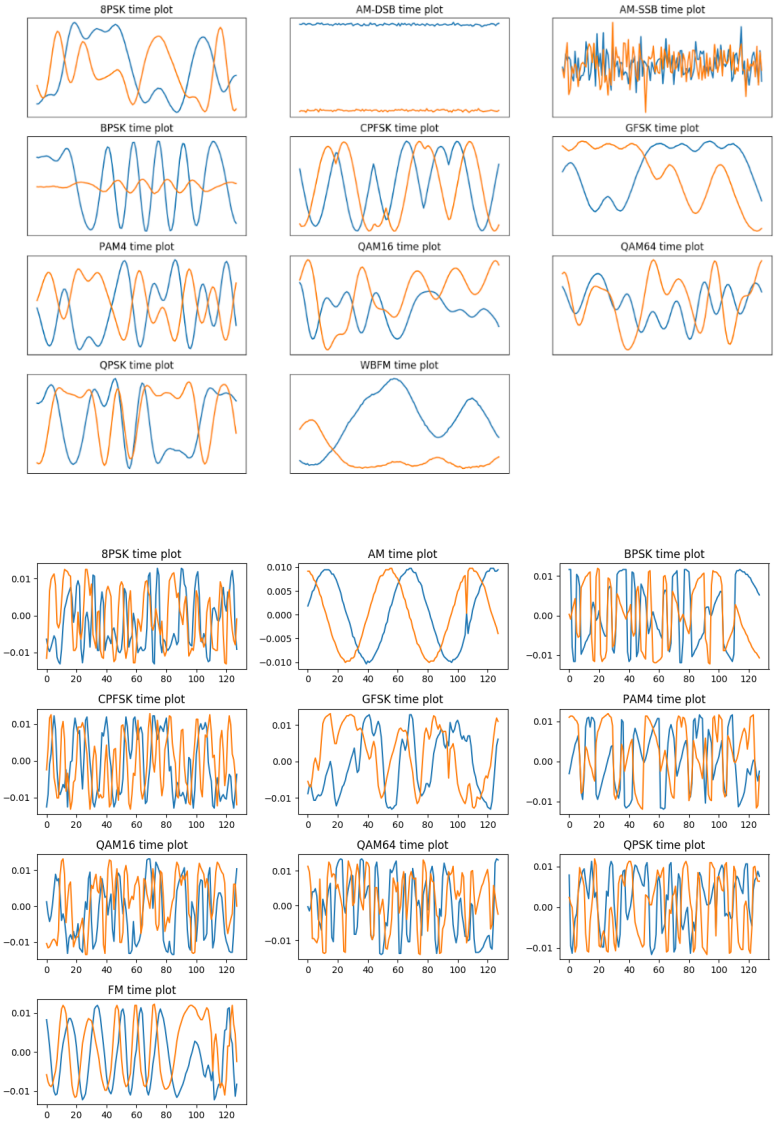

Figure 5: Signals generated by different modulation schemes in radioML (top) and real life (bottom). The blue and orange lines represents I and Q respectively.

In order to train a CNN, we need a large amount of data. A dataset is publicly available from radioML - an organization that provides simulated radio transmission data. [6] There are 220k transmission sequences in total; each transmission sequence is 128 samples long

with in-phase and quadrature (IQ) samples (see figure 5). The SNR of these signals ranges from $-20dB$ to $18dB$. The data provided encompasses signals from 8 digital modulation schemes and 2 analog modulation schemes, including BPSK, 8PSK, CPFSK, GFSK, PAM4, QAM16, QAM64, QPSK, AM-SSB and WBFM. The simulated signals are generated by the dynamic channel API in GNU Radio, which takes into account of sampling rate offset, sampling rate offset, selecting fading and noise [7].

Although radioML is a useful data set for training a CNN, it may not be sufficient to use it to test the CNN because the environmental effects are not fully simulated. Therefore, we also collected real-life wireless data with our test setup (detailed in section 5). In the real-life data, there are 60k transmission sequences (6000 per modulation scheme), each consist of 128 IQ samples. For both synthetic and real data, we used 90% for training and 10% for testing.

3.2 TensorFlow

We used TensorFlow [8], an open-source deep learning framework, to reconstruct and train the neural network with the synthetic and real data set. One of the challenges that we faced when implementing the neural network was the lack of details in the paper. [1] 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, we consulted one of his earlier works, where he made the source code openly available to the public. [3] However, his earlier works used another neural network framework, Caffe. Based on the Caffe implementation we made the following decisions regarding our neural network:

1. The objective function used to optimize the model is the Adam Optimizer [9] with a learning rate of 0.01.

2. Each training sample is used 100 times during training. For the real life data, each training sample is used 200 times during training.

3. Fully connected layers in the Caffe implementation do 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 present. Therefore we disabled the bias term.

4. The convolution is unpadded, i.e. the first convolution takes the first n samples from the input, instead of taking the first input and padding with zeros.

As mentioned in section 2.5, the FPGA hardware is specially designed to handle fixed point additions and multiplications. TensorFlow, by default, uses single precision floating point values in all operations. Because training requires making small adjustments to weights, the TensorFlow documentation recommends training in floating point and then converting the graph to a fixed point representation. In order to convert our graph to a fixed point format, we built the TensorFlow *quantize_graph* tool from source, froze the TensorFlow graph representing our neural network, and applied quantization to it. The resulting neural network achieved an accuracy similar to the original, as shown in figure 6. Unfortunately, it is not straightforward to translate this into our hardware implementation of the neural network and so this process only served as a proof that we could implement a quantized neural network without significant loss of accuracy. We discuss our implementation of quantization in section 3.4. The original neural network architecture is shown in Table 2.

Layers	Output Dimension
Input	2×128
Convolution(128 filters, size 2×8) + ReLU	128×121
Max Pooling (size 2, strides 2)	128×60
Convolution(64 filters, size 1×16)[1]+ ReLU	64×45
Max Pooling (size 2, strides 2)	64×22
Flatten	1408
Fully Connected + ReLU	128
Fully Connected + ReLU	64
Fully Connected + ReLU	32
Fully Connected + softmax	10

Table 2: Original CNN Modulation Classifier Architecture Layout in [1]

The neural network is simply a repeated composition of all of the neural network components that we discussed in section 2.2, except for the flatten operation. A fully connected

[1]The 64 filters in the convolutional operations are applied across 128 rows from the previous layer, hence the size of the filter is effectively 128×16

layer must accept a vector as input, but the output of a convolutional stage in a matrix. Therefore we require the flatten operation after convolution which simply takes the matrix and turns it into a vector via linear indexing, or vectorization. That is:

$$flatten(A) = [a_{1,1}, \ldots, a_{m,1}, a_{1,2}, \ldots, a_{m,2}, \ldots, a_{1,n}, \ldots, a_{m,n}] \tag{9}$$

3.3 Pruning Experiments

During our implementation of the neural network on the FPGA, we found that we were dangerously close to the resource limit, even before implementing the fully connected layer at all. As a result, we performed a very simple kind of pruning on the architecture. We trained 39 different neural networks on the same data varying the size and shape of the architecture. This can be seen as a form of grid search, discussed in section 2.3. The values in Table 3 represent the average accuracy achieved by the given hyperparameters. We see that even after substantial reductions in complexity, there is hardly any effect on the average accuracy of the neural network. This discovery allowed us to produce a much smaller architecture which achieves even better performance.

3.4 Quantization Techniques

Most modern computers uses IEEE floating point format to store decimal numbers. It is coined "floating" since the location of the decimal points floats around the number depending on the value of the number. In a 32-bit computer, there are 1 signed bit (s), 8 exponent bits (exp) and 23 mantissa bits (m), and the floating point number (fp_num) is computed by equation 10

$$fp\_num = (-1)^s * 2^{exp} * m \tag{10}$$

$$fixed\_num = \sum_{i=-m}^{n-m} 2^{-n} = 1 \tag{11}$$

An alternative method in storing decimal numbers is the fixed point format, as shown

4 Fully Connected Layers			**3 Fully Connected Layers**		
$1C, 1F$	$\frac{1}{2}C, 1F$	$\frac{1}{4}C, 1F$	$1C, 1F$	$\frac{1}{2}C, 1F$	$\frac{1}{4}C, 1F$
$1C, \frac{1}{2}F$	$\frac{1}{2}C, \frac{1}{2}F$	$\frac{1}{4}C, \frac{1}{2}F$	$1C, \frac{1}{2}F$	$\frac{1}{2}C, \frac{1}{2}F$	$\frac{1}{4}C, \frac{1}{2}F$
$1C, \frac{1}{4}F$	$\frac{1}{2}C, \frac{1}{4}F$	$\frac{1}{4}C, \frac{1}{4}F$	$1C, \frac{1}{4}F$	$\frac{1}{2}C, \frac{1}{4}F$	$\frac{1}{4}C, \frac{1}{4}F$
$1C, \frac{1}{8}F$	$\frac{1}{2}C, \frac{1}{8}F$	$\frac{1}{4}C, \frac{1}{8}F$	$1C, \frac{1}{8}F$	$\frac{1}{2}C, \frac{1}{8}F$	$\frac{1}{4}C, \frac{1}{8}F$
2 Fully Connected Layers			**1 Fully Connected Layer**		
$1C, 1F$	$\frac{1}{2}C, 1F$	$\frac{1}{4}C, 1F$	$1C, 1F$	$\frac{1}{2}C, 1F$	$\frac{1}{4}C, 1F$
$1C, \frac{1}{2}F$	$\frac{1}{2}C, \frac{1}{2}F$	$\frac{1}{4}C, \frac{1}{2}F$			
$1C, \frac{1}{4}F$	$\frac{1}{2}C, \frac{1}{4}F$	$\frac{1}{4}C, \frac{1}{4}F$			
$1C, \frac{1}{8}F$	$\frac{1}{2}C, \frac{1}{8}F$	$\frac{1}{4}C, \frac{1}{8}F$			

4 Fully Connected Layers			**3 Fully Connected Layers**		
0.569	0.574	0.579	0.569	0.574	0.574
0.574	0.575	0.575	0.572	0.573	0.579
0.569	0.574	0.565	0.571	0.575	0.566
0.562	0.564	0.556	0.567	0.567	0.562
2 Fully Connected Layers			**1 Fully Connected Layer**		
0.581	0.572	0.574	0.562	0.564	0.557
0.571	0.570	0.571			
0.571	0.571	0.564			
0.562	0.564	0.549			

Table 3: Top: Hyperparameters over which to search. The symbol xC corresponds to decreasing the size of the convolutional layers by a factor of x while the symbol yF corresponds to decreasing the size of the fully connected layers by a factor of y. Each sub-table corresponds to a different number of fully connected layers. Bottom: The resulting average accuracy corresponding to the hyperparameters described above

in equation 11. It is denoted by $Q(n.m)$, where $n + m$ represent the number of bits of the number, and m represents the number of bits after the fixed point (fractional bits). Although floating point numbers provide larger range for storing decimal numbers than fixed point numbers, they require complex processing. Since fixed point operations are faster and requires less computation, it is better suited for implementation on an FPGA. In our implementation, we perform our operations as 25-bit fixed point numbers.

$$\begin{cases} max = slope \times 127 + bias \\ min = slope \times -128 + bias \end{cases} \tag{12}$$

$$\begin{aligned} slope &= \tfrac{max-min}{255} \\ bias &= max - 127 \times \tfrac{max-min}{255} \end{aligned} \tag{13}$$

Layers	Output Dimension
Input	2×128
Convolution(32 filters, size 2×8) + ReLU	32×121
Max Pooling (size 2, strides 2)	32×60
Convolution(16 filters, size 1×16) + ReLU	16×45
Max Pooling (size 2, strides 2)	16×22
Flatten	352
Fully Connected + ReLU	64
Fully Connected + ReLU	32
Fully Connected + softmax	10

Table 4: New CNN Modulation Classifier Architecture Layout

There are two main issues in using 25-bit fixed point numbers in our design. First, there is not enough space in memory to store all the model weights as 25-bit numbers. Second, fixed point number provides a smaller range than that of floating point number. Figure **??** shows that the weights have a Gaussian Distribution centered at zero. There is a very narrow range of weight values per layer. To address these issues, we store all the weights in Slope-Bias format[10]. Every layer in the neural network has a specific slope and bias constant. To find these constants, we need to find the maximum and minimum values of weights per layer and form the system of equations in 12. Solving them gives the equations 13 to calculate slopes and biases.

3.5 Model Performance Comparison

Our final architecture, as shown in Table 4, is as accurate as the original model at 10% of its size. When compared to the original model in [1], our CNN has one less fully connected layer and the sizes of the convolutional layers and fully connected layers are reduced by a factor of 2 and 4 respectively. There are 30k weights in this new model, which is 10% of the 330k weights in [1]. Figure 6 shows that our algorithm is as accurate as the model in [1] using full precision. Implementation on an FPGA is best suited for fixed point operations since they are particularly fast and power efficient. When quantized with slope-bias encoding, our model suffers a small loss but maintains comparable accuracy.

However, we need to retrain the model with real life data to classify real life signals. From the confusion matrices in figure 7, we can see that the model trained with synthetic data

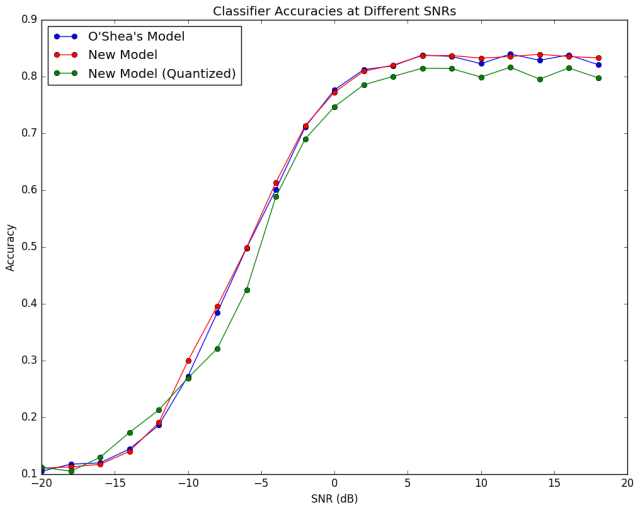

Figure 6: Classification Accuracy of Original and New Models

cannot classify the modulation scheme in real life, and vice versa. This phenomena indicates that simulated data are not sufficient in training a classifier in real life, and perhaps the model needs to be retrained in different environments. This can be an interesting topic in future research.

4 Hardware Design

4.1 Design Consideration

In our hardware design, we aim to minimize logic and memory usage while maintaining low latency and low power. As shown in table 5, there are limited resources available on our FPGA. Each BRAM block has 36 kb, totalling up to 4.9 Mb for 30k weights. Moreover, one multiplication operation requires one DSP slice, implying that there can only be 220 multiplications occurring at the same time. Yet our CNN requires many multipliers for convolution and matrix multiplication. Therefore, our first design goal is to minimize our resource usage. Our second design goal is to perform operations in parallel to minimize latency.

Figure 8 shows that our current model can be broken down into three parts design-wise:

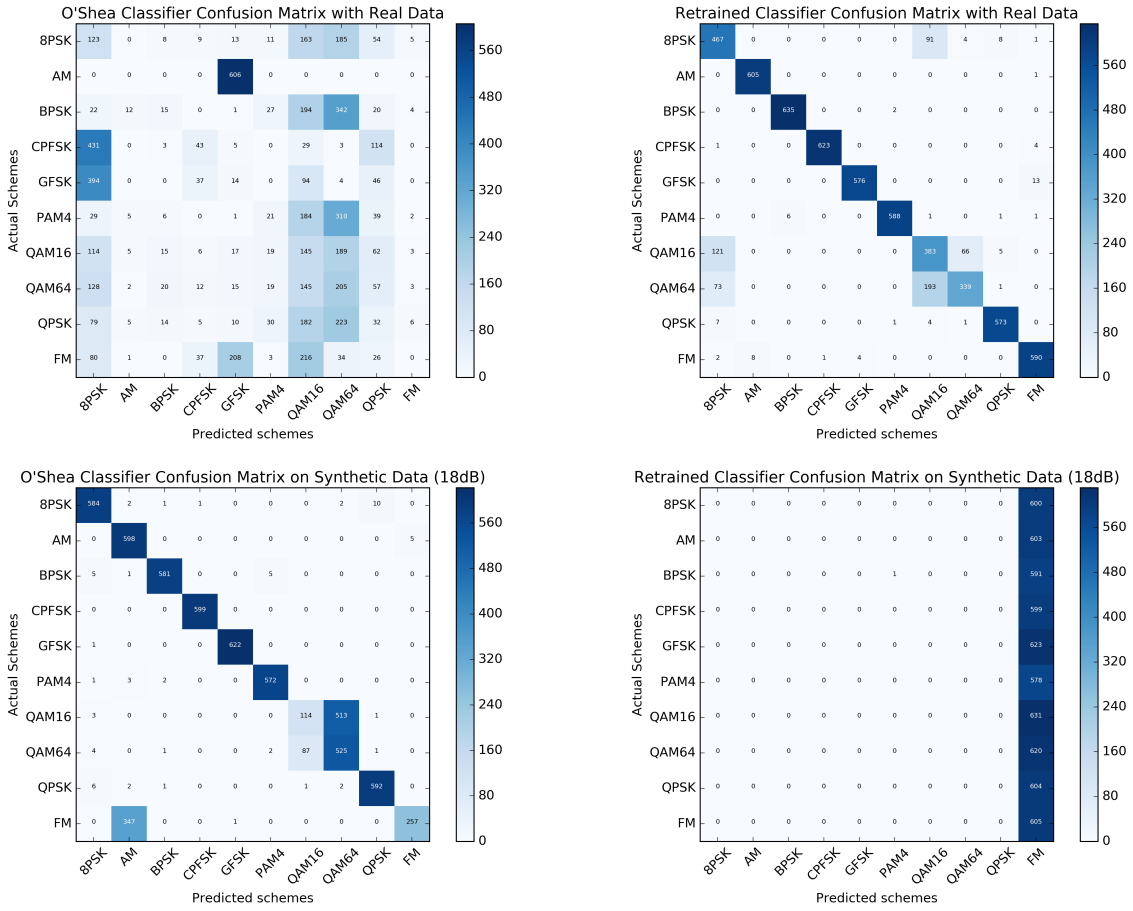

Figure 7: Confusion matrix of original (left) and retrained (right) classifiers when classifying synthetic data (bottom) and real data (top)

first convolutional layer, second convolutional layer and fully conencted layers. From that, we made two observations and design decisions. First, we designed modular units for simple and repeating operations. There are five ReLU layers and two maxpool layers. Moreover, the weights of convolutional layers and fully connected layers are stored in slope bias formats. Therefore, we need to design these layers are simple and generic as possible so that we can apply them across different layers. Second, we designed specialized units for computationally expensive units, such as convolutional and fully connected layers.

The three parts are designed separately and interconnected with circular First-In-First-

Resource	LUT	LUTRAM	FF	BRAM	DSP Slices
Amount	53200	17400	106400	140	220

Table 5: Resources available on Zedboard [11]

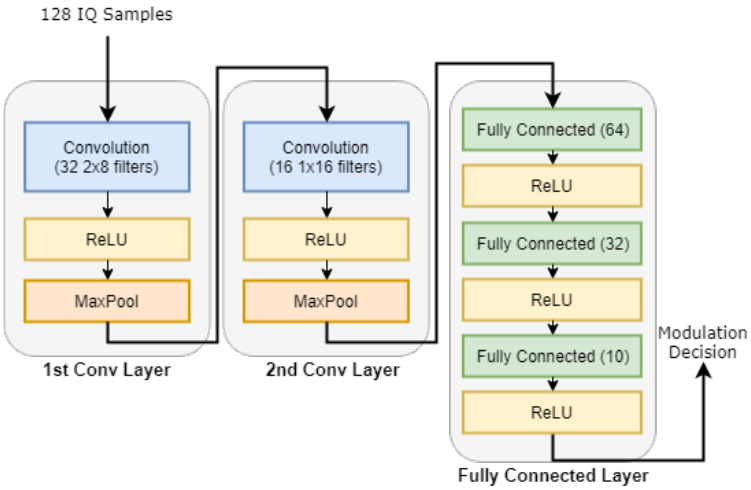

Figure 8: Neural Network architecture in Table 4 organized by design patterns.

Out (FIFO) buffers. Xilinx offers FIFO buffers as proprietary cores and we added modifications to the design to make it circular (see Figure 9). They are used to store all intermediate values between layers and within layers. Moreover, since the convolutional and fully connected operations are Multiple Instruction Single Data (MISD) in nature, the FIFOs need to be circular, i.e. all values that are read needs to be rewritten into the FIFO for another set of convolution.

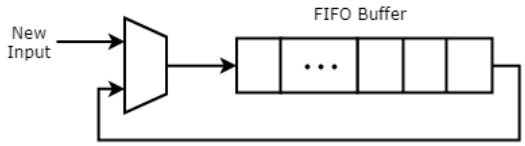

Figure 9: Block diagram of a circular FIFO

4.2 Modular Design

4.2.1 Slope Bias Loader

We reduced the memory requirements for weight storage by quantizing them to eight bits and storing them in a slope-bias format. An N-bit signed integer with bits $b = \{b_i\}$ has the real value $n(b) = \sum_{i=0}^{N-2} b_i 2^i - b_{N-1} 2^{N-1}$. The real value of the slope-bias representation of b is $q(b) = n(b) \times S + B$, where S is the slope and B is the bias.

Because slope-bias multiplication is fairly complicated and costly, we chose to perform

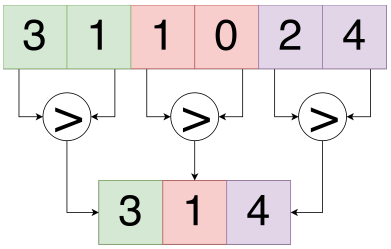

Figure 10: Max Pooling Operation

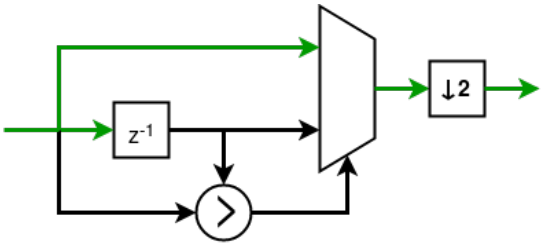

Figure 11: Max Pooling Hardware Design

a one-time conversion from 8-bit weights in slope-bias format to a higher precision where all arithmetic operations took place. The slope is stored in Qn.m format as a multiplierless multiplier and the bias is stored in full precision. Our slope-bias loader performs the following operations to achieve the conversion to full precision, requiring on average 60 LUTs.

4.2.2 Rectified Linear Unit (ReLU)

ReLUs are implemented with multiplexers (mux) throughout our design. Since all numbers are implemented in two's complement in our design, the most significant bit (MSB) acts as the address bit of the mux.

The ReLU is defined to be : $f(x) = max(0, x)$ and is useful in introducing non-linearities into the model. They are functionally equivalent to muxes selecting between zero and their input. Since all processing is done in two's complement, the most significant bit (MSB) act as the address bit of the mux.

4.2.3 Maximum Pooling (Max Pooling)

Maximum Pooling, or max pooling, allows us to down sample along each layer of operation. To max pool an input with a size n window with stride m, we first slide an $n \times 1$ window across the input and take the maximum element as the output, and then downsample the output by a factor of m. Figure 10 shows a 1D max pooling operation with stride 2 and window size 2. Note that for a 2D input, we apply max pooling on each **row** of the input.

The hardware representation of max pooling in our design is shown in Figure 11. The first stage is accomplished by a multiplexer. Given an input x at time t, we take the difference between the $x[t]$ and $x[t-1]$, and use the MSB of the difference to multiplex between $x[t]$

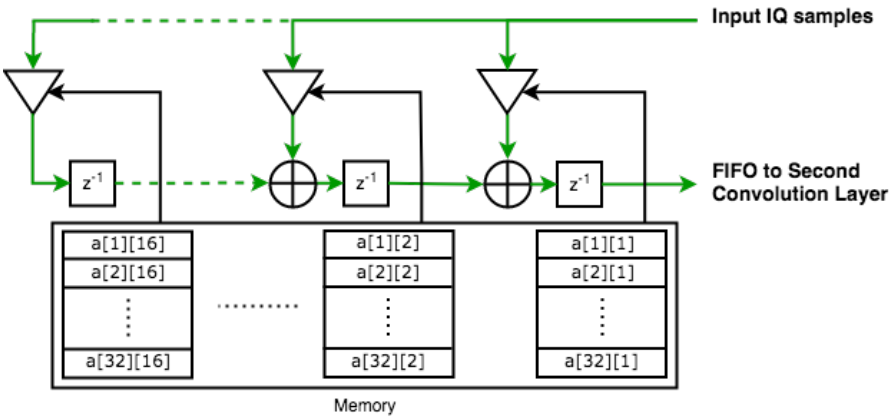

Figure 12: Block Diagram of TDM FIR filter for Conv1 Convolutional Layer

and $x[t-1]$. The second stage is then accomplished by down sampling the mux output by a factor of 2. Although this implementation results in half the computations being wasted, the alternative would be to design a complex machine that requires more logic and therefore more resources.

4.3 Specialized Design

4.3.1 First Convolutional Layer (Conv1 Layer)

For the first convolutional stage, we pursued an architecture by sharing weights between multiple filters. The first convolutional stage is functionally equivalent to 64 8-tap FIR filters that needs to filter the same 128 IQ samples. The implementation of filters in FPGA fabric is usually accomplished by assigning one DSP slice to each coefficient in each filter. However given that each filter in the first stage consists of 8 taps, this would require $64 \times 8 = 512$ DSP slices, which is far beyond the ZedBoard's capacity of 220 DSP slices.

Another option for achieving fully dedicated hardware is multiplierless multiplication. Using this method, multiplication by fixed constants (like coefficients in a neural network) can be achieved without using DSP slices, and instead using a series of bit shifts and LUT adders. This method can be improved further by representing coefficients using canonical signed digit encoding which is a ternary number system consisting of the symbol set {1,0,-1}. Here, a positive number represents addition, negative subtraction, and 0 do nothing. The location of the symbol in the word indicates how many bit shifts to apply before performing

the operation. Using this method of implementation, we were able to create filters using on the order of 50-100 LUTs each. Unfortunately, even with the conservative approximation that each filter would require 50 LUTs, this leads to a total usage of 102,400 LUTs which is far beyond our limitation of 53,200.

To ensure that this stage did not use too many DSP slices or LUTs for the filters, we finally adopted a design similar to a time division multiplexed (TDM) FIR filters (see figure 12), where multiple sets of weights share the same physical filter. Xilinx's Vivado IP core contains an FIR filter module which is capable of implementing many different forms of FIR filters. In particular, the tool supports allows for the structure of a retimed (transposed) FIR filter which is optimized for reducing critical path. There is also built in support for reloading filter coefficients which is exactly what we require. We implemented two filters each using eight DSP slices for the in-phase and quadrature filters.

We implemented a state machine to control the data flow and switching time of filter coefficients. We use a circular FIFO to broadcast the input signal to the filter as well as itself until the data are processed through all 32 filters. Note that the neural network architecture employs "valid" padding, meaning that the first 7 samples are not used as the filter is not "full" yet. In order to account for this, the circular buffer was controlled by a simple state machine which counted to determine when input and output were valid. When the output is valid, a simple LUT adder adds the results from the two filters. This result then passes through the ReLU and maxpool operations, and is written into the interconnecting FIFO between Conv1 and Conv2.

4.3.2 Second Convolutional Layer (Conv2 Layer)

The second convolutional stage is similar to the first but on a much larger scale. The second convolutional functionally requires $32 \times 16 = 512$ 16-tap FIR filters, which greatly exceeds what the ZedBoard can provide. Therefore, we decided to complete 16 convolutions in parallel, but only computing one multiplication at a time. Each multiplication requires two DSP slices because of bit growth, thus using 32 DSP slices at the price of 16x slowdown. The final design of the second convolutional stage is shown in figure 13.

We also designed a complex central controller to control and monitor the actions of

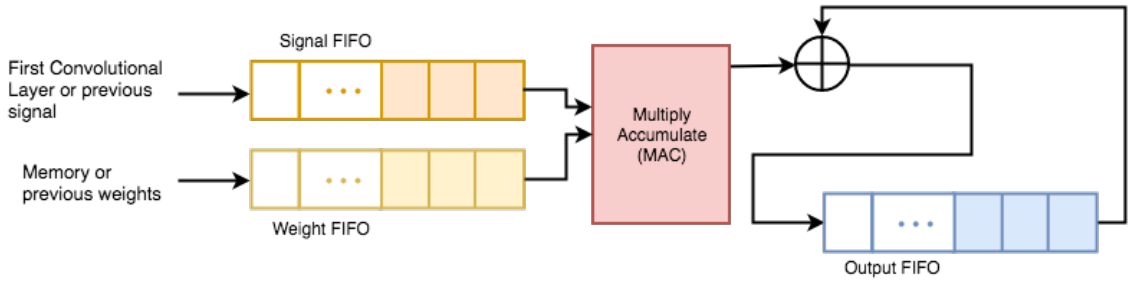

Figure 13: Block Diagram of Conv2 Convolutional Layer

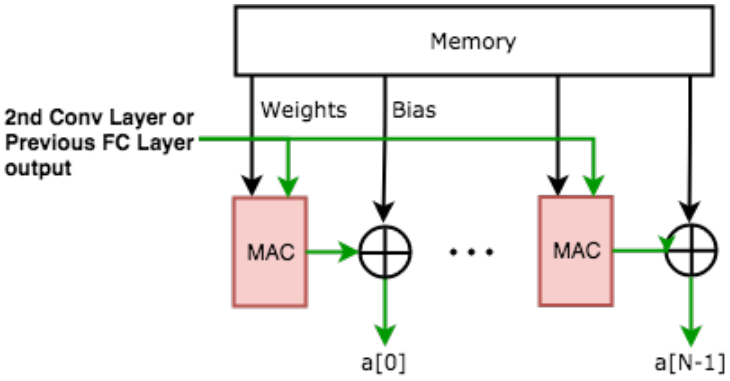

Figure 14: Block Diagram of FC Layer

the MACs, memory and buffers. Once the interconnecting FIFO between first and second convolutional layer is filled up, the controller enters initialization mode, where the first 16 values of the samples are loaded from the input FIFO to signal FIFO, and the 16 weights of the first filter are loaded to a weight FIFO. As the machine enters accumulate mode, the values in the two FIFOs are multiplied and accumulated. Afterwards, the output of the MAC is written to the output FIFO in write mode. Now the machine enters load state and new coefficients are loaded into the filter FIFO. After 32 cycles. We enter the advance mode, where we drop the first value from the signal and add the next value from the interconnection buffer, i.e. sliding the signal in convolution. We repeat the steps until all filters have been used and the correct output has accumulated. Now the machine enters end mode to signal to the following layer that the intermediate values are ready to be processed. We use this controller to drive 16 convolutions in parallel, each using a two DSP slices.

4.3.3 Fully Connected Layers (FC Layer)

The architecture for the last three fully connected layers is shown in figure 14. The weights and biases are stored in memory. Since there are limited DSP slices, we took an output stationary approach [12] so that the output of each DSP slice corresponds to an element of the output array directly. We take the output of the previous stage (flattening the 2D matrix to a 1D vector if needed) and broadcast it to 64 MACs, where it is multiplied with the corresponding weights. The number of MACs used per stage depends on the number of output elements. For instance, the first fully connected layer uses 64 MACs (since its output has 64 elements), and each MAC performs 352 MAC operations. Afterwards, the accumulated value is added to the bias and stored to an output FIFO, which is used for the next fully connected operation.

We successfully shrank three fully connected layer design into one functional unit. There are three fully connected layers in total, and with the aforementioned design, it would have taken 106 MAC units. However, since there are not enough DSP slices for an extra 106 MACs (see table 6), we share the MAC units between fully connected layers. To perform the next fully connected layer operation, we take the values stored in the output FIFO and repeat the operation with different weights, and fewer MACs.

4.4 Timing Considerations

One of the key differences between programming on an FPGA and programming in software is that the programmer has direct access to hardware level mappings. This low level access allows the user to obtain very high efficiency, however it also allows for the possibility of timing errors. A timing error occurs when a hardware level limitation, like finite signal transmission speed, is not accounted for in the design of a device. There are four different kinds of timing errors we encountered in the testing of our design, all of which required us to fix something in our code. Timing errors are particularly difficult to debug since they usually do not appear in behavioral simulations, and thus require implementation for a post-implementation simulation to be run. This makes the process much more difficult because implementation and post-implementation simulation require about 40 to 50 minutes to run

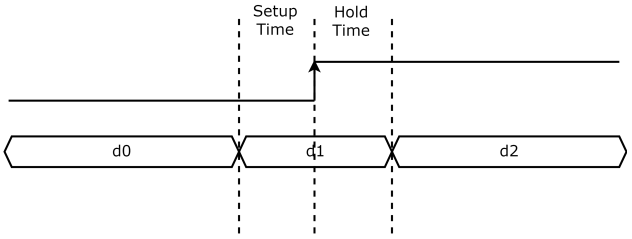

Figure 15: Depiction of Setup/Hold time for a register

to completion. This is a difficult obstacle because the cycle of changing code and testing the output is much slower.

4.4.1 Setup/Hold Time

Setup time and hold time determines how fast the system can run. The best way to illustrate setup/hold time is through the timing diagram of a flip flop in figure 15. A value is latched onto a flip flop at the rising edge of the clock. For a value to latch onto the flip flop successfully, the value needs to stay constant for minimum amounts of time before and after the clock's rising edge, which is called setup and hold time respectively. A setup/hold time violations occur if the correspond times are too short. If there are setup and hold time violations, the system could possibly become nondeterministic and difficult to debug in the long run.

There are two approaches to resolve setup/hold time violations. The first approach is reducing the master clock frequency. Slowing down the clock is a convenient solution because it gives a larger tolerance to propagation delay of data. The second approach is altering propagation delay. Adding and removing buffers in the data path experimentally can fine tune the final implemented design on the FPGA.

In our design, we decided to reduce the master clock frequency to 50MHz, which is a half of the maximum frequency. In our original design, we have a hold time violation issue on the reset pin of the neural network. This is because the reset pin branches to different portions of the whole system. The hold time violation is eliminated by reducing the master clock frequency.

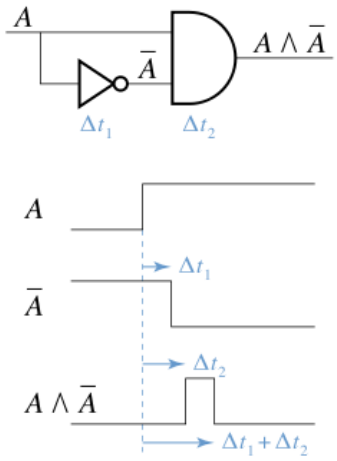

Figure 16: A race condition for very simple logic. [13]

4.4.2 Race Conditions

A race condition occurs when two inputs to a logical device, like an AND gate, follow paths of different lengths. For some short period of time, the output of the gate may be incorrect, or at least not what the designer expects. An example is shown in Figure 16. Since the output of the gate is $A\&\overline{A}$, it should always be 0. However, as the bottom input travels through an inverter, it is delayed slightly with respect to the top signal. Because of this delay when the signal goes high, there is some period of time when the upper input is high and the lower input is still high and so the output is 1.

Race conditions like the one shown above are not often issues because we use asynchronous circuits and the transient is usually cleared by the time the next clock edge comes around. However, issues can arise in feedback loops like a simple MAC. Below we show a simplified version of one of our multiply accumulators, before we fixed its data race condition.

```
entity MAC is
    PORT (  X: in std_logic_vector(7 downto 0);
            Y: in std_logic_vector(7 downto 0);
            PREV: out std_logic_vector(15 downto 0);
            RST: in std_logic);
end MAC;

architecture RTL of MAC is
begin
CARRY <= PREV when CLEAR_CARRY = '0' else (others => '0');
process (CLK) begin
    if rising_edge(CLK) then
        if RST = '1' then
            CLEAR_CARRY <= '1';
        end if;
        PREV <= std_logic_vector(signed(X)*signed(Y) + signed(CARRY));
    end if;
end process;
end RTL;
```

When the signal `RST` is high and a rising clock edge occurs, the signal `CLEAR_CARRY` goes high and the signal `PREV` is set to $X \times Y + CARRY$. The question is: what is `CARRY` at this point? In reality, because the `when` statement is achieved through MUX logic, `CARRY` gets set to zeros slightly after the clock edge, so the `PREV` signal will see the "old" `CARRY`. In behavioral simulation, `CARRY` is seen as zeros at the same time isntead. This behavior is only seen after timing analysis thus leading to a mismatch between behavioral simulations and the post-implementation functional simulation. We fixed the issue by implementing the MAC as shown below

```
...
architecture RTL of MAC is
begin
process (CLK) begin
    if rising_edge(CLK) then
        if RST = '1' then
            PREV <= (others => '0');
        else
            PREV <= std_logic_vector(signed(X)*signed(Y) + signed(PREV));
        end if;
    end if;
end process;
end RTL;
```

4.4.3 Imperfect Gated Clock

An imperfect gated clock might be seen as a special case of a data race. In this particular scenario, the data race is between the input of one clocked flip flop and another. Figure 17 shows a simple scenario of an imperfect gated clock. Because of the small delay introduced by the combinational logic from the first clock to the second results in what is essentially an asynchronous circuit. Again, this setup results in different behavior between a behavioral simulation and a post-implementation functional simulation. To illustrate this, we will again take the example of a MAC with a clock enable pin. The example VHDL code shown below is representative of our MAC before finding this timing issue:

```
...
architecture RTL of MAC is
signal GATED_CLK: std_logic;
begin
GATED_CLK <= CLK and CE;
process (GCLK) begin
    if rising_edge(GCLK) then
        if RST = '1' then
            PREV <= (others => '0');
        else
            PREV <= std_logic_vector(signed(X)*signed(Y) + signed(PREV));
        end if;
    end if;
end process;
end RTL;
```

Behavioral simulation assumes that `GATED_CLK` and `CLK` are exactly synchronous, but in reality `GATED_CLK` goes high slightly later than `CLK` and so the updates to the signals within the process occur later than we would expect and the results are unreliable. Again, we may fix the issue by making the gating synchronous, or latched as in the example below:

```
...
architecture RTL of MAC is
begin
process (CLK) begin
    if rising_edge(CLK) then
        if CE = '1' then
            if RST = '1' then
                PREV <= (others => '0');
            else
                PREV <= std_logic_vector(signed(X)*signed(Y) + signed(PREV));
            end if;
        end if;
    end if;
end process;
end RTL;
```

4.4.4 Metastability

A digital circuit is in a metastable state when some logical voltage level is neither high enough to be considered a '1' nor low enough to be considered a '0'. This can be guaranteed

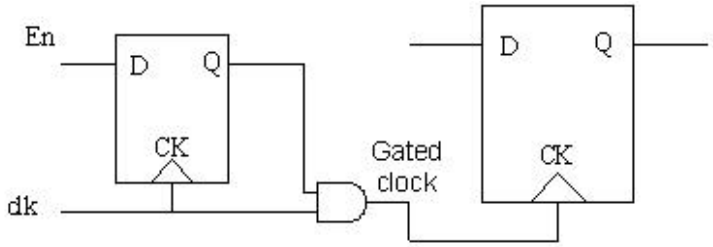

Figure 17: Latch free clock gating. [14]

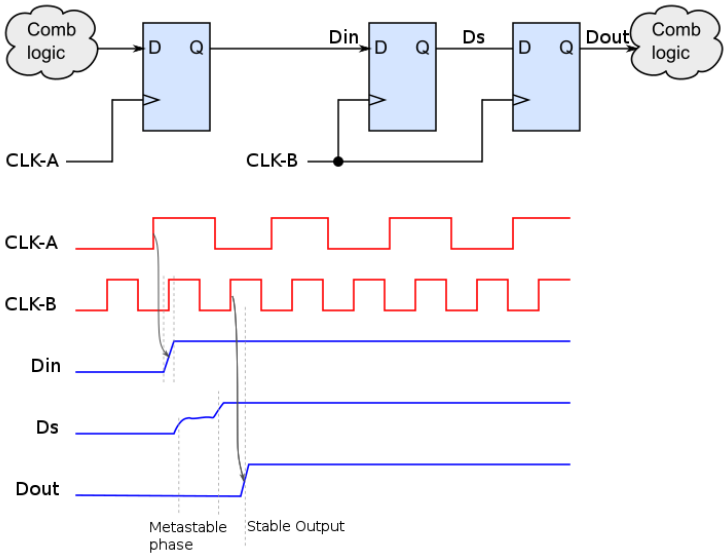

Figure 18: Crossing of clock domains introducing metastability. [15]

to not occur in a synchronous circuit given a low enough clock frequency, however no such guarantees exist for an asynchronous circuit. One of the primary causes of metastability is the crossing of clock boundaries, where the circuit is not synchronous. An example circuit is given in Figure 18

This issue occurred in two places in our design: the interface between the ARM processor and the FPGA fabric, as well as within the FPGA fabric itself in the maxpool design. Because the ARM processor was being run at 100MHz and the FPGA fabric was being run at 50MHz, we encountered metastability issues in crossing. There are hardware designs which aid in crossing clock domains, but they are quite complicated. To solve the issue quickly, we simply reduced the speed of the ARM processor to 50MHz to match the FPGA fabric. Because the majority of the processing is happening on the fabric, this did not significantly reduce our

performance.

In our original design, we downsampled by introducing a FIFO which would be clocked at half the rate of the input samples with a simple digital frequency divider. Because we want the circuit after downsampling to run at full speed, we must then drive the same FIFO with the full speed clock to repeat the signal. In behavioral simulation, this successfully led to a downsampling and repeating of the signal. However, we obtained unexpected results in post-implementation simulation. This is better understood through the VHDL code given below:

```
...
architecture RTL of DOWNSAMPLE is
signal FIFO_CLK: std_logic;
signal COND: std_logic;
begin
FIFO_CLK <= CLK when COND = '1' else HALF_CLK;
process (CLK) begin
    if rising_edge(CLK) then
        HALF_CLK <= not HALF_CLK
    end if;
end process;
end RTL;
```

Driving the FIFO with two different clocks depending on a condition led to metastability at that point and thus a difference between real implementation and behavioral simulation. In order to fix this, we instead drove the write enable pin with a clock of half the frequency, rather than actually changing the clock itself from one frequency to another durin g operation. The implementation is as follows:

Resource	Total Available	Total Used (%)	Conv1	Conv2	FC	Peripherals
LUT	53200	43560 (81.9%)	905	11189	15604	15927
LUTRAM	17400	1986 (11.4%)	154	640	49	1143
FF	106400	33881 (31.8%)	1179	6086	507	26038
BRAM	140	33 (23.6%)	1	17.5	8.5	5
DSP Slices	220	197 (89.6%)	16	48	64	69

Table 6: Post Implementation Resource usage on ZedBoard FPGA. The peripherals column include resource usage of radio module and buses that connects the neural network to the ARM Core.

```
...
architecture RTL of DOWNSAMPLE is
signal FIFO_WR_EN: std_logic;
signal COND: std_logic;
begin
FIFO_WR_EN <= CLK when COND = '1' else HALF_CLK;
process (CLK) begin
    if rising_edge(CLK) then
        HALF_CLK <= not HALF_CLK
    end if;
end process;
end RTL;
```

This implementation achieves the intended results in both behavioral simulation and post-implementation functional simulation.

4.5 Hardware Utilization and Performance

We synthesized and implemented the neural network in Vivado under satisfactory resource constraint. A radio module is also implemented with the neural network to test the modulation classifier. The layout of the implementation is shown in figure 19. Table 6 shows the total resource usage[2] of different layers. As expected, we used a large amount of DSP slices for multiplication operations in the neural network. The weights in the Conv1 are stored as 8-bit numbers while the rest are stored as 16-bit numbers, hence the BRAM used in Conv1 is much less than that of Conv2 and FC layers.

[2]The resource utilization of each CNN layer is an approximation made by Vivado. There might be extra resources use to interconnect between the layers, hence the sum of the individual utilizations is less than total utilization

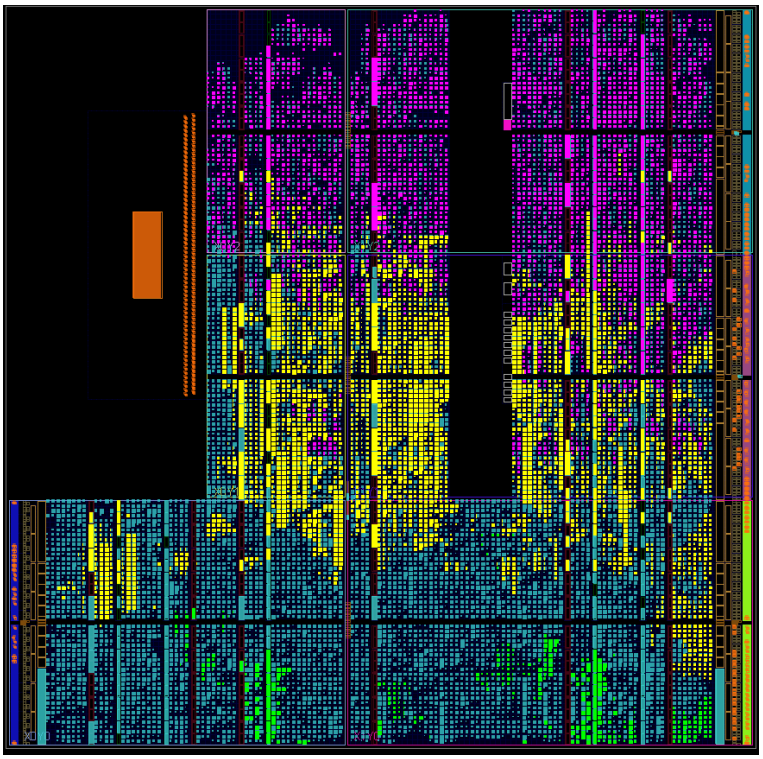

Figure 19: Layout of first convolutional layer (green), second convolutional layer (purple), fully connected layer (yellow) and radio module (Blue) on FPGA fabric with ARM Core (Orange).

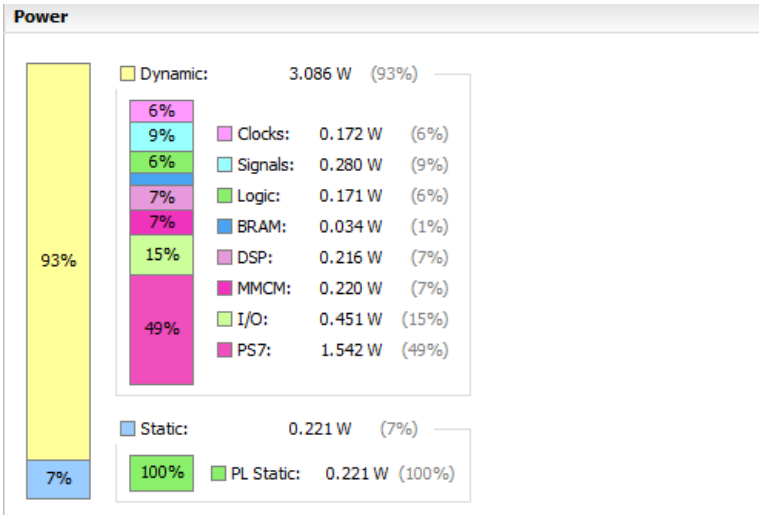

Figure 20: Power Consumption of different units on the FPGA. PL refers to the FPGA fabric, while PS refers to the processor in the zynq chip.

Model	Power Usage (Static)	Power Usage (Dynamic)	Latency
GPU (Original)	68 W	70 W	80 ms
GPU (New Model)	68 W	70 W	90 ms
FPGA (New Model)	0.221 W	3.086 W	600 us

Table 7: Power and latency of FPGA and GPU implementations. Static power refers to power consumption when the device is idle, and dynamic power refers to power consumption when the device is active.

Moreover, the FPGA uses much less power and time than GPU in modulation classification. Table 7 shows the power and latency of GPU vs FPGAs. GPU latency is the average latency per inferred sample, FPGA latency is the estimated latency from the number of clock cycles required with a 50MHz clock, and GPU power usage was measured by hardware monitoring software. Amongst all the hardware and models, the FPGA with the new model uses the least amount of power and time. The power usage report in figure 20 shows that the processor requires the most amount of power for operation. This result suggests that neural networks can be used even in settings which require low latency and low power consumption, particularly mobile applications.

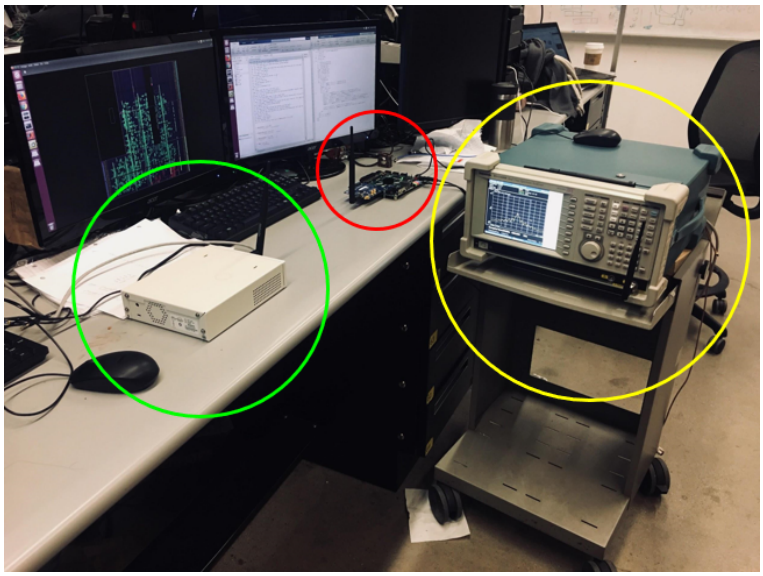

Figure 21: Physical Setup to capture data from transmitter (Green Circle) and test the Neural Network in the Receiver (Red Circle). The transmission is validated with a spectrum analyzer (Yellow Circle)

5 Implementation

In figure 21, we set up a system to test the neural network under real environmental constraints. The overall system has three components:

1. Transmitter to transmit data modulated by different modulation schemes over ISM band

2. Receiver to control radio module and neural network on the FPGA

5.1 Transmitter System

We designed an SDR environment that sends out IQ signals of a selected modulation scheme to test the classifier. The data source and transmitter are implemented based on radioMLś implementation ([16], [6]), hence it can emit wireless signals from 10 different modulations (eight digital and two analog) and from two data sources (one digital and one analog). We implemented a transmitter on a software defined radio (SDR, specifically the USRP N200) with GNU Radio Companion. The transmitter has four stages, as shown in Figure 22. The

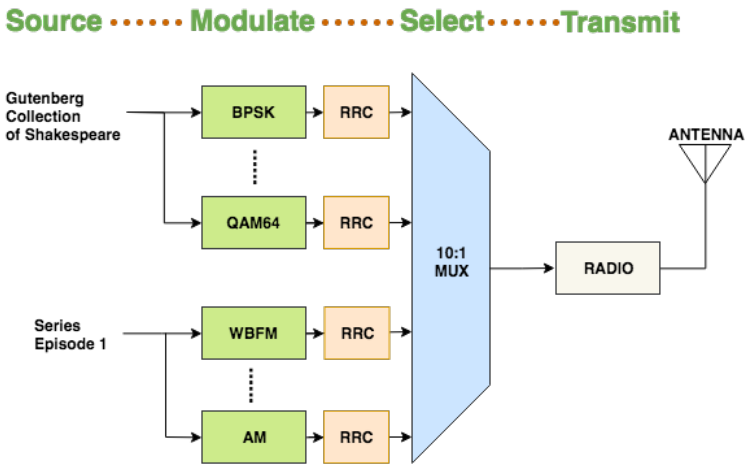

Figure 22: Transmitter Block Diagram

first stage is the source. Since we have both analog and digital modulation schemes, we need both digital and analog sources. Currently, the digital source is ASCII text from Gutenberg Collection of Shakespeare, while the analog source is an MP3 file of an Episode from the podcast *Series*. The second stage is modulation, where the digital and analog sources are modulated by ten different modulation schemes. A root-raised cosine filter is added to reduce intersymbol interference at the receiver. The third stage is select, where only one stream of symbols (selected by the user on the GUI) is sent to the transmitter. The final signal can be sent over the ISM band [17] through the transmitter. The transmission is verified with a spectrum analyzer.

5.2 Receiver System

The receiver system is built on the Zedboard (see Figure 23). It needs to obtain raw IQ samples over the air, process the sample through the FPGA neural network, and display the results in a user-friendly format. To this end, we designed the receiver system shown in figure **??**. The ARM core, DDR3 memory and FPGA fabric is available on the Zedboard. We purchased an SDR module[18] that attaches to the FMC connector of the Zedboard to capture wireless signals from the transmitter.

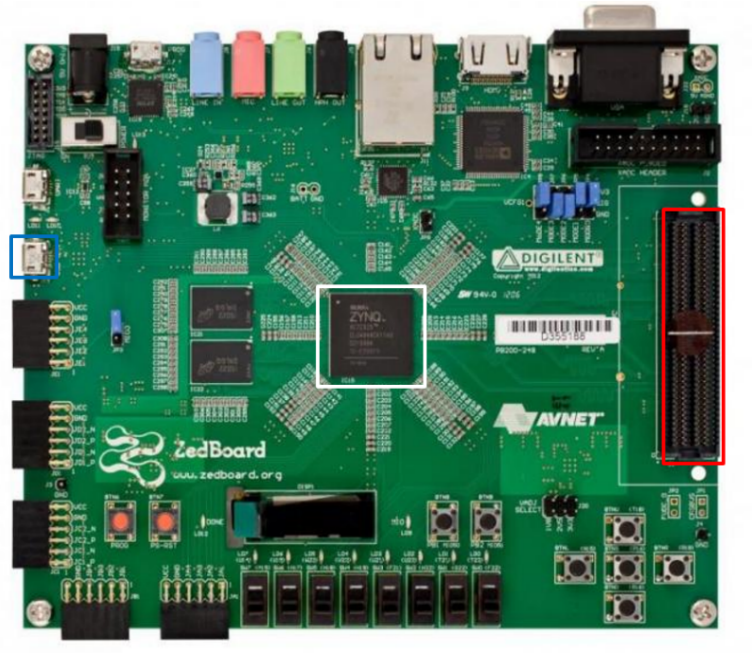

Figure 23: Picture of Zedboard highlighting the Zynq 7000 (in white square), FMC Connector (in red rectangle) and the UART port (in blue square)

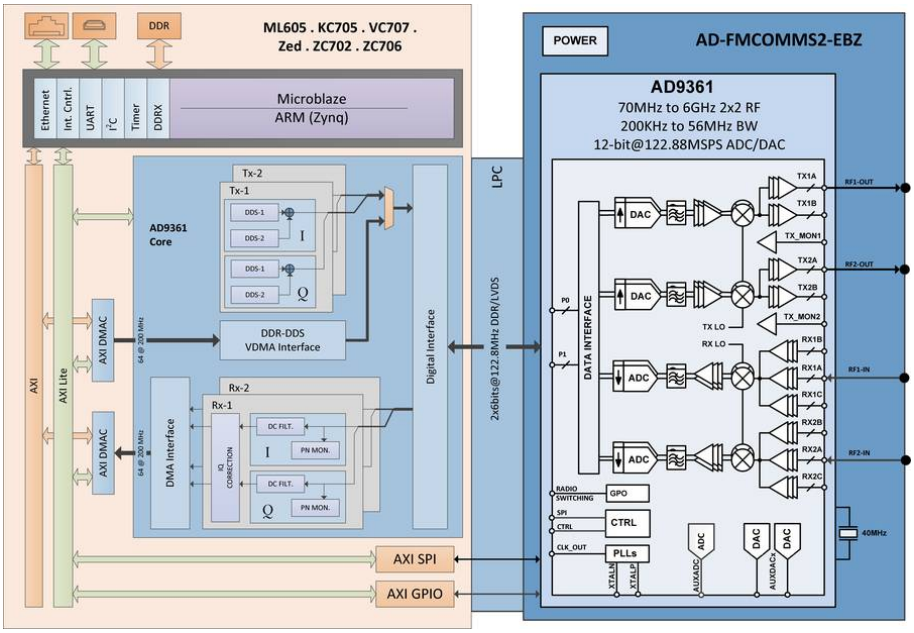

Figure 24: Block Diagram of SDR Receiver used in the Project

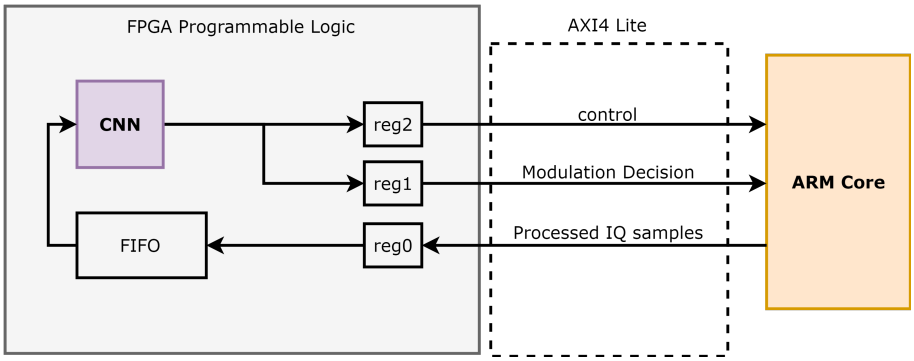

Figure 25: Connection between CNN and ARM Core

5.2.1 SDR Receiver

The SDR receiver is connected to the Zedboard through an FMC connector (see figure 23), where control commands and data can flow between the receiver and the ARM core. Figure 24 shows the high level operation of the receiver. After the SDR receives wireless signals through a VERT2450 antenna, it is brought down to baseband with the built-in local oscillator. This baseband signal is amplified with a low noise filter and filtered with an RF filter to remove the remaining high frequencies. The 12-bit ADC then converts the analog signal in the range $[-0.625, 0.625]$ to the corresponding integers in $[-2048, -2047]$ linearly . Finally, the data collected from the SDR is mapped to memory so that the ARM can access it. The interface between the ARM Core and the SDR is provided by Analog Devices ([19],[20]).

5.2.2 ARM Core

The ARM core acts as the master coordinator in the system. It is programmed in C with the Xilinx SDK in Vivado. It coordinates data flow between the radio receiver and the FPGA neural network with the AXI4 protocol [21], and subsequently displays their results to a host computer. In our system we use only the AXI4 Lite protocol. AXI4 Lite is used to transfer small amounts of data, and hence is useful for our system because it does not have much data transfer between FPGA fabric and ARM core.

A more detailed depiction of the interface and timing with the neural network are shown in figures 25 and 26. The ARM Core first converts the values captured from the SDR to

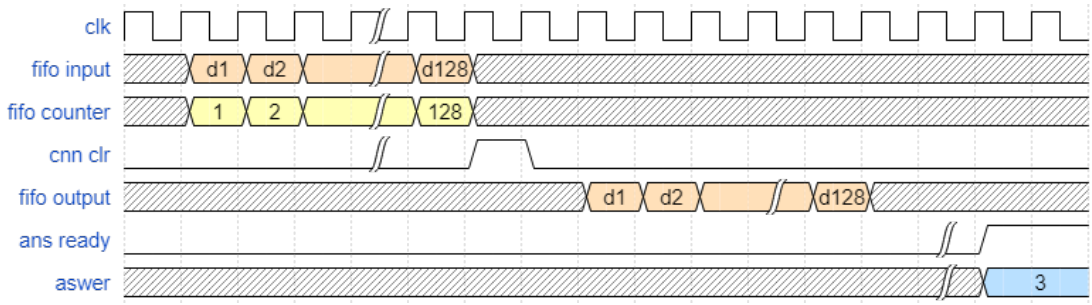

Figure 26: Timing diagram of FIFO and CNN

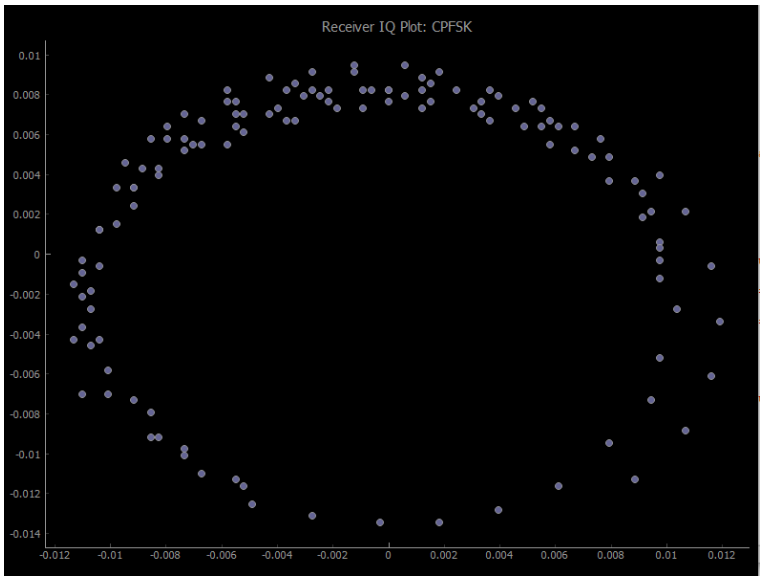

Figure 27: Output GUI on external Computer with IQ plot and classification results

Q(1.15) fixed point numbers. These 128 IQ values are then written sequentially to the input FIFO through a register. Once 128 IQ values are written, the CNN triggers the classification algorithm. Once the classification is ready, the ready bit is set to high and written to register2, while the answer is written to register 1.

Apart from coordinating communication between SDR and CNN, the ARM Core also preprocesses the IQ samples for the CNN. The output of the SDR ranges from -2048 to 2047, and maps linearly to a line with 1.25 Vpp.

To visualize the data The ARM core transmits data through the UART to a computer at a baud rate of 115200. The data is in ASCII format, and we made a data visualization program in Python to visualize the IQ samples and display the classification result (see figure 27).

Model	One-off accuracy	Voting accuracy	Time per classification
ARM	57%	69%	45 ms
FPGA	54%	62%	600 us

Table 8: Accuracy and Latency of FPGA and ARM classifier

6 Results

6.1 Testing Methodology

We performed several tests to test the accuracy and latency of the FPGA neural network. The architecture of the test transmitter is detailed in 5.1. We had two classifiers on the Zedboard – one implemented as fixed point on the FPGA, the other implemented as 32-bit floating point on the ARM CPU. Each classifier performed 1000 instances of classifications (100 per modulation) with each 128-sample time sequences. In addition, we implemented a voting strategy for both classifiers, such that one modulation is determined from the most frequent classification out of ten instant classifications. With the voting mechanism, each classifier performed 1000 classifications with every ten time sequences.

6.2 Results and Analysis

Table 8 shows the trade off between accuracy and latency of using the FPGA classifer. In terms of accuracy, The CPU performs better than the FPGA by approximately 5%. This is expected because floating point gives a higher precision than fixed point. However, the FPGA uses only 1% of the time of the CPU per classification, implying that a rise in precision also leads to a rise in latency.

A more detailed classification result is shown in the confusion matrices in figure 28. The CPU misclassified AM as FM – only the one-off FPGA classifer can classify AM as itself 10% of the time. It implies that the classifier might not be able to classify analog modulations. We also see that there are certain modulations that the classifiers are better at classifying. For instance, the CPU classifiers cannot classify CPFSK while that of the FPGA can; the FPGA classifiers cannot classify BPSK while that of the CPU can. Moreover, we also note that QAM64 are frequently mistaken to be QAM16 in all the classifiers except the CPU Voting

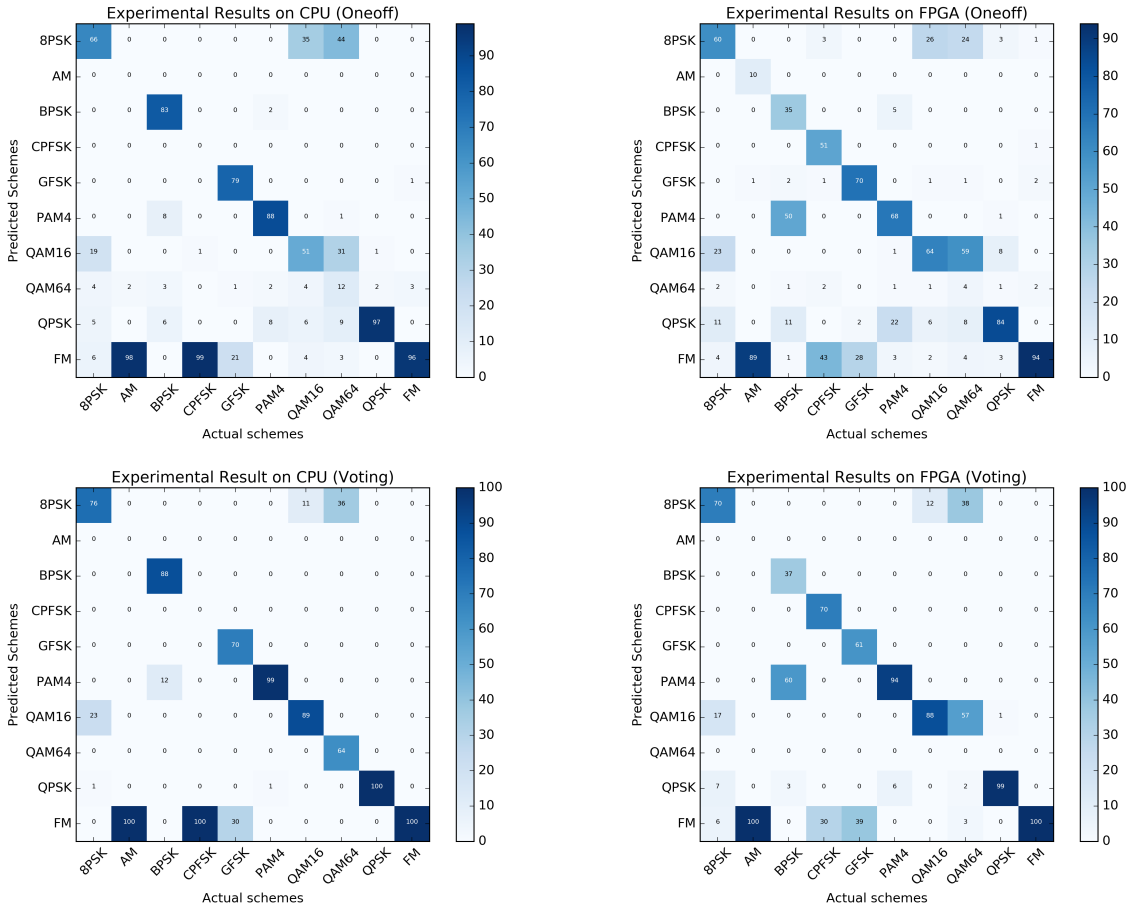

Figure 28: Confusion matrix of CPU (left) and FPGA (right) classifiers with one-off (top) and voting (bottom) strategies

classifier. This is expected because the QAM16 and QAM64 constellations look similar to one another even from the transmission end.

7 Conclusion

We implemented a custom architecture for convolutional neural networks in an FPGA. Our results show that it is possible to implement a state of the art neural network on a low power FPGA with relatively limited resources. We found that custom architectures could make significant improvements over GPUs in terms of power and latency. This opens up possibilities for mobile artificial intelligence solutions, especially aimed toward wireless communication applications. Future work might investigate the automatic generation of hardware archi-

tectures for AI and acquiring modulations in different environments for better classification accuracy.

References

[1] T. J. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer," *CoRR*, vol. abs/1702.00832, 2017. [Online]. Available: http://arxiv.org/abs/1702.00832

[2] S. Rajendran, W. Meert, D. Giustiniano, V. Lenders, and S. Pollin, "Distributed deep learning models for wireless signal classification with low-cost spectrum sensors," *CoRR*, vol. abs/1707.08908, 2017. [Online]. Available: http://arxiv.org/abs/1707.08908

[3] 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

[4] O. A. Dobre, A. Abdi, Y. Bar-Ness, and W. Su, "Survey of automatic modulation classification techniques: classical approaches and new trends," *IET Communications*, vol. 1, no. 2, pp. 137–156, April 2007.

[5] (2017) Deep learning for complete beginners: convolutional neural networks with keras. [Online]. Available: https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html

[6] (2018) Datasets - open radio machine learning datasets for open science. [Online]. Available: https://www.deepsig.io/datasets

[7] T. O'Shea and N. West, "Radio machine learning dataset generation with gnu radio," *Proceedings of the GNU Radio Conference*, vol. 1, no. 1, 2016. [Online]. Available: https://pubs.gnuradio.org/index.php/grcon/article/view/11

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner,

I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[10] M. . Simulink, "Compute Slope and Bias," https://www.mathworks.com/help/fixedpoint/ug/slope-bias-scaling.html, 2018, [Online; accessed 6-April-2018].

[11] Xilinx, "Zynq-7000 All Programmable SoC Data Sheet: Overview," https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, 2017, [Online; accessed 6-April-2018].

[12] V. Sze, Y. Chen, J. S. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," *CoRR*, vol. abs/1612.07625, 2016. [Online]. Available: http://arxiv.org/abs/1612.07625

[13] Wikipedia contributors, "Race condition — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Race_condition&oldid=826451953, 2018, [Online; accessed 6-May-2018].

[14] , "Race condition — Wikipedia, the free encyclopedia," http://asic-soc.blogspot.com/2008/04/clock-gating.html, 2018, [Online; accessed 6-May-2018].

[15] Wikipedia contributors, "Metastability (electronics) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Metastability_(electronics)&oldid=823306003, 2018, [Online; accessed 6-May-2018].

[16] (2016) Ffmpeg/gnu radio integration. [Online]. Available: https://github.com/osh/gr-mediatools

[17] A. Communications, "FCC Rules for Unlicensed Wireless Equipment operating in the ISM bands," http://afar.net/tutorials/fcc-rules/, 2017, [Online; accessed 6-April-2018].

[18] (2017) Ad fmcomms4 ebz user guide. [Online]. Available: https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms4-ebz

[19] (2018) no-os/ad9361 at master : analogdevicesinc/no-os. [Online]. Available: https://github.com/analogdevicesinc/no-OS/tree/master/ad9361

[20] (2018) analogdevicesinc/hdl: Hdl libraries and projects. [Online]. Available: https://github.com/analogdevicesinc/hdl

[21] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc.* UK: Strathclyde Academic Media, 2014.