MAIA: Modulation Classification with Artificial Intelligence Automata

Brenda So, Cory Nezin

December 23, 2017

Contents

1 Introduction **1**

2 Modulation Classification **2**

2.1 Signal Modulation Overview 2

2.2 State of the Art 3

3 Field Programmable Gate Arrays **4**

3.1 Configurable Logic Blocks 5

3.2 Block RAM 5

3.3 Digital Signal Processing Slice 5

4 Neural Network **6**

4.1 Introduction to Neural Networks 6

4.1.1 Rectified Linear Unit 7

4.1.2 Fully Connected Layer 8

4.1.3 Convolutional Layer 8

4.1.4 Maximum Pooling 8

4.1.5 Softmax 9

4.2 Training 9

4.2.1 Data Set 9

4.2.2 Architecture 11

4.2.3 Implementation Details 11

4.2.4 Performance 12

5 Hardware Implementation **12**

5.1 First Convolution Stage 12

5.1.1 Filter Banks 13

5.1.2 Rectified Linear Unit 15

5.1.3 Max Pooling 15

5.1.4 Intermediate Storage 16

5.2 Second Convolution Stage 16
5.3 Fully Connected Stage 19
5.4 Classification Stage 19

6 System Level Architecture **20**
6.0.1 The SDR Receiver 21
6.0.2 Universal Asynchronous Receiver Transmitter (UART) 21

7 Test Setup **22**
7.1 Test Results 23

Abstract

This paper describes work that has been in implementing a fully dedicated architecture for modulation classification on an field programmable gate array, namely the ZedBoard. We first give a background on basic neural network architecture and FPGAs. We follow with progress we have made in implementing a specific neural net architecture and challenges we have faced along the way. We discuss the interfacing of the FPGA fabric with the outside world via the processor on the board.

1 Introduction

Wireless modulation classification has been, and continues to be an important engineering problem. Sensing and classifying wireless signals is relevant to applications including government spectrum regulation, cognitive radio, and situational awareness in military/adversarial environments. Therefore, the need for a low cost, low latency and high accuracy modulation classifier is of the utmost importance. Recent academic research focused on developing algorithms that automatically classify modulation schemes. Since convolutional neural networks (CNN) have recently achieved impressive performance in image and audio recognition, they have recently been put to work in classifying modulation schemes. Although research has shown that CNN classifiers achieve better performance than other machine learning algorithms, there have been few actual implementations of the classifier in a physical environment. The goal of MAIA is to classify modulations with a convolutional neural network in a physical environment. We are going to implement a physical realization of a CNN that classifies modulation schemes and test it against wireless signals of known modulation in the real world. We have chosen to use a Field Programmable Gate Array (FPGA) to implement our solution. Although most neural networks are trained and used on a Graphics Processing Unit (GPU), it is unfeasible in the task of modulation classification. The high throughput of a GPU comes with the the expense of high power consumption and high latency. An FPGA, on the other hand, provides a low-latency and low-power solution. However, one of the challenges in implementing the neural network on the FPGA is the limited amount of resources available.

Section 2 of this paper will discuss the history of the modulation classification problem, and the ways in which is has been modeled and the attempted solutions. Section 3 gives a brief background on FPGAs, how they work and in particular the architecture of the ZedBoard which is the system we were using in our project. Section 4 gives an introduction to neural networks and discusses the architecture that our system will implement. Section 5 describes the architecture we designed for neural net implementation and what we have implemented so far. Section 6 describes the system we use for interfacing with the FPGA's SDR to pick up wireless signals. Section 7 discusses the entire testing setup including the programming

of the transmitter SDR.

2 Modulation Classification

2.1 Signal Modulation Overview

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. Current modulation schemes can be 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.

2.2 State of the Art

On the receiving end of a radio communication system, a matched filter is commonly used to recover the symbol. Expert designed filters are used to convolve over the received signal, forming peaks when the matching symbol is found. [5] 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. [7] 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. [7] 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. [7] 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

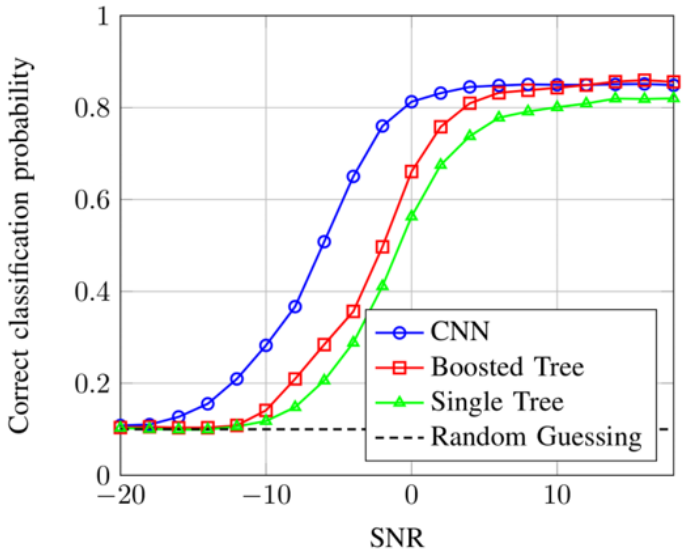

Figure 1: Classification accuracy of different models at different SNR [4]

obtain due to the short time nature of the collected signals. [4] 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 algorithms (see Figure 1).

3 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 Xilinx FPGAs.

3.1 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. The ZedBoard contains 53,200 LUTS.
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" which has the advantage of being close to the logic and therefore very fast and low power. The ZedBoard contains 106,400 flip flops

3.2 Block RAM

Block RAM (BRAM) is another type of static memory that makes up the majority of RAM on an FPGA. Unlike distributed RAM, 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 FIFO logic and error correction. The ZedBoard contains 140 BRAM slices.

3.3 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

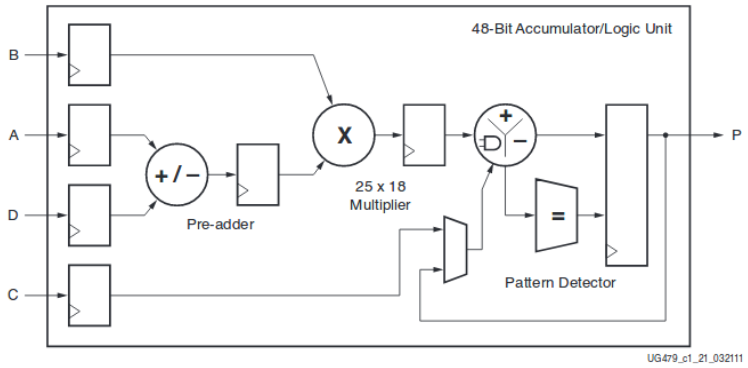

Figure 2: The Xilinx DSP slice.

functions such as X XOR Y, X AND Y, X OR Y, etc. A simplified block diagram is shown in figure 2. The ZedBaord contains 220 DSP slices.

4 Neural Network

4.1 Introduction to Neural Networks

Neural networks are a class of machine learning algorithms that are modeled after the development of neurons. Contrary to traditional machine learning, where the user extracts specific features from the raw data to train the algorithm, neural networks learn the features directly. How the neural network actually functions is defined by the network architecture. A neural network is typically constructed by composing many different functions. A specific composition of function in a neural network is called a neural network architecture. The functions used in building a specific architecture are common design patterns in neural networks with slight alteration.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 this case is the cross

entropy function. The cross entropy function formula for an M-class classifier is shown in equation 4, where y is the actual probability of the observation o being in class c, while p is the probability predicted by the neural network.

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

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.

In a classification problem, the final output of the neural network is a number associated with the class that the raw data is classified as. Note that the input of the neural network is often the raw input itself. In the modulation classification problem, the input to the neural network is a time sequence with 128 IQ values captured over the air. The range of $f(x)$ is the set of integers from 0 to 9, which is the number associated with the modulation class. Now we will discuss some of the common function used in neural networks.

4.1.1 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).

4.1.2 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.

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

Note that all the input values and the weights of the layer are all pairwise "connected" by a weight in the matrix.

4.1.3 Convolutional Layer

A convolutional layer uses convolution to generate another mapping of its 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 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). Convolutional neural networks have recently gained popularity due to their impressive performance on image and time series data sets.

4.1.4 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.

4.1.5 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 interpretation 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.

4.2 Training

4.2.1 Data Set

In order to train a CNN, we need a large amount of data. Luckily, such data is publicly available from radioML - an organization that provides simulated radio transmission data. [1] Each transmission sequence is 128 samples long with in-phase and quadrature samples, as shown in figure 3. The SNR of these signals ranges from $-20dB$ to $18dB$. [3] The data provided encompasses signals from 8 digital modulation schemes and 3 analog modulation schemes, including BPSK, AM-DSB, 8PSK, CPFSK, GFSK, PAM4, QAM16, QAM64, QPSK, AM-SSB and WBFM.

The simulated signals are generated by GNU Radio and passed through the simulation chain in figure 4. The source alphabet is the data source used for signal modulation, which in this case is chosen to be either Serial Episode 1 (for analog modulation) and Gutenberg's work of Shakespeare in ASCII format (for digital modulation). Afterwards, the source is modulated by a chosen modulation scheme and passed through the dynamic channel model of GNU Radio, which takes the following effects [3] into account:

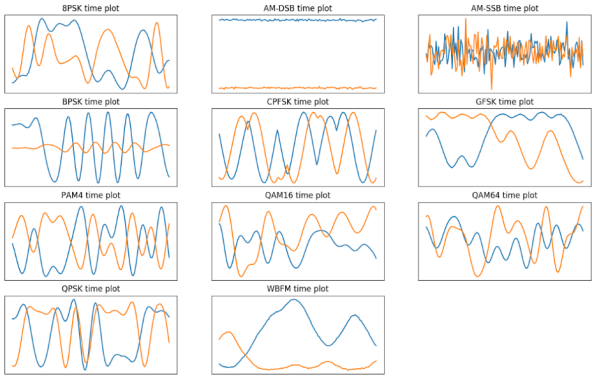

Figure 3: Signals generated by different modulation schemes in radioML. The blue line represents the in-phase samples while the orange line represents quadrature samples.

1. Sampling Rate Offset: Simulated by a clipped random walk process that steps along with a fractional interpolator at a rate of $1 + \epsilon$ input samples per output sample.

2. Center Frequency Offset: Similar to the simulation of sampling rate offset, the frequency offset is simulated by a random walk process that changes the frequency of the local oscillator.

3. Selective Fading Model: The model uses a sum of sinusoids method with random phase noise to simulate Rician and Rayleigh fading processes.

4. Noise Model: Noise sampled from a normal distribution centered at zero is added to the simulated signal to model noise the environment and circuit.

Although radioML is a useful data set for training a CNN, it may not be sufficient to use it to test the CNN. This is because the data set is synthetic, generated by the GNU radio dynamic channel modeling API. [2] This API incorporates effects of sampling rate offset, center frequency offset, selective fading and noise into the simulated channel. [3] However, these effects are still simulated, so they are only approximations of physical channels at best. For instance, the simulated data did not take into account interference from other channels. Hence, the best way to test the usefulness of our CNN is to bring it into a physical environment.

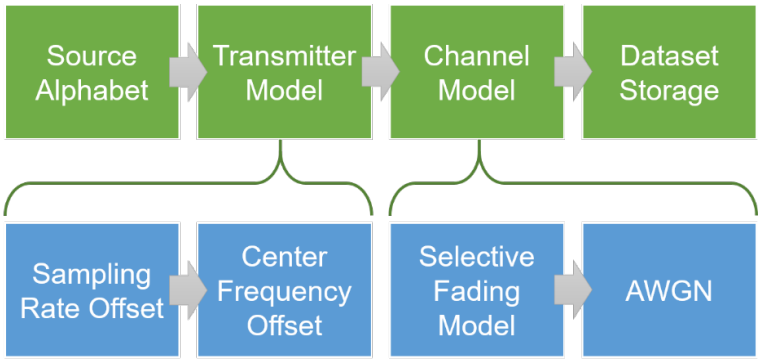

Figure 4: Chain of simulation of radioML. The green boxes represent the main stages of simulation, while the blue boxes represent the effects that the simulation takes into account.

4.2.2 Architecture

Our modulation classifier uses a convolutional neural network that was proven to work in previous literature. The neural network architecture was designed by O'Shea [4], and the high level architecture is shown in Table 4.2.2.

	Layer	Output Dimension
1	Input	2×128
2	Convolution + ReLU (128 filters, size 2×8)	128×121
3	Max Pooling (size 2, stride 2)	128×121
4	Convolution + ReLU (64 filters, 128×16	64×45
5	Max Pooling (size 2, stride 2)	64×22
6	Flatten	1408
7	Fully Connected + ReLU	128
8	Fully Connected + ReLU	64
9	Fully Connected + ReLU	32
10	Fully Connected + Softmax	10

4.2.3 Implementation Details

We used TensorFlow, an open-source deep learning framework, to reconstruct and train the neural network with the radioML data set. One of the challenges that we faced when

implementing the neural network was the lack of details in the paper. [4] 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. [5] 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 [1] with a learning rate of 0.01.

2. Each training sample is used 100 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.

4.2.4 Performance

The model in Table 4.2.2 converges and reaches satisfactory classification accuracy, as shown in figure 5. As expected, the better the SNR, the higher the classification accuracy. Note that at SNR higher than zero, the accuracy stayed around 80%.

5 Hardware Implementation

5.1 First Convolution Stage

As previously discussed in the section on neural network architecture, the first stage of the neural network is convolution. To reiterate, the raw in-phase and quadrature data are separately processed through very large filter banks, requiring 256 filters in total. The result

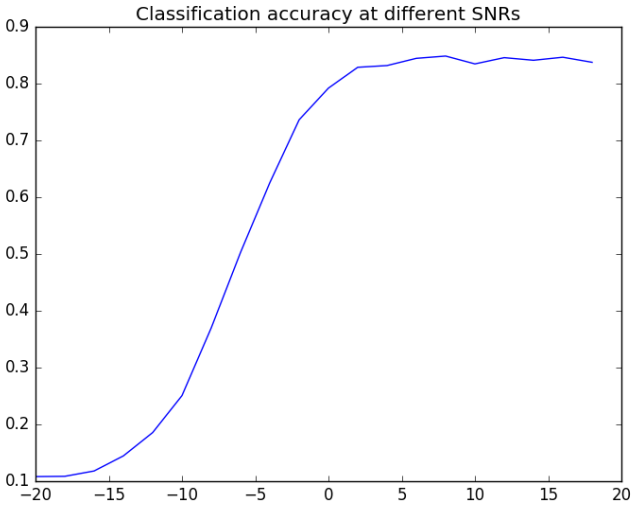

Figure 5: Graph showing classification accuracy of classifier under different SNR. The x-axis is the SNR, the y axis is the classification accuracy

of the in-phase filters and the quadrature filters is then added, yielding 128 filtered signals. Each signal is then processed through the ReLU non-linearity, and max pooling is applied, yielding signals of half the original size.

5.1.1 Filter Banks

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, we can see that given the previous description, this would require $256 \times 8 = 2048$ DSP slices, far beyond the ZedBoard's capacity of 220.

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.

Considering the first stage is the smallest in terms of both number of computations required and number of coefficients required, it is clearly impossible to have fully dedicated hardware. Therefore, we pursued a time division multiplexed architecture. Because we are dealing with a relatively low coefficient precision of 8 bits, it is actually possible to enumerate all of the 256 possible multiplierless multipliers that would be required for time division multiplexing. However in order to make a filter out of this, we would either need to repeat this 8 (the number of taps) times or feed the result back in until the accumulation of one convolution result has finished. The former would be too hardware intensive and the latter would be too slow. Either method would also require fairly complicated and costly muxing logic to choose the right coefficient.

In the final design, we implemented time domain multiplexing with DSP slices. 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 8 DSP slices for the in-phase and quadrature filters. Because the operation is "Multiple Instruction Single Data" (MISD) we had to implement a state machine which controlled flow of the input data and the switching time of the filter coefficients. We made the process efficient by implementing a circular buffer which would broadcast the input signal to the filter as well as itself until it was processed through every filter. 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, the result is added at each step of the filtering using a simple LUT adder. The implementation uses 837 D flip-flops, 1109 LUTs, and 2 BRAM.

We wrote the same convolution in software with full precision to verify that the fixed point

approximation did not significantly impact results at this stage. We confirmed that all values were correct to within three or four decimal places.

5.1.2 Rectified Linear Unit

The rectified linear unit (ReLU) is a relatively simple function to implement in hardware. To reiterate, if a signal is fed through a ReLU nonlinearity, the result is 0 where the signal is negative, and equal to the input otherwise. This action is clearly similar to a mux choosing between 0 and another input, and since all processing is done in twos complement the signal is negative if the most significant bit (MSB) is 1. Therefore we arrive at the elegant implementation of a ReLU, simply a mux with "0" as one input and the signal as the other, while the address bit is tied to the sign bit of the signal. Since many modern neural network nonlinearites consist of similar piece-wise definitions separated at zero, the implementation is fairly general.

5.1.3 Max Pooling

As discussed, max pooling is a form of non-linear down sampling. Max pooling can be thought of as a two stage process: First, apply a maximum filter, i.e. slide a window across the signal and take the maximum of elements within the window as the output. Second, down sample by the appropriate factor. The specific case of this neural network architecture simplifies a few things. The pool width is 2, meaning instead of taking the maximum of multiple numbers (as we will see in the hardware implementation of arg max) we must only take the maximum of two numbers. This can easily be accomplished in hardware by taking the difference and checking the sign bit. We then repeat a design pattern used for the ReLU, and feed the signal into one input of a mux and the signal delayed by one into the other input, with the sign bit controlling the address pin. This accomplishes maximum filtering. Interestingly, the maximum of two numbers is also given by $max(a, b) = 1/2(a + b + |a - b|) = 1/2(a + b + r(a - b) + r(b - a))$ where r denotes the ReLU function. While this form doesn't seem to offer a reduction of hardware, it may allow to reuse other components in our architecture.

All that is left is to apply down sampling by a factor of two (since the stride of this max pool

is two) which is fairly standard in signal processing applications and easily accomplished by feeding the output of the maximum filter into a D flip-flop which is driven by a clock frequency at half the rate of the input clock frequency. Note that this implementation results in half of the computations being wasted, though skipping them would require more logic and therefore more hardware. Despite the additional hardware cost, it may be beneficial as it might reduce power, though it's impossible to tell from the current progress in the project. For the ReLU and max pooling stages, we implemented a simple 16-bit linear feedback shift register (LFSR) in VHDL and tested all possible pairs of 8-bit numbers. We confirmed an exact match to what we expected, calculated in software.

5.1.4 Intermediate Storage

Because the following stage requires multiple values from the first convolution stage, the output values must be stored in an intermediate location. Because they are to be accessed sequentially, a large FIFO was chosen as the intermediate. The first convolution produces 7680 activations at 26 bit precision, but FIFOs come in depths of powers of two and widths of 4. In order to efficiently make use of the built in primitive FIFOs, we truncated to 24 bits yielding utilization of 6 $8Kib \times 4b$ FIFOs.

5.2 Second Convolution Stage

The second convolution stage is similar to the first but on a much larger scale. In terms of neural net coefficients, it is the largest. In terms of total number of computations required, it is rivaled only by the largest fully connected layer (to be discussed later) which is a 1408×100 matrix multiplication. During our first run through on a design for this layer, we misunderstood what was actually implemented in the original paper's neural net. We thought that there were multiple coefficient sets, each of which was applied to the sum of the previous activations. However upon further inspection, this layer is actually performing a very large two dimensional convolution. To make things clear, you may think of the output of the previous stage as being a 128×60 array. The filtering portion of this layer convolves that array with 64 128×16 kernels to produce a 64×45 result which is then fed through a

ReLU and max pool, and flattened into a vector to produce a length 1,408 vector.

Attempting a similar approach to the first convolutional stage, we see that this can be calculated with $64 \times 128 = 8192$ 16 tap FIR filters. If we were to implement this in a similar way, it would require 1024 DSP slices running in parallel, as opposed to our previous 16, which greatly exceeds our limit. Usually in digital arithmetic, it is possible to sacrifice speed for hardware in equal proportions. We suspected that we would be able to reduce the number of DSP slices from 64×16 to 64 at the price of a $\times 16$ slow down. In fact, this method corresponds to the quadratic method of convolution. While it was found to be the slowest possible approach, the non-parallel FIR filter strategy is the only hope of implementing the second convolutional layer with the given hardware constraints.

In implementing quadratic convolution, we no longer have the luxury of built in primitives and therefore it must be designed from scratch. In order to accomplish this, we designed a central controller which would tell the input FIFO and intermediate value FIFOs when to read and write. The controller also communicates with a multiply accumulator which accumulates over 16 products, producing the output of a convolution at one time step. This final value is added to the content of an output FIFO and the process is repeated for all 128 activations yielding two dimensional convolution.

Because the process is very complicated and is essentially 3D processing, the controller is very complex and was one of the longest and hardest components to develop. It is essentially a state machine with 7 states: Idle, initialization, advance, accumulate, output, load, and end.

In Idle, the controller is waiting for a signal to start processing - this signal is expected to be given by the "full" flag on the interconnection FIFO, or "prog full" (meaning programmable full) in the case where the expected signal size is not a power of 2 (for example in our case it is 7680). On this signal, the state machine enters the initialization state, where the first 16 samples of the signal are loaded into one FIFO and the first 16 weights are loaded into another FIFO. The machine then enters accumulate mode, adding up the products of the signal and the filter coefficients (note that the architecture assumes that the coefficients are already flipped so it actually performs correlation rather than convolution). Now that the correct value is accumulated, it is written to the output FIFO. Now, the machine enters the

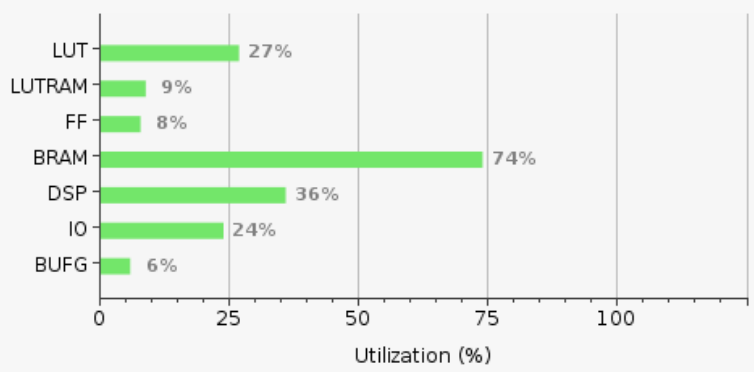

Figure 6: Hardware utilization after implementing both convolution layers.

load state and new coefficients are loaded into the filter FIFO. Now entering the advance state, in order to produce the next output of the convolution, we must drop the first value from the signal and add the next value from the interconnection FIFO, i.e. advance the signal. We then enter the accumulate state and continue until all filters have been used and the correct output has accumulated. Now the machine enters the end state to signal to the following layer that the intermediate values are ready to be processed.

We use this controller to drive 64 convolutions in parallel, each using a single DSP slice wrapped with its own small controller. We attempted to centralize the DSP controllers as well, but it resulted in odd timing issues and didn't significantly reduce the logic so we decided not to pursue that route any more, though it may be of interest in future work.

We wrote software using limited precision and randomized coefficients to confirm that the results of the convolution exactly match what they should be for a smaller simulation of one convolution with four filter coefficient sets. The signal values were given from an 8-bit LFSR. While both stages are confirmed to work together on a small scale, it has yet to be confirmed on the same scale as the neural net.

We "synthesized" (generated logic for the VHDL) and "implemented" (generated a mapping from logic to ZedBoard primitives) all previously described designs. Our current total utilization is summarized in figure 6. This represents our current progress in the hardware implementation of the neural network.

5.3 Fully Connected Stage

The fully connected stage is relatively very simple. It consists of several fully connected layers, each layer simply being a matrix multiply followed by a ReLU. A matrix multiply can be thought of as a series of dot products. In the simpler case of a matrix times a vector, which is what we have in the neural network, a matrix multiply is just the vector dotted with each of the rows of the matrix. The current planned architecture takes the output of the previous stage (in a flattened format) and broadcasts it to several multiply accumulators. The other input to the accumulator is the coefficient of the matrix loaded in from RAM. Using this design, we can reuse the dot product component that we used in the second convolution stage. This design computes some subset of the final result in parallel and then moves onto the next subset until the final result is produced. Note that we would produce the entire result in parallel however there are not enough remaining DSP slices to accomplish this. Producing a relatively small subset of the final answer in parallel also has the benefit that the next fully connected layer will be able to use the same hardware. This is at no loss of hardware to the previous stage since the computation would not be allowed if the result was still begin accumulated anyway.

One very possible road block in implementing this stage is that we will run out of memory on the board. We are currently considering the possibility of storing coefficients in a compressed manner, perhaps just using a simple Huffman coding scheme. This would have the disadvantage of introducing significantly more complexity, but may be necessary in order to allow all of the coefficients to be stored.

5.4 Classification Stage

This is probably the easiest and most simple stage. Normally in a neural network built for classification, the output stage is a softmax function (a continuous approximation of the arg max function) however this is primarily for the purpose of normalizing the result during training. Since we are focusing on inference, we may replace the complex mess of exponentials with a simple arg max. In order to product an arg max in one clock cycle we, in theory, need to compare every number to every other number, which can be done as with

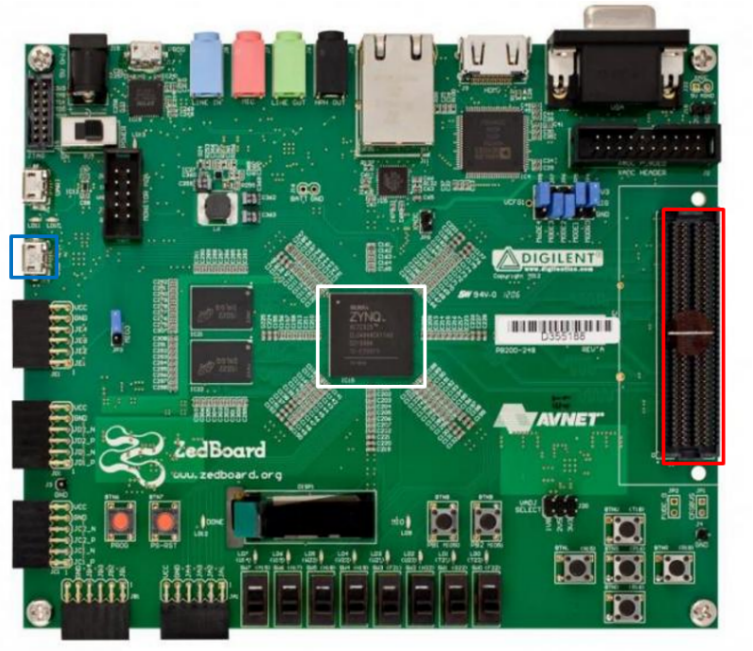

Figure 7: Picture of Zedboard highlighting the Zynq 7000 (in white square), FMC Connector (in red rectangle) and the UART port (in blue square)

the ReLU by subtracting the numbers and checking the sign. Obviously even for our modest case of 10 numbers this is not optimal. However, this is what we have implemented in VHDL so far. A more space efficient, but less time efficient method would be to run through 10 clock cycles and keeping the largest number that appears, thus reducing computation by a factor of 10 and increasing time by a factor of 10. This may be implemented in the future if the previous design proves burdensome.

6 System Level Architecture

In order to test the neural network in a real environment, we must set up a system to interface MAIA with the physical environment. The system needs to take in IQ samples from the external environment, preprocess and send the samples into MAIA, poll the output of MAIA and display the result in a readable format. We used the peripherals available on the Zedboard to build such a system, which is shown in Figures 10 and 8

The SDR receiver accepts signal via the antenna and delivers the data to DDR3 RAM, which

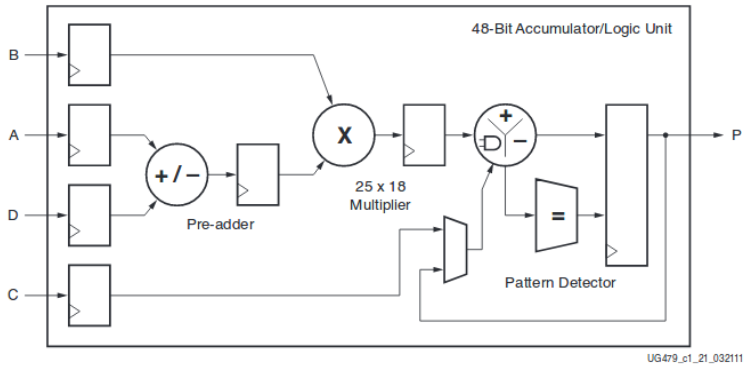

Figure 8: System Level Architecture of modulation classifier. Components of the processing system (in blue) interfaces with the peripherals (in purple) and MAIA, which is on the FPGA Fabric (in green).

is the main memory of the system. The ARM Core requests the data via DDRX, a built-in DDR Controller, to obtain the raw data form the SDR. It then preprocesses the data, such as normalizing the data and converting the data to a fixed point number. The preprocesses data is then delivered to MAIA for classification. Finally, the result from MAIA is displayed to the host computer via UART.

6.0.1 The SDR Receiver

The SDR transceiver is connected to the FPGA through an FMC connector Figure 10). While data is transmitted in chunks through the AXI interface, control commands to the SDR, such as the filter coefficients, are transmitted through the AXI-Lite interface. After receiving the signal from the antenna, the low noise amplifier (LNA) amplifies the signal without perturbing the signal too much. The RF filter filters out noise at higher frequencies and the oscillator brings the high frequency signal down to baseband. The analog digital converter then converts the analog signal to digital form. Finally, the data collected from the SDR is mapped to memory so that the CPU can access it.

6.0.2 Universal Asynchronous Receiver Transmitter (UART)

Generally, UART is used as an intermediary between serial and parallel ports. In this case, it is used to communicate between the ZedBoard (parallel) and the USB port (serial) on a

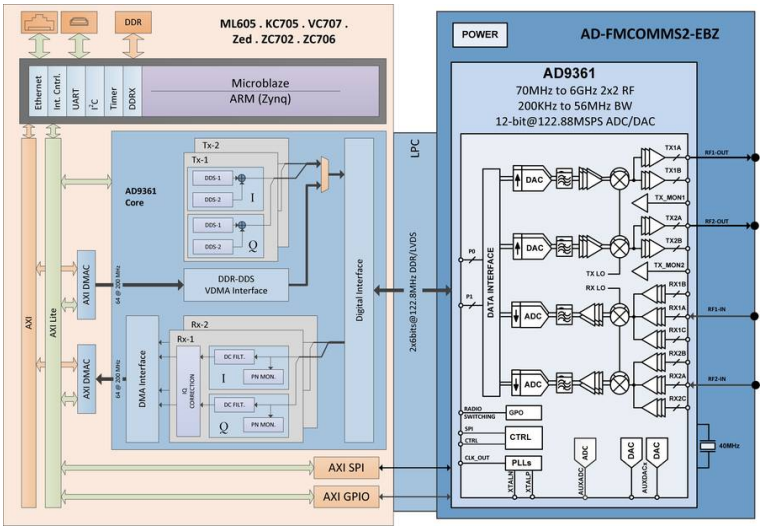

Figure 9: High level overview of ad fmcomms4 ebz.

host computer. In the system, UART is used to send standard output from the ARM Core to a host computer which can display debugging information and results.

7 Test Setup

We designed an SDR environment that sends out IQ signals of a selected modulation scheme to test MAIA. RadioML, the data set 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 Figure 11. 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 source is a random source that generates numbers 0 256. The second stage is modulation, where the digital and analog sources are modulated by ten different modulation schemes. 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

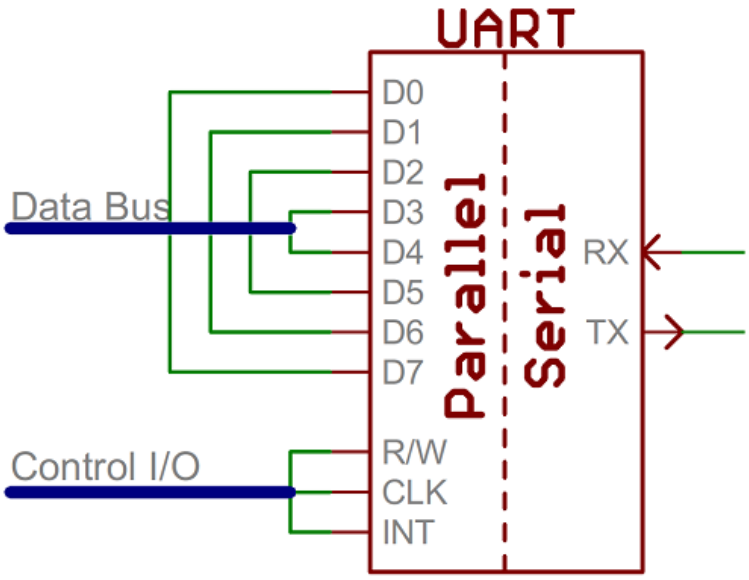

Figure 10: Simplified version of the UART interface.

and a multiplexer 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.

The system was implemented and verified with a spectrum analyzer (see Figure 10). As the SDR was transmitting a signal at the ISM band (2.4 GHz), the spectrum analyzer showed a peak at 2.4 GHz (see Figure 13). We also know that the peak is not due to the presence of Wifi since when the SDR was turned off, the peak disappeared. Additionally the shape of the spectrum matches that of the transmitted signal.

7.1 Test Results

We successfully interfaced the ARM Core with the SDR receiver and UART port. To verify the reception, we built the neural network in C and compiled it with the binary image on the ARM core to classify live signals.

Unfortunately, we did not obtain the accuracy we wanted. In fact, judging from the classified results, the classifier tends to classify the signals as AM signals regardless of the modulation scheme we used. One explanation to this phenomenon is the frequency offset between the transmitter and receiver. Because the transmitter and receiver are not running on the same clock, the frequency offset is not monitored and hence the IQ plot resembles a circle, which

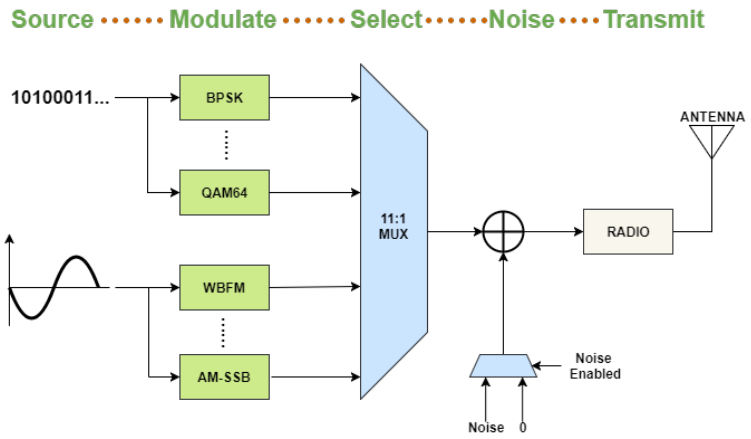

Figure 11: Block diagram of wireless test setup.

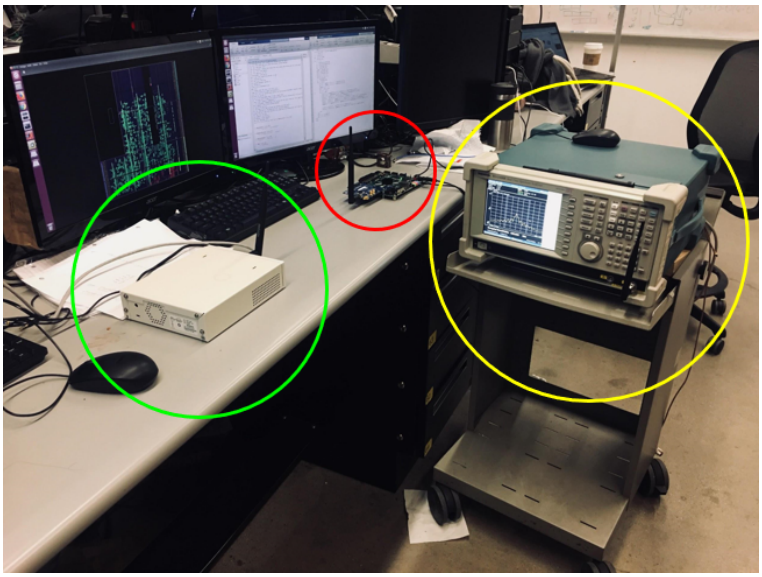

Figure 12: Setup of wireless environment. The SDR (circled in green) sends signals to the FPGA (circled in red) and is verified by the spectrum analyzer (circled in yellow).

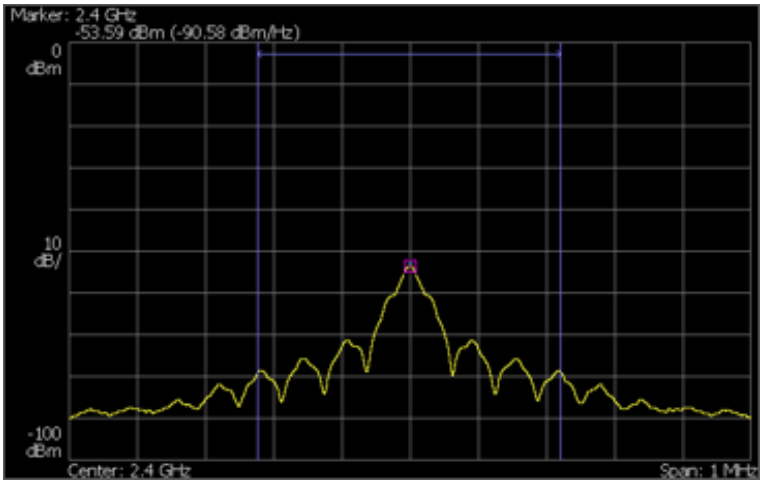

Figure 13: Screen shot from spectrum analyzer. Notice the peak at 2.4 GHz, which is the center frequency of the ISM band.

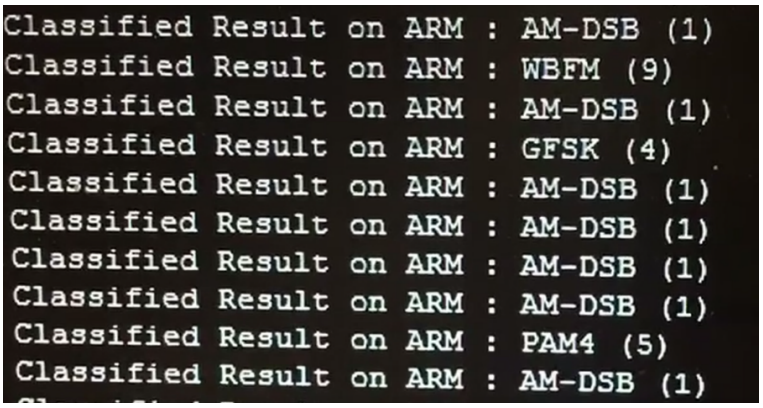

Figure 14: Sample output from ZedBoard to host computer.

is characteristic of AM signals. By synchronizing the clocks, we hope to obtain a cleaner signal at the receiver and increase the classification accuracy.

References

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

[2] N. W. Timothy J OShea, Radio Machine Learning Dataset Generation with GNU Radio, in Proceedings of the GNU Radio Conference, vol. 1, 2016.

[3] T. J. OShea, radioML Dataset, 2016, https://github.com/radioML/dataset.

[4] J. H. Timothy J OShea, An Introduction to Deep Learning for the Physical Layer, CoRR, vol. abs/1702.00832, 2017. [Online]. Available: http://arxiv.org/abs/1702.00832

[5] T. J. OShea and J. Corgan, Convolutional radio modulation recognition networks, CoRR, vol. abs/1602.04105, 2016. [Online]. Available: http://arxiv.org/abs/1602.04105

[6] 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

[7] O. A. Dobre, A. Abdi, Y. B. Ness, and W. Su, Survey of Automatic Modulation Classification Techniques: Classical Appraoches and New Trends, IET Communications, vol. 1, pp. 137156, 2007.