Cairo University
Faculty of Engineering
Electronics and Communications dept.

Wireless Signal Identification Receiver

Prepared by:

Abdelrahman Emad Labib

Abdelrahman Farid Shawkey

Rawan Sayed Ali

Mohammed Hassan Mohammed

Hisham Mohamed Abdelhamed

Supervised by:

Dr. Hassan Aboushady

Dr. Hassan Mostafa

A Graduation Project Report Submitted to
The Faculty of Engineering at Cairo University in Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science
In
Electronics and Communications Engineering.
Faculty of Engineering, Cairo University
Giza, Egypt. August 2020.

ACKNOWLEDGMENTS

Gratitude breeds success. First of all, we have to thank our research supervisors.
Dr. Hassan Aboushady and Dr. Hassan Mostafa for offering the opportunity to work with them at Digital electronics field. Without their assistance, dedicated involvement in every step throughout the process and their faith in our work, this project would have never been accomplished.

Most importantly, none of this could have happened without our families.
We would like to thank them for their constant support, patience, and sacrifices for many long years

ABSTRACT

In the past decade, Deep Learning (DL) and especially Convolutional Neural Networks (CNNs) have shown extremely growth due to their effectiveness at a wide range of applications especially in image and audio. One of these applications is 5G wireless communication systems which require portable devices capable of analyzing the spectrum congestion and establishing communication on the available frequency bands using the appropriate standards. This graduation project presents a CNN algorithms Implemented on FPGA that balance between the three main factors for portable devices: accuracy, area, and power. Algorithms target two active research fields in 5G systems. First field is Wireless Interference Identification used in identifying the source of interference to prevent spectrum congestion. Second field is Modulation Recognition used in recognizing the type of the signal for demodulation purpose which is one of the main tasks in establishing communication between two systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

A	Amplitude
AM-SSB	Amplitude Modulation Single Side Band
AM-DSB	Amplitude Modulation Double Side Band
BPSK	Binary Phase Shift Keying
BFSK	Binary Frequency Shift Keying
CPFSK	Continuous Phase Frequency Shift Keying
CNN	Convolutional Neural Network
CR	Cognitive Radio
DL	Deep Learning
DSA	Dynamic Spectrum Access
F	Frequency
FPGA	Field Programmable Gate Array
GSM	Global System for Mobile Communication
GMSK	Gaussian Minimum Shift keying
IEEE	Institute of Electrical and Electronics Engineers
IOT	Internet of Things
IQ	In-Phase & Quadrature
ISM	Industrial Scientific Medical
LUT	Look Up Table
PAM	Pulse Amplitude Modulation
P	Precision
QAM	Quadrature Amplitude Modulation
R	Recall
ReLU	Rectified Linear Unit
SNR	Signal to Noise Ratio
WBFM	Wide Band Frequency Modulation
WII	Wireless Interference Identification

CHAPTER ONE: INTRODUCTION

1.1 Problem statement

As a starting point, we need first to state the issue we have been dealing with and what really lead us to start doing this project. As a definition, the radio spectrum is considered as a subset of electromagnetic spectrum with waves known as radio waves, these waves are lying starting from 3 kHz reaching up to 300 GHZ, where the spectrum is portioned into ranges with different frequencies and limits of bandwidths. There are very low, low, medium, high, very high, ultra-high, super high and extremely high range of frequencies. As the frequency is increased, the range of the Bandwidth is increased as well. All applications added to the typical services we use; all of the radio-communication services and all of the cellular networks are having different parts of radio spectrum bands by a process defined as frequency allocation. Frequency allocation is done by setting out what radio services can use which frequency band out of the whole spectrum and under what conditions; if the band is licensed or not. All of these terms are applied on the national frequency allocation table. The spectrum allocation of TV, E-GSM, GSM-1800, 3G (UMTS) and Wi-Fi is inside the partition of Ultra-high frequency starting from 880-915 MHZ heading to 5.725-5.875 GHZ.

As an example, let's talk about the cellular networks; the generations we have, 5G spectrum, which is also GSMA, is allocated a range of radio frequencies in the super high frequency partition, referring to the radio frequencies from the user to its serving base station carrying data in a bi-direction. It's allocated at super high frequency. Hence, it needs very wide bandwidth of 1GHZ to achieve higher bit rate compared to the other technologies and the previous generations. Some of the applications that their frequencies are allocated; broadcasting, Air band, Marine band, Citizens' band and personal radio services, Industrial, scientific, medical, Land mobile bands, Radio control, radar and many other unlimited applications.

Another important example is IoT (Internet of things). IoT, as a scientific definition, is a network of internet connected objects able to collect and exchange data. As we can say that you have "objects" around you surrounded by data which it senses and collect in order to send it through the internet. IoT serves a huge number of devices and application upon which definition. It's a matter of fact that the number of IoT products have surpassed a huge number of humans on planet. By 2021, there may be around 20 billion IoT smart devices up and running with an increase in the demand of 5G network. IoT devices are basically which have the possibility to connect through the internet and are able to interact with the other surrounding devices over the internet with a remote access to users so that they can manage the device as per they need. A few examples for such smart devices are; top smart devices in market, smart mobiles, smart refrigerator, smart watches, smart door lock at home or office, medical sensors, fitness trackers, smart security system, smart fire alarm, smart bicycle and many other uncountable IoT devices.

All of the technologies and applications mentioned above need to have their own frequency band in a way so that the spectrum is manageable, to ensure the fact of frequency assignment to all of the services, which leads us to our problem. Consequently, this leads the spectrum to be way crowded as it has never been before and difficult to be accessed. It will require in the future for the spectrum to be extended so that it can take all of these allocations due to the increment of the upcoming technology, applications and services we have in our daily life that are used on certain band in the spectrum. The spectrum extension is not that easy because of the several limitations we have on different bands. In addition to that, the spectrum utilization is not the same for all of the band, some partitions in the band are over-utilized and some bands are under-utilized. This is determined depending on the licensed and unlicensed bands. All of which will guide the subject to a big problem; spectrum scarcity.

The question is; is the spectrum resource scarce? Yes. The main objective for now is to manage its scarcity. The long time needed to make a final decision, the diversity of the working parties, the huge amount of applications and the upcoming technology, the growing number of the participants seem to encourage the idea of that finite resources needed to be shared among more users so that it can be manageable. However, the user must be informed the extraordinary growth of radio systems by being more diverse and powerful, for the whole team benefit. Who could imagine; 50 years ago, the "mobile phone revolution", the world wide satellite services and the broadcasting of thousands of TV programs, everywhere? As a matter of fact, at that time already, the spectrum was thought to be busy in the future! Therefore, Yes. The spectrum scarcity is for real.

In order to assist the spectrum load and usage, there was an activity going by telecommunication experts to evaluate and diagnose to find out that radio services during combination with permanent progress of technology improve the efficiency of the resources year after year. As a claiming thought, about no limitations on the radio services neither on quantity nor quality, but this improvement upon a tighter optimization works. So, what is need to be done is to figure out how the spectrum can be shared, how we can contain the spectrum scarcity and under what terms and conditions this can be happening. All of these wondering prompted to have a whole new level in which the "Cognitive radio" has been approved. Cognitive radio (CR) is an adaptive, intelligent radio and network technology that can automatically detect available channels in a wireless spectrum to allocate frequencies to whoever needs and change the transmission parameters enabling more communications to run simultaneously and improve the operating behavior as well. In other words, Cognitive Radio (CR) is a smart way to deal with the spectrum and services that needs an available channel at specified time slots without sensing the band to be busy or interference with another. Cognitive Radio (CR) has shown a good impact on the spectrum utilization and its resources that require sensing, monitoring and understanding by its awareness and utilizing the under-utilized partitions in the spectrum bands in order to improve a better performance.

1.2 Cognitive radio as a proposed solution

As per the fact that the spectrum utilization lately has been worse, and the crowded spectrum that we all have witnessed as well as the solution which gave us a better utilization with a good performance and desired efficiency, Cognitive radio (CR). Cognitive Radio performs various tasks as spectrum sensing and dynamic spectrum access (DSA) for more awareness about the situation. In order to support CR's tasks, Machine learning has showed up to provide means and methods so that one can learn from and be adapted to the dynamic spectrum access (DSA). And as a particular subject to be specific, deep learning can process "uncooked" spectrum data and operate on the representations by capturing and analyzing high dimensional and dynamic spectrum data that conventional feature-based machine learning algorithms fail to clutch. One of its properties is considered as Convolutional Neural Network (CNN) algorithm for modulation recognition. CNNs have recently witnessed a great success in image classification driven by an improvement in algorithms, large public repositories and high-performed computing systems such as GPUs. Deep neural network has been used to automatically learn the features from the data itself, and develop a data-driven detection approach. Inspired by the powerful capability of Convolutional Neural Network (CNN) in extracting features of matrix-shaped data. As per the fact that the existing spectrum sensing methods make decisions based on using model-driven test statistics that has probability ratio of failure or missed error, this was replaced by deep-learning detection (CNN) mechanism. Figure 1 shows the cognitive radio cycle.

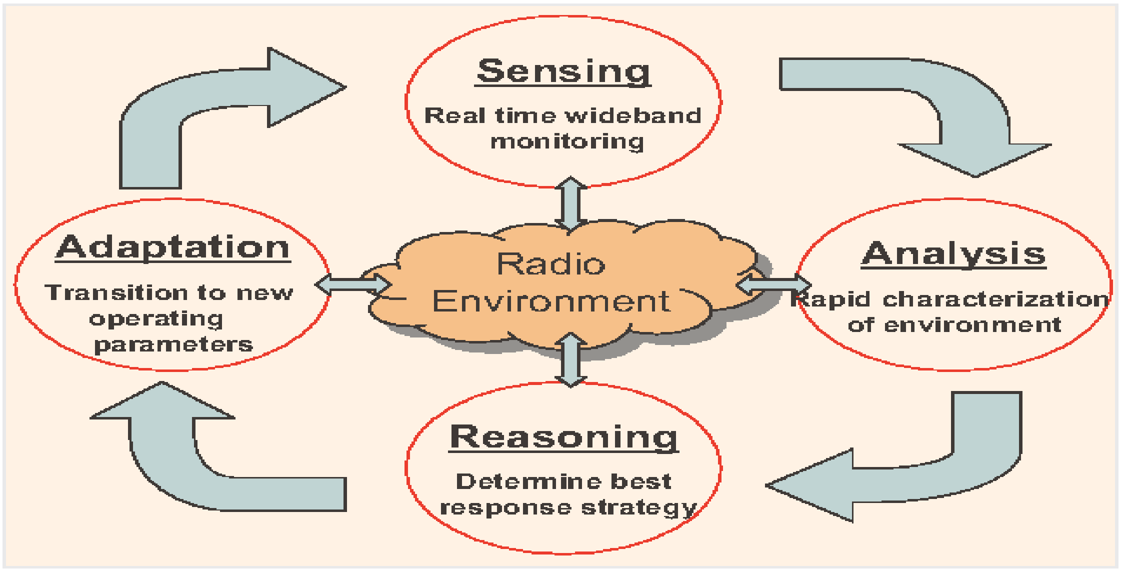

Sensing
Real time wideband monitoring
Analysis
Rapid characterization of environment
Reasoning
Determine best response strategy
Adaptation
Transition to new operating parameters
Radio Environment

FIGURE 1: COGNITIVE RADIO PROCEDURE.

Accordingly, we can say that Modulation Recognition and Wireless Interference Identification are related to cognitive radio. By using CNN algorithm either for a multipath algorithm, including I/Q, Amplitude/Phase and frequency, or only one path which is I/Q CNN model (This will be discussed in details later) seeking for better accuracy, better performance and satisfying results compared to other related works as in O'shea [1] and soltani models [2].

Regarding the data sets [1],[3], Modulation Recognition datasets consists of 11 modulation schemes/classes; 8 Digital and 3 analog modulation, all are widely used in wireless communication systems all around the world. This is classified as; BPSK, QPSK, 8PSK, 16QAM, 64QAM, BFSK, CPFSK, PAM4 for digital modulations and WB-FM, AM-SSB, AM-DSB for analog modulations. Data is modulated at a rate of roughly 8 samples per symbol with a normalized average of transmitted power of zero db. 70% of data is used for training and 30% of data is used for test. Wireless Interference Identification dataset consists of 15 modulation classes with ten, two and three frequency channels of IEEE 802.15.1, IEEE 802.15.4 and IEEE 802.11 b/g complaint signals.

Back to the cognitive radio, it's a task of a human mind. It acts with several steps to be done. First, sensing the radio environment. Second, analysis the data. Third, determining the best strategy to make the best decision. Finally, adaptation for the new transition parameters to make the spectrum sharing happen. We are focused on the middle part starting from sensing the environment then sweeping the spectrum to make the spectrum share.

1.3 Description of Terms

In order to achieve the solution, in other words the CR, we have 2 methods to make it happen. First, Modulation recognition. As a definition, we determine the Modulation Scheme of an unknown signal coming to the receiver antenna which will help in so many applications and diverse function. Second, Wireless Interference. As a definition, we identify the source of interference and its band.

1.3.1 Modulation recognition

As we said, it defines the modulation scheme, this may be useful in diverse functions. One of them is that we can determine the standards of the upcoming signal to the receiver. Since that each generation in the standards has its own modulation scheme (e.g. GSM's modulation scheme is GMSK, DAMPS's modulation scheme is DQPSK, etc...), this will make us define the standards of the signal easily. Another advantage we can be helped with in the Demodulation block itself in its construction and when we need to make the demodulation step at the receiver. Concerning the CR, we can say that we detect any user unlicensed in the licensed band and that's the relation among the monitoring for the spectrum resource. In addition, we cannot forget that modulation recognition leads us to not collide and less interference in band since that each user is assigned by its own part of the band. Therefore, neither collision nor jamming. Another functions can be done by modulation recognition and that what makes it very important.

1.3.2 Wireless Interference

Since we are talking about wireless network and wireless medium, Interference remains the most limiting factor of wireless network capacity. It has its ways to be calculated and measured and also to be recovered from. We managed to make an identification for the wireless interference and that for Wi-Fi, Zigbee, and Bluetooth in the ISM band since it's a free band to be accessed freely (Unlicensed). Functions for the wireless interference detection is either detecting the interferers to reduce the interference effect on other users and that's concerning monitoring the spectrum resources, or methods to detect the interference such as disc on/off method which is all about the common range and the distance between the transmitter and the receiver compared to the common range or exact methods by comparing the SINR of the signal to a certain threshold which gives us an exact value for the interference.

What we have done is that we used the CNN algorithm for both modulation recognition and wireless interference done in software and hardware level as will be discussed in the upcoming chapters all in details.

1.4 State of the Art

Before we talk about previous work in this field, we need to ask an important question. Why did we choose to implement Modulation recognition model and wireless Interference by using CNN (Convolutional Neural Network)? Why specifically CNN?

The answer to that question will lead us to the advantage of CNN over any other techniques (e.g. KNN, SVM, DNN, etc.…) CNN doesn't need any engineering knowledge or years of research. It doesn't need an expert or anything unlike any other old techniques. In addition to its very high accuracy compared to others. It can be used in many things at a time for example modulation recognition and wireless interference both. The graph below clarifies the classification accuracy for different training examples. The solid line as we can see is the highest of all which represents the CNN with different dropouts while the dotted line represents other techniques with low accuracy.

Briefly, we can say that CNN accuracy is the best compared to all of other's accuracy as shown in Figure 2[1].

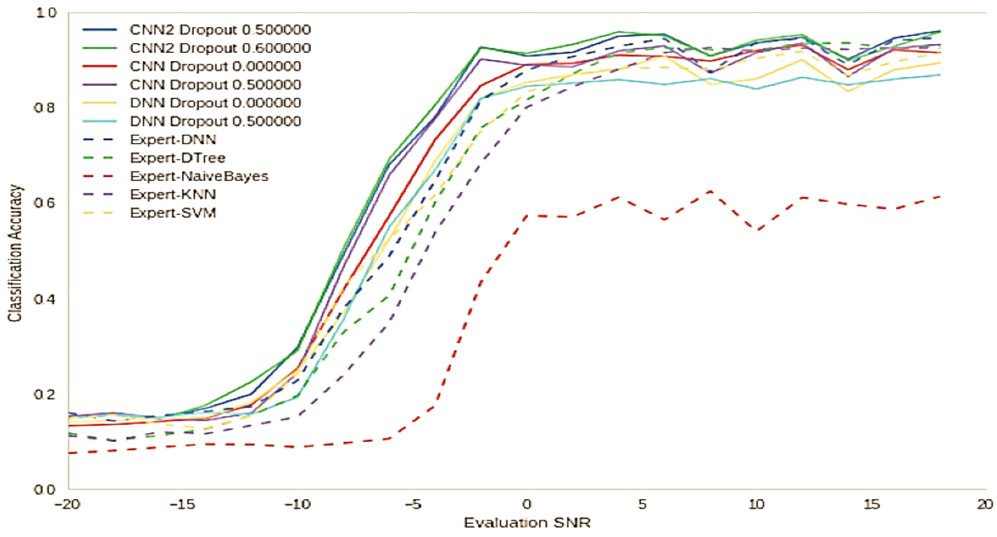


FIGURE 2: ACCURACY CURVES OF DIFFERENT TECHNIQUES.

Back to the previous work, for the RF signal classification, M. Kulin's paper [4] has made a processing pipeline for end-to-end learning from spectrum data. First step is data acquisition concerning getting the data in phase and quadrature phase at the receiver antenna to be put in data vectors as an input for the next step. The second two steps are pre-processing and classification, those what will be highlighted in our project as we did software simulation and hardware implementation for those two steps. For pre-processing, it takes care of analysis and manipulation for data vector from the previous step to be pipelined as an input for signal processing tools that analyze, process, transform data into simple data representation such as frequency, amplitude, phase and more features to be extracted by Machine learning. For classification, its concerned about the spectrum access scheme, modulation format, wireless technology, type of interfering, detecting available spectrum band and so on.

The last step is decision step, the predictions calculated in ML model will be an input for the decision module. To be the output in the last step for the pipelining process. The process illustrated above is summarized in Figure 3 [4].

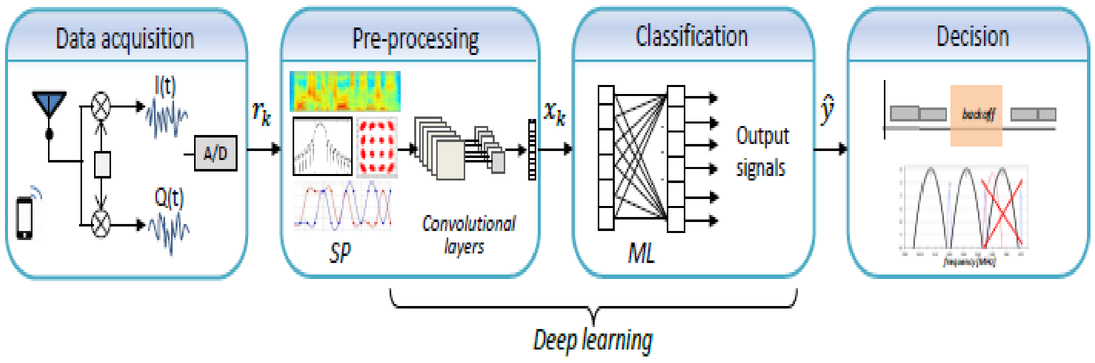


FIGURE 3: WIRELESS IDENTIFICATION PROCESS.

CHAPTER TWO: FUNDAMENTAL CONCEPTS

In order to properly delve into the proposed designs and work, we should first review some fundamental theory behind Convolutional Neural Networks. This Chapter will review the mathematical properties of.

This chapter depends on kulin et al paper [4].

2.1 Machine learning

Machine learning (ML) refers to a set of algorithms that learn a statistical model from historical data. The obtained model is data-driven rather than explicitly derived using domain knowledge.

The goal of ML is to find a mathematical function, *f that* defines the relation between a set of inputs *X*, and a set of outputs *Y,* i.e.

$$f : \mathrm{X} \rightarrow \mathrm{Y} \quad (1)$$

The inputs,$X \in R^{m*x}$, present a number of distinct data points, samples or observations denoted as

$$X = \begin{bmatrix} {X_1}^T \\ {X_2}^T \\ \vdots \\ {X_m}^T \end{bmatrix} \quad (2)$$

Where m is the sample size, while xi 2 Rn is a vector of n measurements or features for the ith observation called a feature vector,

$$x_i = [x_{i1} \quad x_{i2} \quad \cdots \quad x_{in}]^T \ (3)$$

The outputs,$y \in R^m$, are all the outcomes, labels or target values corresponding to the m inputs xi, denoted by

$$y_i = [y_1 \quad y_2 \quad \cdots \quad y_m]^T \ (4)$$

Then the observed data consists of m input-output pairs, called the training data or training set, S,

$$s = \{(x_1, y_1), (x_2, y_2), \dots., (x_2, y_2)\} \ (5)$$

Each pair (x_i, y_i) is called a training example because it is used to train or teach the learning algorithm how to obtain f. In machine learning, f is called the predictor whose task is to predict the outcome yi based on the input values of xi. There are two classical data models depending on the prediction type, described by:

$$f(x) = \begin{cases} regressor, & if\ y \in R \\ classifier, & if\ y \in \{0,1\} \end{cases} \quad (6)$$

In short, when the output variable *y* is *continuous* or quantitative, the learning problem is a *regression* problem. But, if *y* predicts a discrete or *categorical* value, it is a *classification* problem.

Given a training set, *S*, the goal of a machine learning algorithm is to *learn* the mathematical model for *f.* To make sense of this task, we assume there exists a fixed but unknown distribution,$p(x, y) = px(x)p(y|x)$ according to which the data sample is identically and independently distributed (i.i.d).

Here, $px(x)$is the marginal distribution that models the uncertainty in the sampling of the input points, while$p(y|x)$, and is the conditional distribution that describes the statistical relation between the input and output.

Thus, *f* is some fixed but unknown function that defines the relation between *X* and *Y.* The depicted ML algorithm determines the functional form or shape. The unknown function *f* is estimated by applying the selected learning method to the training data, *S*, so that *f* is a good estimator for new unseen data, i.e.

$$y \approx \hat{y} = \hat{f}(x_{new}) \ (7)$$

The predictor f is parametrized by a vector$\theta \in R^n$, and describes a parametric model. In this setup, the problem of estimating f reduces down to one of estimating the parameters

$\theta_i = [\theta_1 \quad \theta_2 \quad \cdots \quad \theta_n]^T$. In most practical applications, the observed data are corrupted versions of the expected values that would be obtained under ideal circumstances. These unavoidable corruptions, typically termed noise, prevent the extraction of true parameters from the observations. With this in regard, the generic data model may be expressed as

$$y = f(x) + \epsilon \quad (8)$$

Where $f(x)$ is the model and ϵ are additive measurement errors and other discrepancies. The goal of ML is to find the input-output relation that will ``best'' match the noisy observations.

Hence, the vector may be estimated by solving a (convex) optimization problem. First, a loss or cost function

$l(x, y, \theta)$ Is set, which is a (point-wise) measure of the error between the observed data point y_i and the model prediction of $\hat{f}(x_i)$ for each value ofθ. However, θ is estimated on the whole training data, S, not just one example. For this task, the average loss over all training examples called training loss, J, is calculated:

$$j(\theta) \equiv j(s, \theta) = \frac{1}{m} \sum_{(x_i, y_i) \in s} l(x, y, \theta) \quad (9)$$

Where S indicates that the error is calculated on the instances from the training set and $i = 1, \ldots., m$. The vector θ that minimizes the training loss J(θ), that is

$$\frac{argmin}{\theta \in R^n} J(\theta) \ (10)$$

Will give the desired model. Once the model is estimated, for any given input x, the prediction for y can be made with $\hat{y} = \theta^T x$.

In engineering parlance, the process of estimating the parameters of a model that is a mapping between input and output observations is called system identification. System Identification or ML classification techniques are well suited for wireless signal identification problems.

2.2 Deep learning

The prediction accuracy of ML models heavily depends on the choice of the data representation or features used for training. For that reason, much effort in designing ML models goes into the composition of pre-processing and data transformation chains that result in a representation of the data that can support effective ML predictions. Informally, this is referred to as feature engineering. Feature engineering is the process of extracting, combining and manipulating features by taking advantage of human ingenuity and prior expert knowledge to arrive at more representative ones, that is

$$\emptyset(d) : \mathrm{d} \rightarrow \mathrm{x} \ (11)$$

I.e. the feature extractor - transforms the data vector $d \in R^d$ into a new form, $x \in R^n$, more suitable for making predictions. The importance of feature engineering highlights the bottleneck of machine learning algorithms: their inability to automatically extract the discriminative information from data.

Feature learning is a branch of machine learning that moves the concept of learning from ``learning the model'' to ``learning the features''. One popular feature learning method is deep learning. In particular, this paper focuses on convolutional neural networks (CNN).

Convolutional neural networks perform feature learning via non-linear transformations implemented as a series of nested layers. The input data is a multidimensional data array, called tensor, which is presented at the visible layer. This is typically a grid-like topological structure, e.g. time-series data, which can be seen as a 1D grid taking samples at regular time intervals, pixels in images with a 2D layout, a 3D structure of videos, etc. Then a series of hidden layers extract several abstract features. Those layers are ``hidden'' because their values are not given. Instead, the deep learning model must determine which data representations are useful for explaining the relationships in the observed data. Each layer consists of several kernels that perform a convolution over the input; therefore, they are also referred to as convolutional layers. Kernels are feature detectors, which convolve over the input and produce a transformed version of the data at the output. Those are banks of finite impulse response filters as seen in signal processing, just learned on a hierarchy of layers. The filters are usually multidimensional arrays of parameters that are learnt by the learning algorithm through a training process called backpropagation. For instance, given a two-dimensional input x, a two-dimensional kernel h computes the 2D convolution by

$$(x * h)_{ij} = x[i,j] * h[i,j] = \sum_{n}\sum_{m} x[n,m].h[i-n][j-m] \; (12)$$

I.e. the dot product between their weights and a small region they are connected to in the input. After the convolution, a bias term is added and a pointwise nonlinearity g is applied, forming a feature map at the filter output. If we denote the l-th feature map at a given convolutional layer as h^l , whose filters are determined by the coefficients or weights w^l , the input x and the bias b^l , then the feature map hl is obtained as follows

$$h^l{}_{ij} = g\big((w^l * x)_{ij} + b_i\big) \; (13)$$

Where * is the 2D convolution while g (.) is the activation function. Typically, the rectifier activation function is used for CNNs, which is defined by g(x) = max (0; x). Kernels using the rectifier are called RELU (Rectified Linear Unit) and have shown to greatly accelerate the convergence during the training process compared to other activation functions. Others common activation functions are the hyperbolic tangent function.

$tanh, g(x) = \frac{2}{1+e^{-2x}} - 11$, and the *sigmoid* activation $g(x) = \frac{1}{1+e^{-x}}$ In order to form a richer representation of the input signal, commonly, multiple filters are stacked so that each hidden layer consists of multiple feature maps,

$$h^{(l)}\{l = 0, \ldots\ldots, L\} \ (14)$$

The number of filters per layer is a tunable parameter or hyper-parameter. Other tunable parameters are the filter size, the number of layers, etc. The selection of values for hyper-parameters may be quite difficult, and finding it commonly is much an art as it is science. An optimal choice may only be feasible by trial and error. The filter sizes are selected according to the input data size so as to have the right level of granularity that can create abstractions at the proper scale. For instance, for a 2D square matrix input, such as spectrograms, common choices are 3×3, 5×5, 9×9, etc. For a wide matrix, such as a real-valued representation of the complex I and Q samples of the wireless signal inR^{2*N}, suitable filter sizes may be 1 ×3, 2 ×3, 2 × 5, etc. The penultimate layer in a CNN consists of neurons that are fully-connected with all feature maps in the preceding layer. Therefore, these layers are called fully-connected or dense layers. The very last layer is a SoftMax classifier, which computes the posterior probability of each class label over K classes as

$$\widehat{y_i} = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_i}} \quad (15)$$

I=1, 2….K that is, the scores z_i computed at the output layer, also called logits, are translated into probabilities. A loss function, l, is calculated on the last fully-connected layer that measures the difference between the estimated probabilities,$\widehat{y_i}$ and the one-hot encoding of the true class labels, y_i. The CNN parameters θ, are obtained by minimizing the loss function on the training set$\{x_i, y_i\}\epsilon s$ of size m,

$$\frac{min}{\theta} \sum_{i\epsilon s} \{\widehat{y_i}, y_i\} \ (16\,)$$

Where L(.) is typically the mean squared error the categorical cross-entropy

For which a minus sign is often added in front to get the negative log-likelihood.

To control over-fitting, typically regularization is used in combination with dropout, which is a new extremely effective technique that ``drops out'' a random set of activations in a layer. Each unit is retained with a fixed probability p, typically chosen using a validation set, or set to 0:5 which has shown to be close to optimal for a wide range of applications.

CHAPTER THREE: SOFTWARE APPROCH

In this chapter we will discuss the software approach we took to solve the problem encountered in modulation recognition by proposing PSI model and Optimized IQ model in section 1 of this chapter, and problem encountered in wireless interference identification by proposing PSI and PSI Lite models in section 2 of this chapter.

3.1 Modulation recognition

Modulation recognition in the cognitive radio is the method that allows the detection of the modulation scheme used without previous knowledge of the sent data.

3.1.1 Dataset

The data set used in the literature and in our work on the modulation recognition is the RadioML 2016.10a Modulation [1] which includes 11 modulation schemes and consists of 220,000 In-phase and quadrature (I/Q) represented data vectors divided to 20 different SNRs limited between [-20 : 18] dB and each symbol consists of 256 sample (2 x 128).

Figure 4 shows the 11 modulation schemes included in the data set.

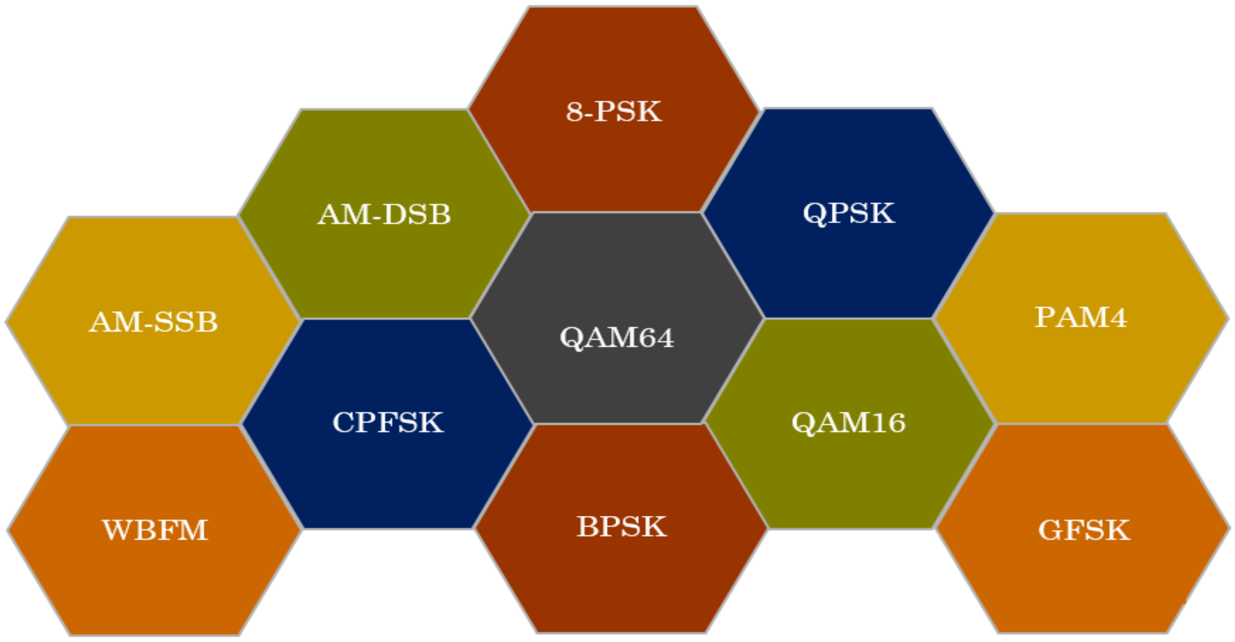


FIGURE 4: MODULATION SCHEMES.

3.1.2 Literature in modulation recognition using CNN

The first model used in modulation recognition is the model proposed by Oshea [1], this model is a sequential model consists of 2 convolutional layers and 2 dense layers, the last one of them classifies between the 11 different modulation schemes in the data set.

Figure 5 shows Oshea’s model block diagram.

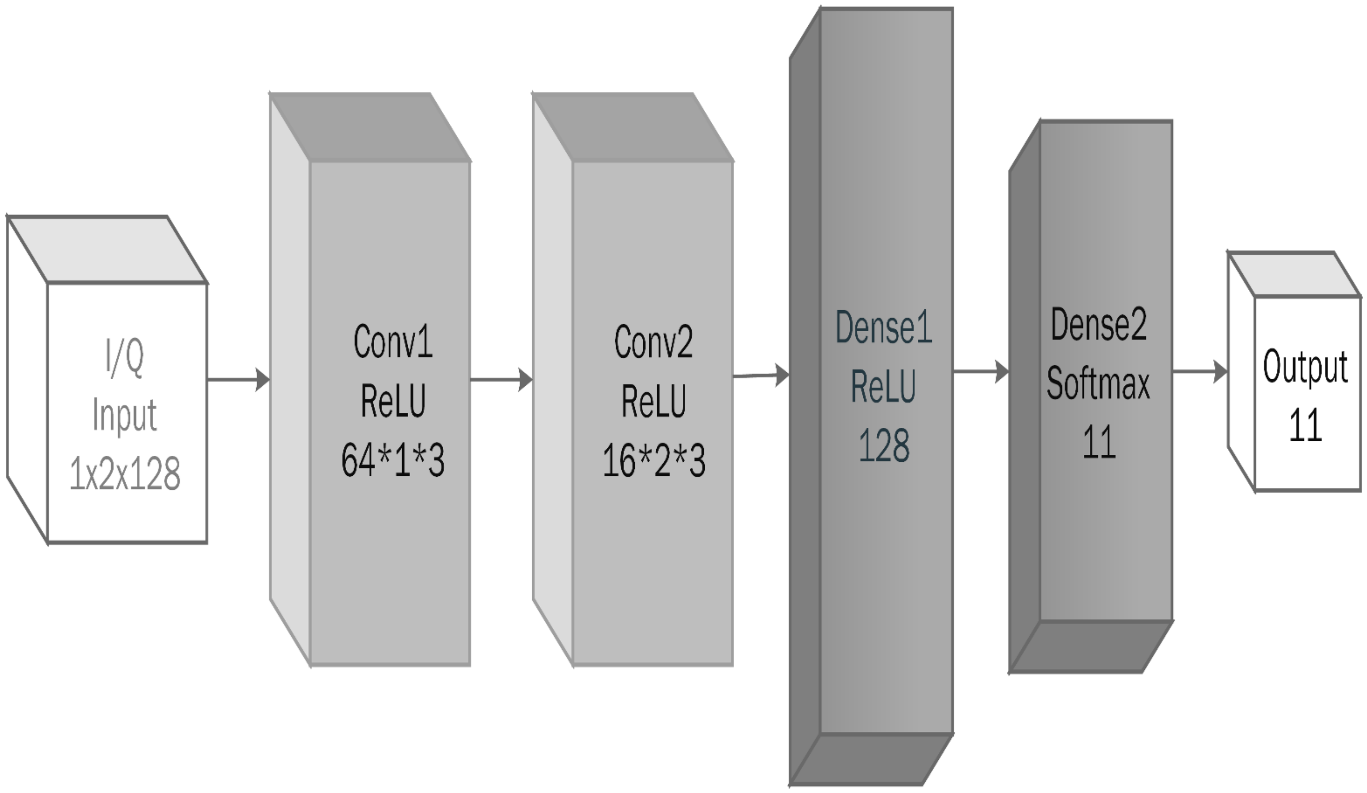


Figure 5: O’Shea’s Sequential Model.

Table 1 shows the detailed structure of Oshea’s model illustrating each layer input size, parameters like number of filters, filter size and number of neurons it also mentions the activation functions of each layer.

TABLE 1: OSHEA'S MODEL STRUCTURE

Layer Type	Input Size	Parameters	Activation Function
Convolution Layer	2 x 128	256 filter filter size 1 x 3 Dropout=0.6	ReLU
Convolution Layer	256 x 2 x128	80 filter filter size 2 x 3 dropout=0.6	ReLU
Fully Connected Layer	10240 x 1	256 neurons dropout=0.6	ReLU
Fully Connected Layer	256 x 1	11 neurons	SoftMax

Then Kulin et al reproduced Oshea's model and made signal processing on the data set RadioML 2016.10a Modulation which In-phase and Quadrature represented, once to convert it to Amplitude and Phase representation (A/φ) and another to convert it to frequency modulated representation (F).

The data set is represented as 2 vectors x_i carries the in-phase samples and x_q holds the quadrature samples.

To transform to Amplitude and Phase representation (A/φ) these equations is used:

- Amplitude samples (A) = $(x_i^2 + x_q^2)^{\ 1/2}$ (17)
- Phase samples (φ) = $arctan\left(\frac{x_q}{x_i}\right)$ (18)

To transform to frequency representation Kulin [4] used Fast Fourier transform(FFT).

Kulin [4] then trained and tested the model with each data representation to study the effect of the representation on the model accuracy.

Figure 6 [4] shows the cognitive processing diagram showing RF module and modulation of the signal from air. The signal is then pass through an analog to digital converter then the signal is processed to the other two representation mentioned above.

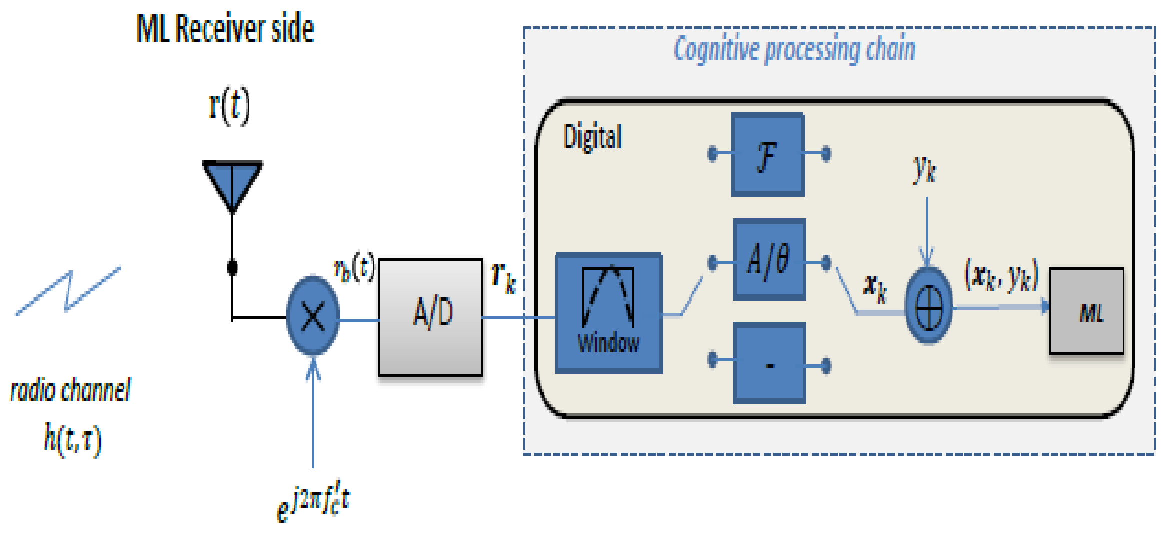


FIGURE 6: COGNITIVE PROCESSING CHAIN.

3.1.3 Multi path model

3.1.3.1 Multi path model idea and advantages

The Multi path model is inspired from the fact that the more ways the data is represented in, the higher chance the model correctly classify it, as some representations are more suitable to some kind of data which will easily be detected by the model if it is in this representation.

The multi path model extracts more features from the data set as it processes three different data representations at the same time allowing single branch optimization, also has a huge size reduction.

This multi path model proposed by us which we named PSI (Ψ) model, as it is shaped like the Greek letter, is based on the three different data representation of the data set RadioML 2016.10a Modulation which are the three-representation used by Kulin [4] The model is formed of three unique branches each branch is optimized separately for each one of the three different representations.

PSI model is the first multi path model in modulation recognition based on convolutional neural network.

3.1.3.2 PSI (Ψ) model Structure

The model is formed of three unique branches each branch is optimized separately for each one of the three different representations as shown in Figure 7.

The first branch is optimized for the In-phase and quadrature representation, the second branch for the Amplitude and phase representation and the third branch is for the Frequency representation. All of the three branches are optimized in two-way operation. first each of the three paths is separately optimized then an overall optimization is done to the whole model.

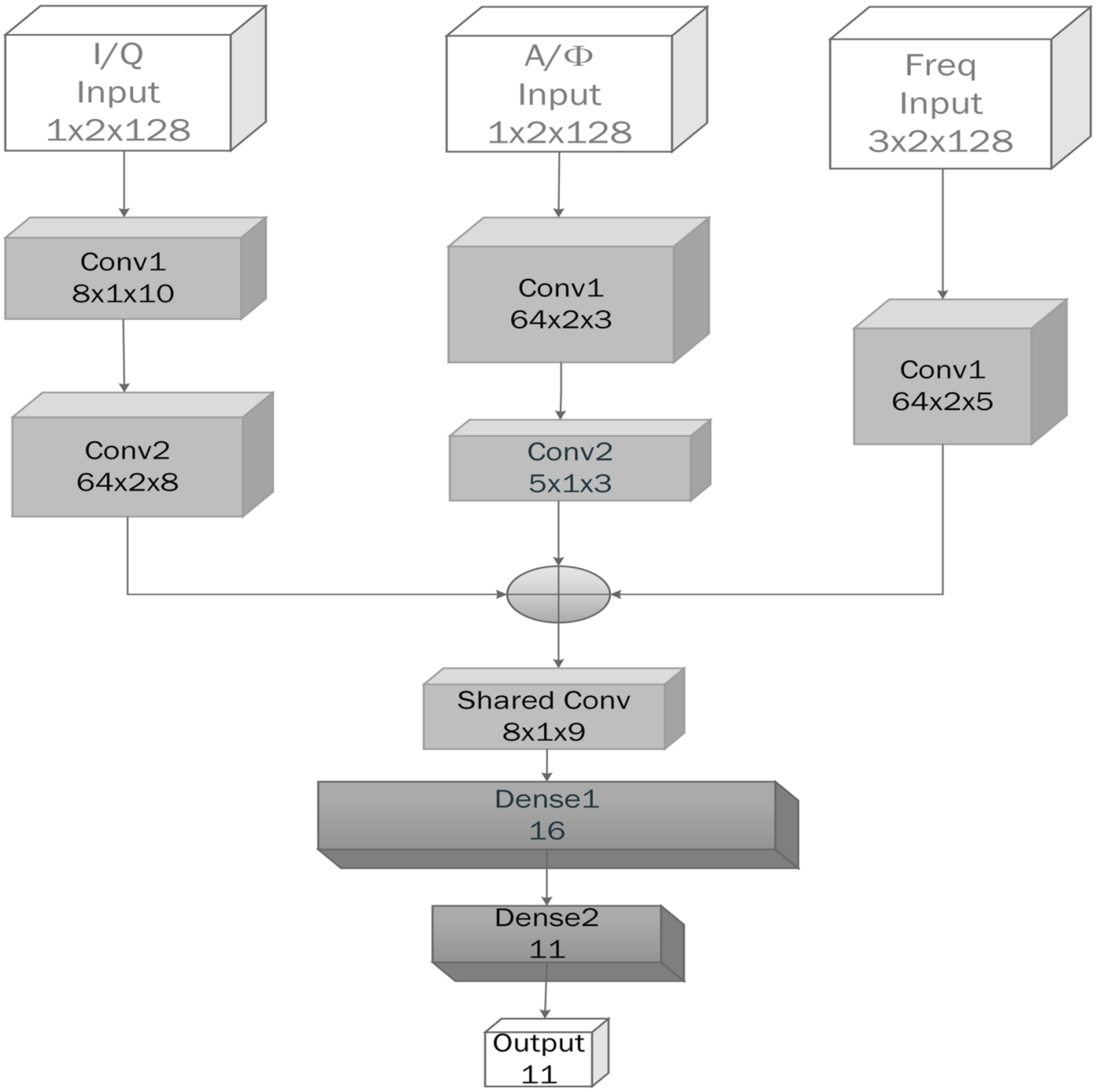


FIGURE 7: Ψ MODEL BLOCK DIAGRAM.

As listed in Table 2 the model consisted of three paths with different layers in each path The first path/branch is the IQ path which consists of 2 convolutional layers, the first consists of 8 filters each of size (1x10) and the second consists of 64 filters each of size (2x8).

The second path is the Amplitude-phase path which consists of the same number and type of the IQ layers as it consists of 2 convolutional layers with the only change in the number of filters and the size of them the first consists of 64 filters each of size (2x3) and the second comes with 5 filters each of size (1x3).

The third path is the frequency path which has a single convolutional layer which consists of 64 filters each filter of size (2x5).

The results of the three paths are gathered together then pass through a shared convolutional layer consists of 8 filters each filter of size (1x9) which is cross optimized through the three paths for the best results of the different paths. This shared convolutional layer increased the overall model accuracy as it is considered as a model to the new set of data generated from a different representation results so the model can learn more features from it.

The results from the shared convolutional layer is then passed through 2 dense layers. the first consists of 16 neurons and the second consists of 11 neurons to classify the data into the 11 different modulation types.

The model structure listed in table 2 is optimized for modulation recognition. Later in this thesis an illustration to a modification, done to PSI model to improve its wireless interference performance, will be discussed in the wireless interference performance subsection.

TABLE 2: Ψ MODEL STRUCTURE

Branch	Layer type	Input size	Structure
IQ	Convolution	1x2x128	8 filters, filter size 1x10
	Convolution	8x2x119	64 filters, filter size 2x8
Amp/Phase	Convolution	1x2x128	64 filters, filter size 2x3
	Convolution	64x1x1126	5 filters, filter size 1x3
Frequency	Convolution	3x2x128	64 filters, filter size 2x5
Shared	Convolution	133x4x128	8 filters, filter size 1x9
Layers	Fully connected	1x960	16 neurons
	Fully connected	16x1	11 neurons

3.1.3.3 PSI (Ψ) model Performance

The model performance is measured on several bases which are the size of the model, the accuracy of the classification done by the model for a range of SNR form [-20: 18] dB and the speed of the model. The number of multiplications-additions will be used to measure and compare the speed of the model which is a good indication of model's speed as the multiplications-additions are the most time-consuming operations.

To clearly show the model performance, the model is compared to our reimplementation of the model proposed by Oshea [1] to compare between sequential model and multi path model. We entered the three data representation as done by Kulin [4].

In the reproduced Kulin model we got nearly the same performance in case of IQ, lower in case of Amp/Phase, and higher in case of Frequency relative to results announced by Kulin. The results of the reimplementation of the sequential model is compared to our proposed multi path model (PSI Model) to show that the multipath model is more efficient in all the performance bases discussed above than the single branch implementation (sequential model).

Table 3 shows the performance metrics comparison between the PSI model and Oshea's model with different representations which shows that the multi path model (PSI model) achieves higher performance in all SNRs levels than the sequential model (Oshea's model).

TABLE 3: PERFORMANCE METRICS COMPARISON FOR MODULATION RECOGNITION

Model	parameters	Multi-add (millions)	SNR	P-average	R-average	F1-score average
PSI	36,888	2.69	High	0.85	0.84	0.84
			Medium	0.82	0.81	0.81
			Low	0.39	0.40	0.36
IQ	2,667,615	20.59	High	0.79	0.78	0.77
			Medium	0.78	0.76	0.74
			Low	0.38	0.35	0.33
Amp/Phase	2,667,615	20.59	High	0.77	0.73	0.71
			Medium	0.65	0.63	0.61
			Low	0.20	0.14	0.10
Frequency	2,667,615	20.59	High	0.72	0.70	0.69
			Medium	0.70	0.69	0.67
			Low	0.37	0.34	0.31

Higher performance difference is shown in the medium and high SNRs as shown in Figure 8 which shows the difference between accuracy of the F1-Score between PSI model and Oshea's model with different representations in high SNR.

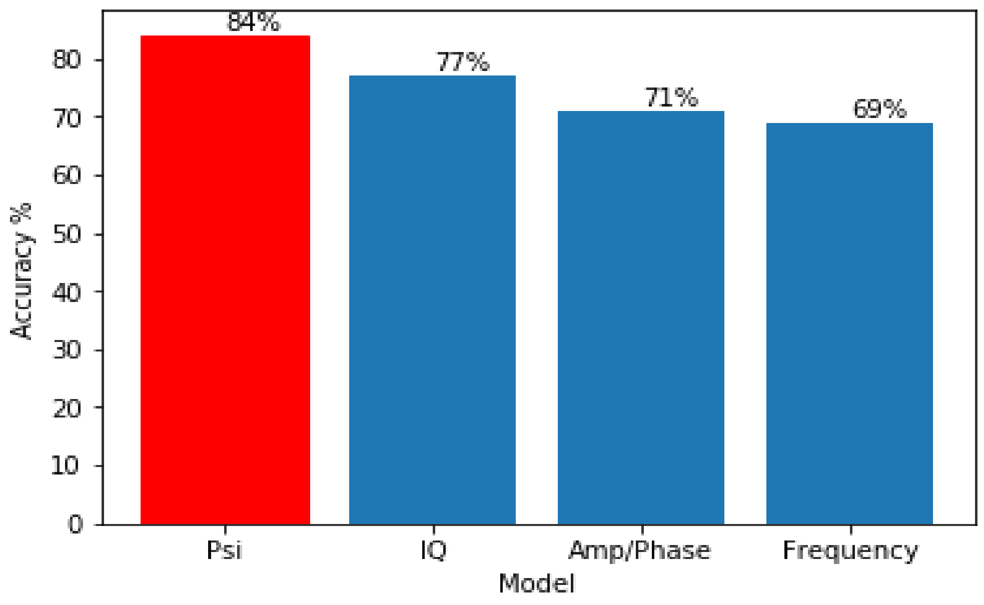


FIGURE 8: F1-SCORE PERFORMANCE COMPARISON.

Table 3 also show a tremendous reduction in the model size of about x72 as the PSI model is about 36,888 parameters compared to Oshea's model 2,667,615 parameters. Table 3 also shows the tremendous reduction in number of multiplications-additions in PSI model relative to Oshea's model as the number of multiplications-additions decreased to 2.69 million compared to 20.59 million in Oshea's model which lead to higher model speed compared to Oshea' model.

Figure 9 clearly show the tremendous reduction of the model size compared to oshea's model. This reduction is due to the reduction on the size of the used convolutional layers in the PSI compared to the used on Oshea's model as Oshea's model was meant to be used with image classification which requires high convolutional layers size.

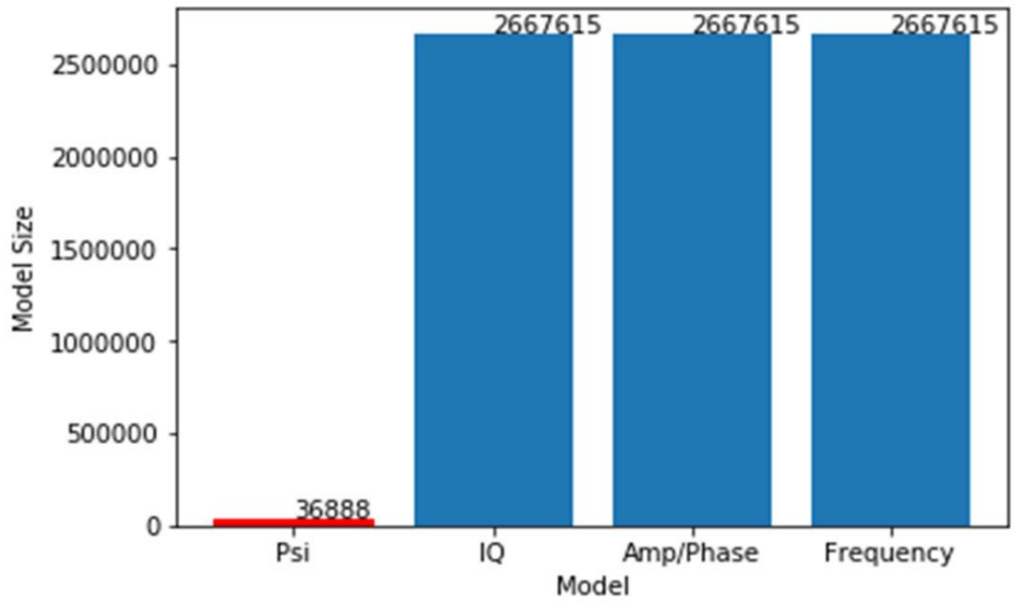


FIGURE 9: MODEL SIZE COMPARISON.

Figure 10 shows that the multi path model (PSI model) achieves higher accuracy in all SNRs than the sequential model (Oshea's model).

As shown in this figure at very low SNRs the accuracy is almost the same for all models as the signal power is very low relative to noise power so the models couldn't detect it as the SNR value increases the PSI models starts to outperforming other models and at high SNRs, the PSI model has good increase in the accuracy than any other models.

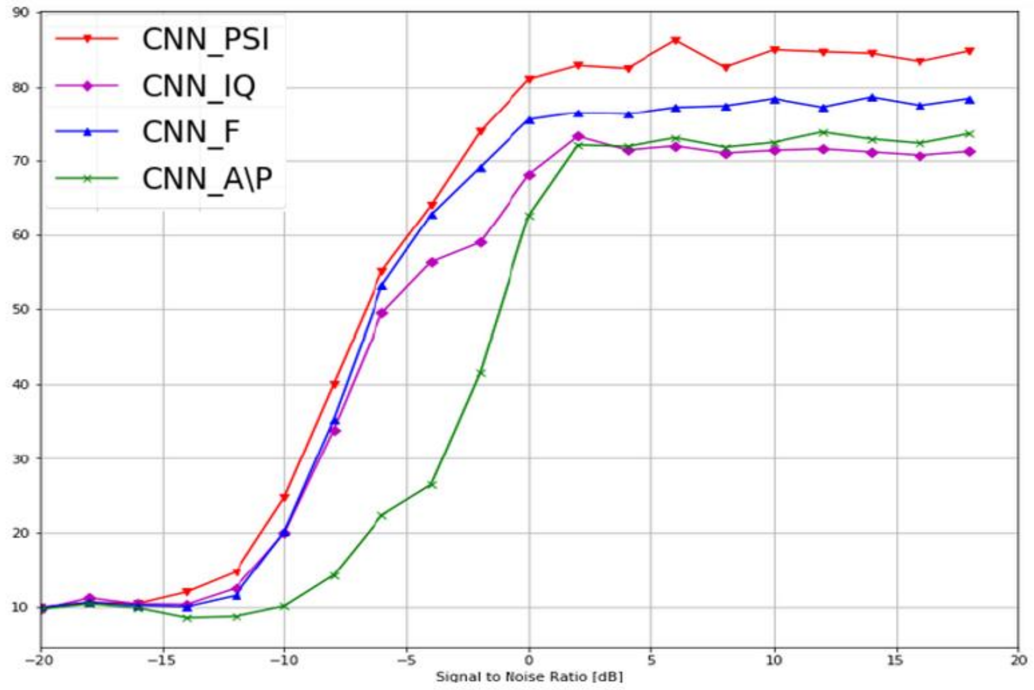


FIGURE 10: ACCURACY COMPARISON FOR MODULATION RECOGNITION.

This slightly higher accuracy, performance metrics values, and small size came from when a certain CNN path from the three paths gives low accuracy estimation for modulation type due to similarity in a certain data representation, the two other paths give higher estimation due to the non-similarity in their domain, so overall the model gives correct prediction. So, the model noise immunity had increased and this reduced error, increased accuracy, and enabled us to reduce number of parameters also. To identify which modulation type PSI model gets better prediction than O'Shea model and which it gets worse, this subsection is concluded with comparing between confusion matrix of Psi model and O'Shea model with different input data representation at SNR value 6dB. Figure 11 shows the confusion matrix of PSI model which shows that PSI gets worst accuracy in QAM16 and QAM64 relative to Amplitude and Phase shown in Figure 13 domain but better than IQ and frequency representations, PSI also misclassify AM-DSB as WBFM which is not at any other domain, but it eliminates misclassification of IQ ,Amplitude and phase and frequency, shown in figure 12 , 13 and 14 , for QPSK as 8-PSK, it also has less misclassification for WBFM as 8-PSK exists also in all data representation, also it can be noticed that it eliminates misclassification of 8-PSK as QAM64 that exists in all data representation with different degrees. Finally, we notice that if the application has expectation at medium SNR that QAM16 and QAM64 and AM-DSB will be dominant, Then Amplitude and phase domain will fit your problem better than PSI, Otherwise PSI will be better for your application.

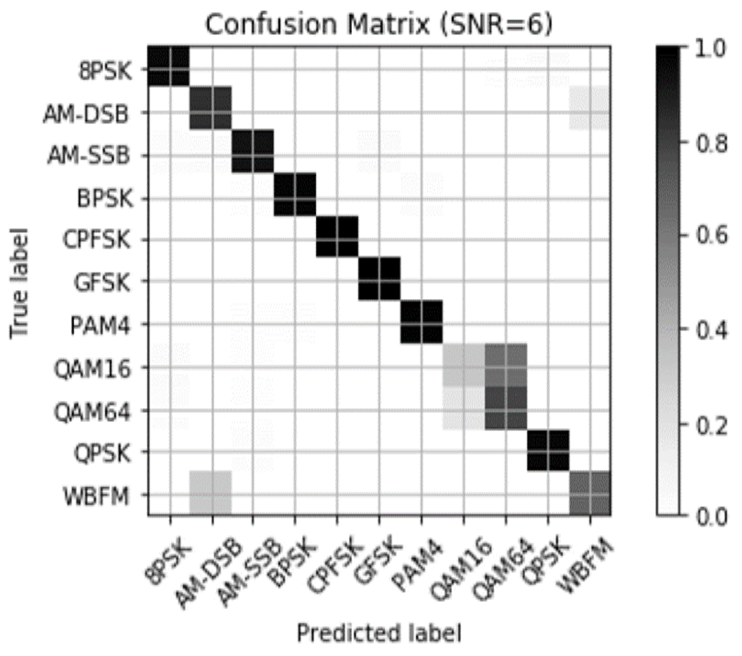


FIGURE 11: PSI WITH SNR 6DB.

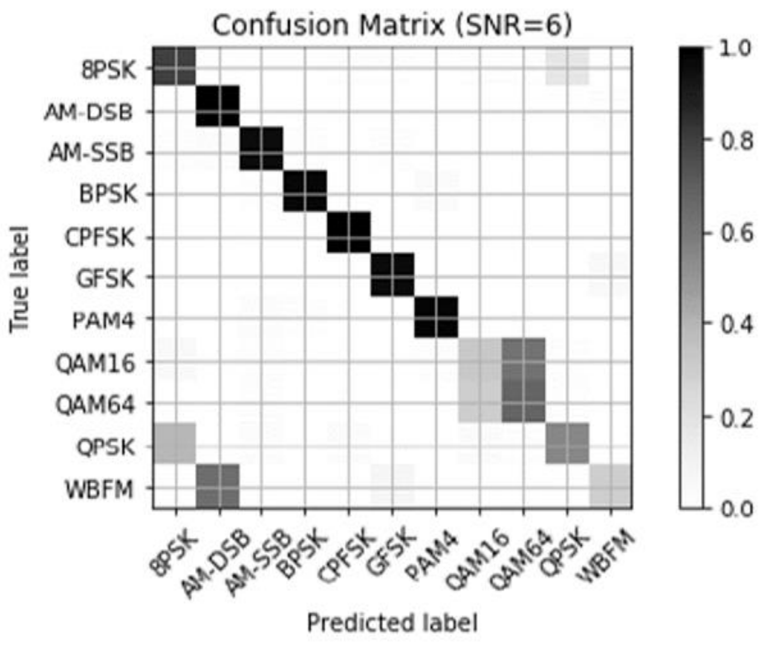


FIGURE 12: IQ WITH SNR 6DB.

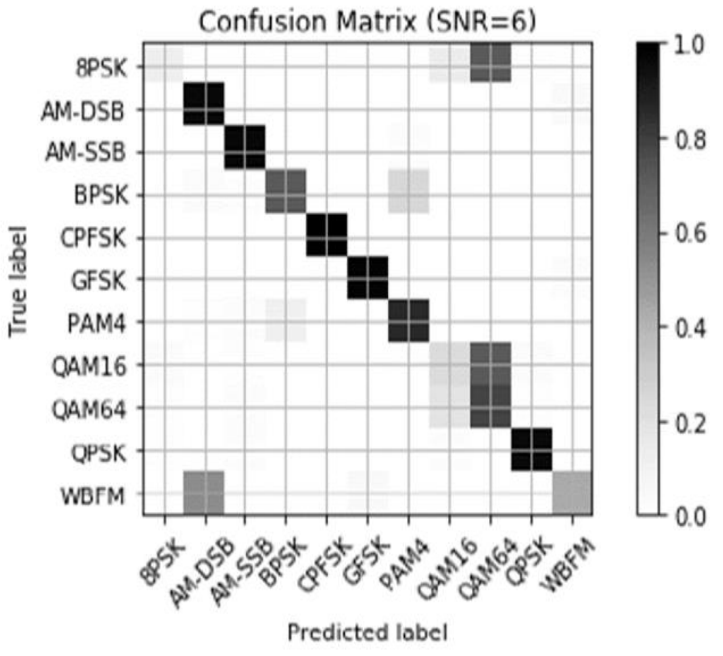


FIGURE 13: AMP/PHASE WITH SNR 6DB.

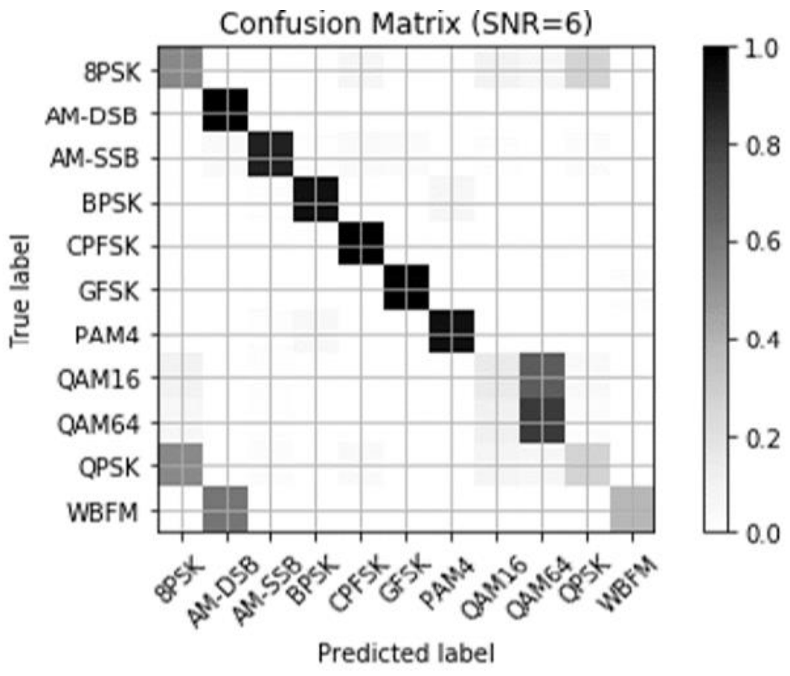


FIGURE14: FREQUENCY WITH SNR6DB.

3.1.4 Optimized IQ model

3.1.4.1 Model Introduction

This model targets modulation recognition problem, Optimized IQ developed from IQ branch of psi to be implemented on FPGA. The size of the model is an important feature as a low size model will have relatively low number of calculations which will highly decrease the cost of the device needed for the implementation. At the same time, it was very important to maintain accuracy as high as possible, the small size of the model also increased the model classification speed due to the reduction on the number of calculations. The model consists of 2 convolution layers and 2 dense layers. The first convolution layer has 45 filters each of 2x8 parameters, the second layers has 9 filters each of 1x6 parameters, the first dense layer has 32 neuron, and the second dense layer has 11 neuron for the classification purpose to classify the results to the 11 modulation types. Model architecture is shown in Figure 15.

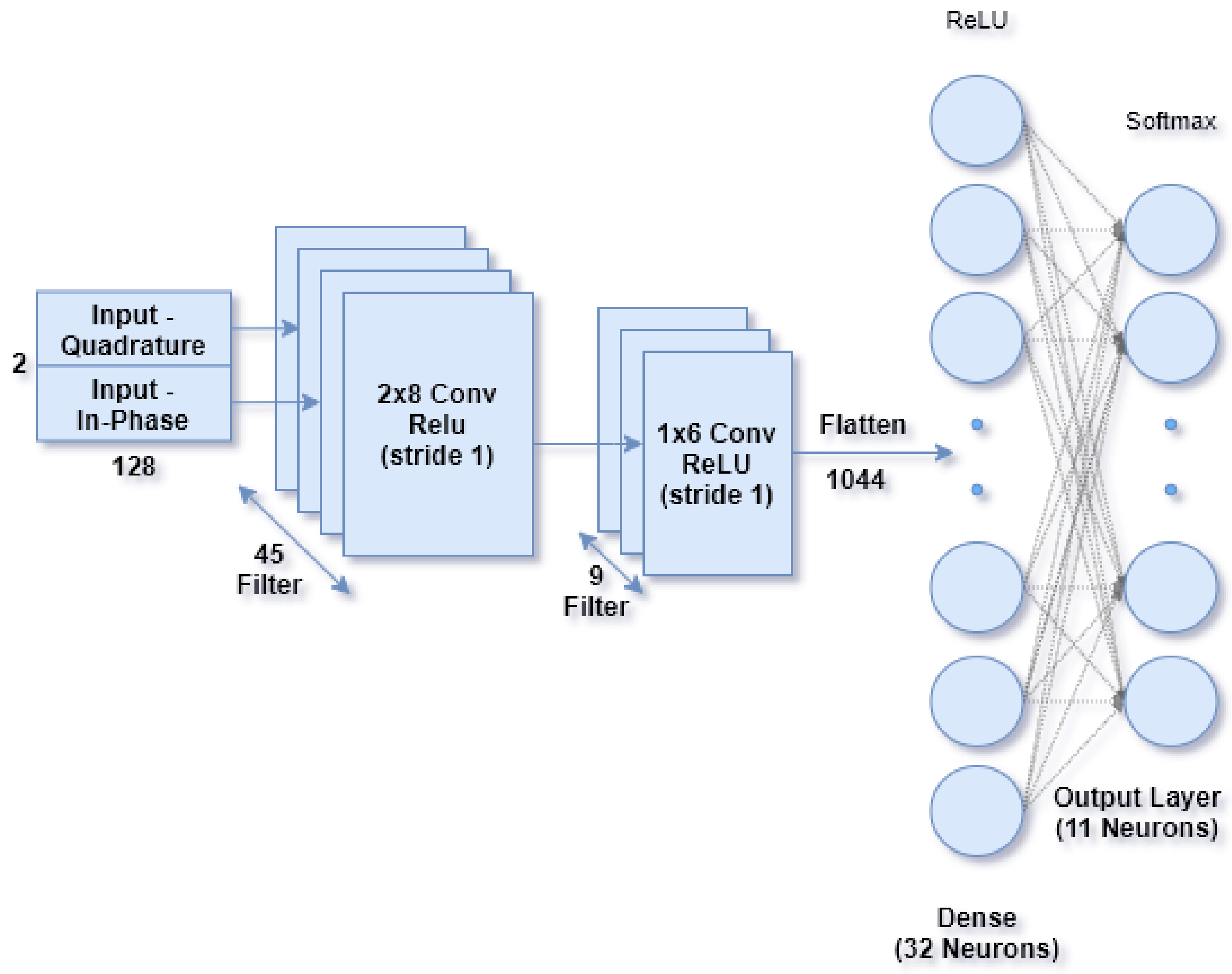


FIGURE 15: PROPOSED IQ MODEL ARCHITECTURE.

3.1.4.2 Training Model

Model trained on Kaggle website using Keras framework with Batch size 512 and 200 epochs. Biases were turned off to facilitate fixed point operation. Adam optimizer was used with learning rate 0.001, validation accuracy reached 54.38%.

3.1.4.3 Results

In this subsection classification accuracy of the proposed model will be discussed in details but to get better sense of the results we will compare the results with another paper [4]. Kulin's paper introduced three data representation techniques for oshea's model. One works on data (IQ) without any preprocessing as our proposed model, the second method transfers data to amplitude / phase representation and last one transfer data to frequency representation.

3.1.4.3.1 ACCURACY CURVES

Classification accuracy curve over different SNRs of proposed model shown in Figure 16 and the classification accuracy curve of Kulin's paper for the three representations can be seen in Figure 17 [4]. We can notice from SNR -20 to -10 dB there is no significant difference between proposed model and Kulin's three representations. From SNR -10 to 5 dB IQ representation in Kulin's model took the advantage and become the highest accuracy compared with the other two representations ,when compared with our proposed model we will find they are much alike, from SNRs after 5 dB amplitude / phase representations gets higher accuracy compared to the other two representation in Kulin's model but when compared with proposed model we can find that the proposed model slightly have higher accuracy peaking at SNR 10 dB with accuracy of 84%. We can conclude now that the proposed model implemented on FPGA succeeded in getting high accuracy all over different override all three representations proposed in Kulin's paper.

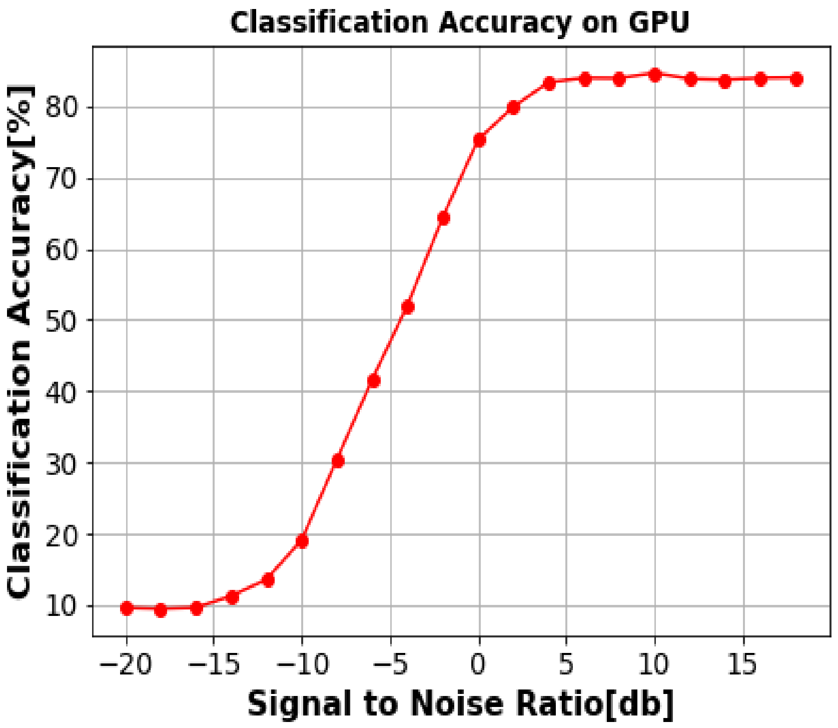


FIGURE 16: ACCURACY OF PROPOSED MODEL.

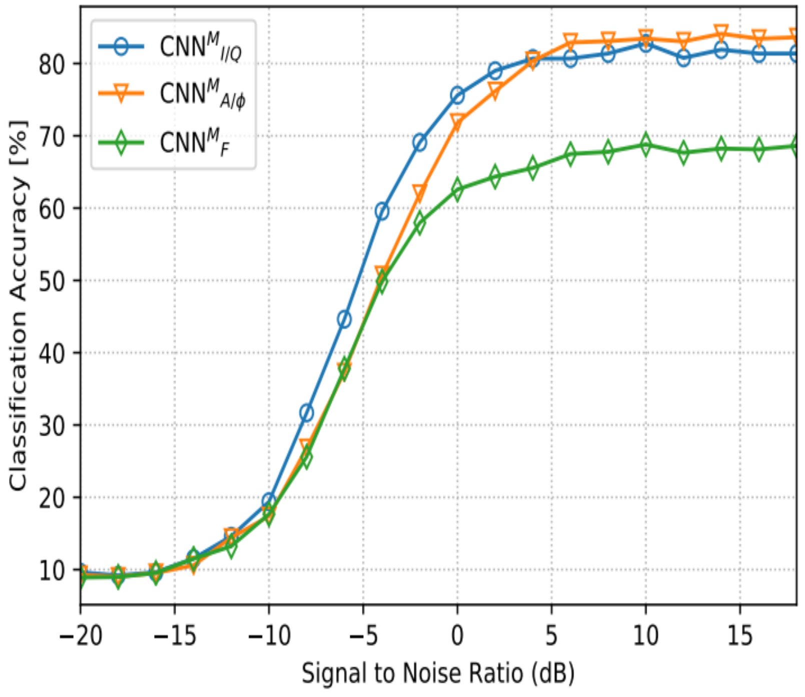


FIGURE 17: ACCURACY OF KULIN'S MODEL.

3.1.4.3.2 PERFORMANCE METRICS

Numerical results is a more detailed way to compare between the two models. table I and table II shows precision, recall, and f1 score at high SNR (18dB), medium SNR (0dB) and low SNR (-8dB) for the proposed IQ model and the three model's in Kulin's paper [4], comparing proposed model implemented on FPGA with IQ model in Kulin's paper it's obvious that at high and medium SNRs the proposed model has much better F1 score compared.

to Kulin's paper [4] however in low SNR Kulin's representation gives better F1 score as the precision of proposed model in low SNR is very low which affected the F1 score, F1 score of the amplitude and phase representation become equal compared with the proposed model at high SNR but in medium and low SNRs the proposed model is much higher, comparing with frequency model in kulin's paper.

TABLE 4: PERFORMANCE METRICS OF KULIN'S PAPER

Model	SNR	Precision	Recall	F1 Score
Proposed CNN_{IQ}	High	0.84	0.86	0.82
	Medium	0.76	0.77	0.74
	Low	0.31	0.33	0.27
CNN_{IQ}	High	0.83	0.82	0.79
	Medium	0.75	0.75	0.72
	Low	0.36	0.32	0.30
$CNN_{A/\Phi}$	High	0.86	0.84	0.82
	Medium	0.70	0.70	0.69
	Low	0.33	0.29	0.26
CNN_{F}	High	0.71	0.68	0.67
	Medium	0.63	0.60	0.59
	Low	0.28	0.25	0.22

3.1.4.3.3 CONFUSION MATRIX

Confusion matrix will help to better understand and visualize classification accuracy and which classes causes wrong predictions. We will compare confusion matrix of the proposed model with the IQ and Amplitude / Phase representations in Kulin's paper [4]. Frequency representation won't be considered as it's obvious from previous scores that frequency gives very bad accuracy and scores so its confusion matrix will be messed up. Comparing proposed model confusion matrix shown in Figure 18 with IQ model in Kulin's paper Figure 19 we can see that miss classification between QPSK and 8PSK in Kulin's model improved a lot in proposed model but the both still miss classify WBFM with AM-DSB and QAM16 with QAM64. comparing proposed model confusion matrix in Figure 18 with Amplitude / Phase Model in Kulin's paper in Figure 20 we can find that Amplitude / Phase improved miss classification in QAM16 with QAM64 but didn't improve miss classification between QPSK and 8PSK and a new miss classification showed up between 8PSK and QPSK which gives advantage to the proposed model compared with Kulin's models.

3.1.4.3.4 NUMBER OF CALCULATIONS AND MODEL SIZE

Number of calculations performed by model and number of model's parameters are considered the biggest advantage of the proposed IQ model, number of operations interpreted on FPGA as multipliers and adders, number of parameters interpreted as flip flops and LUTS on FPGA. Table 5 shows how much the parameters and calculations are reduced of proposed model compared to Kulin's model [4], parameters almost reduced by x 72 and number of million multiplications - Additions reduced by x 46 which made the proposed IQ model very efficient and suitable when implemented on FPGA and if implemented on IC operating in low power applications such as LUTs and embedded devices.

TABLE 5: PERFORMANCE METRICS OF PROPOSED MODEL ON GPU

Model	**Parameters**	**Million Mult-Adds**
Proposed IQ Model	**36,910**	**0.443**
Kulin's Model	**2,667,615**	**20.59**

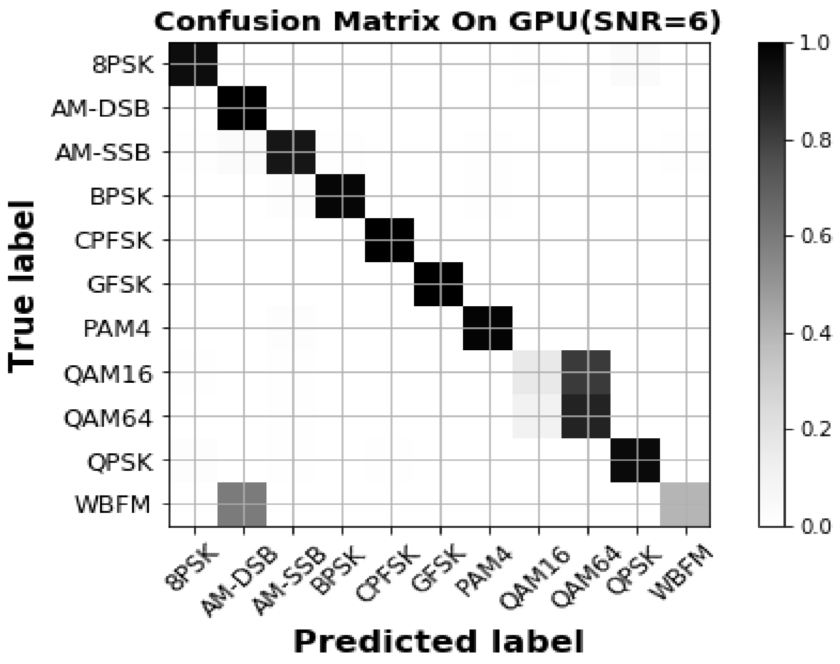


FIGURE 18: CONFUSION MATRIX OF PROPOSED IQ MODEL.

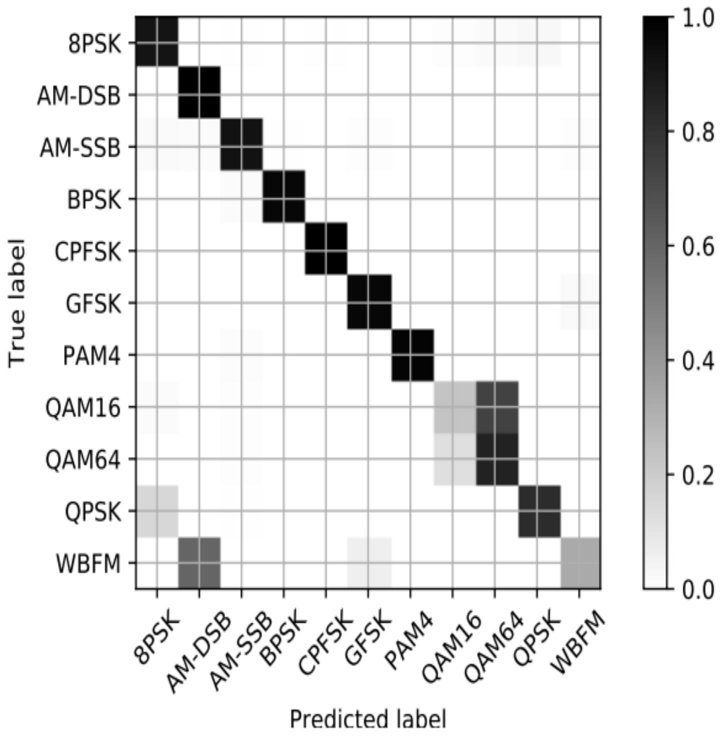


FIGURE 19: CONFUSION MATRIX OF KULIN'S IQ MODEL.

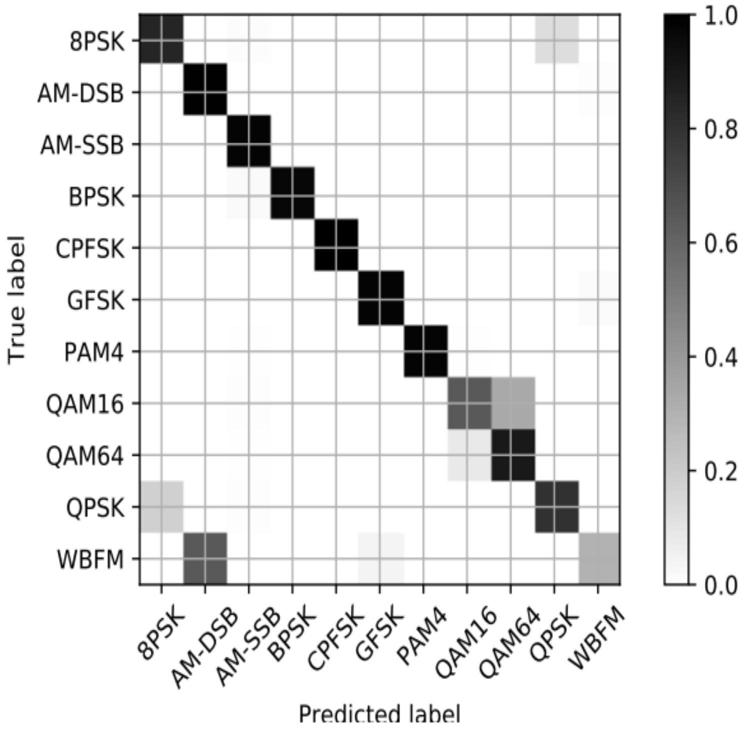


FIGURE 20: CONFUSION MATRIX OF KULIN'S AMP / PHASE MODEL.

3.2 Wireless Interference

3.2.1 Introduction

As mentioned in the introduction section, Wireless Interference Identification (WII) targets classifying signals in the Industrial, Scientific, and Medical (ISM) Band. Schmidt et al.[3] divided the bandwidth into 15 classes: ten classes for IEEE 802.15.1(Bluetooth), two classes for IEEE 802.15.4 (ZigBee), and three class for IEEE 802.11 (Wi-Fi). To make the signal within the desired bandwidth, in WII 10 MHz, he expanded Wi-Fi signal on three different snapshots as Wi-Fi signal takes bandwidth bigger than desired one as shown Figure 21.

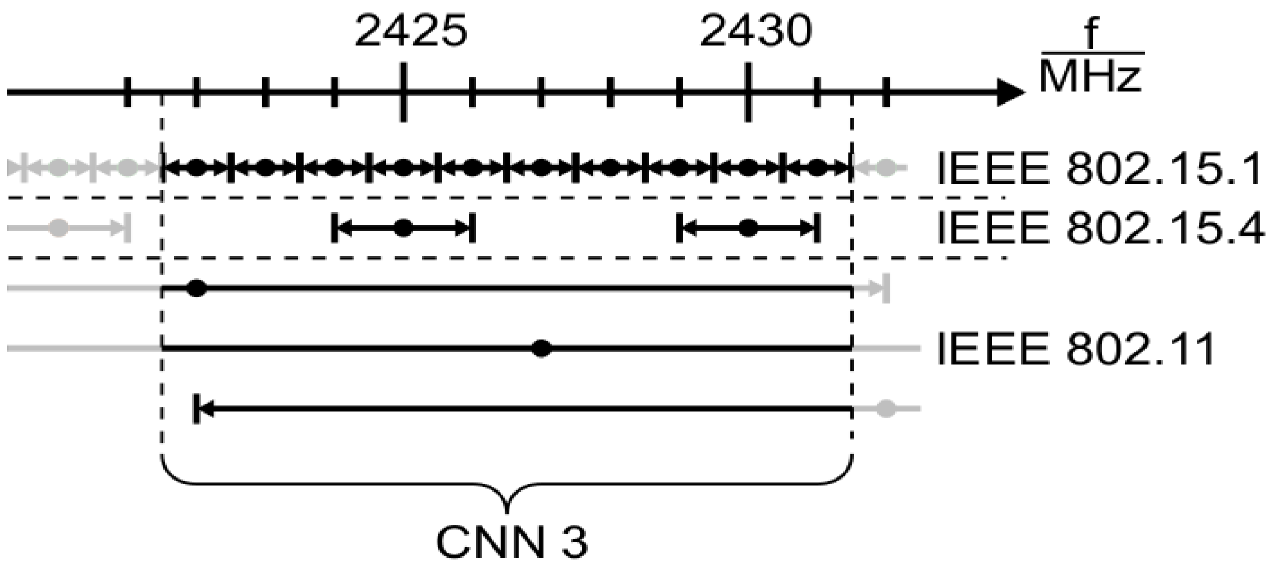


FIGURE 21: FREQUENCY CHANNEL CLASSES OF EXAMPLE CNN IN THE 2.4GHZ-ISM-BAND [3].

To get more intuition about the dataset, we plotted them as shown in Figure 22. Where y-axis represents Quadrature-Branch and x-axis represents In-Phase-Branch. Each subplot represents one of the 15 classes in the dataset. Dataset consists from 225,225 Symbols; each symbol represents one of the 15 classes and consists from 2x128 samples. The 225,225 symbols are distributed on a 21 distinct Signal-to-Noise (SNR) range from [-20 dB : 20 dB] with step size of 2 db. Where each SNR step has 715 symbols. So, the final shape of the used dataset in WII is (15, 21, 715, 2, 128)[1].

[1] Multiplying 15x21x715 will give the 225,225 which represents the total number of symbols, whereas 2x128 is data samples

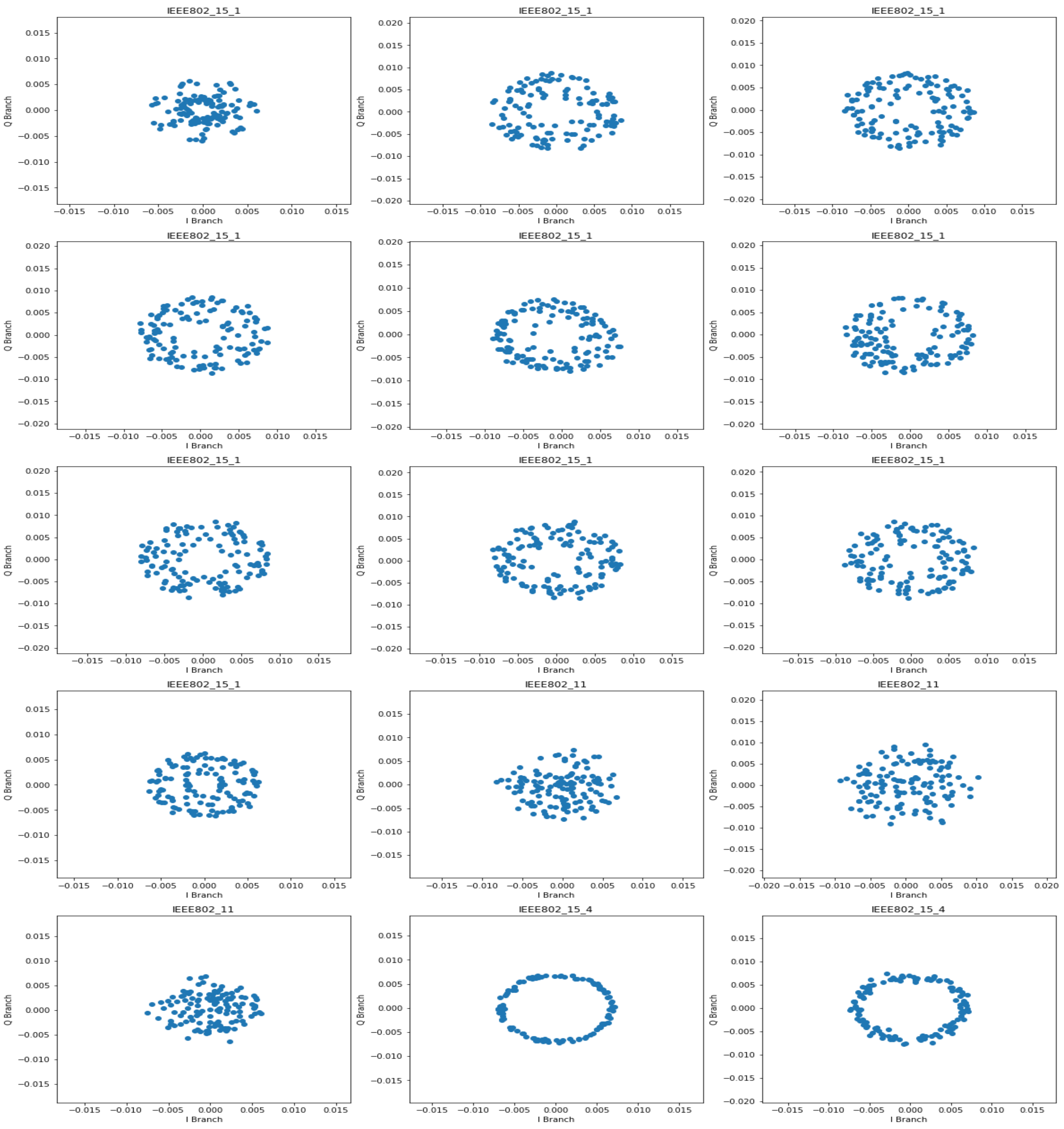


FIGURE 22: WII DATASET REPRESENTATION.

3.2.2 Literature of wireless interference identification

The first work in the wireless interference is done by Kulin et al.[4] using the same model produced by Oshea et al. [1] for modulation recognition using also the same data set used mentioned in this section.

3.2.3 PSI model for wireless interference

3.2.3.1 Introduction

The same PSI model used in modulation recognition is used for wireless interference identification except when we made optimization we could remove shared fully connected layer which was 16 neurons in modulation recognition and instead of 1x10 and 2x8 which was in IQ branch, we change it to 2x10 and 1x8 which gives faster response as the dimension that enter second convolutional layer became 1x8 became 1x119 instead of 2x119 which lead to smaller number of multiplications and learnable parameters which leads at end to smaller neural network.

3.2.3.2 Performance comparison

To compare between O'Shea model with different input data representation and sequential model in wireless interference identification, we again reimplemented the model proposed in by Oshea [1], as in the modulation recognition case. the reimplemented model get higher values for performance metrics in case of IQ and Frequency but lower values in case of Amplitude and Phase than the values in announced by Kulin [4].

Figure 23 shows that multi path PSI model achieves higher accuracy in all SNRs relative to the sequential model with any data representation.

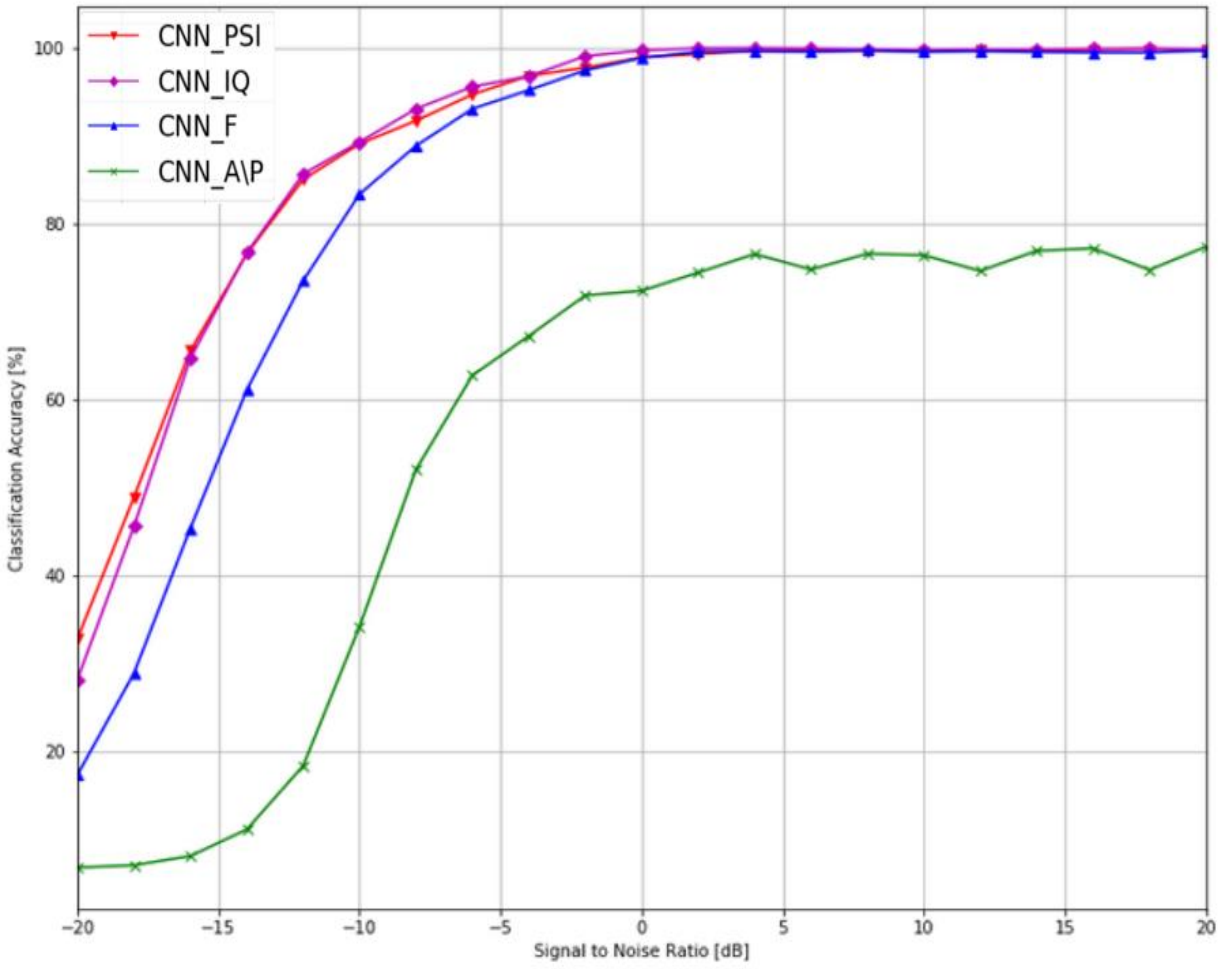


FIGURE 23: ACCURACY COMPARISON FOR WIRELESS INTERFERENCE IDENTIFICATION.

To ensure the previous results, a comparison is done between performance metrics including precision(P), recall (R) and F1-score in wireless interference identification between multipath model and sequential model. the comparison results is shown in Table 6 which shows that Psi model has higher performance metrics values in all SNRs than IQ & Amplitude and Phase sequential models and the same or slightly less than Frequency sequential model implementing O’Shea Model, also it shows huge reduction on the number of parameters relative to O’Shea model, and a smaller number of multiplications-additions which implies higher speed than Oshea’s model.

Table 6: Performance metrics comparison for Wireless Interference Identification

Model	parameters	Multi-add (millions)	SNR	P-average	R-average	F1-score average
PSI	36,888	2.69	High	1.00	1.00	1.00
			Medium	0.99	0.99	0.99
			Low	0.92	0.92	0.92
IQ		20.59	High	0.99	0.99	0.99
			Medium	0.99	0.99	0.99
			Low	0.90	0.89	0.89
Amp/Phase	2,667,615	20.59	High	0.77	0.76	0.74
			Medium	0.74	0.72	0.72
			Low	0.51	0.51	0.51
Frequency	2,667,615	20.59	High	1.00	1.00	1.00
			Medium	1.00	1.00	1.00
			Low	0.94	0.93	0.93

3.2.4 PSI lite model for wireless interference

When trying to implement PSI, explained in previous chapter, in the hardware domain, in our case FPGA, we faced two problems:

- **First**: Phase Branch is the output from dividing Quadrature (Q) branch over In-phase (I) branch and taking arctan for it (shown in Equation (19)). As will be explained in the Fixed-Point Representation Section, this arctan operation will make the output's range within $[\frac{-\pi}{2}: \frac{\pi}{2}]$. So, it will need a bigger number of bits relative to the other branches which make the whole model bigger than blade RF size[2] [3].

- **Second**: Frequency Branch needs complex circuit which will utilizes big part of resources to be implemented.

$$phase = arctan\left(\frac{X_q}{X_i}\right) \quad (19)$$

So, to solve the two problems stated above, we dropped The Frequency branch and Phase part from A/Ɵ branch to facilitate an initial trial, moving the dropped branches to the future work. The final model shape is shown in Figure 24 and data flow between layers is shown in Table 7 We call this structure Psi-Lite as it was built upon Psi where consisting from two branches I/Q branch and Amp branch, also it is much smaller than Psi and much faster.

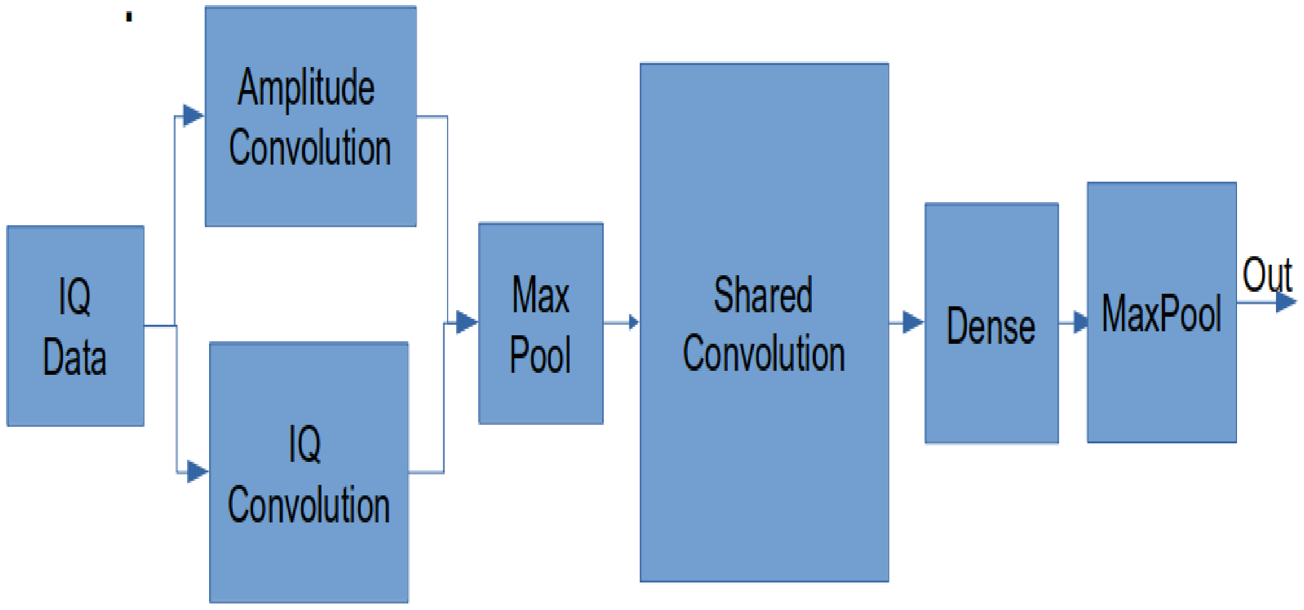


FIGURE 24: MULTI PATH CNN MODEL (PSI-LITE), BLOCK SIZE PROPORTIONAL WITH ITS OPERATIONS.

[2] It's a device for software defined radio that is used as transmitter and receiver which facilitate making real-time experiment.

Table 7: Psi Lite Data Flow

Layer name	Input size	Output size	Weights
Amplitude Convolution	1x2x128	10x1x122	56
IQ Convolution	1x2x128	8x1x122	140
Max-Pool	18x1x122	18x1x61	0
Shared Convolution	18x1x61	10x1x57	900
Dense	570	15	8550
Max-Pool	15	1	0

The model share most of its layers with Psi, but there was an additional layer which is a Maximum Pooling layer[3]. Maximum Pooling layer (a.k.a. Max-Pool[4]) takes 'n' inputs compare between them to find the maximum value, doing this reduce the size of the model (e.g. taking Max-Pool 2x2 from the input features[5] will reduce the size of Max-Pool's output to 1/4. Pooling in Psi-Lite should be 1xn as input to Max-Pool is 1D data. Max-Pool layer in Psi-Lite is 1x2 which means it takes two samples, compare between them, remove the smaller one, and forward the biggest value to the next layer. Max-Pool is shared between IQ and Amplitude branches.

Although it takes two separate samples from each branch, at any instance it has four samples, but as it remove half of its inputs and has a speed double the blocks before it, so it can be shared between them, e.g. when circuit start working Max-Pool takes the first sample from Amplitude Convolution branch[6], at time t0, then start taking samples simultaneously from I/Q Convolution branch and Amplitude branch, at time t1, t2, ….., (there is a deliberately delay one clock between the two branches at the start of any new sample to make the protocol works properly). We can note that at time t1, it has two samples from Amplitude branch and only one sample from IQ branch then it could compare between the two samples, selecting the max between them and feed this value to the shared convolution and clear the Amplitude buffer.

At time t2 it has two samples from IQ branch but one sample from Amplitude branch, so it compares values for IQ branch and remove the max and this operation repeats. To clarify its operation more, we inserted samples at Table 8 to show Max-Pool operation where we assume, for simplicity, that any sample comes is bigger than its previous one (i.e. A1> A0, A3>A2, and A5 > A4).

[3] In the following context, we mean with maximum pooling the medium layer not the last, the last layer targets choosing the Output not reducing size or overfitting

[4] This notation is used in Py-Torch Library.

[5] Any input/output from convolution layer called input/output features as the main target of convolution layer is finding features from the data to be used in distinguishing different classes/categories/objects.

[6] The reason for starting with Amplitude branch not IQ branch is that Amplitude branch consists only from one data row not two which makes it faster to load it from memory and makes its output faster, so instead of buffering all its output until IQ branch output be ready, we buffer all except the last one and feed the first buffered element into the Max-Pool Circuit.

TABLE *8*: CLARIFYING MAXIMUM POOLING LAYER OPERATIONS

Time	t0	t1	t2	t3	t4	t5
Amplitude buffer	A0	A1	A2	A3	A4	A5
IQ buffer	None	IQ0	IQ1	IQ2	IQ3	IQ4
Max-Pool output	None	None	A1	IQ1	A3	IQ3

To emphasize on the effect of Max-Pool Layer in Psi-Lite, we inserted in Table 9 Flow of data in case Max-Pool doesn't exist, the last Max-Pool used in finding the number of the class, so it's mandatory. Comparing Table 7 with Table 9 shows the effect of The Max-Pool Layer whereas in Table 9 there is a big input to Dense layer which leads to a big number of weights leading also to a big memory footprint.

TABLE *9*: PSI-LITE DATA FLOW WITHOUT MAX-POOL LAYER

Layer name	Input size	Output size	Weights
Amplitude Convolution	1x2x128	10x1x122	56
IQ Convolution	1x2x128	8x1x122	140
Shared Convolution	18x1x122	10x1x118	900
Dense	1180	15	17700
Max-Pool	15	1	0

Although Psi-Lite has small size, but it also has a good accuracy relative to other models such as Kulin's IQ and A/Θ [9], but as WII dataset was generated in the frequency domain, so any model can generalize and get high accuracy in the frequency domain. Figure 25 shows that Psi-Lite has a bigger accuracy than any model not working with data in the frequency domain.

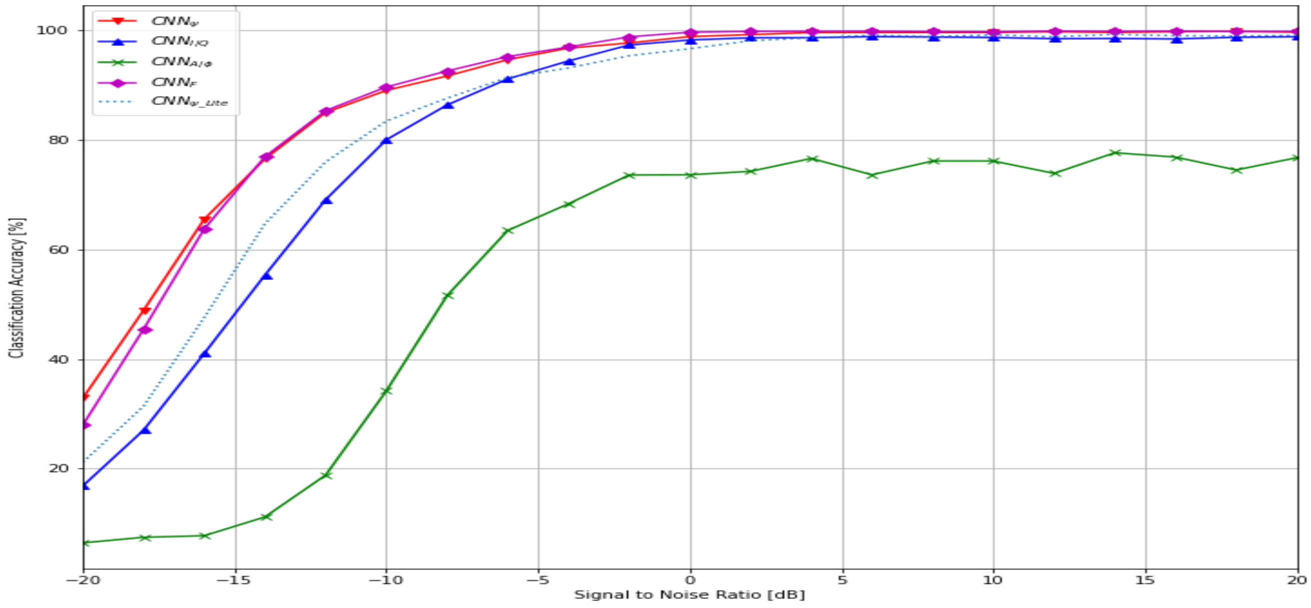


FIGURE 25: PSI-LITE ACCURACY.

3.3 Software work summary

In this section we explained different solution to the modulation recognition and wireless interference identification problems as PSI, optimized IQ, and PSI-Lite models showing their layered architecture and data flow. Finally,Table 10 shows a comparison between Literature and software models proposed in this chapter.

TABLE 10: SOFTWARE COMPARISON

Model Name	target[7]	Accuracy	Size	Million Mult-Adds
Oshea IQ	WII	84.3	2,667,615	20.59
Oshea A/Ɵ	WII	56.2	2,667,615	20.59
Oshea F	WII	89.21	2,667,615	20.59
Psi	MR/WII	56.2/89.25	36,888	2.69
Optimized IQ	MR	54.38	36,910	0.443
Psi-Lite	WII	84.5	9,646	0.095

[7] Modulation Recognition Empty Entries weren't provided in Literature

3.4 Fixed Point Representation

3.4.1 Implementation Explanation

Deep Learning Tools use floating point representation for weights as computers support floating point operations, allowing for a better accuracy and search space for weights than fixed point representation, but to move from Software Domain to Hardware Domain, we need to convert weights from floating point into a Fixed Point as FPGA doesn't support Floating point operation except with additional circuits.

Our target in weights conversion is finding the minimum number of bits that represents weights in the fixed-point domain without any loss in the accuracy.

Solovyev et al's Paper [9] proposed a technique for this low bit's conversion. To make this conversion we need to constraint values from each layer to prevent overflow, i.e. making input/output to/from any layer between [-1, 1], this can be guaranteed if weights between [-1, 1] and inputs between [-1, 1] then multiplying input with weights will also be between [-1, 1].

we are imposed with bits of input[8] as we have control only on the weights not the input, but this will not increase size and power[9] dramatically as size of inputs is in range of hundreds, it's 2x128 in the project dataset, but size of weights is in range of thousands and if model big it will be millions and in some cases it reaches billions,in our case it's 36,888 in Psi and 36,910 Optimized IQ.

So we take input and divide by its number of bits to be in range [-1, 1], but if we made the range of weights between [-1, 1] by taking the biggest value of weights and divided the weight with it, this will guarantee that all weights' values between [-1, 1] but this doesn't guarantee that layer's output is between [-1, 1].

So in [9] they proposed flowing input through the model's layers and taking the absolute of the biggest output from each layer, as the negative value may be bigger than positive value, and divide weights by this biggest output. This will guarantee that output from the current layer is between [-1, 1] and consequently input for the next layer is between [-1, 1].

This operation is completely equivalent to the floating point operation although we change the values from each layer until the maximum layer but we kept the maximum

[8] In case of bladeRF the input is 12 bits

[9] Increasing number of bits will increase number of bits multiplication which consequently increase required power

value relative to other which is what is importance for making the prediction, as last Max-Pool layer choose the maximum value relative to other values regardless its Magnitude.

The next step will be quantizing the weights values because until now we didn't convert weights[10] to fixed point representation which is the target. In this step, we need to quantize weights without affecting the accuracy and this achieved with try and error. To mathematically formulate this we can assume input i = $ax2^N$ and weights w = $bx2^M$ where 'a' and 'b' are values between [-1, 1], 'N' is imposed by the A/D in the stage before the FPGA, and 'M' is what we target to find in this quantization step.

Assuming that 'N' and 'M' is now known and in the last step we need to make shifting. First, we are going to explain why we need shifting then discussing the shifting operations method.

To implement convolution and dense Layers, we need to make two operations: Multiplication and Addition. Multiplication: in floating point

z = a x b but in fixed point Z_{bits} = I x w = ($ax2^Nxbx2^M$) so we have three cases:

1. N = M so zbits $_=$ ($axbx2^{2M}$)
2. N > M and in this case, we need shift weights (shift left) in the first layer to be the same number as N so $z_{bits\,=}$ ($axbx2^{2N}$)
3. N < M and in this case, we shift input (shift left) after it enters FPGA so $z_{bits\,=}$ ($axbx2^{2M}$) we can consider $z_{bits\,=}$ ($axbx2^{2M}$) but we need to constrain it within 2^M range, so it doesn't overflow as after the next layer it will be 2^{3M} and this shows needing for shifting operation. Addition: z = a+b and we constrain value of 'z' between [-1, 1] in the previous steps, but in floating point z_{bits} = i+w = ($ax2^N$ + $bx2^M$), and by using the same approach as above, ($ax2^M$ + $bx2^M$) = ((a+b)$x2^M$) = $zx2^M$ and that is the target value.

There are two ways to make shifting explained in [9], either shifting after each multiplication operation, or shifting after finishing the convolution operation. As addition is linear operation so it will not affect the result.

3.4.2 Visualization

We will try to visualize the operation of fixed-point in this subsection, assuming that we have to classify data into two classes/categories:

- **Get Data**: Data that you need to train your model with, assuming random data to visualize with it.

[10] We need to quantize weights only as input came from A/D and quantized by default

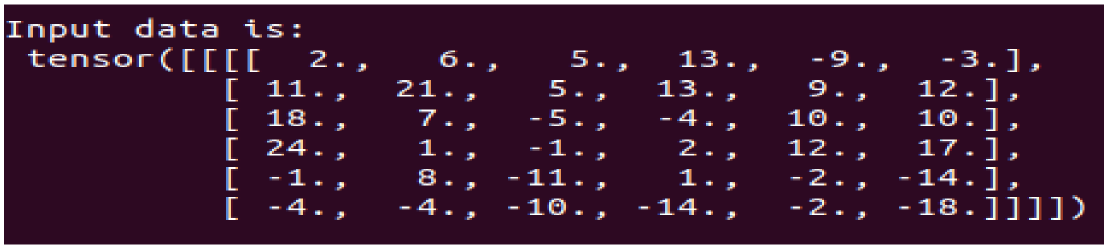

```
Input data is:
 tensor([[[[  2.,   6.,   5.,  13.,  -9.,  -3.],
          [ 11.,  21.,   5.,  13.,   9.,  12.],
          [ 18.,   7.,  -5.,  -4.,  10.,  10.],
          [ 24.,   1.,  -1.,   2.,  12.,  17.],
          [ -1.,   8., -11.,   1.,  -2., -14.],
          [ -4.,  -4., -10., -14.,  -2., -18.]]]])
```

Figure 26: Random Input Data.

[11]Data: Normalize it to be in range [-1, 1], assuming that max value of input data is 31(i.e. 5bits, this number is getting from the input such as A/D or Camera but in this random data, the max value is 24 which can be fitted in 5bits which represents data from 0 until 31) so we divide it by 31.

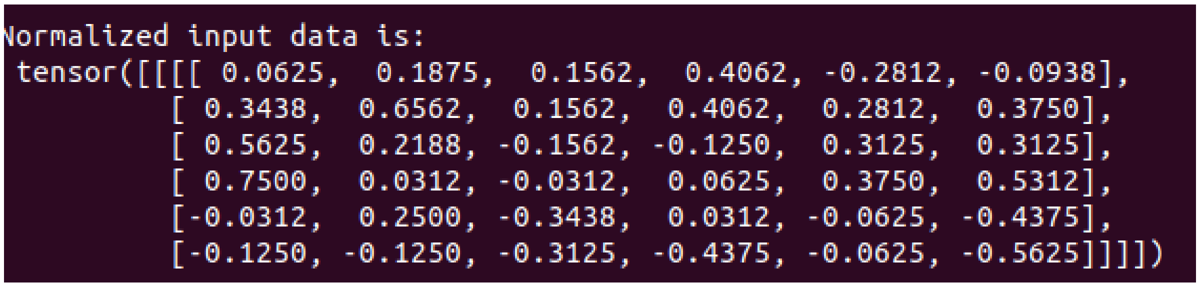

```
Normalized input data is:
 tensor([[[[ 0.0625,  0.1875,  0.1562,  0.4062, -0.2812, -0.0938],
          [ 0.3438,  0.6562,  0.1562,  0.4062,  0.2812,  0.3750],
          [ 0.5625,  0.2188, -0.1562, -0.1250,  0.3125,  0.3125],
          [ 0.7500,  0.0312, -0.0312,  0.0625,  0.3750,  0.5312],
          [-0.0312,  0.2500, -0.3438,  0.0312, -0.0625, -0.4375],
          [-0.1250, -0.1250, -0.3125, -0.4375, -0.0625, -0.5625]]]])
```

Figure 27: Normalized Input Data.

- **Get Model**: After searching for a model that fits this data and give high accuracy, usually this step is done using Keras [10] or PyTorch[12] [11] or any other Deep Learning Framework, let's assume that model consists from two convolution layers and one Dense.

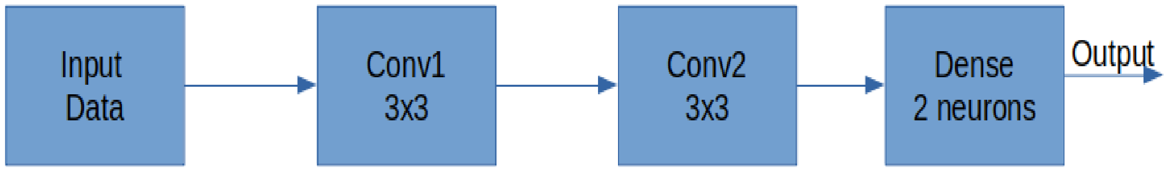
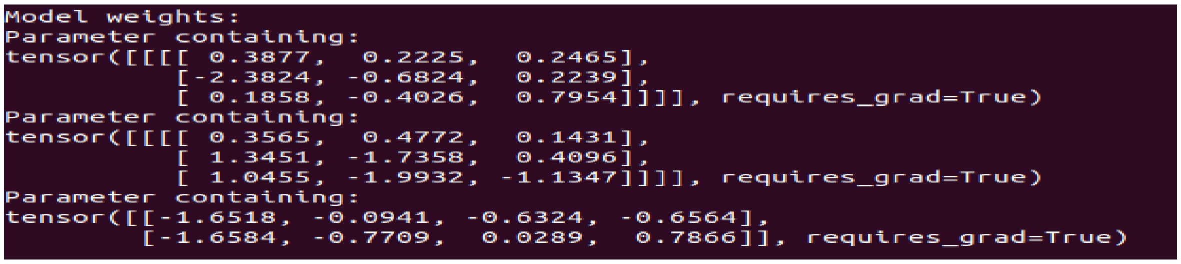


Figure 28: Model structure.

11 Normalize means in this section dividing by the biggest value to make the data within range [-1, 1] and we don't mean to subtract by mean and divide by standard deviation

12 This visualization step was made by using PyTorch

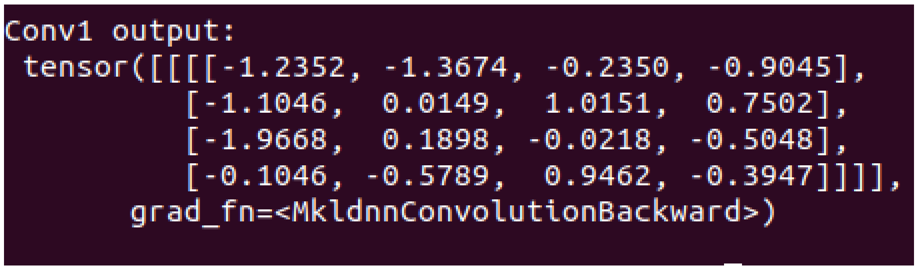
- **Get weights**: After searching for the best model that fit data with small size and latency, the next step is to get its weights, usually framework provide Function/Method that enable you to get weights after training the model



```
Model weights:
Parameter containing:
tensor([[[[ 0.3877,  0.2225,  0.2465],
          [-2.3824, -0.6824,  0.2239],
          [ 0.1858, -0.4026,  0.7954]]]], requires_grad=True)
Parameter containing:
tensor([[[[ 0.3565,  0.4772,  0.1431],
          [ 1.3451, -1.7358,  0.4096],
          [ 1.0455, -1.9932, -1.1347]]]], requires_grad=True)
Parameter containing:
tensor([[-1.6518, -0.0941, -0.6324, -0.6564],
        [-1.6584, -0.7709,  0.0289,  0.7866]], requires_grad=True)
```

FIGURE 29: MODEL WEIGHTS. THE FIRST TENSOR IS CONV1, SECOND CONV2, AND THE LAST IS DENSE.

- **Layer Inputs/Outputs**: After that we pass all inputs through the model, in this case we have only one sample but in real life example they are in range of thousands such as in WII where dataset is 255255 symbols. The input for the first layer (Conv1) is between [-1, 1] as that is data input which was normalized in the first step, but Conv1 output isn't within this range



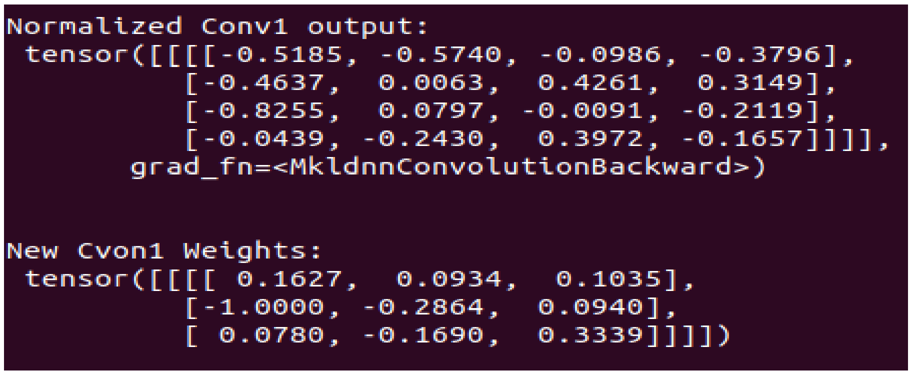
```
Conv1 output:
 tensor([[[[-1.2352, -1.3674, -0.2350, -0.9045],
          [-1.1046,  0.0149,  1.0151,  0.7502],
          [-1.9668,  0.1898, -0.0218, -0.5048],
          [-0.1046, -0.5789,  0.9462, -0.3947]]]],
       grad_fn=<MkldnnConvolutionBackward>)
```

FIGURE 30: CONV1 OUTPUT.

- **Normalize Conv1 weights**: We check output from each layer and divide the weights over the max value for each layer[13] which guarantee that max output for each layer[14] is within range [-1, 1]

[13] Weights / (max (weight. Abs ().max (), output. Abs ().max ()))

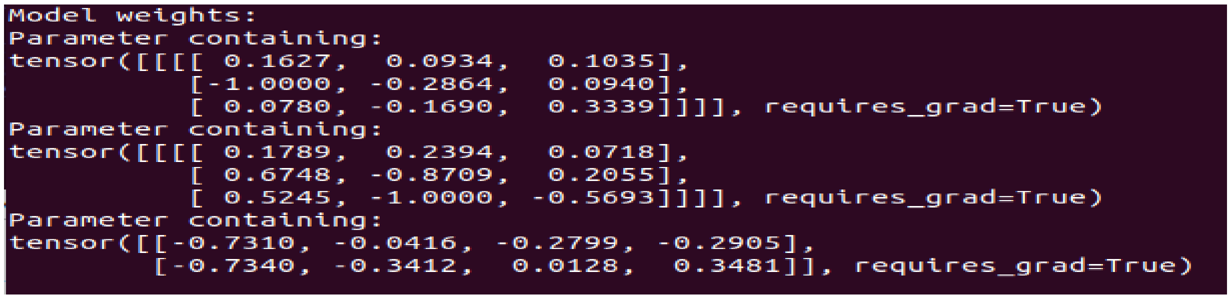
[14] We took also a safety factor in case of Psi-Lite as there may be an input in real-time experiment that bigger than inputs in the dataset, also if input maybe exceed after this safety factor, we can create a monitor circuit that check if input bigger than this safety factor and trigger an alarm.


```
Normalized Conv1 output:
 tensor([[[[-0.5185, -0.5740, -0.0986, -0.3796],
          [-0.4637,  0.0063,  0.4261,  0.3149],
          [-0.8255,  0.0797, -0.0091, -0.2119],
          [-0.0439, -0.2430,  0.3972, -0.1657]]]],
       grad_fn=<MkldnnConvolutionBackward>)


New Cvon1 Weights:
 tensor([[[[ 0.1627,  0.0934,  0.1035],
          [-1.0000, -0.2864,  0.0940],
          [ 0.0780, -0.1690,  0.3339]]]])
```

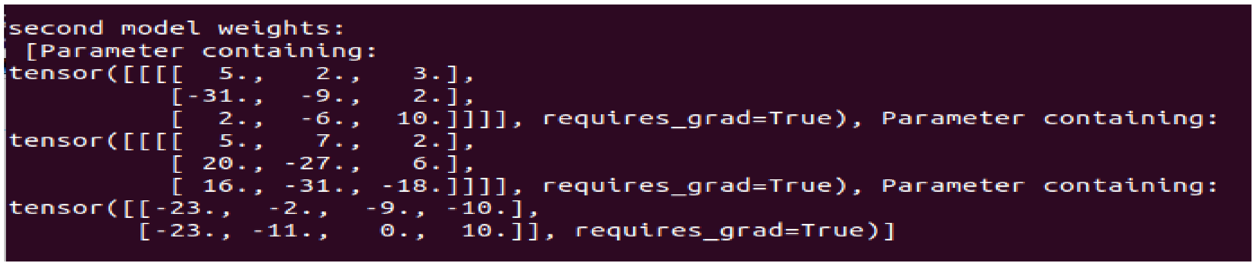
FIGURE 31: CONV1 OUTPUT IN RANGE [-1, 1] AND NEW WEIGHTS.

- **<u>Layer Weights and/or Input/Output</u>**: Repeat the previous two steps on all model layers and get the final weights, Note that weights in Figure 32 is the normalized version of weights in Figure 31, where Conv1 weights divided by 2.38(max weights in Conv1), Conv2 is divided by 1.99(max weights in Conv2), and finally Dense is divided by 2.25(max output from Dense). This guarantee that all outputs and weights will be in range [-1, 1]


```
Model weights:
Parameter containing:
tensor([[[[ 0.1627,  0.0934,  0.1035],
          [-1.0000, -0.2864,  0.0940],
          [ 0.0780, -0.1690,  0.3339]]]], requires_grad=True)
Parameter containing:
tensor([[[[ 0.1789,  0.2394,  0.0718],
          [ 0.6748, -0.8709,  0.2055],
          [ 0.5245, -1.0000, -0.5693]]]], requires_grad=True)
Parameter containing:
tensor([[-0.7310, -0.0416, -0.2799, -0.2905],
        [-0.7340, -0.3412,  0.0128,  0.3481]], requires_grad=True)
```
FIGURE 32: WEIGHTS AFTER NORMALIZATION.

- **<u>Fixed-point Representation</u>**: Now we normalized weights and input, but the input from A/D that enters FPGA will not be normalized or floating and also we can't put a float weight, as explained in the previous subsection, we need to find 'M' which enable us represent weights in Fixed-point without loss in accuracy, let's assume that 'M' = 'N' where 'N' is the input '5' bits and multiply weights in Figure 33 with 31 and take floor[15] for it.

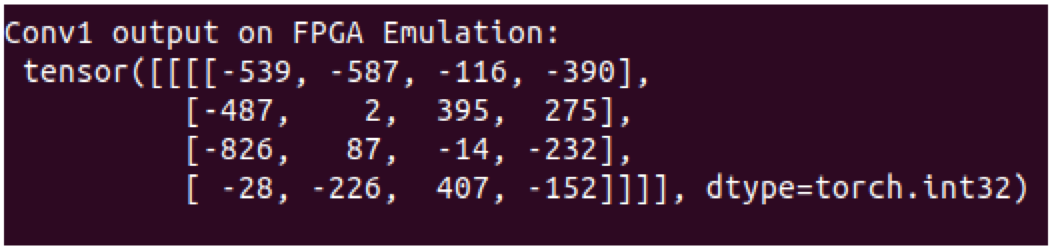
[15] Taking floor directly will down negative number to less than its value, but as we multiplied by 31, so down -31 to -32 will not have any effect on the FPGA because for 5bits + one-bit sign, the result is between -32 to 31



```
second model weights:
 [Parameter containing:
tensor([[[[  5.,   2.,   3.],
          [-31.,  -9.,   2.],
          [  2.,  -6.,  10.]]]], requires_grad=True), Parameter containing:
tensor([[[[  5.,   7.,   2.],
          [ 20., -27.,   6.],
          [ 16., -31., -18.]]]], requires_grad=True), Parameter containing:
tensor([[-23.,  -2.,  -9., -10.],
        [-23., -11.,   0.,  10.]], requires_grad=True)]
```

FIGURE 33: WEIGHT AFTER MULTIPLYING BY (25-1).

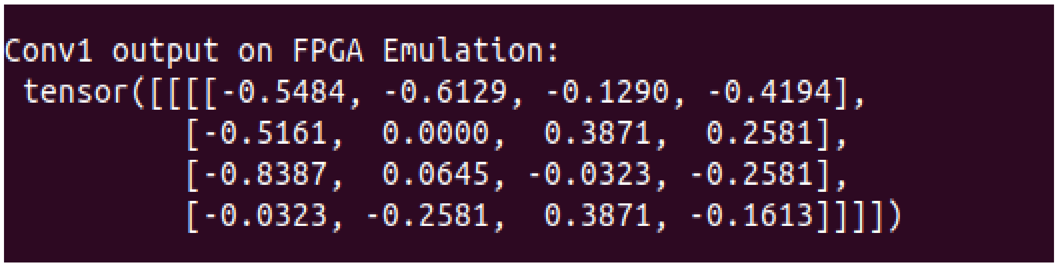
- Now, we got the weights in Figure. 33 that can be sent to FPGA memory and data in Figure 26 that enters from A/D. We need know to emulate operation on FPGA and see if it gives true or wrong output compared to floating point.



```
Conv1 output on FPGA Emulation:
 tensor([[[[-539, -587, -116, -390],
          [-487,    2,  395,  275],
          [-826,   87,  -14, -232],
          [ -28, -226,  407, -152]]]], dtype=torch.int32)
```

FIGURE 34: CONV1 OUTPUT IN FIXED-POINT OPERATION.

- <u>**The last step**</u>: is checking if this visualization was true or not, that can be done by taking output shown in Figure. 34 dividing it by 2^5, comparing the output with data shown in Figure. 30. We can note from Comparison that 5 bits for weights was not the best choice. Although the output keeps the same ratio between values, negative, and positive signs and will mostly give a true classification in this symbol/data, but it may be wrong in other cases.



```
Conv1 output on FPGA Emulation:
 tensor([[[[-0.5484, -0.6129, -0.1290, -0.4194],
          [-0.5161,  0.0000,  0.3871,  0.2581],
          [-0.8387,  0.0645, -0.0323, -0.2581],
          [-0.0323, -0.2581,  0.3871, -0.1613]]]])
```

FIGURE 35: DIVIDING CONV1 FIXED-POINT OUTPUT WITH 31.

3.5 Proposed Models with Fixed-point

3.5.1 Optimized IQ

We did the same as discussed in the previous two subsections except that we didn't normalize the input before training the model, this was handled by avoiding shifting in the first block but this leads to 16 bits to represent the weights with 1% Error Increasing.

3.5.2 Psi

Phase branch in Psi is the output from dividing Quadrature (Q) branch over In-phase (I) branch and taking arctan for it as was mentioned in the Psi-Lite Section. This arctan function if it takes small input data range, it will give output like shown in figure 36, where if it takes big input data range, it will give output like shown in Figure 37. Figure 38 shows arctan with data symbol from WII data.

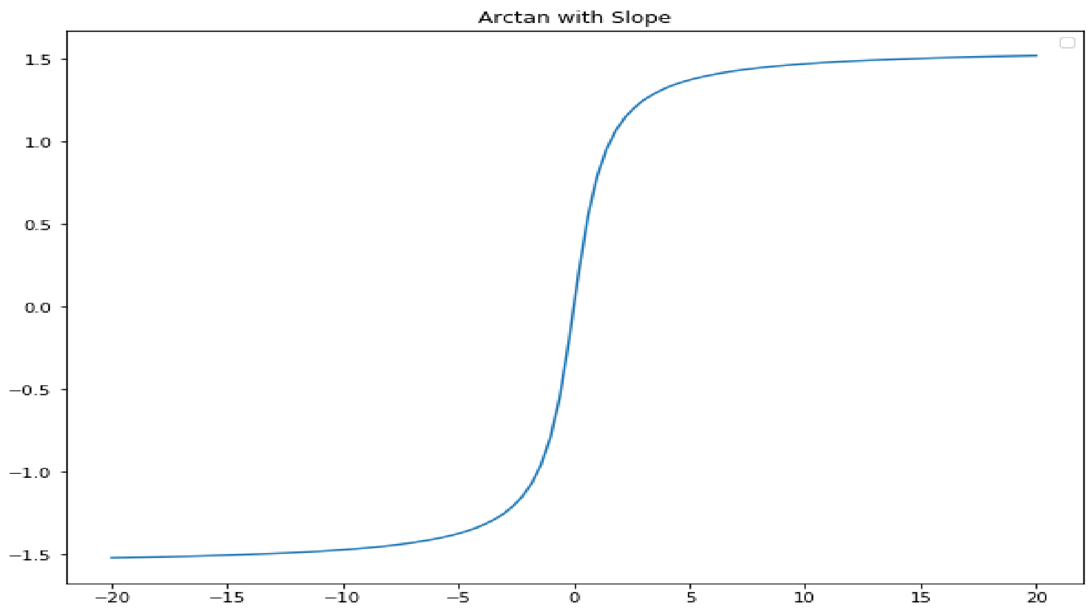


FIGURE 36: ARCTAN FUNCTION WITH SMALL DATA RANGE.

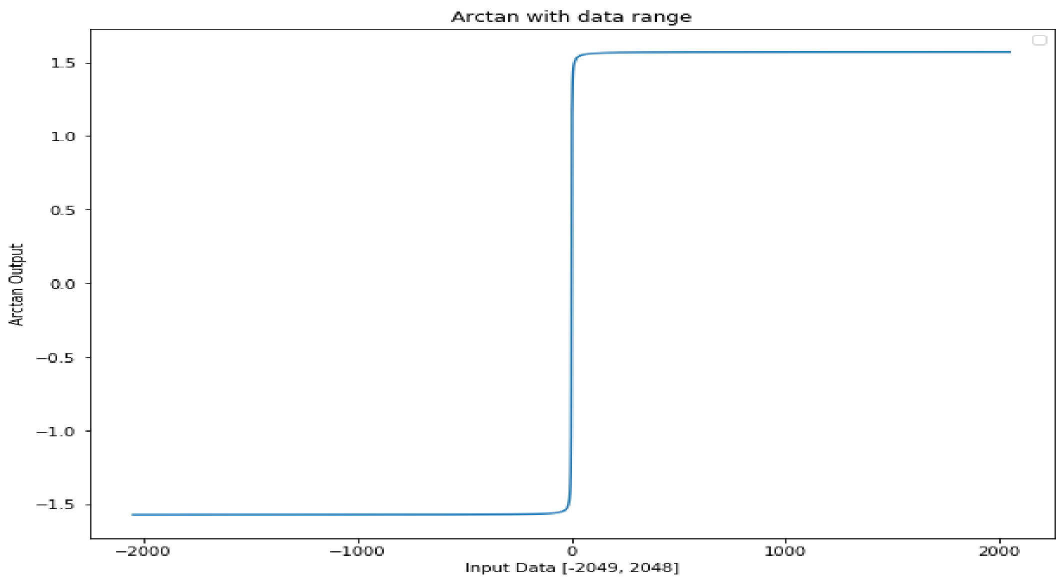


FIGURE 37: ARCTAN FUNCTION WITH BIG DATA RANGE.

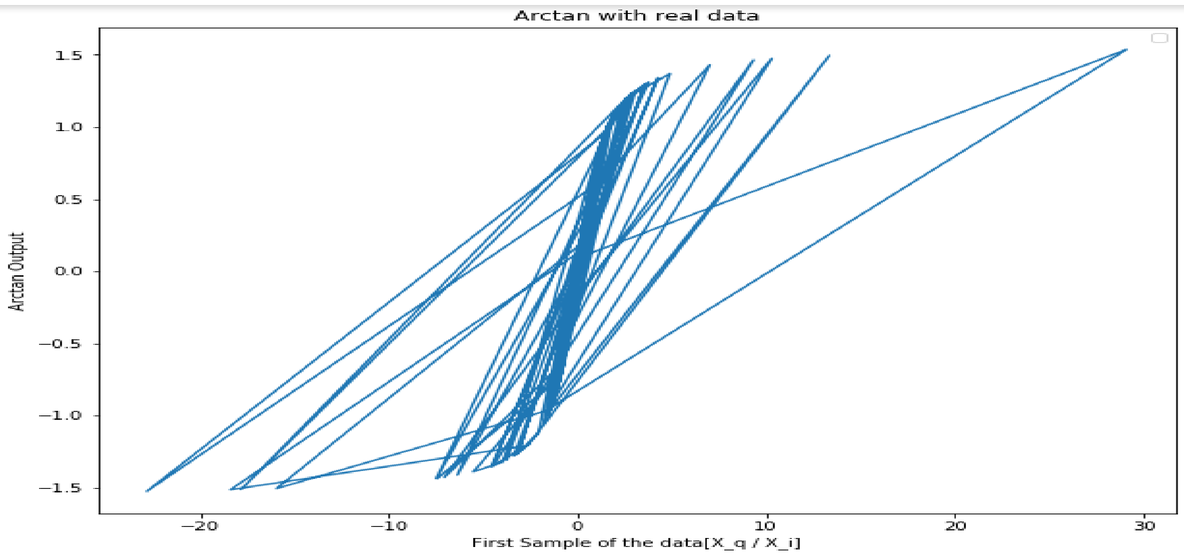


FIGURE 38: ARCTAN FUNCTION WITH SYMBOL DATA.

In the three Figures above, arctan makes the output's range within range $[\frac{-\pi}{2}, \frac{\pi}{2}]$. So, IQ range is [-2048: 2047](i.e. 2^{12} bit + 1 bit sign) and arctan range is [-1.57, 1.57]. This will leads to different in data ranges and need a bigger number of bits to represents weights 'N' as explained in the previous section which make the whole model bigger than bladeRF size and make real-time experiment impossible. So, this gives an intuition about the reason for moving from Psi to Psi-Lite.

3.5.3 Psi-Lite

We did the same as discussed in the previous two subsections except that existence of two branches imposed a new constrain. This constrain comes from the need to unify scale factor of both branches (e.g. if IQ branch is divided by 'x' factor and Amplitude branch is divided by 'y' factor and this cause next layer with weights, Shared Convolution layer in case of Psi-Lite but it exists in Psi Model, will have inputs with different scales leads to biasing towards the branch with smaller scale factor which will leads to a wrong classification. This problem can be solved by dividing weights in both branches by the biggest value of both branches (max (x, y)). if there are more than layer in each branch, which is not the case in Psi-Lite, this will lead 'x' and 'y' to be scaling factor from each branch. We can consider it as a factorization problem and solve it.

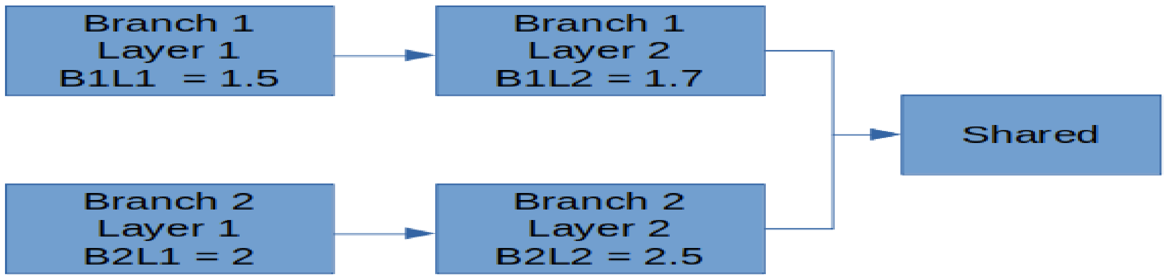


FIGURE 39: MULTI-BRANCHES WITH MULTI-LAYERS.

Figure 39 shows a branch with this Problem where we can deduce from this figure that x=B1L1*B1L2=1.5*1.7 = 2.55, y=B2L1*B2L2=2*2.5 = 5 and as y > x, we can divide weights of branch2 and layer1 with 2 and weights of branch2 and layer2 with 2.5 but for Branch 1 it should be 5 not 2.55 and we can distribute[16] this increasing in the division factor x, one way is as shown in Equation (20)

$$\begin{gathered}(B1L1+\delta)*(B1L2+\delta)=5\\ \delta^2+3.2\delta-2.45=0\\ \delta=.6383\end{gathered} \qquad (20)$$

So now we can divide weights of branch1 and layer1 with (1.5 + .6383) and weights of branch2 and layer2 with (1.7 + .6383)

[16] Distribution increase in the factor doesn't guarantee achieving the best solution but the intuition with this solution is distributing this new increasing in the factor on all layers and taking the floor function after it will reduce its effect which leads to reduce in the required number of bits

Chapter Four: Hardware approach

In this chapter, hardware implementation and results for two models: optimized IQ and psi lite are discussed. Section 1 shows Optimized IQ and section 2 shows psi lite.

4.1 optimized IQ model Hardware:

This section discusses implementation of proposed IQ model targeting modulation recognition problem. Verilog used as a hardware description language. Vivado 2017.4 Used as synthesis tool. Targeted FPGA is ZYNQ Ultra scale+ ZU104 FPGA [5].

The reasons to choose this model on FPGA not psi model are: it has low number operations as discussed in chapter 3 section 1.4 , low number of bits needed when using fixed point operations as discussed in chapter 3 section 5.1, and no need to implement FFT operation as hardware.

4.1.1 Fixed Point Operations

Extracted weights from Keras are represented in floating point which was a problem from the perspective of number of calculations and processing time. Also using floating points consumes more resources of FPGA which means more power so it was wise to avoid floating point operation and use fixed point operation. After simulating model again with integer weights using fixed point operation it was concluded that storing weights in 16 bits minimizes the error and maintains accuracy almost unchanged.

4.1.2 Implementations of used Operations

This subsection, shows how to implement used operation in FPGA.

4.1.2.1 Convolution Operations

Convolution layers in the models are 2D convolution. In convolutional layers of 2D CNNs, the input and output features have multiple channels. A 2D convolution is applied to each channel of the input feature and the generated outputs are then accumulated resulting in one channel of the final output feature, For simplicity, X used to indicate the input samples with a size of $h \times w$ and Y used to indicate the output with a size of $m \times h_o \times w_o$. Here, m is the number of output channels. h_i and w_i are height and width of input, and h_o and w_o are height and width of output. W used to indicate the weights feature with $m \times k_1 \times k_2$, k_1 and k_2 are height and width of kernel. h_o equals to $h+k_1-1$, w_o equals to $w+k_2-1$. Each pixel Y[mm] [hh] [ww] in the output feature is calculated by:

$$Y[mm][hh][ww] = \sum_{rr=0}^{k1-1} \sum_{cc=0}^{k2-1} W[mm][rr][cc] * X[ch][hh+rr][ww+cc] \quad (21)$$

To Implement this equation on hardware, figure 40 shows the implementation.

Input data to the filter is multiplexed each clock cycle with address counter works as selection line. The input data each clock is a shifted version by one as stride is one then the selected data is multiplied with corresponding weights from memory after that all outputs of multiplier is summed and de-multiplexed to the corresponding output line . After all inputs is multiplied by filters, next filters weights replace current weights and multiplied by input as shown.

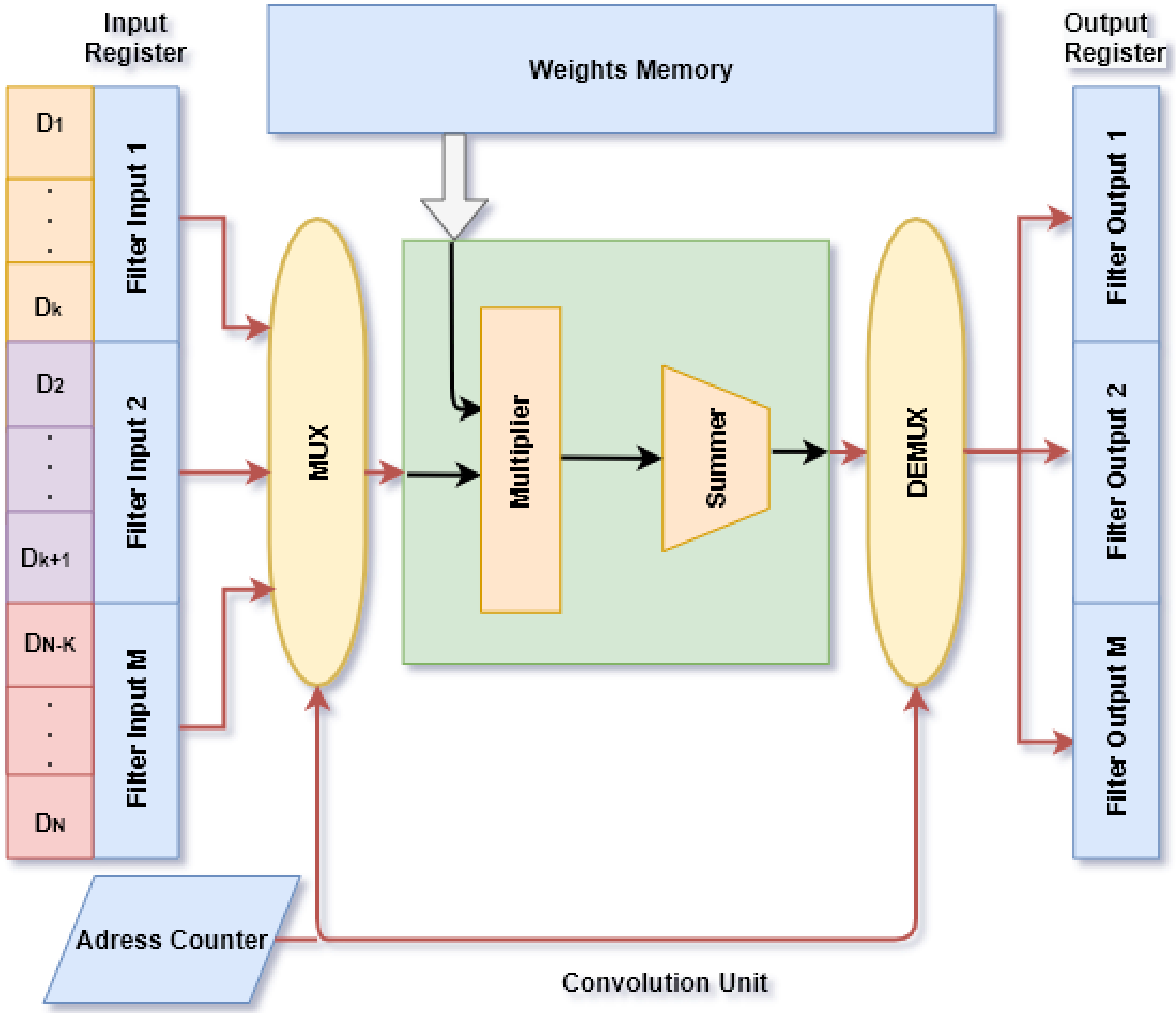


FIGURE 40: CONVOLUTION OPERATION IMPLEMENTATION.

4.1.2.2 Neuron Implementation

Operations in neuron can be described as weighted sum next equation elaborate this statement:

$$Y\ output[i] = \sum_{j=0}^{N} W[i][j] * X[j] \qquad (22)$$

X used to indicate the input with a size of h_x, h_y is the height of input X . Y used to indicate the output with a size of N, and N is number of neurons. We used to indicate the weights feature with size of N $\times h_x$.

To implement dense operation, weights are being read from memory then multiplied with its corresponding inputs then all outputs are summed as shown in figure 41. Number of multiplications in the neurons of first fully connected layer was 1044 which is very large number and processing, all these multiplications at one time consumes almost all FPGA resources so it was found that neuron operations should be reduced almost by x1/12 (87Multiplications/Clock) to get reasonable utilization and at the same time keeping processing time small as possible.

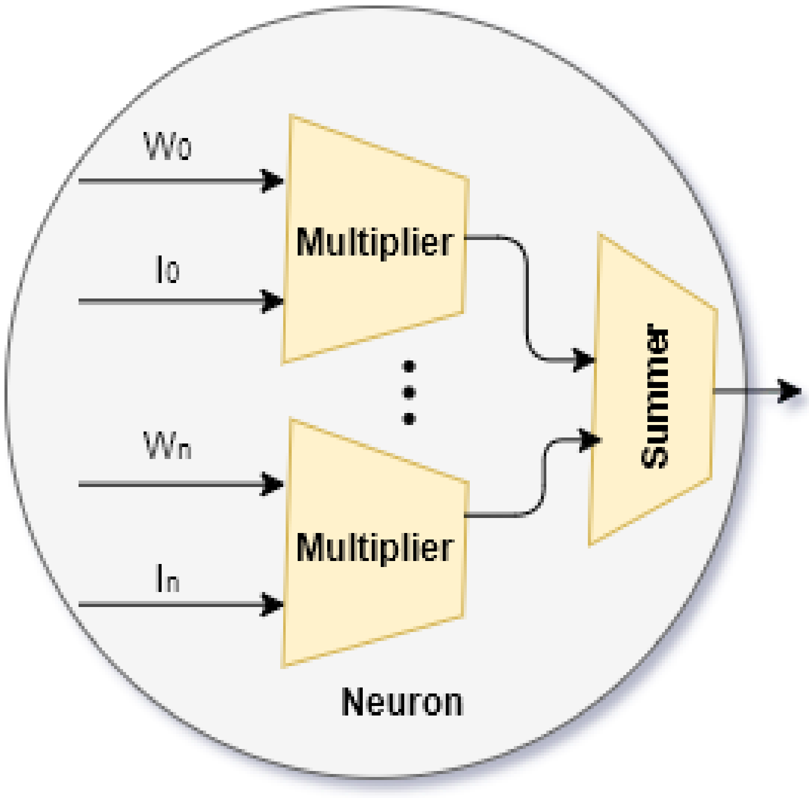


FIGURE 41: DENSE OPERATION IMPLEMENTATION.

4.1.2.3 Activation Functions

4.1.2.3.1 RELU

ReLU was chosen as activation function due to its outstanding performance, high accuracy and simplicity in figure 42, as it is the only one that does not need division operation. we can notice how ReLU function can be easily implemented without any approximations and complex calculations.

It's just filtering out negative numbers to zero and permits positive number, it's implemented as multiplexer its selection line the sign bit.

So when sign bit is zero (positive number) the output will be as input and if sign bit is one (negative number) the output will be zero which can be interpreted from figure 43.

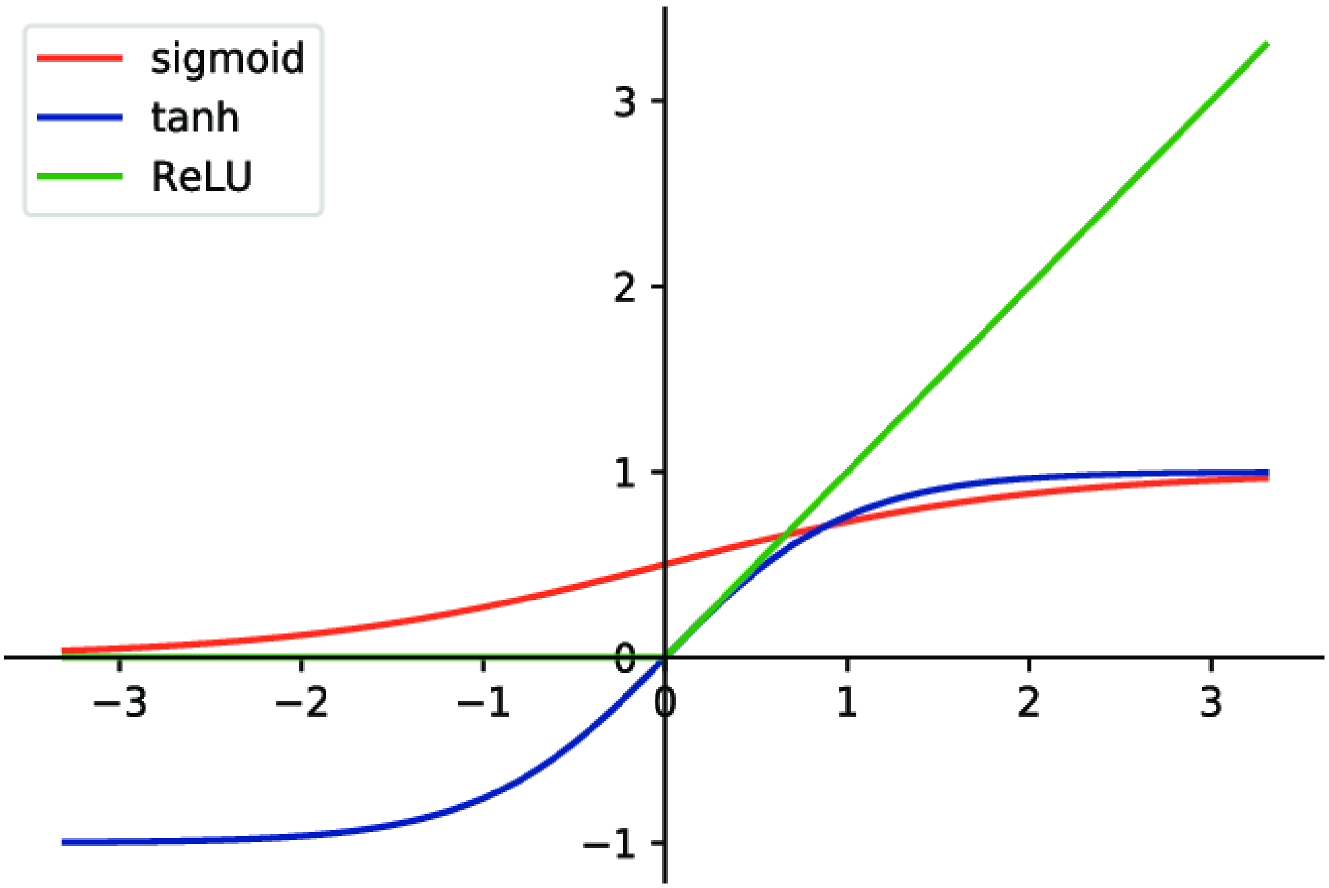


FIGURE 42: ACTIVATION FUNCTIONS.

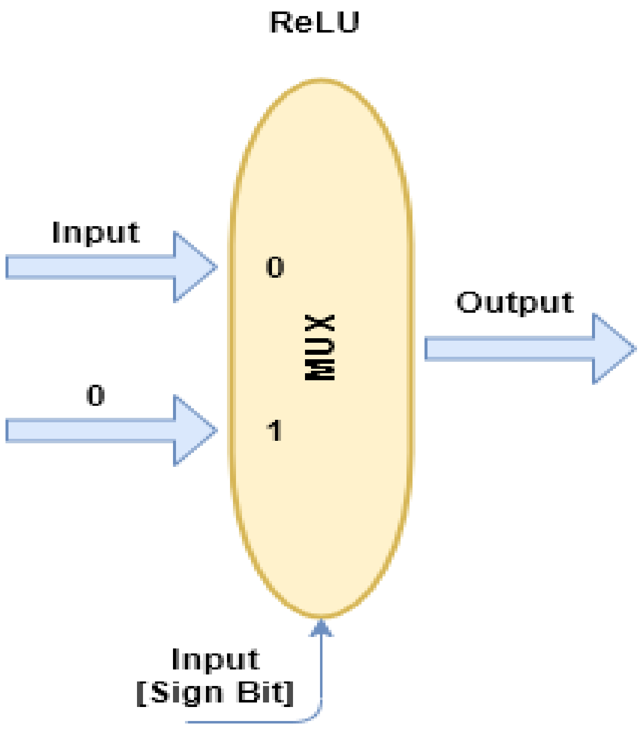


FIGURE 43: RELU OPERATION.

4.1.2.3.2 SOFTMAX

Output layer consists of eleven neurons each one of them corresponds to a class (Modulation Scheme), the activation function of output layer is softmax, figure 44 explains how sofmax works and it’s mathematical representation it turns outputs of neurons to a probability distribution of each class, the predicted class is the one with the highest probability.

In order to implement softmax many complex calculations were needed to calculate exponentials and dividers but softmax can be implemented in much simpler way that will save much calculation and memory, Simply find out the largest neuron and the output is its position which will correspond to the predicted class, figure 45 shows how algorithm works. Each neuron compared with adjacent neuron using subtractor as a comparator, the sign bit of subtractor output determines which operand is larger so if first operand is larger, its position will be selected through multiplexer and vice versa, this operation done over all neuron until position of largest neuron figured out.

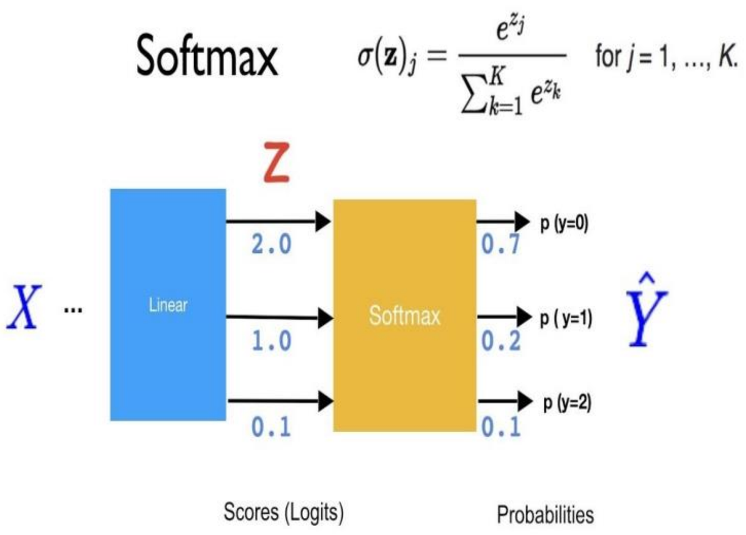


FIGURE 44: SOFTMAX OPERATION.

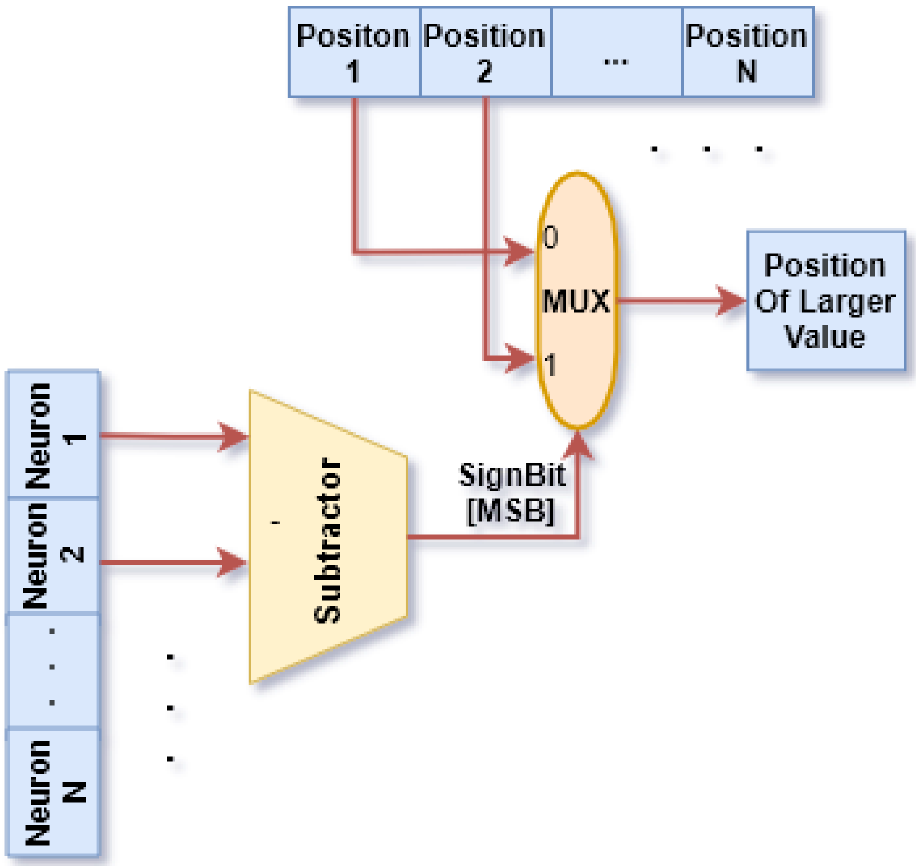


FIGURE 45: SOFTMAX IMPLEMENTATION.

4.1.3 Model Flow

4.1.3.1 Convolution Layer 1

In convolution layer 1, Sample for each filter is calculated every clock To speed the convolution layer 1 operation. the block in figure 46, compute one sample for one filter so it is repeated 45 times, to calculate all 45 output channels from hh to hh+7 of input height, then all outputs (Y14, Y24… Y444) saved in LUTS.

The repeated block is constructed by using multiplier block, adder block, ReLU block, and rescale block. Multiplier block multiply input and the correspondent weights. Adder block adds the outputs of multiplier block by using tree adder. ReLU block do ReLU operation. Rescale block reduces number of bits of ReLU block output by doing shift right operation.

In first clock hh equals to 0 so all 45 output channels from 0 to 7 of input height are calculated, in the next clock period hh equals to 1 so all 45 output channels from 0 to 7 of input height are calculated, and repeat this steps until hh is equal 127. When hh is equal 127 convolution layer 1 stop working unless the input changes, so weights does not change every clock, but input is shifted by 1 every clock.

4.1.3.2 Convolution Layer 2

Convolution layer 2 block is like the repeated block in convolution layer 1, figure 47 Shows that. Convolution layer 2 block is constructed by using multiplier block, adder block, ReLU block, and rescale block. After adder operation number of bits does not increase as the inputs after convolution 1 most of the values is not high values because of ReLU operation and rescale operation and making Sure no overflow happens.

To start the operation of convolution layer 2, convolution layer 1 must finish its first six outputs as it is the minimum input for convolution layer 2. When convolution 2 starts hh is equal to zero and mm is equal to zero. In each clock convolution layer 2 calculates, after that hh is equal to hh+1. When hh is equal to 120, hh will reset to be zero and mm is equal to which means channel mm of output of convolution 1 is calculated for all convolution layer 2 input height. When mm is equal to 9 that’s mean convolution layer 2 operation is finished.

4.1.3.3 Dense Layer 1

Dense layer 1 starts working when convolution layer 2 operation is finished. Each neuron is done in several clock because large number of multiplications, when a neuron is finished the next one is calculated, when the last one is finished dense layer 1 stops working as shown in figure 48.

4.1.3.4 Dense Layer 2

Figure 48 shows how process, dense layer 2 stars its process when dense layer 1 is finished. Each neuron in dense layer 2 done in one clock. When the last neuron is finished, dense layer output goes to softmax unit, and then the last value is computed. Softmax works as described before.

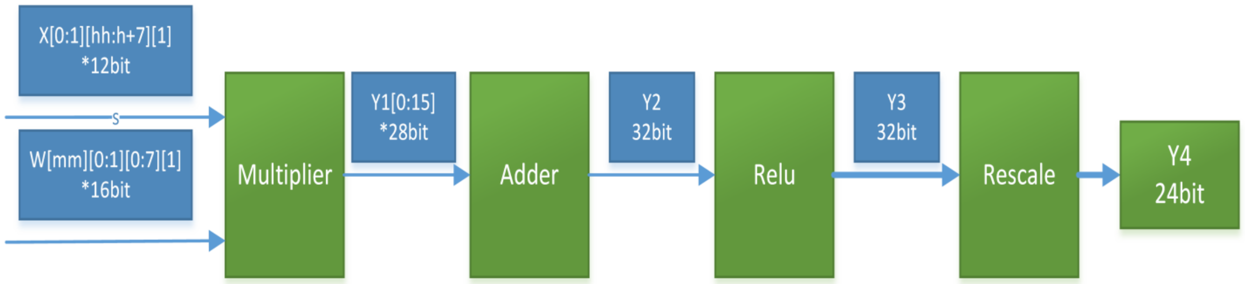


FIGURE 46: CONVOLUTION LAYER 1 BLOCK.

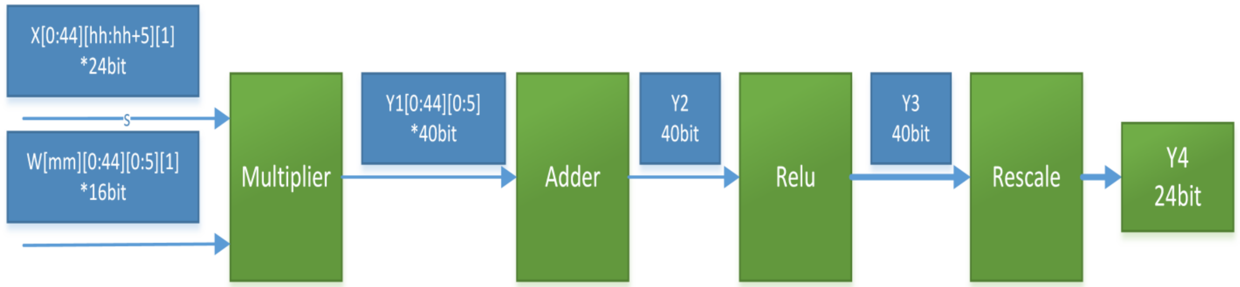


FIGURE 47: CONVOLUTION LAYER 2 BLOCK.

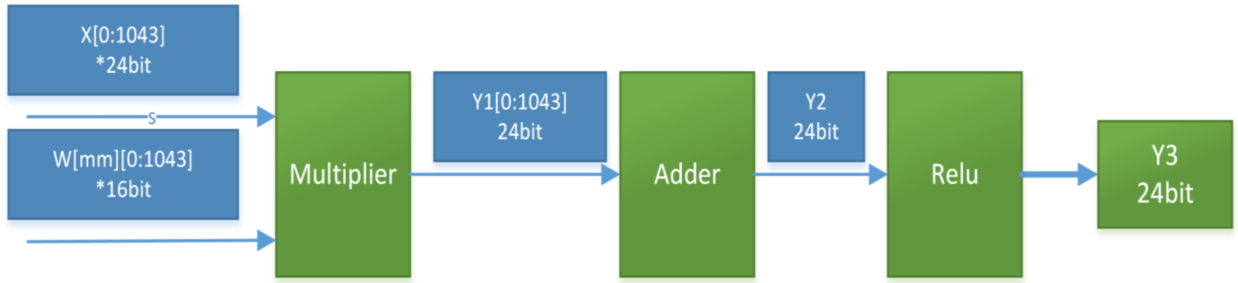


FIGURE 48: DENSE LAYER 1 BLOCK.

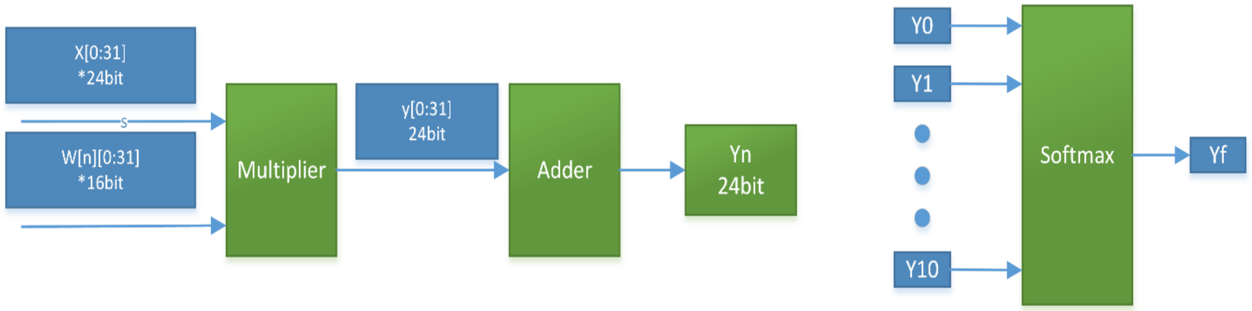


FIGURE 49: DENSE LAYER 2 BLOCK.

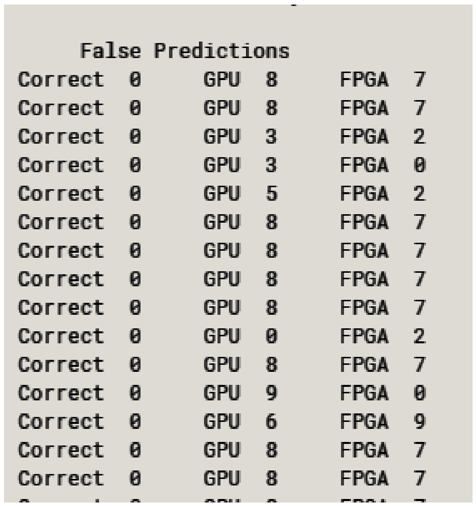
4.1.4 Hardware results

In this section results will be discussed after implementing model on FPGA and thus results will be compared with the results given on GPU or other models.

4.1.4.1 Estimated error

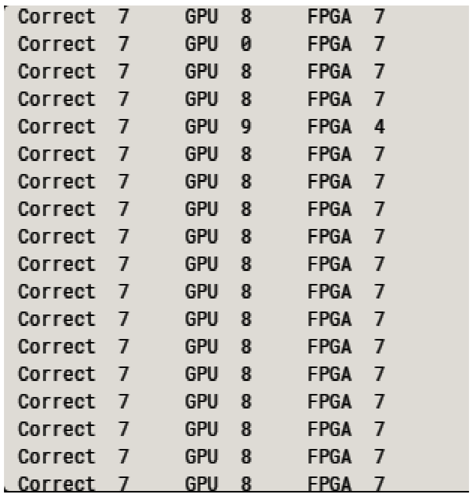
The estimated error due to quantization error and using fixed point operations instead of floating point was about 1.19% compared to GPU predictions but after implementing the model on FPGA the results were almost the same and the validation accuracy on FPGA reached 54.38% however the validation accuracy on GPU was 54.36% , it was a little bit confusing how 1.19% mismatch with GPU results didn't affect accuracy on FPGA and still almost same as GPU so and investigation of the mismatched samples was done to better understand and explain this phenomena.

Neutral Mismatch



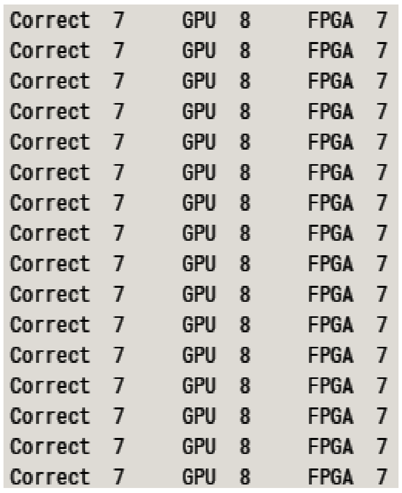
```
    False Predictions
Correct  0     GPU  8     FPGA  7
Correct  0     GPU  8     FPGA  7
Correct  0     GPU  3     FPGA  2
Correct  0     GPU  3     FPGA  0
Correct  0     GPU  5     FPGA  2
Correct  0     GPU  8     FPGA  7
Correct  0     GPU  8     FPGA  7
Correct  0     GPU  8     FPGA  7
Correct  0     GPU  8     FPGA  7
Correct  0     GPU  0     FPGA  2
Correct  0     GPU  8     FPGA  7
Correct  0     GPU  9     FPGA  0
Correct  0     GPU  6     FPGA  9
Correct  0     GPU  8     FPGA  7
Correct  0     GPU  8     FPGA  7
```

Positive Mismatch



```
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  0     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  9     FPGA  4
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
```

Negative Mismatch

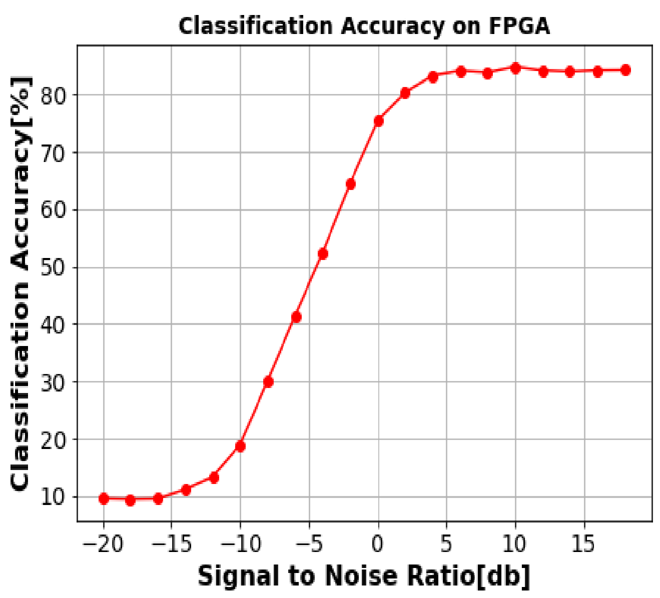


```
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
Correct  7     GPU  8     FPGA  7
```

FIGURE 50: ESTIMATED ERROR.

The mismatch with GPU prediction can be classified into three groups Neutral, Positive and Negative mismatches .Neutral mismatch happens when the predictions on FPGA mismatches with GPU prediction but both are wrong which won't cause any harm to accuracy, the positive mismatch happens when the predictions on FPGA mismatches with GPU prediction but matches with the correct predictions which will increase accuracy, the last mismatch is negative mismatch which eliminates effect of positive mismatch it happens when FPGA predictions mismatches with GPU predictions however GPU predictions are correct so this will cause reduction of accuracy so the total effect of the three mismatches will cancels each other and the overall accuracy won't be affected and this will be very obvious in resulting scores and accuracy curves.

4.1.4.2 Accuracy Curves



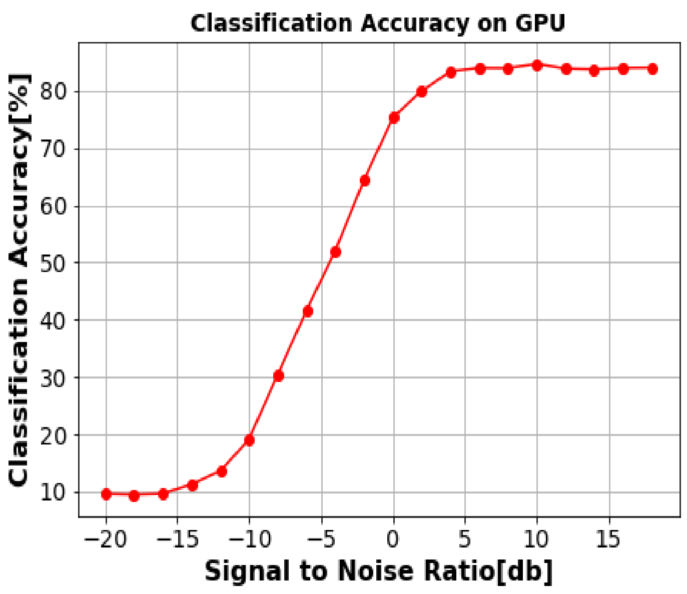
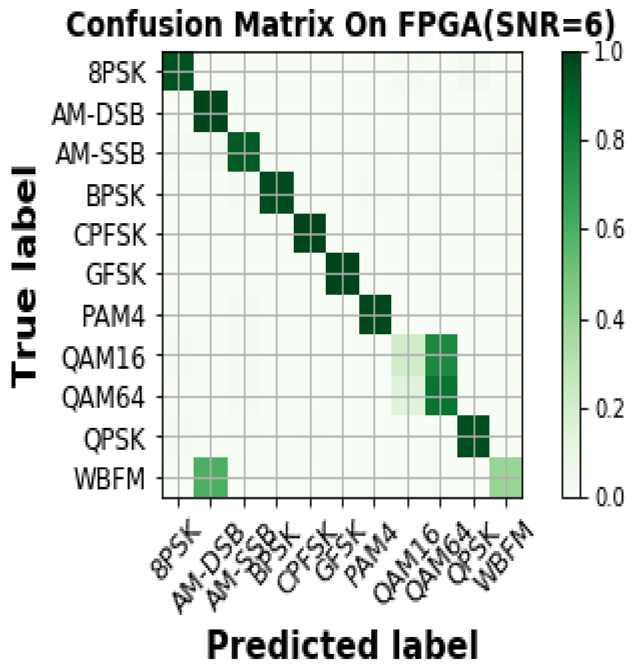


FIGURE 51: CLASSIFICATION ACCURACY ON FPGA VS GPU.

4.1.4.3 Confusion Matrix



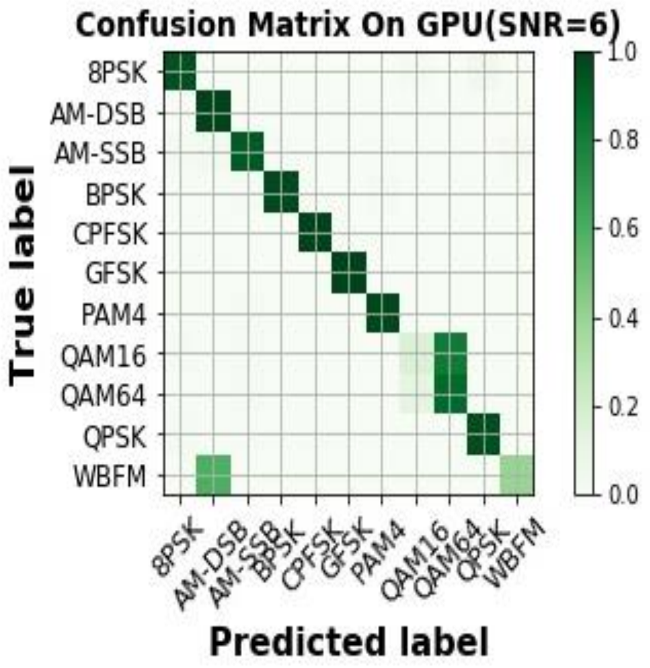


FIGURE 52: CONFUSION MATRIX ON FPGA VS GPU.

4.1.4.4 Performance Metrics

TABLE 11: PERFORMANCE METRICS OF PROPOSED MODEL ON GPU VS FPGA.

Model	SNR	Precision	Recall	F1 Score
GPU-optimized CNN_{IQ}	High	0.84	0.86	0.82
	Medium	0.76	0.77	0.74
	Low	0.31	0.33	0.27
FPGA- optimized CNN_{IQ}	High	0.84	0.86	0.82
	Medium	0.76	0.77	0.75
	Low	0.30	0.33	0.27

From the above results and accuracy curves and confusion matrix we can see how the results of FPGA and GPU are very similar. Number of Calculations and Model Size.

4.1.4.5 Layers Utilization

Convolution Layer1 uses large number of DSPs due to parallel operations to decrease delay of model but more power consumed in DSP blocks.

Convolution Layer1, Convolution Layer2, and Dense Layer1 uses more LUTS due to large output of convolution layer1 and 2, in case odd dense layer1 because of it has large number of weights. To free the design from memory Bandwidth limitations the weights and results stored on LUTRAM and registers that's why they are utilized by 14% and 12% respectively. Next Table shows Utilization in each layer.

TABLE 12: LAYERS UTILIZATION OF OPTIMIZED IQ.

Layer/Utilization	LUTS	Flip Flop	DSP
Convolution Layer1	28864	213	720
Convolution Layer2	19253	3413	270
Dense Layer1	19864	1330	87
Dense Layer2	1223	73	39

4.1.4.6 Compare with related work

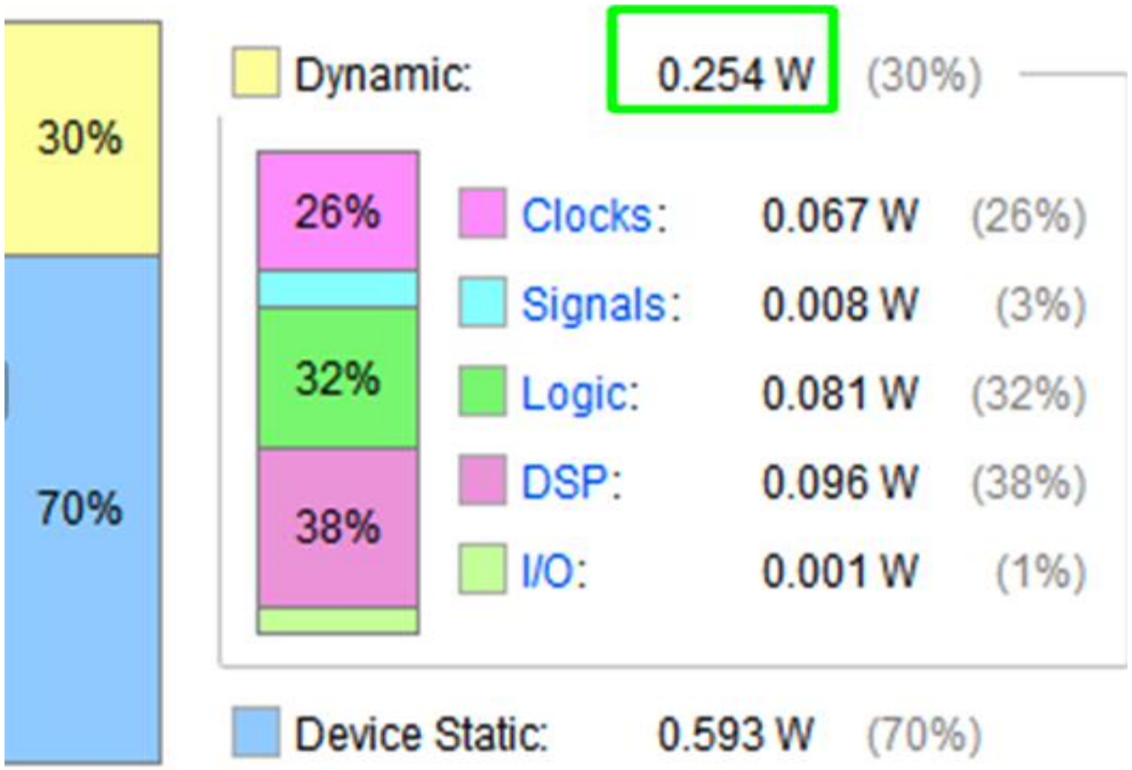
Table 13 shows the proposed model (optimized IQ) utilization and total power against model in paper [2] (soltani model).

In optimized IQ model 11 works with 11 modulation types. In soltani model, 6 modulation types are used. Both models used same frequency. LUTS utilization in optimized IQ model is smaller than in soltani model, this could because of soltani model used only dense layers witch has more parameters than convolution layers wish was used in optimized IQ.

Optimized IQ has less total power on chip but has more DSP power which shown on figures 53, and 54 [2]. DSP was used to decrease delaying model as power was not have constraints.

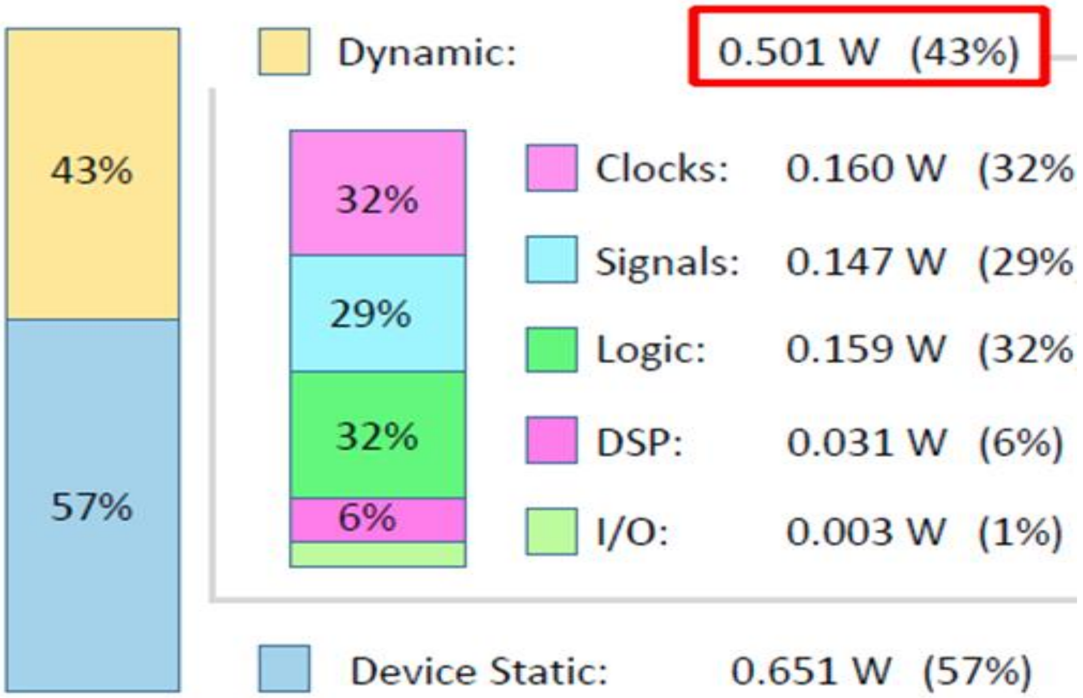
TABLE 13: UTILIZATION OF OPTIMIZED IQ VS SOLTANI MODEL.

Comparison \ Model	optimized IQ	soltani model
number of modulation types	11	6
FPGA	ZYNQ UltraScale+ MPSoC ZCU104	ZYNQ UltraScale+ MPSoC ZCU102
Frequency	70 MHz	70 MHz
LUT	81,358	158,435
LUTRAM	14,832	117,380
FF	58,386	16,222
DSP	1,118	210
IO	5	10
BUFG	22	4
Total On-Chip Power	0.847 W	1.152 W



Dynamic:		0.254 W	(30%)
	Clocks:	0.067 W	(26%)
	Signals:	0.008 W	(3%)
	Logic:	0.081 W	(32%)
	DSP:	0.096 W	(38%)
	I/O:	0.001 W	(1%)
Device Static:		0.593 W	(70%)

FIGURE 53: OPTIMIZED IQ POWER.



Dynamic:		0.501 W	(43%)
	Clocks:	0.160 W	(32%
	Signals:	0.147 W	(29%
	Logic:	0.159 W	(32%
	DSP:	0.031 W	(6%)
	I/O:	0.003 W	(1%)
Device Static:		0.651 W	(57%)

FIGURE 54: SOLTANI MODEL POWER.

4.2 Psi-Lite Hardware

4.2.1 Fixed Point

As explained in Section Psi-Lite, Fixed-Point implementation with two branches and how to solve the two branches division factor Problem. To move to hardware with fixed point weights' representation, we tried different number of bits and calculate the loss relative to floating-point software. Table 14 shows the different number of bits and their accuracy loss relative to software. We chose number of bits equal 11 bits as it compromises between error rate and bits required for fixed-point representation.

TABLE 14: PSI-LITE NUMBER OF BITS

Number of bits	8	9	10	11	12	13
Error Rate	1.9%	1.1%	0.7%	0.1%	0.1%	0%
Miss Number	19	11	7	1	1	0

4.2.2 Device Targeted

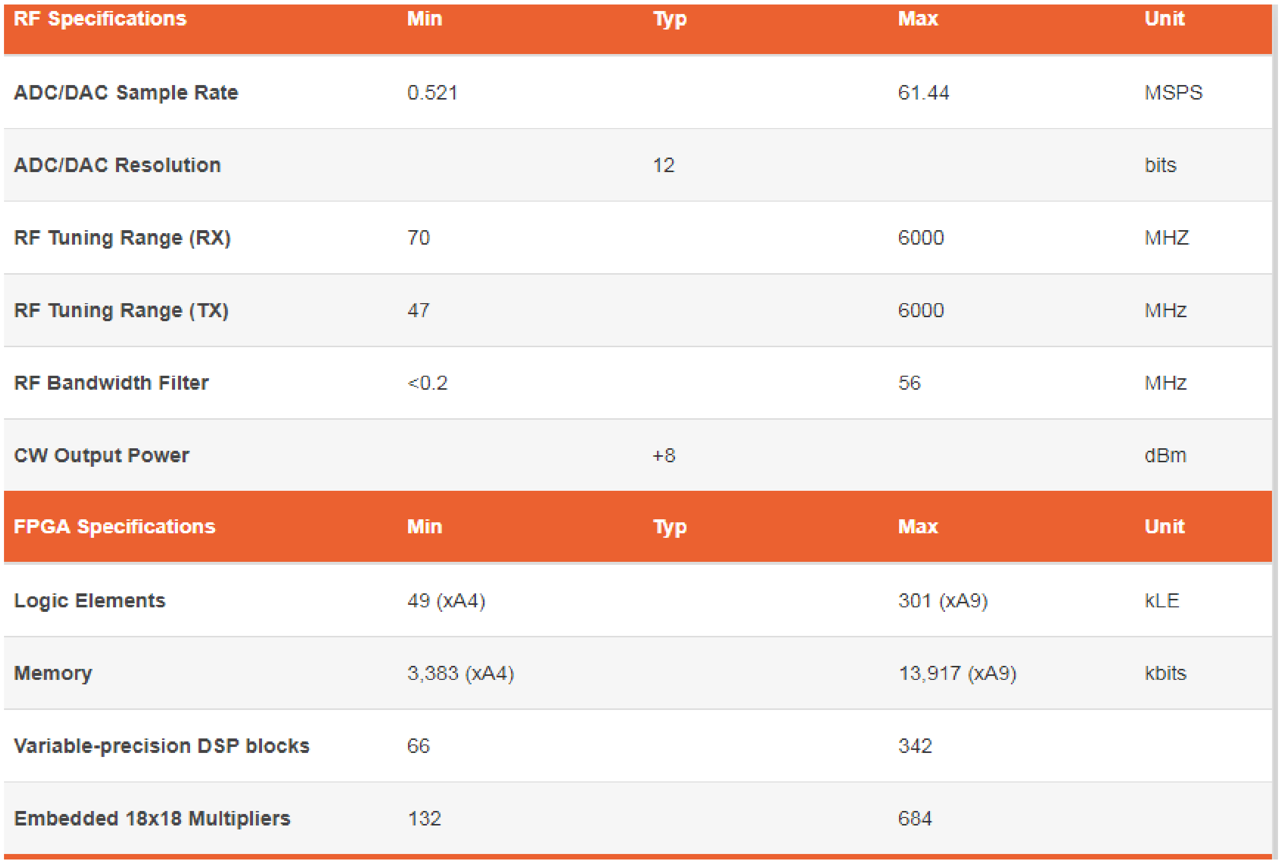
As stated in the Psi-Lite introduction, we targeted bladeRF device [12] which has small FPGA sizes but will enable us doing real-time test. There are four types of bladeRF: bladeRF x40, bladeRF x115, bladeRF 2.0 micro xA4, and bladeRF 2.0 micro xA9[17], Figure 55 shows a comparison between bladeRF 2.0 micro xA4 and XA9[12]. We selected the biggest bladeRF available which has a big number of logic elements and DSP blocks relative to other bladeRF versions. But even though, it has a small number relative to Deep Learning requirements and that was the reason for moving from Psi to Psi-Lite as explained before.

4.2.3 Verilog Code

We used Verilog as a hardware description language to implement psi-lite on Intel FPGA Cyclone V A9[13]. In the following subsections we will show the output from each part of Psi-Lite

[17] Numbers x40 and x115 refer to number of kilo logic elements available on bladeRF, while number xA4 and xA9 refers for key that Intel gives to its FPGAs (e.g. xA9 stands for Cyclone V A9 FPGA)

parts. Also, in the following as weights and data are small, we saved them on FPGA memory. We assume that A/D will sample data in the air then send them to dedicated memory blocks, where we specified two blocks for that purpose. FPGA contains two types of memory M10k and MLAB[18]. M10k memory contains 1220 blocks where we store weights on them



RF Specifications	Min	Typ	Max	Unit
ADC/DAC Sample Rate	0.521		61.44	MSPS
ADC/DAC Resolution		12		bits
RF Tuning Range (RX)	70		6000	MHZ
RF Tuning Range (TX)	47		6000	MHz
RF Bandwidth Filter	<0.2		56	MHz
CW Output Power		+8		dBm
FPGA Specifications	**Min**	**Typ**	**Max**	**Unit**
Logic Elements	49 (xA4)		301 (xA9)	kLE
Memory	3,383 (xA4)		13,917 (xA9)	kbits
Variable-precision DSP blocks	66		342	
Embedded 18x18 Multipliers	132		684	

FIGURE 55: BLADERF CHARACTERISTICS.

4.2.3.1 IQ Convolution Layer output

IQ branch needs two weights per clock or we can take them on two clocks as convolution has shared weights[19] so we putted weights of In-Phase branch and Quadrature branch on the same Memory block. As shown in Table 7, IQ branch has 140 weights multiplying by 12 bits, 11 bits as explained and 1 bit for sign (positive or negative weight), gives 1680 bits which less than M10k size.

[18] M10k is 10k bits memory dedicated for storing while MLAB stands for Memory logic array blocks which can be used as logic elements or as memory units.

[19] Shared weights mean that for different input size, the filter weights are the same

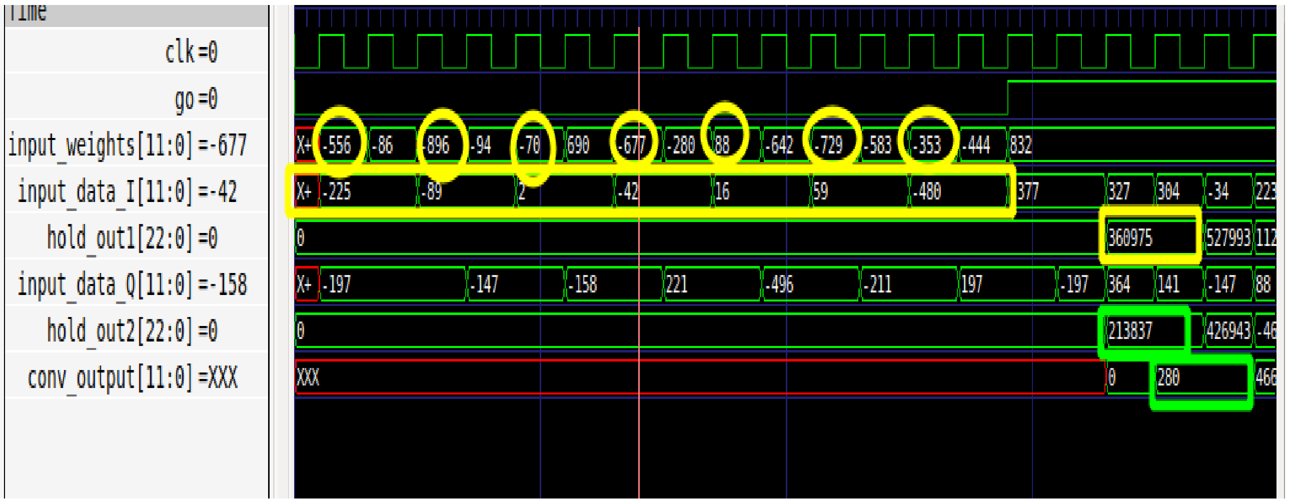


FIGURE 56: IQ CONVOLUTION BRANCH OPERATIONS.

Figure 56 shows the operation of IQ convolution block, input weights in yellow is the In-Phase branch weights and multiplied by input_data_I then sum gives hold_out1. Weights after yellow circles are Quadrature-phase branch weights and multiplied by input_data_Q then sum gives hold_out1. Sum hold_out1 and hold_out2 and shift by 11[20] bits gives conv_output.

4.2.3.2 Amplitude Convolution Layer output

Amplitude branch doesn't need more than one weight per clock and it has 56 weights, as shown in Table 7, multiplying by 12 bits gives 672 bits which enabled us putting them on a single M10k block.

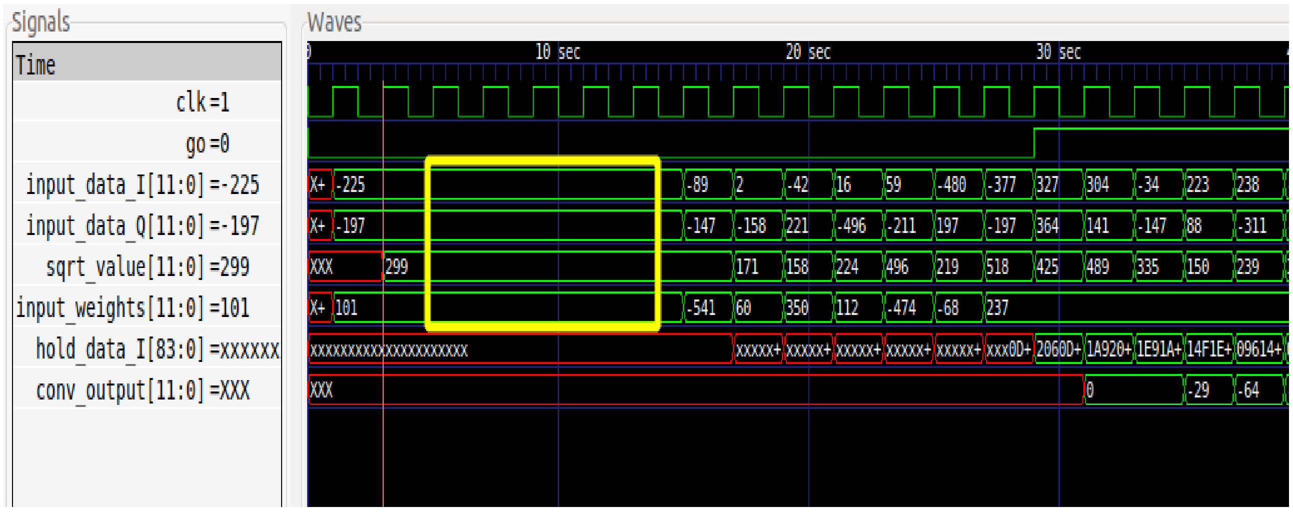


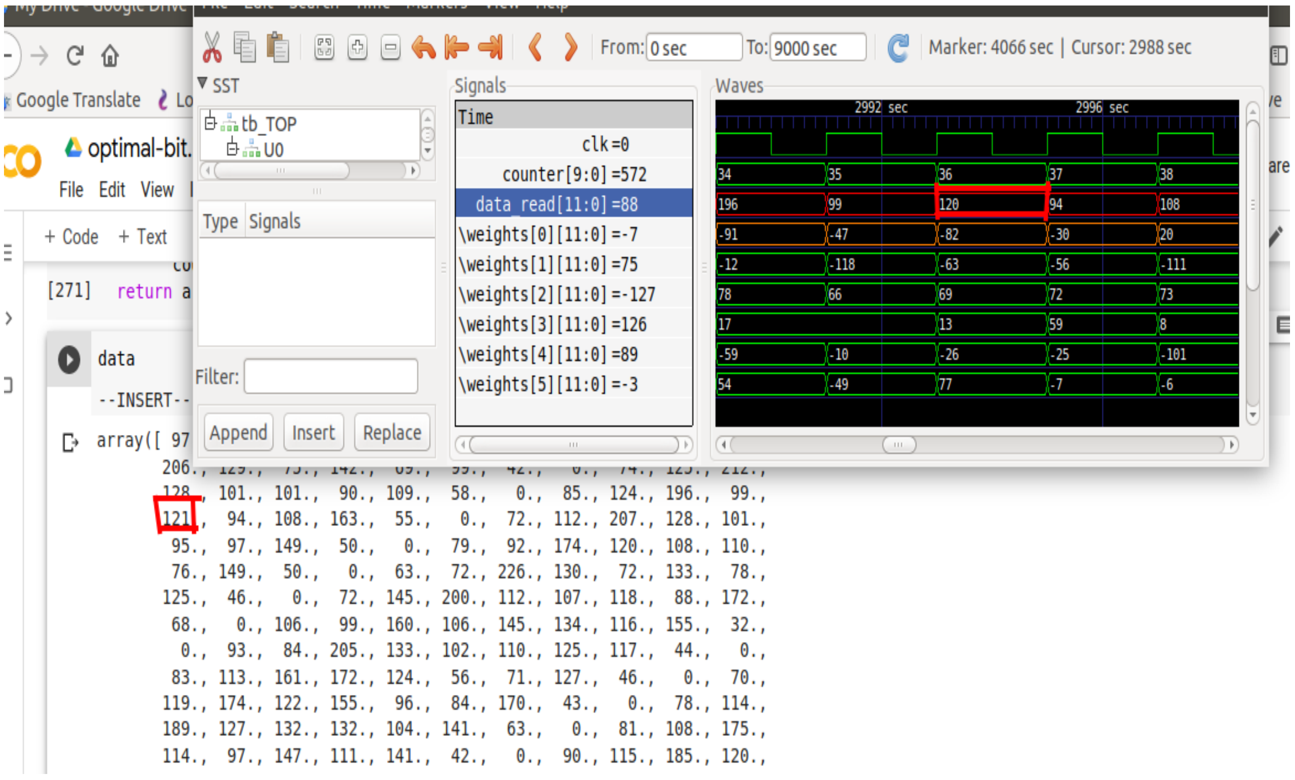
FIGURE 57: AMPLITUDE CONVOLUTION BRANCH OPERATIONS.

[20] We hold the output from multiplication in 23 bit not 24 as multiplying two bits sign will result in one bit not two

Figure 57 shows the operation of IQ convolution block. The space shown in yellow is to synchronize the Amplitude branch with IQ branch[21]. To calculate Square root, we used a function available on the internet [15]. Square root output has been held in 12 bit which can hold up to 4096 bigger than biggest output from Amplitude branch as shown in Equation 23.

$$\sqrt{X_i^2 + X_q^2} = \sqrt{(2^{11})^2 + (2^{11})^2} = \sqrt{2^{22} + 2^{22}} = \sqrt{2^{23}} = 2^{(23/2)} = 2896 \quad \text{equation(23)}$$

Square root function contains for loop to calculate the square, increasing for loop number leads to increasing accuracy in cost of clock period which takes bigger time. So, to compromise between two operations, we allowed a small difference in calculations relative to software, as shown in Figure 58. as the difference is small, so it doesn't affect the final output or final class prediction as will be seen in Dense Layer below.



```
From: 0 sec   To: 9000 sec   Marker: 4066 sec | Cursor: 2988 sec

Signals                      Waves   2992 sec            2996 sec
Time
clk=0
counter[9:0]=572             34    35    36    37    38
data_read[11:0]=88           196   99    120   94    108
\weights[0][11:0]=-7         -91   -47   -82   -30   20
\weights[1][11:0]=75         -12   -118  -63   -56   -111
\weights[2][11:0]=-127       78    66    69    72    73
\weights[3][11:0]=126        17          13    59    8
\weights[4][11:0]=89         -59   -10   -26   -25   -101
\weights[5][11:0]=-3         54    -49   77    -7    -6
```

```
[271]  return a

data
--INSERT--

array([ 97
        206., 129.,  75., 142.,  69.,  99.,  42.,   0.,  74., 125., 212.,
        128., 101., 101.,  90., 109.,  58.,   0.,  85., 124., 196.,  99.,
        121.,  94., 108., 163.,  55.,   0.,  72., 112., 207., 128., 101.,
         95.,  97., 149.,  50.,   0.,  79.,  92., 174., 120., 108., 110.,
         76., 149.,  50.,   0.,  63.,  72., 226., 130.,  72., 133.,  78.,
        125.,  46.,   0.,  72., 145., 200., 112., 107., 118.,  88., 172.,
         68.,   0., 106.,  99., 160., 106., 145., 134., 116., 155.,  32.,
          0.,  93.,  84., 205., 133., 102., 110., 125., 117.,  44.,   0.,
         83., 113., 161., 172., 124.,  56.,  71., 127.,  46.,   0.,  70.,
        119., 174., 122., 155.,  96.,  84., 170.,  43.,   0.,  78., 114.,
        189., 127., 132., 132., 104., 141.,  63.,   0.,  81., 108., 175.,
        114.,  97., 147., 111., 141.,  42.,   0.,  90., 115., 185., 120.,
```

FIGURE 58: SQUARE ROOT DIFFERENCE BETWEEN SOFTWARE AND HARDWARE.

[21] There is one clock difference as explained above for the MaxPool layer operations

4.2.3.3 Max-Pool Layer output

Max-Pool layer has no parameters, so it doesn't deal with memory. Max-Pool layer operations was explained in Psi-Lite software, section 3.2.4, and its operation with real data is shown in Figure 59. Yellow rectangle areas show no-operation and Max-Pool waits previous layers, either IQ_conv or Amplitude_conv[22]. Blue shows zero and that means output from Amp_Phase was negative and we compare and either output zero or biggest value between two consecutive values.

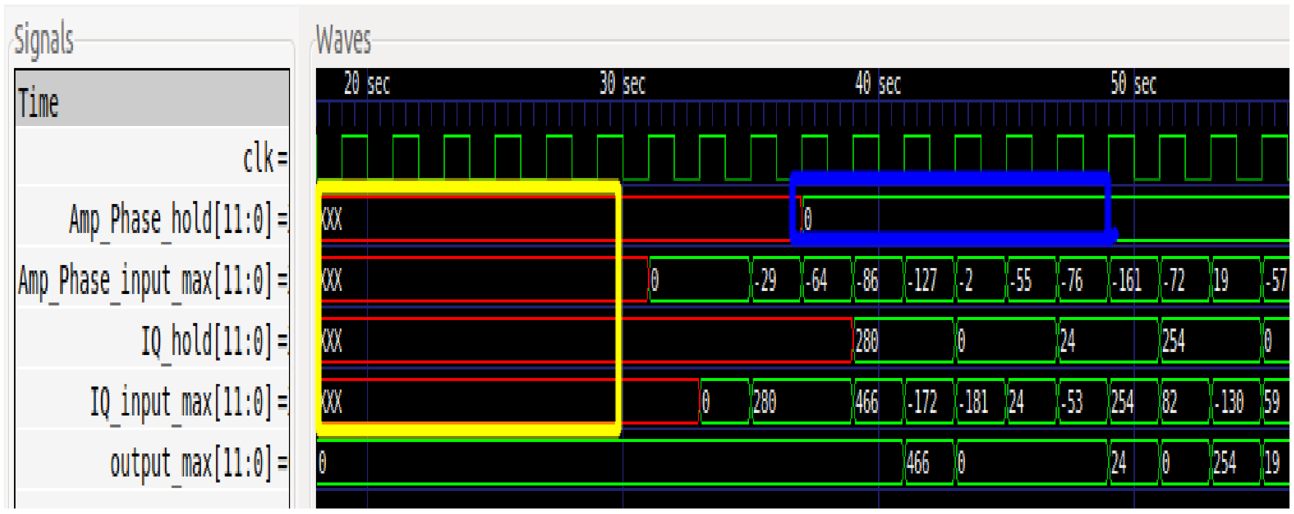


FIGURE 59: MAX-POOL BRANCH OPERATIONS.

4.2.3.4 Shared Convolution Layer output

To synchronize shared convolution layer with its previous inputs, we need to load weights on 25 clocks. As we need 100 weights[23], so we need to load four samples per clock. To solve this problem, we distributed weights of shared convolution layer on four M10K blocks. Shared convolution layer has 900 weights, as shown in Table 7, So each block will have 225 weight multiplying by 12 bits gives each block 2700 bits.

Figure 60 shows operations of shared convolution layer. IQ_or_Amp_Phase is a signal which works as a clock or oscillator to make Shared Convolution layer either multiply weights of IQ branch or Amplitude branch which leads to truncated number of multiplications required to half and also required number of DSP blocks.

[22] Here written Amplitude_Phase_hold as a general case and to be used in the future with Psi Model
[23] Shared convolution has ten filters, each filter size is 5. So we need to multiply 5*10 per clock, given that Shared convolution deals with two branches, So we need 100 weights to be loaded before MaxPool data complete 5 samples

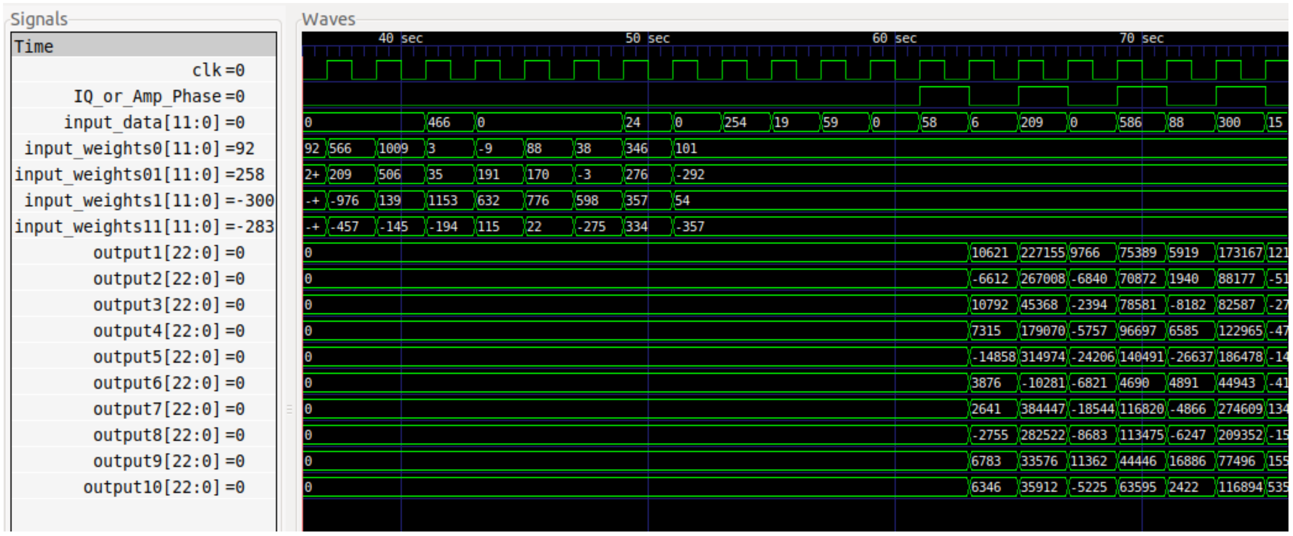


FIGURE 60: SHARED CONVOLUTION LAYER OPERATIONS.

4.2.3.5 Dense Layer output

Finally to synchronize Dense layer with Shared Convolution layer, we need 15 memory blocks, each block works with a neuron, as Dense has 15 neuron with weights 8850, as shown in Table 7, so each block has 590 weights multiplying by 12 bits gives 7080. Figure 61 shows operations of Dense Layer. Dense layer unlike convolution, which has shared weights, has weights for each input enter it[24].

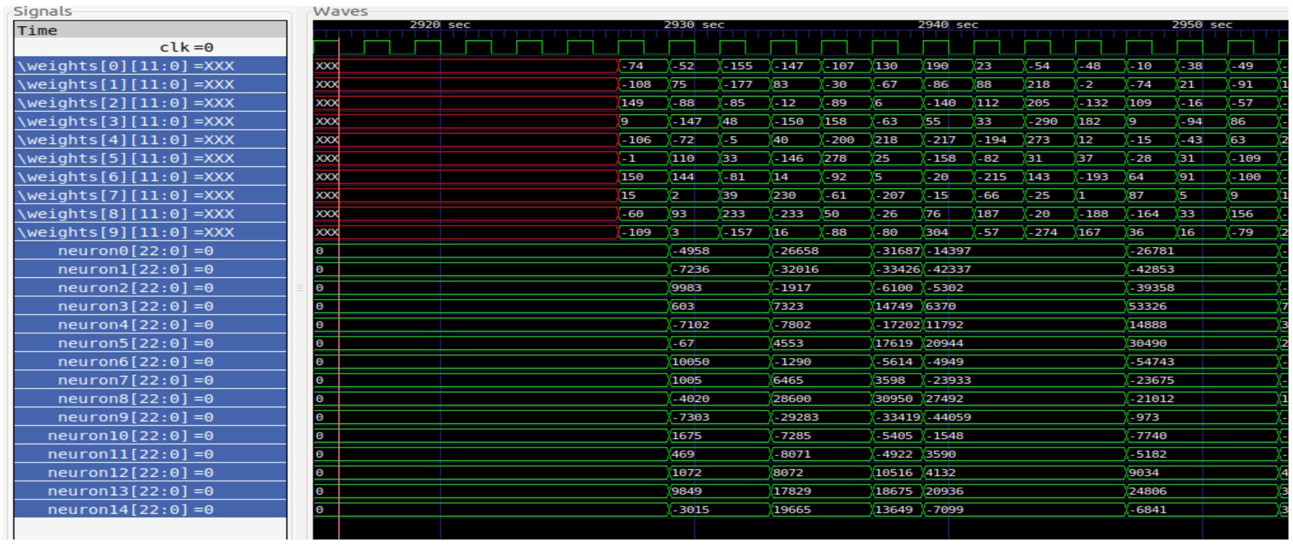
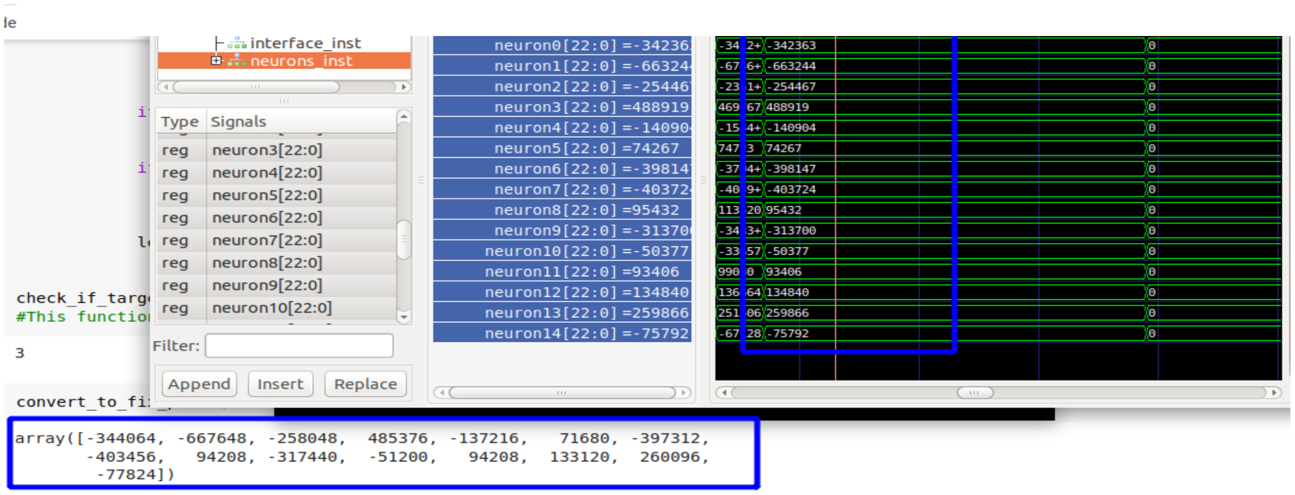


FIGURE 61: DENSE LAYER OPERATIONS.

[24] That is why it's called also fully connected. Dense means that it has many weights

4.2.3.6 Max-Pool Layer output

This is the final layer that targets making classification for the output of the dense layer. As Circuit works on multiple filters[25] dense layer puts its data into memory interface and in the next filter operation, it calls the data from the memory and add the new data for it and push the data again. After the end of classification, Max-Pool Layer calls data from the interface and choose the biggest between them. Figure 62 shows the output and compare them with the output of software. We can note that output from hardware is different from software due to square root in the Amplitude branch as discussed in 5.2.3.3 above but the final result doesn't affect with this difference as the output is very big relative to the difference caused by square root.



```
check_if_targ
#This functio
3
convert_to_fi
```

Type	Signals
reg	neuron3[22:0]
reg	neuron4[22:0]
reg	neuron5[22:0]
reg	neuron6[22:0]
reg	neuron7[22:0]
reg	neuron8[22:0]
reg	neuron9[22:0]
reg	neuron10[22:0]

```
 neuron0[22:0] =-34236   -34 2+ -342363   0
 neuron1[22:0] =-66324   -67 6+ -663244   0
 neuron2[22:0] =-25446   -23 1+ -254467   0
 neuron3[22:0] =488919   469 67 488919    0
 neuron4[22:0] =-14090   -15 4+ -140904   0
 neuron5[22:0] =74267    747 3  74267     0
 neuron6[22:0] =-39814   -37 4+ -398147   0
 neuron7[22:0] =-40372   -40 9+ -403724   0
 neuron8[22:0] =95432    113 20 95432     0
 neuron9[22:0] =-31370   -34 3+ -313700   0
neuron10[22:0] =-50377   -33 57 -50377    0
neuron11[22:0] =93406    990 0  93406     0
neuron12[22:0] =134840   136 64 134840    0
neuron13[22:0] =259866   251 06 259866    0
neuron14[22:0] =-75792   -67 28 -75792    0
```

```
array([-344064, -667648, -258048,  485376, -137216,   71680, -397312,
        -403456,   94208, -317440,  -51200,   94208,  133120,  260096,
         -77824])
```

FIGURE 62: FINAL OUTPUT COMPARISON.

4.2.4 Memory problem

Noting in Shared Convolution Layer or in Dense Layer that although two memory blocks are enough for Shared Layer or eleven blocks for Dense Layer, but we used four and fifteen blocks respectively. We partitioned the memory blocks to increase the speed and achieve the required synchronization but with the cost of memory lost. In real applications they use one of two solutions for this problem, one of them is implementing ASIC (application specific integrated circuit) and buying suitable memory sizes, even if memory is standard but they can search for a memory with the most suitable size to reduce wasted size. The other solution depends on tackling the difference between the memory speed and the circuit speed which is called multi-

[25] IQ convolution branch has ten filters and Amplitude branch has eight, if they worked together, this will require at least ten times the current used DSP, expected 750, whereas Intel Cyclone V xA9 has only 342 which will make the whole process of design failed and prevent using bladeRF. So, we made IQ conv and Amplitude conv makes filter, two filters from each works simultaneously, which required ten times time.

pumping [14]. Multi-Pumping depends on memory works with speed multiple of circuit speed[26]. So, we can take more than one sample from memory at the same cycle. Figure 63 shows multi-pumping concept. Multi-Pumping solution intended to be used with Psi implementation in a future work as Psi will require more samples per clock which will leads to bigger partition of memory.

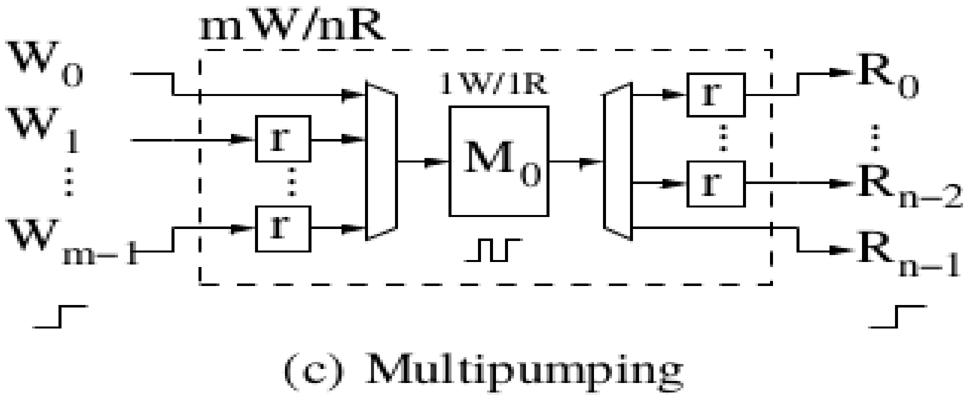
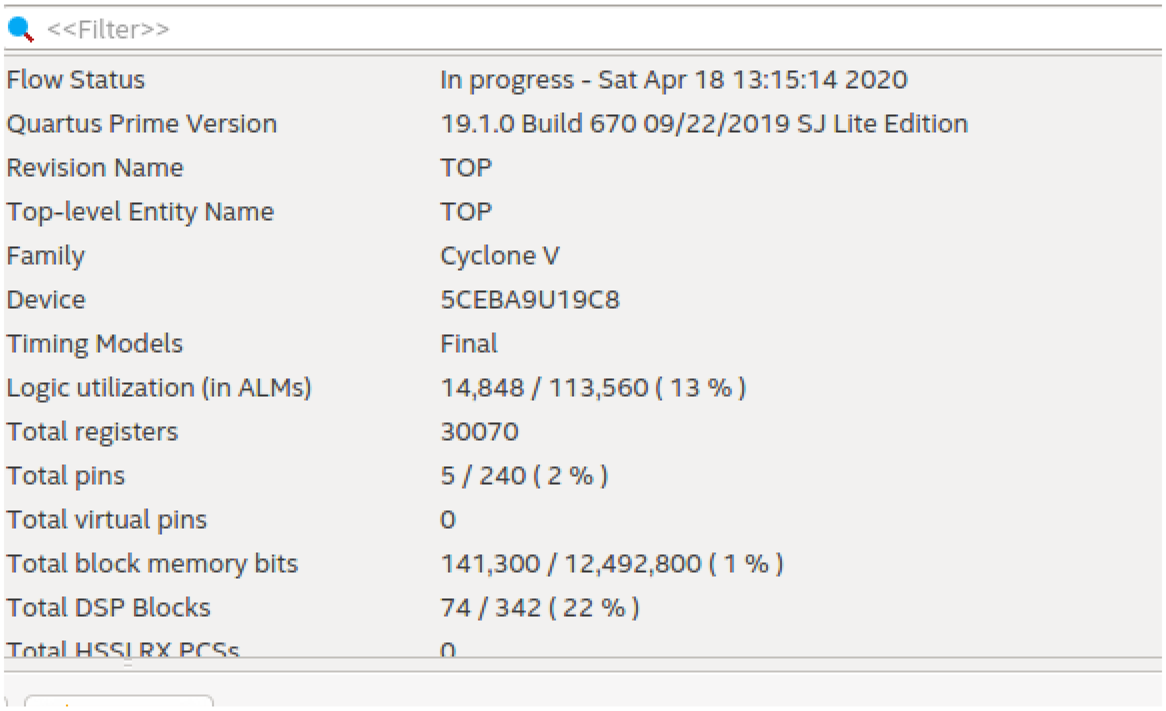


FIGURE 63: MULTI-PUMPING SOLUTION [14].

4.2.5 Verilog Synthesis

We used Intel Quartus Prime Lite 19.1 as it's a free edition and has support for Intel Verilog Cyclone V family. **Error! Reference source not found.** shows the synthesis of the Verilog Code. We can note that we use a small number of Memory and logic elements, but a big number of DSP blocks available on the chip. We can move to Cyclone V xA4 instead of xA9 but we need to use methods to reduce the number of required DSP blocks so it reduces from 74 required block to 66 block which are available on the bladeRF xA4. One of these methods can be by making convolution layer works then stop and neurons works or a combination of them works to reduce the required DSP but this will come in cost of speed required for communication applications, and in case of combination of convolution and dense layers will leads to additional complexity of the design. Another method which can be used is multi-pumping in DSP blocks [16] which can be used in the implementation of Psi Model in a future work.

[26] This leads also to make design first with memory partition to see the speed of the circuit then redesign the memories blocks to take more than one sample per clock.


<<Filter>>

Flow Status	In progress - Sat Apr 18 13:15:14 2020
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	TOP
Top-level Entity Name	TOP
Family	Cyclone V
Device	5CEBA9U19C8
Timing Models	Final
Logic utilization (in ALMs)	14,848 / 113,560 (13 %)
Total registers	30070
Total pins	5 / 240 (2 %)
Total virtual pins	0
Total block memory bits	141,300 / 12,492,800 (1 %)
Total DSP Blocks	74 / 342 (22 %)
Total HSSI RX PCSs	0

FIGURE 64: PSI-LITE SYNTHESIS.

4.2.6 Results

As there isn't any FPGA work regarding the Wireless interference Identification until writing these lines. So, we will not be able making comparison with literature. Table 15 shows our results. Memory bits equal weights * bits representation + data interface, where weights = 9646 and number of bits to represent data is 12 bits. We can note that we make test/ interference identification 20000 each second. Number of tests is big or small depends on the application and band that it operates at (e.g. free band like ISM will be more interference than communication bands that companies hold). We can note also that we consume little dynamic power which prove that FPGA size is big and it's better to move for smaller one.

TABLE 15: PSI-LITE HARDWARE RESULTS

Model	Memory bits	DSP	Speed	Total Power	Dynamic Power	Static Power
Ψ Lite	141,300 bits	74	20k	660.92 mW	128.55 mW	526.26 mW

CHAPTER FIVE: TOOLS

5.1 Software tools

The software tools used in this work is python as the programming language as it is the most used language used by experts on the field also top deep learning libraries are available on the Python ecosystem like Theano and TensorFlow. Tap into their power in a few lines of code using Keras, the best-of-breed applied deep learning library.

Used libraries are PyTorch and Keras used in Deep Learning for construction the model, train it, and test it. The other libraries used in python are NumPy for multidimensional array operations, Matplotlib which is used to draw the graphs and charts illustrating the model characteristics and performances as it is a comprehensive library for creating static, animated, and interactive visualizations in Python, and Pandas which is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

5.2 Hardware tools

Verilog HDL is used as the hardware description language used to implement the model on FPGA as it is a powerful language and the most used one in industry.

Synthesis tools used in the project are Quartus for wireless interference identification in the PSI Lite mode which will be illustrate later on section 2 of chapter 5, and Vivado for modulation recognition in the optimized in-phase and quadrature model which will be illustrated in section 1 of chapter 5.

Targeted FPGAs are ZYNQ Ultra scale 104 for modulation recognition in the optimized in-phase and quadrature model which will be illustrated in section 1 of chapter 5, Cyclone V A9 for wireless interference identification in the PSI Lite mode which will be illustrate later on section 2 of chapter 5.

CHAPTER SIX: CONCLUSION

To conclude our work. First of all, we studied the literature of the field and compared between the provided solutions then reproduced the best work done to make sure we have the base to be compared with our proposed solutions. We took into our consideration the size of the model as the intended goal of our work was to implement the most efficient model from the proposed ones on FPGA, Increasing the accuracy also was one of our intended goals. The most suitable model that combined between lower in size and higher in accuracy than the literature was Multipath Model named PSI-Model which is the first multipath model in the literature in both ways either modulation recognition or wireless interference. After that, we proposed an optimized In-phase and quadrature phase model for modulation recognition and after testing in software level it showed us relative reduction in the size compared to PSI model without any need for any other signal processing unit. However, we had to scarify with its accuracy to become lower than the PSI model. Consequently, we implemented the optimized IQ model on FPGA ZYNQ Ultra scale+ MPSoC ZCU104 which shows high accuracy with very low size and much higher speed than GPU. Last but not least, we managed to propose the PSI lite model with two branches only from the full PSI model in order to identify the wireless interference. The model scarified 5% of the full PSI model accuracy to be suitable for implementation on small FPGA with RF model to make the real-time wireless signal identification possible. The PSI lite model was implemented on FPGA Cyclone V A9 which was the first model to be implemented on FPGA for wireless interference identification purpose.

CHAPTER SEVEN: FUTURE WORK

The future work to be done is:

I. Testing PSI Lite with real time data and detecting source of interference at that data.

II. Making real time modulation recognition and wireless interference with PSI model on FPGA by providing bigger FPGA on RF circuit to process the big number of FLOPS (I.e multiplication-addition) of PSI Model.

III. Generate a more efficient data set to increase model accuracy, and tackle different problems such as detecting spectrum holes in the frequency spectrum.

REFRENCES

1. T. J. O'Shea, J. Corgan, and T. C. Clancy, ``Convolutional radio modulation recognition networks,'' in *Proc. Int. Conf. Eng. Appl. Neural Netw.*, 2016,pp. 213226.
2. Sohraab Soltani., (2019). ``Real-Time and Embedded Deep Learning on FPGA for RF Signal Classification.''
3. M. Schmidt, D. Block and U. Meier," Wireless interference identification with convolutional neural networks," 2017 IEEE 15th International Conference on Industrial Informatics (INDIN), Emden, 2017, pp. 180-185.
4. M. Kulin, T. Kazaz, I. Moerman and E. De Poorter," End-to-End Learning from Spectrum Data: A Deep Learning Approach for Wireless Signal Identification in Spectrum Monitoring Applications," in IEEE Access, vol. 6, pp. 18484-18501, 2018.
5. UG1267 - ZCU104 Board User Guide
6. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/ss-cyclone-v-fpgas.pdf
7. https://www.nuand.com/product/bladerf-xa9/
8. A. Karpathy. Convolutional Neural Networks for Visual Recognition [Online]. Available: http://cs231n.github.io/convolutional-networks/ , 2018. 4
9. *Solovyev, Roman A., Alexandr A. Kalinin, Alexander G. Kustov, Dmitry V. Telpukhov and Vladimir S. Ruhlov. "FPGA Implementation of Convolutional Neural Networks with Fixed-Point Calculations." ArXiv abs/1808.09945 (2018)*
10. *F. Chollet et al. (2015). Keras. [Online]. Available:* https://github.com/fchollet/keras
11. Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam. "Automatic differentiation in PyTorch." NIPS-W, 2017
12. https://www.nuand.com/bladerf-2-0-micro/
13. https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf
14. Charles Eric LaForest and J. Gregory Steffan. 2010. Efficient multi-ported memories for FPGAs. In Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '10).
15. https://verilogcodes.blogspot.com/2017/11/a-verilog-function-for-finding-square-root.html
16. Bajaj, Ronak & Fahmy, Suhaib. (2015). Minimizing DSP block usage through multi-pumping. 184-187. 10.1109/FPT.2015.7393146.