THE COOPER UNION

ALBERT NERKEN SCHOOL OF ENGINEERING

OFDM Modulation Recognition Using Convolutional Neural Networks

by

Justin Alexander

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Engineering

Advised by: Dr. Sam Keene

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

_____

Dr. Richard J. Stock - Apr. 26 2017

Dean, School of Engineering

_____

Dr. Sam M. Keene - Apr. 26 2017

Candidate's Thesis Advisor

Acknowledgements

First and foremost, I would like to thank my God, Lord Almighty, for giving me wisdom, strength, and courage to finish this project.

I would also like to thank my family, especially my parents, for their support and encouragement. They pushed me to aim high, and that is why these pages are being written.

I would also like to thank my thesis adviser Dr. Sam Keene for his constant guidance over this project and my career life. Thank you for helping me succeed in both.

I wouldn't have made it through the data capture stage without the experience I gained from working at MaXentric Technologies. Special thanks to Brian Woods for his help in debugging my code.

Special thanks to Chris Curro for teaching me Neural Networks, and for being a great friend.

ABSTRACT

The task of determining the relevant parameters of a transmission scheme is known as modulation classification. Possible parameters of interest include carrier frequency, symbol time or modulation order. In this thesis we focus on modulation format (linear vs OFDM) and modulation order. This task has applications in signal intelligence receivers (SIG INT), and cognitive radios. One possible application is in military use, where friendly or non-friendly transmissions can be identified. Another possible application is in a cognitive radio network, where knowledge of primary users transmission scheme could be used as part of an underlay network scheme. A common approach for modulation classification is support vector machines (SVM) with high order cumulants as features, however the problem of classification of orthogonal frequency division multiplexing (OFDM) signals has not been fully explored. Furthermore, deep neural networks (DNN) have made tremendous advances in classification problems, and there has been no prior work done on using live captured data to test modulation classification using DNN. Therefore, four linear and OFDM modulations are captured live over a range of signal powers, and tested against with both SVM and DNN classifiers. The SVM classifier with high order cumulant features achieved a classification rate of 99% for OFDM modulations, but only achieved 93% accuracy for linear modulations. A convolutional neural network (CNN) achieved 99% classification for all 8 modulations. Additionally, the CNN generalizes better than the SVM classifier when trained over a range of SNR values. When trained in this manner, the convolutional network significantly outperforms the SVM classifier when the SNR value is not known at the receiver.

Contents

List of Figures

Nomenclature

$Adam$ Adaptive Moment Estimation

BER Bit Error Rate

BN Batch Normalization

CNN Convolutional Neural Network

CR Cognitive Radio

DNN Deep Neural Network

FDE Frequency Domain Equalizer

FSK Frequency Shift Keying

GD Gradient Descent

GRC GNU Radio Companion

HOC Higher Order Cumulants

ISI Inter Symbol Interference

MLP Multi Layer Perceptrons

$NNet$ Neural Network

OOT Out Of Tree

$ORBIT$ Open-Access Research Testbed for Next-Generation Wireless Networks

PCA Principal Component Analysis

PRT Pattern Recognition Toolbox

QAM Quadrature Amplitude Modulation

RBF Radial Basis Function

$ReLU$ Rectifier Linear Unit

RRC Root Raised Cosine

RX Receiver

SDR Software Defined Radios

SNR Signal to Noise Ratio

SVM Support Vector Machine

TX Transmitter

UHD USRP Hardware Driver

$USRP$ Universal Software Radio Peripherals

*

1

Introduction

1.1 Motivation for Research

There are many uses for modulation recognition algorithms; one of its primary uses are in a signal intelligence (SIG INT) receivers. A SIG INT receiver's job is to monitor and intercept foreign communication signals, and gain information regarding the message being transmitted. To do such task, a SIG INT receiver needs to know what type of modulation has been applied on the transmitted signal. This is where a robust modulation recognition algorithm, which is able to classify among various modulation type, comes in useful.

Modulation algorithms can also be used in cognitive radios (CR). CR are programmed for efficient utilization of the frequency spectrum for transmission which is being shared among multiple users [2]. This is typically done by detecting and transmitting when the channel is unoccupied. A modulation recognition algorithm can be used here to detect

if a channel is in use. CR can perform more intelligently in an underlay network if the underlying modulations of the primary users is also known.

There has been much research into modulation recognition algorithms for many different modulation classes. This thesis will be expanding upon the research by applying neural networks (NNet) to modulation classification. Many experimenters have already done this, but with synthesized data. Current state of the art classifiers are support vector machine (SVM) with high order cumulants (HOC) as its features. Neural networks are a more recent type of classification algorithm, and their effectiveness on classifying modulation schemes has not yet fully been explored.

1.2 Related Works

There has been some prior work into classifying OFDM modulations. In [12] the authors design a system for OFDM signal classification. In their model a gaussianity test is done to distinguish between OFDM and single carrier modulations. Furthermore, their system then estimates the relevant parameters of the OFDM modulations, such as the OFDM symbol rate, cyclic prefix interval, and number of sub carriers. They have noted that their system is able to classify between OFDM and non OFDM signal with 80% accuracy, and their parameter extraction accuracy is greater than 90% for SNR above 15 dB.

In [26], the authors apply convolutional neural nets (CNN) to classify several lin-

ear modulations. Digital modulations that were classified includes BPSK, QPSK, 8PSK, 16QAM, 64QAM, BFSK, CPFSK, and PAM4. They also included 3 analog modulations: WB-FM, AM-SSB, and AM-DSB. Their dataset was synthesized in GNU Radio using harsh channel models. Their CNN was trained with SNR data ranging from -20dB to +20dB. Their overall accuracy across all the SNR levels was roughly around 87%.

1.3 Problem statement

In this thesis, four linear modulations will be classified: BPSK, QAM4, QAM16, and QAM32, along with 4 OFDM based modulation: OFDM-BPSK, OFDM-QAM4, OFDM-QAM16, and OFDM-QAM32. Two different classifiers will be trained and tested for their performance in classifying these modulations. The classic SVM classifier with cumulant features are considered state of the art for the modulation classification. Neural Networks have recently become popular for their ability to learn features from complex data sets. To the author's knowledge neither of these classifier have been tested to classify OFDM modulations with live captured data. Therefore, these two classifiers will be trained and tested with live captured data sets. They will be tested for their ability to simultaneously classify both linear and OFDM modulations across several SNR levels.

This thesis is organized as follows. In chapter 2, a brief explanation of digital modulations will be given. Chapter 3 will describe how the data was captured and validated.

Chapter 4 explains the classifiers that will be used, and in Chapter 5 the results will be presented. In Chapter 6 we conclude and list the future works for interested readers.

2

Communication Basics

This chapter is presented to give a brief overview on modulation's waveform design parameters and the types of impairment that can be seen on a signal transmitted over the air. A typical communication system model can be seen in the figure 2.1.

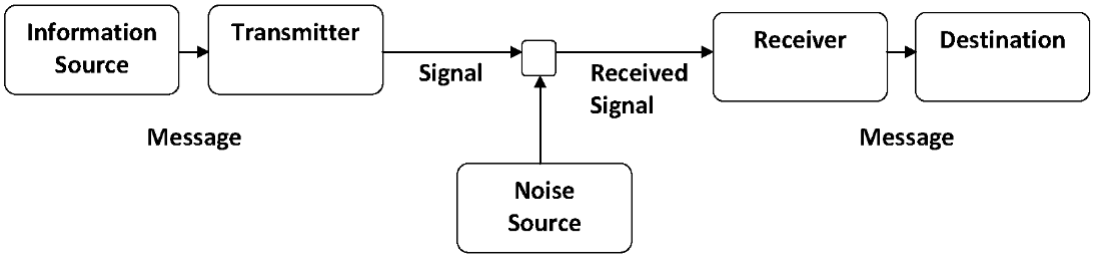


Figure 2.1: A simple communication system model. Figure taken from [11].

Information Source is the message being transmitted; these are usually in the form of a binary bit stream. *Transmitter* modulates the bit stream into the carrier frequency and transmits it into the air. The signal goes through a channel, which adds impairment to the signal. If the *receiver* is not synchronized with the transmitter then additional impairments

will be seen at the receiver. Both of these subjects, modulations and impairments, will be discussed in the next two sections.

2.1 Digital Modulations

As mentioned previously, 8 different type of digital modulation signals are classified in this research. They can be categorized as linear and OFDM modulations. OFDM modulated schemes are relatively new compared to linear modulation schemes like QAM, and they are both used widely today.

2.1.1 Linear Modulations

Researchers have come up with wide variety of digital modulations for data transmission. Most common of them all are Frequency Shift Keying (FSK) and Quadrature Amplitude Modulations (QAM). These modulations are simple to implement in a transmitter, and simple enough to build a receiver for. Figure 2.2 shows a typical transmitter for the 4 modulations of interest.

The bit streams are fed in from top level API (Host), which is mapped into complex symbols by the modulator. Those symbols are then up sampled with repetition or digitally interpolated to the sampling rate required for the transmission. Then, the complex stream is multiplied by the local oscillator (LO), which up converts the signal to radio frequency (RF).

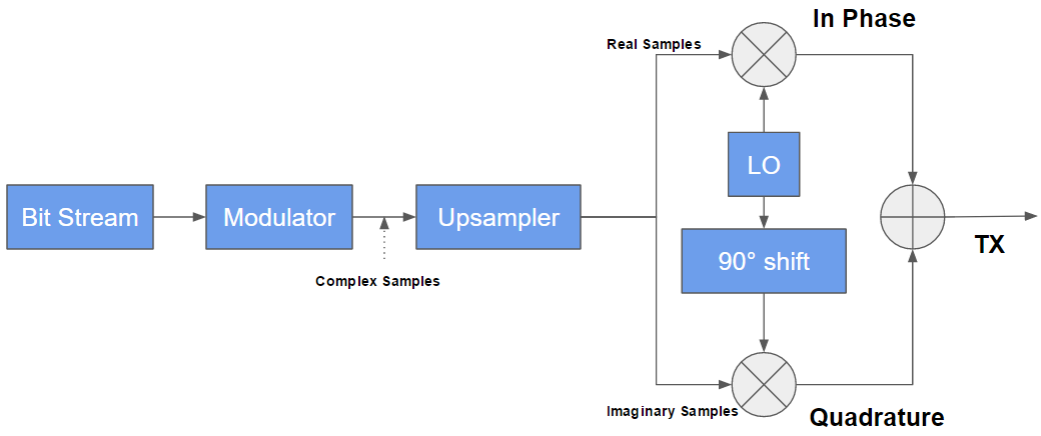


Figure 2.2: Transmitter block diagram for linear modulations.

This is the model that has been used to transmit linear modulations for this research. Pulse shaping can be applied to the signal to reduce inter symbol interference (ISI), however that is not common in modern transmitters. For this project pulse shaping is not applied, and a rectangular pulse shape is assumed for all modulations.

Equation 2.1 shows the output values of the transmitted signal:

$$s(t) = I(t)cos(2\pi f_0 t) - Q(t)sin(2\pi f_0 t) \tag{2.1}$$

where I(t) and Q(t) are the complex baseband symbols in the time domain. I stands for in-phase, and Q stands for quadrature phase. The digital modulator maps the bits onto symbol. It is easier to visualize the mapping by viewing it in a constellation diagram. The constellation diagram for the 4 linear modulations are shown in figure 2.3. The symbols are mapped based on the bit order seen on top of them.

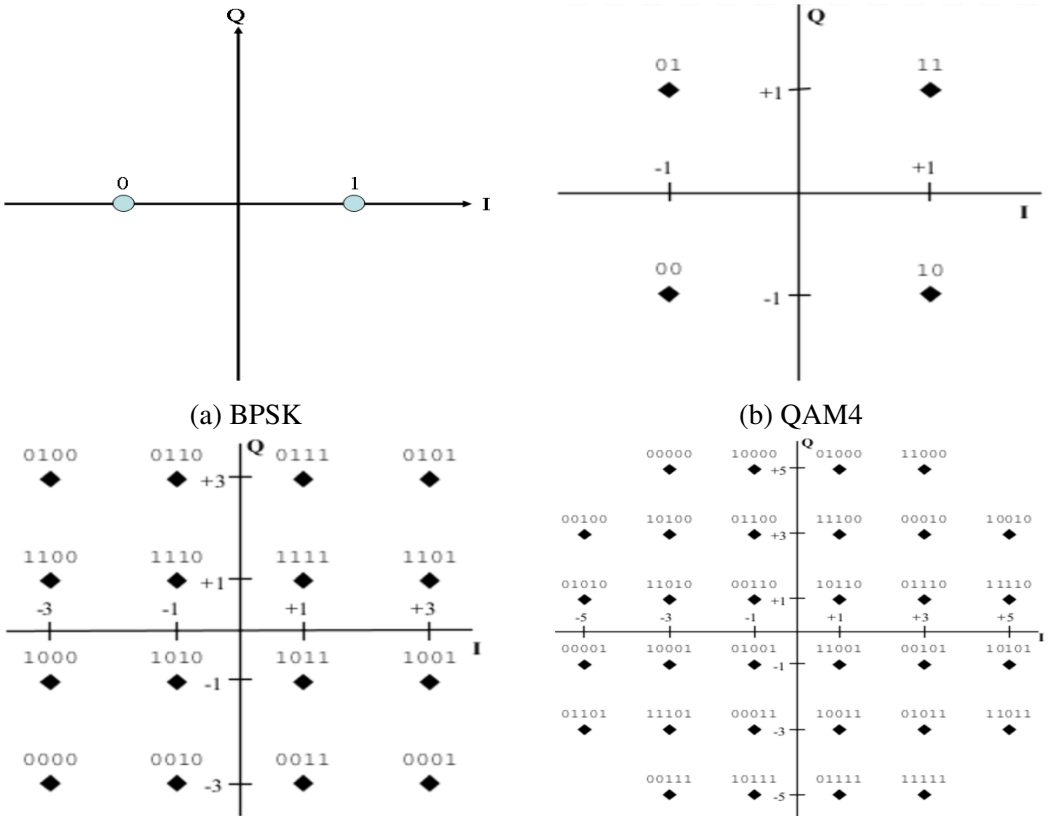


Figure 2.3: Symbol mapping for each of the 4 linear modulations. BPSK figure taken from [29]; QAM figures taken from [8].

Higher order QAM packs more bits per symbol, hence they have higher spectral efficiency compared to lower order QAM. One drawback is that the receiver has to be more complex for higher order QAMs. It is easy to track the frequency offset for BPSK or QAM4 modulations, but requires power detection circuitry for higher level QAMs. Consequently higher order QAMs tends to have higher bit error rate (BER).

2.1.2 OFDM Modulations

Orthogonal Frequency Division Multiplexed (OFDM) modulations are multi carrier modulations. These modulations are considered as wide band signals, because their bandwidth is significantly larger compared to linear modulations due to having multiple carriers. The main benefit of having multiple carrier system is that that it is robust against severe channel conditions such as multipath fading [5]. Another benefit is the ability to use frequency domain equalizer (FDE) to compensate for channel impairments. FDEs are simple to implement, and can be learned fairly quickly with the use of training frames.

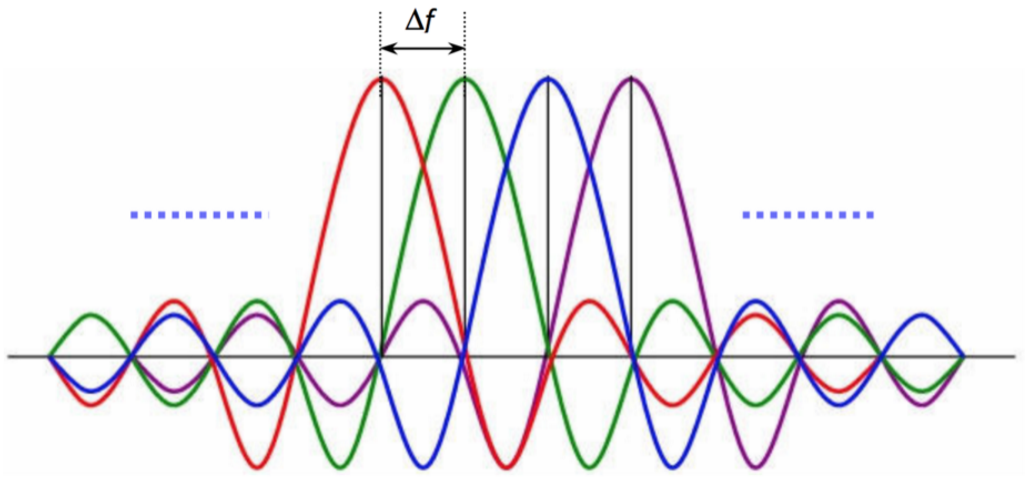


Figure 2.4: OFDM sub carriers. Figure taken from [20].

OFDM modulations are considered orthogonal because at the optimum sampling point the contribution from adjacent carriers is exactly zero as seen in figure 2.4. This alignment, however, is easily broken if there is a frequency offset, which is why the receiver needs complex algorithms to ensure frequency synchronization.

$$\Delta f = SamplingRate/N_{carriers} \tag{2.2}$$

There are many ways to implement an OFDM system; the most popular way of doing it is by using the FFT operation. This method is also used in 802.11a wireless standard [7]. The block diagram of an OFDM system using this method is shown in figure 2.5.

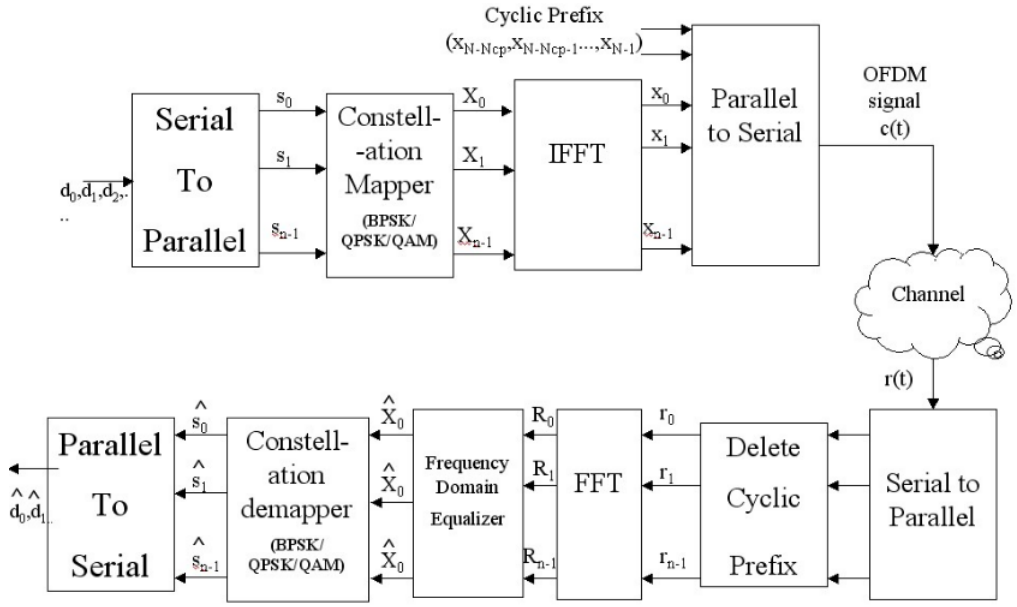


Figure 2.5: OFDM transmitter and receiver. Figure taken from [15].

The input bit stream is mapped into symbols by the *Constellation Mapper* block on a frame by frame basis. The complex symbols are then loaded into an *IFFT* block to get a time domain representation of the input signal. Cyclic prefix addition operation is performed by copying the last N_{cp} samples into the front of the frame. The frames are then serialized

by a *Parallel to Serial* block and transmitted. At the receiver, the reverse operation is done, with the addition of a *Frequency Domain Equalizer* block, which is needed to remove ISI.

OFDM signals are made up of frames. The frame size is determined by the length of FFT and cyclic prefix. For example, in the 802.11a standard, the FFT size is 64, and the CP size is 16. So there are 80 samples per OFDM frame [7]. There are different types of frame that are sent periodically. For our purposes there are only three different kinds of frames: preamble, training, and data frames. Both the preamble and the training frames are known at the receiver, which is utilized for synchronization. The receiver uses the preamble frame to detect frame boundaries. The training frames are used to learn the FDE coefficients. Once the equalizer weights are learned, it is possible to extract the transmitted symbols from the data frame.

Each FFT bin can be thought of as a separate carrier frequency. In this particular OFDM system there are 64 carriers. Typically some of the carriers are nulled to taper the frequency spectrum around the edges. Data frames, in 802.11a standard, are transmitted using the loading pattern shown in figure 2.6

There are 4 pilot carriers, # 7,21,-21, and -7, that are used in the standard. The receiver knows the values of the pilot symbols in advance, so these can be used to measure frequency offset, and also to fix timing offset. The rest of the carriers are loaded with data symbols, which are mapped by the appropriate modulator. Thus, out of 64 carriers, 48 are data carriers,

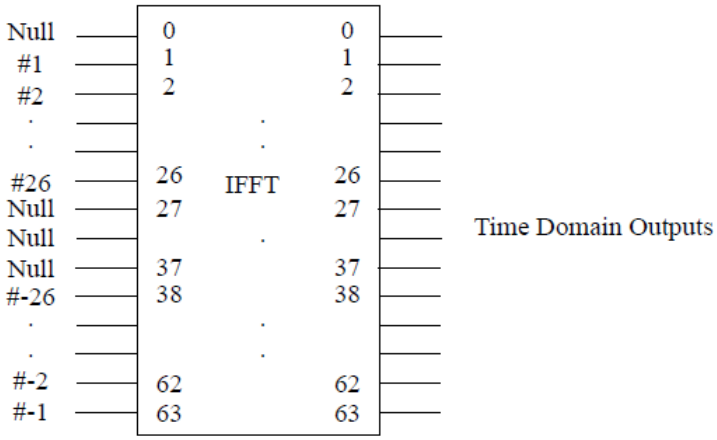


Figure 2.6: OFDM frame loading for 802.11a standard frames. Figure taken from 802.11a standard [7].

4 are pilot carriers, and 12 are null carriers.

The values of preamble frames are also taken from the 802.11a standard [7]. The training frame is a custom frame that was created to train the FDE. It is loaded the same way a data frame would be loaded, except the data sequence are BPSK symbols with alternating bits. The symbols going into consecutive data carriers are +1, -1, +1, -1 ... Data symbols from the training frame are used to compute the FDE coefficients.

2.2 Signal Impairments

There are a couple of signal impairments that are expected to be seen at the receiver. The effects are different for narrow band and wide band signals. They are listed in the following sections.

2.2.1 Path Loss

The biggest noticeable impairment to be seen is the severe attenuation of the transmitted signal while it travels over the air. This happens because signal loses its power as it covers distance. The free space path loss is shown in equation 2.3, in which we see that the signal loss in power is proportional to square of the distance traveled. Path loss is the biggest factor in signal to noise ratio (SNR) degradation for over the air transmitted signal. In the setup used for data capturing, the distance between the transmitter and receiver is not known, however they are close enough to properly transmit and receive data. The setup will be explained in detail in the next chapter.

$$FSPL = (4\pi d/\lambda)^2 \tag{2.3}$$

2.2.2 Carrier Frequency Offset

Typically the effect of this impairments can be seen in constellation diagram, especially for the linear modulations. Carrier frequency offset impairment occurs due to the mismatch between local oscillator of the transmitter and receiver, and also if either of those two are mobile. This makes the constellation of a particular modulation to rotate. Figure 2.7 shows an example of constellation rotation for QAM4 modulation.

There are many effective algorithms designed to mitigate this type of impairment, most notable of them all is the costas loop that can be applied on BPSK and QAM4 modulated

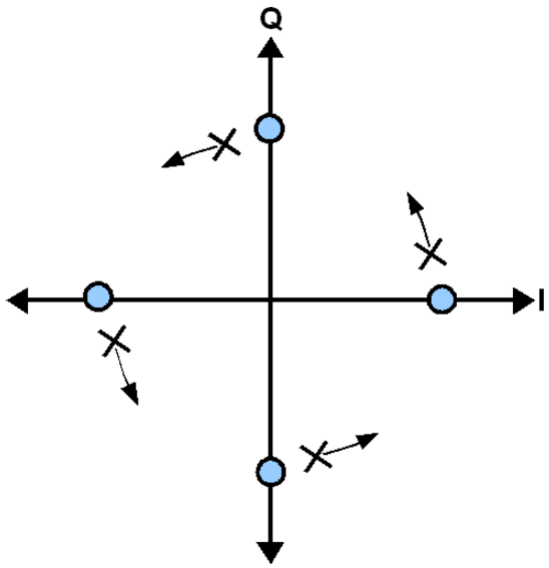


Figure 2.7: Effect of carrier frequency offset on QAM4 constellation. Figure taken from [30].

signals [25]. Variants of this algorithm are applied for higher order QAM modulations, but they are more complex.

This impairment has more profound effect on OFDM modulations because it breaks the orthogonality of the carriers. The carrier starts leaking into adjacent carriers which degrades the SNR. This impairment is termed as Inter-carrier interference (ICI). Many pilots based techniques can be implemented to detect and mitigate this particular impairment for OFDM signals [9] [13].

2.2.3 Timing Offset

This has different meaning for linear and OFDM modulations. In linear modulations, this impairment is seen when there is an offset in the clock frequency of the transmitter and receiver. TX clock frequency determines the optimal sampling time of the signal, and if RX has a different sampling time then the SNR is reduced. This impairment causes a shifting in the optimal sampling point. Algorithms like early-late gate were designed to keep track of the optimal sampling point at the receiver. Digital communications book by Michael Rice has a dedicated chapter on timing synchronization and covers early-late gate algorithm [24].

In OFDM modulations, timing offset refers to the offset between what receiver considers to be the frame boundary and the actual frame boundaries. It is important to know where the frames are starting, otherwise the demapping of known symbols from pilot and training will be incorrect. Typically, a correlator type algorithm can be used to match the frame boundaries [14].

2.2.4 Multipath Fading

Another distortion that is seen, especially in a wide band signal, is multipath fading. This arises due to multiple copies of the transmitted signal arriving at the receiver, with different delays. See figure 2.8 for an illustration. Some of the copies can combine destructively with the main dominant signal degrading the SNR. This problem is seen mostly in an urban

environments where the signal paths are longer creating a relatively large delay profile.

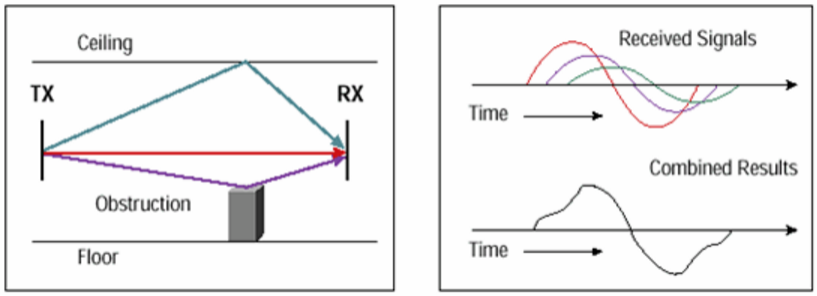


Figure 2.8: Signal degradation due to multiple copies of the signal arriving at the recevier with delays. Illustration taken from [3].

Fading creates dips in the frequency spectrum, which degrades the signal quality. The setup in which all the data has been captured is an indoor environment. Therefore the delay profile associated with that environment is small. This means that the fading shouldn't be too severe for narrow band signal. Wide band signal like OFDM modulations may experience fading. It is possible that one of the data carrier may be severely attenuated, but these dips in the spectrum can be equalized in an OFDM receiver by the FDE.

3

Data Capture and Validation

This chapter will go through the infrastructure and tools used to capture data in detail. It will also go through the methods employed to verify that the data was captured properly.

3.1 ORBIT Testbed

ORBIT stands for Open-Access Research Testbed for Next-Generation Wireless Networks [31]. It is maitained by Rutgers University, with the intent of giving researchers and experimenters the ability to test their algorithms and get reproducible results in a real environment. Going from simulated data to actual real world data is an important step in establishing the credibility of algorithm or protocols. This testbed is equipped with wide range of radio resources including: WiFi 802.11a/b/g 802.11n 802.11ac, Bluetooth (BLE), ZigBee, Software Defined Radio (SDR) platforms (USRP, WARP, RTL-SDR, USRP N210, USRP X310)[31]. For this thesis a pair of SDRs were configured to transmit and capture

live data over the air.

ORBIT lab contains several test domains in which experimenters may use the hardware for their wireless experimentation. The biggest lab setup they have is called the Grid, which contains 20 by 20 set of radio nodes. This grid is also equipped with an arbitrary waveform generator, which can add synthetic noise or interference to the grid system allowing for experimenters to create various topologies. However, for a simple job of data capturing, the Grid is not necessary. ORBIT also provides 9 sandbox lab environment which is setup with fewer number of radio nodes. Sandbox # 3 contains a pair of universal software radio peripheral (USRP2) radio nodes, which was configured and used as transmitter and receiver pair.

The best part about using ORBIT is that one can configure any of radio nodes from a remote terminal. For this project, no hardware was touched. There are two main components that had to be configured inside the ORBIT testbed. First was the hardware itself, which was used to transmit and capture the data. Second was the code that generated the data to be sent. Each code contained different message and waveform based on the modulation type used. The next two section explains the configuration done for both.

3.1.1 Software Defined Radios

Software defined radios (SDR) are relatively new and powerful tool for wireless experiments. A typical transceiver contains mixers, filters, amplifiers, modulator and demodulators, which are implemented in hardware. However, in an SDR these components are implemented in software, which gives the experimenters a lot of flexibility in the waveform design [27]. By a simple change of code, they can change the modulation type of the transmitter. SDRs are good for rapidly prototyping custom wireless communication systems. This lets the researcher implement and test his theory through an SDR relatively quickly.

As mentioned previously, a pair of USRP2s were used from sandbox # 3, provided by ORBIT. This particular sandbox contained two USRP2 N210 nodes with SBX daughter card. They are manufactured by Ettus Research, which is part of National Instrument, and met the spec needed for transmitting and capturing data. The relevant specs from the USRP2 N210's manual are listed in table 3.1.

Operating frequency	400 MHz - 4.4GHz
ADC sample rate	100 MS/s
DAC sample rate	400 MS/s
Host sample rate	25 MS/s
Clock rate	100 MHz
Bandwidth	40 MHz

Table 3.1: USRP N210 Specs [23].

The USRP2 N210's top level schematic is shown in the figure 3.1. The *Gigabit Ethernet*

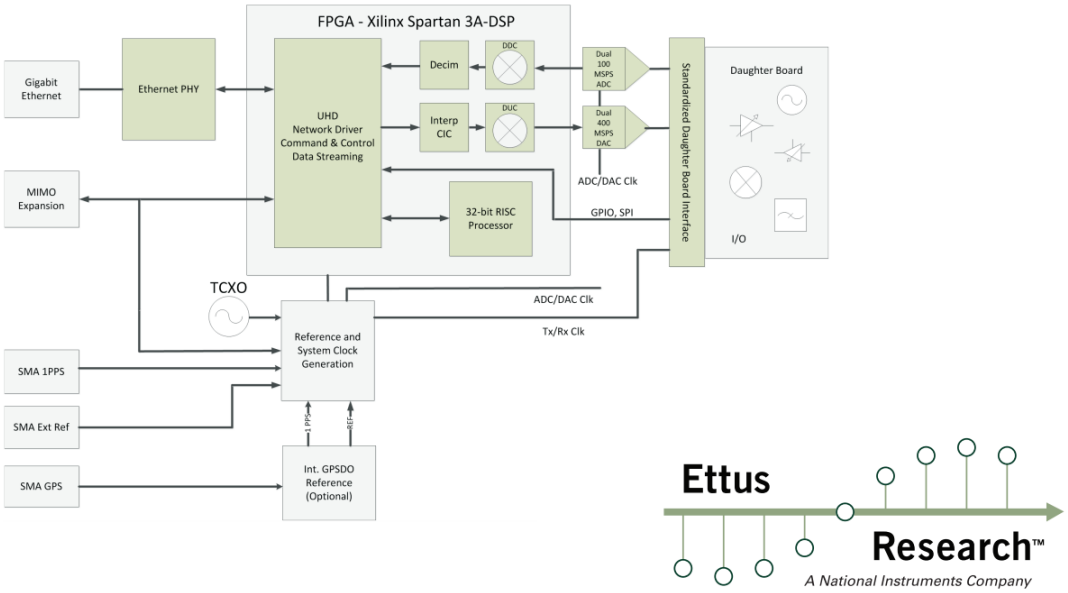


Figure 3.1: Schematic of USRP2 N210. Figure taken from USRP N210 manual [23].

block on the top left is connected to the host PC in the sandbox. That is where the modulated data is sent to, and the USRP transmits it after up conversion to RF. To create the code for designing custom modulation, Ettus Research provides USRP Hardware Driver (UHD), which is a library of functions that can be called using either C or Python to configure the SDR and to transmit data from it [22]. This requires learning the library structure, which can be done, however it is not needed. GNU Radio toolkit provides an easier way to configure the radio nodes.

3.1.2 GNU Radio

GNU Radio is a toolkit built specifically for SDR. It comes with a companion software called GNU Radio Companion (GRC), which is a graphical coding platform. Similar to MATLAB's simulink and NI's LABVIEW, in GRC, one can place digital communication blocks to customize the waveform to be transmitted through a USRP. There are many useful digital communication blocks available in GRC. Custom blocks can also be created in GRC using out of tree (OOT) module. Figure 3.2 shows a template of a simple BPSK transmitter block which was used to transmit BPSK modulated signal.

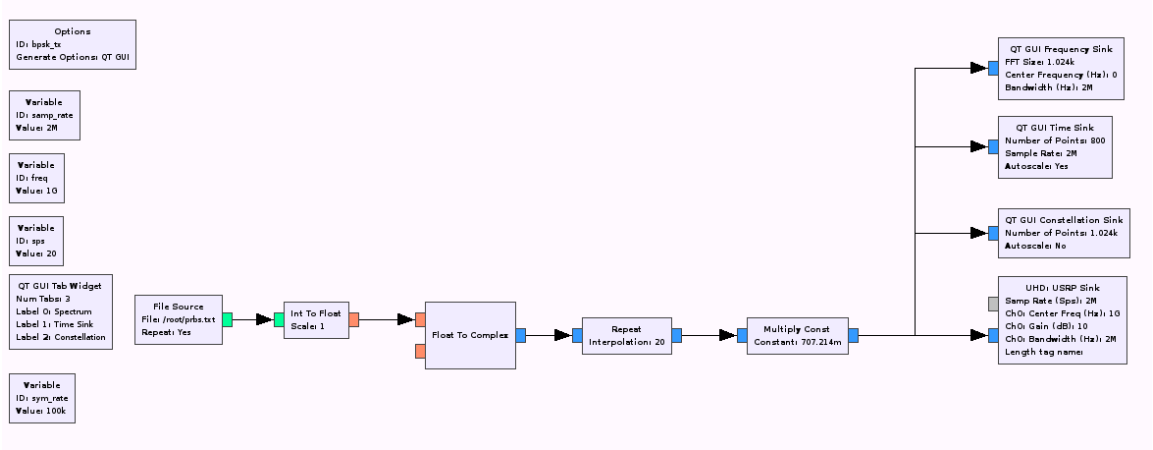


Figure 3.2: BPSK transmitter code in GNU Radio Companion.

There are a few important blocks shown in figure 3.2. The bits to be transmitted are pre generated and stored in appropriately labeled files, which can be read in using *File Source* block. These values are converted to complex samples using *Float to Complex* block. The *Repeat* block upsamples the symbols with repetition to meet the transmitter's sampling rate.

The *UHD: USRP Sink* block establishes a connection between the host API and the USRP N210 hardware through the ethernet port. The signal is fed through this block to the USRP, however it needs to be scaled so that it doesn't saturate. The *Multiply Const* block is used to ensure that the signal values does not reach saturation. The maximum float value that *UHD: USRP Sink* can accept before saturation is 1.0.

There are four important parameters that is configured via *UHD: USRP Sink* block: sampling rate, carrier frequency, bandwidth, and TX gain. In the next section, the GRC code for other modulations, QAM and OFDM will be explained. Undoubtedly this is a powerful and convenient software to use for creating and transmitting data of various modulation.

3.2 Modulation Parameters

All of the parameters listed in this section was configured through GRC. Template code for linear and OFDM modulation are also shown. There are two parameters which remained fixed for both linear and OFDM based modulations, and they are listed in table 3.2.

Carrier Frequency	1 GHz
TX Gain	0/5/10 dB

Table 3.2: Fixed Modulation Parameters

3.2.1 Linear Modulations

The GRC code for BPSK transmitter was shown in figure 3.2. The GRC code for QAM4 transmitter is shown in figure 3.3. QAM16 and QAM32 transmitter has the same blocks except *qam4_ic* is substituted with *qam16_ic* and *qam32_ic* respectively. These blocks were created using OOT module. Their implementation code can be found in the appendix. They return a QAM symbol corresponding to integer valued symbol number. Similar to BPSK transmission, the bits to be transmitted are pre generated for all the QAM modulations. For QAM4 modulations, *prbs.txt* contains integer values ranging from 0 to 3. Accordingly, for QAM16 the values range from 0 to 15, and for QAM32 the values range from 0 to 31.

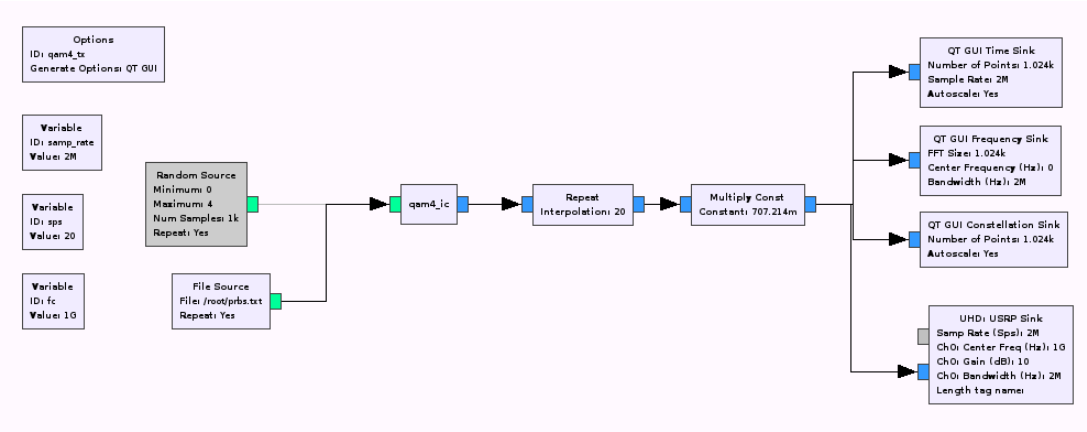


Figure 3.3: QAM4 transmitter code in GNU Radio companion

Instead of using the *Random Source* block that is provided by the GNU Radio, the random stream is fed in through a file. Through MATLAB, a list of 1000 different random integer files were generated, each with a different seed. This allows for reproducibility of the transmitted signal for debugging purposes. The code can be found in appendix A.1.1.

Typically a root raise cosine (RRC) pulse shaping filter would have been applied to combat ISI. The main reason for not using RRC is that this pulse shape is not widely used anymore in modern systems. It would have made the classification more complex.

The modulated signal is up sampled by 20 by repetition. Up sampling by 10 was the initial choice, but the corresponding captured signal's pulse shape was too small to produce an acceptable signal constellation. Since the received signal constellation was not good enough for reception we chose an up sampling rate of 20. The input sampling rate was set to 100 KHz, and after up sampling by 20, the output rate would be 2 MHz. The TX gain value was varied between 0, 5, and 10 dB, which will be explained in the Relative SNR section.

3.2.2 OFDM Modulations

Four OFDM modulation formats were chosen for this experiment: OFDM-BPSK, OFDM-QAM4, OFDM-QAM16, and OFDM-QAM32 which all shared the same GRC template. The only difference was in how the frames were loaded. The GRC template is shown in figure 3.4.

For this setup the frames were generated in MATLAB (appendix A.1.2) and loaded into GRC through *frame.txt* file. There are 3 different frames that were generated: Preamble, Training, and Data Frames. All 3 of these frames were based on the 802.11a standard. Preamble frames are used at the receiver for frame synchronization and timing offset

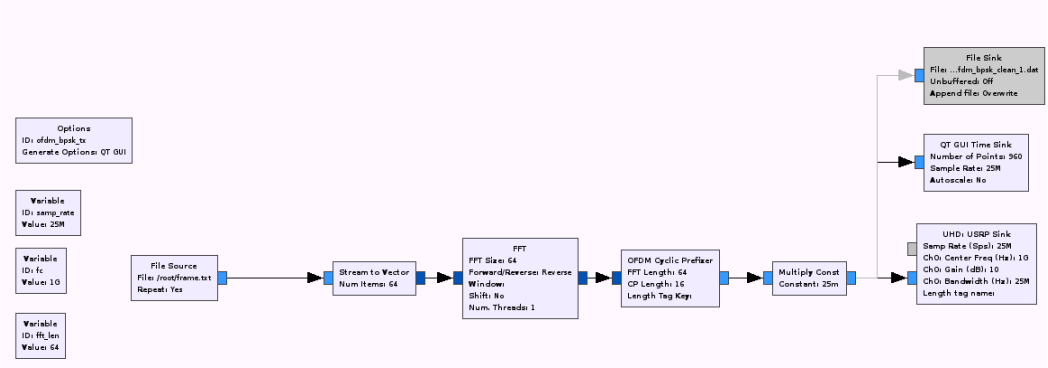


Figure 3.4: OFDM BPSK transmitter code in GNU Radio Companion.

recovery. Training frames are used to update the FDE weights, so that the following data frames, which contains the message, can be equalized.

In a classification problem, repetitive sequences like preambles and training frames can be used to an advantage by the classifier. While that would improve the accuracy, it wouldn't be realistic. To limit the repetitions it was decided to follow up preamble and training frames with 10 data frames. The frame sequence is shown in figure 3.5

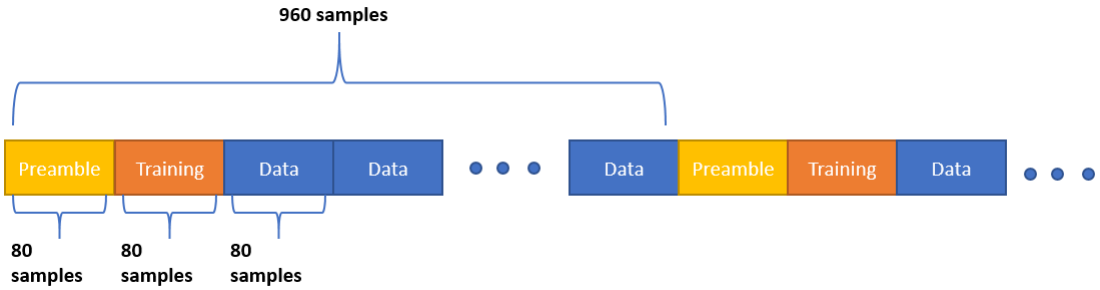


Figure 3.5: OFDM frame sequence.

The first block that the stream sees is the *IFFT* block, which accepts N=64 complex symbols at a time, and outputs a time domain signal. This signal is then passed through a

cyclic prefix block which inserts last 16 samples into the front of time domain frame. This signal is then multiplied by a constant such that highest possible value going into *UHD: USRP Sink* block is less that 0.707. The constant factor used is different for the 4 different OFDM modulations. The multiplicative constant value is listed in table 3.3.

Modulations	scaling constant
OFDM BPSK	1/40
OFDM QAM4	1/40
OFDM QAM16	1/80
OFDM QAM32	1/110

Table 3.3: Multiplicative constant value for the OFDM modulations.

The bandwidth is set to 25 MHz. In 802.11 standard the bandwidth is 20 MHz, but setting that bandwidth produced a CIC rolloff warning, which was generated by the SDR. The clock rate is fixed at 100 MHz, so having a sample rate of 25 MHz is preferable over 20 MHz. This is because $100MHz/25Mhz = 4$ (even decimation number) vs $100MHz/20MHz = 5$ (odd decimation number). Few captures at 20 MHz were inspected to check for any extra impairments and none were found. It is possible that the warning was bogus, yet some impairment may show up randomly. To keep things reproducible, the bandwidth was set to 25 MHz. This higher bandwidth signal is still a wide band signal so there is nothing to lose by choosing this bandwidth. Similar to linear modulations, the transmitter gain values were varied between 0, 5 and 10 dB, which will be explained in the next section.

3.2.3 Relative SNR

To make the problem of classification more realistic, it was desired to have multiple SNR data captures. There are two ways to go about doing this. The first way of changing SNR would have been to increase the noise power. This would have been possible to do if we were working in the ORBIT Grid, but reserving time slots for those are hard to do. Data capturing in general takes a lot of time, so it would have been troublesome to use this method with limited time slot.

The other option was to keep the noise power fixed, and increase the signal power. It was determined by looking at test captures that the noise power level was not changing significantly from capture to capture. With the assumption that noise power is constant through all the capture time, SNR change can be implemented by simply increasing the TX gain value. Thus, the TX gain is varied between 0 dB, 5 dB and 10 dB to get 3 different SNR data captures. TX gain of 15dB was also tried, but it saturated the receiver. So, this is the method used to get relative SNR captures of 0, 5 and 10 dB. The actual SNR of the captures is not known, but that isn't as important as knowing how well the classifier performs with different relative noise power levels.

3.3 Validation

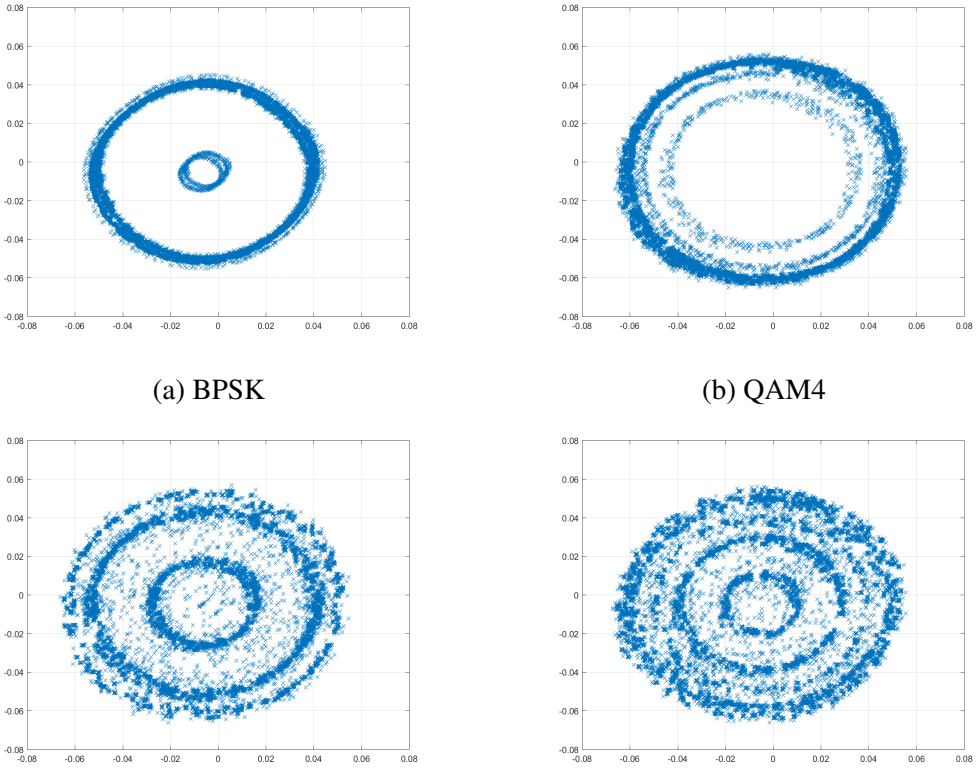
It is very important to verify that the transmitted signal is captured properly. Otherwise the algorithms might be trained to classify bogus captures. In chapter 2 we learned about various impairments that are introduced in an over the air transmitted signal. Thus, it is essential to check that original message is recoverable.

One of the example function provided in the UHD was used to capture the data. The name of that function is *rx_samples_to_file.cpp* [21]. This function captures the raw baseband samples after the signal has been down converted from RF to baseband. 1 million samples were captured for all modulations, at a sample rate of 2 Msps for linear modulations and 25 Msps for OFDM modulations. The bandwidth was set equal to sampling rate. To validate the data captures a MATLAB script read through them, then created and saved an image of the constellation diagram of the raw baseband signal. These constellations were visually inspected to ensure they were correct.

To fully validate the signal it must be synchronized. The next section presents the constellation diagram of the unrecovered signal, which will show the impairments that was described in chapter 2. In the following section the method used for synchronization will be discussed along with the results.

3.3.1 Constellation Diagrams

The constellation diagrams for all the unsynchronized linear modulations: BPSK, QAM4, QAM16 and QAM32 are shown in figure 3.6.



(a) BPSK

(b) QAM4

(c) QAM16

(d) QAM32

Figure 3.6: Unsynchronized constellation diagrams for all the linear modulations.

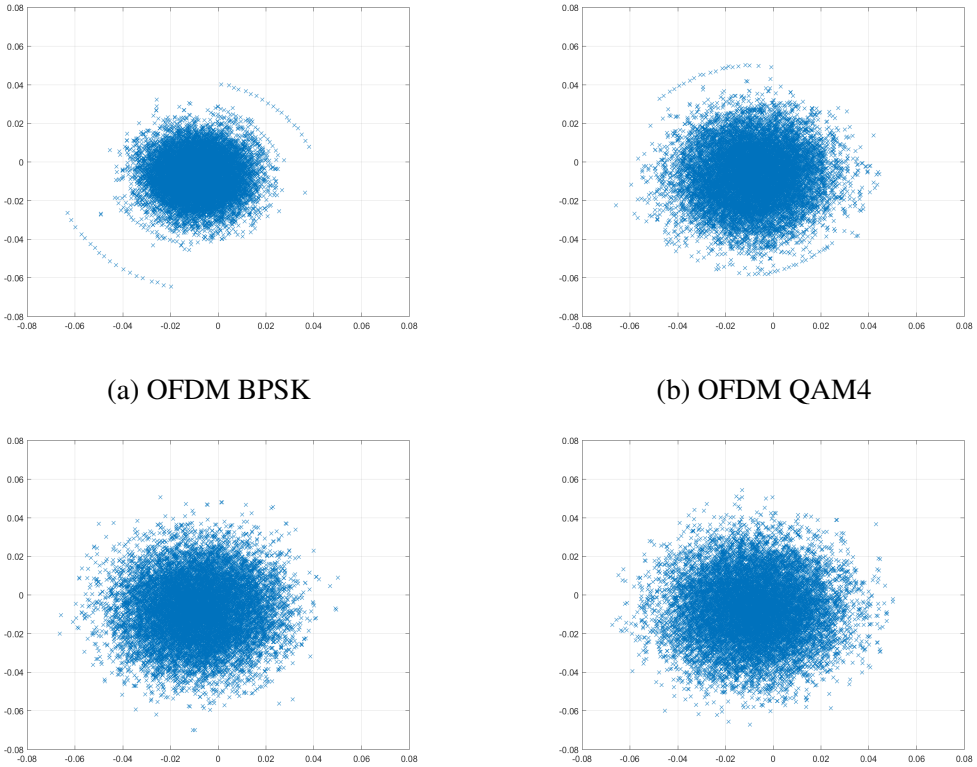
The TX gain was set to 0dB, so the scale of the constellation is similar across the modulations. The constellation diagram is only made up of 20K samples to keep the memory requirement short. There are couple of impairments visible in the constellation diagrams. Primarily, you can see the effects of carrier frequency offset. The transmitted

constellation has rotated by more than 360 degrees. Another impairment we can see is the DC offset, since the constellation is not rotating around the origin.

Timing offset impairment is not easily visible, but it can be seen in BPSK and QAM4 constellations. The valid 'ring' is the outer ring, where the actual symbols have rotated around the origin. The inner rings that are visible are due to symbol transitioning between 1 and -1. If there was no timing offset, then there wouldn't be an inner ring, as the transitions are not be captured under optimum sampling times.

Notice the number of rings that are seen in each modulations. For QAM16 3 rings are expected to be seen, and for QAM32 5 rings are expected. The rings are barely visible in QAM32's constellation diagram because the 4^{th} and the 5^{th} ring are very close. It is safe to say that the data was captured properly. Only thing left to do is remove the impairments and synchronize. The results of that will be presented in the next section. OFDM modulations are inspected next in figure 3.7.

At a glance the constellation diagrams for the OFDM modulations are not as informative to look at. The presence of DC offset is visible as the constellation are not centered around origin. The training frame's values are visible in OFDM BPSK and OFDM QAM4; they are 2 dis-joined outer semi circles. The training symbols without any frequency offset would have been fixed to just one spot. The rotation seen here is evidence of frequency offset impairment. The reason training frames are not noticeable in OFDM QAM16 and OFDM



(a) OFDM BPSK

(b) OFDM QAM4

(c) OFDM QAM16

(d) OFDM QAM32

Figure 3.7: Unsynchronized constellation diagrams for all the OFDM modulations.

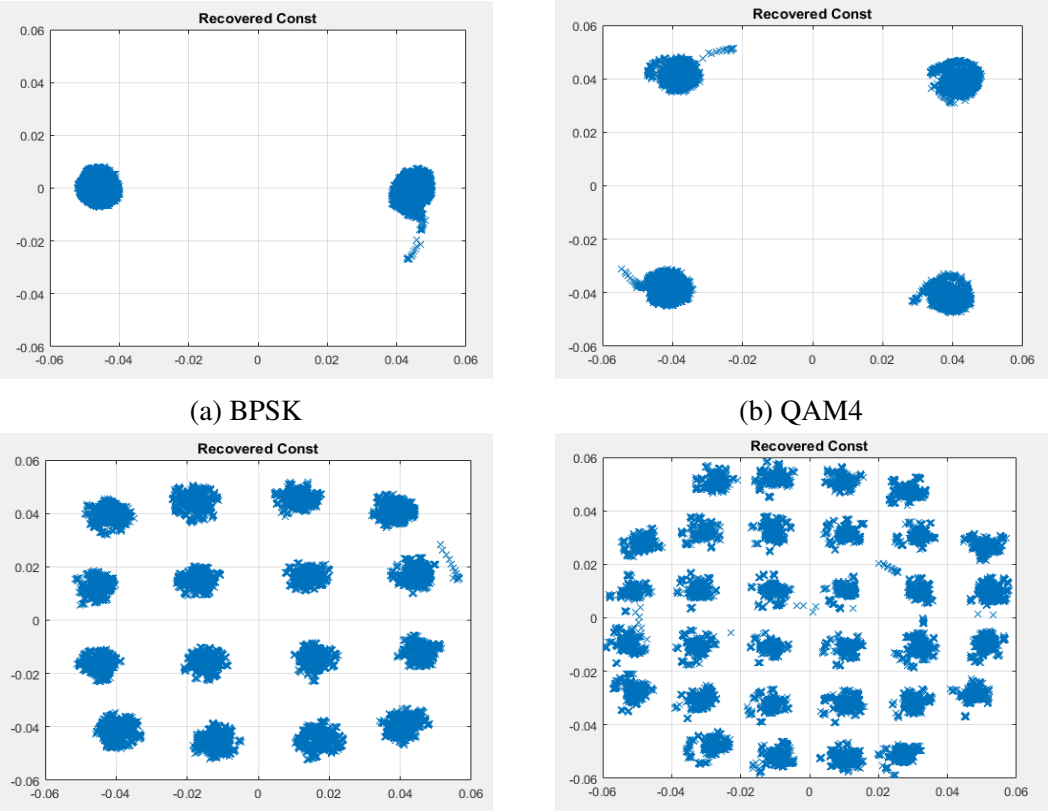
QAM32 are because they are being dwarfed by the data frame symbols.

It is reasonable to assume that a classifier would have more difficulty in classifying OFDM modulations compared to linear modulations. In the next section, the modulations are synchronized and their results are shown.

3.3.2 Synchronization

To fully verify that the captured signal were valid MATLAB receiver code for each modulations were designed to synchronize the signal. Removing DC offset is as easy as subtracting the mean of in-phase and quadrature phase of the signal. For linear modulations, frequency offset can be removed if the exact offset between TX and RX LO is known. Typically a phase locked loop (PLL) like costas loop is used, but they are harder to construct for higher order QAMs. The other alternative is to manually apply a correction factor till the constellation looks synchronized. The code for the receivers, which uses manual correction, can be found in the appendix A.1.3. Figure 3.8 shows all the recovered constellations for the linear modulations.

OFDM modulations are more complicated, and synchronizing them requires more effort. The first thing to do is to find the frame boundaries, and this is where the preamble comes in handy. A delay line auto correlator can be used to get good estimate on preamble frame boundaries. A simpler method is to do a cross correlation between the transmitted signal and the received signal. The transmitted signal was saved from within GRC using the *File Sink* block, as can be seen in top right of figure 3.4. This cross correlation will show peaks at the frame boundaries. After frame boundaries are located the cyclic prefix can be removed. Carrier frequency offset can be measured from the pilot symbols. Then, it is a simple matter of applying a correction factor to remove the frequency offset. Before the



(a) BPSK

(b) QAM4

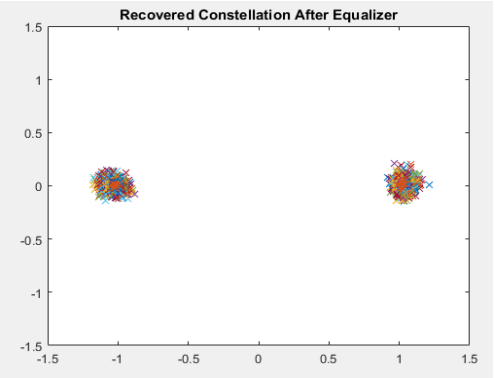
(c) QAM16

(d) QAM32

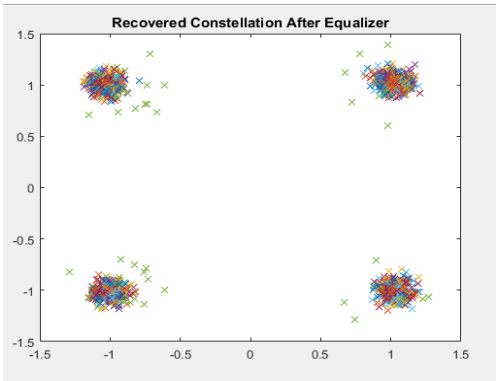
Figure 3.8: Synchronized constellation diagrams for all the linear modulations.

constellation can be recovered, the training frame values are extracted and used to update FDE weights. The FDE is then applied on the data symbols to remove multipath fading. The code can be found in appendix A.1.4. Figure 3.9 shows the synchronized constellation diagram for all the OFDM modulations.

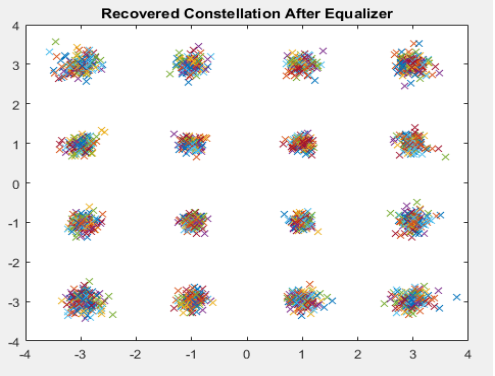
The constellation diagram for OFDM QAM32 does not look tight. This is because the FDE coefficients have become stale. The FDE is updated once for 10 data frames, so the coefficients becomes stale towards the 9th or 10th data frame. This results in the edge



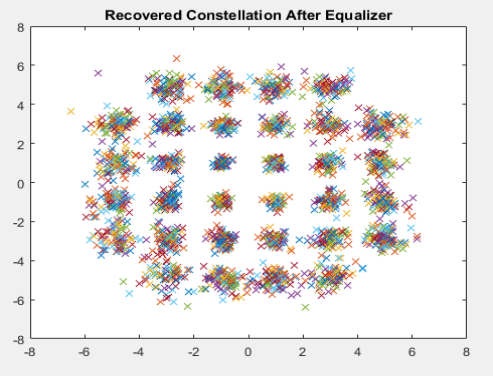
(a) OFDM BPSK



(b) OFDM QAM4



(c) OFDM QAM16



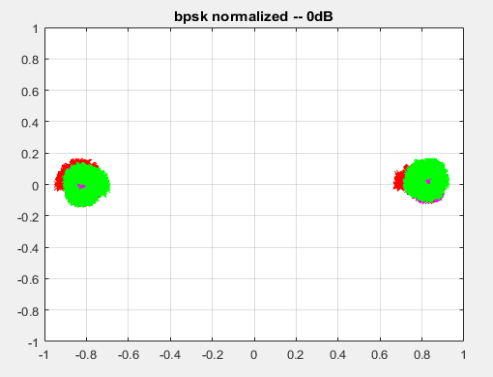
(d) OFDM QAM32

Figure 3.9: Synchronized constellation diagrams for all the OFDM modulations.

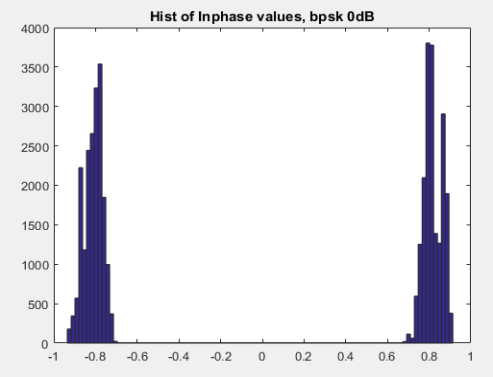
constellation points becoming noisier. The goal of the project was not to build a state of the art receiver, so this issue is not of concern. However, it is visible from these results, that the captured data was the result of a valid transmission.

Finally, the relative SNR data captures were validated. To show that the SNR has indeed improved for TX gain of 5dB and 10dB, a sync was performed on those BPSK captures. Figure 3.10 illustrates the improvement in the signal due to higher SNR data captures. Notice that the constellation points gets more concentrated as the SNR goes up.

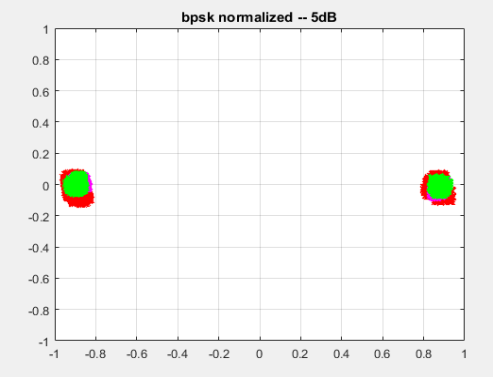
This can also be seen from the histogram plots, which shows that the sample variances are decreasing as the SNR increases. This is expected since noise variance, which is determined by the noise power, is also decreasing as the SNR increases.



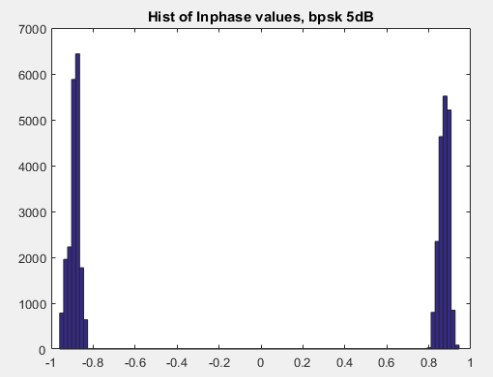
(a) Synchronized Constellation @ 0dB TX gain



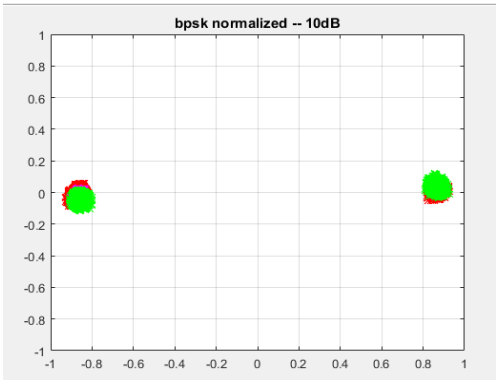
(b) Hist. of I samples @ 0dB TX gain



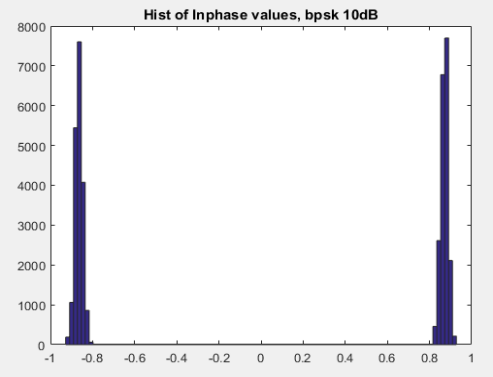
(c) Synchronized constellation @ 5dB TX gain



(d) Hist. of I samples @ 5dB TX gain



(e) Synchronized constellation @ 10dB TX gain



(f) Hist. of I samples @ 10dB TX gain

Figure 3.10: Synchronized constellation becomes tighter when TX gain is increased.

4

Algorithms for Modulation Recognition

In this chapter a brief background information is given to explain the two different classifier that will be tested for modulation classification. Key points about the SVMs will be discussed in the next section, and NNet will be introduced in the following section.

4.1 Support Vector Machines

The general formula for SVM classification is shown in equation 4.1. The weights w^T and bias b parameters are adjusted such that $y(x) > 0$ for x in class 0, and y(x) < 0 for x in class 1. Thus, the decision boundary is at $y(x) = 0$. This is only possible to do if the data is linearly separable in the feature space. An example of support vectors and decision boundaries is shown in figure 4.1. On the cases where data is not linearly separable, a slack variable $\xi = |t_n - y(x)|$ is introduced. The discussion of support vector machines is beyond the scope of this thesis. The reader is encouraged to read up on how SVM algorithm locates

the support vectors at [1].

$$y(x) = w^T \phi(x) + b \tag{4.1}$$

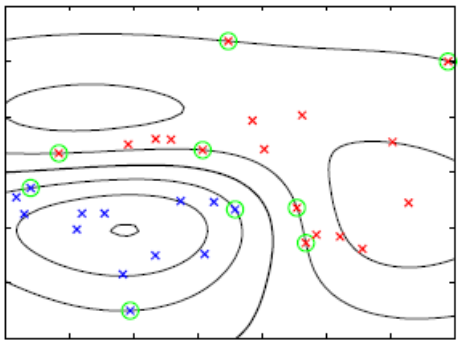


Figure 4.1: Two classes shown in red and blue. Contours of constant y(x) are shown through the solid line. The decision boundary is between the support vectors which are circled in green. Figure taken from page 331 of [1].

SVMs are a type of sparse kernel machines. A kernel $\phi(x)$ is a feature space transformation of the input x, and in a sparse kernel machine, the new input's class is predicted based on only a subset of training data points, which are called support vectors. Which means, for most of the training data points w^T is set to zero.

It is possible for SVM to have high generalization with small training set. They are good at generalization, because they are optimized based on the concept of margins. A margin is the distance between the decision boundary and any data point, see figure 4.2. SVM's algorithm tries to find the maximum separation between two classes by mapping the input into a high dimensional feature space and constructing optimum separating hyperplane, thus providing the lowest generalization error possible.

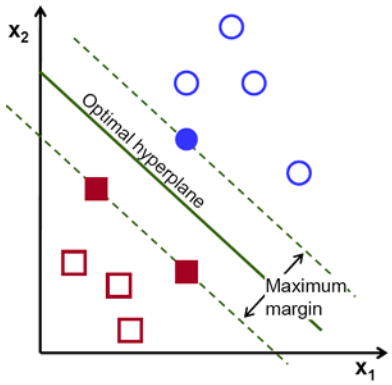


Figure 4.2: Maximum margin between two classes and the optimal hyperplane in between. Figure taken from [17].

For a classifier to work well it needs to be trained with good features. This classifier generally works well for modulation classification with Higher Order Cumulants (HOC) as its features. Cumulants are computed from the probability distribution of the signal. For example, the first and second order cumulants are mean and variance of the signal respectively. A captured signal typically has gaussian noise added to it, which is zero mean. Thus, the distribution of captured signal is not much different from the distribution of the modulated signal in it. This makes HOC less sensitive to noise [4], which means the classifier is able to generalize across various SNR level using cumulants. One drawback of using cumulants is that a lot of data is needed to accurately compute them.

4.2 Neural Networks

Neural Networks were initially designed around the concept of how our brain's neuron works. The neuron is the basic working unit of the brain, a specialized cell designed to

transmit information to other nerve cells, muscle, or gland cells [16]. Neuron transmits information only if is activated which happens when the input is above a threshold. Neural networks are constructed similarly; bunch of layers of nodes which are activated by the input that has been adjusted using adaptable weights.

Neural Networks are popular because they are able to extract relevant features out of raw input data and perform classification or regression work. NNets are considered as a black box algorithm because the extracted features are hidden and not known to the experimenter. The beauty of using NNet is in its ability to extract information that the experimenter may not know about. They may even perform better than traditional classifier which are trained with optimal features. The drawback of NNets is that the model may require a lot of weights for certain type of tasks. NNets may be known as black box, but it doesn't mean that they will work with just any input. If the input signal is not good for the architecture then the network will not learn anything. There are two architectures of NNet that are worth talking about and they are presented in the next two sections.

4.2.1 Multi Layer Perceptrons

Multi Layer Perceptrons (MLP) consists of multiple layers of neurons. The output of each layer is fed into the next layer. The neuron itself contains a non linear activation function, which may activate based on the input value. Figure 4.3 shows a 3 layer MLP.

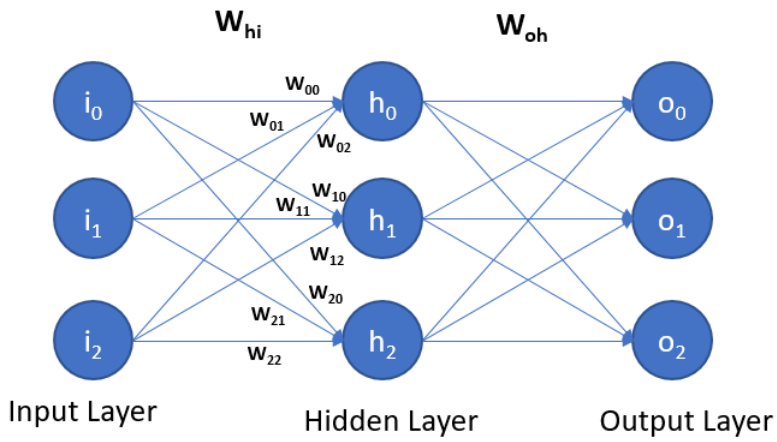


Figure 4.3: 3 layer MLP.

In figure 4.3 there are total of 6 neurons, 3 at the hidden layer and another 3 at the output layer. 3 input samples are being fed into 3 neurons at the hidden layer. Each neuron receives three weighted inputs from the previous layer, which are summed and fed to an activation function of choice. Typical activation functions include sigmoid, rectifier linear unit (ReLU), and tanh.

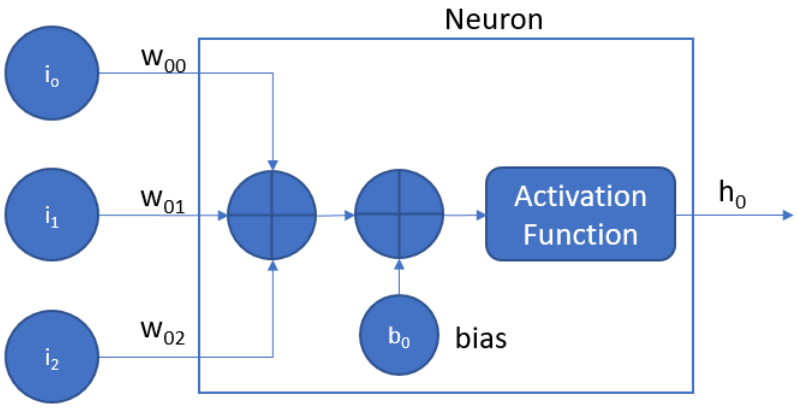


Figure 4.4: Inside neuron h_0 of figure 4.3.

The key here is that the weights W_{hi} and W_{oh} are adaptive and learned by the neural network. The way to do that is by using back propagation. First a loss function which is differentiable w.r.t all the weights is constructed based on output error. Then, an optimizer of choice is used to minimize the loss function. This is usually done by some kind of iterative gradient descent (GD) algorithm. The weights are updated based on the gradient. The gradient always points towards the positive slope, therefore to minimize the loss, the weights are updated based on negative of gradient. Typical weight update equation can be seen in Eq 4.2.

$$w(\tau + 1) = w(\tau) - \eta \Delta E(w(\tau)) \tag{4.2}$$

In equation 4.2, w are the weights in the neural networks, and $\Delta E(w)$ is the gradient of neural network with respect to the weights. Neural Networks are updated iteratively, and τ represents the iteration time steps. Learning rate η adjusts the rate at which the weights adapt and reach the gradient's minima. With a high η value it is possible to overshoot and miss the minima. Making η low would guarantee reaching the minima, however that might take forever to converge. A better approach is to start with high η value so that it converges faster, and then gradually lower the value so that it doesn't overshoot the minima.

A complete discussion of MLP is beyond the scope of this thesis. The reader is encourage to learn more about them in the following book about deep learning [6].

4.2.2 Convolutional Neural Networks

Another popular architecture of neural network are convolutional neural networks (CNN). These were built for image classification. Image classifications are harder to do with MLP because the image has to be flattened into a layer. Most of the spatial information is lost due to flattenning, which causes MLP to perform poorly with image like input. In an image the fact the two pixels are next to each other means something; that is spatial information that would have been lost when flattened. The CNN allows for more robust feature extraction for image like input samples. The structure of a CNN net is shown in figure 4.5.

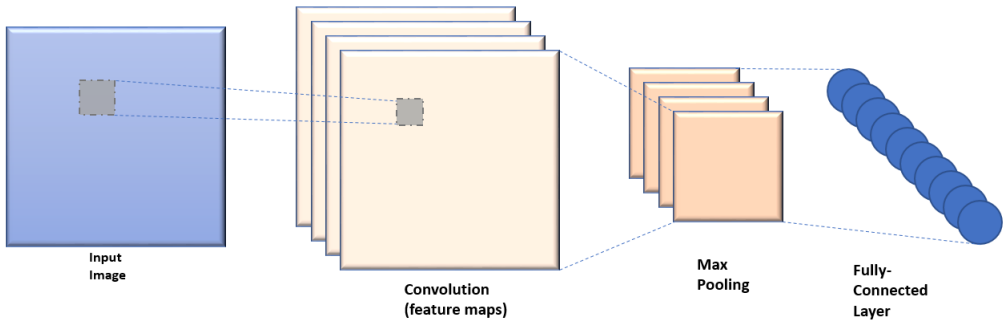


Figure 4.5: Simple convolutional neural network. Figure taken from [18].

Convolutional network works by applying 2d convolutions on small patches of the input image (shown in gray). The patches are then shifted around the original input image based on the stride parameter. With stride value of 1 and proper padding around the edges, the second layer's feature map can have the same size as the input image. In the example shown above, 4 different sets of weights are applied on the input image, which results in 4

convolution feature maps. The CNN can adapt the weights such that they extract different features from the input image. For example, one of the feature map may extract edges, and another one may extract circular shapes. This type of multi-level feature extraction allows CNN to classify images with high accuracy. For classification problems, the network is usually terminated with a fully connected layer, so it helps to reduce the dimensionality of the features maps as the network goes deep.

Activation functions are usually applied after the 2d convolution functions. After that, if needed, the feature map's dimensionality are reduced. There are a few ways to reduce the dimensions of the feature maps. The most obvious way to do so is by increasing stride value. By setting the stride to 2, one can roughly lower the dimension by half. Max Pooling is another way to lower the dimension. Max pooling takes a patch of the input and returns the max value within that patch. A 2x2 Max pooling with stride 2 will reduced the dimension of the feature map by half, see figure 4.6.

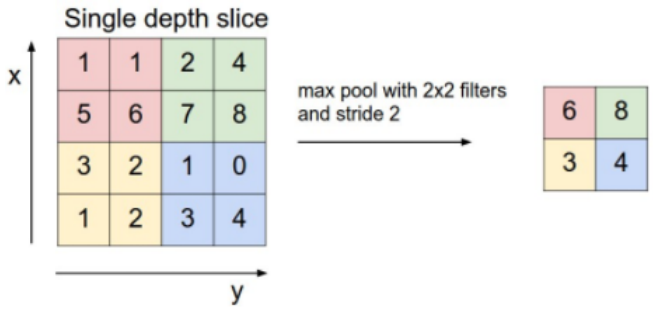


Figure 4.6: Max Pooling example. Figure taken from [28]

If the NNet is being build for classification, then eventually the feature map is flattened

and applied to an MLP layer with a softmax activation function. A complete discussion of CNN is also beyond the scope of this thesis. The reader is encourage to learn more about them in the following book about deep learning [6].

5

Modulation Classification

In this chapter the results of the two different classifiers, SVM and NNets, are presented. Both classifiers had to be tuned to get good classification accuracy, which is also discussed in this chapter. SVM is considered as the baseline classifier, and its performance is evaluated first. NNet are then tested to see if they can perform better.

5.1 SVM

Pattern recognition toolbox (PRT) was used to build the SVM classifier. PRT is built for use in MATLAB [19]. It contains many classifiers, including multi-class SVM, and provides functions that helps with data visualization.

Cumulants up to 8^{th} order were used as features for the SVM classifier. They were extracted from I and Q samples separately, so each training example contained a total of 16 features (appendix A.1.6). To accurately estimate higher order cumulants large number of

data samples are required. Each captured signal contained only 1 million samples, so the estimates could only be made from those.

The computation time for calculating the estimates were rather high, and since there were 1000 captures for each modulation and noise power level it was decided to have the cumulants estimated from a reduced sample set. Two different estimates were made, one from 200K and another from 50K I & Q samples. The performance may differ based on the accuracy of the cumulant estimates.

Some pre processing steps were taken to improve the input feature space. Firstly, the received samples were severely attenuated; the maximum value for most of the high TX gain signal was somewhere around 0.1. Cumulant estimates based on such small sample values were too little for the classifier to work with. The classifier performed better with cumulant estimates from normalized captured signal.

The entire feature set's mean was subtracted to bring the mean to zero, and the variance was also scaled to one. This was mostly done because of the next step, which was running the features through principal component analysis (PCA). PCA rearranges the correlated feature space into non correlated feature space. It is typically done so that SVM can find the optimum hyperplane with ease. From the testing it was seen that PCA improved training time and classification results.

There are two hyper parameters for SVM: cost and gamma (γ). Cost parameter determines the sparsity of the model. A higher cost value will produce more non-zero weights on the data points, whereas a low cost value will tend to produce more zero weights. The optimal value for cost depends on the separation of classes in the feature space. If the classes are heavily mixed then a higher cost value will be needed to get good classification. The risk of having a high cost value is that the classifier might overfit to training data and perform poorly against test set.

The γ parameter is associated with the type of kernel used in SVM. For this classifier, the gaussian radial basis function (RBF) kernel is used, since it performed better compared to others. This is a non linear kernel which transforms the input into a different feature space. Equation 5.1 shows the transformation from which it can be seen that the value of γ determines the transformation space. Optimal γ value should produce a more separable feature space.

$$k(x_1, x_2) = e^{(-(\sum(x_1 - x_2)^2)(\gamma^2))} \tag{5.1}$$

The optimal values for these parameters are picked by exhaustive grid search. See figure 5.1 for an example of grid search over linear modulations. Here you see that the accuracy for the test case improves with cost value of 10 and gamma value of 0.01. It was interesting to note that the optimal values of cost and gammas were different when tested for linear and OFDM modulations. The final cost and gamma values which worked well for

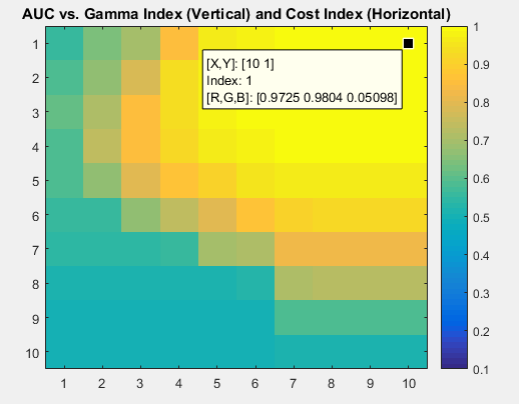


Figure 5.1: Grid search example for finding optimal Cost and Gamma values. This SVM model is classifying between QAM16 and QAM32 modulations. A bigger grid search can provide a more optimal pairs of Cost and Gamma.

both training and testing data sets are listed in table 5.1.

Modulation	Linear	OFDM
Cost (c)	30	10
Gamma (γ)	0.01	0.01

Table 5.1: SVM Hyper parameters.

This difference in parameter value for linear and OFDM modulations meant that a single SVM classifier trying to simultaneously classify all 8 modulations would perform poorly. A hierarchical approach is used instead as shown in figure 5.2. Three different SVMs will be working together: SVM # 1 will classify between linear and OFDM modulations; SVM # 2 will classify the linear modulations, and SVM # 3 will classify the OFDM modulations.

The fist SVM classifier between linear and OFDM modulations was very easy to train. It was able to classify between the two types of modulation with 100% accuracy.

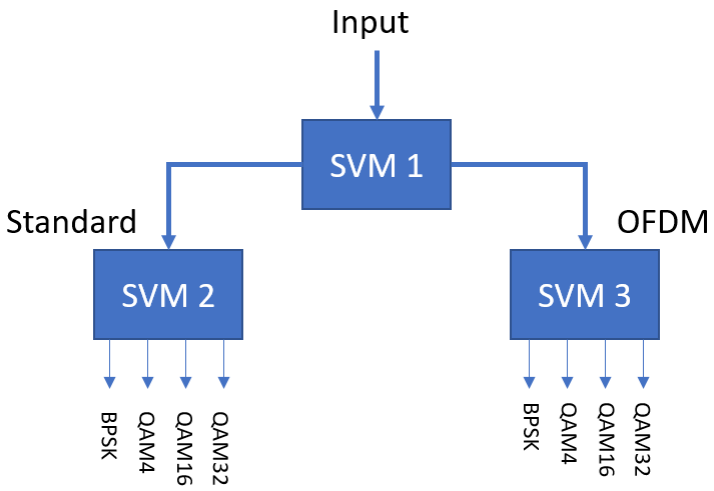


Figure 5.2: Hierarchical Model for SVM classification for all modulations.

Kfold validation test with k = 10 was run to quickly determine the performance of SVM # 2 and # 3 . The reader is reminded that there were 3 different signal power levels at 0dB, 5dB, and 10 dB, as well as 2 different cumulants estimation based on 50K and 200K samples. The Kfold validation results for linear and OFDM modulations are shown in figure 5.4, and 5.5 respectively.

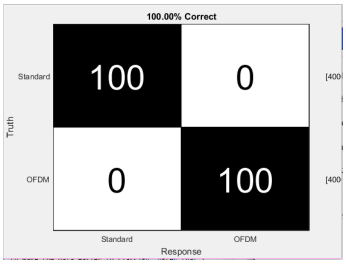
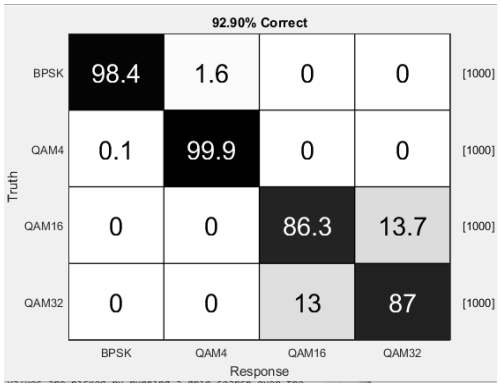
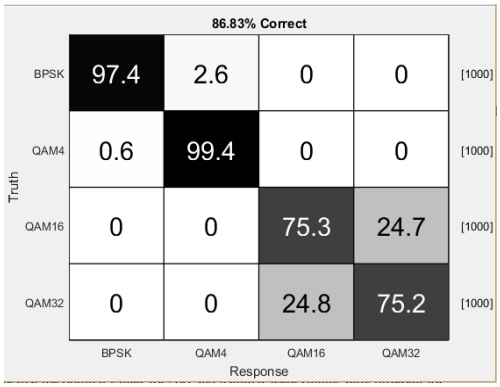


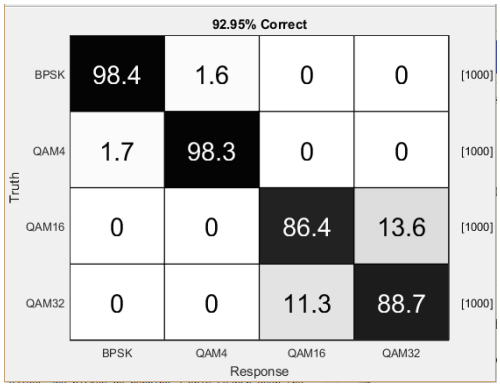
Figure 5.3: Kfold validation results for SVM # 1. This SVM classified between linear and OFDM modulations.



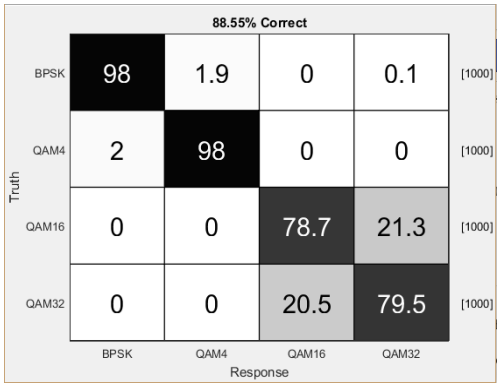
(a) 200k sample estimate @ 10dB TX gain



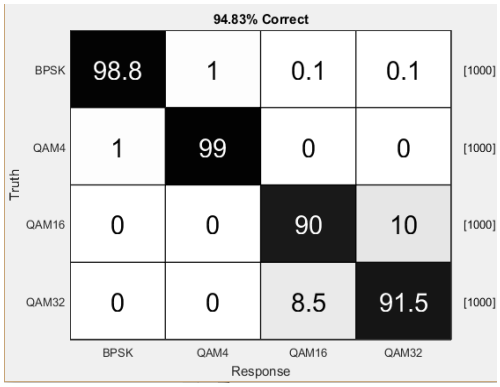
(b) 50k sample estimate @ 10dB TX gain



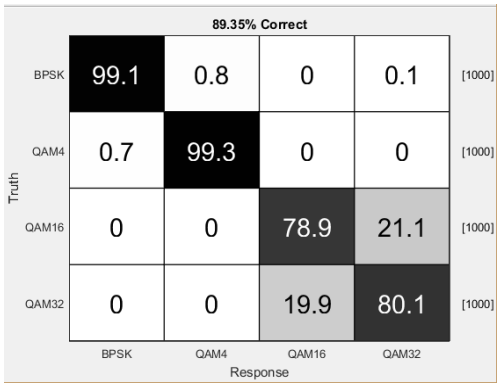
(c) 200k sample estimate @ 5dB TX gain



(d) 50k sample estimate @ 5dB TX gain

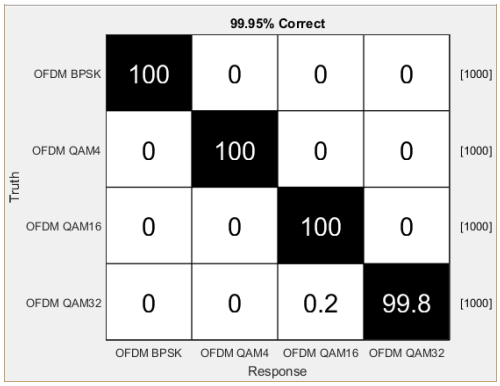


(e) 200k sample estimate @ 0dB TX gain

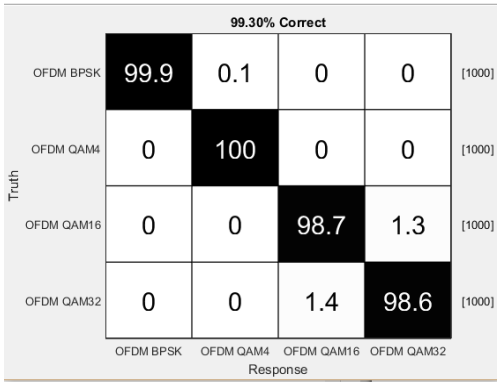


(f) 50k sample estimate @ 0dB TX gain

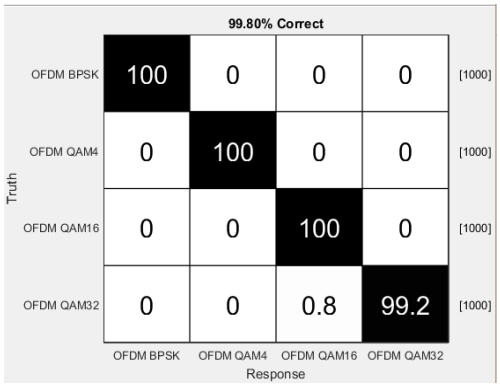
Figure 5.4: Kfold validation results for linear modulations (SVM # 2).



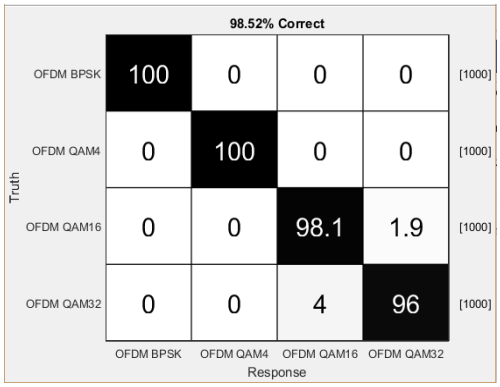
(a) 200k sample estimate @ 10dB TX gain



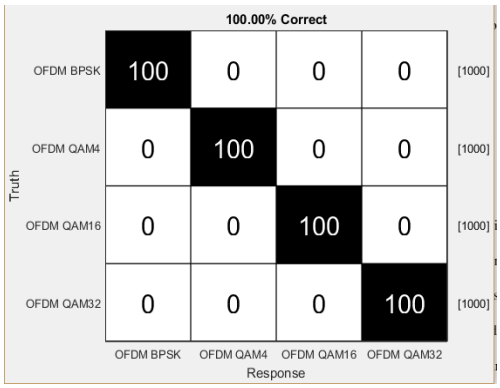
(b) 50k sample estimate @ 10dB TX gain



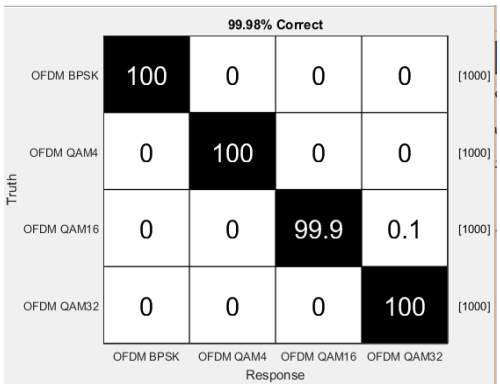
(c) 200k sample estimate @ 5dB TX gain



(d) 50k sample estimate @ 5dB TX gain



(e) 200k sample estimate @ 0dB TX gain



(f) 50k sample estimate @ 0dB TX gain

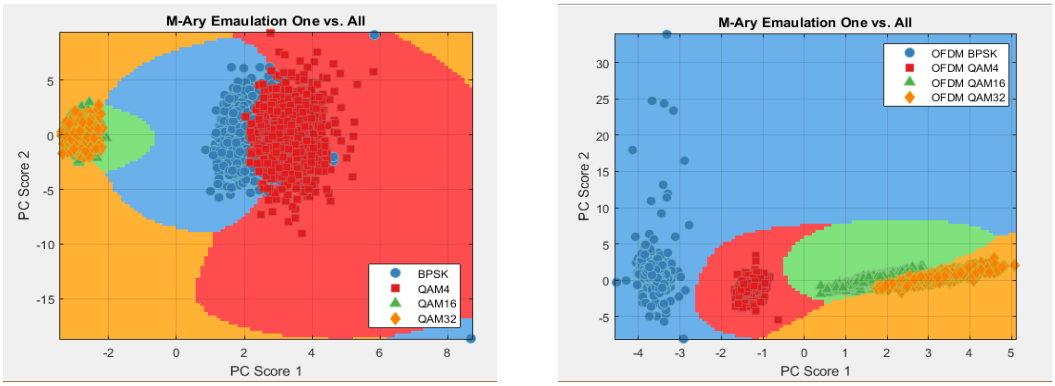
Figure 5.5: Kfold validation results for OFDM modulations (SVM # 3).

There are a few interesting things to note for the linear modulation classifier. Higher SNR modulations and more accurately estimated cumulants are easier to classify. There is a 4 to 5% improvement in classification performance with cumulant estimates of 200K over 50K samples. Across the board, the linear modulation classifier has trouble with higher order QAMs. One thing to note here is that it is possible to improve the classification performance for any of the case by tuning the hyper parameters. Since SVM is being tested for all cases simultaneously the hyper parameters should remain fixed.

The OFDM modulation classifier performed really well across the board. Surprisingly, OFDM modulations are easier to classify with cumulants. Before we move on to test the classifier's ability to generalize over multiple SNR data, it is worth looking at the feature space to understand the classifier's performance.

Figure 5.6 shows the decision boundaries for linear and OFDM modulations when classifying with just 2 PCA-feature. Notice that OFDM classes are cleanly separable with just 2 features. Linear modulation classes, however, have lots of mis-classfication regions between QAM16 and QAM32. This particular classifier, with just 2 features, got an accuracy of 72%. There are 16 features in total, so it is possible to get better decision boundaries in higher dimensional feature space, which is why the overall accuracy rate for linear with all features is around 93% as was seen in figure 5.4.

Hierarchical approach of classifying OFDM and linear modulation is showing decent



(a) Linear SVM Classifier

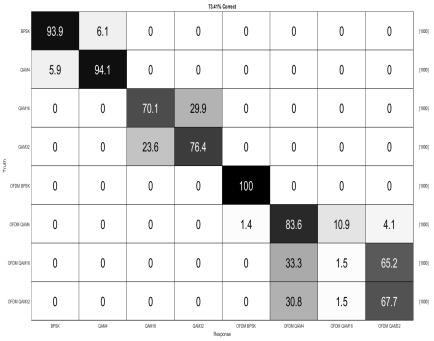
(b) OFDM SVM Classifer

Figure 5.6: Decision boundaries for the classes in linear and OFDM modulations. The classifier is taken from one of the ten models in the Kfold validation test.

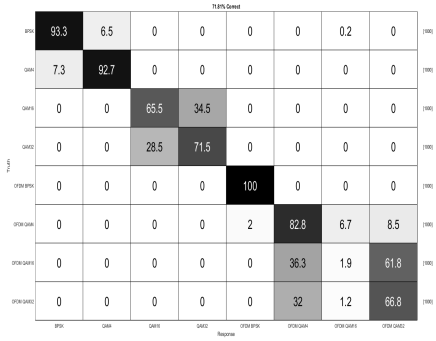
results. For completeness, a single SVM classifier is built and trained with all modulations. The cost parameters is set to 30 which is suitable for linear modulations. The Kfold results for this classifier is shown in figure 5.7.

It is clear to see from figure 5.7 that a single SVM classifier is not good at simultaneously classifying both linear and OFDM modulations. Since the cost value is biased towards linear modulations, it is able to classify them better. It is interesting to see that OFDM modulations are poorly classified here, since so far SVM was able to classify them with high accuracy.

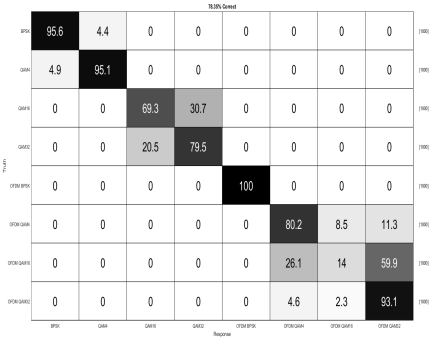
In the next test, we get to gauge the SVM classifier's ability to generalize across multiple SNR levels. In this test the classifiers are trained with data from all SNR levels. 200K and 50K cumulant estimates are run in separate tests. The results for linear modulations are shown in figure 5.8, and the results for OFDM modulations are shown in figure 5.9. The result with all modulations are presented in figure 5.10.



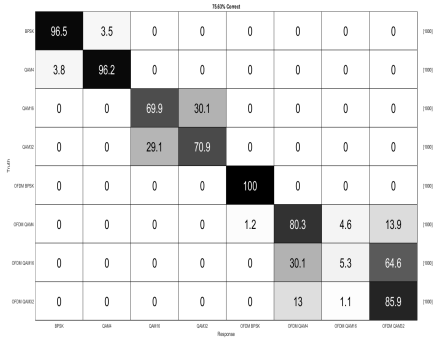
(a) 200k sample estimate @ 10dB TX gain



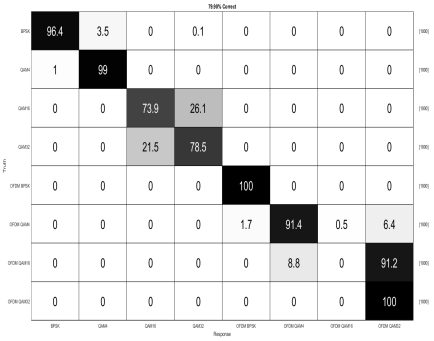
(b) 50k sample estimate @ 10dB TX gain



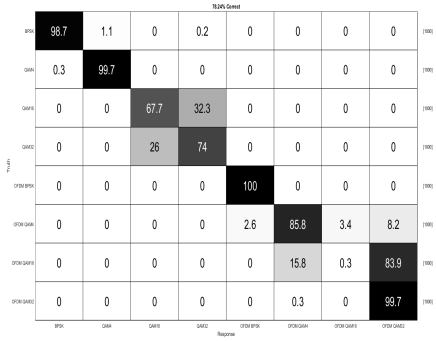
(c) 200k sample estimate @ 5dB TX gain



(d) 50k sample estimate @ 5dB TX gain



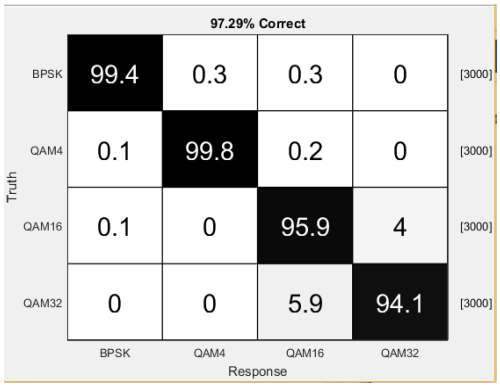
(e) 200k sample estimate @ 0dB TX gain



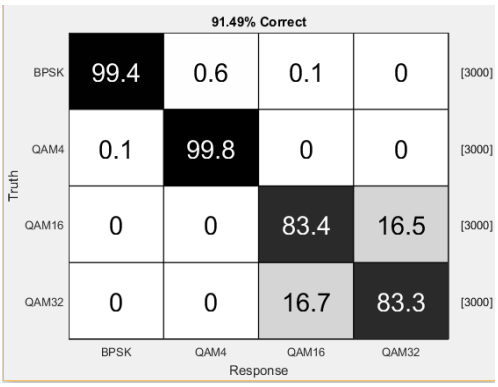
(f) 50k sample estimate @ 0dB TX gain

Figure 5.7: Kfold validation results for all modulations together.

The OFDM classifier performs well again, as expected from its earlier performance. The classifier for linear modulations performs fairly well as well. In this test the SVM is

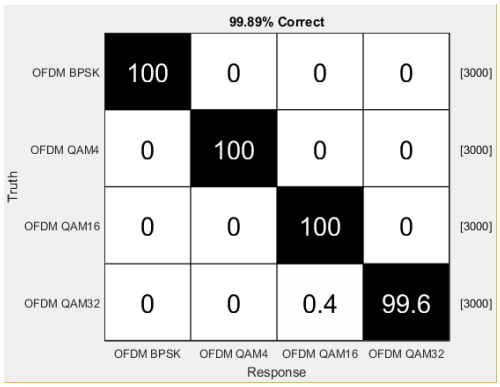


(a) 200k sample estimate results.

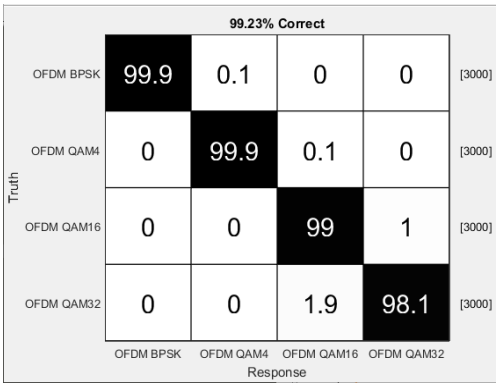


(b) 50k sample estimate results.

Figure 5.8: Kfold results for linear modulations when SVM is fed with data from all TX gain levels.



(a) 200k sample estimate results.

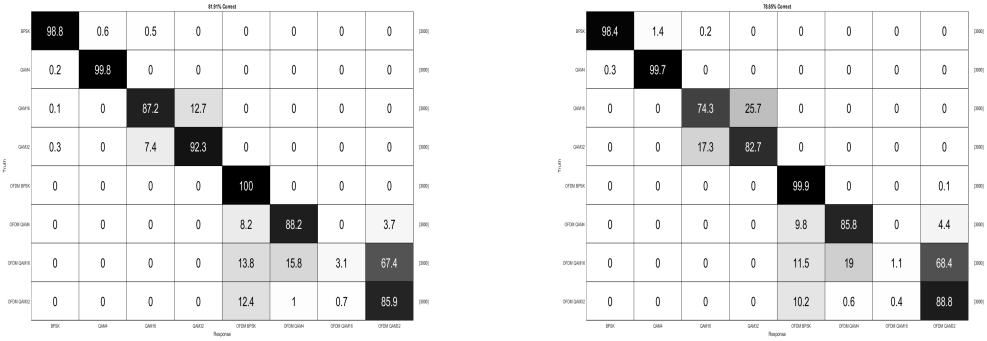


(b) 50k sample estimate results.

Figure 5.9: Kfold results for OFDM modulations when SVM fed with data from all TX gain levels.

given more observations to train with. There are 3000 observations per modulation to do Kfold testing with. Previously, there were only 1000 observations per modulation. Still there is 4% to 6% mis-classifications for higher order QAMs which is bad. Single SVM which is trained with all modulations and all SNR levels consistently performs poorly.

From looking at figures on OFDM constellations one would expect the classifiers to



(a) 200k sample estimate @ all of TX Gain (b) 50k sample estimate @ all of TX Gain

Figure 5.10: Kfold results when SVM is fed with data from all modulations and TX gain levels.

perform poorly against them. This result is unexpected, yet the issues seen with linear modulations shows the limits of SVM classifier. They are very much dependent on the choice of features the experimenter selects. NNet which is able to extract its own features is tested next.

5.2 NNet

There are many type of architectures of NNets, and more are being invented. For classification problem architectures like MLP and Convolutional NNets can be used. First, a MLP network is built and tested with the raw I & Q samples as the inputs. Then, a convolutional NNet is tried to see if that performs better.

5.2.1 MLP Results

For testing MLP architecture, a simple 3 layer network was built with ReLU activation functions in the neurons. See figure 5.11 for the MLP model. N_{input} raw I and Q samples are fed in at the input layer, which are then multiplied by weights W_{hi} and fed to the hidden layer. At each neurons in the hidden layer, the weighted inputs are summed and a bias value is added. This summation is passed through a ReLU activation function, and pushed on to the output layer after being multiplied by weights W_{oh}. At the output layer, a softmax activation function is used so that the output of all nodes sum to 1. This is a classic way to do classification using MLP.

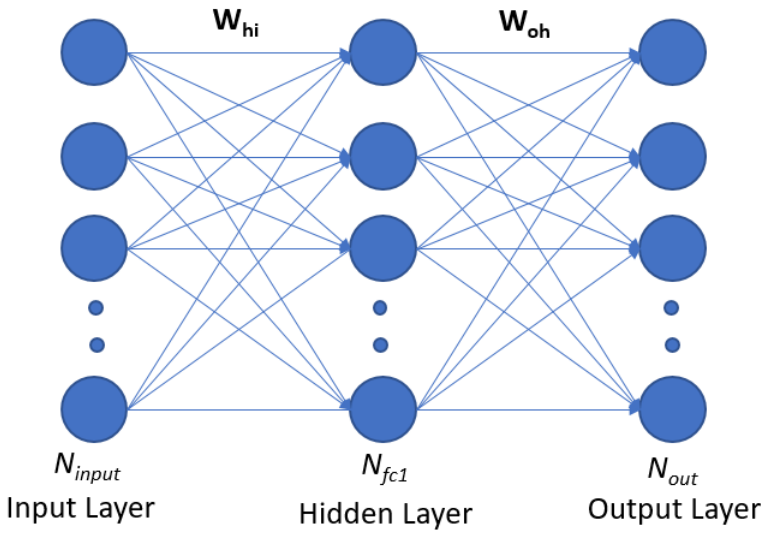


Figure 5.11: Multi Layer Perceptrion Model

Cross Entropy is the most commonly used loss function for classification problems. The equation for the loss function is shown in equation 5.2. Cross entropy is typically

evaluated over 2 distribution p and q, and it measures the similarity between those two probabilities. A softmax function is used across the output layer to turn the output into a probability distribution. Thus the loss value is equal to the cross entropy of the output of the MLP and a one hot encoding of the classes.

$$H(p,q) = -\sum_{n} Y\_(n) log(Y(n)) \tag{5.2}$$

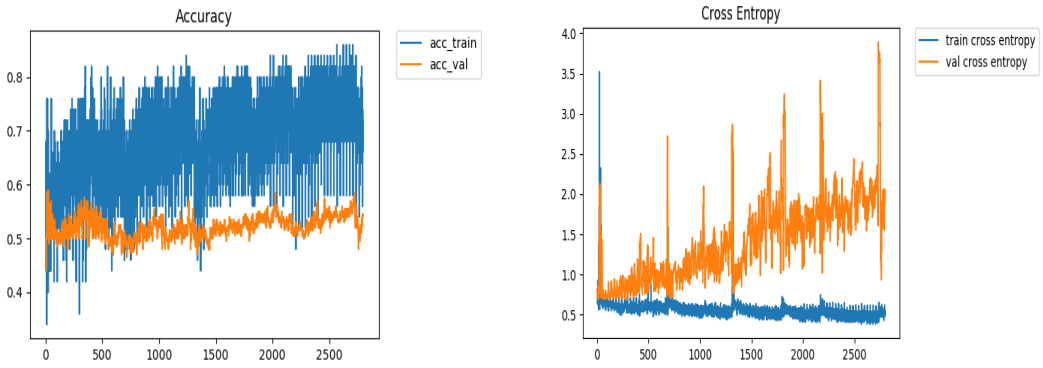
In equation 5.2, $Y\_$ is the ground truth, and Y is the output of the MLP. Two different optimizers were tried to minimize the loss: Gradient Descent (GD), and Adaptive Moment Estimation (Adam). GD optimizer was explained in Chapter 4. Adam optimizer algorithm updates the weights based on the mean and variance of the gradient of previous iterations [10]. This is different from GD in which the weight are updated based only on current iteration's gradient. The weight update equation for Adam is shown in equation 5.3.

$$\theta_t \leftarrow \theta_{t-1} - \alpha \times m_t/(\sqrt{v_t} + \epsilon) \tag{5.3}$$

In equation 5.3, α is the learning rate parameter, m_t and v_t are the estimates of the mean and variance of the gradients. Adam's update rule is similar to GD, and it has shown to perform well compared to other GD optimizers. The reader is encourage to read the Adam paper for additional information [10].

MLP was, however, unable to perform well with the raw I & Q samples. The parameters

of the networks, N_{input}, N_{fc1}, N_{out} were changed around, but none could give a good result. See figure 5.12 for the accuracy and loss function for an example MLP model.



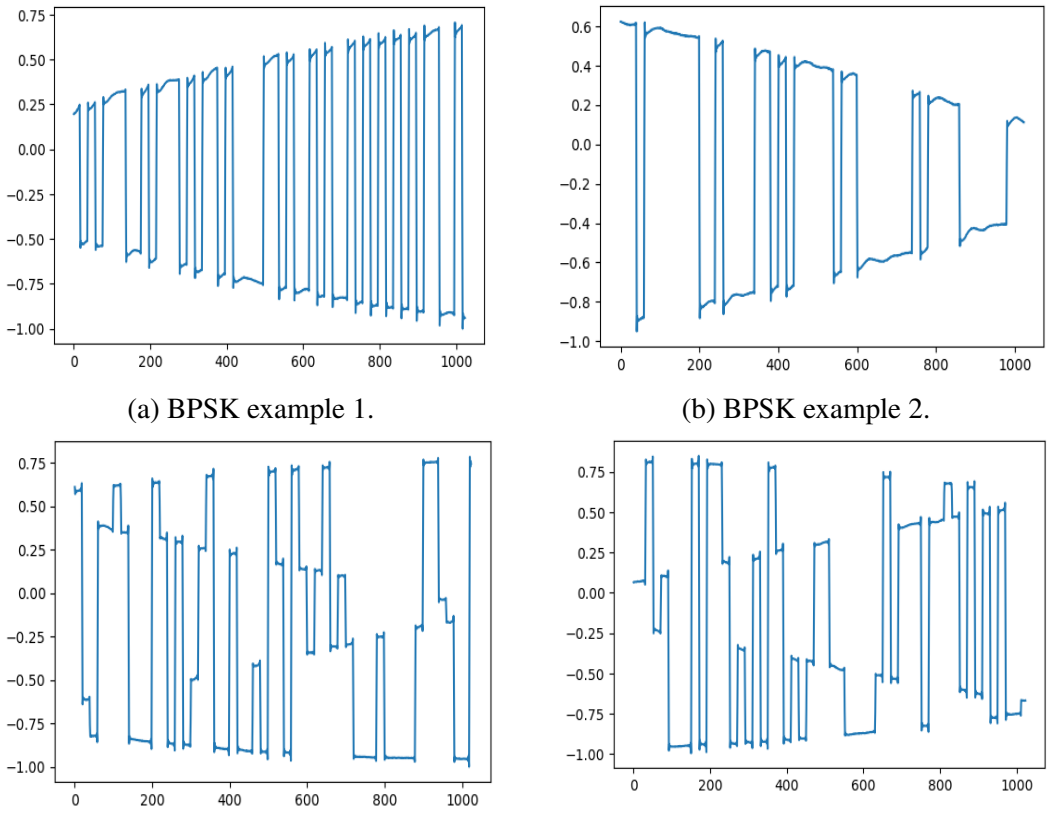
(a) Training and validation accuracy.

(b) Training and validation loss function.

Figure 5.12: MLP model results with N_{input}=1024, N_{fc1}=512, N_{out}=2. This model was attempting to classify between BPSK and QAM4 modulated signals.

From figure 5.12 we see that MLP was unable to perform for a binary classification between BPSK and QAM4. The validation accuracy hovered around 50% while the training loss function went down to zero. This leads to the conclusion that MLP architecture is not fit to work with raw I & Q samples as its input. It is easy to see why by looking at a few inputs of BPSK and QAM4 modulation signals in figure 5.13.

The frequency offset makes these signal difficult for a MLP architecture to classify. Each input from within a modulation class is quite different because of frequency offset. It is easy for us to see which modulation is not BPSK, because of the sudden dips in the signal, but that is spatial information which is lost on an MLP architecture. A convolutional NNet however, should be able to extract spatial information from the input so that might



(a) BPSK example 1.

(b) BPSK example 2.

(c) QAM4 example 1.

(d) QAM4 examples 2.

Figure 5.13: 1024 I samples from BPSK and QAM4 modulations.

fare better with this type of input.

5.2.2 CNN Results

There has been prior work done on modulation classification using convolutional NNet (CNN) [26]. A similar model is built for testing with few changes in parameters. The template for the CNN model can be seen in figure 5.14.

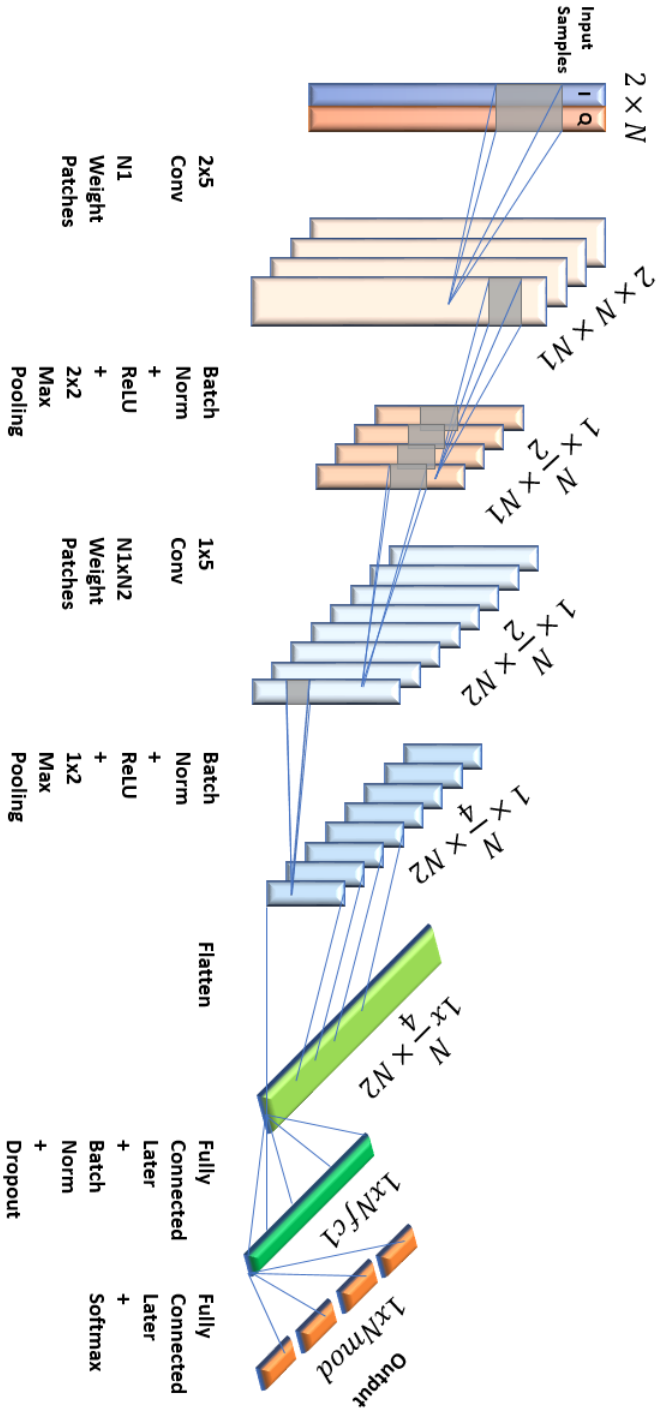


Figure 5.14: Conv Net Model Template

Table 5.2 list the parameter values for the two different models that was built. The last row shows the total number of adaptive parameters used in the build when 1024 I and Q samples are fed in.

Parameters	Model # 1	Model # 2
N1	16	32
N2	32	64
Nfc1	256	1024
Total Adaptive Parameters	2.1mil	16.8mil

Table 5.2: Model Parameters

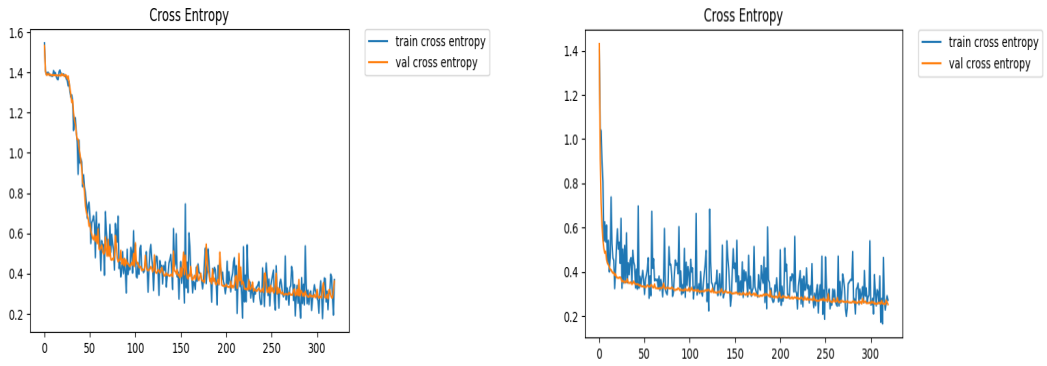
The input is a 2xN image, with the first column containing N I samples, and the second column containing N Q samples. This way the input looks like an image with its spatial information preserved. There were two techniques applied in the model to improve performance: batch normalization (BN) and dropout.

BN is a pre-processing step. There are some algorithms that learn faster if the input has zero mean and unit variance, such is also the case for CNN. BN is applied in between each convolutional layer of CNN. It is usually not applied at the input layer, because we want the CNN to be able to extract features from raw input. BN has two steps to it. First the input is normalized using equation 5.4. Then, the normalized input is scaled by γ and shifted by β as seen in equation 5.5.

$$\hat{x} = (x_i - \mu)/\sqrt{\sigma^2 + \epsilon} \tag{5.4}$$

$$BN(x) = \gamma\hat{x} + \beta \tag{5.5}$$

γ and β are adaptable parameters. Which means that if BN was the wrong thing to do in a particular layer, then CNN can reverse the effect of BN by setting γ to $1/\sqrt{\sigma^2 + \epsilon}$ and β to μ. ϵ is small value of 1e-3 placed to avoid division by zero error. The effects of BN can be seen in figure 5.15. Notice that without BN the weights take longer to minimize the loss function. BN makes it okay to have a higher learning rate, which means faster convergence. Both of the models uses rectifier linear unit (ReLU) activation functions, so the γ term is removed in the BN process. Scaling the x by γ would have minimal effect on the output.

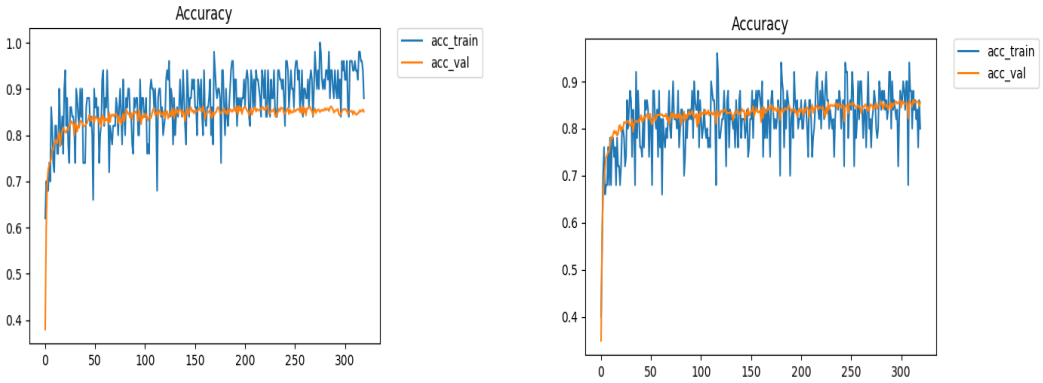


(a) without Batch Normalization (b) with Batch Normalization

Figure 5.15: Training and validation cross entropy loss function for OFDM modulation classification with Model # 1.

In machine learning algorithms, regularization is applied so that the model does not overfit to training data. When a model is overfit to training data, it performs poorly on validation and testing set. Models with a lot of degrees of freedom can overfit easily, and a

CNN is not an exception to that rule. Dropout is a form of regularization that is typically applied in NNets. When the model is trained with, lets say 50% dropout, then half of all the adaptive parameters, such as weights and biases, are zeroed out at random. Those weights do not contribute to the result, nor do they get updated in that training iteration. Figure 5.16 shows the impact of dropout. Without dropout the validation accuracy levels off early even though the training accuracy keeps improving. That is a symptom of an overfit model. Adding dropout regularizes the model and we can see that the validation accuracy conitnues to improve as well. For all the models, the dropout was set to 50%.

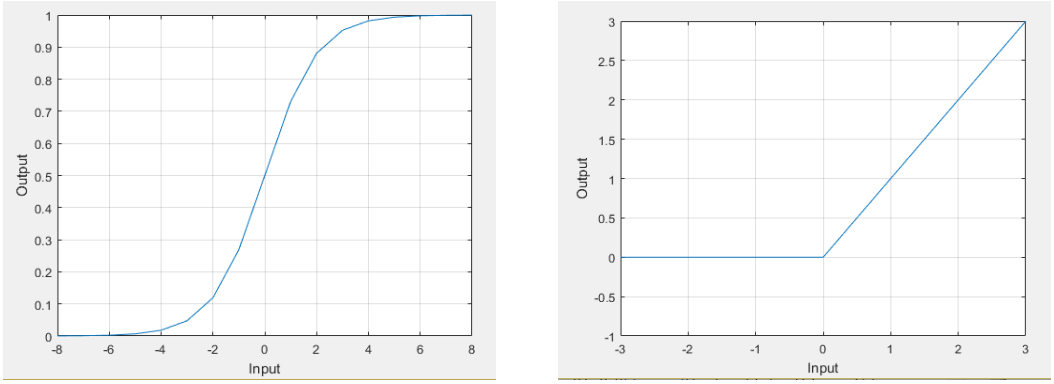


(a) without dropout (b) with 50% dropout

Figure 5.16: Training and Validation accuracy for OFDM modulation classification with Model # 1.

There were a few choices for activation functions. Two populars ones are shown in figure 5.17. There is a problem with deep NNets called vanishing gradient which occurs when the gradients goes to zero. To avoid such problem, ReLUs are good because their gradient cannot go to zero for positive input. With sigmoids the gradient can still go to zero

for both positive and negative inputs. Another benefit of using ReLU is that we get to omit the γ term from BN, because rescaling the normalize input $\hat{x}$ with γ would have minimal effect on the activation output.



(a) Sigmoid (b) Rectifier Linear Unit

Figure 5.17: Activations functions.

For NNets there were lot more examples available for training. At max, only 1024 I & Q samples are fed into the network, but each example file contains 1e6 I & Q samples. 20 blocks of N I & Q samples were extracted from each file, where N could be 256, 512, and 1024. For each case value of N there was 20,000 examples to train and test with.

Model # 1 was giving promising results till the test in which it is trained with multiple SNR data. The results are shown for both models, because it gives us insight into the flexibility of NNets. In the first test, the CNN is fed with three different sets of modulations: linear modulations only, OFDM modulations only, and All modulations. The results for both models at all noise power level is shown in figure 5.18

N	Linear	OFDM	All
256	99.76%	86.04%	92.21%
512	99.94%	92.01%	95.88%
1024	99.98%	99.99%	99.96%

(a) M1 @ 10dB TX gain.

N	Linear	OFDM	All
256	99.70%	86.46%	93.10%
512	99.91%	92.65%	96.40%
1024	100%	100%	99.98%

(b) M2 @ 10dB TX gain.

N	Linear	OFDM	All
256	99.68%	87.20%	93.34%
512	99.96%	93.38%	96.15%
1024	99.98%	99.81%	99.98%

(c) M1 @ 5dB TX gain.

N	Linear	OFDM	All
256	99.70%	87.36%	93.63%
512	99.63%	93.57%	97.03%
1024	100%	100%	99.98%

(d) M2 @ 5dB TX gain.

N	Linear	OFDM	All
256	99.17%	90.98%	95.27%
512	99.78%	95.93%	97.84%
1024	99.93%	99.93%	99.93%

(e) M1 @ 0dB TX gain.

N	Linear	OFDM	All
256	99.15%	92.25%	95.88%
512	99.50%	97.08%	98.53%
1024	99.99%	99.86%	99.96%

(f) M2 @ 0dB TX gain.

Figure 5.18: CNN performance with each SNR data. NNet Model # 1 (M1) is on the left; NNet Model #2 (M2) is on the right. These models are trained for 5 epoch with batch size of 50. Roughly 16K examples per class used for training, and roughly 2K examples per class where used as test set.

The linear modulations are very easily classifiable with just 256 I and Q samples. The classification rate is above 99% for all cases! This is definitely an improvement over SVM which could only achieve 93% accuracy on linear modulations with 200K sample estimates. To classify OFDM modulations at the same rate, at least 1024 I and Q samples are needed. With 1024 samples the network is able to see an entire set of OFDM frames, which are only 960 samples wide. It is possible that CNN is able to localize one of the repetitive frames like preamble or training frames. The results of the last column "All" is consistent in that it is the average of linear and OFDM modulation results. SVM would not have been able to

simultaneously classify both types of modulation with this high of an accuracy. Overall, we see that CNN has significantly higher performance than the SVM classifier.

The confusion matrix for few of the poor classification rate cases are shown in figure 5.19. In these poor performance cases higher order modulations are misclassified more. In the All modulation case, OFDM QAM16 and OFDM QAM32 are heavily misclassified. The raw samples of those modulations are very similar, which explains the mis classification with limited sample size.

BPSK	100	0	0	0
QAM4	0	99.95	0	0.05
QAM16	0	0	99.55	2.85
QAM32	0	0.05	0.35	97.1
Acc = 99.15%	BPSK	QAM4	QAM16	QAM32

(a) 10 dB TX gain with N = 256

OFDM BPSK	100	0	0	0
OFDM QAM4	0	98.55	2.3	0.3
OFDM QAM16	0	1.45	72.75	25.15
OFDM QAM32	0	0	24.95	74.55
Acc = 86.46%	OFDM BPSK	OFDM QAM4	OFDM QAM16	OFDM QAM32

(b) 0 dB TX gain with N = 256

BPSK	100	0	0.1	0	0	0	0	0
QAM4	0	100	0.05	0.3	0	0	0	0
QAM16	0	0	99.6	0.15	0	0	0	0
QAM32	0	0	0.25	99.55	0	0	0	0
OFDM BPSK	0	0	0	0	100	0	0	0
OFDM QAM4	0	0	0	0	0	99.05	2.55	0.5
OFDM QAM16	0	0	0	0	0	0.9	59.25	12.15
OFDM QAM32	0	0	0	0	0	0.05	38.2	87.35
Acc = 93.1%	BPSK	QAM4	QAM16	QAM32	OFDM BPSK	OFDM QAM4	OFDM QAM16	OFDM QAM32

(c) 10 dB TX gain with N = 256

Figure 5.19: CNN confusion matrix from model # 2 for a) linear, b) OFDM and c) All modulation cases with the specified parameters. These are the poor performance cases in their respective categories.

In the last test, both of the models were tested for their classification ability with multiple SNR data. They were fed with examples from all TX gain levels. Figure 5.20 presents the results.

N	Linear	OFDM	All
256	77.71%	77.05%	79.40%
512	79.11%	84.48%	83.96%
1024	78.43%	99.93%	95.88%

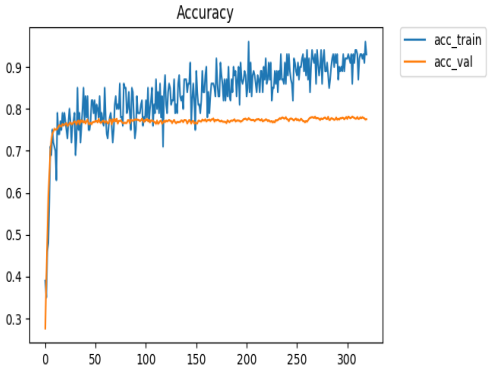
(a) Model # 1 Classification Rate

N	Linear	OFDM	All
256	99.11%	82.23%	90.33%
512	99.46%	90.11%	94.93%
1024	99.82%	99.95%	99.65%

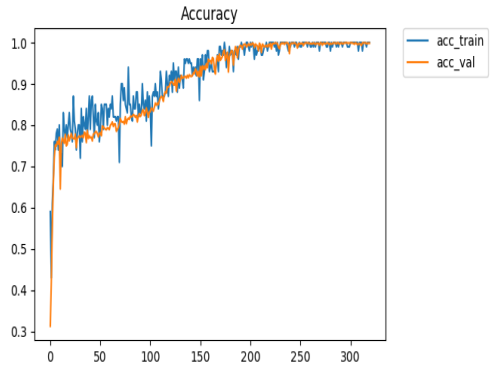
(b) Model # 2 Classification Rate

Figure 5.20: CNN performance of both models when trained with examples from all TX gain level. These models were trained for 10 epoch with batch size of 100. Roughly 16K examples per class and TX gain level were used for training. Tested against roughly 2K examples per class and TX gain level.

Notice that Model # 1 is unable to perform well with this observation set. The training and validation accuracy graphs from one of the runs is shown in figure 5.21. Notice how the validation accuracy plateaus. This is a symptom of a NNet that is unable to learn. One way to fix that is by adding more degrees of freedom, i.e. more weights. So that is how model # 2 was designed. After adding more weights, the new model was able to easily learn the data set, and we get a network that can now work with multiple SNR data.



(a) Model # 1 plateaus at 77%.



(b) Model # 2 continues learning.

Figure 5.21: Training and validation accuracy values for CNN Models.

The confusion matrix for one of poor classification rate case is shown in figure 5.22 a).

Here also OFDM QAM16 and OFDM QAM32 are heavily misclassified. The confusion matrix for model # 2 which is trained with all SNR level and all modulations with 1024 I & Q samples is shown in 5.22 b). CNN can be seen to simultaneously classify all 8 modulations with multiple SNR level really well with just 1024 I & Q samples.

OFDM BPSK	97.35	5.7	0.75	1.05
OFDM QAM4	2.45	87.95	1.95	0.9
OFDM QAM16	0	4.65	71.4	25.8
OFDM QAM32	0.2	1.7	25.9	72.25
Acc = 82.23%	OFDM BPSK	OFDM QAM4	OFDM QAM16	OFDM QAM32

(a) OFDM Modulations with N = 256

BPSK	100	0	0.1	0	0	0	0	0
QAM4	0	100	0.05	0	0	0	0	0
QAM16	0	0	98.65	1	0	0	0	0
QAM32	0	0	1.3	99	0	0	0	0
OFDM BPSK	0	0	0	0	100	0	0	0
OFDM QAM4	0	0	0	0	0	100	0	0
OFDM QAM16	0	0	0	0	0	0	100	0.45
OFDM QAM32	0	0	0	0	0	0	0	99.55
Acc = 99.65%	BPSK	QAM4	QAM16	QAM32	OFDM BPSK	OFDM QAM4	OFDM QAM16	OFDM QAM32

(b) All Modulations with N = 1024

Figure 5.22: CNN confusion matrix of model # 2 with all TX gain data for a) OFDM worse performance case, and b) All modulations best performance case with the specified parameters.

5.3 Comparison

In this section SVM and NNets results are summarized and a comparison is made between the two. The side by side comparison table for SVM and CNN model # 2 is presented in figure 5.23.

In SVM two samples sizes were selected, 50K and 200K. It was seen that classification

with higher sample size estimate showed 5% improvement over the smaller sample size estimate. SVM classifier performed well when it was classifying linear and OFDM modulations separately. SVM could not simultaneously classify both linear and OFDM modulations. In the SNR generalization test it was found that SVM generalized well across multiple SNR, even showing a little improvement for linear modulations.

MLP architecture was unsuitable for raw baseband I & Q samples. CNN architecture was much more suited for these inputs. They performed well in classifying linear modulations with just 256 I & Q samples. OFDM modulations could also be classified with high accuracy, but required at least 1024 I & Q samples. CNN were also good at simultaneous classification of both linear and OFDM modulations. It also generalized well in the multiple SNR case.

0dB TX gain	SVM		CNN		
	Estimate Sample Size		Input Size		
Modulation	50k	200k	256	512	1024
Linear	89.35	94.83	99.20	99.50	99.99
OFDM	99.98	100.00	92.25	97.08	99.86
ALL	78.24	79.90	95.88	98.53	99.96

(a) with TX gain 0dB

5dB TX gain	SVM		CNN		
	Estimate Sample Size		Input Size		
Modulation	50k	200k	256	512	1024
Linear	88.55	92.95	99.70	99.63	100.00
OFDM	98.52	99.80	87.36	93.57	100.00
ALL	75.63	78.35	93.63	97.03	99.98

(b) with TX gain 5dB

10dB TX gain	SVM		CNN		
	Estimate Sample Size		Input Size		
Modulation	50k	200k	256	512	1024
Linear	86.83	92.90	99.70	99.91	100.00
OFDM	99.30	99.95	86.90	92.65	100.00
ALL	71.81	73.41	93.10	96.40	99.98

(c) with TX gain 10dB

All dB TX gain	SVM		CNN		
	Estimate Sample Size		Input Size		
Modulation	50k	200k	256	512	1024
Linear	91.49	97.29	99.11	99.46	99.82
OFDM	99.23	99.89	82.23	90.11	99.95
ALL	78.85	81.91	90.33	94.93	99.65

(d) with all TX gain data

Figure 5.23: Performance of SVM and CNN at different TX gain level, and with all TX gain level.

It is clear to see that for overall performance across the board, CNN is the winner. The obvious drawback to using the CNN model is huge number of weights that is used. The biggest number of weights is between the second convolution layer and the first fully connected layer. It should be possible to reduce the number of weights, by adding more sub sampling layers like convolutional or max pooling, without losing the classification accuracy. This is something worth investigating in future.

6

Conclusion and Future Work

The biggest takeaway from these experiments is that NNet is more flexible for the task of modulation recognition. For a SVM classifier to perform well, it needs to be trained with good features, which is up to the experimenter. A proper NNet can easily learn useful features from complex data set if it has the right architecture and enough adaptable parameters. With just 1024 I & Q samples, CNN's classification accuracy was consistently above 99%. SVM is still very powerful in the hands of an experienced experimenter who knows what features works best. However, a robust SIG INT or CR receiver would probably do better with a NNet's flexibility with a complex data set.

There are many things to try out to continue this experiment. It would be interesting to see how well these classifiers perform with changes in the modulation parameters. For linear modulations experimenters can try adding different pulse shaping filters at the transmitter. They may also try varying the no. of samples per symbol, and symbol time. For OFDM

modulations, they can change the FFT size, or make the frame sequence longer, for example putting 20 data frames instead of 10 data frames between preamble and training frames. They can also try removing preamble and training frames all together to check if CNN performs well without these repetitive frames. SVM's capability can be too limited with cumulants, so experimenter may try using wavelet features which are also very good at modulation classification. Experimenters can also try adding other types of modulations to increase the complexity for the classifiers. As suggested in last chapter, experimenters can also focus on reducing the number of weights in CNN by adding extra sub sampling layers.

The intention of doing this thesis was to see if NNets were viable for modulation classification of OFDM modulations. Even though the number of parameters involved in the complete model are too high, it is clear to see from the results that CNN can perform excellently at classifying OFDM modulations.

A

Relevant Source Code

A.1 MATLAB Code

A.1.1 Linear modulation bit generator

BPSK bit generator

```
% generate random ints
for seed = 1:1000
rng(seed);
prbs = randi([0,1],10000,1)*2-1;

% store that as int
fileID = fopen(['prbs_',num2str(seed),'.txt'],'w');
fwrite(fileID, prbs, 'int');
fclose(fileID);
end
```

QAM symbol generator

```
% generate random integers from 0 to M-1
for seed = 1:1000
rng(seed);
M = 4; % 16 for QAM16, 32 for QAM32
prbs = randi([0,M-1],10000,1);

% store that as int
fileID = fopen(['qam4_sym_',num2str(seed),'.txt'],'w');  % for QAM4
%fileID = fopen(['qam16_sym_',num2str(seed),'.txt'],'w'); % for QAM16
%fileID = fopen(['qam32_sym_',num2str(seed),'.txt'],'w'); % for QAM32
fwrite(fileID, prbs, 'int');
fclose(fileID);
end
```

A.1.2 OFDM frame sequence generator

```
clc; clear all; close all;

%% Premable
```

```
N = 64;
preambleVals = (1.472+1i*1.472)*[-1, -1, 1, 1, 1, 1, 1, -1, 1, -1, -1, 1];
preambleIdx  = [5:4:25,41:4:61];
preamble     = zeros(N,1);
preamble(preambleIdx) = preambleVals;
preambleFrame = preamble;

%% Training

nullCarriers  = [0,27:37] + 1;
pilotCarriers = [7,21,43,57] + 1;
dataCarriers  = [1:6,8:20,22:26,38:42,44:56,58:63] + 1;

% altenative ones +1,-1,+1,... on data carriers
trainVals      = ones(2,length(dataCarriers)/2);
trainVals(2,:) = -1.*trainVals(2,:);
trainVals      = trainVals(:);

% Train frame
training                = zeros(N,1);
training(dataCarriers) = trainVals;
trainingFrame           = complex(training,zeros(N,1));


for seed = 1:1000
    rng(seed);
    %% Data Frame

    % generate random bpsk syms
    nDataFrames = 10000;
    prbs        = randi([0,1],length(dataCarriers)*nDataFrames,1)*2-1;
    prbsSym     = complex(prbs,zeros(size(prbs))); % for OFDM BPSK
    % Un comment this for OFDM QAM4
    %prbs        = randi([0,3],length(dataCarriers)*nDataFrames,1);
    % prbsSym     = qammod(prbs,4);
    % Un comment this for OFDM QAM16
    %prbs        = randi([0,15],length(dataCarriers)*nDataFrames,1);
    % prbsSym     = qammod(prbs,16);
    % Un comment this for OFDM QAM16
    %prbs        = randi([0,31],length(dataCarriers)*nDataFrames,1);
    % prbsSym     = qammod(prbs,32);

    %% Interleave frames

    frameNo = 0;
    idx     = 1;
    dataIdx = 1;
    for k = 1:nDataFrames

        if(mod(frameNo,10) == 1)
            Out(idx:idx-1+N) = preambleFrame;
            idx = idx + N;
            Out(idx:idx-1+N) = trainingFrame;
            idx = idx + N;
        end

        dataFrame                = complex(zeros(N,1),zeros(N,1));
        dataFrame(pilotCarriers) = complex(1,0);
        dataFrame(dataCarriers)  = prbsSym(dataIdx:dataIdx-1+length(dataCarriers));
        dataIdx                  = dataIdx + length(dataCarriers);
        Out(idx:idx-1+N)         = dataFrame;
        idx                      = idx + N;

        frameNo = frameNo + 1;
    end

    % store that as float
    outReal  = real(Out);
    outImag  = imag(Out);
    outFrame = [outReal; outImag];
    outFrame = outFrame(:);

    fileID = fopen(['frame_802_bpsk_',num2str(seed),'.txt'],'w');
    fwrite(fileID, outFrame, 'float');
    fclose all;
```

```
end
```

A.1.3 Linear modulation receiver

```
clc; clear all; close all;

%% Read received signal

% fileIn = fopen('bpsk_cap1.dat','r');
fileIn = fopen('bpsk_cap2.dat','r');
invar  = fread(fileIn, 'float');
fclose all;

%% Split into I and Q samples

bb_samp = invar(1:1e5);

I = bb_samp(1:2:end);
Q = bb_samp(2:2:end);

%% Remove DC Offset

I = I - mean(I);
Q = Q - mean(Q);

%% Unrecovered Constellation Diagram
figure, plot(I,Q,'x'); title('Un recovered Const');
figure, pwelch(I+1i.*Q,[],[],[], 1e6, 'centered'); title('Rx Spectrum');

%% Constellation recovery by supplying a freq offset correction factor

bb_sig = I + 1i*Q; % baseband signal
fs     = 2e6; % sampling rate
t      = (0:length(I)-1)/fs; % time scale

% frec  = 309; %bpsk_cap1.dat
% deg   = -60;
frec  = 217.5; %bpsk_cap2.dat
deg   = 77;
recLO = exp(-1i.*2*pi*frec.*t.'); % correction term
rxRec = bb_sig .* recLO .*exp(1i.*deg*pi/180); % recovered term

%% Recovered Constellation Diagram

N  = length(rxRec);
N1 = floor(0.25*N);
N2 = floor(0.5*N);
figure, plot(rxRec(1:N1), 'xr'); title('Recovered Const');
hold on,
plot(rxRec(N1:N2), 'xm');
plot(rxRec(N2:N), 'xg');

%% Eye diagram
sps    = 20; % samples per symbol
rx_eye = reshape(rxRec(41:end), sps*2, []);
figure, plot(real(rx_eye),'x'); title('In Phase Eye Diagram');

%% Decimated Constellation Diagram
start_phase = 25+sps;
bb_dc       = rxRec(start_phase:sps:end);
figure, plot(bb_dc, 'x'); title('Decimated Constellation');

%% Remove edge points

% edge = 18; %bpsk_cap1
edge = 22; %bpsk_cap2
bb_dc  = rxRec(edge:end);
N      = floor(length(bb_dc)/(sps));
bb_eye = reshape(bb_dc(1:N*sps), sps, N);
bb_eye_trunc = bb_eye(4:19,:);

bb_eye_trunc = bb_eye_trunc(:);
N  = length(bb_eye_trunc);
N1 = floor(0.33*N);
N2 = floor(0.66*N);
```

```
figure, plot(bb_eye_trunc(1:N1), 'xr'); title('Recovered Const without edge samples');
grid on;
hold on,
plot(bb_eye_trunc(N1:N2), 'xm');
plot(bb_eye_trunc(N2:N), 'xg');
axis(0.06*[-1,1,-1,1])
```

A.1.4 OFDM modulation receiver

```
clc; clear all; close all;

%% Read captured file and transmitted file

% captured file example
noisyF    = fopen('ofdm_bpsk_100.dat', 'r'); % Noisy File
noisyData = fread(noisyF, 'float');
noisyData = noisyData(1:1e6);

% transmitted file example
cleanF    = fopen('ofdm_bpsk_clean_100.dat', 'r'); % Tx File % current format.
cleanData = fread(cleanF, 'float');

%% Split into I and Q samples

% Baseband samples
Inoisy  = noisyData(1:2:end);
Qnoisy  = noisyData(2:2:end);
bbnoisy = Inoisy + 1i.*Qnoisy;
noisyFs = 25e6;

Iclean  = cleanData(1:2:end);
Qclean  = cleanData(2:2:end);
bbClean = Iclean + 1i.*Qclean;
cleanFs = 25e6;

%% Unrecovered constellation diagrams

figure, pwelch(bbnoisy, [], [], [], noisyFs, 'centered'); title('Baseband Spectrum');
figure, plot(bbnoisy, 'x'); title('Constellation diagram')

%% Remove DC offset

I = real(bbnoisy) - mean(real(bbnoisy));
Q = imag(bbnoisy) - mean(imag(bbnoisy));
bbnoisy = I + 1i.*Q;

%% Frequency offset correction

% Adjust this so that the pilot's horizontal.
frec = 250;
deg  = -120;
t = ((0:numel(bbnoisy)-1)/(noisyFs)).';
bbnoisy = bbnoisy(:).*exp(-1i*2*pi*frec.*t)*exp(1i.*deg*pi/180);

%% Preamble based Frame detection (better way listed ahead)

rx = bbnoisy;
N  = 64;
L  = 16;

% Find Frame Boundaries using preambles
P = zeros(length(rx)-(N+L),1);
R = zeros(length(rx)-(N+L),1);
corr = zeros(length(rx)-(N+L),1);
for i = 1:length(rx)-(N+L)

    % preambles are identical halves
    samp1 = rx(i:i-1+N/2);
    samp2 = rx(i+N/2:i+N/2-1+N/2);

    P(i) = sum(samp1.*conj(samp2));
    R(i) = sum(abs(samp2.^2));
    corr(i) = P(i)/R(i);
end

figure, stem(abs(corr(1:2500)));
```

```
title('Preamble detection');

%% Correlation against clean signal to get Frame boundary

[r,lags] = xcorr(bbnoisy, bbClean);
figure, plot(lags, abs(r));
title('correlation against transmitted signal');
% Use the peak index as the startIdx and add 80 to it


%% Remove Cyclic Prefix

startIdx = 18109+80; % start of preamble frame
nFrames  = 400;      % number of frames to extract from the signal
bbFrames = reshape(bbnoisy(startIdx:startIdx-1+(N+L)*nFrames),(N+L),[]); % frames

bbSig    = bbFrames(L+1:end,:); % cp removed from frames

%% Extract traiing symbols to compute FDE coeffecients

% Frame Sequence P | T | D x 10 | P | T | D x 10 | P ...

trainFrameIdx = [2:12:nFrames];
trainFrames   = bbSig(:,trainFrameIdx);

% Train Frame Format
nullCarriers  = [0,27:37] + 1;
pilotCarriers = [7,21,43,57] + 1;
dataCarriers  = [1:6,8:20,22:26,38:42,44:56,58:63] + 1;

trainFftOut      = fft(trainFrames,N);
trainVals        = trainFftOut(dataCarriers,:);
% trainVals values should be alternating ones if there are no impairments
nullVals         = trainFftOut(nullCarriers,:);
figure, plot(trainVals,'x');
title('Train Vals');

figure, plot(nullVals,'x');
title('Null Vals');

%% Extract Data Pilots to check frequency offset

dataFrameIdx = 1:nFrames;
dataFrameIdx(1:12:end) = [];
dataFrameIdx(1:11:end) = [];
dataFrames   = bbSig(:,dataFrameIdx);

dataFftOut      = fft(dataFrames,N);
pilotVals       = dataFftOut(pilotCarriers,:);
figure, plot(angle(pilotVals(1,:))*180/pi,'x');
title('Pilot Phase');
hold on,
plot(angle(pilotVals(2,:))*180/pi,'rx');
plot(angle(pilotVals(3,:))*180/pi,'mx');
plot(angle(pilotVals(4,:))*180/pi,'gx');
legend('1st','2nd','3rd','4th');
ylim([-360,360]);
% If there is a freq offset then these angles will not be horizontal.
% add freq offset correction term till they are horizontal.

%% FDE coeffs based on Training Frames

% zero forcing equalizer
equalizer = zeros(N,size(trainVals,2));
equalizer(dataCarriers,:)          = 1./(trainVals);
equalizer(dataCarriers(2:2:end),:) = equalizer(dataCarriers(2:2:end),:)*-1;

%% Equalize data frame

eqIdx  = 1;
dataEQ = zeros(size(dataFrames));
for k = 1:size(dataFrames,2)

    currEqualizer = equalizer(:,eqIdx);
    dataEQ(:,k)   = dataFftOut(:,k).*currEqualizer;
    if(mod(k,10)==0)
```

```
        % one training frame for every 10 data frames.
        % increment after 10 data frames have been equalized.
        eqIdx = eqIdx+1;
    end
end
dataOut = dataEQ(dataCarriers,:);
figure, plot(dataOut,'x'); title('Recovered Constellation After Equalizer');
ylim([-1.5,1.5])
xlim([-1.5,1.5])
```

A.1.5 Cumulant function

```
function [rcum] = compute_cumulants(data, ncumulant)

%ncumulant represent total number of cumulants to be calculated.

    cum = zeros(ncumulant,1);

    cum(1) = mean(data);

    % pre compute non central moments
    mom = zeros(ncumulant,1);
    for n = 1:ncumulant
       mom(n) = mean(data.^n);
    end

    for n = 2:ncumulant
        y = 0;
        for m = 1:n-1
            if((n-m) ~= 0)
                y = y + nchoosek(n-1,m-1)*cum(m)*mom(n-m);
            else
                y = y + nchoosek(n-1,m-1)*cum(m);
            end
        end

        cum(n) = mom(n) - y;

    end
    rcum = cum;
end
```

A.1.6 Cumulant script for captured data

```
% Computes cumulants of captured data for all of the 8 modulations.
% Ignores transients at the begining.
% change sample size between 50k and 200k

clc; clear all; close all;
format long;
home = 'path_to_home_dir';

% Unnormalized dir
unnorm_dir = 'path_to_save_location_for_unnormalize_cumulants';
% normalized dir
norm_dir = 'path_to_save_location_for_normalized_cumulants';

% switch
type = 'unnormalized'; % unnormalized or normalized

% do CTRL + H and replace bpsk with qam4 or another modulations and rerun
% the file
%% bpsk captures
tic
cd(home)

% cd into directory with raw captures
cd plus10/bpsk

bpsk_cum = zeros(1000, 16); % placeholder, 1000 examples per modulation class
h = waitbar(0, 'Please wait...');
for i = 1:1000
```

```
    waitbar(i/1000, h, sprintf('computing cumulant on %d', i));
    % read captured data
    filename = ['bpsk_10dB_',num2str(i),'.dat'];
    infile   = fopen(filename, 'r');
    invar    = fread(infile, 'float');
    fclose(infile);

    % split into I and Q
    I  = invar(1:2:end);
    Q  = invar(2:2:end);
    bb = I + 1i.*Q;
    bb = bb(1000:999+50e3); % 50k sample estimate
    %bb = bb(1000:999+50e3); % 200k sample estimate

    % compute cumulants for I and Q samples (un normalized cumulants)
    bpsk_cum(i,1:8) = compute_cumulants(real(bb),8);
    bpsk_cum(i,9:16) = compute_cumulants(imag(bb),8);

    % save
    if(strcmp(type,'unnormalized'))
     cd(unnorm_dir);
     cd bpsk;
     filename = ['bpsk_10dB_cumulant_unnorm.mat'];
    end

    if(strcmp(type,'normalized'))
     cd(norm_dir);
     cd bpsk;
     filename = ['bpsk_10dB_cumulant_norm.mat'];
     % normalize the signal
     bb_norm = bb./max(abs(bb));
     % compute cumulants
     bpsk_cum(i,1:8) = compute_cumulants(real(bb_norm),8);
     bpsk_cum(i,9:16) = compute_cumulants(imag(bb_norm),8);
    end

    cd(home)
    cd plus10/bpsk
end
close(h);
toc

% cd into where you want to save the file
cd('path_to_save_location\cumulants\separate\50k');
%cd('path_to_save_location\cumulants\separate\200k');
save(filename, 'bpsk_cum');
```

A.1.7 SVM for single SNR data

```
clc; clear; close all;
addpath('../Subfunctions')
addpath(genpath('../PRT-master'))
load fig_pos.mat

home = pwd;
% path to where the cumulants are stored
data_dir = 'path_to_cumulants_dir\cumulants\separate\200k\original';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\200k\plus5';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\200k\plus10';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\50k\original';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\50k\plus5';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\50k\plus10';

cd(data_dir)

standardMods = {'bpsk', 'qam4', 'qam16', 'qam32'};
ofdmMods     = {'ofdm_bpsk', 'ofdm_qam4', 'ofdm_qam16', 'ofdm_qam32'};
allMods      = [standardMods ofdmMods];

modsClass = allMods;
normVal   = 'norm';  % norm or unnorm

% Load cumulants
```

```
Nmod = length(modsClass);
CumData  = zeros(1000*Nmod,16);
for i = 1:Nmod
    filename = [modsClass{i},'_cumulant_',normVal,'.mat'];  % for 0dB caps
   %filename = [modsClass{i},'_5dB_cumulant_',normVal,'.mat']; % for 5dB caps
   %filename = [modsClass{i},'_10dB_cumulant_',normVal,'.mat']; % for 10dB caps
    load(filename);
    CumData((i-1)*1000+1:i*1000,:) = data; % 1000 examples per modulation class
end

%% OFDM vs Non OFDM (SVM 1)

standardData = CumData(1:4000,:);
OfdmData     = CumData(4001:8000,:);

data         = [standardData; OfdmData];
target       = zeros(8000,1);
target(4001:end) = 1;

ds         = prtDataSetClass(data, target);
dsNorm     = rt(prtPreProcZmuv,ds);
pca        = prtPreProcPca('nComponents',16);
pcaNorm    = pca.train(dsNorm);
dsPca      = pcaNorm.run(ds);
dsPca.classNames{1} = 'Standard';
dsPca.classNames{2} = 'OFDM';

classifier =  prtClassBinaryToMaryOneVsAll;    % Create a classifier
classifier.baseClassifier = prtClassLibSvm;    % Set the binary
classifier.internalDecider = prtDecisionMap;

yOutKfolds = classifier.kfolds(dsPca, 10);
yOutKfolds.classNames{1} = 'Standard';
yOutKfolds.classNames{2} = 'OFDM';
prtScoreConfusionMatrix(yOutKfolds,dsPca);

%% Standard Only

standardData = CumData(1:4000,:);
data   = standardData;
target = zeros(4000, 1);
target(1001:2000) = 1;
target(2001:3000) = 2;
target(3001:4000) = 3;

ds         = prtDataSetClass(data, target);
dsNorm     = rt(prtPreProcZmuv,ds);
pca        = prtPreProcPca('nComponents',16);
pcaNorm    = pca.train(dsNorm);
dsPca      = pcaNorm.run(dsNorm);
dsPca.classNames{1} = 'BPSK';
dsPca.classNames{2} = 'QAM4';
dsPca.classNames{3} = 'QAM16';
dsPca.classNames{4} = 'QAM32';

classifier =  prtClassBinaryToMaryOneVsAll;    % Create a classifier
classifier.baseClassifier = prtClassLibSvm;    % Set the binary
classifier.baseClassifier.cost  = 30;          % Cost parameter
classifier.baseClassifier.gamma = 0.01;        % Gamme parameter
classifier.internalDecider = prtDecisionMap;

[yOutKfolds, TrainedActions, CrossValKeys] = classifier.kfolds(dsPca, 10);
yOutKfolds.classNames{1} = 'BPSK';
yOutKfolds.classNames{2} = 'QAM4';
yOutKfolds.classNames{3} = 'QAM16';
yOutKfolds.classNames{4} = 'QAM32';
prtScoreConfusionMatrix(yOutKfolds,dsPca);

%% OFDM Only
OfdmData     = CumData(4001:8000,:);
data   = OfdmData;
target = zeros(4000, 1);
target(1001:2000) = 1;
target(2001:3000) = 2;
```

```
target(3001:4000) = 3;

ds          = prtDataSetClass(data, target);
dsNorm      = rt(prtPreProcZmuv,ds);
pca         = prtPreProcPca('nComponents',16);
pcaNorm     = pca.train(dsNorm);
dsPca       = pcaNorm.run(dsNorm);
dsPca.classNames{1} = 'OFDM BPSK';
dsPca.classNames{2} = 'OFDM QAM4';
dsPca.classNames{3} = 'OFDM QAM16';
dsPca.classNames{4} = 'OFDM QAM32';

classifier =  prtClassBinaryToMaryOneVsAll;    % Create a classifier
classifier.baseClassifier = prtClassLibSvm;    % Set the binary
classifier.baseClassifier.cost  = 10;          % cost parameter
classifier.baseClassifier.gamma = 0.01;        % gamma parameter
classifier.internalDecider = prtDecisionMap;

yOutKfolds = classifier.kfolds(dsPca, 10);
yOutKfolds.classNames{1} = 'OFDM BPSK';
yOutKfolds.classNames{2} = 'OFDM QAM4';
yOutKfolds.classNames{3} = 'OFDM QAM16';
yOutKfolds.classNames{4} = 'OFDM QAM32';
prtScoreConfusionMatrix(yOutKfolds,dsPca);

%% All

data   = CumData;
target = zeros(8000, 1);
target(1001:2000) = 1;
target(2001:3000) = 2;
target(3001:4000) = 3;
target(4001:5000) = 4;
target(5001:6000) = 5;
target(6001:7000) = 6;
target(7001:8000) = 7;

ds          = prtDataSetClass(data, target);
dsNorm      = rt(prtPreProcZmuv,ds);
pca         = prtPreProcPca('nComponents',16);
pcaNorm     = pca.train(dsNorm);
dsPca       = pcaNorm.run(dsNorm);
dsPca.classNames{1} = 'BPSK';
dsPca.classNames{2} = 'QAM4';
dsPca.classNames{3} = 'QAM16';
dsPca.classNames{4} = 'QAM32';
dsPca.classNames{5} = 'OFDM BPSK';
dsPca.classNames{6} = 'OFDM QAM4';
dsPca.classNames{7} = 'OFDM QAM16';
dsPca.classNames{8} = 'OFDM QAM32';

classifier =  prtClassBinaryToMaryOneVsAll;    % Create a classifier
classifier.baseClassifier = prtClassLibSvm;    % Set the binary
classifier.baseClassifier.cost  = 30;    % Set the binary
classifier.baseClassifier.gamma = 0.01;    % Set the binary
classifier.internalDecider = prtDecisionMap;

yOutKfolds = classifier.kfolds(dsPca, 10);
yOutKfolds.classNames{1} = 'BPSK';
yOutKfolds.classNames{2} = 'QAM4';
yOutKfolds.classNames{3} = 'QAM16';
yOutKfolds.classNames{4} = 'QAM32';
yOutKfolds.classNames{5} = 'OFDM BPSK';
yOutKfolds.classNames{6} = 'OFDM QAM4';
yOutKfolds.classNames{7} = 'OFDM QAM16';
yOutKfolds.classNames{8} = 'OFDM QAM32';
prtScoreConfusionMatrix(yOutKfolds,dsPca);

%% gamme and c search grid

% This can only work with binary classification

Data   = CumData(2001:4000,:); % qam16 & qam32
```

```
target = zeros(2000, 1);
target(1001:2000) = 1;

ds          = prtDataSetClass(Data, target);
dsNorm      = rt(prtPreProcZmuv,ds);
pca         = prtPreProcPca('nComponents',16); % create PCA selfect
pcaNorm     = pca.train(dsNorm); % train PCA
dsPca       = pcaNorm.run(dsNorm); % create new dataset based on PCA
ds          = dsPca;

gammaVec = logspace(-2,1,10);
costVec  = logspace(-2,1,10);

auc         = nan(length(gammaVec),length(costVec));
kfoldsInds = ds.getKFoldKeys(3);
for gammaInd = 1:length(gammaVec);
    for costInd = 1:length(costVec);

        % classifier is defined in previous sections
        c = classifier;
        c.baseClassifier.cost = costVec(costInd);
        c.baseClassifier.gamma = gammaVec(gammaInd);
        yOut = crossValidate(c,ds,kfoldsInds);
        auc(gammaInd,costInd) = prtScoreAuc(yOut);

        imagesc(auc,[.1  1]);
        colorbar
        drawnow;
    end
end
title('AUC vs. Gamma Index (Vertical) and Cost Index (Horizontal)');
```

A.1.8 SVM for multiple SNR data

```
clc; clear; close all;
addpath('../Subfunctions')
addpath(genpath('../PRT-master'))
load fig_pos.mat

home = pwd;
% path to where the cumulants are stored
data_dir = 'path_to_cumulants_dir\cumulants\separate\200k\original';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\200k\plus5';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\200k\plus10';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\50k\original';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\50k\plus5';
%data_dir = 'path_to_cumulants_dir\cumulants\separate\50k\plus10';

cd(data_dir)

standardMods = {'bpsk', 'qam4', 'qam16', 'qam32'};
ofdmMods     = {'ofdm_bpsk', 'ofdm_qam4', 'ofdm_qam16', 'ofdm_qam32'};
allMods      = [standardMods ofdmMods];

modsClass = allMods;
normVal   = 'norm';  % norm or unnorm

% Load cumulants
Nmod = length(modsClass);
CumData  = zeros(1000*Nmod,16);
for i = 1:Nmod
    filename = [modsClass{i},'_cumulant_',normVal,'.mat'];  % for 0dB caps
   %filename = [modsClass{i},'_5dB_cumulant_',normVal,'.mat']; % for 5dB caps
   %filename = [modsClass{i},'_10dB_cumulant_',normVal,'.mat']; % for 10dB caps
    load(filename);
    CumData((i-1)*1000+1:i*1000,:) = data; % 1000 examples per modulation class
end

%% OFDM vs Non OFDM (SVM 1)

standardData = CumData(1:4000,:);
OfdmData     = CumData(4001:8000,:);

data         = [standardData; OfdmData];
```

```
target       = zeros(8000,1);
target(4001:end) = 1;

ds          = prtDataSetClass(data, target);
dsNorm      = rt(prtPreProcZmuv,ds);
pca         = prtPreProcPca('nComponents',16);
pcaNorm     = pca.train(dsNorm);
dsPca       = pcaNorm.run(ds);
dsPca.classNames{1} = 'Standard';
dsPca.classNames{2} = 'OFDM';

classifier =  prtClassBinaryToMaryOneVsAll;    % Create a classifier
classifier.baseClassifier = prtClassLibSvm;    % Set the binary
classifier.internalDecider = prtDecisionMap;

yOutKfolds = classifier.kfolds(dsPca, 10);
yOutKfolds.classNames{1} = 'Standard';
yOutKfolds.classNames{2} = 'OFDM';
prtScoreConfusionMatrix(yOutKfolds,dsPca);

%% Standard Only

standardData = CumData(1:4000,:);
data   = standardData;
target = zeros(4000, 1);
target(1001:2000) = 1;
target(2001:3000) = 2;
target(3001:4000) = 3;

ds          = prtDataSetClass(data, target);
dsNorm      = rt(prtPreProcZmuv,ds);
pca         = prtPreProcPca('nComponents',16);
pcaNorm     = pca.train(dsNorm);
dsPca       = pcaNorm.run(dsNorm);
dsPca.classNames{1} = 'BPSK';
dsPca.classNames{2} = 'QAM4';
dsPca.classNames{3} = 'QAM16';
dsPca.classNames{4} = 'QAM32';

classifier =  prtClassBinaryToMaryOneVsAll;    % Create a classifier
classifier.baseClassifier = prtClassLibSvm;    % Set the binary
classifier.baseClassifier.cost  = 30;          % Cost parameter
classifier.baseClassifier.gamma = 0.01;        % Gamme parameter
classifier.internalDecider = prtDecisionMap;

[yOutKfolds, TrainedActions, CrossValKeys] = classifier.kfolds(dsPca, 10);
yOutKfolds.classNames{1} = 'BPSK';
yOutKfolds.classNames{2} = 'QAM4';
yOutKfolds.classNames{3} = 'QAM16';
yOutKfolds.classNames{4} = 'QAM32';
prtScoreConfusionMatrix(yOutKfolds,dsPca);

%% OFDM Only
OfdmData     = CumData(4001:8000,:);
data   = OfdmData;
target = zeros(4000, 1);
target(1001:2000) = 1;
target(2001:3000) = 2;
target(3001:4000) = 3;

ds          = prtDataSetClass(data, target);
dsNorm      = rt(prtPreProcZmuv,ds);
pca         = prtPreProcPca('nComponents',16);
pcaNorm     = pca.train(dsNorm);
dsPca       = pcaNorm.run(dsNorm);
dsPca.classNames{1} = 'OFDM BPSK';
dsPca.classNames{2} = 'OFDM QAM4';
dsPca.classNames{3} = 'OFDM QAM16';
dsPca.classNames{4} = 'OFDM QAM32';

classifier =  prtClassBinaryToMaryOneVsAll;    % Create a classifier
classifier.baseClassifier = prtClassLibSvm;    % Set the binary
classifier.baseClassifier.cost  = 10;          % cost parameter
classifier.baseClassifier.gamma = 0.01;        % gamma parameter
```

```
classifier.internalDecider = prtDecisionMap;

yOutKfolds = classifier.kfolds(dsPca, 10);
yOutKfolds.classNames{1} = 'OFDM BPSK';
yOutKfolds.classNames{2} = 'OFDM QAM4';
yOutKfolds.classNames{3} = 'OFDM QAM16';
yOutKfolds.classNames{4} = 'OFDM QAM32';
prtScoreConfusionMatrix(yOutKfolds,dsPca);

%% All

data   = CumData;
target = zeros(8000, 1);
target(1001:2000) = 1;
target(2001:3000) = 2;
target(3001:4000) = 3;
target(4001:5000) = 4;
target(5001:6000) = 5;
target(6001:7000) = 6;
target(7001:8000) = 7;


ds         = prtDataSetClass(data, target);
dsNorm     = rt(prtPreProcZmuv,ds);
pca        = prtPreProcPca('nComponents',16);
pcaNorm    = pca.train(dsNorm);
dsPca      = pcaNorm.run(dsNorm);
dsPca.classNames{1} = 'BPSK';
dsPca.classNames{2} = 'QAM4';
dsPca.classNames{3} = 'QAM16';
dsPca.classNames{4} = 'QAM32';
dsPca.classNames{5} = 'OFDM BPSK';
dsPca.classNames{6} = 'OFDM QAM4';
dsPca.classNames{7} = 'OFDM QAM16';
dsPca.classNames{8} = 'OFDM QAM32';

classifier =  prtClassBinaryToMaryOneVsAll;    % Create a classifier
classifier.baseClassifier = prtClassLibSvm;    % Set the binary
classifier.baseClassifier.cost  = 30;    % Set the binary
classifier.baseClassifier.gamma = 0.01;    % Set the binary
classifier.internalDecider = prtDecisionMap;

yOutKfolds = classifier.kfolds(dsPca, 10);
yOutKfolds.classNames{1} = 'BPSK';
yOutKfolds.classNames{2} = 'QAM4';
yOutKfolds.classNames{3} = 'QAM16';
yOutKfolds.classNames{4} = 'QAM32';
yOutKfolds.classNames{5} = 'OFDM BPSK';
yOutKfolds.classNames{6} = 'OFDM QAM4';
yOutKfolds.classNames{7} = 'OFDM QAM16';
yOutKfolds.classNames{8} = 'OFDM QAM32';
prtScoreConfusionMatrix(yOutKfolds,dsPca);

%% gamme and c search grid

% This can only work with binary classification

Data   = CumData(2001:4000,:); % qam16 & qam32
target = zeros(2000, 1);
target(1001:2000) = 1;

ds         = prtDataSetClass(Data, target);
dsNorm     = rt(prtPreProcZmuv,ds);
pca        = prtPreProcPca('nComponents',16); % create PCA selfect
pcaNorm    = pca.train(dsNorm); % train PCA
dsPca      = pcaNorm.run(dsNorm); % create new dataset based on PCA
ds         = dsPca;

gammaVec = logspace(-2,1,10);
costVec  = logspace(-2,1,10);

auc        = nan(length(gammaVec),length(costVec));
kfoldsInds = ds.getKFoldKeys(3);
for gammaInd = 1:length(gammaVec);
```

```
    for costInd = 1:length(costVec);

        % classifier is defined in previous sections
        c = classifier;
        c.baseClassifier.cost = costVec(costInd);
        c.baseClassifier.gamma = gammaVec(gammaInd);
        yOut = crossValidate(c,ds,kfoldsInds);
        auc(gammaInd,costInd) = prtScoreAuc(yOut);

        imagesc(auc,[.1 1]);
        colorbar
        drawnow;
    end
end
title('AUC vs. Gamma Index (Vertical) and Cost Index (Horizontal)');
```

A.2 GNU Radio Code

A.2.1 qam4_ic implementation in OOT module

Implementation header file

```
/* -*- c++ -*- */
/*
 * Copyright 2017 <+YOU OR YOUR COMPANY+>.
 *
 * This is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#ifndef INCLUDED_MYNEWMOD_QAM4_IC_IMPL_H
#define INCLUDED_MYNEWMOD_QAM4_IC_IMPL_H

#include <mynewmod/qam4_ic.h>

namespace gr {
  namespace mynewmod {

    class qam4_ic_impl : public qam4_ic
    {
     private:
      // Nothing to declare in this block.

     public:
      qam4_ic_impl();
      ~qam4_ic_impl();

      // Where all the action really happens
      void forecast (int noutput_items, gr_vector_int &ninput_items_required);

      int general_work(int noutput_items,
                       gr_vector_int &ninput_items,
                       gr_vector_const_void_star &input_items,
                       gr_vector_void_star &output_items);
    };
```

```
  } // namespace mynewmod
} // namespace gr

#endif /* INCLUDED_MYNEWMOD_QAM4_IC_IMPL_H */
```

Implementation source file

```
/* -*- c++ -*- */
/*
 * Copyright 2017 <+YOU OR YOUR COMPANY+>.
 *
 * This is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "qam4_ic_impl.h"

namespace gr {
  namespace mynewmod {

    qam4_ic::sptr
    qam4_ic::make()
    {
      return gnuradio::get_initial_sptr
        (new qam4_ic_impl());
    }

    /*
     * The private constructor
     */
    qam4_ic_impl::qam4_ic_impl()
      : gr::block("qam4_ic",
              gr::io_signature::make(1,1, sizeof(int)),
              gr::io_signature::make(1,1, sizeof(gr_complex)))
    {}

    /*
     * Our virtual destructor.
     */
    qam4_ic_impl::~qam4_ic_impl()
    {
    }

    void
    qam4_ic_impl::forecast (int noutput_items, gr_vector_int &ninput_items_required)
    {
        ninput_items_required[0] = noutput_items;
    }

    int
    qam4_ic_impl::general_work (int noutput_items,
                        gr_vector_int &ninput_items,
                        gr_vector_const_void_star &input_items,
                        gr_vector_void_star &output_items)
    {
        const int *in = (const int *) input_items[0];
```

```
        gr_complex *out = (gr_complex *) output_items[0];

        for(int i = 0; i < noutput_items; i++)
        {

                switch(in[i])
                {
                        case 0:
                                out[i] = gr_complex(-1,-1);
                                break;
                        case 1:
                                out[i] = gr_complex(-1,1);
                                break;
                        case 2:
                                out[i] = gr_complex(1,-1);
                                break;
                        case 3:
                                out[i] = gr_complex(1,1);
                                break;
                        default:
                                out[i] = gr_complex(0,0);
                }
        }

        // Do <+signal processing+>
        // Tell runtime system how many input items we consumed on
        // each input stream.
        consume_each (noutput_items);

        // Tell runtime system how many output items we produced.
        return noutput_items;
    }

  } /* namespace mynewmod */
} /* namespace gr */
```

Implementation XML file

```
<?xml version="1.0"?>
<block>
  <name>qam4_ic</name>
  <key>mynewmod_qam4_ic</key>
  <category>mynewmod</category>
  <import>import mynewmod</import>
  <make>mynewmod.qam4_ic()</make>
  <sink>
    <name>in</name>
    <type>int</type>
  </sink>
  <source>
    <name>out</name>
    <type>complex</type>
  </source>
</block>
```

A.2.2 qam16_ic implementation in OOT module

Implementation header file

```
/* -*- c++ -*- */
/*
 * Copyright 2017 <+YOU OR YOUR COMPANY+>.
 *
 * This is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * This software is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#ifndef INCLUDED_MYNEWMOD_QAM16_IC_IMPL_H
#define INCLUDED_MYNEWMOD_QAM16_IC_IMPL_H

#include <mynewmod/qam16_ic.h>

namespace gr {
  namespace mynewmod {

    class qam16_ic_impl : public qam16_ic
    {
     private:
      // Nothing to declare in this block.

     public:
      qam16_ic_impl();
      ~qam16_ic_impl();

      // Where all the action really happens
      void forecast (int noutput_items, gr_vector_int &ninput_items_required);

      int general_work(int noutput_items,
                       gr_vector_int &ninput_items,
                       gr_vector_const_void_star &input_items,
                       gr_vector_void_star &output_items);
    };

  } // namespace mynewmod
} // namespace gr

#endif /* INCLUDED_MYNEWMOD_QAM16_IC_IMPL_H */
```

Implementation source file

```
/* -*- c++ -*- */
/*
 * Copyright 2017 <+YOU OR YOUR COMPANY+>.
 *
 * This is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "qam16_ic_impl.h"

namespace gr {
  namespace mynewmod {

    qam16_ic::sptr
    qam16_ic::make()
```

```
{
  return gnuradio::get_initial_sptr
    (new qam16_ic_impl());
}

/*
 * The private constructor
 */
qam16_ic_impl::qam16_ic_impl()
  : gr::block("qam16_ic",
          gr::io_signature::make(1,1, sizeof(int)),
          gr::io_signature::make(1,1, sizeof(gr_complex)))
{}

/*
 * Our virtual destructor.
 */
qam16_ic_impl::~qam16_ic_impl()
{
}

void
qam16_ic_impl::forecast (int noutput_items, gr_vector_int &ninput_items_required)
{
     ninput_items_required[0] = noutput_items;
}

int
qam16_ic_impl::general_work (int noutput_items,
                   gr_vector_int &ninput_items,
                   gr_vector_const_void_star &input_items,
                   gr_vector_void_star &output_items)
{
    const int *in = (const int *) input_items[0];
    gr_complex *out = (gr_complex *) output_items[0];

    for(int i = 0; i < noutput_items; i++)
    {

            switch(in[i])
            {
                    case 0:
                            out[i] = gr_complex(-3,-3);
                            break;
                    case 1:
                            out[i] = gr_complex(-3,-1);
                            break;
                    case 2:
                            out[i] = gr_complex(-3,3);
                            break;
                    case 3:
                            out[i] = gr_complex(-3,1);
                            break;
                    case 4:
                            out[i] = gr_complex(-1,-3);
                            break;
                    case 5:
                            out[i] = gr_complex(-1,-1);
                            break;
                    case 6:
                            out[i] = gr_complex(-1,3);
                            break;
                    case 7:
                            out[i] = gr_complex(-1,1);
                            break;
                    case 8:
                            out[i] = gr_complex(3,-3);
                            break;
                    case 9:
                            out[i] = gr_complex(3,-1);
                            break;
                    case 10:
                            out[i] = gr_complex(3,3);
```

```
                        break;
                    case 11:
                        out[i] = gr_complex(3,1);
                        break;
                    case 12:
                        out[i] = gr_complex(1,-3);
                        break;
                    case 13:
                        out[i] = gr_complex(1,-1);
                        break;
                    case 14:
                        out[i] = gr_complex(1,3);
                        break;
                    case 15:
                        out[i] = gr_complex(1,1);
                        break;
                    default:
                        out[i] = gr_complex(0,0);
                }
            }

            // Do <+signal processing+>
            // Tell runtime system how many input items we consumed on
            // each input stream.
            consume_each (noutput_items);

            // Tell runtime system how many output items we produced.
            return noutput_items;
        }

    } /* namespace mynewmod */
} /* namespace gr */
```

Implementation XML file

```
<?xml version="1.0"?>
<block>
  <name>qam16_ic</name>
  <key>mynewmod_qam16_ic</key>
  <category>mynewmod</category>
  <import>import mynewmod</import>
  <make>mynewmod.qam16_ic()</make>
  <sink>
    <name>in</name>
    <type>int</type>
  </sink>
  <source>
    <name>out</name>
    <type>complex</type>
  </source>
</block>
```

A.2.3 qam32_ic implementation in OOT module

Implementation header file

```
/* -*- c++ -*- */
/*
 * Copyright 2017 <+YOU OR YOUR COMPANY+>.
 *
 * This is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#ifndef INCLUDED_MYNEWMOD_QAM32_IC_IMPL_H
#define INCLUDED_MYNEWMOD_QAM32_IC_IMPL_H

#include <mynewmod/qam32_ic.h>

namespace gr {
  namespace mynewmod {

    class qam32_ic_impl : public qam32_ic
    {
     private:
      // Nothing to declare in this block.

     public:
      qam32_ic_impl();
      ~qam32_ic_impl();

      // Where all the action really happens
      void forecast (int noutput_items, gr_vector_int &ninput_items_required);

      int general_work(int noutput_items,
                       gr_vector_int &ninput_items,
                       gr_vector_const_void_star &input_items,
                       gr_vector_void_star &output_items);
    };

  } // namespace mynewmod
} // namespace gr

#endif /* INCLUDED_MYNEWMOD_QAM32_IC_IMPL_H */
```

Implementation source file

```
/* -*- c++ -*- */
/*
 * Copyright 2017 <+YOU OR YOUR COMPANY+>.
 *
 * This is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "qam32_ic_impl.h"

namespace gr {
  namespace mynewmod {

    qam32_ic::sptr
    qam32_ic::make()
    {
      return gnuradio::get_initial_sptr
```

```
    (new qam32_ic_impl());
}

/*
 * The private constructor
 */
qam32_ic_impl::qam32_ic_impl()
  : gr::block("qam32_ic",
          gr::io_signature::make(1,1, sizeof(int)),
          gr::io_signature::make(1,1, sizeof(gr_complex)))
{}

/*
 * Our virtual destructor.
 */
qam32_ic_impl::~qam32_ic_impl()
{
}

void
qam32_ic_impl::forecast (int noutput_items, gr_vector_int &ninput_items_required)
{
     ninput_items_required[0] = noutput_items;
}

int
qam32_ic_impl::general_work (int noutput_items,
                    gr_vector_int &ninput_items,
                    gr_vector_const_void_star &input_items,
                    gr_vector_void_star &output_items)
{
    const int *in = (const int *) input_items[0];
    gr_complex *out = (gr_complex *) output_items[0];

    for(int i = 0; i < noutput_items; i++)
    {

            switch(in[i])
            {
                    case 0:
                            out[i] = gr_complex(-3,5);
                            break;
                    case 1:
                            out[i] = gr_complex(-1,5);
                            break;
                    case 2:
                            out[i] = gr_complex(-3,-5);
                            break;
                    case 3:
                            out[i] = gr_complex(-1,-5);
                            break;
                    case 4:
                            out[i] = gr_complex(-5,3);
                            break;
                    case 5:
                            out[i] = gr_complex(-5,1);
                            break;
                    case 6:
                            out[i] = gr_complex(-5,-3);
                            break;
                    case 7:
                            out[i] = gr_complex(-5,-1);
                            break;
                    case 8:
                            out[i] = gr_complex(-1,3);
                            break;
                    case 9:
                            out[i] = gr_complex(-1,1);
                            break;
                    case 10:
                            out[i] = gr_complex(-1,-3);
                            break;
                    case 11:
```

```
                        out[i] = gr_complex(-1,-1);
                        break;
                case 12:
                        out[i] = gr_complex(-3,3);
                        break;
                case 13:
                        out[i] = gr_complex(-3,1);
                        break;
                case 14:
                        out[i] = gr_complex(-3,-3);
                        break;
                case 15:
                        out[i] = gr_complex(-3,-1);
                        break;
                case 16:
                        out[i] = gr_complex(3,5);
                        break;
                case 17:
                        out[i] = gr_complex(1,5);
                        break;
                case 18:
                        out[i] = gr_complex(3,-5);
                        break;
                case 19:
                        out[i] = gr_complex(1,-5);
                        break;
                case 20:
                        out[i] = gr_complex(5,3);
                        break;
                case 21:
                        out[i] = gr_complex(5,1);
                        break;
                case 22:
                        out[i] = gr_complex(5,-3);
                        break;
                case 23:
                        out[i] = gr_complex(5,-1);
                        break;
                case 24:
                        out[i] = gr_complex(1,3);
                        break;
                case 25:
                        out[i] = gr_complex(1,1);
                        break;
                case 26:
                        out[i] = gr_complex(1,-3);
                        break;
                case 27:
                        out[i] = gr_complex(1,-1);
                        break;
                case 28:
                        out[i] = gr_complex(3,3);
                        break;
                case 29:
                        out[i] = gr_complex(3,1);
                        break;
                case 30:
                        out[i] = gr_complex(3,-3);
                        break;
                case 31:
                        out[i] = gr_complex(3,-1);
                        break;
                default:
                        out[i] = gr_complex(0,0);
        }
}

// Do <+signal processing+>
// Tell runtime system how many input items we consumed on
// each input stream.
consume_each (noutput_items);
```

```
        // Tell runtime system how many output items we produced.
        return noutput_items;
    }

  } /* namespace mynewmod */
} /* namespace gr */
```

Implementation XML file

```
<?xml version="1.0"?>
<block>
  <name>qam32_ic</name>
  <key>mynewmod_qam32_ic</key>
  <category>mynewmod</category>
  <import>import mynewmod</import>
  <make>mynewmod.qam32_ic()</make>
  <sink>
    <name>in</name>
    <type>int</type>
  </sink>
  <source>
    <name>out</name>
    <type>complex</type>
  </source>
</block>
```

A.3 TensorFlow Code

A.3.1 Data capture segmentation

```
# For 0dB Samples
import numpy as np
import struct
import os
os.chdir('path_to_location_of_captured_file')

FileClass = ['bpsk', 'qam4', 'qam16', 'qam32',
             'ofdm_bpsk', 'ofdm_qam4', 'ofdm_qam16','ofdm_qam32']
N_list    = [512, 1024, 2048]

for filestr in FileClass:
    FileDir = 'path_to_location_of_captured_file/'+filestr
    os.chdir(FileDir)

    for N in N_list:
        os.chdir(FileDir)
        # load datasets
        #N        = 512  # No. of Input samples per block {N/2 I and Q samples}
        Nfiles   = 1000  # No. of files to play with
        Ndiscard = 2000  # No. of initial samples to discard
        Nblocks  = 20    # No. of blocks per file

        data = np.zeros([N,Nfiles*Nblocks])

        block_no = 0;  # Iterator for blocks of 1000 samples each
        for k in range(Nfiles):
            filename = filestr+'_'+str(k+1)+".dat"
            print(filename)
            fh = open(filename,'rb');
```

```
            # discard M samples
            for i in range(Ndiscard):
                struct.unpack('f', fh.read(4))

        # For Standard Modulations we have 1e6 I and Q samples, so total of 2e6 samples.
        # After discarding 2e3 samples, we are left with 2e6-2e3 samples.
        # Only 200e3 samples are 'unique symbols' which gives us
        # 200 chunks 1e3 samples each.
        # Since that is too much memory to work with. we are
        # only picking 20 blocks with 2e3 samples gaps in between

            for _ in range(Nblocks):
                for i in range(N):
                    (data[i,block_no],) = struct.unpack('f', fh.read(4))

                # discard M samples
                for i in range(Ndiscard):
                    struct.unpack('f', fh.read(4))

                block_no             = block_no + 1;


        fh.close()

        os.chdir('path_to_location_for_saving/array_save')

        outfilename = filestr + '_data_0dB_' + str(int(N/2)) + '.dat';
        #outfilename = filestr + '_data_5dB_' + str(int(N/2)) + '.dat';
        #outfilename = filestr + '_data_10dB_' + str(int(N/2)) + '.dat';
        fh = open(outfilename,'wb');
        np.save(fh, data)
        fh.close()
print('done!')

# To Load the File
##data2 = np.zeros([N,1000]);
##outfilename = 'ofdm_qam32_data_0dB_' + str(N) + '.dat';
##fh = open(outfilename,'rb');
##data2 = np.load(fh)
##fh.close()
##
##print('done!')
```

A.3.2 MLP for single SNR data

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import os
import random

# load datasets
Nfiles  = 1000
N       = 500  # No. of Input samples
Nblocks = 20    # No of block per file
Nmod    = 4
Ntrain  = 900 * Nblocks # No. training data examples
Nval    = 100 * Nblocks # No. validation data examples
NtrainFiles  = 900   # No. of train data files (*20)

os.chdir('path_to_saved_files/original/array_save')

def load_data(filename,N):
    infile = open(filename, 'rb')
    tmp    = np.load(infile)
    tmp    = tmp.T
    data   = tmp[:,0:N*2:2]
    infile.close()
    return(data)

def normalize_data(input):
```

```
    output = (input / max(abs(input)))
    return output

Nstr = 1000
filename = 'bpsk_data_0dB_' + str(Nstr) + '.dat';
bpsk_data = load_data(filename,N)

filename = 'qam4_data_0dB_' + str(Nstr) + '.dat';
qam4_data = load_data(filename,N)

filename = 'qam16_data_0dB_' + str(Nstr) + '.dat';
qam16_data = load_data(filename,N)

filename = 'qam32_data_0dB_' + str(Nstr) + '.dat';
qam32_data = load_data(filename,N)

# Normalize all data (Tried this because I though the input might be too small)
for i in range(Nfiles*Nblocks):
    bpsk_data[i,:]  = normalize_data(bpsk_data[i,:])
    qam4_data[i,:]  = normalize_data(qam4_data[i,:])
    qam16_data[i,:] = normalize_data(qam16_data[i,:])
    qam32_data[i,:] = normalize_data(qam32_data[i,:])

# shuffle array
sm_index = np.arange(0,Ntrain);  # small index for each class
random.shuffle(sm_index)

# training with 900 * 20 = 18000 examples
Ntrain                          = 900 * Nblocks
train_data                      = np.zeros([Ntrain*Nmod,N])
y_train                         = np.zeros([Ntrain*Nmod,Nmod])+0.1 # Ground Truth
train_data[0*Ntrain:1*Ntrain,:] = bpsk_data[sm_index[0:Ntrain],:]
train_data[1*Ntrain:2*Ntrain,:] = qam4_data[sm_index[0:Ntrain],:]
train_data[2*Ntrain:3*Ntrain,:] = qam16_data[sm_index[0:Ntrain],:]
train_data[3*Ntrain:4*Ntrain,:] = qam32_data[sm_index[0:Ntrain],:]
y_train[0*Ntrain:1*Ntrain,0]    = 1
y_train[1*Ntrain:2*Ntrain,1]    = 1
y_train[2*Ntrain:3*Ntrain,2]    = 1
y_train[3*Ntrain:4*Ntrain,3]    = 1

# testing with 100 * 20 = 2000 examples
Nval                          = 100 * Nblocks # validation data
val_data                      = np.zeros([Nval*Nmod,N])
y_val                         = np.zeros([Nval*Nmod,Nmod])+0.1
val_data[0*Nval:1*Nval,:] = bpsk_data[Ntrain:,:]
val_data[1*Nval:2*Nval,:] = qam4_data[Ntrain:,:]
val_data[2*Nval:3*Nval,:] = qam16_data[Ntrain:,:]
val_data[3*Nval:4*Nval,:] = qam32_data[Ntrain:,:]
y_val[0*Nval:1*Nval,0]    = 1
y_val[1*Nval:2*Nval,1]    = 1
y_val[2*Nval:3*Nval,2]    = 1
y_val[3*Nval:4*Nval,3]    = 1

# Randomize training set
big_index = np.arange(0,Nmod*Ntrain);  # big index for all class
random.shuffle(big_index)

train_data[:,:] = train_data[big_index,:]
y_train[:,:]    = y_train[big_index,:]

# model input/output (Placeholders)
x  = tf.placeholder(tf.float32, shape=[None,N])
y_ = tf.placeholder(tf.float32, shape=[None,Nmod])

M = 200  # Frist Layer Output Nodes
L = 4    # Second Layer Output Node
# first layer
W1  = tf.Variable(tf.truncated_normal([N,M], stddev=0.1))
b1  = tf.Variable(tf.constant(0.1, dtype=tf.float32, shape=[M]))
fc1 = tf.nn.relu(tf.matmul(x,W1) + b1)
```

```
# Readout layer
W2 = tf.Variable(tf.truncated_normal([M,L], stddev=0.1))
b2 = tf.Variable(tf.constant(0.1, dtype=tf.float32, shape=[L]))

#model
y = (tf.matmul(fc1,W2)+b2) #linear regression model
# loss function
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_,
                                                                       logits=y))

# Training Method
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

# check model performance
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# Session
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())

Nbatch = 100
Nrange = int((Ntrain*Nmod)/Nbatch)
acc  = np.zeros(int(Nrange/5))
xent = np.zeros(int(Nrange/5))
out  = np.zeros([Nbatch,Nmod])

# Train
k      = 0;
for i in range(720):
    train_step_data = feed_dict={x:train_data[i*Nbatch:(i*Nbatch)+Nbatch,:],
                                 y_:y_train[i*Nbatch:(i*Nbatch)+Nbatch,:]}
    sess.run(train_step, train_step_data)
    # check progress
    if i%5 == 0:
        wo,yo,a,c = sess.run([W1,y,accuracy, cross_entropy], train_step_data)
        wout      = wo
        out       = yo
        acc[k]    = a
        xent[k]   = c
        k         = k + 1;

plt.figure(1)
plt.plot(acc, label="acc")
plt.title('Accuracy')
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

plt.figure(2)
plt.plot(xent, label="cross entropy")
plt.title('Cross Entropy')
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

print(accuracy.eval(feed_dict={x:val_data, y_:y_val}))
sess.close()
```

A.3.3 CNN for single SNR data

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import os
import random

# Select
modClassStandard = ['bpsk', 'qam4', 'qam16', 'qam32']
modClassOfdm     = ['ofdm_bpsk', 'ofdm_qam4', 'ofdm_qam16', 'ofdm_qam32']
modClassAll      = modClassStandard + modClassOfdm
Nlist            = [256, 512, 1024] # Ex: 256 I and Q samples (total samples 256 * 2)

#FileDir = 'D:/Justin/Documents/Thesis/Data_Acq/captures/original/array_save'
#FileDir = 'D:/Justin/Documents/Thesis/Data_Acq/captures/plus5/array_save'
FileDir = 'D:/Justin/Documents/Thesis/Data_Acq/captures/plus10/array_save'
os.chdir(FileDir)
```

```
Classtype = modClassAll
N         = Nlist[2]

# Training & Testing size
Nfiles  = 1000 # Fixed
Nmod    = len(Classtype)    # No. of classes (modulations)
Nblocks = 20   # No of block per file
Ntrain  = 800 * Nblocks # No. training data examples
Nval    = 100 * Nblocks # No. validation data examples
Ntest   = 100 * Nblocks # No. validation data examples

# Function Definitions
def weight_variable(shape, name):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial, name=name)

def bias_variable(shape, name):
    initial = tf.constant(0.1, shape=shape)  # Positive bias because we will be using RelU
    return tf.Variable(initial, name=name)

def conv2d(x,W,name):
    return tf.nn.conv2d(x, W, strides=[1,1,1,1], padding='SAME', name=name)

def max_pool_2x2(x, name):
    return tf.nn.max_pool(x, ksize=[1,2,2,1], strides=[1,2,2,1],
                          padding='SAME', name=name)

def max_pool_1x2(x, name):
    return tf.nn.max_pool(x, ksize=[1,1,2,1], strides=[1,1,2,1],
                          padding='SAME', name=name)

def load_data(filename,N):
    infile = open(filename, 'rb')
    tmp    = np.load(infile)
    tmp    = tmp.T
    data   = tmp[:,0:N]  # I and Q interleaved
    infile.close()
    return(data)

def tensor_reshape(x):
    K = int(x.shape[0])
    N = int(x.shape[1])
    y = np.zeros([K, 2, int(N/2)])
    for i in range(K):
        y[i,0,:] = x[i,0:N:2]
        y[i,1,:] = x[i,1:N:2]
    return(y)

def randomize_data(data, index):
    tmp = np.zeros(data.shape)
    for k in range(data.shape[0]):
        tmp[k,:] = data[index[k],:]
    return(tmp)


# load datasets
allData = np.zeros([Nmod, Nfiles*Nblocks, 2, N])

# Randomize indexes
sm_index = np.arange(0,Nfiles*Nblocks);  # small index for each class
random.shuffle(sm_index)

Nstr = N
i    = 0
for fileclass in Classtype:
    #filename = fileclass + '_data_0dB_' + str(Nstr) + '.dat';
    #filename = fileclass + '_data_5dB_' + str(Nstr) + '.dat';
    filename = fileclass + '_data_10dB_' + str(Nstr) + '.dat';
    tmp      = load_data(filename,N*2)
    tmp      = tensor_reshape(tmp)
    tmp      = randomize_data(tmp, sm_index)
    allData[i,:] = tmp
```

```
    i = i + 1

train_data                          = np.zeros([Nmod*Ntrain,2,int(N)])
y_train                             = np.zeros([Nmod*Ntrain, Nmod]) # Ground Truth
for i in range(Nmod):
    train_data[i*Ntrain:(i+1)*Ntrain,:]  = allData[i,0:Ntrain,:];
    y_train[i*Ntrain:(i+1)*Ntrain,i]     = 1;

val_data                        = np.zeros([Nmod*Nval,2,int(N)])
y_val                           = np.zeros([Nmod*Nval, Nmod]) # Ground Truth
for i in range(Nmod):
    val_data[i*Nval:(i+1)*Nval,:]  = allData[i,Ntrain:Ntrain+Nval,:];
    y_val[i*Nval:(i+1)*Nval,i]     = 1;

test_data                        = np.zeros([Nmod*Ntest,2,int(N)])
y_test                           = np.zeros([Nmod*Ntest, Nmod]) # Ground Truth
for i in range(Nmod):
    test_data[i*Ntest:(i+1)*Ntest,:]  = allData[i,Ntrain+Nval:,:];
    y_test[i*Ntest:(i+1)*Ntest,i]     = 1;


# Randomize training set
big_index = np.arange(0,Nmod*Ntrain);  # big index for all class
random.shuffle(big_index)

train_data[:,:] = train_data[big_index,:]
y_train[:,:]    = y_train[big_index,:]

# Randomize validation set
big_index = np.arange(0,Nmod*Nval);  # big index for all class
random.shuffle(big_index)

val_data[:,:] = val_data[big_index,:]
y_val[:,:]    = y_val[big_index,:]


# Randomize test set because of batch normalization (only needed for extreme case)
big_index = np.arange(0,Nmod*Ntest);  # big index for all class
random.shuffle(big_index)
test_data[:,:] = test_data[big_index,:]
y_test[:,:]    = y_test[big_index,:]

################################################################################
################################################################################

# Model 1
Nfc     = 256  # Readout layer
Ncv1    = 16
Ncv2    = 32
lr_start = 0.001

# Model 2
Nfc     = 1024  # Readout layer
Ncv1    = 32
Ncv2    = 64
lr_start = 0.005
epsilon = 1e-3  # Batch Normalization Denominator Constant

# model input/output (Placeholders)
x       = tf.placeholder(tf.float32, shape=[None,2, int(N)])
y_      = tf.placeholder(tf.float32, shape=[None,Nmod])
x_image = tf.reshape(x, [-1, 2, int(N), 1])

# First Layer (Conv -- ReLU -- Max Pooling)
with tf.name_scope("layer1"):
    W_conv1 = weight_variable([2,5,1,Ncv1], "W_conv1")
    b_conv1 = bias_variable([Ncv1], "b_conv1")
    y1      = conv2d(x_image, W_conv1, "h_conv1") + b_conv1
    batch_m1, batch_v1 = tf.nn.moments(y1, [0,1,2])
    beta1   = tf.Variable(tf.zeros([Ncv1]))
    y1_hat  = (y1-batch_m1)/tf.sqrt(batch_v1+epsilon)
    y1_BN   = y1_hat + beta1
    h_conv1 = tf.nn.relu(y1_BN)
    h_pool1 = max_pool_2x2(h_conv1, "h_pool1")
```

```
# Second Layer (Conv -- ReLU -- Max Pooling)
with tf.name_scope("layer2"):
    W_conv2 = weight_variable([1,5,Ncv1,Ncv2], "W_conv2") # Minimal Activations
    b_conv2 = bias_variable([Ncv2], "b_conv2")
    y2      = conv2d(h_pool1, W_conv2, "h_conv2") + b_conv2
    batch_m2, batch_v2 = tf.nn.moments(y2, [0,1,2])
    y2_hat  = (y2-batch_m2)/tf.sqrt(batch_v2+epsilon)
    beta2   = tf.Variable(tf.zeros([Ncv2]))
    y2_BN   = y2_hat + beta2
    h_conv2 = tf.nn.relu(y2_BN)
    h_pool2 = max_pool_1x2(h_conv2, "h_pool2")

# Fully connected layer
with tf.name_scope("fc1"):
    W_fc1   = weight_variable([1*int(N/4)*Ncv2,Nfc], "W_fc1")
    b_fc1   = bias_variable([Nfc], "b_fc1")
    h_pool2_flat = tf.reshape(h_pool2, [-1, 1*int(N/4)*Ncv2])
    y3      = tf.matmul(h_pool2_flat, W_fc1) + b_fc1
    batch_m3, batch_v3 = tf.nn.moments(y3, [0])
    y3_hat  = (y3-batch_m3)/tf.sqrt(batch_v3+epsilon)
    beta3   = tf.Variable(tf.zeros([Nfc]))
    y3_BN   = y3_hat + beta3
    h_fc1   = tf.nn.relu(y3_BN)

    # Dropout
    keep_prob  = tf.placeholder(tf.float32)
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# Readout Layer
with tf.name_scope("output"):
    w_fc3  = weight_variable([Nfc, Nmod], "w_fc3")
    b_fc3  = bias_variable([Nmod], "b_fc3")
    y_conv = tf.matmul(h_fc1_drop, w_fc3) + b_fc3

# Loss Function
with tf.name_scope("Xent"):
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_,
                                                                           logits=y_conv))
# Training Method
with tf.name_scope("Train"):
    step = tf.Variable(0, trainable=False)
    rate = tf.train.exponential_decay(lr_start, step, 1, 0.9999)
    train_step = tf.train.AdamOptimizer(rate).minimize(cross_entropy, global_step=step)

# check model performance
with tf.name_scope("Accuracy"):
    correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

##############################################################################
##############################################################################

Nepoch      = 5
Nbatch      = 50
Nobs        = 20
Nrange      = int((Ntrain*Nmod)/Nbatch)
acc_train   = np.zeros(int(Nepoch*Nrange/Nobs))
acc_val     = np.zeros(int(Nepoch*Nrange/Nobs))
xent_train  = np.zeros(int(Nepoch*Nrange/Nobs))
xent_val    = np.zeros(int(Nepoch*Nrange/Nobs))

# Training Session
init = tf.global_variables_initializer()
sess = tf.InteractiveSession()
sess.run(init)
writer = tf.summary.FileWriter("C:/tmp/conv_arch1", sess.graph) # for 0.8
merged = tf.summary.merge_all()
acc_val_tmp = np.zeros([int(val_data.shape[0]/4000)])
xent_val_tmp = np.zeros([int(val_data.shape[0]/4000)])

# Training & Validation
```

```
k       = 0;
training_itr = 0;
for m in range(Nepoch):
    for i in range(Nrange):
        train_step_data = feed_dict={x:train_data[i*Nbatch:(i*Nbatch)+Nbatch,:],
                                     y_:y_train[i*Nbatch:(i*Nbatch)+Nbatch,:],
                                     keep_prob: 0.5}
        sess.run(train_step, train_step_data)

        # Check progress validation
        if training_itr%Nobs == 0:
            a_train,c_train  = sess.run([accuracy, cross_entropy], train_step_data)
            if not ((Nmod==8) and (Classtype == modClassAll) and (N==1024)):
                for p in range(int(val_data.shape[0]/4000)):
                    acc_data= feed_dict={x:val_data[p*4000:4000*(p+1),:],
                                         y_:y_val[p*4000:4000*(p+1),:],
                                         keep_prob:1.0}
                    a_val,c_val      = sess.run([accuracy, cross_entropy], acc_data)
                    acc_val_tmp[p]   = a_val;
                    xent_val_tmp[p]  = c_val;
                acc_val[k]       = np.mean(acc_val_tmp)
                xent_val[k]      = np.mean(xent_val_tmp)
                acc_train[k]     = a_train
                xent_train[k]    = c_train
                print("[Epoch=" + str(m) + ", Itr=" + str(i) + "] train acc:  " +
                      str(a_train) + " | val acc = " + str(a_val))
            else:
                print("[Epoch=" + str(m) + ", Itr=" + str(i) + "] train acc:  " +
                      str(a_train))
            k                = k + 1;

        training_itr = training_itr+1

# Do testing in blocks (memory issue)
Nmax = test_data.shape[0]
Nsize  = 1000
Nrange = int(Nmax/Nsize)

test_acc = np.zeros([Nrange,1])
import pandas as pd
confusion = np.zeros([Nmod,Nmod],int)

for i in range(Nrange):
    test_acc[i] = accuracy.eval(feed_dict={x:test_data[i*Nsize:Nsize*(i+1),:],
                                           y_:y_test[i*Nsize:Nsize*(i+1),:],
                                           keep_prob:1.0})

    # Confusion Matrix
    res = tf.stack([tf.argmax(y_conv,1), tf.argmax(y_,1)])
    ans = res.eval(feed_dict={x:test_data[i*Nsize:Nsize*(i+1),:],
                              y_:y_test[i*Nsize:Nsize*(i+1),:],
                              keep_prob:1.0})
    for p in ans.T:
        confusion[p[0],p[1]]+=1

print(np.mean(test_acc))
print(pd.DataFrame(confusion))
print(pd.DataFrame(confusion/int(test_data.shape[0]/Nmod)))


# Figures
plt.figure(1)
plt.plot(acc_train, label="acc_train")
plt.plot(acc_val, label="acc_val")
plt.title('Accuracy')
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

plt.figure(2)
plt.plot(xent_train, label="train cross entropy")
plt.plot(xent_val, label="val cross entropy")
plt.title('Cross Entropy')
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

sess.close()
```

A.3.4 CNN for multiple SNR data

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import os
import random

# Select
modClassStandard = ['bpsk', 'qam4', 'qam16', 'qam32']
modClassOfdm     = ['ofdm_bpsk', 'ofdm_qam4', 'ofdm_qam16', 'ofdm_qam32']
modClassAll      = modClassStandard + modClassOfdm
Nlist            = [256, 512, 1024] # Ex: 256 I and Q samples (total samples 256 * 2)

FileDir0 = 'D:/Justin/Documents/Thesis/Data_Acq/captures/original/array_save'
FileDir5 = 'D:/Justin/Documents/Thesis/Data_Acq/captures/plus5/array_save'
FileDir10 = 'D:/Justin/Documents/Thesis/Data_Acq/captures/plus10/array_save'

Classtype = modClassOfdm
N         = Nlist[0]

# Training & Testing size
Nfiles  = 1000 # Fixed
Nmod    = len(Classtype)    # No. of classes (modulations)
Nblocks = 20   # No of block per file
Ntrain  = 800 * Nblocks # No. training data examples
Nval    = 100 * Nblocks # No. validation data examples
Ntest   = 100 * Nblocks  # No. validation data examples

# Function Definitions
def weight_variable(shape, name):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial, name=name)

def bias_variable(shape, name):
    initial = tf.constant(0.1, shape=shape)  # Positive bias because we will be using RelU
    return tf.Variable(initial, name=name)

def conv2d(x,W,name):
    return tf.nn.conv2d(x, W, strides=[1,1,1,1], padding='SAME', name=name)

def max_pool_2x2(x, name):
    return tf.nn.max_pool(x, ksize=[1,2,2,1], strides=[1,2,2,1],
                          padding='SAME', name=name)

def max_pool_1x2(x, name):
    return tf.nn.max_pool(x, ksize=[1,1,2,1], strides=[1,1,2,1],
                          padding='SAME', name=name)

def load_data(filename,N):
    infile = open(filename, 'rb')
    tmp    = np.load(infile)
    tmp    = tmp.T
    data   = tmp[:,0:N]  # I and Q interleaved
    infile.close()
    return(data)

def tensor_reshape(x):
    K = int(x.shape[0])
    N = int(x.shape[1])
    y = np.zeros([K, 2, int(N/2)])
    for i in range(K):
        y[i,0,:] = x[i,0:N:2]
        y[i,1,:] = x[i,1:N:2]
    return(y)

def randomize_data(data, index):
    tmp = np.zeros(data.shape)
    for k in range(data.shape[0]):
        tmp[k,:] = data[index[k],:]
    return(tmp)

def randomize_data2(data, index):
    tmp = np.zeros(data.shape)
    for k in range(data.shape[1]):
```

```
        tmp[:,k,:] = data[:,index[k],:]
    return(tmp)

def normalize_data(input):
    output = np.zeros(input.shape)
    for i in range(input.shape[0]):
        output[i,:] = (input[i,:] / max(abs(input[i,:])))
    return output


# load datasets
allData = np.zeros([Nmod, Nfiles*Nblocks*3, 2, N])

# Randomize indexes
sm_index = np.arange(0,Nfiles*Nblocks);  # small index for each class
random.shuffle(sm_index)

Nstr = N
i    = 0
for fileclass in Classtype:
    os.chdir(FileDir0)
    filename = fileclass + '_data_0dB_' + str(Nstr) + '.dat';
    tmp      = load_data(filename,N*2)
    tmp      = tensor_reshape(tmp)
    tmp      = randomize_data(tmp, sm_index)
    allData[i,0:Nfiles*Nblocks] = tmp
    os.chdir(FileDir5)
    filename = fileclass + '_data_5dB_' + str(Nstr) + '.dat';
    tmp      = load_data(filename,N*2)
    tmp      = tensor_reshape(tmp)
    tmp      = randomize_data(tmp, sm_index)
    allData[i,1*Nfiles*Nblocks:2*Nfiles*Nblocks] = tmp
    os.chdir(FileDir10)
    filename = fileclass + '_data_10dB_' + str(Nstr) + '.dat';
    tmp      = load_data(filename,N*2)
    tmp      = tensor_reshape(tmp)
    tmp      = randomize_data(tmp, sm_index)
    allData[i,2*Nfiles*Nblocks:3*Nfiles*Nblocks] = tmp
    i = i + 1

# Randomize the big array
big_array_index = np.arange(0, allData.shape[1])
random.shuffle(big_array_index)

allData = randomize_data2(allData, big_array_index)

train_data                      = np.zeros([Nmod*Ntrain,2,int(N)])
y_train                         = np.zeros([Nmod*Ntrain, Nmod]) # Ground Truth
for i in range(Nmod):
    train_data[i*Ntrain:(i+1)*Ntrain,:]  = allData[i,0:Ntrain,:];
    y_train[i*Ntrain:(i+1)*Ntrain,i]     = 1;

val_data                    = np.zeros([Nmod*Nval,2,int(N)])
y_val                       = np.zeros([Nmod*Nval, Nmod]) # Ground Truth
for i in range(Nmod):
    val_data[i*Nval:(i+1)*Nval,:]  = allData[i,Ntrain:Ntrain+Nval,:];
    y_val[i*Nval:(i+1)*Nval,i]     = 1;

test_data                     = np.zeros([Nmod*Ntest,2,int(N)])
y_test                        = np.zeros([Nmod*Ntest, Nmod]) # Ground Truth
for i in range(Nmod):
    test_data[i*Ntest:(i+1)*Ntest,:]  = allData[i,Ntrain+Nval:Ntrain+Nval+Ntest,:];
    y_test[i*Ntest:(i+1)*Ntest,i]     = 1;


# Randomize training set
big_index = np.arange(0,Nmod*Ntrain);  # big index for all class
random.shuffle(big_index)

train_data[:,:] = train_data[big_index,:]
y_train[:,:]    = y_train[big_index,:]

# Randomize validation set
big_index = np.arange(0,Nmod*Nval);  # big index for all class
```

```
random.shuffle(big_index)

val_data[:,:] = val_data[big_index,:]
y_val[:,:]    = y_val[big_index,:]

# Randomize test set because of batch normalization (only needed for extreme case)
big_index = np.arange(0,Nmod*Ntest);  # big index for all class
random.shuffle(big_index)
test_data[:,:] = test_data[big_index,:]
y_test[:,:]    = y_test[big_index,:]

##############################################################################
##############################################################################

# Model 1
Nfc     = 256  # Readout layer
Ncv1    = 16
Ncv2    = 32
lr_start = 0.001
# Model 2
Nfc     = 1024  # Readout layer
Ncv1    = 32
Ncv2    = 64
epsilon = 1e-3  # Batch Normalization Denominator Constant
lr_start = 0.005
# model input/output (Placeholders)
x       = tf.placeholder(tf.float32, shape=[None,2, int(N)])
y_      = tf.placeholder(tf.float32, shape=[None,Nmod])
x_image = tf.reshape(x, [-1, 2, int(N), 1])

# First Layer (Conv -- ReLU -- Max Pooling)
with tf.name_scope("layer1"):
    W_conv1 = weight_variable([2,5,1,Ncv1], "W_conv1")
    b_conv1 = bias_variable([Ncv1], "b_conv1")
    y1      = conv2d(x_image, W_conv1, "h_conv1") + b_conv1
    batch_m1, batch_v1 = tf.nn.moments(y1, [0,1,2])
    beta1   = tf.Variable(tf.zeros([Ncv1]))
    y1_hat  = (y1-batch_m1)/tf.sqrt(batch_v1+epsilon)
    y1_BN   = y1_hat + beta1
    h_conv1 = tf.nn.relu(y1_BN)
    h_pool1 = max_pool_2x2(h_conv1, "h_pool1")

# Second Layer (Conv -- ReLU -- Max Pooling)
with tf.name_scope("layer2"):
    W_conv2 = weight_variable([1,5,Ncv1,Ncv2], "W_conv2") # Minimal Activations
    b_conv2 = bias_variable([Ncv2], "b_conv2")
    y2      = conv2d(h_pool1, W_conv2, "h_conv2") + b_conv2
    batch_m2, batch_v2 = tf.nn.moments(y2, [0,1,2])
    y2_hat  = (y2-batch_m2)/tf.sqrt(batch_v2+epsilon)
    beta2   = tf.Variable(tf.zeros([Ncv2]))
    y2_BN   = y2_hat + beta2
    h_conv2 = tf.nn.relu(y2_BN)
    h_pool2 = max_pool_1x2(h_conv2, "h_pool2")

# Fully connected layer
with tf.name_scope("fc1"):
    W_fc1   = weight_variable([1*int(N/4)*Ncv2,Nfc], "W_fc1")
    b_fc1   = bias_variable([Nfc], "b_fc1")
    h_pool2_flat = tf.reshape(h_pool2, [-1, 1*int(N/4)*Ncv2])
    y3      = tf.matmul(h_pool2_flat, W_fc1) + b_fc1
    batch_m3, batch_v3 = tf.nn.moments(y3, [0])
    y3_hat  = (y3-batch_m3)/tf.sqrt(batch_v3+epsilon)
    beta3   = tf.Variable(tf.zeros([Nfc]))
    y3_BN   = y3_hat + beta3
    h_fc1   = tf.nn.relu(y3_BN)

    # Dropout
    keep_prob  = tf.placeholder(tf.float32)
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# Readout Layer
with tf.name_scope("output"):
```

```
    w_fc3  = weight_variable([Nfc, Nmod], "w_fc3")
    b_fc3  = bias_variable([Nmod], "b_fc3")
    y_conv = tf.matmul(h_fc1_drop, w_fc3) + b_fc3

# Loss Function
with tf.name_scope("Xent"):
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_,
                                                                           logits=y_conv))

# Training Method
with tf.name_scope("Train"):
    step = tf.Variable(0, trainable=False)
    rate = tf.train.exponential_decay(lr_start, step, 1, 0.9999)
    train_step = tf.train.AdamOptimizer(rate).minimize(cross_entropy, global_step=step)

# check model performance
with tf.name_scope("Accuracy"):
    correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

################################################################################
################################################################################

Nepoch      = 5
Nbatch      = 100
Nobs        = 20
Nrange      = int((Ntrain*Nmod)/Nbatch)
acc_train   = np.zeros(int(Nepoch*Nrange/Nobs))
acc_val     = np.zeros(int(Nepoch*Nrange/Nobs))
xent_train  = np.zeros(int(Nepoch*Nrange/Nobs))
xent_val    = np.zeros(int(Nepoch*Nrange/Nobs))

# Training Session
init = tf.global_variables_initializer()
sess = tf.InteractiveSession()
sess.run(init)
writer = tf.summary.FileWriter("C:/tmp/conv_arch1", sess.graph) # for 0.8
merged = tf.summary.merge_all()
acc_val_tmp = np.zeros([int(val_data.shape[0]/4000)])
xent_val_tmp = np.zeros([int(val_data.shape[0]/4000)])
# Training & Validation
k       = 0;
training_itr = 0;
for m in range(Nepoch):
    for i in range(Nrange):
        train_step_data = feed_dict={x:train_data[i*Nbatch:(i*Nbatch)+Nbatch,:],
                                     y_:y_train[i*Nbatch:(i*Nbatch)+Nbatch,:],
                                     keep_prob: 0.5}
        sess.run(train_step, train_step_data)

        # Check progress validation
        if training_itr%Nobs == 0:
            a_train,c_train  = sess.run([accuracy, cross_entropy], train_step_data)
            if not ((Nmod==8) and (Classtype == modClassAll) and (N==1024)):
                for p in range(int(val_data.shape[0]/4000)):
                    a_val,c_val      = sess.run([accuracy, cross_entropy],
                                                feed_dict={x:val_data[p*4000:4000*(p+1),:],
                                                           y_:y_val[p*4000:4000*(p+1),:],
                                                           keep_prob:1.0})
                    acc_val_tmp[p]   = a_val;
                    xent_val_tmp[p]  = c_val;
                acc_val[k]       = np.mean(acc_val_tmp)
                xent_val[k]      = np.mean(xent_val_tmp)
                acc_train[k]     = a_train
                xent_train[k]    = c_train
                print("[Epoch=" + str(m) + ", Itr=" + str(i) + "] train acc:  "
                      + str(a_train) + "  | val acc = " + str(a_val))
            else:
                print("[Epoch=" + str(m) + ", Itr=" + str(i) + "] train acc:  "
                      + str(a_train))
            k                = k + 1;
```

```
        training_itr = training_itr+1

# Testing Session
# Do it in blocks of 4000
Nmax = test_data.shape[0]
Nrange = int(Nmax/4000)

test_acc = np.zeros([Nrange,1])
import pandas as pd
confusion = np.zeros([Nmod,Nmod],int)

for i in range(Nrange):
    test_acc[i] = accuracy.eval(feed_dict={x:test_data[i*4000:4000*(i+1),:],
                                           y_:y_test[i*4000:4000*(i+1),:],
                                           keep_prob:1.0})

    # Confusion Matrix
    res = tf.stack([tf.argmax(y_conv,1), tf.argmax(y_,1)])
    ans = res.eval(feed_dict={x:test_data[i*4000:4000*(i+1),:],
                              y_:y_test[i*4000:4000*(i+1),:],
                              keep_prob:1.0})
    for p in ans.T:
        confusion[p[0],p[1]]+=1

print(np.mean(test_acc))
print(pd.DataFrame(confusion))
print(pd.DataFrame(confusion/int(test_data.shape[0]/Nmod)))

# Figures
plt.figure(1)
plt.plot(acc_train, label="acc_train")
plt.plot(acc_val, label="acc_val")
plt.title('Accuracy')
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

plt.figure(2)
plt.plot(xent_train, label="train cross entropy")
plt.plot(xent_val, label="val cross entropy")
plt.title('Cross Entropy')
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

sess.close()
```

Bibliography

[1] Christopher M. Bishop. “Pattern Recognition and Machine Learning”. In: Springer, 2006. Chap. 7.1.

[2] Abdelaali Chaoub. *Cognitive Radio - Underlay, Overlay or Interweave: How the spectrum is shared?* URL: http://cognitive-radio-networks.blogspot.com/2014/05/cognitive-radio-underlay-overlay-or.html.

[3] Cisco:Omni Antenna vs. Directional Antenna. *Multipath Fading example*. URL: http://www.cisco.com/c/en/us/support/docs/wireless-mobility/wireless-lan-wlan/82068-omni-vs-direct.html.

[4] Han Gang, Li Jiandong, and Lu Donghua. “Study of modulation recognition based on HOCs and SVM”. In: *2004 IEEE 59th Vehicular Technology Conference. VTC 2004-Spring (IEEE Cat. No.04CH37514)*. Vol. 2. May 2004, 898–902 Vol.2. DOI: 10.1109/VETECS.2004.1388960.

[5] Andrea Goldsmith. “Wireless Communications”. In: Cambridge University Press, 2012. Chap. 12.4.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[7] IEEE. *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications; High-speed Physical Layer in the 5 GHz Band*. URL: http://standards.ieee.org/about/get/802/802.11.html.

[8] National Instrument. *QAM Constellation Diagrams*. URL: https://awrcorp.com/download/faq/english/docs/VSS_System_Blocks/QAM_MAP.htm.

[9] A. R. Khedkar and P. Admane. “Estimation and reduction of CFO in OFDM system”. In: *2015 International Conference on Information Processing (ICIP)*. Dec. 2015, pp. 130–134. DOI: 10.1109/INFOP.2015.7489364.

[10] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2014). URL: http://arxiv.org/abs/1412.6980.

[11] Financial Planning Body of Knowledge. *Communication system model*. URL: http://financialplanningbodyofknowledge.com/wiki/Types_of_structured_communication.

[12] Hong Li et al. “OFDM Modulation Classification and Parameters Extraction”. In: *1st International Conference on Cognitive Radio Oriented Wireless Networks and Communications* (2006), pp. 1–6. DOI: 10.1109/CROWNCOM.2006.363474.

[13] T. C. Lin and S. M. Phoong. “A New Cyclic-Prefix Based Algorithm for Blind CFO Estimation in OFDM Systems”. In: *IEEE Transactions on Wireless Communications* 15.6 (June 2016), pp. 3995–4008. ISSN: 1536-1276. DOI: 10.1109/TWC.2016.2532325.

[14] Y. Ma et al. “Design of OFDM Timing Synchronization Based on Correlations of Preamble Symbol”. In: *2016 IEEE 83rd Vehicular Technology Conference (VTC Spring)*. May 2016, pp. 1–5. DOI: 10.1109/VTCSpring.2016.7504190.

[15] Mathuranathan. *OFDM Transmitter and Receiver Model*. URL: http://www.gaussianwaves.com/2011/07/simulation-of-ofdm-system-in-matlab-ber-vs-ebn0-for-ofdm-in-awgn-channel/.

[16] Society for Neuroscience. *The Neuron*. URL: http://www.brainfacts.org/brain-basics/neuroanatomy/articles/2012/the-neuron/.

[17] OpenCV. *Introduction to Support Vector Machines*. URL: http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html.

[18] Vojtech Pavlovsky. *Introduction to Convolutional Neural Networks*. URL: https://www.vaetas.cz/blog/intro-convolutional-neural-networks/.

[19] Peter. *Pattern Recognition Toolbox*. URL: https://www.mathworks.com/matlabcentral/fileexchange/46392-pattern-recognition-toolbox.

[20] Frank Rayal. *OFDM Sub Carriers*. URL: https://www.wirelessdesignmag.com/blog/2012/09/how-non-line-sight-backhaul-really-works.

[21] Ettus Research. *UHD:rx samples to file.cpp*. URL: https://github.com/EttusResearch/uhd/blob/master/host/examples/rx_samples_to_file.cpp.

[22] Ettus Research. *USRP Hardware Driver and USRP Manual*. URL: https://files.ettus.com/manual/.

[23] Ettus Research. *USRP2 N210 product page*. URL: https://www.ettus.com/product/details/UN210-KIT.

[24] Michael Rice. *Digital Communications: A Discreet Time Approach*. Prentice Hall, 2009.

[25] B. Shamla and K. G. Gayathri Devi. "Design and implementation of Costas loop for BPSK demodulator". In: *2012 Annual IEEE India Conference (INDICON)*. Dec. 2012, pp. 785–789. DOI: 10.1109/INDCON.2012.6420723.

[26] T. Charles Clancy Timothy J O'Shea Johnathan Corgan. "Convolutional Radio Modulation Recognition Networks". In: *arXiv preprint arXiv:1602.04105* (2016), pp. 1–6.

[27] Various. *Software-defined radio*. URL: https://en.wikipedia.org/wiki/Software-defined_radio.

[28] CS231n Convolutional Neural Networks for Visual Recognition. *Max Pool*. URL: http://cs231n.github.io/convolutional-networks/.

[29] Wikipedia. *BPSK Constellation Diagrams*. URL: https://commons.wikimedia.org/wiki/File:BPSK_Gray_Coded_WMC.png.

[30] Wikipedia. *Carrier Recovery*. URL: https://en.wikipedia.org/wiki/Carrier_recovery.

[31] Winlab. *ORBIT-LAB*. URL: http://www.orbit-lab.org/.