Training Data Generation Framework For Machine-Learning Based Classifiers

by

Kyle W. McClintick

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

December 2018

APPROVED:

Professor Alexander Wyglinski, Major Advisor

Professor Wickramarathne Thanuka

Professor Donald R. Brown III

Abstract

In this thesis, we propose a new framework for the generation of training data for machine learning techniques used for classification in communications applications.

Machine learning-based signal classifiers do not generalize well when training data does not describe the underlying probability distribution of real signals. The simplest way to accomplish statistical similarity between training and testing data is to synthesize training data passed through a permutation of plausible forms of noise. To accomplish this, a framework is proposed that implements arbitrary channel conditions and baseband signals. A dataset generated using the framework is considered, and is shown to be appropriately sized by having 11% lower entropy than state-of-the-art datasets.

Furthermore, unsupervised domain adaptation can allow for powerful generalized training via deep feature transforms on unlabeled evaluation-time signals. A novel Deep Reconstruction-Classification Network (DRCN) application is introduced, which attempts to maintain near-peak signal classification accuracy despite dataset bias, or perturbations on testing data unforeseen in training.

Together, feature transforms and diverse training data generated from the proposed framework, teaching a range of plausible noise, can train a deep neural net to classify signals well in many real-world scenarios despite unforeseen perturbations.

Acknowledgements

I would like to express my deepest gratitude to my advisor Professor Alexander Wyglinski for his continuous guidance and support towards my degree. I am very thankful for the opportunity to work with him in the Wireless Innovation Laboratory at Worcester Polytechnic Institute.

I want to thank Professor Wickramarathne Thanuka and Professor Donald Brown for serving on my committee and providing valuable suggestions and comments with regards to my thesis.

I would like to thank James Kingsley, scientific computing system specialist, and Spencer Pruitt, computational scientist from the WPI Academic and Research Computing group at Worcester Polytechnic Institute. Consulting and GPU cluster computing support from the WPI Academic and Research Computing Group contributed to results reported within this thesis.

I would also like to thank my Wilab team members Dr. Srikanth Pagadarai, Kuldeep, Renato, and Nivetha for their immense support during my graduate studies. I would like to thank my friends abroad and in the states who have stayed in contact and given me the support I need, including my room-mates from Terre Haute. I would like to thank good beer, catchy music, and sunny weekends. Finally, I'm thankful for my lovely girlfriend Zhijie, and my warm family: Colin, Dawn, and George.

Contents

List of Figures

List of Tables

List of Acronyms

AC Alternating Current

ACM Adaptive Coding and Modulation

ADC Analog to Digital Converter

AI Artificial Intelligence

AWGN Additive White Gaussian Noise

BER Bit Error Rate

BPF Band-Pass Filter

BPSK Binary Phase Shift Keying

BW Band Width

CDF Cumulative Density Function

CFO Carrier Frequency Offset

CLDNN Convolutional Long Short Term Deep Neural Network

CLT Central Limit Theorem

CMBR Cosmic Microwave Background Radiation

CNN Convolutional Neural Network

CPU Central Processing Unit

DAC Digital to Analog Converter

DC Direct Current

DPD Digital Pre-Distortion

DRCN Deep Reconstruction-Classification Network

DUT Dataset Under Test

ECC Error Control Coding

FET Field Effect Transistor

GAN Generative Adversarial Networks

GBN Ghost Batch Normalization

GPU Graphics Processing Unit

GRC GNU Radio Companion

IF Intermediate Frequency

IIR Infinite Impulse Response

IQ In-phase Quadriture

KLD Kullback-Leibler Divergence

LNA Low-Noise Amplifier

LO Local Oscillator

LOS Line Of Sight

LSTM Long Short Term Memory

MAC Medium Access Control

ML Machine Learning

MIMO Multiple Input Multiple Output

NF Noise Figure

NN Neural Network

OFDM Orthogonal Frequency-Division Multiplexing

OLOS Obstructed Line Of Sight

OTA Over The Air

PAM Pulse Amplitude Modulation

PCA Principal Component Analysis

PCM Pulse Code Modulation

PDF Probability Density Function

QAM Quadrature Amplitude Modulation

QPSK Quadrature Phase Shift Keying

RCS Radar Cross Section

ReLU Rectified Linear Unit

RF Radio Frequency

RFFE Radio Frequency Front End

RL Reinforced Learning

RMS Root Mean Square

RRC Raised Root Cosine

RV Random Variable

SDR Software Defined Radio

SGD Stochastic Gradient Descent

SNR Signal to Noise Ratio

SPS Samples Per Symbol

STO Symbol Timing Offset

SVM Support Vector Machine

USRP Universal Software Radio Peripheral

WSSUS Wide-Sense Stationary Uncorrelated Scattering

Chapter 1

Introduction

1.1 Motivation

Since the late 1990s, the use of Neural Networks (NNs) in wireless communications has gained a significant following [7]. In an effort to reduce long analysis and design cycles, as well as improve performance in certain aspects of wireless communications, NNs have been implemented as a data-driven approach to solving many challenges. They have been proven to be an effective approach due to several attractive properties, including adaptive processing, universal approximation, and computational efficiency [8].

With numerous existing closed-form solutions in the field of communications, it may be difficult to know when it is appropriate to use a NN to complete a task. To start, for NNs in wireless communications to be suitable for the challenge under consideration, there must not be a direct, closed-form solution. After it is decided the use of a NN is suitable, it must be decided if the whole problem should be solved using a NN, or just to solve for a portion of the ultimate answer by leveraging a NN. The accuracy of NNs depends first on the quality and relevance of training data to testing data, so whether simulating or experimentally collecting the data, it is important to take care and consider its quality and relevance. Another consideration when implementing NNs is that they can very quickly become computationally burdensome if their architecture is large. Additionally, over-training NNs to noise in the data can present issues when decisions are made by the NN on testing data, and is caused by too many neural connections [7].

1.2 State of the Art

At present, there exist many opportunities for applying data-driven Artificial Intelligence (AI) to communications tasks. Digital Pre-Distortion [9] (DPD), localization [10], modulation classification [11], Error Control Code (ECC) decoding [12], Multiple Input Multiple Output (MIMO) detection [13], entire transmit and receive chains [14], Software-Defined Radios [15] (SDRs), and channel modeling [16] have all witnessed

improvements due to recent advances in AI. The focus of this thesis, however, is in generalized training and testing, and so three of this years most impactful papers on this topic are discussed:

- Deepsig's 2018 dataset [17] aims to increase machine-learning based signal classification accuracy in the presence of unforeseen noise. A highly-plausible set of noise is synthesized and applied to signals paired with 24 modulation class labels (including high-order modulation schemes such as QAM256). Each signal is additionally paired with a Signal to Noise Ratio (SNR) value ranging from -20 to +20 (a total of 239,616 across all class combinations). A signal contains 1024 complex-valued samples double-precision floating-point values. Each transmission is affected by the non-idealities of Rayleigh fading (taps defined as $H \triangleq \sum_i \delta(t - Rayleigh_i(\tau))$ where $\tau = [0, 0.5, 1, 2]$), carrier frequency offset $\Delta f_c \sim N(0, \sigma_{clk})$, pulse shaping using Root-Raised Cosine (RRC) filters with roll-off values $\alpha \sim U(0.1, 0.4)$, phase ambiguity $\theta_c \sim U(0, 2\pi)$, sampling frequency offset $\Delta f_s \sim U(0, \sigma_{clk})$, and timing offset $\Delta_t \sim U(0, 16)$. Using an Ettus Lab B210 Universal Software Radio Peripheral (USRP), the authors found that a NN trained with their simulated 2018 dataset only suffered a 7% peak modulation classification accuracy penalty compared to the same NN trained with a measured Over The Air (OTA) dataset.

- Researchers from the Israel Institute of Technology [18] investigated the cause of what they call the "generalization gap", or the phenomenon where NNs trained using mini-batch SGD have a testing accuracy less generalizable to unforeseen noise the bigger the mini-batch size. They found that the generalization gap is caused by having few parameter updates and not from the mini-batches being large. Furthermore, the generalization gap can be reduced by adjusting the training method to include more parameter updates such that $\eta \propto \sqrt{M}$, where η is the parameter update rate and should be increased by a rate proportional to the square root of the mini-batch size, $\sqrt{M}$. They also showed that weight updates during NN training should be scaled by a unit-mean Gaussian Random Variable (RV) of variance $\sigma^2 \propto M$ such that local minima in the objective function can be escaped, and the absolute minimum can be reached (thus achieving maximum signal classification performance). Notably, with this method of gradient update noise, the authors did not find much benefit in also implementing the very popular dropout, drop-connect, or label noise, which have been the industry standard for almost a decade. Finally, they present a "Ghost Batch Normalization" method which performs SGD with few parameter updates yet with a insignificant generalization gap. By calculating the mean $\mu^l_{B_s}$ of the l^{th} batch and standard deviation $\sigma^l_{B_s}$ of small (significantly smaller than batch size B_L) "Ghost Mini-Batches", generalization error can be significantly reduced without increasing η by shifting and scaling the weights γ during SGD by updating in the form $\gamma \frac{X^l - \mu^l_{B_s}}{\sigma^l_{B_s}} + \beta$ where β are the gradient updates and γ are the current weight values. The authors found in experiments that

the learning rate needs to be adjusted from standard SGD such that $\eta_L = \sqrt{\frac{|B_L|}{|B_s|}}\eta_s$.

- Researchers from Google Brain found that the generalization of a NN corresponds to an "input-output Jacobian norm" and "number of transitions" metrics [19]. Consider the Jacobian $\mathbf{J}(x) = \delta \mathbf{f}_\sigma(x)/\delta x^T$, where the NN inputs x are passed through the soft-max function $\mathbf{f}_\sigma$, resulting in the generalization sensitivity metric "Jacobian norm": $E_{xtest}\left[||\mathbf{J}(x_{test})||_F\right]$ around the points of interest x_{test}. The second generalization sensitivity metric they define is the number of transitions, or the number of activated ReLU functions ($f(x) = max(0, x)$) in the hidden layers of a NN. They define this metric as $E_{xtest}[t(x_{test})]$, where $t(x) = \int_{z \in T(x)} \left|\left|\frac{\delta c(z)}{\delta dz}\right|\right|_1 dz$. Neurons are sampled and put into the space $T(x)$, and the last hidden layer's output is described as $c(x)$. The authors plot generalization sensitivity described by those two metrics between inter-class and intra-class perturbations, revealing where in high-dimensional feature-space that perturbations are likely to cause incorrect classification. The paper gives the example of a Gaussian perturbation $\Delta x \sim N(0, \epsilon I)$ resulting in a change at the NN's output equal to $\epsilon||\mathbf{J}(x)||_F^2$ and to the number of transitions equal to $\epsilon t(x_{test})$. If the change at the output is large enough, incorrect classification occurs.

1.3 Current Issues

A number of issues remain open in the field of machine learning communications [20]. In order to facilitate ML in systems, current wireless network infrastructures need to be updated to deploy Graphics Processing Units (GPUs) at network edges, allowing for computationally efficient use of ML-based solutions. Network slicing, or the allocation of wireless network resources for different use cases, is an area of communications that still has seen very little attention from recent advances in AI. Additionally, standardized datasets and environments for fair comparisons between architectures still do not exist. While some are more popular than others, there is still not a set of renowned datasets in the wireless community as there is in the computer vision community. Also, there is a significant lacking of theory behind hyper-parameter optimization and data generalization, where both of these tasks are mostly performed by iterating through every option in processes such as hyper-parameter validation, which is a very time consuming process. Distillation or transfer learning is a technique that has not yet been successfully implemented in wireless communications either, or transferring training results from one NN to another for use in a task similar to the first. Although a defensive application was discovered in [21], novel attacks on NNs have shown that the technique became obsolete within months of its writing [22].

Below, issues concerning the three state-of-the-art papers on generalized training are discussed:

- Deepsig's 2018 dataset [17] shows a significant increase in size over their 2016 dataset, going from

220,000 transmissions of 128 samples to 239,616 transmissions of 1024 samples, or 225.28 MB of double-precision (64-bit) complex values to 1.96 GB. While still a small memory allocation compared to many computer vision datasets, generating larger datasets with fewer assumptions increases training time and complexity. It would be valuable for datasets to cover a large range of statistical behaviors using very few samples, a task made difficult given the law of large numbers. Additionally, the authors note that matching channel models to real-world deployment conditions is difficult, taking time to estimate and implement to simulate training signals, leaving NNs inoperable for long down-periods.

- Israel Institute of Technology's GBN method [18] is powerful, however their results show that they can only limit generalization error, at best, to as low as small batch SGD can (but more often reaches an error in-between small and large batch SGD). It would be valuable to navigate around that limit by leveraging a new training method other than small-batch SGD or by manipulating training datasets.

- While the development of the Jacobian norm and number of transitions matrices in [19] are powerful metrics in finding the weak points of classifiers, the metrics are not very intuitive to interpret and analyze, and there currently exists no convex-optimization solution to manipulating sensitivity to minimize incorrect classifications. It is difficult to know how to act on how the Jacobian norm and number of transitions to improve peak accuracy.

1.4 Thesis Contributions

This work contains the following contributions:

- A survey of wireless channel environments was given in Chapter 2 that can be used in the proposed framework for dataset generation.

- A survey of machine-learning based signal classification was given in Chapter 3 that can be used to implement an unsupervised domain adaptation architecture from Chapter 5.

- A framework for wireless transmission dataset synthesis, implementing arbitrary channel environments and baseband waveforms.

- A dataset generated using the framework from Section 4 was proposed, and was shown to be properly sized, having 11% lower entropy than state-of-the-art datasets.

- A Deep Reconstruction-Classification Network (DRCN) was proposed in Chapter 5, which attempts to maintain peak classification accuracy despite heavy data bias resulting in a 16% peak testing

accuracy drop compared to an experiment with all else equal but no data bias. These contributions show both data-side (pre-training) and testing-phase manipulations to increase NN generalization and avoid retraining.

1.5 Thesis Organization

The thesis is organized as follows: Chapter 2 will give a survey of background knowledge learned by the author on the topics of wireless channel modeling. Chapter 3 surveys neural networks with an emphasis on training and data sets, and modulation classification. Chapter 4 presents the author's work on generalized training through the development and use of a low bias, low decay framework that synthesizes low-entropy data sets modeling state-of-the-art wave-forms. Finally, Chapter 5 present's the author's ongoing work on generalized training through the use of the domain adaptation technique, and concluding thoughts are discussed in Chapter 6.

1.6 List of Related Publications

The following publications resulted from the activities of this thesis research:

- K. McClintick, A. Wyglinski. "Physical Layer Neural Network Framework for Training Data Formation." *IEEE 88th Vehicular Technology Conference*, Fall 2018.
- K. Gill, K. McClintick, N. Kanthasamy, "Experimental Test-Bed for Bumblebee-Inspired Channel Selection in an Ad-hoc Network." *IEEE 88th Vehicular Technology Conference*, Fall 2018.

Chapter 2

Understanding the Wireless Communications Environment

The contributions of this thesis require a systematic understanding of wireless environments. Consequently, this chapter presents a survey of classical channel model theory to provide context and knowledge needed in discussion of Chapter 3, dataset synthesis.

Although transmitted waveforms begin as well defined, man-made, synthetic structures, a virtually endless number of probabilistic, and sometimes non-linear, phenomenon alter the observed waveforms receive-side [5]. Even within a single noise model, there can exist a limitless number of variations of that imperfection from one wireless channel to another. Some of the most prevalent and common imperfections include:

1. Additive White Gaussian Noise (Section 2.1), a model used to mimic the effects of many wideband noise sources
2. Path loss (Section 2.2) reduction of signal power density due to reflection (Section 2.3), diffraction (Section 2.4), scattering (Section 2.5), absorption, aperture-medium coupling loss, and free-space loss
3. Doppler shifts (Section 2.6) resulting from motion of the transmitter, receiver, or scatterers and reflectors within the wireless channel
4. Carrier Frequency Offset (Section 2.7.1) of both the transmitter and receiver's local oscillators, which drive each radio's mixers
5. Phase ambiguity (Section 2.7.2) introduced by the unknown distance between transmitter and receiver
6. Random Symbol Timing Offset (Section 2.7.3) resulting from independently running sample clocks

7. IQ imbalance (Section 2.7.4) resulting from phase and magnitude mismatches between the sine and cosine sections of receiver and transmitter chains

8. Rounding of sampled voltages and digital filter coefficients due to Quantization (Section 2.7.5)

9. Electronic Noise (Section 2.7.6) caused by semi-conductors such as shot and flicker noise

10. Coupled noise (Section 2.8) resulting from inter-modulation, interference from same and adjacent channels, industrial noise, Cosmic and terrestrial events

When a communications transmit-receive pair move information from one point to another, there is a great deal of sequential tasks that are performed by the transmit and receive chains (see Figure 2.1). It is the goal of this section to describe popular models which discuss the impact of imperfections of these tasks, as well as perturbations introduced during the informations journey from sender to receiver, over the wireless channel. The first such model that is often discussed in this field is Additive White Gaussian Noise (AWGN).

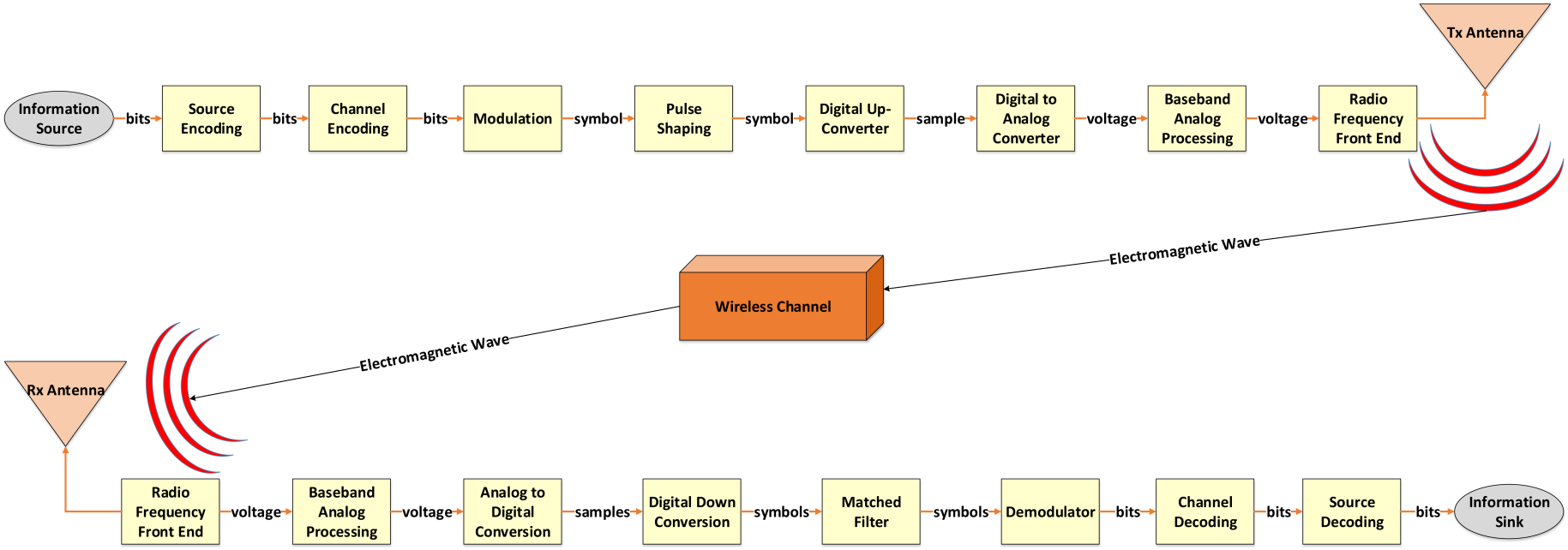


Figure 2.1: A flow chart of a transmit and receive chain of communications tasks. Arrows indicate movement of information from one block to another. The form that information takes at each step is communicated through annotations. Antennas are pictured as upside-down triangles.

2.1 Additive White Gaussian Noise

As it will become apparent in this section, there is a virtually unending number of types and sources of noise in a wireless channel. The Central Limit Theorem (CLT) states that as independent random variables are summed, their joint distribution approaches a Gaussian probability density function (PDF):

$$f_x(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{2.1}$$

where σ is the standard deviation and μ the mean value of the RV, written as $\mathcal{N}(\mu, \sigma^2)$. AWGN is described as additive because it is added to any transmission. There exists enough noise sources to approximate the CLT at all frequencies such that the noise has uniform power across the frequency band. This is why AWGN is described as white, an analogy for how white frequencies of optical wavelengths are the sum of all other colors of the visible light frequency band.

AWGN is fundamental in wireless communications because it imposes a performance ceiling for any task, and that idea extends to NNs involved with wireless data. One way of describing this ceiling was presented by Claude Shannon in [23]. The channel capacity C of a wireless channel with power constraint $\frac{1}{k}\sum_{i=1}^{k} x_i^2 \leq P$ for k message codewords $x_1, x_2, ..., x_k$, is:

$$C = \frac{1}{2}\log_2\left(1 + \frac{P}{N}\right), \tag{2.2}$$

where N is the wireless channel AWGN variance. Codewords make up a codebook, or all possible messages sent. Consider the optical telegraph [24], a mid-1700s invention of the French Chappe brothers. Five brightly colored panels are painted onto a board and hidden by shutters that can be either open (1) or closed (0), conveying $2^5 = 32$ messages, $x \in \{1,0,0,0,0\}, \{0,1,0,0,0\}, ..., \{1,1,1,1,1\}$. The larger the number and dimensionality of codewords, the larger the power constraint P, the larger the channel capacity C.

2.2 Path Loss

When implementing an AWGN channel model, a *Link Budget* technique can be used to approximate the signal power, which is needed to determine SNR. A link budget is a calculation that determines received power P_r of a wireless transmission as a function of the transmit chain, wireless channel, and receive chain parameters. According to the Friis formula, the received power can be calculated in dBm as [1]:

$$P_r = P_t + G_t + G_r + 20\log_{10}\left(\frac{\lambda}{4\pi d}\right), \tag{2.3}$$

where P_t is the transmitted power, G_r, G_t are the receiver and transmit antenna gains, λ is the wavelength of the transmitted signal, and d is the transmission distance. The formula has a few assumptions: far-field

transmission ($d > \frac{2D^2}{\lambda}, d > 10D, d > 10\lambda$ for antenna length D), the signal is narrowband, and antennas are isotropic in the direction of transmission. Notice that power received decreases with both increased frequency and distance.

For high power, long range transmissions of frequencies up to L-band (2000 MHz) from a tall base-station tower to a mobile user, the most popular topographical path loss model to be used in the free space Friis formula defined by equation (2.3) is the Okumara-Hata [25] empirical path loss model, whose behavior was collected in Tokyo, Japan using isotropic antennas, and has seen widespread use [26]. The formula is rated for use up to 100 km and for at least 1 km, and is rated for a base-station height up to 200 m and a mobile receiver height of up to 3 m. Over the years, many measurement campaigns [27] have been conducted to expand the range of distances and frequencies the model can accommodate. The original Okumara model can be described [28] as use of the Friis formula in (2.3) where L_{path} is instead:

$$L_{50,dB} = L_F + A_{mu}(f, d) - G(h_{te}) - G(h_{re}) - G_{AREA}, \tag{2.4}$$

where $L_{50,dB}$ is the 50th percentile (mean) of the propagation loss, L_F is the free space propagation loss, $A_{mu}(f, d)$ is the median attenuation relative to free space (see Figure 3.24 in [28]), $G(h_{te}), G(h_{re})$ are the transmitting and receiving antenna gain factors in (2.5), respectively, and G_{AREA} is the gain with respect to the type of environment (see Figure 3.23 in [28]).

For various antenna heights, the antenna gain factors can be described as [1]:

$$G(h_{te}) = 20\log\left(\tfrac{h_{te}}{200}\right), \quad 1000m > h_{te} > 30m \tag{2.5a}$$

$$G(h_{re}) = 10\log\left(\tfrac{h_{re}}{3}\right), \quad h_{re} \leq 3m \tag{2.5b}$$

$$G(h_{re}) = 20\log\left(\tfrac{h_{re}}{3}\right), \quad 10m > h_{re} > 3m. \tag{2.5c}$$

2.3 Reflections and Fading

Instead of link budgets, *Ray Tracing* Algorithms can be a preferred category of techniques when determining signal power, especially in physically smaller wireless channels with well-defined dimensions. A popular ray tracing model for representation of a channel with path loss is the discrete delay channel model [1] (see Figure 2.2). The model behaves differently for narrow and wideband signals.

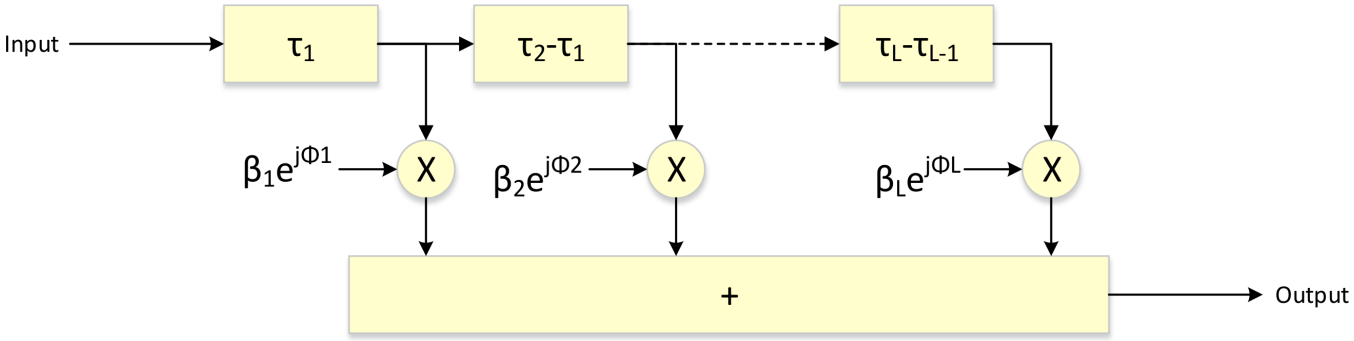


Figure 2.2: The discrete delay channel model, adapted from [1]. Inputs each have isolated time delays τ_i, ray powers $|\beta_i|^2 = A_0 a_i/d_i$, and ray phases $e^{j\phi_i}$.

A narrowband signal is defined as a transmissions whose bandwidth does not significantly exceed the coherence bandwidth of the channel the signal is traveling through. A received signal is considered significantly wide if the inverse of the channel's root mean squared (rms) delay spread τ_{rms}:

$$\tau_{rms} = \sqrt{\overline{\tau^2} + (\overline{\tau})^2}, \tag{2.6}$$

is five times smaller than the signal's bandwidth [1]:

$$\frac{5}{\tau_{rms}} < BW. \tag{2.7}$$

The delay spread $\overline{\tau^n}$ is defined as:

$$\overline{\tau^n} = \frac{\sum_{i=1}^{L} \tau_i^n |\beta_i|^2}{\sum_{i=1}^{L} |\beta_i|^2}, \tag{2.8}$$

where the additional time required for the ith signal path, or ray, to arrive is τ_i, and the power of the ith ray is $|\beta_i|^2 = A_0 a_i/d_i$. The path distance of the ith ray is d_i, the overall reflection coefficient of the ith ray is:

$$a_i = \sum_{j=1}^{K_i} a_{ij}, \tag{2.9}$$

where a_{ij} is one of K_i reflections for the ith ray. $A_0 = \sqrt{P_0}$, the power of the received signal from one meter away:

$$P_0 = P_t G_r G_t (\lambda/4\pi)^2. \tag{2.10}$$

If a signal is determined to be narrowband using (2.7), the received power can be formulated as:

$$P_r = P_0 \left| \sum_{i=1}^{L} \frac{a_i}{d_i} e^{j\phi_i} \right| \tag{2.11}$$

where the received phase offset $\phi_i = -2\pi d_i/\lambda$. Notice how the phase of each ray $e^{j\phi_i}$ has the potential to add constructively or destructively (see Figure 2.3).

The discrete delay channel model (see Figure 2.2) behaves differently for wide band signals. A wideband signal in the frequency domain can be shown to be of a short time duration in the time domain [28]. Often

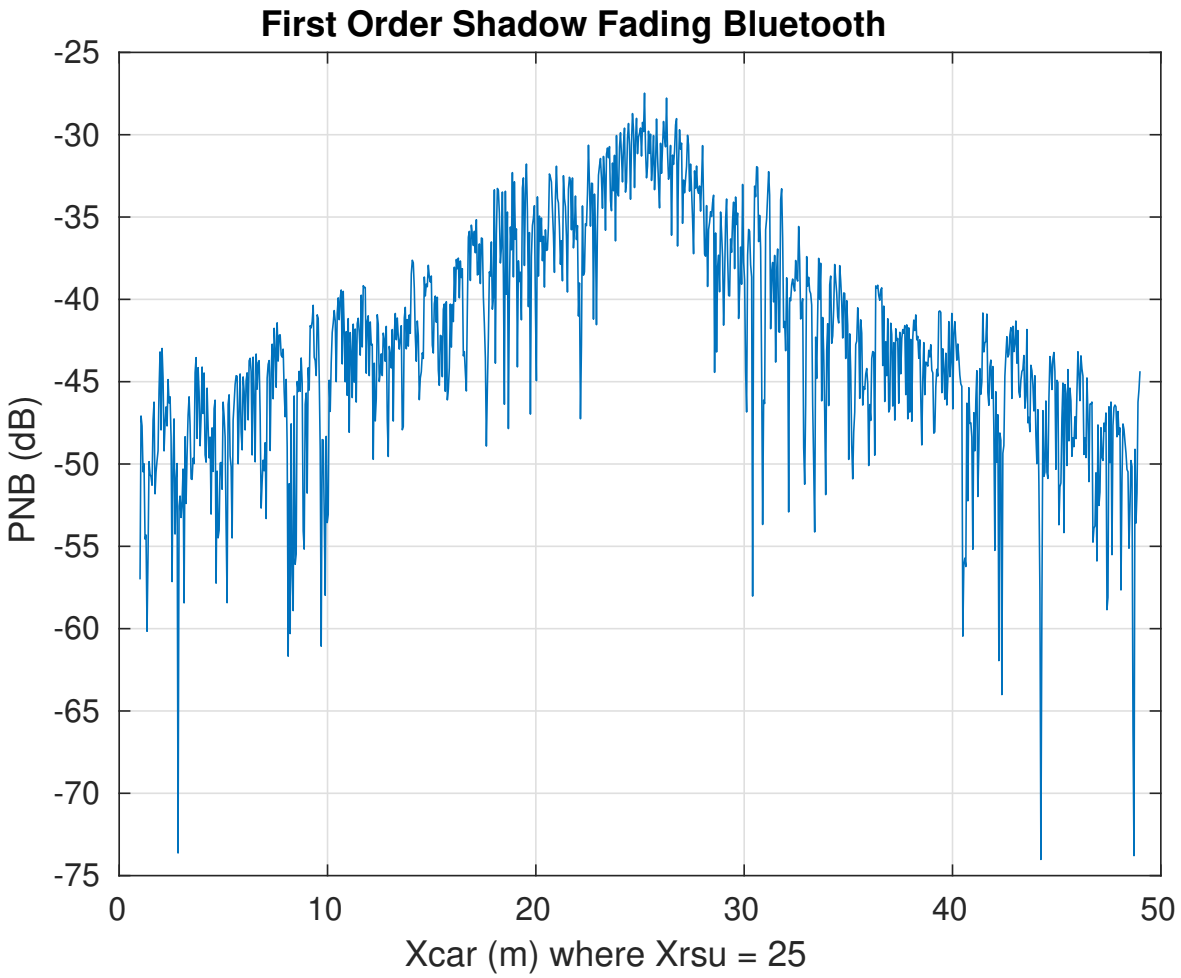


Figure 2.3: Bluetooth P_r (2.11) observed by a receiver (See Appendix A.1) from a single car over a 50 m stretch of road, whose reflections correspond to the scenario described in Figure 2.4. Significant fluctuations occur frequently due to phase interference.

these bursts of signals are modeled as impulses, $\delta(t)$ (typically Gaussian pulses in application). In ideal wideband communications, each path of arrival an impulse makes from transmitter to receiver are isolated. Additionally, since impulse durations are instantaneous compared to time delays τ_i and do not overlap in the time domain, the phase offset of each ray does not add constructively or destructively. Consequentially, the received power of a wideband signal can be formulated as [1]:

$$P_r = P_0 \left| \sum_{i=1}^{L} \frac{a_i}{d_i} \right| = \sum_{i=1}^{L} |\beta_i|^2, \tag{2.12}$$

and unlike (2.11), the phase term $e^{j\phi_i}$ is missing. The result is that wideband path loss does not vary from (2.3) very much, unlike (2.11). For a simulated wideband path loss model, see Figure 2.5.

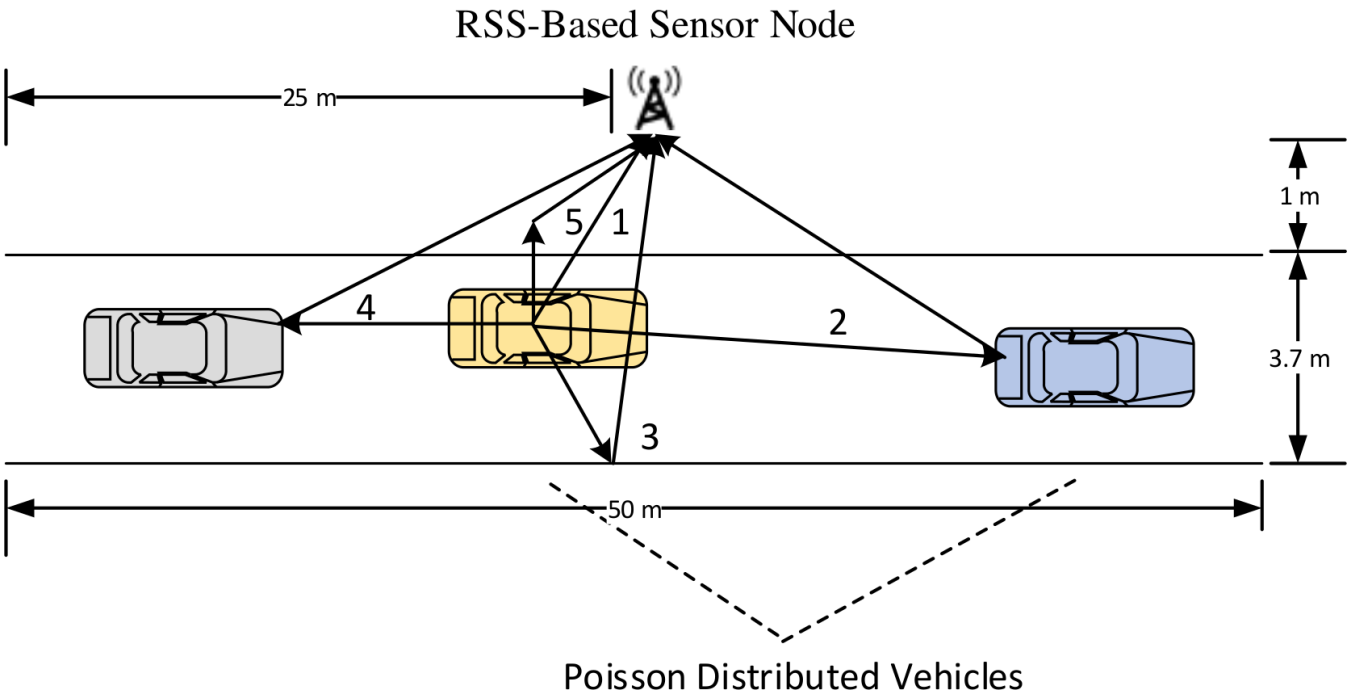


Figure 2.4: Physical Characteristics of the 5-reflection 3D Doppler channel. The paths are attenuated assuming the ground path is dry asphalt[ref] ($a_5 = 0.34$), the roadsides are concrete ($a_3 = 0.38$), and the cars are metallic ($a_{2,4} = 0.8$). The direct path is not attenuated by reflection ($a_1 = 1$). Blue-tooth sources are placed at a height of 1.5 m and the receiver at a height of 5m.

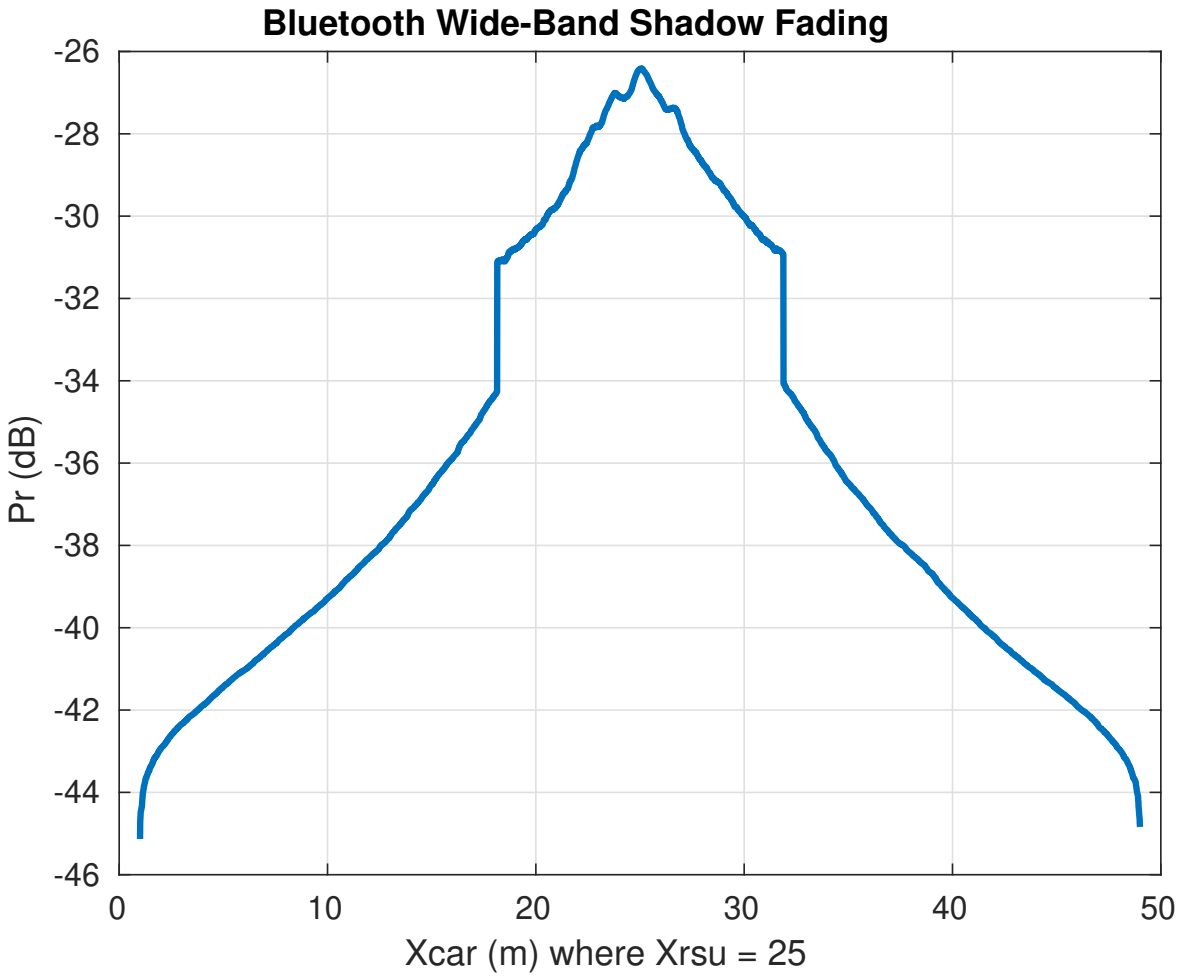


Figure 2.5: Blue-tooth P_r (2.11) without phase interference observed by a receiver (See Appendix A.1) from a single car over a 50 m stretch of road, whose reflections correspond to the scenario described in Figure 2.4.

2.4 Diffractions due to Obstructions

Besides reflections, ray tracing algorithms must consider a phenomenon with electromagnetic waves known as diffraction. Diffraction allows wireless signals to propagate to receivers they do not have Line of Sight to, most notably long range transmissions to receivers blocked by the curvature of the earth. A popular diffraction model used to modify (2.3) is the knife-edge diffraction model [28]. Given the transmitter (see Figure 2.6) Υ receiver R, obstruction of height h above the direct path from Υ to R, and distances d_1, d_2 from the transmitter and receiver to the obstruction respectively, the Fresnel-Kirchoff diffraction parameter can be calculated as:

$$v = h\sqrt{\frac{2(d_1 + d_2)}{\lambda d_1 d_2}}, \tag{2.13}$$

to estimate the additional gain to free space (2.3) as:

$$G_{dB} = 20 \log |F(v)|, \tag{2.14}$$

where the Fresnel integral can be calculated as:

$$F(v) = \frac{(1+j)}{2} \int_v^\infty e^{(-j\pi t^2)/2} dt. \tag{2.15}$$

While it is often sufficient [28] to model just the largest diffraction, there are situations where a multiple knife-edge diffraction model would increase the accuracy of a path loss model significantly.

2.5 Scattering due to Corrugated Surfaces

The final consideration when determining received power is how electromagnetic waves are scattered. If one is to model path loss given by (2.3) just using the reflection expression of (2.11) and diffraction expression of (2.14), the observed path loss would be higher [28] due to the energy being scattered in all directions when impacting rough surfaces. Surface roughness is described using the Rayleigh criterion [28] defined as the critical height h_c in meters:

$$h_c = \frac{\lambda}{8 sin\theta_i}, \tag{2.16}$$

where λ is the signal's center frequency and θ_i the signal's angle of incidence on the flat surface under consideration. A surface is considered to only have significant scattering if its protuberances are of height greater than h_c. If so, that surfaces reflection coefficient a_i should be multiplied by a scattering loss factor:

$$\rho = exp\Big[-8\Big(\frac{\pi\sigma_h sin\theta_i}{\lambda}\Big)^2\Big] I_0\Big[8\Big(\frac{\pi\sigma_h sin\theta_i}{\lambda}\Big)^2\Big], \tag{2.17}$$

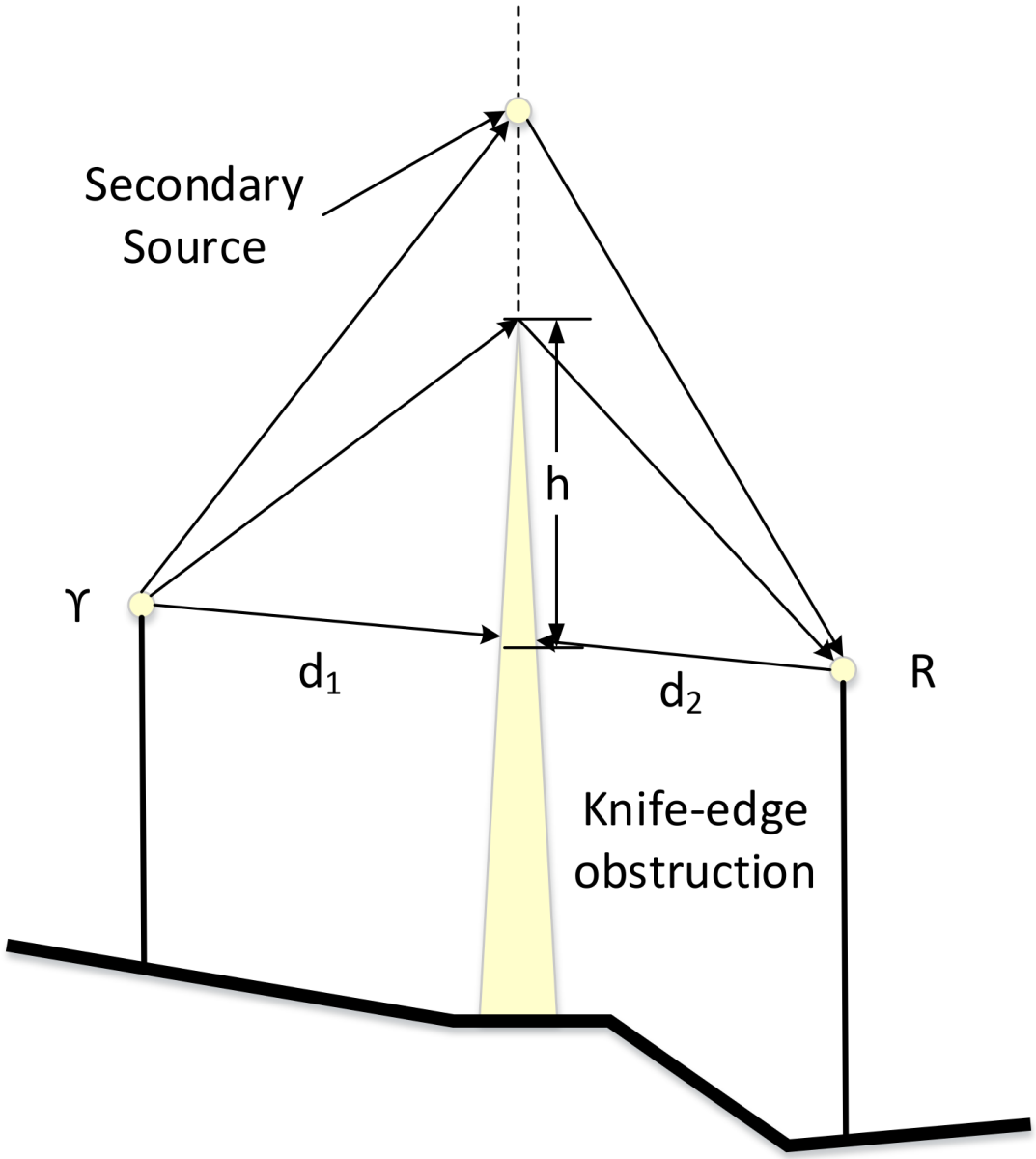


Figure 2.6: An illustration of (2.14), where Υ is the transmitter, R is the receiver, h is the height of the obstruction starting from the direct path from Υ to R, and d_1, d_2 from the transmitter and receiver to the obstruction, respectively. The Huygens secondary source mimics a potentially strong reflected path, often taking the form of a reflection off a layer of the earth's ionosphere.

to form $a_{rough} = a\rho_s$.where I_0 is the Bessel function of the first kind and zero order, and σ_h is the standard deviation of the assumed Gaussian variations of the surface's protuberances about the mean.

A popular link budget model for scattering is the radar cross section model [28], which assumes a large, distant (in the far-field of both the receiver and transmitter, $R > \frac{2d^2}{\lambda}, R > 10d, R > 10\lambda$), rough surface (defined as $h > h_c$) is scattering a transmission. Like in (2.3), transmit power, antenna gains, and path

loss are considered, with a few additional terms:

$$P_R(dBm) = P_T(dBm) + G_T(dBi) + 20log_{10}(\lambda) + RCS(dB \cdot m^2) - \\ 30log_{10}(4\pi) - 20log_{10}(d_T) - 20log_{10}(d_R), \tag{2.18}$$

where d_T and d_R are the distance from the transmitter and receiver to the scatterer, and the Radar Cross Section (RCS) can be approximated as the surface area of the scatterer in square meters measured in dB with respect to a one square meter reference.

2.6 Doppler Spectrum

Another wireless phenomenon that can affect system performance is Doppler Shift. In practice, information is often sent in a radio system under mobile conditions. The impacts of this can be considerable, especially in satellite and railway communications. Consider a sine wave being transmitted via a radio link in a mobile environment, where a transmitter is moving at velocity V_m at an instantaneous distance d_0 from a stationary receiver.

As the transmitter moves towards the receiver, the instantaneous distance d_0 will shrink, impacting transmission time as a function of time [1]:

$$\tau(t) = \tau_0 - \frac{V_m}{c}t, \tag{2.19}$$

where c is the speed of light in free-space, $3 \cdot 10^8$ m/s, and $\tau_0 = d_0/c$ is the starting transmission time. Given this, the transmitted sine wave can be formulated by Euler's formula as [1]:

$$r(t) = A_r e^{j2\pi f_c[t-\tau(t)]} = A_r e^{j[2\pi(f_c+f_d)t-\phi]}, \tag{2.20}$$

where f_c is the sine waves carrier frequency, A_r is the amplitude of the received signal, $\phi = 2\pi f_c tau_0$ is the current phase offset, and f_d is the Doppler shift caused by movement:

$$f_d = \frac{V_m}{c} f_c \cos(\theta), \tag{2.21}$$

where θ is the direction of movement, for zero degrees being moving straight towards the receiver. The frequency shift observed by the receiver is positive or negative depending on the direction of movement, where magnitude is maximized by the cosine when moving exactly toward or away from the transmitter. Commonly these maximal outcomes of (2.21) are described as the maximum Doppler shift $f_M = \pm\frac{V_m}{c} f_c$ of bandwidth $B_D = 2f_M$.

However, as shown in Section 2.2, rarely is there one ray (see Figure 2.2) in a wireless channel. Each ray is affected differently, and as consequence the frequency domain representation of the signal is affected by a Doppler spectrum, or Doppler spread $D(\lambda)$. Consider the wireless channel responding to probing

impulse responses $\delta(t)$ at time delays τ in the form $h(\tau, t)$, where for the input $x(t)$, the channel output is $y(t) = x(t) \circledast h(t)$:

$$h(\tau, t) = \sum_{i=1}^{L} \beta_i e^{j\phi_i} \delta(t - \tau_i). \tag{2.22}$$

The autocorrelation of the observed impulse response at two different delays and times can then be formulated as [1]:

$$R_{hh}(\tau_1, \tau_2; t_1, t_2) = R_{hh}(\tau_1; \Delta t)\delta(\tau_1 - \tau_2), \tag{2.23}$$

for $\Delta t = t_2 - t_1$. Given R_{hh}, the autocorrelation in the frequency domain given the time-varying frequency domain impulse response $H(f; t) = \int_{-\infty}^{\infty} h(\tau; t)e^{-j\omega\tau} d\tau$ is formulated in [1] as:

$$R_{Hh}(f_1, f_2; \Delta t) = R_{Hh}(\Delta f; \Delta t), \tag{2.24}$$

where $\Delta f = f_2 - f_1$. The channel is assumed to have Wide-Sense Stationary Uncorrelated Scattering (WSSUS), which is valid for most wireless channels [1]. A WSSUS wireless channel is one in which the delays τ_i are uncorrelated, and the correlations between paths of equal delays are stationary, or time-invariant. Given this, the autocorrelation can be estimated as $R_{Hh}(\Delta f)$ if the channel is slowly time-varying or time-invariant.

Finally, we can derive the Doppler spectrum $D(\lambda)$ using the Fourier transform of (2.24):

$$R_{HH}(\Delta f; \lambda) = \int_{-\infty}^{\infty} R_{Hh}(\Delta f; \Delta t)e^{-j2\pi\lambda\Delta t} d(\Delta t), \tag{2.25}$$

such that:

$$D(\lambda) = R_{HH}(0; \lambda). \tag{2.26}$$

The spectrum is limited by $\pm f_M$, and the amount of variation of the spectrum over frequency is described by the Doppler spread:

$$B_{D,rms}^2 = \frac{\int_{-f_M}^{f_M} \lambda^2 D(\lambda) d\lambda}{\int_{-f_M}^{f_M} D(\lambda) d\lambda}. \tag{2.27}$$

Equations (2.23) through (2.25) are summarized in Figure 2.7.

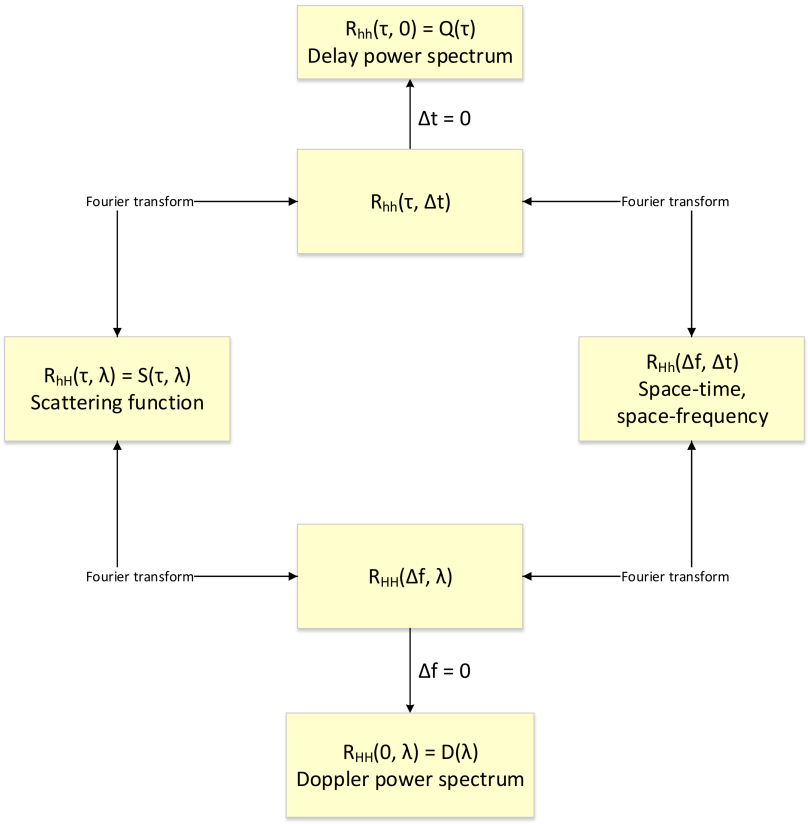
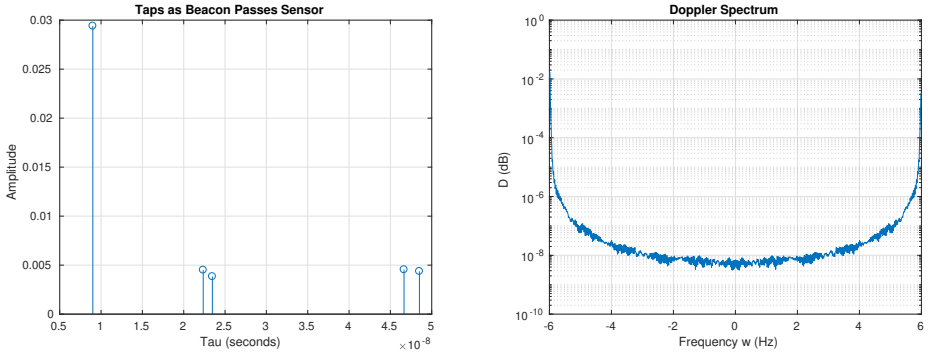


Figure 2.7: A flow chart adapted from [1] summarizing equations (2.23) through (2.25). Arrows indicate Fourier (down) and inverse Fourier (up) transforms.

In the most basic multi-path case, (2.26) takes the form:

$$D(f) = \frac{1}{2\pi f_m}\left[1 - \left(\frac{f}{f_m}^2\right)\right]^{-1/2}, |f| \leq f_m, \tag{2.28}$$

or Jakes Doppler spectrum, where (2.21) is assumed and can be plotted over normalized frequency as seen in Figure 2.8.



(a) Five taps plotted to display phasor delay and magnitude.

(b) Jakes Doppler spectrum (b) of the scenario described by Figure 2.4.

Figure 2.8: Taps (a) (See Appendix A.1) calculated from the phasers and time delays described by Figure 2.4 as the transmitting car passes by the receiver at $x = 25m$. Jakes Doppler spectrum (b) where frequency offset is maximal at $\pm f_M$ (2.21), or movement directly away from and towards the receiver. Small fluctuations in the spectrum is caused by movement within the channel caused by the leading and lagging vehicles.

However, there is a whole field dedicated to modeling unique cases of Doppler shift. Some common cases in addition to Figure 2.8 are displayed in Figure 4.9 of [1]. The series of experiments include a series of impulse responses (2.22) and their Fourier transforms, describing a LOS experiment using a stationary radio transmit-receive pair in a stationary environment ($B_D = 0Hz$ see (2.27)), and a LOS experiment using a stationary receiver, but mobile transmitter that moved randomly within a 12 meter radius of a fixed point to simulate a pseudo-stationary mobile user pacing on their telephone ($B_D = 4.9Hz$). An obstructed LOS (OLOS) experiment is described, using stationary devices 4 meters apart, but with heavy pedestrian traffic around the transmitter ($B_D = 5.7Hz$), and an OLOS experiment with stationary devices, but the transmitter is rotated at a rate of 2.5 rotations per second ($B_D = 5.2Hz$). Each experiment describes a family of movement, corresponding to time and frequency domain Doppler signals that can be expected in each, while also showing that B_D can be equal for very different spectrum's $D(\lambda)$ shapes.

2.7 The Radio Front End

The RFFE (see Figure 2.1) is a term used to group all of the analog circuitry between a transmit or receive chain's antenna and mixer. At the most basic, RFFE's contain:

- A Band Pass Filter (BPF), used to pass through the expected signal at the expected carrier frequency and block out all other signals and noise. In-band noise and interference is still present. BPFs can also damage signals due to in-band ripple, and are vulnerable to thermal noise, shot noise, and

transit-time noise (see Section 2.7.6).

- A Low-Noise Amplifier (LNA), used to increase the power of in-band signals above the noise floor. LNAs must have a low noise figure (NF), and are often only needed at frequencies above 30 MHz due to the increased path loss (2.3).
- a Mixer, used to combine the carrier waveform with the transmitted or received waveform to form the base-band signal or the Intermediate Frequency (IF) signal.
- A Local Oscillator (LO), used to drive the up-converting and down-converting mixers by creating a carrier cosine waveform. Phase noise (see Section 2.7.6) can be introduced by flicker noise, and the frequency of the carrier can drift with time (see Section 2.7.1).

Besides the issues listed above, the initial spacing between a transmit and receive radio can introduce an initial phase offset, introducing phase ambiguity (see Section 2.7.2) even after frequency correction is performed. Digital filters and the DAC/ADC (see Figure 2.1) can introduce significant error to signals through quantization (see Section 2.7.5), and in the case of pulse-shaped (see Figure 3.6) waveforms, symbol timing offset (STO) (see Section 2.7.3) can push bit error rates (BER) to their limits, 1/M (3.48).

Furthermore, the cosine and sine paths of the RFFE (see Figure 2.14) experience phase and magnitude imbalances, resulting in stretched IQ plots (see Section 2.7.4).

Finally, various forms of electronic noise (see Section 2.7.6) can impede SNR, sometimes significantly.

2.7.1 Carrier Frequency Offset due to Local Oscillator Mismatch

Each radio system (see Figure 2.1) has either a down-converter or an up-converter (see Figure 2.9), shifting the center frequency of the signal (see Figure 2.10) either up to the carrier frequency if transmitting or down to the base-band if receiving.

However, this frequency shifting is not perfect and is often affected by a phase offset (see Section 2.7.6), a frequency offset, and the creation of signal images. Images are lower-amplitude copies of $m(t)$, the signal being converted by the mixer, appearing at locations:

$$f_{image} = \begin{cases} f_c + 2f_{IF} & \text{if } f_{LO} > f_c \\ f_c - 2f_{IF} & \text{if } f_{LO} < f_c \end{cases}, \tag{2.29a}$$

$$f_{image} = f_c + 2f_{LO}, \tag{2.29b}$$

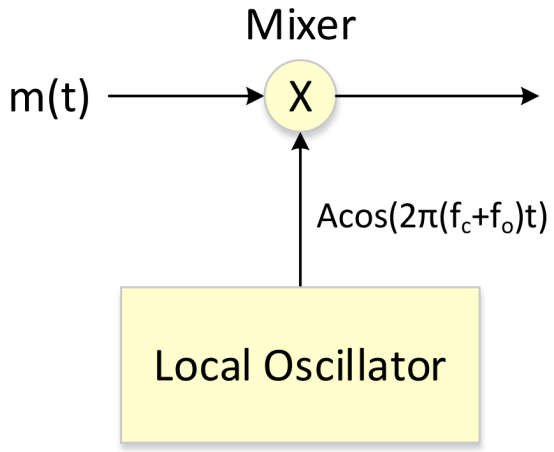


Figure 2.9: An illustration of the base-band signal $m(t)$ being up-converted to the intermediate frequency f_c by a mixer, where the carrier waveform $A_c cos(2\pi(f_c + f_o)t)$ is generated by a local oscillator (LO). The error introduced by the LO, f_o being random and unequal at the transmitter and receiver, frequency offset is leftover after being down-converted to baseband (see Figure 2.10, whether the receiver be a super-heterodyne or direct one.

for down converters and up-converters respectively, where the intermediate frequency $f_{IF} = |f_c - f_{LO}|$. The frequency offset introduced by LOs can be modeled as [**?**]:

$$f_{o,max} = \frac{f_c \times PPM}{10^6}, \tag{2.30}$$

where PPM is the parts per million resolution of the LO, often listed in a radio's user manual, f_c is the carrier frequency, and $f_{o,max}$ is the maximum possible negative or positive frequency offset. For a transmit receive pair, the total offset can then be defined as $f_{o_1} + f_{o_2}$, where each offset is a Gaussian random variable bounded by each radio's $f_{o,max}$, making the random variable no longer Gaussian. This results in a minimum possible offset of zero Hertz when each offset is zero ($f_{o_1} = f_{o_2} = 0$) or equal and opposite to each other ($f_{o_1} = -f_{o_2}$), and a maximum possible offset of $\pm 2 \times f_{o,max}$ when the offsets are equal to the maximum offset ($f_{o_1} = f_{o_2} = \pm f_{o,max}$).

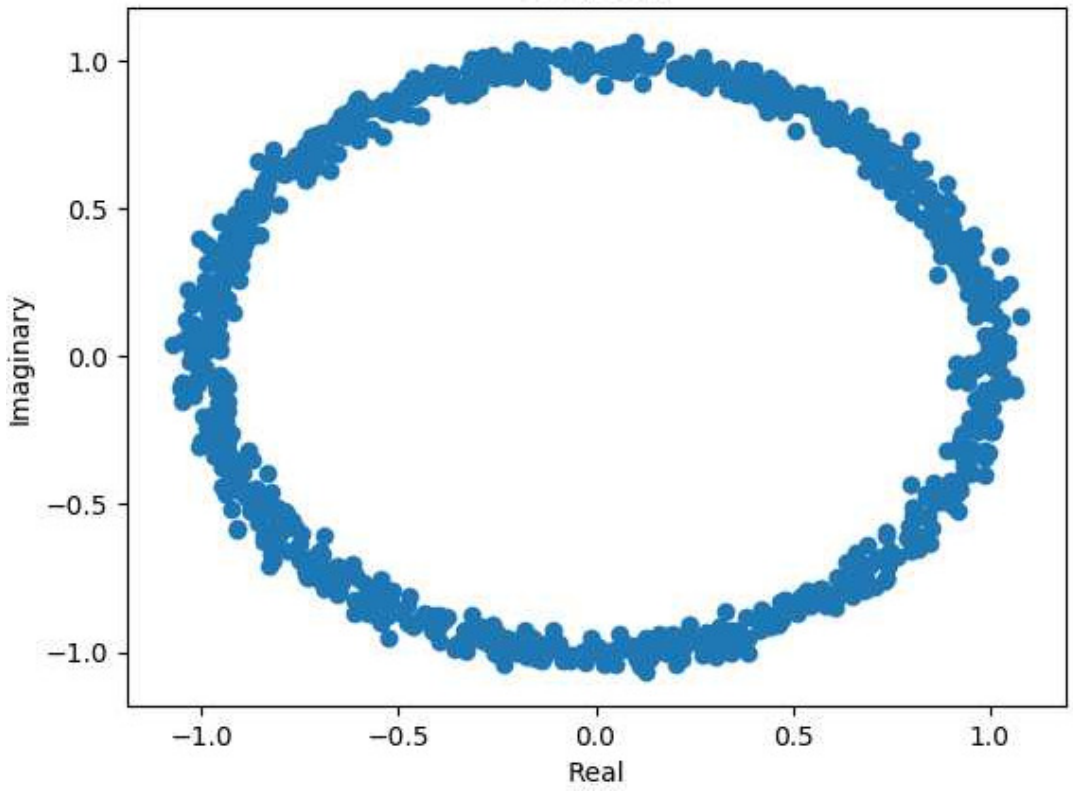


Figure 2.10: An IQ plot of a QPSK message offset in frequency. Phase rotation over time makes demodulation inaccurate and Bit Error Rate (BER) high.

2.7.2 Phase Ambiguity after Frequency Correction

As seen in Figure 2.3, the received phase offset $\phi_i = -2\pi d_i/\lambda$ can have significant effects on path loss. However, this issue goes much deeper than received power: when a demodulator (see Figure 2.1) maps a base-band waveform from IQ points to bits, this phase offset rotates the constellation by an amount that has been found experimentally [1] to be random according to the uniform distribution $\mathcal{U}(0, 2\pi)$ radians. Consequently, constellation plots such as Figure 3.21 can become very ambiguous (see Figure 2.11), where the plot could be flipped along the real or imaginary axis and still look identical. As a result, various forms of precautions have been developed to avoid errors resulting from phase ambiguity, including equalization, codewords, and differential encoding.

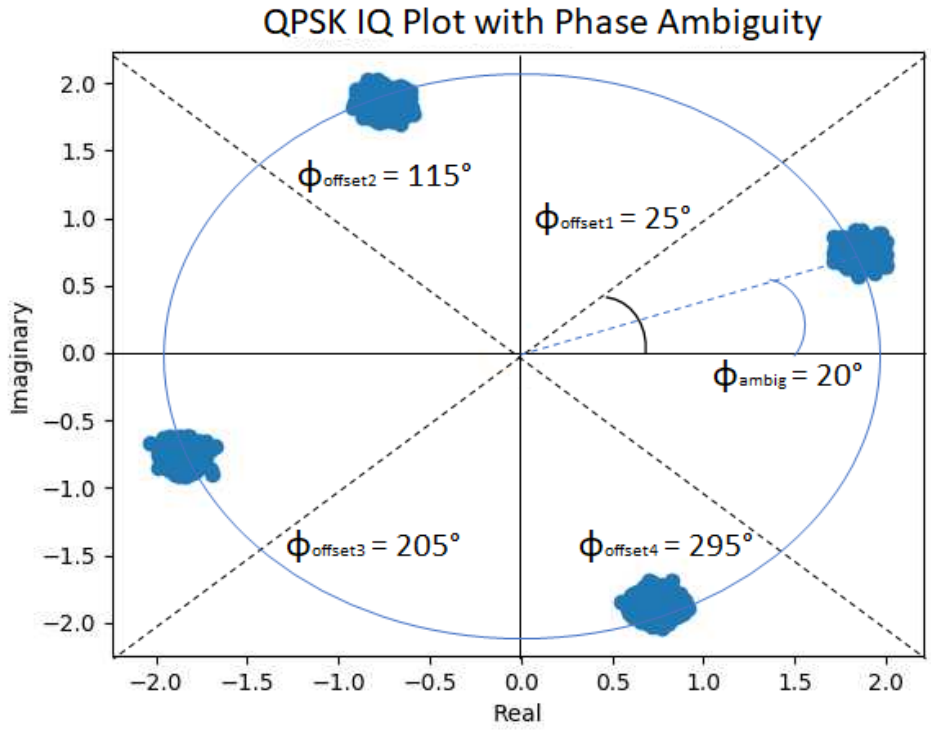
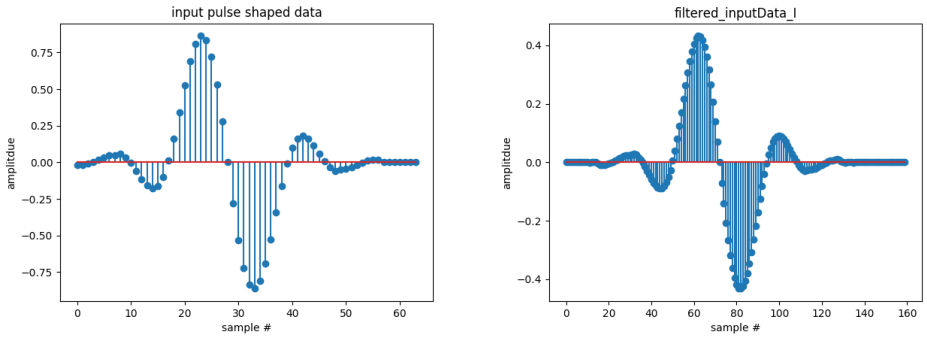


Figure 2.11: A constellation plot of a QPSK transmission. The four code-words are tilted by $\phi_{ambig} = 20°$. It is the job of a receive chain (see Figure 2.1) to determine if the transmission should be corrected by adding one of the rotations: $\phi_{offset1} = 25°, \phi_{offset2} = 115°, \phi_{offset3} = 205°, \phi_{offset4} = 295°$.

2.7.3 Symbol Timing Offset when Down-Sampling at the Receiver

In a radio transmit receive chain (see Figure 2.1), the receiver does not typically have knowledge as to which sample is to be picked to down sample when pulse shaping is implemented (see Figure 2.12). The waveform takes time to transmit, and may be subject to additional delays due to the path taken (see Section 2.6). Further, this delay is always changing. As a result, a receiver's down-sampler must constantly estimate the timing error, filter that error with a loop filter to avoid jerky changes to compensation, and finally apply a correction to the incoming signal.



(a) Signal without timing error (b) Signal with timing error

Figure 2.12: The in-phase dimension of a pulse-shaped, two symbol (+,-) QPSK transmission (a) and its interpolated, Blackman Harris filtered realization. Additionally, the signal is shifted in time due to a transmission delay this time.

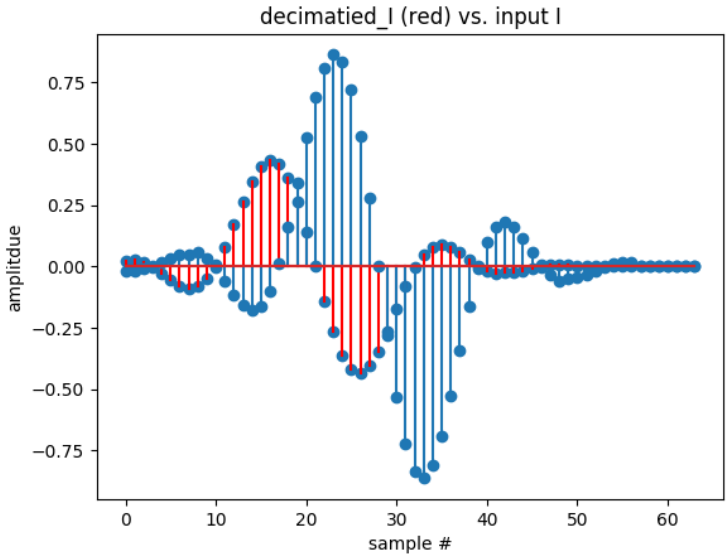


Figure 2.13: A comparison of Figure 2.7.3 (blue) and its time-shifted realization from Figure 2.7.3. The amplitude is reduced due to the amplitudes of a Blackman Harris interpolation filter's coefficient values.

2.7.4 IQ Imbalance when Modulating

The transmit and receive chains of a radio system (see Figure 2.1) contain a similar modulation (see Figure 2.14) and demodulation operations. The magnitude and phase seen by the cosine (in-phase) and sine (quadrature, 90 degrees out of phase) paths of these operations are often not perfectly matched in hardware implementations, resulting in a stretched IQ plot (magnitude imbalance) at the receiver before mapping to bits, or a situation where the I and Q axis are no longer perpendicular due to phase imbalance

(I is not truly 0 degrees, Q is not truly -90 degrees).

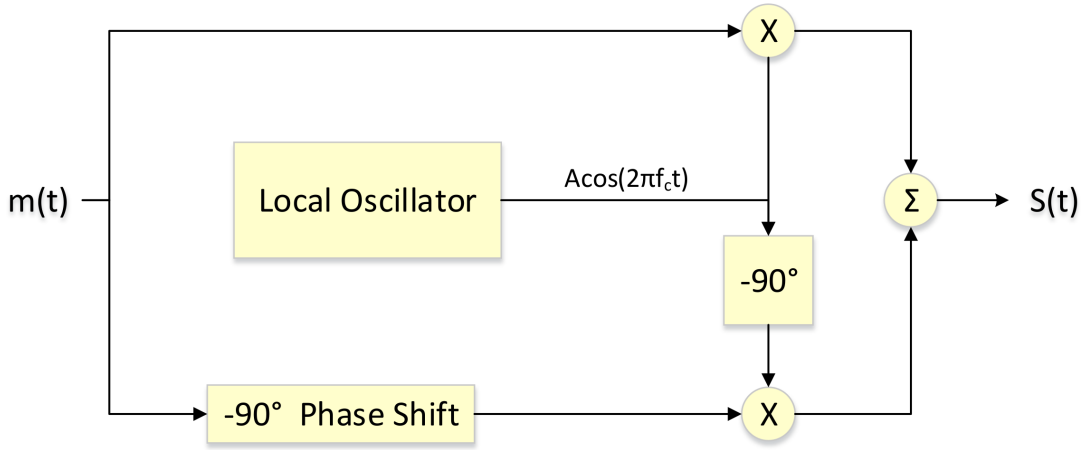


Figure 2.14: An illustration of the in-phase and quadrature paths of the modulator block in a RFFE (see Figure 2.1). The in-phase (top) and quadrature (bottom) signals may not experience the same gains or appropriate phases of 0 and 90 degrees due to manufacturing imperfections or normal wear.

Quality instruments tend to keep this low, although it can vary by large amounts over frequency [29]. IQ imbalance can be modeled at mapping time for each symbol through the expression:

$$sI = kI \times sI', \tag{2.31a}$$

$$sQ = -kQsin(\phi_\epsilon) \times sI' + kQcos(\phi_\epsilon) \times sQ', \tag{2.31b}$$

where sI', sQ' are the balanced in-phase and quadrature components of the symbol, sI, sQ are the IQ imbalanced in-phase and quadrature components of the damaged symbol, kI, kQ are the linear in-phase and quadrature gains, and ϕ_ϵ is the phase difference between the two paths. Notice that for $\phi_\epsilon = 0$ and $kI = kQ = 0$, IQ imbalance is not present, $sI = sI', sQ = sQ'$.

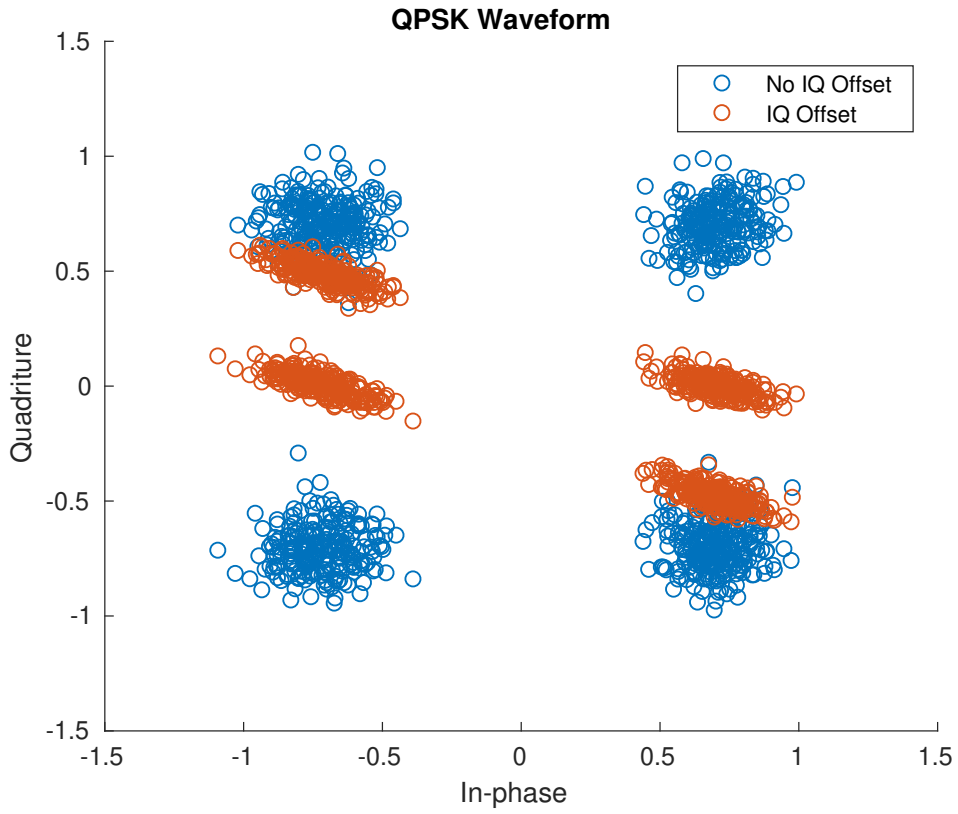


Figure 2.15: An IQ plot (see Appendix A.5) of 1,000 QPSK samples before and after IQ offset (2.31). Values of $\phi_\epsilon = 20^o$, $kI = 1$, and $kQ = 0.7$ are used. Notice how the smaller Q gain causes the spread of values to be vertically squeezed on the Quadrature axis.

2.7.5 Quantization at Filters and DACs/ADCs

A radio system (see Figure 2.1) experiences numerous forms of discrete approximations of continuous values [30]. Two such examples are the digital approximations of analog waveforms by the ADC, and the floating-point values assigned to derived continuous IIR filter coefficients (16-bit, 32-bit, or 64-bit typically, or 2^b discrete amplitudes). While a received waveform has a continuous value, computers only have so much computational power and must reduce the value of voltages and and filter coefficients to discrete values of so many points of precision (see Figure 2.16).

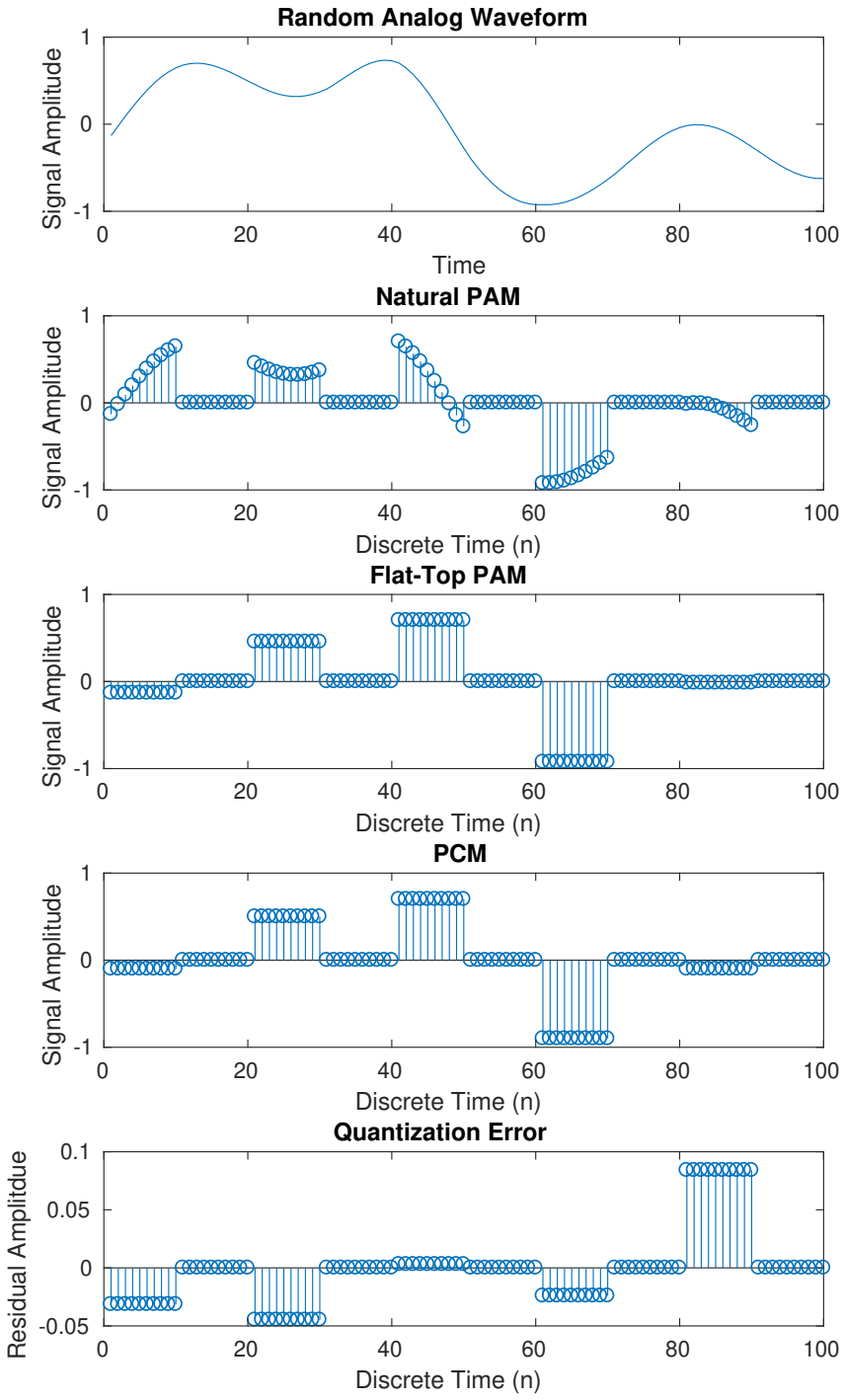


Figure 2.16: A random analog waveform (see Appendix A.6), its natural Pulse Amplitude Modulation (PAM) waveform obtained through multiplication with an impulse train, its flat-top PAM waveform formed by holding the first value of each pulse, its Pulse Code Modulation (PCM) waveform obtained by quantizing to the nearest value in the codebook $[-0.9, -0.7, -0.3, -0.1, 0.1, 0.3, 0.7, 0.9]$, and the residual error resulting from that quantization.

2.7.6 Electronic Noise

Thermal Noise, or Johnson-Nyquist noise, is a form of white noise (see Section 2.1) that results from the agitation brought on by any applied voltage to the charge-carrying electrons inside a radio's conductive components of its RFFE (see Figure 2.1). For a circuit with resistance R in Ohms, a signal of bandwidth Δf in Hertz generates an RMS voltage of:

$$v = \sqrt{4k_B T R \Delta f}, \tag{2.32}$$

where $k_B = 1.38 \times 10^{-23} J/K$ is Boltzmann's constant, and T is the temperature of the circuit in Kelvin. This voltage is white, or applied at all frequencies, adding the power:

$$P_{dBm} = -174 + 10 \log_{10}(\Delta f), \tag{2.33}$$

assuming room temperature, $T = 298.15K$.

The unit used to describe current, Amperes = Coulombs/Sec, describes an average rate of movement of charge past a constant point. Current is a flow of electrons with small random variations in the arrival rates of charge. The error introduced by these variations is what is described as shot noise in electronics, and it is usually insignificant. However, at high frequencies and low temperatures, shot noise can overpower other forms of electronic noise such as thermal noise as the dominating source. A form of white noise, shot noise power begins by modeling electron flow as a Poisson process, ultimately deriving power as [**?**]:

$$P = \frac{1}{2} q I \Delta f R, \tag{2.34}$$

where $q = 1.602 \times 10^{-19} C$ is the charge of an electron, I is the average DC current flowing through the conductor, Δf is the bandwidth of the signal in Hertz, and R is the resistance of the circuit in Ohms.

Flicker Noise, or 1/f noise, occurs in all electronic devices as a low-frequency phenomenon resulting from small changes, or flickers, in temperature changing the resistivity, and ultimately inducing a small voltage, in conductive, current carrying sections of the RFFE [**?**]. In communications, low-frequency noise might not seem like an issue, but local oscillators mix up flicker noise to frequencies close to the carrier, which is called oscillator phase noise. Flicker noise is typically characterized by the corner frequency f_c (typically several kHz, determined by which Field Effect Transistor (FET) the RFFE (see Figure 2.1) uses), below which electronic noise is dominated by flicker noise, and above which is dominated by the various white band noise sources such as thermal and shot noise. As an Infinite Impulse Response (IIR) filter of order N, flicker noise can be modeled by convolving time-domain samples with a filter defined by the numerator coefficient λ and the denominator coefficients γ_i are determined recursively as:

$$\lambda = \sqrt{2\pi f_o 10^{L/10}}, \tag{2.35a}$$

$$\gamma_i = (i - 2.5)\frac{\gamma_{i-1}}{i-1}, \tag{2.35b}$$

where f_o is the frequency offset in Hz, L is the phase noise level in dBc/Hz compared to the carrier power, and the filter coefficient initialization begins with $\gamma_1 = 1$. An IIR filter is defined by these coefficients and input waveform $x[n]$ in the sample domain as:

$$y[n] = \frac{1}{\gamma_1}(\lambda x[n] - \gamma_2 y[n-1] - ... - \gamma_i y[n-i+1]). \tag{2.36}$$

Burst Noise results from the summed voltage errors resulting from Gibb's phenomenon (see Figure 2.17) on several discrete voltage signals changing state simultaneously [**?**]. Gibb's phenomenon is a model used to describe how sharp features in time-domain signals require higher and more frequency components (perfectly square waveforms would require an infinite number of frequencies including frequencies of infinite cycles/Sec), but due to limitations in hardware high frequency components may not be available and small errors can occur at signal edges. If the various triggers, states, and clocks used in RFFE's transition states at the same time, the sum of these Gibb's errors can exceed hundreds of mV.

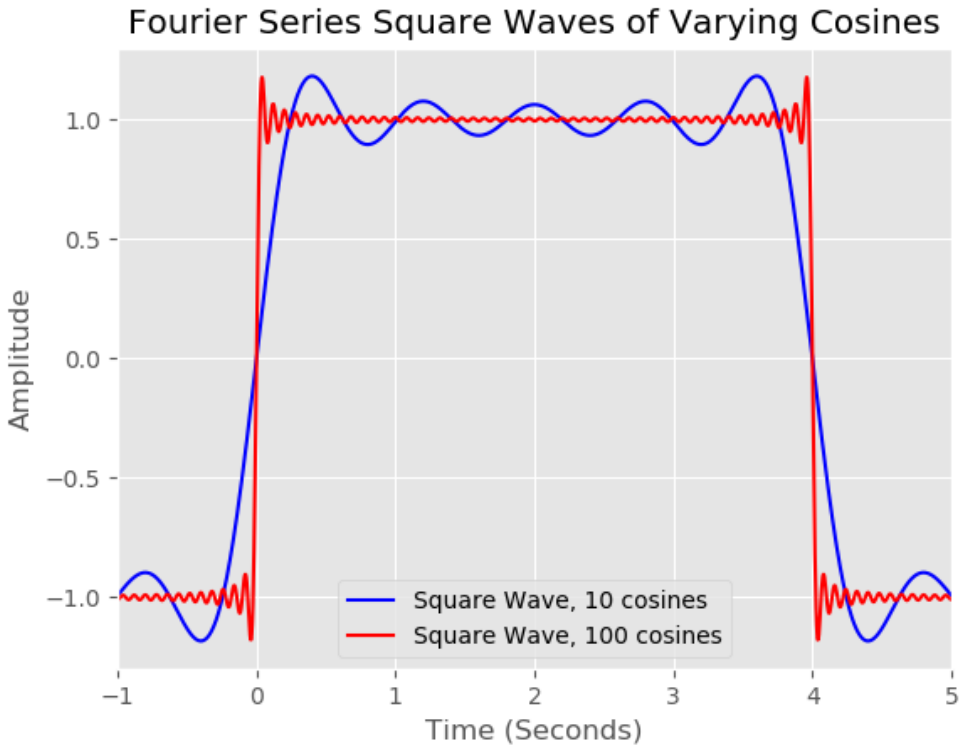


Figure 2.17: A section of a time-domain square waveform formulated by summing cosines. While the states of the square waveform aims to have values of negative and positive one, there are large deviations near high-definition edges, and small ripples in flat sections. If unlucky, these analog deviations can sum to mV values.

Transit-Time Noise is a high frequency noise that occurs in transistors [**?**]. Transistors are gate-like semi-conductors used in a radio's RFFE (see Figure 2.1), consisting of three nodes: a base, collector, and emitter. When the voltage at a transistor's base is large enough, current is allowed to flow from emitter to

base, functioning as a switch. When this transit time from emitter to base is long compared to the period of the Alternating Current (AC) signal, issues arise in the form of transit-time noise, as the transistor is no longer being operated in its designed range. The noise will increase with frequency as the signal period becomes shorter compared to transit-time.

2.8 Coupled Noise

While Section 2.7 describes models for noise generated by the radio itself, this section describes popular models for noise coupled into the transmission by the environment.

Inter-modulation is the creation of copies of waveforms at unwanted frequencies when multiple signals modulated by different carrier waveforms (see Section 2.7.1) share the same non-linear wireless channel (see Figure 2.18). In the example of two sinusoids of frequencies F_1, F_2, their sum can be described as:

$$x = sin(2\pi F_1 t) + sin(2\pi F_2 t) \tag{2.37}$$

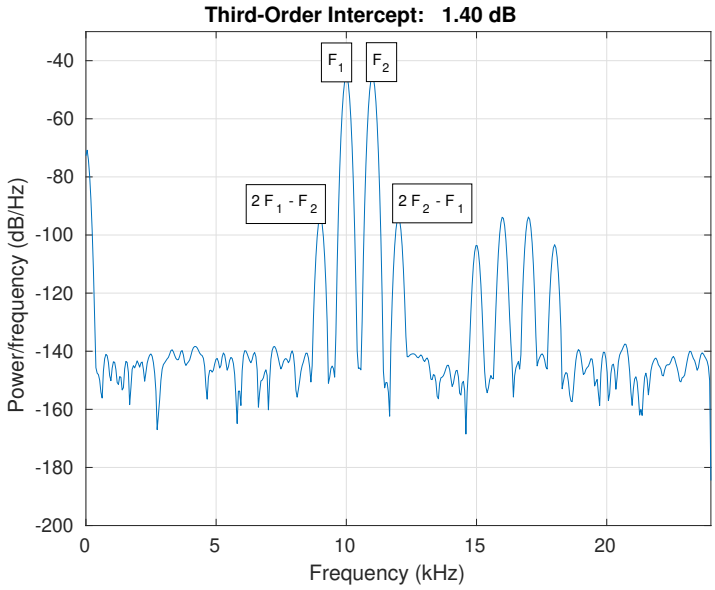


Figure 2.18: A periodogram (see Appendix A.2) calculated using a Kaiser window displaying the non-linear sum of two sinusoids ($F_1 = 10kHz, F_2 = 11kHz$). The sum is made non-linear by evaluating each time-domain sample of the sum through the polynomial $y = 0.0005x^3 + 0.0000001x^2 + 0.1x + 0.003$.

Crosstalk is the undesired interference in one allocation of a wireless channel from a usually neighboring allocation. Commonly this occurs in the form of transmissions near the edge of a frequency band (see Figure 2.19).

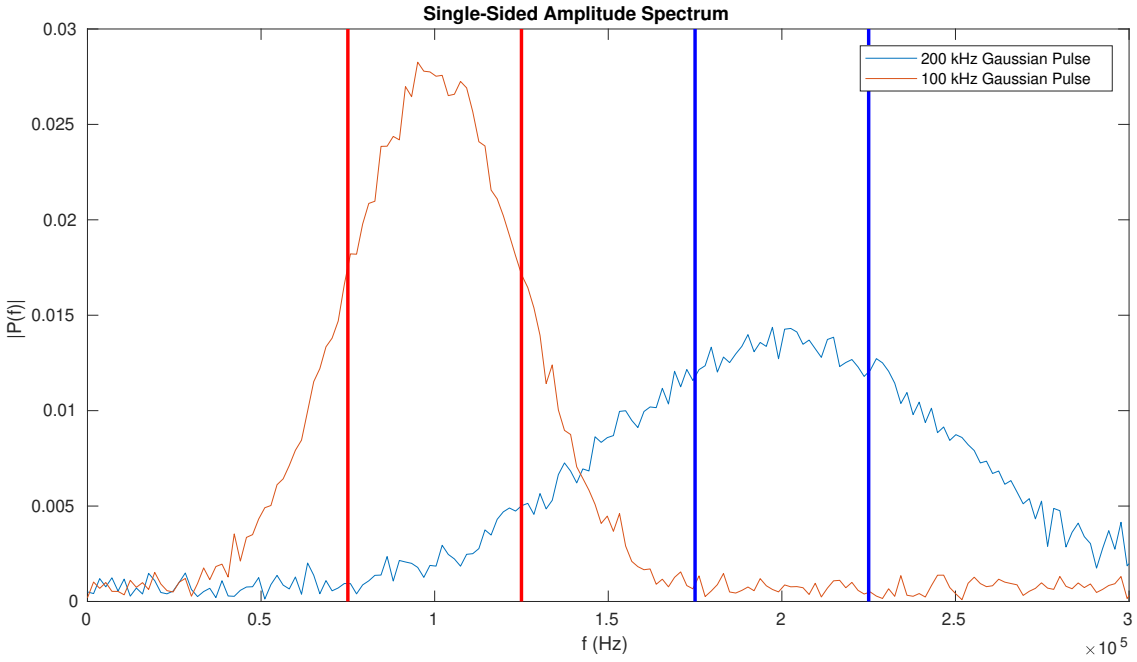


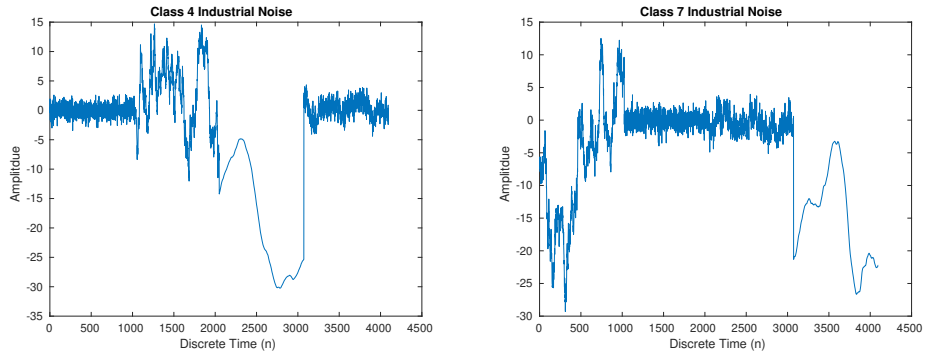
Figure 2.19: Two Gaussian pulses (see Appendix A.3) displayed in the frequency domain. If guard bands (displayed here as 125-175 kHz) are not used or bandwidth (100 kHz in this example) is not carefully allocated, interference can result from neighboring channels (displayed as vertical lines), as shown here.

Atmospheric noise is caused by an average of 3.5 million lightning flashes per day. Cloud to cloud strikes are typically weaker and can be observed at a radio receiver as white noise (see Section 2.1) [31], while cloud to ground strikes are stronger and can be seen as impulse noise (see tables of empirically derived coefficients in CCIR 322 [31]).

Modern urban settings are populated by a host of electronics and machinery, all of which generate electromagnetic emissions. Automobiles, aircraft, rotating machines such as alternators and engines, power distribution systems, and lights are examples of such electronics and machinery. The power spectra of the sum of these emissions can be described as [32]:

$$P(f) \propto \frac{1}{f^{\beta}}, \tag{2.38}$$

where f is the frequency in cycles and β is the fractional exponent. When $\beta = 0$, the power spectrum is uniform, or AWGN (see Section 2.1). Exponent values of $\beta = 1, 2, 3$ are expressive of pink, brown, and black noise, although any value $0 < \beta < \infty$ can be used in this model. Industrial noise can be modeled [32] as a periodic permutation over time of various spectra (see Figure 2.8).



(a) A class 4 industrial noise sequence, modeled as in (2.38) as white, brown, black, then pink noise.

(b) A class 7 industrial noise sequence, modeled as brown, white, pink, then black noise.

Figure 2.20: Class 4 (a) and class 7 (b) industrial noise sequences (see Appendix A.4), each period lasting 1024 samples. This model is reflective of the time-varying nature of industrial noise, as the authors of [32] found different power spectra dominate over others for often periodic time intervals.

There are numerous electromagnetic sources throughout the universe whose emissions travel very well through the vacuum of space and disturb radio transmissions. Cosmic noise is a categorical term that embodies various forms of noise, most commonly encountered at frequencies above 30 MHz [33]:

- Receivers pointed towards nearby stars such as our sun, super massive black holes at the center of galaxies, and quasars.
- Charged particles and meteorites that fall into the earth's orbit are deflected by the fundamental electro-mechanical Lorentz force due to the earth's magnetic field. This process produces Synchrotron radiation [33] of emitted power $P \sim m^{-4}$, which results in most all radiation resulting from electrons and positrons due to their small mass. Synchrotron radiation is phase coherent for showers of large side surface area smaller than wavelengths emitted. Pulse amplitude is then defined [33] as:

$$A \sim E_p e^{-\frac{r}{r_0}}, \tag{2.39}$$

  where E_p is the energy in Joules of the primary particle, r is the distance to the shower core, and r_0 is an experimentally determined constant.
- Super wide band noise generated by Cosmic Microwave Background Radiation (CMBR), a form of radiation persisting from the big bang, which is present homogeneously throughout the known universe (see Figure 2 of [34] for a plot of the intensity of the CMBR over frequency).

2.9 Chapter Summary

In this chapter, various classical models for transmission imperfections and channel models were surveyed, including AWGN, link budgets, ray models approximating reflections, diffractions and scattering, Doppler shift, RFFE errors, electronic noise, and coupled noise. These channel environments serve as options for arbitrary channel models for use in the proposed framework in Chapter 4. In Chapter 3, we will survey machine-learning based signal classifiers.

Chapter 3

Understanding Machine-Learning Based Signal Classifiers

In order to better understand why test data perturbations unforeseen in the training data decrease signal classification accuracy, this chapter communicates the basic concepts of linear classifiers (see Section 3.1), Convolutional Neural Network (CNN) classifiers leveraging non-linear activation functions, and how to design a CNN architecture (see Section 3.2). Furthermore, fields of generalized (see Section 3.5 and Section 3.7) and robust training are presented (see Section 3.4) to further discuss the state-of-the-art and provide background knowledge needed for Chapter 5.

3.1 Linear Classifiers

A linear classifier [2] is the simplest form of supervised machine learning (meaning the classifier is taught to classify using labeled examples, where as in unsupervised learning no such examples are available). A linear classifier is comprised of two main parts: a score function that reduces raw data to K class scores (where K is the number of class categories), and a loss function, which is a metric that describes how closely a training signal's class scores match the ground truth.

The word neural in the term neural network is inspired by an analogy bridging the worlds of math and biology. The basic computational cell of brain is the neuron and its mathematical model (see Figure 3.1) is very rough. In reality, dendrites perform non-linear, time-varying operations on signals coming in from axons [2] rather than the multiplication of a scalar, as seen in the mathematical model. Furthermore, there are many types of neurons, and a very important aspect of their behavior that is ignored in machine learning is the timing of their axon firings. A linear classifier, as its name implies, does not make use of an activation function, and each neuron simply computes a biased dot product of its inputs.

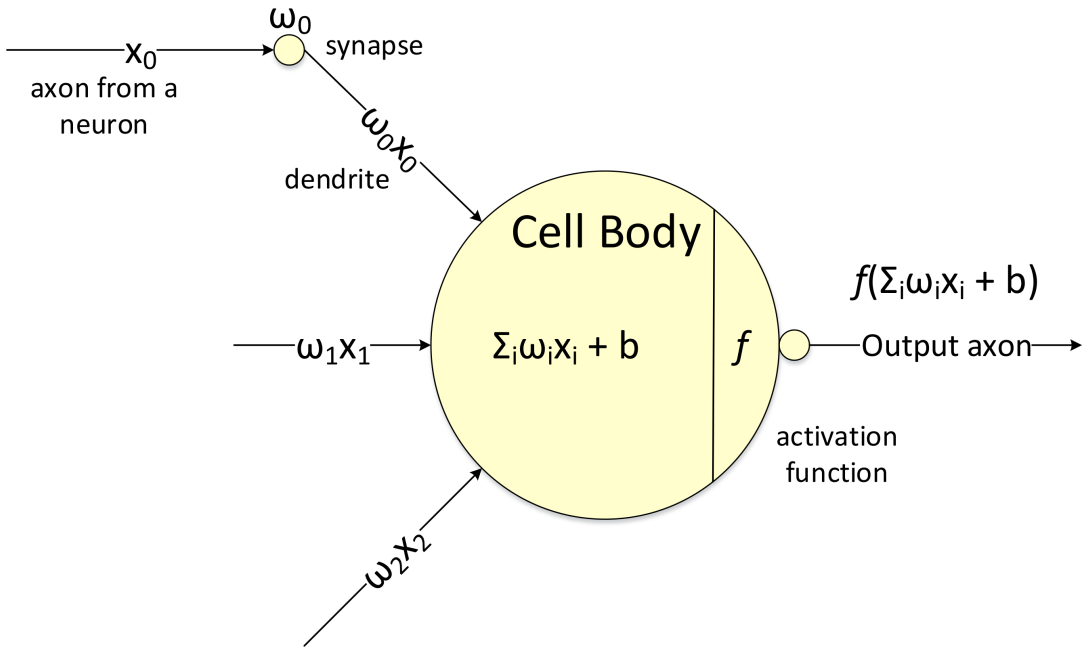


Figure 3.1: A neuron cell (adapted from [2]) is composed of a nucleus which receives signals from many dendrites. The amount of influence a dendrite has on a neuron is determined by synapses. When the sum of incoming signals is above a threshold, the nucleus fires a signal down its axon, which in turn splits into many dendrites, feeding into other neurons. In this mathematical model inspired by this phenomenon, the previous neurons' axons carry the signals x_0, x_1, x_2, split into many dendrites. Synapses influence that value by a weight, (w_0, w_1, w_2). The cell body adds weights to each incoming dendrite (b_0, b_1, b_2), computes the dot product of all dendrites, and outputs a signal on its axon defined as the output of some activation function f whose input is the dot product.

The Multi-class Support Vector Machine [2] (SVM) is a commonly used loss function that wants the correct class for each signal to have a score higher than the wrong ones by a margin of Δ. The SVM is defined as:

$$L_i = \sum_{j \neq y_i} max(0, S_j - S_{y_i} + \Delta), \tag{3.1}$$

where S_{y_i} is the highest class score and S_j is the set of all other scores. As an example, if the ground truth for the ith signal used to train a three-class linear classifier is $y_i = 0 \neq 1 \neq 2$, the margin $\Delta = 10$, and the class scores $S = [13, -7, 11]$:

$$L_i = max(0, S_1 - S_0 + \Delta) + max(0, S_2 - S_0 + \Delta), \tag{3.2a}$$

$$L_i = max(0, -7 - 13 + 10) + max(0, 11 - 13 + 10), \tag{3.2b}$$

$$L_i = max(0, -10), max(0, 8), \tag{3.2c}$$

$$L_i = 8. \tag{3.2d}$$

The $j = 1$ term contributes no loss to L_i because the correct class score $S_0 = 13$ was more than $\Delta = 10$ larger than the incorrect score $S_1 = -7$. The $j = 2$ term was not, so the loss score was increased by how

much the margin was missed by, eight.

A commonly used reduction for the dot product of terms $Wx_i + b$ is to combine W, b into one matrix [2]. This is done by adding a unit value to the end of each flattened row vector signal x_i as shown in Figure 3.2.

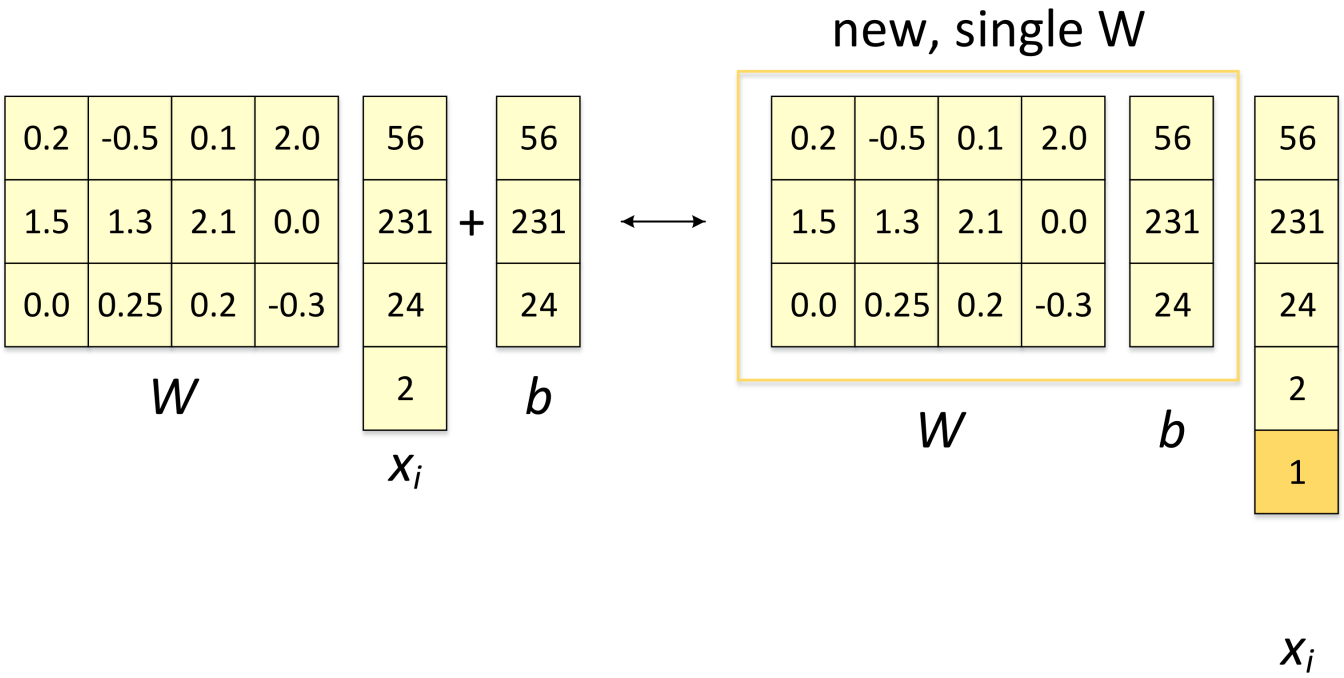


Figure 3.2: An illustration (adapted from [2]) of reducing $W \in [3, 4]$ and $b \in [3, 1]$ into a single matrix, $W \in [3, 5]$ by adding a unit value to the end of $x_i \in [5, 1]$.

For a single-layer Linear Classifier (input is directly connected to neurons which are directly used to calculate class scores) and linear score function $S = f(x_i, w)$, SVM can be vectorized and written as:

$$L_i = \sum_{j \neq y_i} max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta), \tag{3.3}$$

where each of the N training signals $x_i, i = 0, 1, ..., N-1$ are the flattened row vectors of two-dimensional ($D = 2$) signals, where each element represents an alternating in-phase and quadrature (IQ) sample. All flattened signals together make up the matrix of samples $x \in [N, k \times D]$.

Suppose we are given a new set of scores $S = [13, -7, -5]$, such that now in (3.2) we achieve $L_i = 0$. If we set all weights $W = \lambda W$, for $\lambda = 3$, through (3.3) we would achieve in (3.2) $L_i = max(0, (-7-13)\lambda + 10) + max(0, (-5-13)\lambda + 10) = 0$. For any $\lambda > 1$, in fact, the loss function would remain at zero. To keep weights minimal and remove this ambiguity, the SVM employs a regularization penalty [2], defined as:

$$R(w) = \sum_k \sum_l w_{k,l}^2, \tag{3.4}$$

such that the SVM is now defined as:

$$L_i = \sum_{j \neq y_i} max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta) + \lambda \sum_k \sum_l w_{k,l}^2. \tag{3.5}$$

This way, no single input dimension of a training example x_i can have a dominating impact on all K scores by itself. For example, the unit amplitude, two-symbol, two-SPS Quadrature Phase Shift Keying (QPSK) signal $x_i = [1, 1, 1, 1]$ and two weight vectors $w_0 = [1, 0, 0, 0]$ and $w_1 = [0.25, 0.25, 0.25, 0.25]$ result in the same dot product $w_0^T x_i = w_1^T x_i = 1$, but the regularization penalty for $R(w_0) = 1^2 + 0^2 + 0^2 + 0^2 = 1 > R(w_1) = 0.25^2 + 0.25^2 + 0.25^2 + 0.25^2 = 0.25$. As a result, w_1 would achieve the lower regularization loss $\lambda R(w)$ and total loss L_i because the influence of its weights are more evenly distributed across both the I and Q components of both the first and second symbol of the transmission x_i. Using w_0, all influence is given to the in-phase component of the first symbol. In [2], it is suggested that Δ, λ can safely be set to one in all cases, and can be tuned together as a single hyper-parameter (a concept covered later in this chapter).

The other popular loss function, besides SVM, is the soft-max function [2] defined as $f_j(z)$ used in the loss function L_i:

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}, \tag{3.6a}$$

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right), \tag{3.6b}$$

where f_j is the j-th element of the K class scores $f_j, j = 0, 1, ..., K-1$. This function can be though of as the normalized probability that the label y_i is correct given the signal x_i parameterized by weights matrix W:

$$P(y_i, x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}. \tag{3.7}$$

The function is analogous to a probability in the sense that each soft-max rating $0 \leq f_j \leq 1$ and $\sum_j f_j = 1$. A three-class modulation linear classifier soft-max vector $f = [0.2, 0.2, 0.6]$, for instance, may represent that the classifier believes that the test-time signal given to it is 60% likely to be QPSK modulated IQ data, 20% likely to be BPSK modulated, and 20% likely to be QAM-16 modulated.

A comparison between SVM and soft-max is displayed in Figure 3.3. In practice, the performance of both loss functions is near equal [2]. However, one can train faster or classify more accurately than the other, given the right circumstances. Since the SVM operates properly when an incorrect score is more than a margin lower than the correct score, time is not wasted training a linear classifier to decide between two easily distinguishable labels. The soft-max classifier, on the other hand, always benefits from a lower score for incorrect labels, and a higher score from the correct labels.

In this section so far, its been mentioned that λ, Δ are defined as hyper-parameters. Throughout Section 3 this term will be used to describe constants used in various machine learning tasks that might

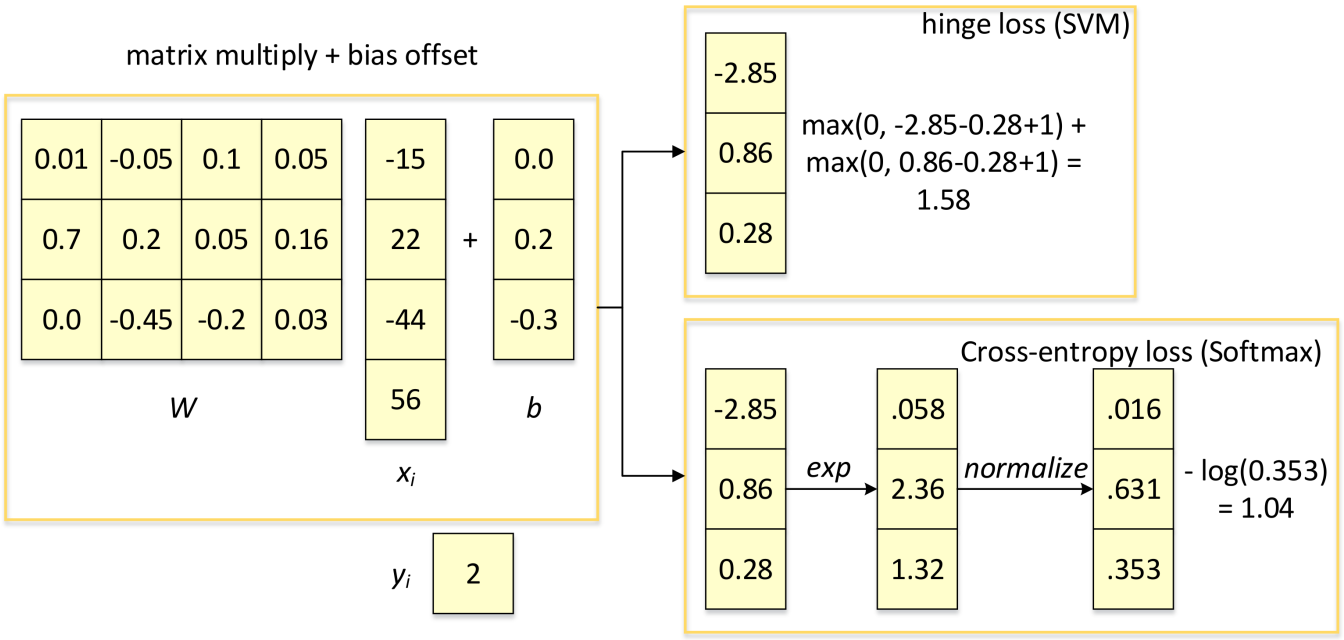


Figure 3.3: An illustration (adapted from [2]) of the computation of class cores f and the resulting loss score L_i using both SVM and soft-max functions. Both use the same class scores f, but have very different interpretations of their results, 1.58 and 1.04. The SVM considers each incorrect score less than a margin below the correct score as a contributor to loss, while the soft-max classifier relays a value proportional to the belief that the label assigned to each signal is correct.

not have clear values. The way researchers typically tune hyper-parameters is through a data-driven approach known as validation [2].

Validation is a simple concept: try out different values for each hyper-parameter and see what works best. An important detail here is that evaluation-time signal data cannot be used to tune hyper-parameters. The information contained in test data should never be used until evaluation time for any reason, as the fundamental assumption of machine learning is that decisions are made through learning from training data. This is called over-fitting the classifier, and typically results in poor performance if the test signals are switched.

Consider the code below as Python pseudo-code for a validation example. For $lambda = [1, 2, 5, 10, 100, 500, 1000]$, we find out how good out linear classifier LC is at determining the modulation scheme of each training signal in Xtr_rows. Instead of using all 10,000 signals to train, we take 500 of them for this purpose, and determine which value of λ nets the highest classification accuracy, using that value at evaluation-time.

```
import numpy as np
# assume we have Xtr_rows, Ytr, Xte_rows, Yte as the flattened signals and
# their associated modulation labels
# Say that Xtr_rows is our training data: a 10,000 x 3200 matrix,
# representing 10,000 transmissions of 1,600 alternating IQ samples
Xval_rows = Xtr_rows[:500, :] # take first 500 signals for validation
Yval = Ytr[:500]
Xtr_rows = Xtr_rows[500:, :] # keep last 9,500 for training
Ytr = Ytr[500:]

# find hyperparameters that work best on the validation set
validation_accuracies = []
for lambda in [1, 2, 5, 10, 100, 500, 1000]:

  # use a particular value of lambda and evaluation on validation data
  LC = LinearClassifier()
  LC.train(Xtr_rows, Ytr)
  # here we assume a modified LinearClassifier class that can take a lambda
  # as input for its SVM
  Yval_predict = LC.predict(Xval_rows, lambda = lambda)
  acc = np.mean(Yval_predict == Yval)
  print 'accuracy: %f' % (acc,)

  # keep track of what works on the validation set
  validation_accuracies.append((lambda, acc))
```

As the number of signals used in validation, the number of hyper-parameters, and the number of values swept over for validation increase, it should become apparent that validation quickly becomes a highly costly endeavor. If the amount of training data available is small, it becomes hard to trust the outcomes of validation sweeps, and the even more costly cross-validation technique is often implemented, if needed. The goal of cross-validation is to split up training data into folds, performing validation once for each fold, where that fold acts as the current validation data. Referencing the Python pseudo-code above, a 4-fold cross-validation would perform four validations: first with signals 1 through 2,500 as validation data and signals 2,501 through 10,000 as training data, second with signals 2,501 through 5,000 as validation data and signals 1 through 2,500 and 5,001 through 10,000 as training data, and so on (see Figure 3.4). The classification accuracy for each lambda value is then averaged for all four folds, and the value of lambda

with the highest average is selected as the best to use at evaluation-time.

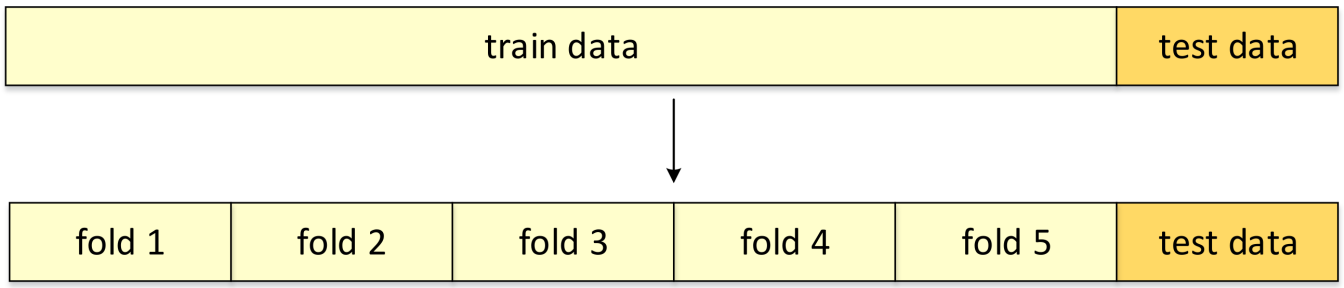


Figure 3.4: An illustration (adapted from [2]) of common data splits between training, validation, and testing data. In this image, validation is performed on fold 5, training on folds 1-4, and testing on the rest. Next, fold 1 would be used as validation data, and folds 2-5 as training data, and so on, until all 5 folds have been used as the validation data.

In this section, it has so far been discussed how the loss function L_i evaluates the quality of the current weights W used in a linear classifier. Using that information, how can the weights be improved to better perform at supervised learning? Consider the example loss function in Figure 3.5, where dark blue represents low loss and red represents high loss. The axes represent varying weight values for two weights, although in practice the number of weights would take the dimensions of this plot higher than what can be visualized. Computing the gradient of that landscape would return a vector pointing in the most blue direction, representing how those two weights should be changed to most significantly reduce the loss function used.

Computing the gradient involves solving for many derivatives. This can either be done analytically (a slow but accurate method) or numerically (a faster but less accurate method). In practice, analytic gradients typically take too long to derive [2], and numerical derivatives have been found to be the most accurate when calculated using the centered difference formula:

$$\frac{\delta f}{\delta x} = [f(x+h) - f(x-h)]/2h, \tag{3.8}$$

where $h << 1$ (typically 1e-5 [2]). In a three-weight example case is used to calculate the gradient:

$$\nabla f(w_0, w_1, w_2) = \frac{\delta f}{\delta w_0}\hat{\boldsymbol{w}}_\mathbf{0} + \frac{\delta f}{\delta w_1}\hat{\boldsymbol{w}}_\mathbf{1} + \frac{\delta f}{\delta w_2}\hat{\boldsymbol{w}}_\mathbf{2}, \tag{3.9}$$

where w_0, w_1, w_2 are weights used and $\hat{\boldsymbol{w}}_\mathbf{0}, \hat{\boldsymbol{w}}_\mathbf{1}, \hat{\boldsymbol{w}}_\mathbf{2}$ are the weights' unit vectors. It is important to note that the gradient is just a vector, and doesn't determine to what magnitude weight updates should occur. As shown in Figure 3.5, there are consequences to updating by too much or too little.

To generalize in the extreme, a linear classifier might find that the weights corresponding to samples surrounding the max and min samples of Figure 3.6 should be increased to minimize its loss function. Giving those weights too much influence, however, may cause the classifier to ignore imperfections or phenomenon

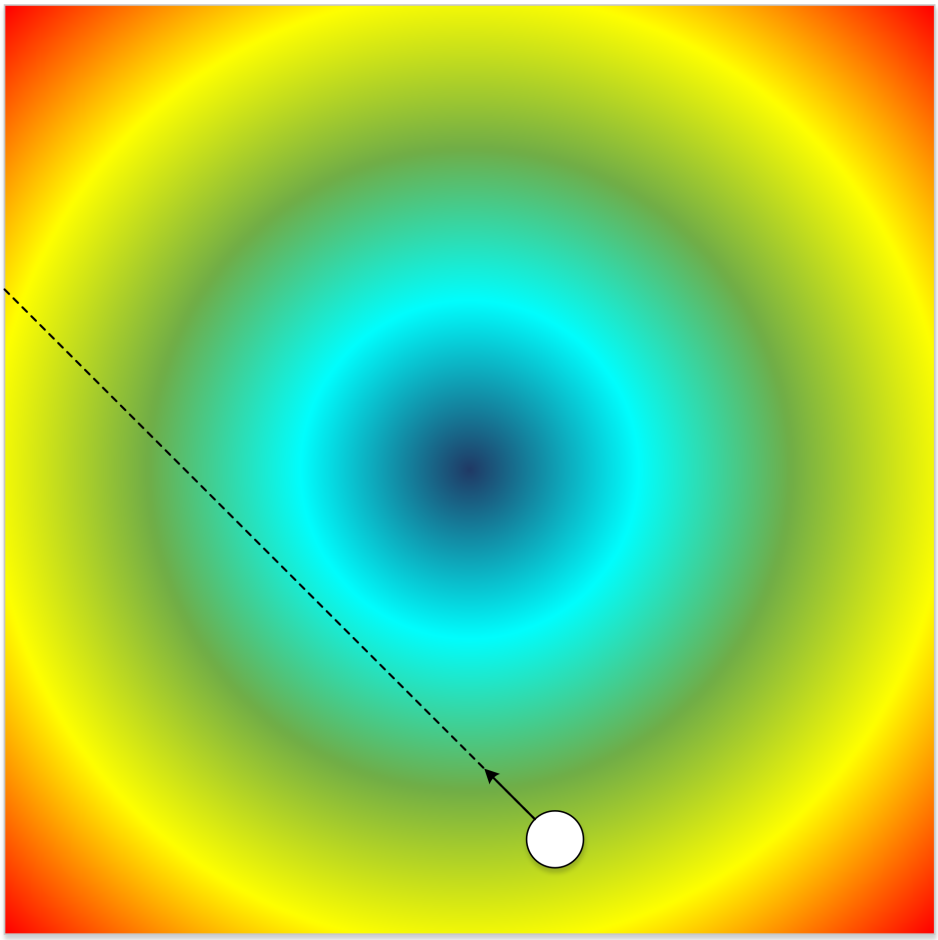


Figure 3.5: An illustration (adapted from [2]) of an SVM loss function's gradient vector (3.9) for a two-weight linear classifier. Each of the two axis represent values assigned to a weight. In practice, loss functions have pockets of local minima/maxima, and cannot be visualized due to the number of dimensions required to represent each weight used. The color gradient red represents high loss, while blue low loss. The white circle represents the current values chosen for weights w_o, w_1, the arrow the gradients unit vector, and the dashed line an extension of that vector. Updating the weights by too much will put the weights (currently green loss) in perhaps a higher loss section of the graph (yellow or red), but adjusting the weights by too little each update will be computationally expensive and perhaps get the SGD stuck in a local minimum of the SVM loss function.

that reveal themselves at other samples in the signal such as zero crossings or the tails trailing or leading the symbols due to convolution with the RRC filter.

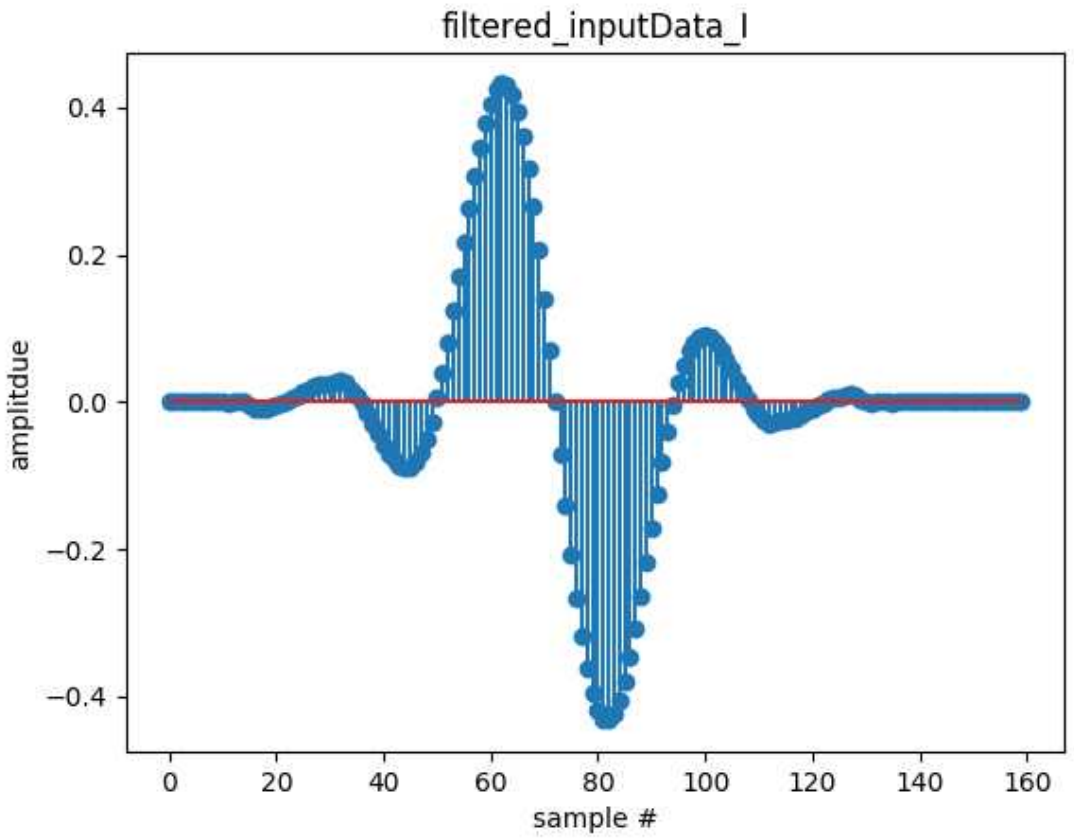


Figure 3.6: The in-phase components of one positive and one negative QPSK symbol, up-sampled to 16 SPS by a Raised-Root Cosine (RRC) filter with a roll-off coefficient of 0.35. Making up half the values of an example flattened training signal vector x_i, classification decisions of a linear classifier using this signal would likely depend most heavily on samples surrounding the 60th and 80th sample, as they most strongly correlate to what bits are being transmitted. As a result, weights corresponding to those samples would likely be pushed to higher values during SGD.

How should it be decided by how much and via what method to update weights? Most methods are iterative, performed by updating weights, calculating the loss function, and adjusting the weights according to an algorithm. These methods are called *Stochastic Gradient Descent* (SGD) because the loss function is being navigated via the gradient in the presence of random variations. The comparison is often made [2] between SGD and a person walking down a mountain blindfolded. The calculation of a gradient is sometimes called doing a back-pass through the neural network, as the process begins at the classification neurons and works backwards, while computing the loss function for the current SGD iteration is called a forward-pass since the NN is traversed from the input layers to the classification ones.

Vanilla updates are the simplest choice, where the weights matrix W is changed after each gradient calculation according to:

$$W \mathrel{+}= \eta \nabla W, \tag{3.10}$$

where the hyper-parameter η is the learning rate or step size, and ∇W is the current gradient computed on weights W. Typically weight updates are halted by automated processes parameterized by patience and convergence values. If the classifier's loss function converges to within a certain percentage difference of the last loss value, supervised training will end after a patience period after some amount of weight updates have passed without exceeding the convergence threshold. The purpose of the patience period is to give natural variations in the loss function a chance to overcome a local minimum.

Momentum updates almost always result in faster convergence [2], and can often avoid the pitfall of getting trapped in local minima that vanilla updates can be susceptible to. Think of momentum updates as a ball in 3D space rolling down a hill side. In this interpretation of updates, the current weight values W can be thought of as the current position in space, and the update a velocity value, displacing the weights:

$$v_{(i)} = \mu v_{(i-1)} - \eta \nabla W, \tag{3.11a}$$

$$W \mathrel{+}= v_{(i)}, \tag{3.11b}$$

where the velocity is initialized at zero as $v_{-1} = 0$, and momentum μ is a tunable hyper-parameter, typically 0.9 in practice [2], $v_{(i)}$ is the current velocity value, and $v_{(i-1)}$ is the previous back pass velocity value.

Nesterov Momentum [35] performs slightly better, evaluating the gradient at the weights' location where the momentum step μ has carried the values, rather than the current position. For the ith SGD back pass, the Nesterov Momentum update on W is defined as:

$$v_{(i)} = \mu v_{(i-1)} - \eta \nabla W, \tag{3.12a}$$

$$W \mathrel{+}= (1 + \mu) v_{(i)} - \mu v_{(i-1)}. \tag{3.12b}$$

Adagrad [36] is another method that caches updates, where the updates are defined as:

$$cache = (\nabla W)^2, \tag{3.13a}$$

$$W \mathrel{+}= -\frac{\eta \nabla W}{cache + h}, \tag{3.13b}$$

where $h = 0^+$ to avoid divide by zero errors. While adagrad begins fast, the aggressive update term $\frac{\eta \nabla W}{cache + h}$ often stops learning too early [2] without proper annealing (covered later in this section).

RMSprop [37] or Root Mean Square (RMS) propagation begins by looking at the signs of the last two gradients for the weight. If they are the same sign, that means parameter updates are on the right course, and learning rate η should be accelerated by a small increase factor IF. If different, the weight update was too large and jumped over a local minimum, and the learning rate should be made smaller by a moderate decrease factor DF. Next, the learning rate should be increased or reduced such that it stays between a

range of limits specified at the start of learning. Then, the update is applied:

$$\eta_{(i)} = \begin{cases} IF \times \eta_{(i-1)} & \text{if } \nabla W > 0 \\ DF \times \eta_{(i-1)} & \text{if } \nabla W < 0 \end{cases}, \tag{3.14a}$$

$$W \mathrel{+}= \eta_{(i)} \nabla W. \tag{3.14b}$$

Adam [38] is a recent and recommended [2] parameter update algorithm that performs better than the other parameter updates defined so far. Adam updates utilize RMSprop combined with adagrad, defined as:

$$m = \beta_1 m + (1 - \beta_1) \nabla W, \tag{3.15a}$$

$$mt = \frac{m}{1 - \beta_1^t}, \tag{3.15b}$$

$$v = \beta_2 v + (1 - \beta_2)(\nabla W)^2, \tag{3.15c}$$

$$vt = \frac{v}{1 - \beta_2^t}, \tag{3.15d}$$

$$W \mathrel{+}= \frac{-\eta \times mt}{\sqrt{vt} + h}. \tag{3.15e}$$

where m, v are smoothed and squared smoothed gradients ∇W, mt, vt are bias correction mechanisms which account for the fact that $m, v = 0$ at initialization, and h is used as in [36] to avoid divide by zero errors. The constants β_1, β_2 are tunable hyper-parameters.

Annealing the learning rate η is typically to speed up SGD, as the first few back passes typically require large changes [2] to the weights W. Below are the three common methods of annealing.

Step decay is the simplest method, reducing the learning rate every few back passes by a factor α:

$$\eta_{(i)} = \begin{cases} \eta_{(i-1)} - \alpha & \text{if } i \bmod s \end{cases}, \tag{3.16}$$

where $\alpha \in [0, 1]$. This decay is piece-wise, applied at every s multiple of the current back-pass i. A good way to tell when and by how much to reduce η is to look at the loss at each back pass. If the loss is not reducing, that may be a sign that the weights are bouncing around a minimum, overshooting each time. Reducing the learning rate at this point would allow the loss function to fall into that minimum.

Another method is exponential decay [2], which is continuous (applied at every back-pass), and decays such that at back-pass i, learning rate is:

$$\eta_{(i)} = \eta_{(i-1)} - \alpha_0 e^{-ki}, \tag{3.17}$$

decaying more rapidly than (3.16), where the values α_0 and k are hyper-parameters.

Inverse decay is a third method of annealing which decays η more slowly than exponential decay expressed in (3.17) but faster than step decay expressed in (3.16), defined by:

$$\eta_{(i)} = \eta_{(i-1)} - \alpha_0 / (1 + ki). \tag{3.18}$$

Computing gradients in a linear classifier can be thought of as flowing backwards, or back-propagating, through the neurons. Since neurons (see Figure 3.1) mostly interact with their inputs through multiplication and addition, circuit graphics (see Figure 3.7) are often used to visualize SGD.

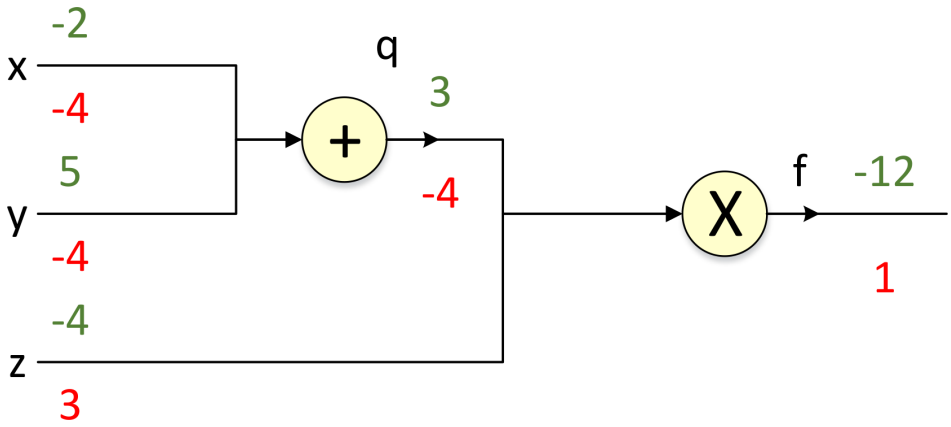


Figure 3.7: A circuit model (adapted from [2]) showing the forward pass (green) by applying inputs to the gates operators and backward pass (red) by applying the chain rule recursively. Gates represent a few local operations done by a linear classifier's neurons. Gates can do both passes totally independent of other gates, without knowledge of the full circuit, or classifier structure.

Beginning with a forward pass, consider the weights $x = -2, y = 5, z = -4$ and states $q = x + y = 5 - 2 = 3$ and $f = z \times q = -4 \times 3 = -12$. Using (3.9), the backward pass unit start value $\frac{\delta f}{\delta f} = 1$ would be changed by $\nabla f = \left[\frac{\delta f}{\delta q}, \frac{\delta f}{\delta z}\right] = [z, q] = [-4, 3]$, and $\nabla q = \left[\frac{\delta q}{\delta x}, \frac{\delta q}{\delta y}\right] = [1, 1]$. The resultant back pass values are then $x = 1 \times \frac{\delta f}{\delta q} \times \frac{\delta q}{\delta x} = -4, y = 1 \times \frac{\delta f}{\delta q} \times \frac{\delta q}{\delta y} = -4, z = 1 \times \frac{\delta f}{\delta z} = 3$, which would now be used as the new forward pass values. This process is very powerful due to its independence.

In Figure 3.8, forward passes through circuit models were done using weight values in green. What values should be used on the first forward pass? One might first want to try all zero values, however this would result in all gates computing zero value outputs, which would result in all gates computing the same gradient ∇W, and the same updates using (3.15). Consequently, all weights would be the same, all neurons equally influencing classification.

The next thought might be to perform symmetry breaking by setting all weights $w \sim \mathcal{N}(0, 1)$ such that each neuron has unique updates and more importance (higher weight values) can be associated with some signal samples than others. However, it should be noted that the unit variance $\sigma^2 = 1$ is not preserved throughout the classifier, as the variance grows with each neuron's dot product $s = \sum_i^n w_i x_i$ (see

Figure 3.1):

$$\text{Var}(s) = \text{Var}(\sum_{i}^{n} w_i x_i), \tag{3.19a}$$

$$= \sum_{i}^{n} \text{Var}(w_i x_i), \tag{3.19b}$$

$$= \sum_{i}^{n} [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i)\text{Var}(w_i). \tag{3.19c}$$

Consequently, in [2, 39] it is suggested for ReLU (3.22) classifiers to initialize each neurons' weights as $W_0 \sim \mathcal{N}(0, \sqrt{2/n})$ where n is the number of weights in that neuron. For zero-mean activation function classifiers such as Tanh (3.21), $W_0 \sim \mathcal{N}(0, \sqrt{1/n})$ is suggested [2].

While addition and multiplication gates cover both operations involved in a dot product (see Figure 3.1), an activation function must also be traversed in SGD in *Non-Linear Classifiers*, which will be referred to synonymously with Neural Networks (NNs) from now on. The first popular non-linear activation function is the sigmoid, defined as:

$$\sigma(x) = 1/(1 + e^{-x}), \tag{3.20}$$

which was historically popular because it serves as a good metaphor for the firing of a neuron [2], having outputs ranging from not firing at $\sigma(x) = 0$ to saturated firing at max frequency, $\sigma(x) = 1$. A significant issue with (3.20) is that during back passes of SGD, saturated regions produce a gradient of near zero using (3.8). This keeps all but the largest of signals from flowing through neurons. Another issue is that (3.8) is positive for all x. If this is the case, the gradient on the weights w during back passes will always be all positive or all negative, causing jerky zig-zag weight updates that can make it difficult for the loss function to converge to a minimum (see Figure 3.5).

The *tanh* non-linear function solves the all positive issue [2] in (3.20) by zero-centering the equation:

$$\tanh(x) = 2\sigma(2x) - 1. \tag{3.21}$$

However, (3.21) still saturates to -1 and 1, resulting in gradient-killing back passes. With the improvement of zero-centering, (3.21) is always preferred to (3.20).

A very common activation function is the Rectified Linear Unit (ReLU) function [2]:

$$f(x) = \max(0, x). \tag{3.22}$$

The reasons for ReLU's popularity are that it was found to make for very fast [40] weight convergence through SGD, and is computationally cheap. However, one should take note that the gradient (3.9) of (3.22) is computed as:

$$\frac{\delta f}{\delta x}(y) = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases} \tag{3.23}$$

Notice how this gate routes all the gradient to the larger input. The implications of this can be seen in Figure 3.8.

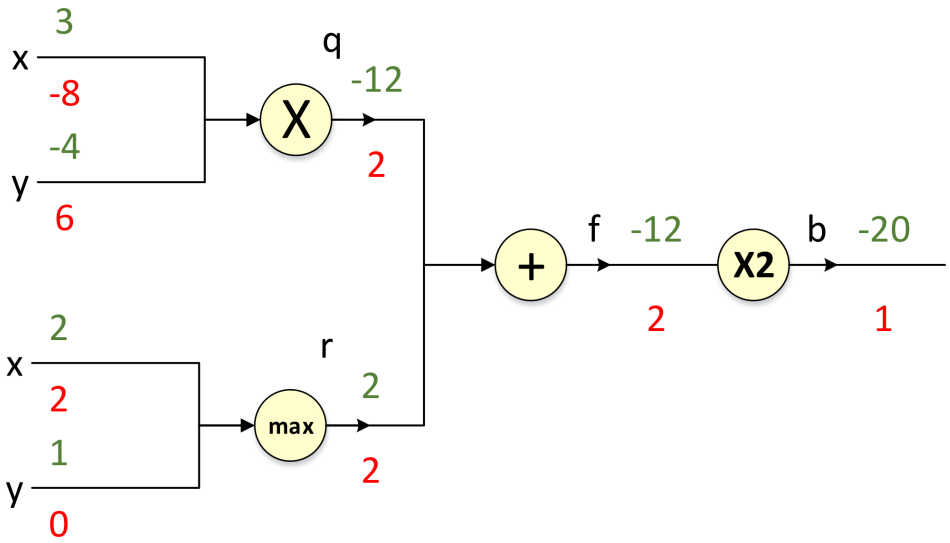


Figure 3.8: A circuit model (adapted from [2]) showing the forward pass (green) by applying inputs to the gates operators and backward pass (red) by applying the chain rule recursively of a circuit featuring a ReLU max gate. The blacked out w weight would cause all gates before it to have a gradient of zero, killing those neurons. Using (3.9), the back pass value for w can be shown to be $w = 1 \times \frac{\delta}{\delta a} 2a \times \frac{\delta}{\delta r}(r+p) \times \frac{\delta}{\delta w} \max(w, z) = 1 \times 2 \times 1 \times 0 = 0$

Leaky ReLU attempts to fix the blackout issue [2] that (3.22) has by giving a small negative slope α for values $x < 0$:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases} \tag{3.24}$$

While this sometimes prevents blackouts, other times it does not, and many are unsure why it works when it does and why it does not when it does not [2]. The leaky slope α can be parametrized and tuned to optimize a classifier (called PReLU) as seen in [41], although the reason why this is of benefit is still unclear.

Most researchers use (3.22) in practice, and if a large number of neurons are dying during SGD, leaky ReLU and PReLU are often used.

Signal data does not always come in a form that is friendly for machine learning. Sometimes one or several operations must be performed on IQ, time domain, or frequency domain data to prevent certain errors from occurring in training or testing a linear classifier. As mentioned in (3.21), zero-centered values are of great importance, for example. It is important to note that these operations should only be calculated

using training data, and then applied equally to all training, validation, and testing data. Below are several forms of data preprocessing that are used and why they are important.

Mean subtracting data, defined as subtracting the mean vector μ (or sometimes by sample mean) of the set of data vectors $x \in X$:

$$x = x - \mu. \tag{3.25}$$

This is the most common form of data preprocessing in computer vision, and is needed if the data is not already zero-centered such that SGD avoids having back pass steps with all negative or all positive gradient updates [2]. Such updates would be jerky and difficult to converge to any low-loss state. Mean subtraction can be one constant value (mean over all elements of all signals), or sometimes an average signal (mean elements over all signals). In wireless communications this is unnecessary if the probability to send each bit or symbol is equal and the data sequence is sufficiently long. Most modulation schemes (BPSK, QPSK, QAM, etc) are already zero-centered.

Normalization, defined as scaling each zero-centered data vector by the vector or set inverse standard deviation $1/\sigma$:

$$x = \frac{x - \mu}{\sigma}. \tag{3.26}$$

Normalization is necessary if different inputs are expected to have different scales, units, or amplitudes, but are equally important [2]. This may be caused by different SNR values, or by the use of a diverse set of modulation schemes. As a consequence, larger amplitude modulations like QAM-64 may end up with larger weights associated with some samples than they deserve during SGD.

Principal Component Analysis [42] (PCA) is the last method to consider, which reduces the dataset X to the p highest variance signals $x \in X$. This is meant to save time, as classifiers trained with PCA-treated data are only training on the strongest linearly independent vectors, and have been shown to perform well [2] despite not using the full set of vectors X. PCA-reduced data X_{PCA} is also called dimensionally reduced, defined as the dot product of the data X with U_p (the last p columns of each row of the eigenvectors U) formed by the SVD factorization of the covariance matrix $cov(X)$ of the mean-subtracted data X_{ZC}:

$$x_{i_{ZC}} = x_i - \frac{\sum x_i}{k \times D}, x_i \in X, x_{i_{ZC}} \in X_{ZC}, \tag{3.27a}$$

$$cov(X_{ZC}) = \frac{X_{ZC}^T \cdot X_{ZC}}{N}, \tag{3.27b}$$

$$USV^* = cov(X_{ZC}), \tag{3.27c}$$

$$X_{PCA} = X_{ZC} \cdot U_p, U_p \in [N, p]. \tag{3.27d}$$

for eigenvalues S, conjugate transpose of the unitary matrix V^*, k samples per signal, dimensionality $D = 2$ for in-phase and quadrature components, N is the number of signals in the dataset, X^T is the

matrix transpose of X, and $\cdot$ is the element-wise dot product between two matrices:

$$X \cdot Y = x_1 \times y_1 + x_2 \times y_2 + ... + x_{n-1} \times y_{n-1}, \tag{3.28}$$

for elements $x \in X, y \in Y$ of length n.

3.2 Convolutional Neural Networks Architecture and Design

In Section 3.1 it was discussed how classification is performed for a given set of weights and testing signals, and how those weights are trained using SGD. In this section the topics of Convolutional Neural Networks (CNNs), what makes one classifier different from another and what it means to train too much will be discussed.

In linear classifiers, each layer of the architecture is fully-connected with each neuron's output getting fed to the input of each neuron in the subsequent layer. A key metric that differentiates one neural network from another is the number of parameters it has [2], or the sum of its weights and biases (w, b) across all neurons. This value is directly proportional to the learning capacity and training cost of the architecture. For example, consider Figure 3.9, which has $[3 \times 4] + [4 \times 4] + [4 \times 1] = 32$ weights w, one for each connection, and $4+4+1 = 9$ biases b, one for each non-input neuron. The resulting number of parameters is $w + b = 32 + 9 = 41$. Many modern architectures contain 100s of millions of parameters and 10s of layers [2].

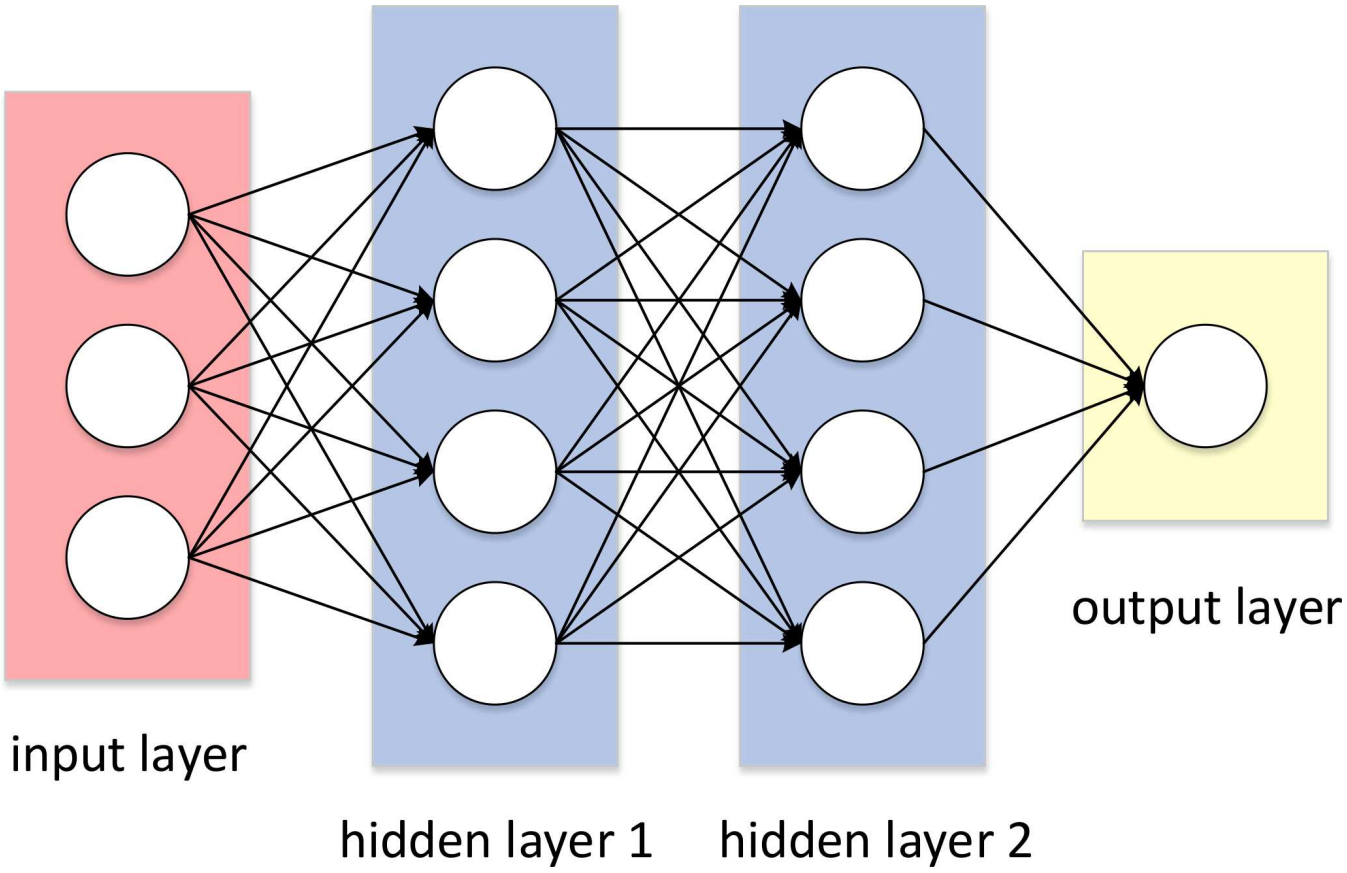


Figure 3.9: A three-layer neural network (adapted from [2]) with two fully-connected hidden layers. Each hidden layer has four neurons, and the input has three samples. As shown in Section 3.2, not all architectures use fully connected layers, and for good reason.

In Section 3.1, we discussed the linear classifier and how it reduces many-dimensioned test signals into a few values communicating its belief as to which category a signal belongs to, calculated using weights and biases calculated using SGD. It was also mentioned that classification performance rarely improves with additional layers beyond two hidden layers. This is not the case with CNNs [2], which results in tens of hidden layers. For all but the smallest signals, this translates to billions or trillions of parameters (see Figure 3.9). Fully connecting the neurons of each hidden layer would be hugely computationally expensive, and more importantly would result in significant amounts of over-fitting [2].

In order to address these two issues as well as to achieve levels of classification accuracy not found in linear classifiers, CNNs deploy a three-dimensional architecture (see Figure 3.10), convolutional layers in place of hidden layers, and a troupe of regularization layers that separate each convolutional layer.

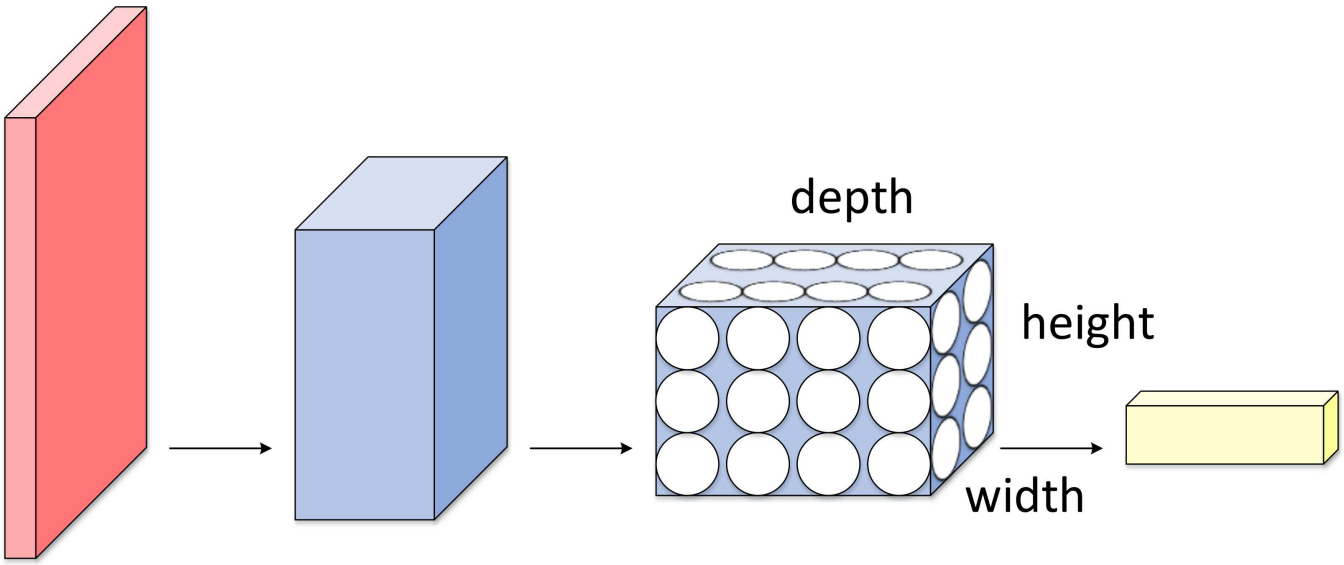


Figure 3.10: A comparable CNN (adapted from [2]) to the linear classifier in Figure 3.9. Convolutional layers are three-dimensional, and only the last few layers are fully connected. The rest of the CNN is much more sparsely connected in an effort to reduce over-fitting and computational cost.

In Section 3.1, we defined the flattened signal $x_i \in [1, k \times D]$ as having elements representing alternating IQ samples. The differences between linear classifiers and CNNs begin with the input vectors, which we now define in three dimensions as $x_i \in [D, k]$ for $D = 2$ where the top row of the signal contains k in-phase samples, and the bottom row represents k quadrature samples. Together the two rows form the complex sample (I, Q).

The next key difference comes in the form of how neurons compute their output axon (see Figure 3.1). In a CNN, the dot product operation is replaced by the convolutional layer. Consider a convolutional layer with a three-dimensional output shape containing a depth value equal to the number of filters and an equal width and height equal to:

$$out\_dim = (W - F + 2P)/S + 1, \tag{3.29}$$

where W is the side length of the square input, F is the receptive field size of the square filters used, P is the amount of zero padding (or value-zero elements) applied to all four sides of the input, and S is the stride or step size in the change in filter position between each convolution computation. See Figure 3.11 to see an example of calculating 18 axon values.

Notice that the output volume is smaller than the input volume. A key role of zero-padding is to maintain dimension size throughout forward passes without adding information to the data. Stride can be set to one to capture the most information from each input, but can be set higher as a form of regularization, or to reduce computational complexity. The next difference is that a CNN applies its activation function as a separate layer, performing an element-wise function, leaving the output shape unchanged. A ReLU

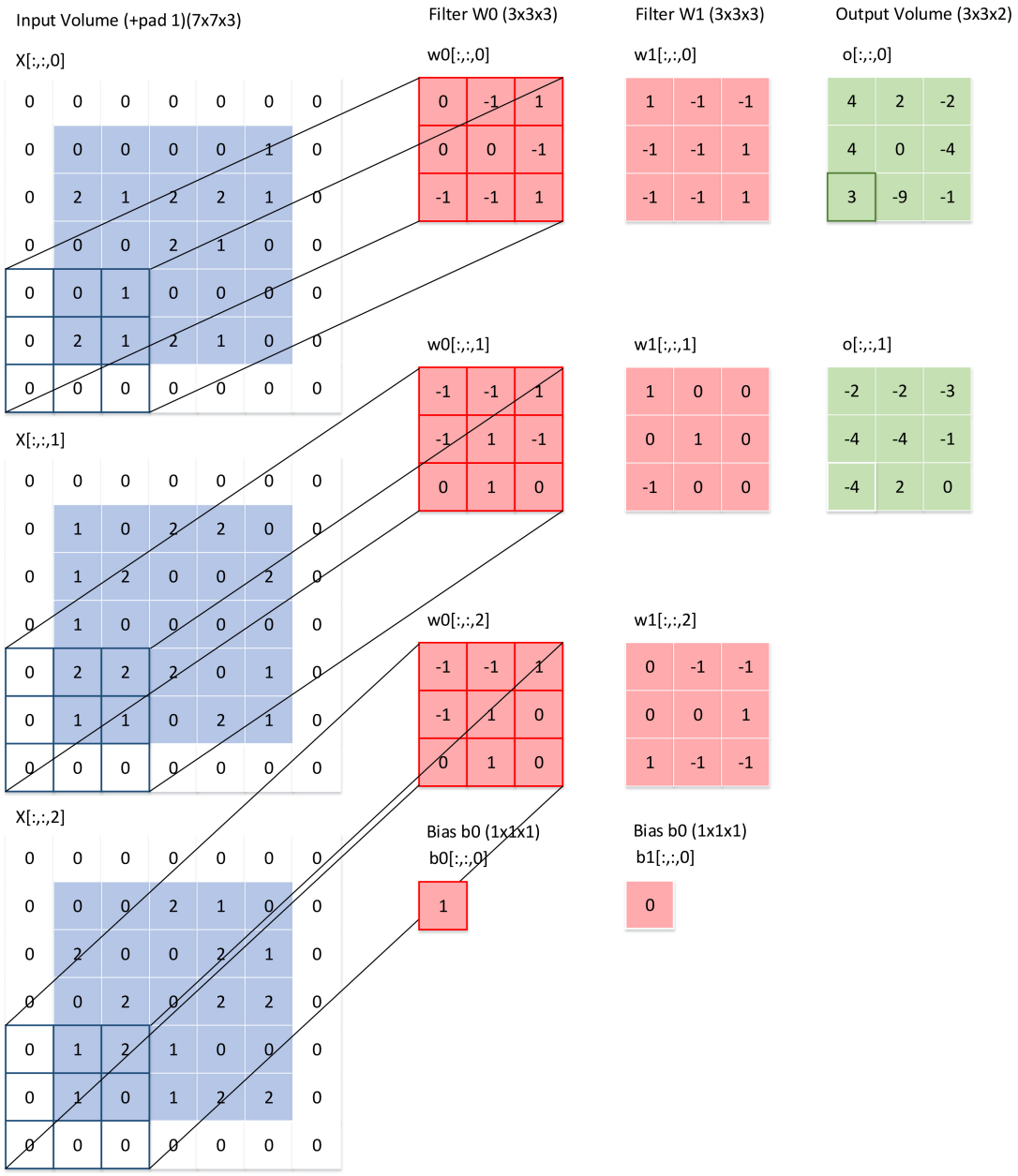


Figure 3.11: Two convolution computations (adapted from [2]) applied to an input (blue) of size $W = 5$, filters (red) of size $F = 3$, zero-padding (gray) $P = 1$, and stride $S = 2$. Through (3.29), we obtain the output matrix (green) height/width $(5 - 3 + 2 \times 1)/2 + 1 = 3$ of depth two due to using two filters for an output shape $\in [3, 3, 2]$. Highlighted is the computation of $o[2, 0, 0] = 3$, computed as $x[4 : 6, 0 : 2, 0] \circledast w0[: , :, 0] + x[4 : 6, 0 : 2, 1] \circledast w0[:, :, 1] + x[4 : 6, 0 : 2, 2] \circledast w0[:, :, 2] + b0[:, :, 0] = 3$.

(3.22) activation layer would maintain output shape, for example, computing each element $y = \max(0, x)$ from input element x.

A key operation CNNs perform is down-sampling, which is often achieved by pool layers. Pool layers reduce the height/width of the current output shape by taking the population mean or max-valued element of a group of input values, governed by a filter size and stride, similarly to convolutional layers. This critical step has been shown [2] to reduce over-fitting while maintaining the key feature information of signals.

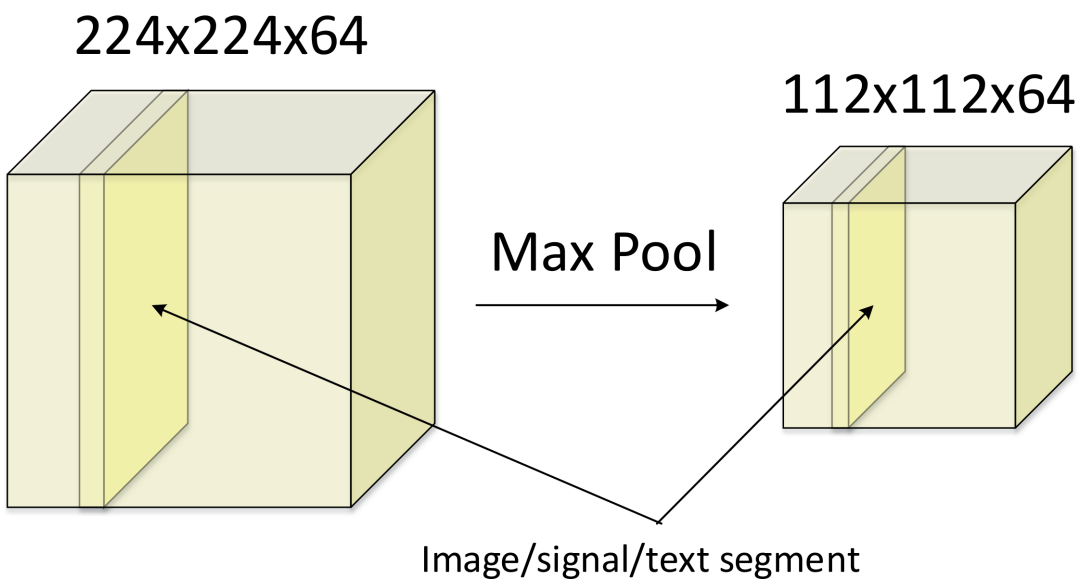


Figure 3.12: Max pooling (adapted from [2]) of a 244 by 244 pixel image. Input size $W = 224$, filter size $F = 2$, and a stride $S = 2$ resuls in an output shape (3.29) of $(224 - 2 + 2 \times 0)/2 + 1 = 112$. Depth is maintained.

It has been found in practice that repeating sequences of convolutional and down-sampling layers can extract higher dimension features in signals and images [2]. CNN layer sequences are typically terminated with a few fully connected dense layers whose elements correspond to class scores, such as the ConvNet CNN architecture in Figure 3.13.

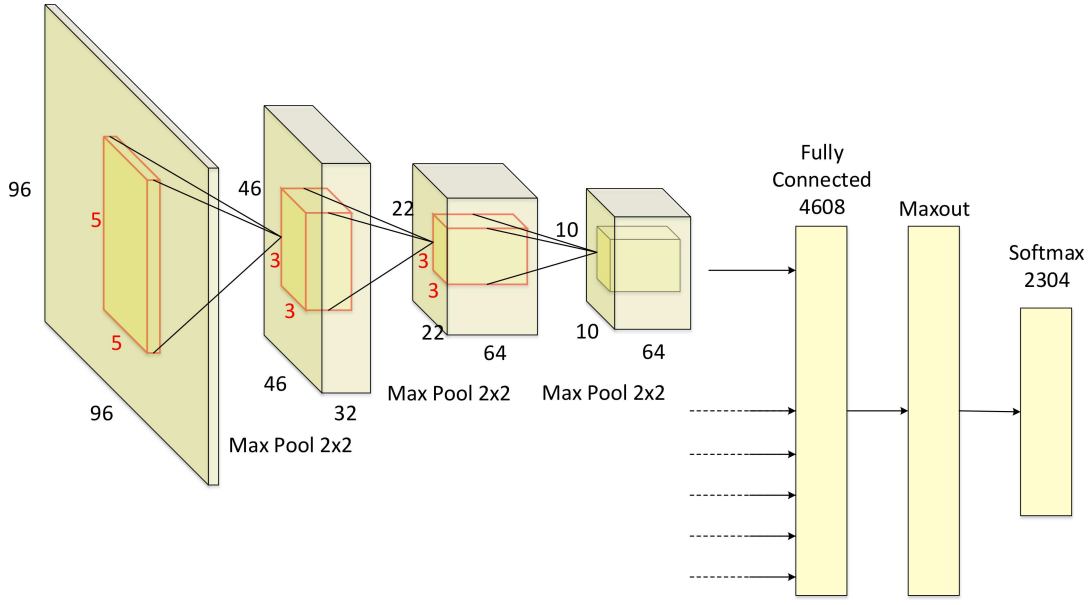


Figure 3.13: A ConvNet [3] architecture that passes raw image data through three convolutional layers, a fully connected dense layer, a maxout layer, and a softmax classifier.

There have been many investigations in the last decades into what the best CNN architectures are. There is a lot to decide, including where to place down-sampling, convolution, fully-connected, and ac-

tivation layers, as well as how many sequences of those placements to use. Additionally, each process is dictated by potentially dozens of hyper-parameters. Below is a brief list of powerful, popular, and recent CNN architectures (a more detailed survey is given recently by [43]):

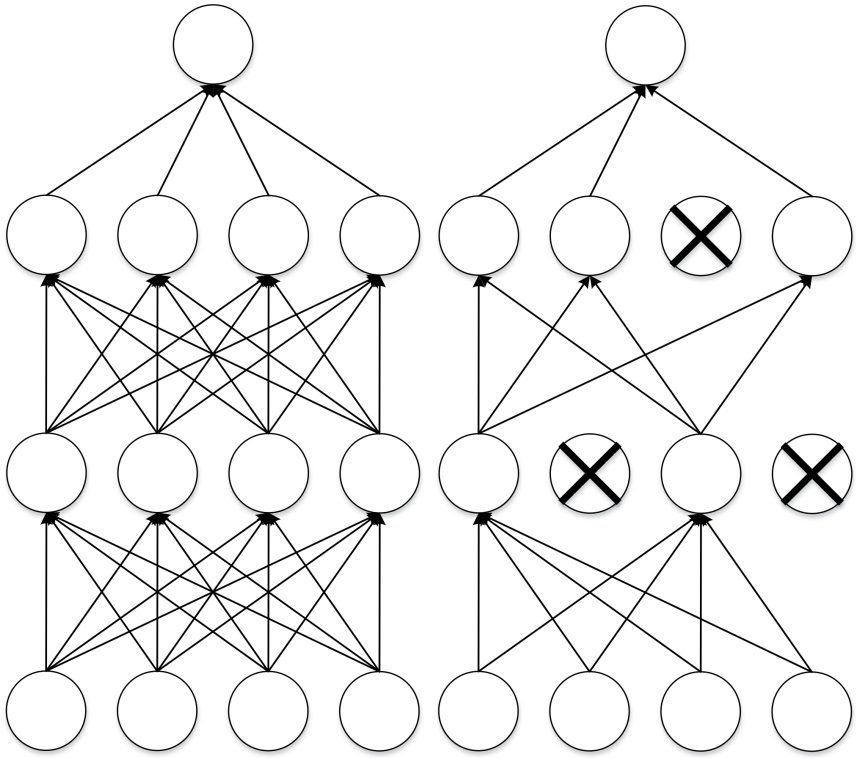
- LeNet [44] is the first successful CNN architecture, used to read zip codes and digits.
- AlexNet [45] is very similar to LeNet but is much deeper and more popular. Presented in 2012, AlexNet featured stacked convolutional layers, where previously such layers were always followed immediately by pooling layers.
- ZFNet [46] was presented in 2013 as an improvement to AlexNet, designing the middle convolutional layers to be bigger than the starting and ending ones. Additionally, hyper parameters were tweaked, making the stride and filter size of the first convolutional layer very small.
- GoogLeNet [47] was designed in 2014 by a group of staff from Google. The architecture made use of an Inception Module (V1, later followed by Inception V2 [48], V3 [49], and Inception-ResNet [50]), reducing the number of parameters by an order of magnitude. Additionally, GoogLeNet found that the parameters contained in the ending fully-connected layers in previous CNNs did not have much impact on classification accuracy. By removing these and replacing the fully-connected layers with average-pooling layers, parameter counts can be further reduced.
- VGGNet [51] was also introduced in 2014, drawing attention to network depth. The optimal depth they found was having 16 layers, but the architecture is mostly popular for its very homogeneous composition of repeating 3x3 convolutional layers and 2x2 pooling layers from start to finish.
- ResNet [52] was presented in 2015, featuring skip connections, batch normalization, and a complete lack of fully-connected layers. Skip connections helped SGD training (3.9) avoid gradient saturation, while batch normalization and a lack of fully-connected layers reduced training complexity.

While pooling is an intuitive and computationally simple method of down-sampling, there are several other popular methods to control the capacity of a CNN to prevent over-fitting:

- L2 regularization (3.4) adds the term $R(w) = \frac{1}{2}\lambda w^2$ to the loss function (3.3). The $\frac{1}{2}$ constant is used such that the gradient calculated during SGD is equal to λw. L2 regularization requires weight decay (see Section 3.1: Parameter Updates) to be linear (or vanilla).
- L1 regularization is similar to L2 regularization, often combined in the form $R(w) = \lambda_1 \mid w \mid + \lambda_2 w^2$. This combination is defined as elastic net regularization, resulting in most weights being near zero, causing the weight matrix to be sparse. Consequently, most parameters can be ignored, and only parameters corresponding to the strongest features need to be used.

- Max norm constraints bound weight values by imposing the upper limit $\|\vec{w}\|_2 < c$. This prevents weights from significant increases, but adds ambiguity to weight information in cases where many weights are clamped.

- Dropout [53] is the most effective and simple regularization method [2]. Dropout sets weights that connect any two layers to zero during training with probability p. It is effective because over-fitting is caused by an overabundance of neural connections [2], and dropout very directly reduces the number of those.



(a) Before dropout is implemented. (b) After dropout is implemented.

Figure 3.14: An illustration (adapted from [2]) of full-connected neurons before (a) and after (b) connections are dropped. Arrows represent connections between neurons, while neurons with x's through them represent neuron connections terminated by being dropped out.

3.3 Neural Networks: Universal Approximators

The reason neural networks can classify any set of signals is because they can reduce many-element signals down to a few values using any set of functions. A neural network with just one hidden layer is a universal approximator, or its outputs' relation to its inputs can be written as any function one can come up with [54].

This is an enormous claim, and any skeptic should reasonably meet it with caution. Consider Figure 3.3, a 2-layer NN with two neurons in its hidden layer. Using the sigmoid (3.20) function $\sigma(wx + b)$ For large w, small b, it is shown in Figure 3.3 that the output can be shaped into a step function, centered where ever desired by varying b.

It is easy to see, then, knowing the fundamental theorem of calculus, that at a subsequent neuron which performs a sigmoid operation on the dot product of its weighted inputs, any function can be approximated given an infinite number of step functions with customizable location and height (using weights h_i, see Figure 3.3). More formally, as shown in [54], for any continuous function $f(x)$ and some $\epsilon > 0$, there is a neural network with one hidden layer $g(x)$ that uses some non-linear function (i.e., tanh (3.21) such that $\forall x, \mid f(x) - g(x) \mid < \epsilon$.

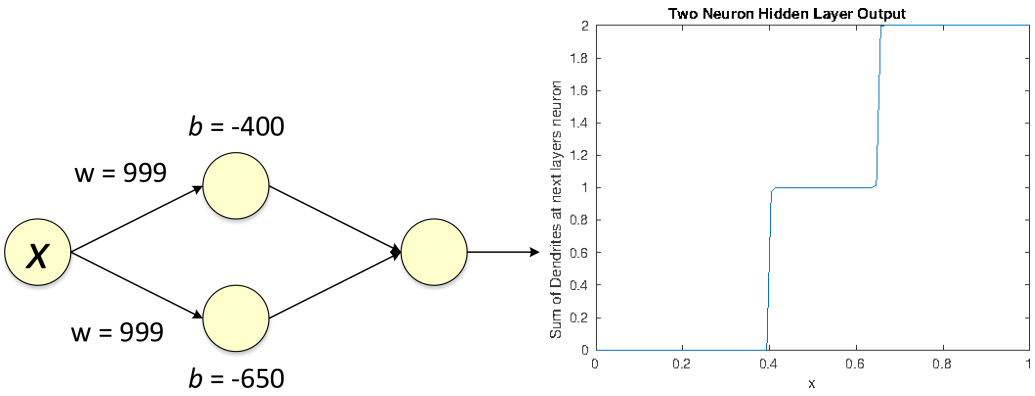
Although only one hidden layer is needed to fully represent any continuous function $f(x)$, in practice multiple layers are much more practical, as they fit better with observed functions and statistics [2]. For fully connected neural networks, it has been found that classification accuracy is rarely increased for architectures beyond two hidden layers [2]. In Section 3.2, we will show this is not at all the case for CNNs, which often benefit from having tens of hidden layers.

3.4 Bayesian Optimization of Machine Learning Algorithms

Bayesian optimization [55] aims to find the global maximizer x^* of the unknown objective function f such that:

$$x^* = \arg\max_{x \in \chi} f(x), \tag{3.30}$$

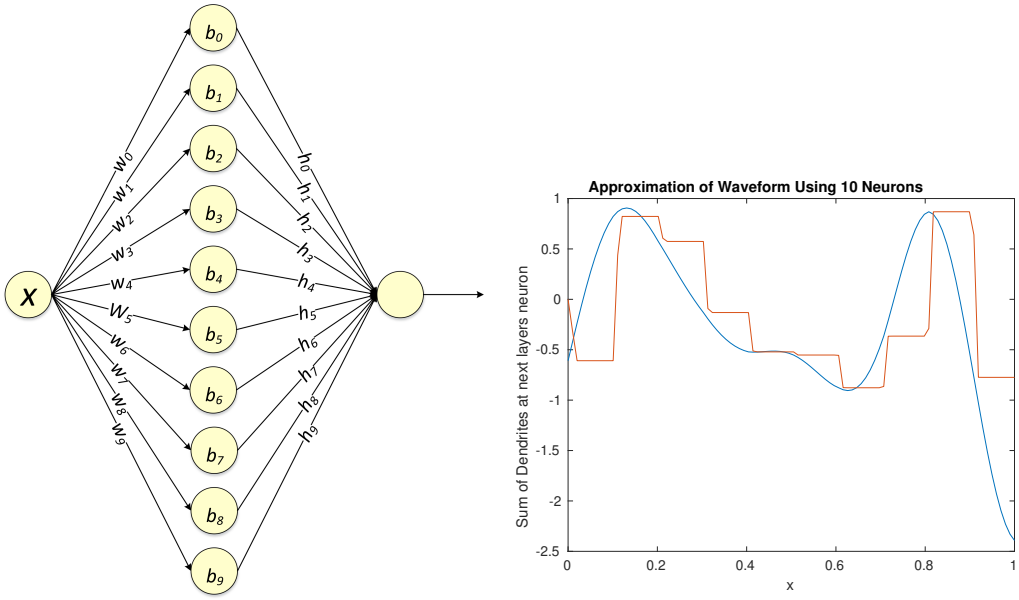
where χ is the design space. Neural Networks can be viewed as a form of Bayesian optimization. The reason for this is because Bayesian optimization is any sequential model-based approach to solving a problem. A CNN (see Section 3.2) can be represented as a Bayesian optimization problem by mapping the tunable parameters (weights) $W \rightarrow x$ and the observed classification accuracy (loss function) $L = f$ (*i.e.,* equation (3.3)). That, however, is where the comparison ends, as Bayesian optimization is categorically a form of Reinforcement Learning (RL), or trying to learn about an environment. Updates to the parameters x are provided via Bayesian posterior updates, or our updated beliefs given data. Posterior updates are guided by



(a) Sigmoid as in (3.20) parameter w shapes the transient part of the output of each hidden layer neuron, while b shifts it in x.

(b) The sigmoid function saturates at zero and one, so output ranges from zero to two for this network.

Figure 3.15: A diagram (a) of a simple non-linear neural net with a two neuron hidden layer and a plot (b) describing its output over values $0 < X < 1$ (see Appendix A.7).



(a) Input $0 < X < 1$ is weighted and passed through sigmoid non-linear activation functions.

(b) The continuous waveform (blue) can be approximated by the neural net output (red).

Figure 3.16: A diagram (a) of a ten neuron hidden layer and a plot (b) of their summed output. With each additional neuron in the hidden layer, the sum of sigmoids at the neuron in the subsequent layer (see Appendix A.7) is a closer approximation of a given continuous waveform.

acquisition functions $\alpha_n : \chi$, where the next parameters in time x_{n+1} are chosen by maximizing the current time acquisition function α_n. To accomplish this goal, Bayesian optimization asks for two ingredients: a probabilistic surrogate model which contains a prior distribution describing our current beliefs about the unknown loss function, and a known loss function that describes how optimal a series of queries are at accomplishing a task. The expected loss function is minimized to select the optimal queries, and the observed outputs cause the prior to be updated to provide a more accurate distribution.

Use of an acquisition function is often much more computationally expensive than optimizing the black box function f [55], so it is critical that the acquisition functions be simple to evaluate. See Figure 3.17 for an illustration of three time iterations of Bayesian optimization.

Given an *a priori* distribution $p(w)$ which describes probable values for parameters w before observing data, the *a posteriori* distribution $p(w|\mathcal{D})$ can be inferred using Bayes' rule:

$$p(w|\mathcal{D}) = \frac{p(\mathcal{D}|w)p(w)}{p(\mathcal{D})}, \tag{3.31}$$

which describes our updated beliefs about w after observing data $\mathcal{D}$. This is known as a Parametric Model. The choice of statistical model used now is paramount to the effectiveness of the Bayesian optimization [55]. The simplest such model is the Beta-Bernoulli Bandit Model (see Algorithm 1). The metaphor the name is based on is a gambling one, where a bandit Bernoulli problem is considered. A slot machine has K arms or levers, each with some probability of winning money. The effectiveness of each arm on the bandit is modeled as the function f, taking function input $a \in 1, ..., K$, returning the Bernoulli parameter $\in (0, 1)$. With the outcome of winning money or not denoted as $y_i \in \{0, 1\}$, the outcome of pulling arm a_i has mean parameter $f(a_i)$. With the K arms available, f can be fully described by parameters $w \in (0, 1)^K$.

Once arms start getting pulled, it can be seen how often each arm actually wins money, and the comparison is made to probability the arm was believed to have to win money. This data is represented as $\mathcal{D} = \{(a_i, y_i)\}_{i=1}^n$, where a_i indicates which of the K arms were pulled, and y_i is one of money was won and zero otherwise. We can compute the *a posteriori* distribution using the prior distribution over w:

$$p(w|\alpha, \beta) = \prod_{a=1}^{K} beta(w_a|\alpha, \beta), \tag{3.32}$$

which is a good choice because beta distributions are the conjugate prior to the Bernoulli likelihood, or part of the same probability distribution family. The *a posteriori* distribution can be derived using (3.32) as:

$$p(w|\alpha, \beta) = \prod_{a=1}^{K} beta(w_a|\alpha + n_{a,1}, \beta + n_{a,0}), \tag{3.33}$$

where $n_{a,0} = \sum_{i=1}^n \mathbb{I}(y_i = 0, a_i = a)$ is the number of losses resulting from pulling arm a and $n_{a,1} = \sum_{i=1}^n \mathbb{I}(y_i = 1, a_i = a)$ is the number of wins resulting from pulling arm a. The hyper-parameters α, β

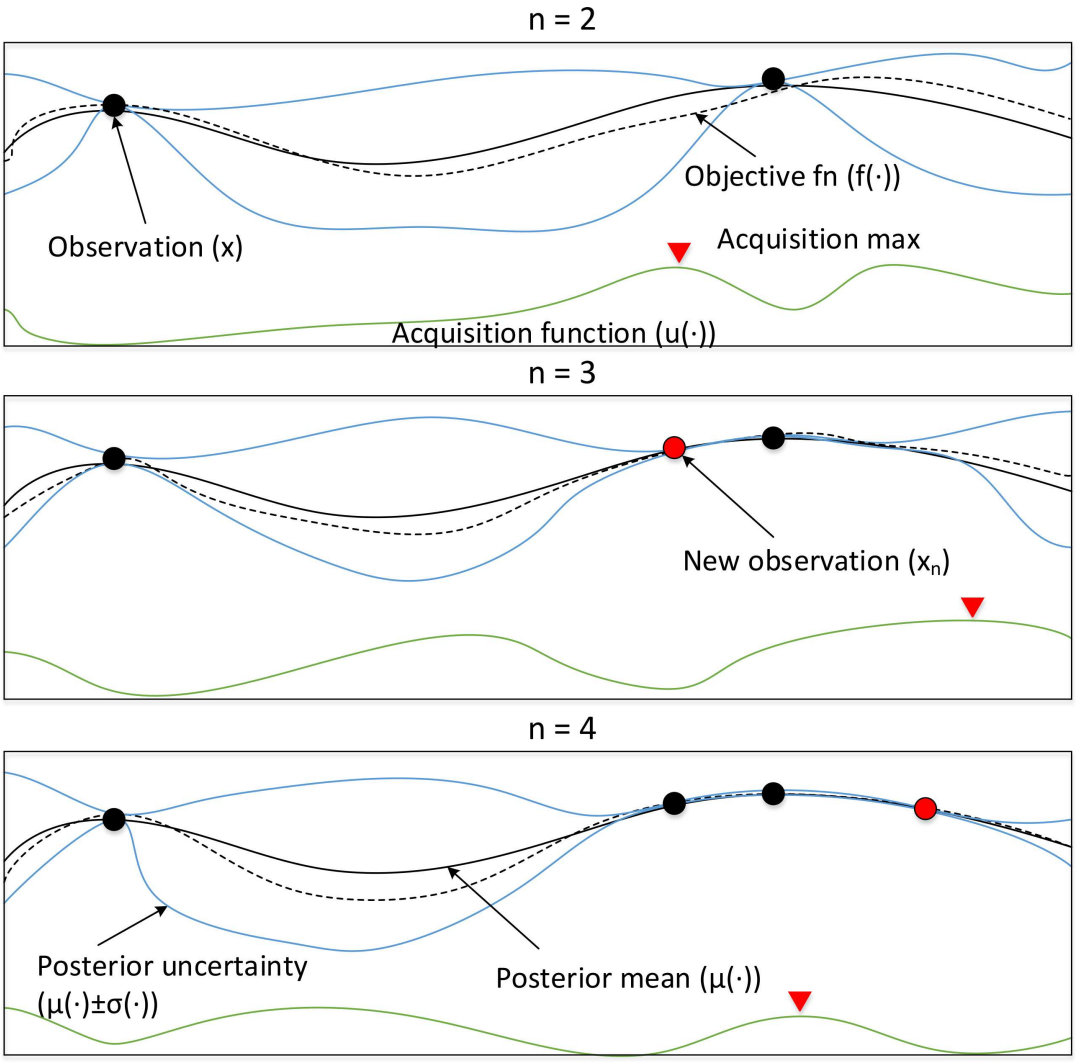


Figure 3.17: An illustration (adapted from [55]) of three time iterations of (3.30). The black line is the estimated objective or loss function f, while the dashed black line is the true f (unknown but visualized). The acquisition function α is in green, whose maxima are highlighted with red arrows, indicating either exploration (when uncertainty $\sigma(\cdot)$, blue, is large) or exploitation (model prediction is high, solid and dashed black lines match). Observations x_n are marked as black dots, with the new observations in the $n = 3$ and $n = 4$ sub-figures highlighted in red. Notice how new observations reduce uncertainty, and are first taken at high value points (right skewed) to maximize impact on acquisition function reduction.

must be tuned. Finally, we decide the next arm a_{n+1} to pull by posterior sampling a single sample $\tilde{w}$ from the posterior and maximizing the surrogate $f_{\tilde{w}}$:

$$a_{n+1} = \arg\max_a f_{\tilde{w}}(a), \tilde{w} \sim p(w|\mathcal{D}_n). \tag{3.34}$$

Arms are only explored under this model if they are likely under the belief of the posterior to be optimal, or bring in the most wins. Using the bandit parametric model in deep learning, weights W would be updated at each step n in descending order from the most to least impact on reducing the loss function.

Algorithm 1 The multi-armed bandit algorithm

procedure *Input*: α,β: HYPER-PARAMETERS REQUIRED
  Initialize $n_{a,0} = n_{a,1} = i = 0$ for all a
  repeat
  for $a = 1, ..., K$ **do**
    $\tilde{w}_a \sim beta(\alpha + n_{a,1}, \beta + n_{a,0})$
  end for
  $a_i = argmax_a \tilde{w}_a$
  Observe y_i by pulling arm a_i
  if $y_i = 0$ **then**
    $n_{a,0} = n_{a,0} + 1$
  else
    $n_{a,1} = n_{a,1} + 1$
  end if
  $i = i + 1$
  until stop criterion reached
end procedure

Can we make predictions on outcomes from arm-pulling without the parameters w? These are known as Non-Parametric Models. Using a kernel mapping trick [55], rather than mapping features to labels, we can describe a similarity between points, depending on which paradigm is more computationally tractable. In other words, it is simpler to work with the distances between points rather than to map those points in high-dimensional space. This requires only a $J \times J$ matrix inversion on kernels as opposed to the parametric models' $n \times n$ matrix of time indexed observation periods. Consider the following kernel model:

$$K_{i,j} = k(x_i, x_j) = \Phi(x_i) V_0 \Phi(x_j)^T = \langle \Phi(x_i), \Phi(x_j) \rangle_{V_0}, \tag{3.35}$$

where $\Phi = \phi(X)$ is the feature mapping matrix on design matrix X, V_0 a hyper-parameter denoting the variance of a zero-mean Gaussian random variable, x_i, x_j are all similar pairs of points in X, and $\langle \Phi(x_i), \Phi(x_j) \rangle_{V_0}$ is the inner product. The benefit of the kernel line of thinking is a normally distributed, data-driven posterior distribution can be described by mean and variance below, with no parameters known as a Gaussian Process:

$$\mu_n(x) = \mu_0(x) + k(x)^T (K + \sigma^2 I)^{-1} (y - m), \tag{3.36a}$$

$$\sigma_n^2(x) = \mathtt{k}(x, x) - k(x)^T (K + \sigma^2 I)^{-1} k(x), \tag{3.36b}$$

where $k(x)$ is the covariance between point x and all previous observations, and $\mathtt{k}(x_i, x_j)$ is the kernel at

$K_{i,j}$. The above mean and variance describe the non-parametric model's estimation and uncertainty at point x representing the solid black line and blue uncertainty area in Figure 3.17, accomplished without weights W but with kernels K. A simple and common kernel $\mathtt{k}(x, x')$ is the Matérn [56] stationary kernel covariance function:

$$\mathtt{k}(x, x') = \theta_0^2 \exp(-r), \tag{3.37}$$

for $r^2 = (x, x')^T \Lambda (x, x')$, and Λ the diagonal matrix populated by d length scales θ_i^2. All θ values are hyper-parameters. This section only begins to scratch the surface of Bayesian optimization. Besides the kernel above there is a host of other options, as well as many parametric and non-parametric methods such as linear models, sparse spectrum Gaussian Processes, Sparse Pseudo-input Gaussian Processes, and Random Forest [55]. Finally there is a choice of acquisition functions not even covered here, although in [55] it is mentioned that hyper-parameter and acquisition function choice does not have a strong impact on performance.

3.5 Distillation of Neural Network Weights

Distillation is an idea [57] that begins with the thought that a very costly but effective way of improving a classifier's classification accuracy on a given signal would be to have many identical neural networks classify a signal and average the results, due to the random nature of SGD, SVM, or other training methods.

In Section 3.1: Soft-Max Classifier, the soft-max function (3.6) was discussed in its use to describe a neural network's belief in a signal belonging to a class. What that equation leaves out is a scaling parameter named temperature, which determines how far or closely spaced probabilities are. Soft-max functions with high temperature tend towards equal probabilities, where low temperature soft-max functions tend to award the highest probability a value of one, all else zero. In its simplest form, distillation trains two neural networks, one with a low parameter count and one with a cumbersome, high parameter count, with high temperature T in its soft-max:

$$f_j(z) = \frac{e^{z_j/T}}{\sum_k e^{z_k/T}}. \tag{3.38}$$

After training, the temperature is set to one in the distilled network, but kept constant in the cumbersome network. This high-entropy form of training has seen considerable investigation since the foundational paper [57] for its increases in classification accuracy [57] and increased security against adversarial perturbations [21].

3.6 Generative Adversarial Networks (GAN)

Generative Adversarial Networks [58] achieve deep feature extraction on unannotated data through back passing through pairs of neural networks. Consider a decoy neural network that generates signals with the objective of fooling a modulation classifier NN into thinking the signals it is observing are not of one modulation scheme, but another. It has been shown in many works (see Figure 8 of [59]) that small or very large changes can be made to signals, images, and voice that cause brittle or too generally trained classifiers to poorly classify.

In a GAN setup, the decoy network described above is called the generator, while the modulation classifier is the discriminator. See Figure 3.18 for an overview of how networks in a GAN interact with each other.

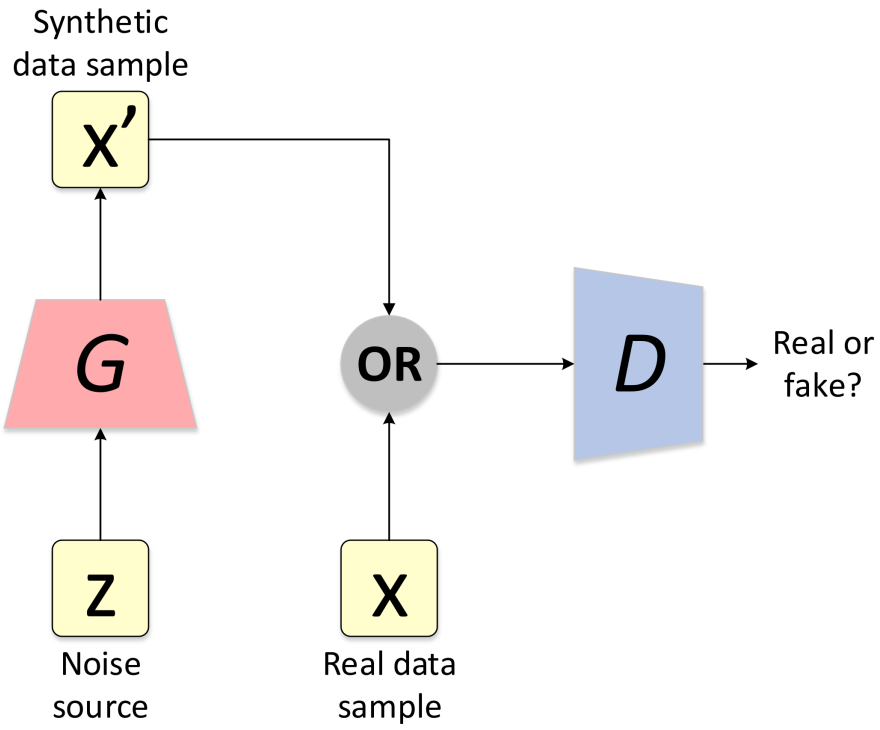


Figure 3.18: A flow diagram (adapted from [58]) of a GAN testing process (training stage is complete). Synthetic data samples are formed by the generator via a noise source, and the discriminator tries to correctly classify them as fake while classifying real data samples as real. Discriminator average accuracy is bounded by 50% (guessing) and 100% (always correct). Depending on the learning capacity of each neural network and the methods of training, evaluation-time accuracy can fall anywhere in-between.

The goal of training a GAN is to find the parameters of the discriminator that maximize classification accuracy, and the parameters of the generator which minimize the discriminator's classification accuracy. For the generator, this means optimal parameters have been achieved when the discriminators accuracy falls to 0.5, or correctly classifying fake versus real samples half of the time. Given that there are only two labels, this accuracy is equal to that of a coin flip, or the best possible confusion.

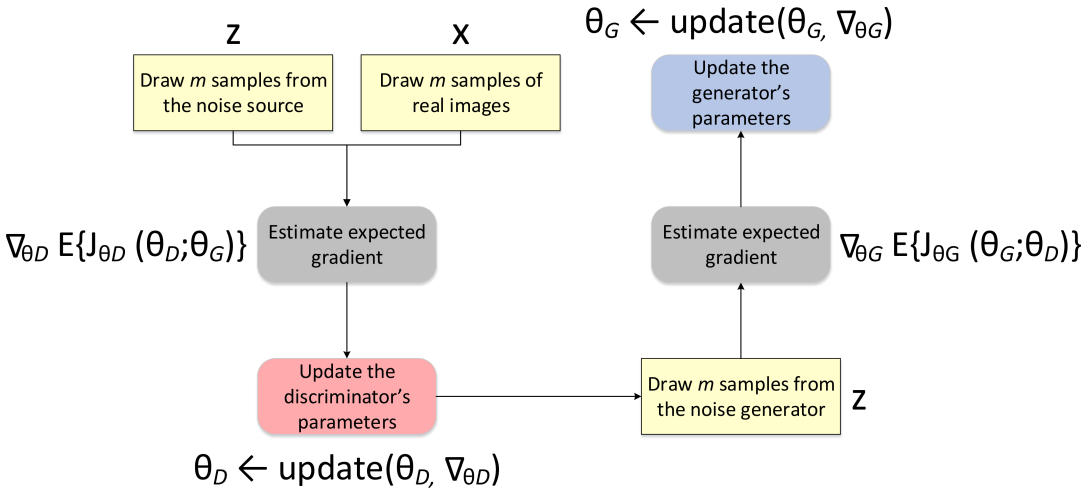


Figure 3.19: A flow diagram (adapted from [58]) describing the SGD (3.9) training feedback loop between the generator and discriminator (used to drive the testing stage shown in Figure 3.18). Parameter updates continue until the learning capacity of the networks are reached and average classification accuracy of the discriminator converge to a steady state value $\in (0.5, 1)$.

3.7 Neural Network Feature Transformations Performed Via Domain Adaptation

There is a method of generalized training [60] that has been explored in computer vision recently, motived by autonomous vehicle technology. There is not enough annotated (tagged with classification labels) images of roads environments, so a method was developed for training autonomous vehicle computer vision neural networks with computer-generated 3D images that still allow the networks to test well with real-world pictures at evaluation time (see Figure 1 of [60]).

The core idea of domain adaptation is to create a latent space Z (see Figure 3.20) that is characteristic agnostic that can perform feature transformations on feature vectors learned from annotated data from the source domain X into data from the target domain Y using unannotated data from Y such that a sum of weighted loss functions is minimized (see Figure 3 of [60], which describes how the weighted loss functions (3.40)-(3.45) interact with each domain X, Y, and Z).

Domain adaptation attempts to perform SGD (3.9) on the general loss function:

$$Q = \lambda_c Q_c + \lambda_{id} Q_{id} + \lambda_z Q_z + \lambda_{tr} Q_{tr} + \lambda_{cyc} Q_{cyc} + \lambda_{trc} Q_{trc}, \tag{3.39}$$

where each individual loss function Q is weighted by λ. Below, the role and contents of each loss function Q is discussed.

The core loss function Q_c aims to perform the task of even the most basic linear classifier (see Section 3.1): calculate some difference between each ground truth label c_i and each evaluation signal. This is

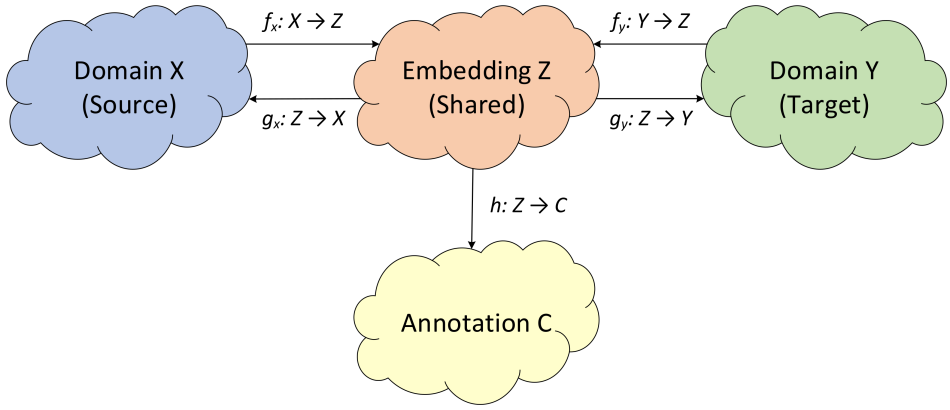


Figure 3.20: A flow diagram (adapted from [60]) describing the various transforms f_x, g_x, h, f_y, g_y and spaces X, Z, Y, C and their interactions at the highest level in domain adaptation. The field is motivated by scarcity of annotated real pictures, but has much wider applications. Implemented correctly, training of classifiers becomes highly generalizable, making testing well under conditions not trained under becomes very robust when domain adaptation is performed on a set of unlabeled data from the new target domain.

a standard neural net operation, independent from domain adaptation, described as [61]:

$$Q_c = \sum_i l_c(h(f_x(x_i)), c_i), \tag{3.40}$$

where l_c is some loss function (*i.e.,* L_1 norm, L_2 norm cross entropy), $h : Z \Rightarrow C$ is the transform in Figure 3.20 from Z space to the annotations C, f_x is the mapping from the source to the embedded domain, and x_i is a signal from the source domain X.

First and foremost in domain adaptation, it is desired for the transforms in Figure 3.20 to only remove structured noise from signals, not information bits. The mapping from the X or Y domain to the Z domain and back should be as close to an identity mapping as possible. That is the role of the individual loss function Q_{id} [6], which is described as:

$$Q_{id} = \sum_i l_{id}(g_x(f_x(x_i)), x_i) + \sum_j l_{id}(g_y(f_y(y_j)), y_j), \tag{3.41}$$

where g_x, f_x, g_y, f_y are domain mappings (see Figure 3.20), and l_{id} is the sample or pixel-wise loss function (*i.e.,* L_1 or L_2 norm).

Secondly, it is very important for the latent Z space to be domain agnostic. This is achieved by training the Z space using a GAN (see Section 3.6) discriminator $d_z : Z \to \{c_x, c_y\}$ (c are annotations mapped to Z) which tries to classify if a feature in the latent space Z was generated from the X or Y domain. The GAN's loss function contributing to (3.39) can be described as [62]:

$$Q_z = \sum_i l_a(d_z) f_x(x_i)), c_x) + \sum_j l_a(d_z(f_y(y_j)), c_y), \tag{3.42}$$

where l_a is a suitable loss function for GANs.

As an extra precaution to make sure the transforms in Figure 3.20 are consistent, discriminators in the source domain $d_x : X \to \{c_x, c_y\}$ and target domain $d_y : Y \to \{c_x, c_y\}$ are trained [63] to classify whether a sample is fake (from the other domain) or real (from its own domain). The loss function minimized to accomplish this must match ground truths c_x to signals from the X domain mapped through the Z domain and into the target Y domain, and likewise for the ground truths c_y of signals from the Y domain:

$$Q_{tr} = \sum_i l_a(d_y(g_y(f_x(x_i))), c_x) + \sum_j l_a(d_x(g_x(f_y(y_j))), c_y), \tag{3.43}$$

Similarly to (3.41), a cycle loss function [63] was developed to ensure the mapping from any signal in the X domain to the Z domain, Y domain, back to the Z domain, and finally back to the X domain is as similar as possible to the original image. The equivalent is added for images from the target Y domain to formulate identity mappings through the use of:

$$Q_{cyc} = \sum_i l_{id}(g_x(f_y(g_y(f_x(x_i)))), x_i) + \sum_j l_{id}(g_y(f_x(g_x(f_y(y_j)))), y_j), \tag{3.44}$$

Similarly to (3.40), a translations loss function formulated in [60] is minimized such that classifications on signals additionally passed through the fake domain (Y if the signal is from X, X if the signal is from Y) are correct in the Z domain when mapped to annotations C:

$$Q_{trc} = \sum_i l_c(h(f_y(g_y(f_x(x_i)))), c_i). \tag{3.45}$$

3.8 Modulation Classification

In Section 3, Chapter 4, and Chapter 5, the concept of classification labels is often mentioned, and used in the context of modulation classification. The aim of this section is to communicate to the reader what a modulation scheme is (see Section 3.8), and the different forms neural nets take to classify them (see Section 3.8.1). This area of AI communications is driven by a number of things, including Adaptive Coding and Modulation [64] (ACM), a commonly used technique to adapt wireless transmissions to highly time-varying phenomenon such as fading. One such thing ACM adjusts according to the environment and wireless channel is the best modulation scheme to use to maximize throughput, and the receiver in an ACM system may not always have prior knowledge of what scheme is being used to transmit.

Modulation

In the field of communications, a modulated signal $y(t)$ is simply the multiplication of the signal to be transmitted, $u(t)$, with a cosine described by ω_0, its carrier frequency, $\cos(\omega_0 t)$:

$$y(t) = u(t)\cos(\omega_0 t), \tag{3.46}$$

and that signal is typically demodulated into a base-band signal $z(t)$ by the receiver chain through the use of a demodulator, which can be described as:

$$z(t) = u(t)\cos(\omega_0 t). \tag{3.47}$$

However, this process has been shown [1] to result in frequency-domain periodic copies of the signal in $z(\omega)$, so a low-pass filter $H(\omega)$ must be implemented to remove them.

A primary use of signal modulation is traffic control. Two signals being sent by two transmit-receive pairs in the same location at the same time can be clearly detected and recovered [1] when modulated differently, or using the same scheme at different carrier frequencies. Another key benefit of modulation is to reduce the corruption of signals when traveling through a noisy channel [28].

While an exhaustive list of modulation schemes and their performance can be found in [65], the design and use of the Quadrature Phase Shift Keying (QPSK) modulation scheme is described in this section to give a better idea behind modulation scheme use.

Typically modulation schemes are described as being $M-ary$, or having M unique constellation points. For information represented by a number of bits b, M is calculated as:

$$2^b = M. \tag{3.48}$$

Modulation schemes are typically expressed as $s_n(t)$, time domain cosines that are functions of the message, n. In the case of QPSK $M = 4 = 2^b$ for $b = \log_2(M) = 2$, meaning each message contains two bits of information. Each possible message $n = 1, 2, 3, 4$ represents the binary message transform $n = i : b \rightarrow (b_1, b_2), i = 1, 2, 3, 4$, resulting in $n = 1 : b \rightarrow (0, 0) = 0, n = 2 : b \rightarrow (0, 1), n = 3 : b \rightarrow (1, 0)$, and $n = 4 : b \rightarrow (1, 1)$. For QPSK, the constellation map $s_n(t)$ is described as:

$$s_n(t) = \sqrt{\frac{2E_s}{T_s}}\cos(2\pi f_c t + (2n-1)\frac{\pi}{4}), n = 1, 2, 3, 4, \tag{3.49}$$

where E_s is the energy in Joules per symbol $n = i : b \rightarrow (b_1, b_2), i = 1, 2, 3, 4$, T_s is the sampling period of the ADC/DAC in Hz, and f_c is the modulation carrier frequency. It can be shown that any modulation constellation can be represented by a set of basis functions and amplitudes $s_n(t) = s_{n1}\phi_1(t) + s_{n2}\phi_2(t)$. Basis functions must be orthogonal, and the whole set orthonormal, meaning each basis is a vector in euclidean space on a unique axis of the coordinate space, mathematically defined as the inner product of any two basis functions being zero $\langle \phi_i(t), \phi_j(t) \rangle = \int_0^T \phi_i(t)\phi_j(t) = 0$.For QPSK, these are derived [28]:

$$\phi_1(t) = \sqrt{\frac{2}{T_s}}\cos(2\pi f_c t), \tag{3.50a}$$

$$\phi_2(t) = \sqrt{\frac{2}{T_s}}\sin(2\pi f_c t), \tag{3.50b}$$

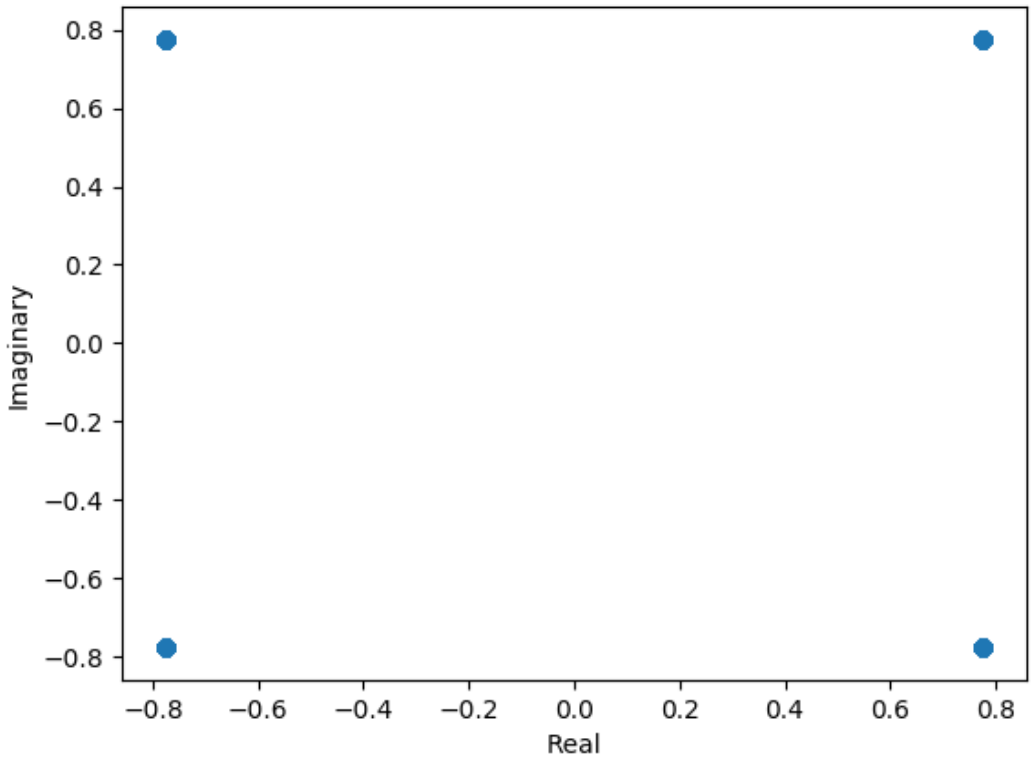


Figure 3.21: A set of QPSK constellation points (3.49) for $E_s = 4$. The horizontal axis is defined as $\phi_1(t)$ or the real valued element in a complex tuple, and the vertical axis as $\phi_2(t)$, traditionally represented as the imaginary valued element in a complex tuple. The resulting transformations are $n = 1 : b \to (0,0) : s \to (2/\sqrt{2}, 2/\sqrt{2})$, $n = 2 : b \to (0,1) : s \to (-2/\sqrt{2}, 2/\sqrt{2})$, $n = 3 : b \to (1,0) : s \to (-2/\sqrt{2}, -2/\sqrt{2})$, and $n = 4 : b \to (1,1) : s \to (2/\sqrt{2}, -2/\sqrt{2})$

resulting in the points $\left(\pm \sqrt{E_s/2}, \pm \sqrt{E_s/2} \right)$ for $n = 1, 2, 3, 4$, shown in Figure 3.21. A common metric used to describe the effectiveness of any given modulation scheme is its efficiency ϵ_p, defined as the squared minimum euclidean distance between any two constellation points $s_n(t)$, divided by the population mean energy in Joules per bit:

$$\epsilon_p = \frac{d^2_{min}}{\bar{E}_b} \tag{3.51}$$

For the case of QPSK, there are two bits per symbol, so $\bar{E}_b = \frac{1}{2}\bar{E}_s$. For (3.50), the L_2 norm of the basis functions is $d^2_{min} = \int_0^T (\phi_1(t) - \phi_2(t))^2 dt = ||s_1(t) - s_2(t)||^2 = (2/\sqrt{2} + 2/\sqrt{2})^2 = 8$, $s_1(t), s_2(t)$ chosen since all points are equidistant. The energy per symbol is $\bar{E}_s = \langle s_1(t), s_1(t) \rangle = (2/\sqrt{2})^2 + (2/\sqrt{2})^2 = 4$, so the efficiency comes out to be $\epsilon_p = 8/(4/2) = 4$, which is the highest efficiency a modulation scheme can achieve when using all constellation points available [28].

3.8.1 Neural Network Modulation Classification

In a communications transmit receive pair, it is not always known *a priori* which modulation scheme is to be used. This can cause critical failure, as if an incoming signal cannot be properly mapped to the

right message bits, bit error rate can increase significantly. Although numerous [4] Automatic Modulation Classification (AMC) methods have existed for decades, the area has seen new life with the data-driven, high accuracy classification results coming in from new deep neural networks [4]. It is the goal of this section to investigate the architecture of a foundational [4] Convolutional Long short-term Deep Neural Network (CLDNN) that performs modulation classification, as to provide insight on what is happening behind-the scenes when modulation classification is mentioned in other parts of this work.

In [4], signals $x_i \in [2, 128]$ are defined by two rows for in-phase and quadrature components, and 128 columns for each complex sample. The architecture used is exceedingly simple: two convolutional layers (see Section 3.2) and a single dense layer followed by a soft-max (3.6) classifier. The hidden layers are followed by ReLU (3.22) activation layers, and dropout layers (see Figure 3.2) with $p = 0.5$. In [4] figures describing their optimization process are detailed, resulting in choosing a filter size $F = 8$, stride $S = 1$, and 50 filters (see Figure 3.11 for an example convolution).

See Figure 11a of [4] for the time (top) and frequency (bottom) domain magnitude plots [4] of a trained $[1, 8]$ filter like those in Figure 3.22. This filter's first, second, and seventh weights have the most influence on classification. See Figure 11b of [4] for the time-domain IQ plot of a $[2, 128]$ output signal from the $[1, 8]$ filter in Figure 11a of [4]. The signal input to the filter was random, but trained to maximally activate the filters eight weights. The result is a Binary Phase Shift Keying (BPSK) waveform, indicating that this filter was trained to maximize the eventual soft-max class scores of BPSK signals.

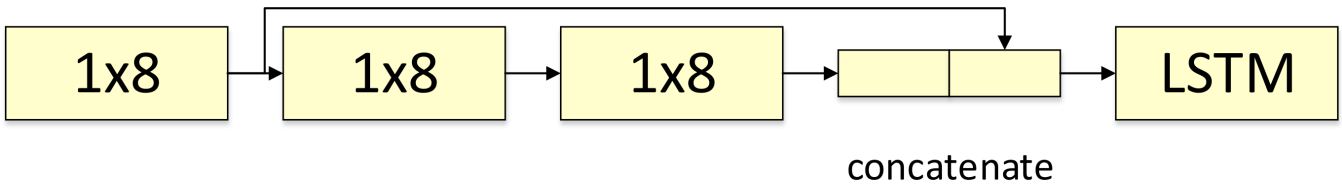


Figure 3.22: A flow chart (adapted from [4]) describing the forward pass (see Figure 3.8) of a set of eight input values through the CLDNN. A $[1, 8]$ input vector is concatenated with values filtered through a $[1, 8]$ filter in both the first and second convolutional layer. Each filter (see Figure 11a of [4]) contains eight weights and one bias value (see Figure 3.11 for example filters), which are calculated during SGD (3.9). The Long Short-Term Memory (LSTM) cell holds the values for the soft-max classification layer.

Two common metrics in modulation classification used to evaluate, in detail, the performance of a classifier is the testing accuracy curve (see Figure 3.23) and the confusion matrix (See Figure 3.24).

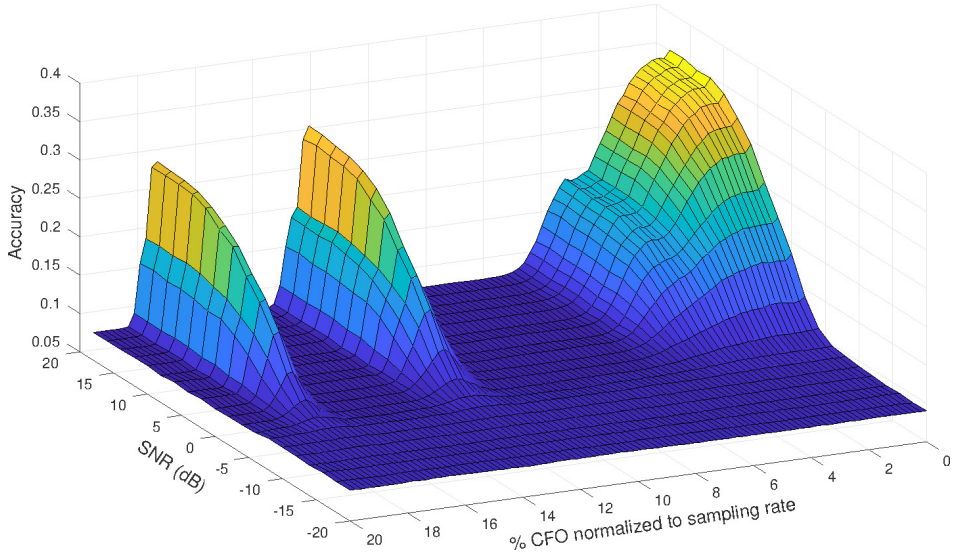


Figure 3.23: A 3D modulation classification accuracy plot obtained by testing a range of frequency offset RML2016.10a [5] data samples on a poorly-designed CNN over a range of SNR values. This shows the accuracy floor of 1/11, which indicates the CNN guessing one of the eleven modulation schemes in the dataset due to overwhelming frequency error. The peak accuracy of 35% is quite low due to poor hyper-parameter tuning and a low learning capacity architecture (caused by too much or not enough dropout, filter layers not correctly extracting features, not enough neurons in dense layers, etc). Modulation accuracy spikes at certain periodic values of frequency offset, perhaps due to aliasing (so much spinning that the IQ data doesn't look like its spinning anymore).

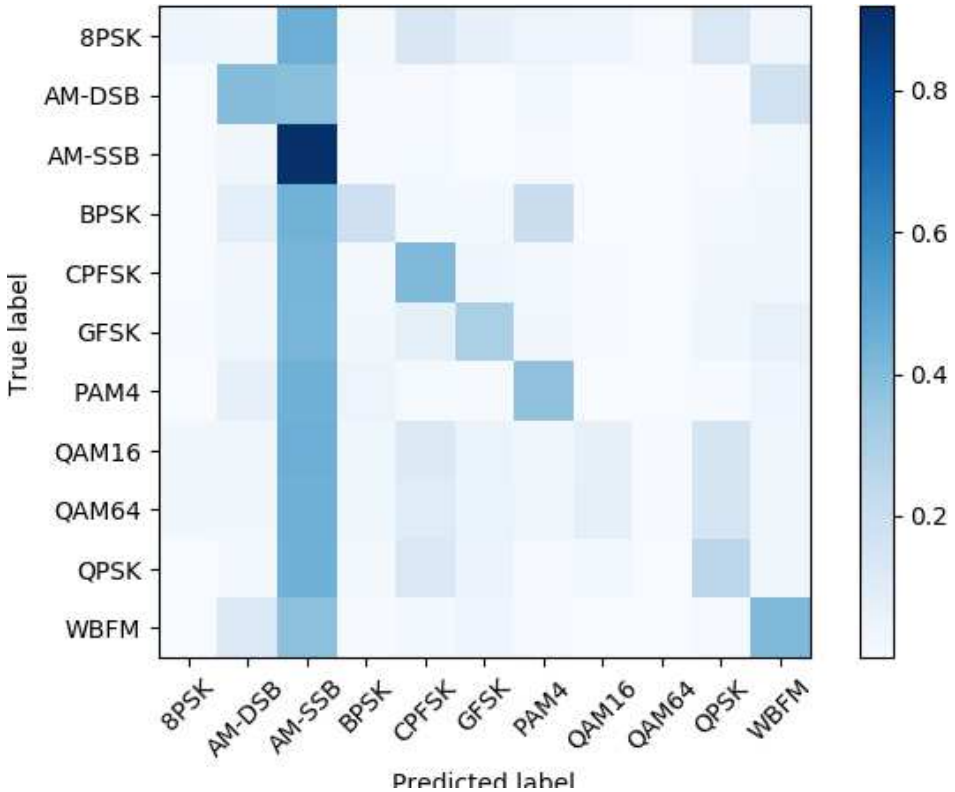


Figure 3.24: A confusion matrix obtained from a constant CFO (2.30) line drawn down the CFO axis of Figure 3.23 at 13% CFO normalized to sampling rate. The color gradient communicates classification accuracy averaged over SNR values ranging from -20 dB to 20dB. The horizontal axis displays the modulation scheme that the CNN classifies signals by, and the vertical axis the ground truth of those signals. A perfectly performing classifier would have a deep blue diagonal matrix, where each signal of each modulation type of each SNR is correctly classified by having the highest soft-max value at its index corresponding to the signals' ground truth label.

3.9 Chapter Summary

In this chapter, the fundamentals of deep learning was discussed, specifically topics concerning training. Additionally, Section 3.8 presented a survey of modulation classification, the primary form of neural network evaluation discussed in this thesis. This survey provides the knowledge required to understand and derive architectures for unsupervised domain adaptation in Chapter 5. Additionally, insight has been provided for the type of machine-learning based signal classifiers datasets generated from the framework in Chapter 4 may be used with.

In the next Chapter 4, a novel framework for low-decay, low-bias dataset generation is presented.

Chapter 4

Physical Layer Neural Network Framework for Training Data Formation

In this chapter, a low decay, low bias dataset synthesis framework is proposed that models Machine Learning (ML) dataset theory using Python classes and instruction files, and whose simulation results show an 11.58% entropy decrease at classification time relative to state-of-the-art training sets. There has been a growing interest in choosing appropriate training data in order to enhance NN performance at classification time. Developing ML based signal classifiers requires training data that captures the underlying probability distribution of real signals. To synthesize a set of training data that can capture the large variance in signal characteristics, a robust framework that can support arbitrary baseband signals and channel conditions is presented.

The proposed framework is intended to be used in conjunction with arbitrary wireless environments. Chapter 2 presents a survey of wireless environments that one might be interested in modeling. This work is intellectually controlled and its source code is consequently absent from the Appendix.

4.1 Introduction

Radio Frequency (RF) Neural Networks (NN) have recently received significant attention within the wireless research community [66, 67]. However, RF NNs do not possess the openly available datasets and established benchmarks that are frequently associated with other NN applications [5, 66]. Consequently, there is a need within the wireless community for establishing freely available datasets and benchmarks that can be used to evaluate and compare the implementations of RF NNs.

In a dense RF environment, a receiver must be able to first detect and isolate a transmission [1] before a RF NN can extract features and perform signal classification. This detection and isolation is impeded

by a variety of real-world impairments [27,28], which can negatively affect a NNs classification accuracy in highly time-varying and probabilistic environments. In recent publications [5,68], the effect on evaluation-time classification accuracy of signal bandwidth (BW), limits and statistics of powerful imperfections, and size of training sets has been explored but not fully understood.

For any signal-domain NN, an often used framework for dataset generation is the GNU Radio Companion (GRC) channel model blocks [5]. Each channel block offers a modular, sequential, and parallelized method of dataset manipulation. However, GRC cannot readily construct massive ensembles of varied wireless channels. Another resource for RF NNs is the RadioML 2016.10A dataset and Github repository, which contains their generation code [5]. While the dataset is popular due to its availability, some researchers have had difficulty in replicating the peak accuracy of their classifier [69], and the statistics of the datasets are bound to a single, time-invariant channel model. Should a test set stray too far in terms of Signal-to-Noise-Ratio (SNR), pulse shaping, Local Oscillator (LO) drift, or some other parameter, the classification accuracy can potentially suffer [68] as a result of data bias, or training under a false assumption. A strength of signal-domain ML is that it is easy to simulate more data in order to grow a dataset. Consequently, as long as the memory and computational resources exist, generating large datasets that cover numerous permutations of channel imperfections would be a valuable tool with respect to the classification of real signals.

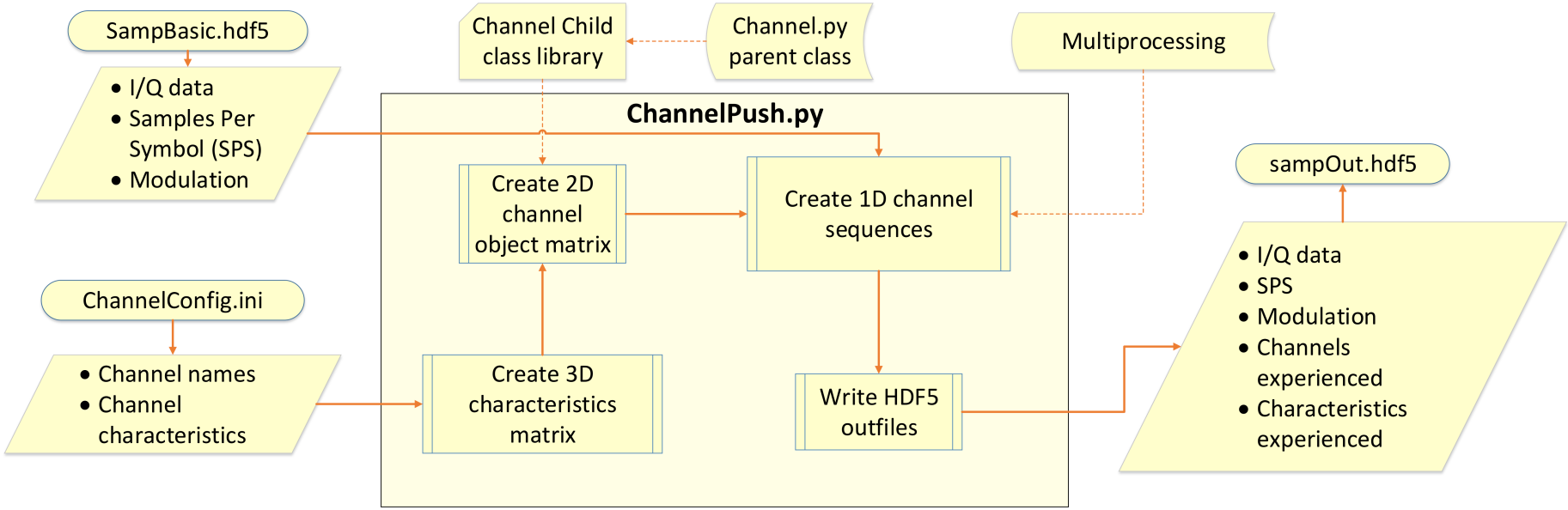


Figure 4.1: Illustration of the proposed framework and the ChannelPush.py script. SampBasic.hdf5 acts as the Dataset Under Test (DUT) while ChannelConfig.ini as the instructions file. SampOut.hdf5 files are written as outputs. The 3D matrix is formed by the instructions file, containing the 2D matrix's (see Table 4.1) instance variables. The 2D matrix objects are formed by run-time channel class imports. 1D channel sequences (see Figure 4.2) are formed by permuting the channel imperfection objects from the 2D matrix, and the DUT is pushed sample by sample through each sequence in parallel.

The work presented in this chapter possesses the following contributions to the current state of the art:

- A brief list of desirable variations for signal-domain ML datasets to achieve full coverage of the statistics of real signals.
- A framework that synthesizes datasets with the objective of reducing data decay and data bias, and capturing the underlying statistics of real signals.
  - Channel imperfection models are sequentially applied to input baseband signals modularly, where channel imperfections' constants and random variables (RV) are defined through the use of an instructions file.
- Example transmissions synthesized with our framework, constrained by parameters corresponding to relevant hardware specifications and signal structures.
- Simulation results showing that the generated datasets contain low entropy at evaluation time, and guidelines for generating diverse and robust RF-domain datasets through Kullback-Leibler Divergence (KLD) entropy analysis of approximations of Probability Density Functions (PDFs).

The rest of this chapter is organized as follows: The proposed framework and dataset variations are presented in Section 4.2, and applications of the framework in Section 4.3. Section 4.4 presents KLD entropy analysis of training set approximations of PDFs and their implications on training set size, and concluding thoughts are presented in Section 4.5.

4.2 Proposed Framework

The proposed framework is described in Figure 4.1. The framework requires a Dataset Under Test (DUT) as well as instructions for which channel imperfection models are to be applied to the DUT. The framework outputs are instances of the DUT that have been modified by a unique permutation of channel imperfections. Channel imperfection models may be added, modified, or removed from the framework by minimal editing of the instructions. Each 1D sequence's output is computed and written in a separate Central Processing Unit (CPU) process.

Regarding the size of an RF dataset and the choice of information it should contain, two pitfalls that need to be avoided when building a dataset for ML use are data decay and data bias, which are defined as follows:

- *Data decay* is the gradual decrease in testing accuracy over time as simulated training data and empirical testing data have statistically drifted apart. In the RF domain, data decay can be caused by hardware advances or changes in communication standards.

- *Data bias* is any large difference in training and testing accuracy due to the training set being designed based on false assumptions. Examples for this are numerous, ranging from false assumptions about Signal-to-Noise-Ratio (SNR) and other channel characteristics to false assumptions about hardware condition and use.

A framework is useful for the prevention of data decay since it allows for flexible and easy generation, keeping datasets current and relevant. Data bias can be avoided through the use of a framework by influencing training data by a wider range of effects than expected at evaluation time. A framework makes this tuning process simple and quick using small edits to an instructions file designed to maximize the amount of information contained in commands relevant to data bias and decay (*i.e.*, link budget parameters).

Table 4.1: Example 2D Channel Object Matrix (refer to Figure 4.1). Objects are instances of run-time imported Carrier Frequency Offset (CFO) and Additive White Gaussian Noise (AWGN) Python classes. Instance variables of the objects are imported from the 3D characteristics matrix. Some characteristic sweeps should be linearly spaced (phase ambiguity in radians), and others log spaced (SNR of an AWGN model)

Channel	Feature	Variations	
1	AWGN1	Variance: 0.01	SNR: 0 dB
	AWGN2	Variance: 0.01	SNR: 20 dB
	AWGN3	Variance: 0.1	SNR: 0 dB
	AWGN4	Variance: 0.1	SNR: 20 dB
2	CFO1	offset norm to samp rate: 2.5%	
	CFO2	offset norm to samp rate: 5.0%	

ML datasets have K categories (or labels), $y = \{0, 1, ..., K-1\}$, and N examples (*i.e.*, images, text blocks, or an RF waveform) of dimensionality D, $x = [N \times D]$ [2]. In the context of a modulation classifier of IQ datasets, K is the number of modulation schemes to be used as labels, D is the dimensions of a training signal, with a length equal to the number of complex samples per signal and a depth of 2 representing the in-phase and quadrature components of each complex sample, and N is the number of transmissions in the dataset.

Statistical heuristic methods of varying complexities suggest different amounts of data to fully represent the underlying statistics of a set of transmissions for the purpose of classification [70]. A popular and simple heuristic that we consider represents the number of training examples N as the product:

$$N = K \times C \tag{4.1a}$$

$$C = (f \times F) \times (v \times V) \tag{4.1b}$$

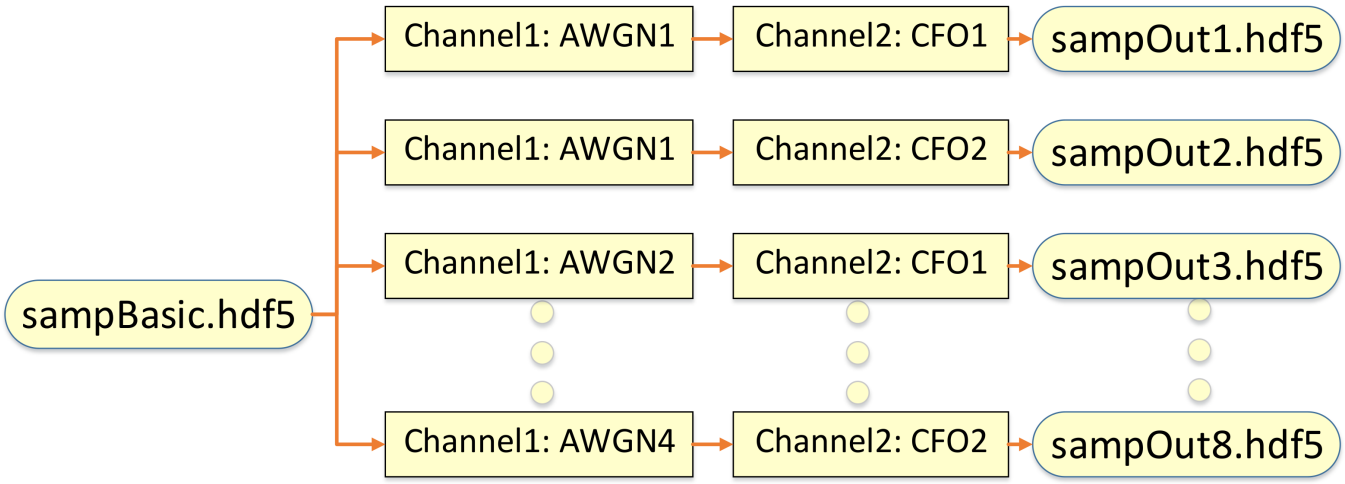


Figure 4.2: Example set of eight 1D channel sequences (refer to Figure 4.1) formed by permuting through the 2D channel object matrix. SampBasic.hdf5 is the DUT, and is pushed through each sequence sample by sample, leveraging Multiprocessing.

where C is the number of training transmissions per class, F is the number of transmissions per input feature the dataset has, V is the number of transmissions per variation of those input features, and f and v are the number of input features and variations. The robust computer vision datasets CIFAR-10 and CIFAR-100 [71] choose a C value of 6,000 and 600, respectively, and the RF dataset RML2016.10A 1,000 [67].

To avoid data bias, one wants to include as many channel imperfections in training transmissions as possible in a signal-domain ML training set to avoid training under a false set of assumptions of channel conditions (see Table 4.2 for examples). It is important to note that not all of these input features will have a significant effect on evaluation time accuracy, which is why the framework is designed to easily add and remove channel imperfection models, which rescales the number of training examples size by an integer change of:

$$\Delta N = (\Delta f \times F) \times (\Delta v \times V). \tag{4.2}$$

While several input variations are displayed in this chapter, the proposed framework is designed with the explicit future purpose of allowing for an endless contribution of channel imperfection models from the physical layer ML community in addition to the list in Table 4.2. Refer to Chapter 2 for a survey of wireless environments that one might be interested in modeling when implementing the proposed framework.

Many features have well defined classical models that describe their behavior, excluding certain non-linear phenomenon such as amplifier non-linearities [72]. It is the goal of this proposed framework to treat each one of these channel model imperfections as functions $f(a, b, ...)$ described by variations $a, b, ...$, which serve as function inputs.

In the proposed framework, the choice of features and variations to be used in an experiment are

Table 4.2: Examples of variations in computer vision image datasets, and a collection of analogies for their signal domain parallel [2].

Computer Vision	Communications
Orientation	Phase ambiguity
Size	Signal amplitude
Deformation	AWGN, STO, more
Lighting	Frequency-fading multi-path
Occlusion	Signal jamming
Camouflaged	Co-band, neighbor-band interference
Intra-class Variation	Alternate modulation (i.e. non-rect QAM)
Image stretching	IQ imbalance
Motion Blur	Frequency offset (i.e. CFO, Doppler shift)

described by the instructions file. Instruction files have two lines of code per channel imperfection model, with one describing a key-value pair of the imperfection's name, and the other a key-value pair describing the imperfection model's function inputs (variations). The framework is described by Algorithm 2.

4.3 Applications of Proposed Framework

In order to obtain real-world wireless data, we used in this work a USRP N210 software-defined radio (SDR) from Ettus Research employing an SBX daughter board [73]. Datasets influenced by state-of-the-art radio front-end imperfection models are valuable because the radios are current and widely-used, and thus the datasets are resistant to data decay and bias. In this instruction file, imperfection models are defined by data sheets describing the Ettus N210 with an SBX daughter-board [29]. Figure 4.3 illustrates an example transmission from the dataset, x_i of dimensionality D, where each sample of x_i is a complex tuple (I, Q). Each x_i represents a transmission sent from an N210 impacted by CFO, AWGN, STO, and phase ambiguity, the four of which are believed by the authors to be amongst the most influential RF variations on IQ shape and behavior, and thus evaluation-time accuracy of real signals.

Most RF environments are not occupied by a single signal, but by a dense clutter of unique signals organized by time-varying Medium Access Control (MAC) and network-layer protocols. Consequently, certain channel imperfections such as multi-path fading can be correlated across multiple transmissions (*i.e.*, due to common landmarks), while imperfections such as a transmitter and receivers LO drift are statistically independent. To synthesize training and testing sets that use state-of-the-art signal structures, these imperfections need to be appropriately correlated and applied, and their transmissions summed.

Algorithm 2 The proposed framework in Figure 4.1 implementing KLD analysis (see Figure 4.6) to properly size datasets.

procedure *Input*: ARBITRARY BASEBAND SIGNAL $b[n]$, PARAMETERS A, AND k STOCHASTIC PROCESSES $\{X_k(a_k)\}_{a_k \in A}$, ENTROPY CRITERION E
  repeat
  for $a_k \in A$ **do**
    for constant a_{k_i} **do**
      for j values **do**
        define $\{X_{k_{1,j\ldots,i,j}}(a_{k_{1,j}}, ..., a_{k_{i,j}})\}$
      end for
    end for
  end for
  define $m = k \times i \times j$ series $X_m = \binom{X_{k,i,j}(a_{k,i,j})}{k}_m$
  for X_m **do**
    framework noisy signal $s_m[n] = X_m(b[n])$
  end for
  calculate $KLD(s_m[n], X_m) = e$
  if $e > E$ **then**
    redefine $b[n]$
  else
    entropy criterion satisfied, **exit**
  end if
end procedure

Figure 4.4 presents a set of three separate transmissions produced from the proposed framework in pursuit of this goal.

4.4 Simulations and Results

In the previous section, the number of examples a training set should have and what information those examples should contain was discussed. As shown in [68], the number of observations or samples that are contained in an example can have a significant impact on training time and testing accuracy of a NN. In this section, it is investigated how many samples pulled from RVs are needed to estimate their theoretical PDFs to within a specific degree of accuracy.

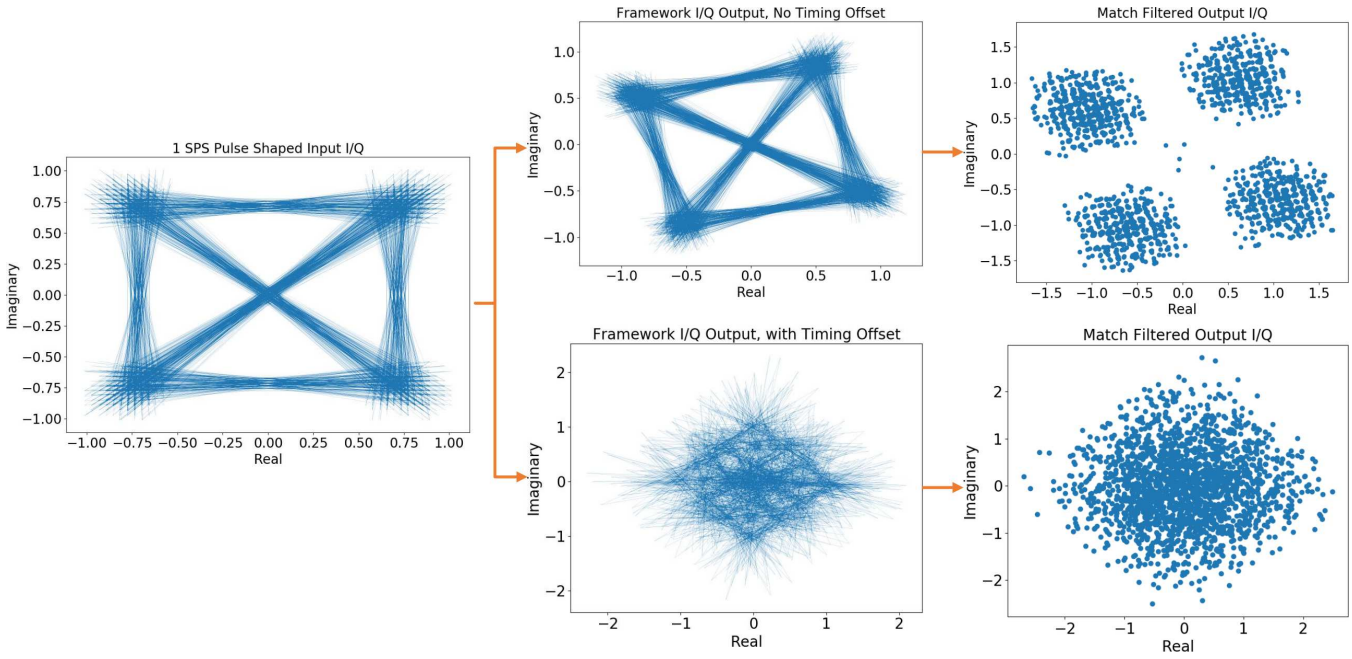


Figure 4.3: 1 SPS pulse shaped Quadrature Phase-Shift Keying (QPSK) IQ data representing the baseband data of an Ettus Research N210 transmission. For the sake of visualization, frequency offset from Local Oscillator (LO) drift has been left out. The top track displays the dataset influenced by phase ambiguity and AWGN, then the matched filtering of that data. The bottom track additionally shows STO, where the data is interpolated and filtered up to an intermediate 2 SPS, offset in time, then decimated (and once again match filtered like the top track).

It is desirable to have the number of samples be integers of base two since it is computationally efficient [2] and allows for an integer number of symbols to be contained in each training example, as most commonly SPS is also base two. Furthermore, each training example needs to create enough instances of the RVs that govern its variations such that a PDF formed from that data is similar to the theoretical PDF. In this way, testing data can be correctly classified because the NN has learned the behavior of the data. The Central-Limit Theorem (CLT) states that when properly normalized, independent RVs are summed, the distribution tends towards a Gaussian one. Furthermore, the Law of Large Numbers states that the average result of a certain number of trials tends towards the expected value as the number of trials increases. To investigate the implications of these two laws in signal-domain dataset synthesis as they pertain to Figure 4.3, KLD analysis (4.3) is performed, with results shown in Figure 4.6:

$$D_{KL}(P||Q) = \sum_i P(i) log_2 \left(\frac{P(i)}{Q(i)} \right), \tag{4.3}$$

where the expected probability $P(i)$ can be defined as the definition of a Gaussian PDF:

$$P(i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\pi\sigma^2}}, \tag{4.4}$$

and the measured probability $Q(i)$ is defined as the histogram formed from the data $q(i)$:

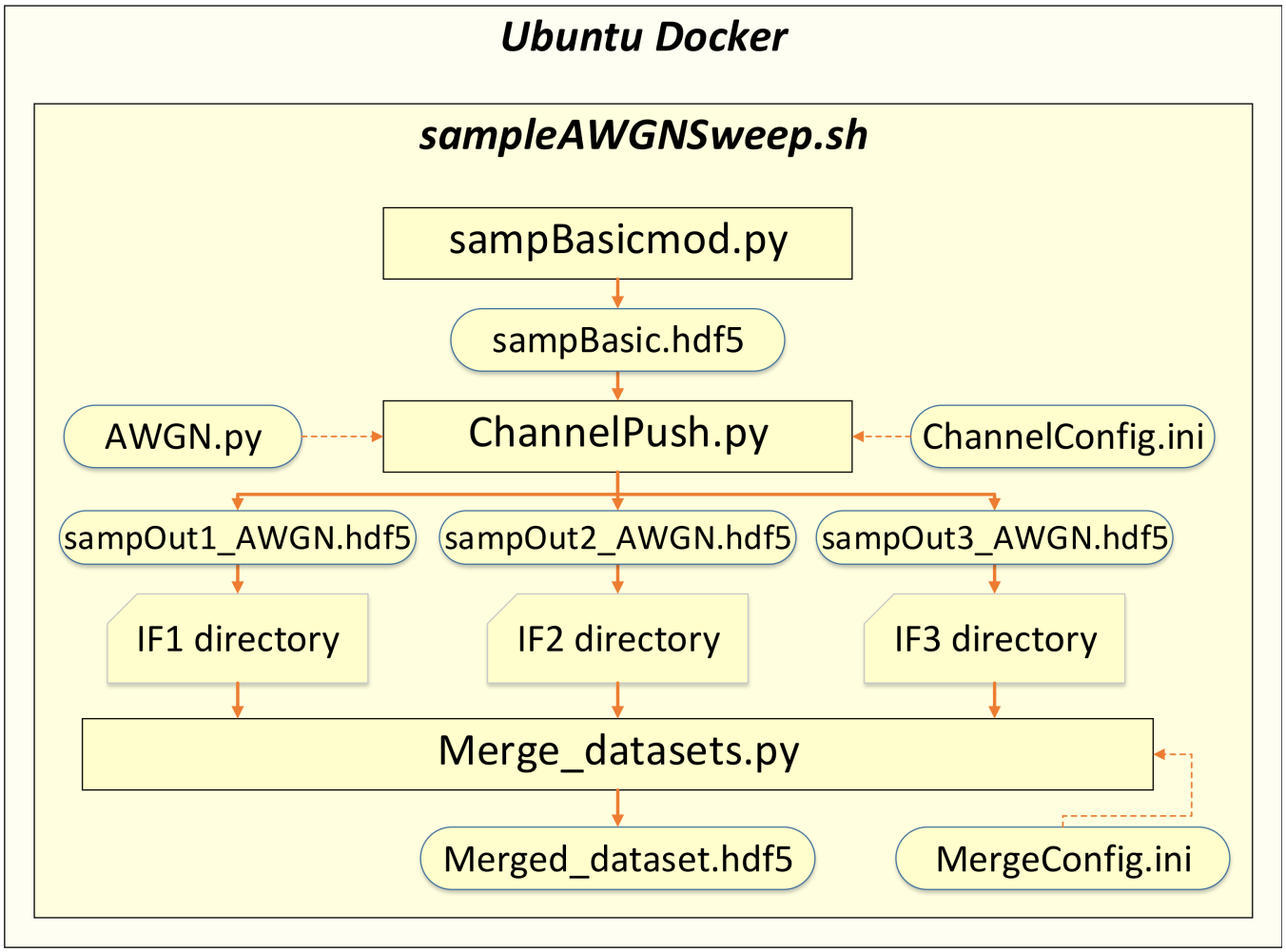


Figure 4.4: The AWGN channel effect is described by its SNR and Gaussian RV variance, σ. Three AWGN channels of varying SNR but constant σ described by ChannelConfig.ini are applied to the same infile sampBasicmod.py. The outputs of which are manually moved to Intermediate Frequency (IF) folders corresponding to a secondary instructions file, MergeConfig.ini. Merge_datasets.py (see Figure 4.5) modulates and sums the independent transmissions.

$$q(i) = CDF_{norm}^{-1}(u) + eps, \tag{4.5}$$

where $eps \sim U(0, 0^+)$ to avoid $log_2(0)$ errors in (4.3) resulting from zero-observation bins, $u \sim U(0, 1)$, and $CDF_{norm}^{-1}(u)$ is the inverse Cumulative Distribution Function (CDF) of the Gaussian distribution:

$$CDF_{norm}(x) = \frac{1}{2}\left[1 + \text{erf}\,\frac{x-\mu}{\sigma\sqrt{2}}\right]. \tag{4.6}$$

RML2016.10A is a dataset generated using GRC's dynamic channel model, which combines their Symbol Rate Offset (SRO), CFO, and flat-frequency fading models. The probabilistic nature of the dataset is dictated by the SRO and CFO Gaussian RVs, and the AWGN Gaussian RV. Our proposed application (see Figure 4.3) is described by a uniform RV from a phase ambiguity and STO model, two Gaussian RVs

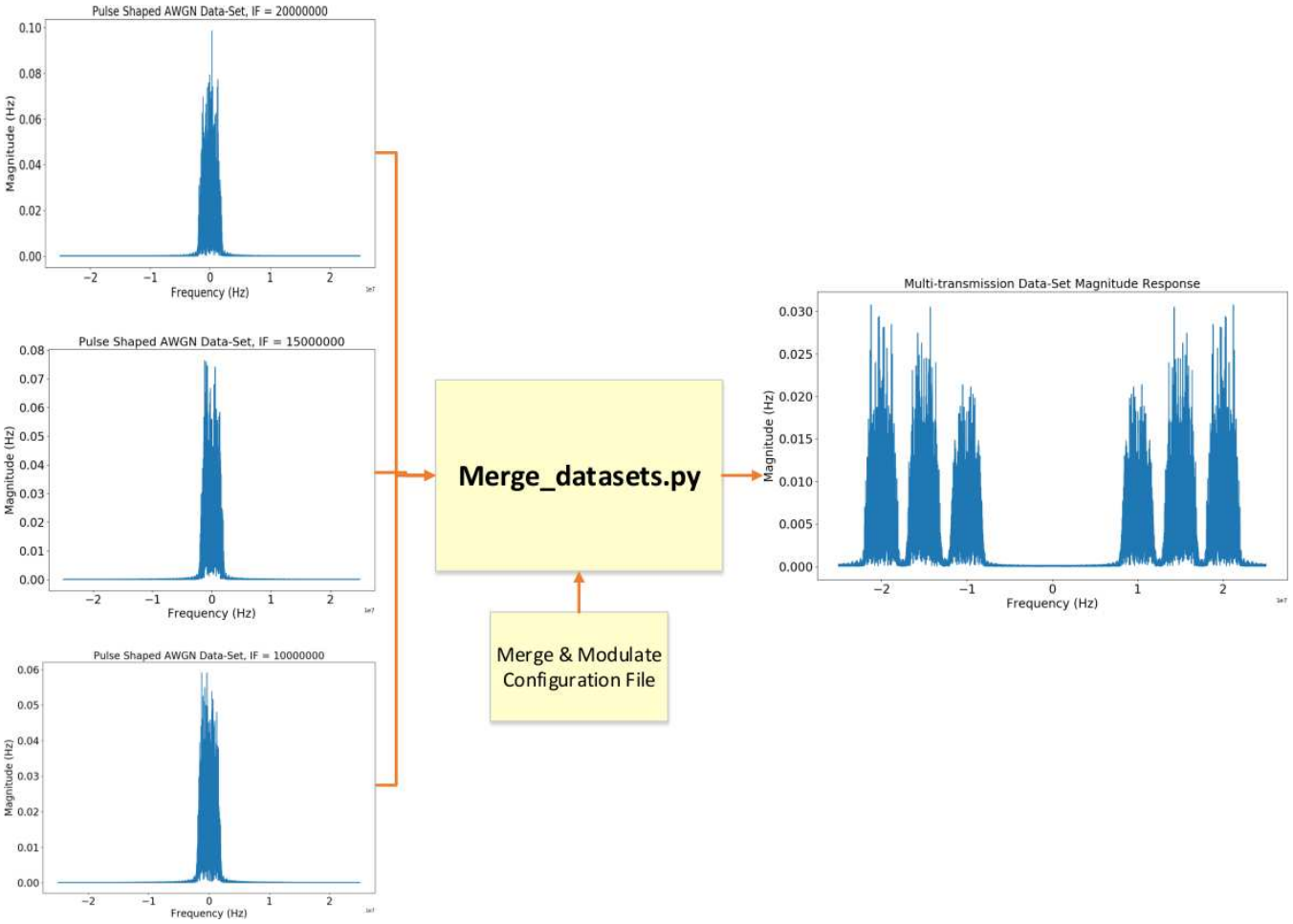


Figure 4.5: Three 16 SPS pulse shaped QPSK datasets from Figure 4.4 are modulated to intermediate frequencies 10, 15, and 20 MHz. Each dataset was pushed through the framework as a DUT and modified by a unique AWGN channel block independently, each representing a transmitted signal. Future work will implement this feature to produce MIMO and OFDM datasets.

describing the CFO of a transmitting and receiving LO, and a Gaussian RV contained in the AWGN model. Since KLD is additive (4.7) for independent distributions (*i.e.*, P_1, P_2, Q_1, Q_2), the values in Figure 4.6 are calculated as the sum of KLD for each RV of the dataset under consideration.

$$D_{KL}(P||Q) = D_{KL}(P_1||Q_1) + D_{KL}(P_2||Q_2). \tag{4.7}$$

4.5 Chapter Summary

The proposed framework showcases the ability to model fundamental ML theory through the use of an instruction file and a library of Python classes. The proposed framework has low data decay and data bias, and the KLD entropy analysis shows that an application of the proposed framework is properly sized, achieving 11.58% less entropy than state-of-the-art datasets at classification time.

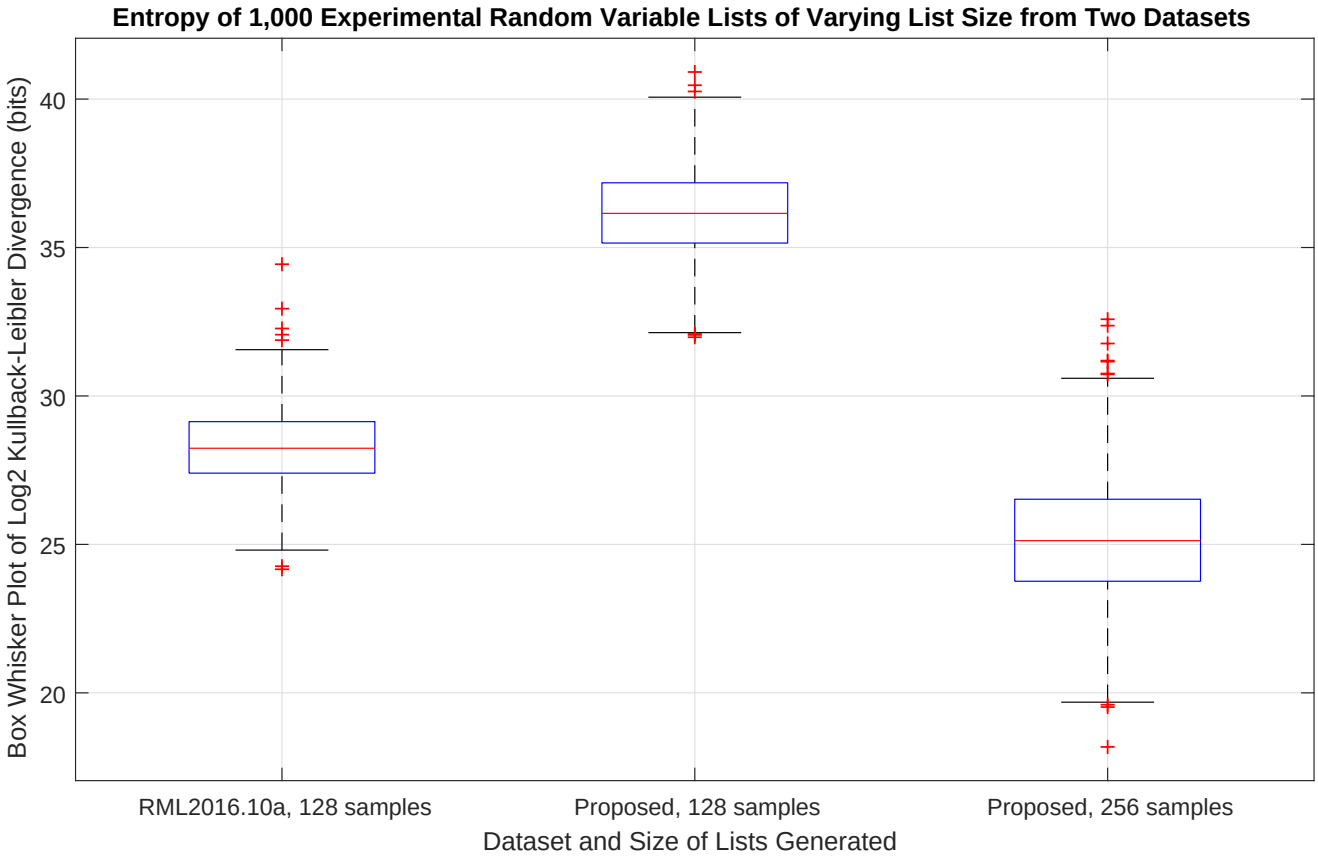


Figure 4.6: RML2016.10A is composed of 1,000 training sets containing 128 samples each per class per SNR value. Transmissions average 28.3 bits divergence from theory. The proposed application (see Figure 4.3) averages 36.2 bit divergence from theory. In order to achieve similar KLD entropy at an RF NNs evaluation time to state-of-the-art datasets, this analysis shows the proposed application requires at least 256 samples per transmission. The resulting divergence from theory is 25 bits, or a 11.58% decrease from RML2016.10A.

Chapter 5

Domain Adaptation of Wireless Channels

Despite training machine-learning based signal classifiers on permutations of plausible noise models, unforeseen test perturbations are still likely to remain and result in classification errors. What else can be done to get classifiers to perform as if test and training data was statistically equivalent?

In this chapter, a novel application of domain adaptation is presented to combat dataset bias of wireless transmissions using a CNN. Using the proposed method, it is attempted to maintain peak testing accuracy despite substantial dataset bias resulting in a loss of 16% peak testing accuracy. Research on this topic has just begun and research is ongoing.

5.1 Introduction

Dataset bias [74] presents a significant problem for NNs across many industries. Bias arises when false or incomplete assumptions are made about testing data, and there exist statistical differences between testing and training data either due to unforeseen perturbations or incomplete or low quality training data.

Domain adaptation [60] is a method of treating dataset bias, motivated by the automotive industry to construct images for computer vision NNs out of 3D rendered graphics of roads. It is a loosely defined field that also goes under the names of class imbalance [75], covariate shift [76], and sample selection bias [77]. The method involves reconstructing data from a source domain into a target domain (see Chapter 2). However, as far as the the authors know, this method has only been applied to computer vision scenarios.

The parallel drawn in this chapter is that source and target domains can represent a statistical space of stochastic processes (see Chapter 2) corresponding to a real world transmission utilizing a specific set of hardware and a specific wireless medium. Given the time-varying nature of wireless channels, even wireless NNs trained on Over-The-Air (OTA) data can suffer dataset bias and decay by the time the testing phase is reached. It is the goal of the authors to investigate maintaining peak modulation classification accuracy

by performing domain adaptation transforms on deep features learned from training data transmitted using certain hardware over a certain wireless medium, but tested using data sent over a different wireless medium using different hardware. The method of domain adaptation used at the infancy of this work is the simpler Deep Reconstruction-Classification Network [6] (DRCN). This method is categorically a form of unsupervised domain adaptation, meaning data samples used from the target domain to learn transforms are not labeled, or contain no ground truth modulation scheme.

The work presented in this chapter suggests the following contributions to the current state of the art [60]:

- A novel reconstruction network application that can be used to reconstruct wireless transmissions from one set of stochastic processes to another via a characteristic-agnostic wireless channel domain.
- A novel application of method of dataset bias reduction via unsupervised domain adaptation.

The rest of this chapter is organized as follows: Section 5.2 describes the DRCN architecture and its application in the communications field, Section 5.3 describes the training and testing of the DRCN, and concluding thoughts are discussed in Section 5.4.

5.2 System Architecture

A domain is described as a PDF over the input space X and output space Y, denoted as $\mathbb{D}_{XY}$. In unsupervised domain adaptation, labeled samples generated from a source domain $\{(x_i^s, y_i^s)\}_{i=1}^{n_s} \sim \mathbb{S}_X$ and unlabeled samples from the target domain $\{(x_i^t)\}_{i=1}^{n_t} \sim \mathbb{T}_X$ are mapped by the labeling function $f : X \to Y$ and $g : X \to F$ such that the domains $\mathbb{S}$ and $\mathbb{T}_X$ are i.i.d. and minimally different by designing F.

To define this projects mappings, let the supervised learning mapping be $f_c : X \to Y$ and the unsupervised mapping be $f_r : X \to X$. Additionally, let the encoder feature mapping which transforms features to the characteristic-agnostic domain $\mathbb{Z}$ be $g_{enc} : X \to F$, the decoder mapping the $\mathbb{Z}$ domain to the target domain be $g_{dec} : F \to X$, and the featuring label mapping be $g_{lab} : F \to Y$. See Figure 5.1 for an overview of these mappings in relation to domain space.

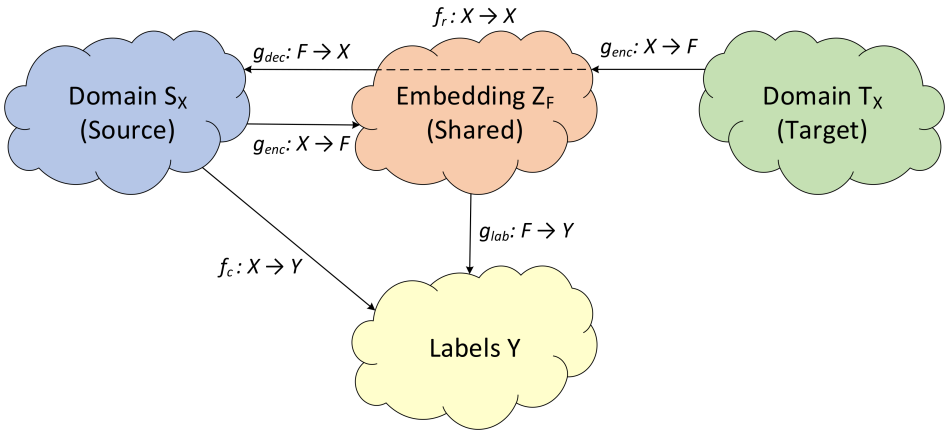


Figure 5.1: An overview of the DRCNss domain mappings and the space each domain occupies. See (5.2) to see how they're used in the DRCNs objective function.

Given a signal from the source input space, $x \in X$, the mappings f_c, f_r can be expressed as:

$$f_c(x) = (g_{lab} \circ g_{enc})(x), \tag{5.1a}$$

$$f_r(x) = (g_{dec} \circ g_{enc})(x). \tag{5.1b}$$

To define the DRCN's learning parameters, let $\theta_c = \{\theta_{enc}, \theta_{lab}\}$ denote the supervised domain adaptation model's weights and bias' and let $\theta_r = \{\theta_{enc}, \theta_{dec}\}$ denote the unsupervised model's.

To define the DRCN's loss function, let the signal space $x \in X$ and label space $y \in Y$ (in this case representing 11 modulation schemes, $y \in \{0,1\}^{11}$) be used to define the classification loss function $l_c : Y \times Y$ and the reconstruction loss function $l_r : X \times X$. The loss at any point then in SGD is defined as:

$$L_c^{n_s}(\{\theta_{enc}, \theta_{lab}\}) \triangleq \sum_{i=1}^{n_s} l_c(f_c(x_i^s \ \{\theta_{enc}, \theta_{lab}\}), y_i^s), \tag{5.2a}$$

$$L_r^{n_s}(\{\theta_{enc}, \theta_{dec}\}) \triangleq \sum_{j=1}^{n_t} l_r(f_r(x_j^t \ \{\theta_{enc}, \theta_{dec}\}), x_j^t). \tag{5.2b}$$

The objective function of the DRCN is then defined as:

$$L \triangleq \min_{\theta} \lambda L_c^{n_s}(\{\theta_{enc}, \theta_{lab}\}) + (1-\lambda) L_r^{n_t}(\{\theta_{enc}, \theta_{dec}\}), \tag{5.3}$$

where $0 < \lambda < 1$ is a hyper-parameter giving a bias or importance to classification or reconstruction. The DRCN architecture (see Appendix C.4) is composed of two sections, as shown in Figure 5.2.

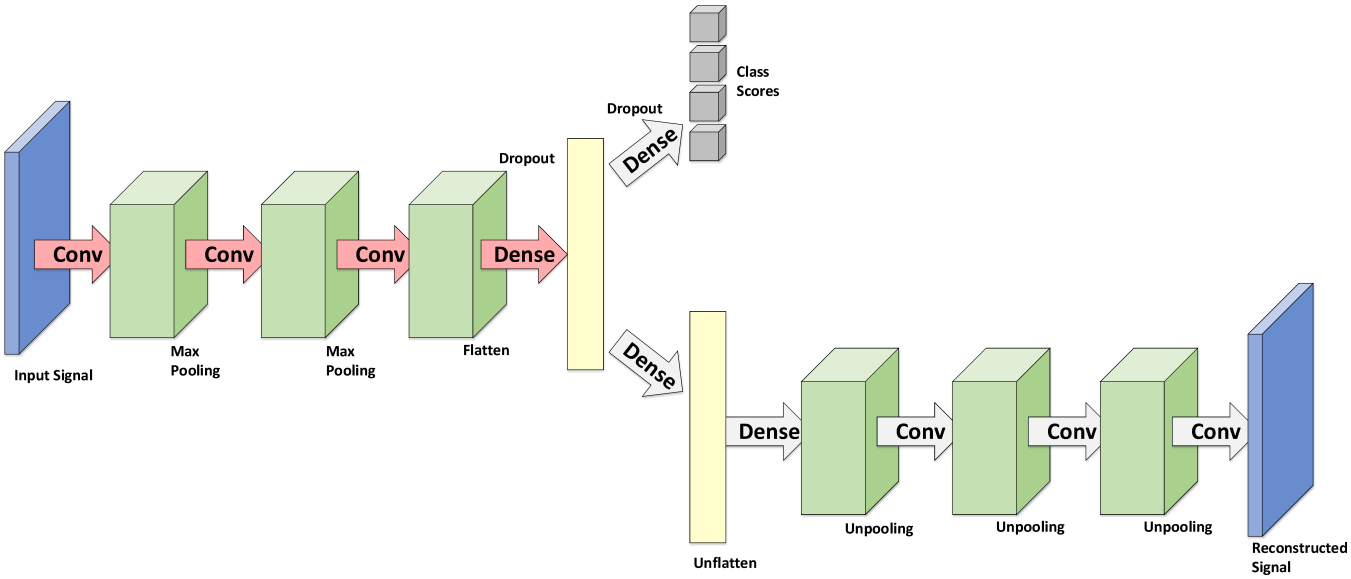


Figure 5.2: The DRCN (adapted from [6]) is composed of two sections: the Convnet [3] classification NN (left) and reconstruction Convae [6] NN (right). The classification NN performs source label prediction and is composed of three convolutional layers of depth (number of filters) $n_b = [100, 150, 200]$ where filter size is 3×3. Each max pooling layer condenses a 2×1 grid of values into one equal to the largest of the four. Dropout probability $\rho = 0.5$, dense layers have 1024 neurons, and the soft-max classification layer has 11 outputs, one for each modulation scheme. The reconstruction NN is an auto-encoder that performs data reconstruction. This teaches commonalities between classifying in both domains, helping the formation of F, transforming to a characteristic-agnostic domain.

The RML2016.10a dataset [5] and its target domain counterpart are described by Table 5.1. Each dataset contains 11 modulation labels and 20 E_s/N_0 labels ranging from -20 dB to +20 dB, for 220 class combinations of labels. Each label pair generates 1,000 transmissions containing 128 complex valued 64-bit (double-precision) samples representing IQ data. Having both multiple samples per transmission and multiple transmissions per class combination of labels allows the random variables that drive the stochastic wireless channel processes to generate a number of samples sufficient enough to realize the processes theoretical probability functions, in accordance with the law of large numbers. The design goal of the target domain wireless channel is for it to be significantly statistically different from the source domain wireless channel, as to challenge this chapter's hypothesis most strongly, but not for it to be too significantly noisier or cleaner, as that would simply reflect on the CNNs ability to classify noisy data.

5.3 DRCN Results

Training and testing was done on Worcester Polytechnic Institute's Turing High-Performance Computing cluster (HPC). Jobs were submitted using bash scripts (see Appendix B.1) calling a meta file (see Appendix C.1).

Table 5.1: A summary describing the statistical differences between the source domain and target domain datasets. Maximum Doppler frequency is denoted as f_D, multi-path taps are defined by their time delay τ and amplitude a, N_{sin} describes the number of sinusoids used in the frequency-selective fading model, F_S is the sampling frequency of the simulated transmitter and receiver, Δ_{max_t} and Δ_{max_f} are the maximum symbol rate and carrier frequency offsets, σ_t and σ_f are the standard deviations of the Gaussian-distributed symbol rate and carrier frequency offsets, and K is the Rician K-factor ratio of specular to diffuse power.

Variable	**Source Domain**	**Target Domain**
f_D	1 Hz	10 Hz
τ	$[0, 0.9, 1.7]$	$[0, 0.5, 0.7, 1.5, 2.1, 4.7]$
a	$[1, 0.8, 0.3]$	$[0.8, 1, 0.5, 0.25, 0.2, 0.08]$
N_{sin}	8	2
F_S	200 kHz	50 kHz
Δ_{max_t}	50 Hz	500 Hz
σ_t	0.01	0.1
Δ_{max_f}	500 Hz	5 kHz
σ_f	0.01	0.1
K	4	40

5.3.1 Training

Partitioning was done according to a 0.6/0.2/0.2 training/validation/testing ratio (see Appendix C.2). Mini-batch training was then completed on the Convnet and DRCN architectures according to Table 5.2 using a batch size of 1024 over 30 epochs (see Figure 5.3).

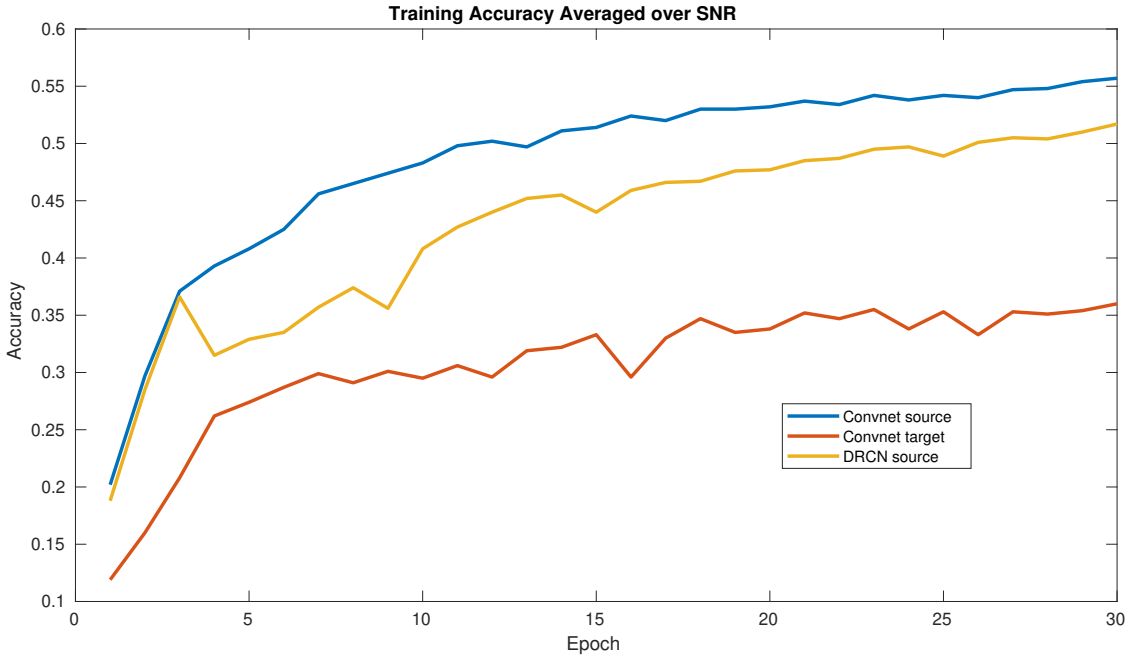


Figure 5.3: Convnet and DRCN training accuracies at each epoch of SGD. The legend indicates whether the DRCN or Convnet architecture was used, and the domain the training partition was from.

5.3.2 Testing and Discussion

Test results were printed to the terminal through a .out file (see Appendix B.4). Architectures were tested against partitions as describe in Table 5.2. The Convnet used in the DRCN to encode/classify (see Figure 5.1) is trained and tested against partitions of the source (experiment 1, see Appendix B.2) and target (experiment 2, see Appendix B.3) domain data to establish benchmarks and the difference in how challenging the wireless channels are. Ideally, the channels should be equally challenging and have similar accuracies. This would show that the decrease in accuracy from experiment 1 to 4 is not caused because the wireless channel is more challenging, but because of dataset bias. Additionally, the Convnet is trained and tested against statistically different data (experiment 4, see Appendix B.5) to show the penalty of dataset bias, hypothesized to result in near-floor accuracy of 9%, or 1/11 (the number of modulation labels). Finally, the DRCN is likewise trained and tested (experiment 3, see Appendix B.4), hypothesized to obtain a peak testing accuracy between (1) and (4), reducing the affect of dataset bias on classification.

5.4 Chapter Summary

In this chapter, the state of the art of domain adaptation in communications was discussed in Section 5.1, Section 5.2 described the DRCN architecture and its application in the communications field, and Section 5.3 described the training and testing of the DRCN.

Table 5.2: At each stage of training the Convnet classifier is evaluated against a 20% testing partition. The peak testing accuracy obtained over 30 epochs is presented below for each of the four experiments.

Experiment	**Architecture**	**Dataset (train/test)**	**Peak Accuracy**
1	Convnet	source/source	51.7%
2	Convnet	target/target	35.1%
3	DRCN	source/target	41.7%
4	Convnet	source/target	43.5%

The data bias had a smaller effect on the Convnet than hypothesized (expected accuracy to be at the accuracy floor 9%, but instead only dropped from 51.7% to 43.5%). Although the DRCN has not yet shown the ability to compensate for dataset bias via data reconstruction (experiment 3 in Table 5.2 performed at best equally to experiment 4), its effectiveness in computer vision [6] warrants further investigation in this novel application. Research continues to discover why the ConvAE decoder is ineffective, which could be caused by a low learning capacity or over-fitting in the classification/decoder architecture. Similarly, the two wireless channels in Table 5.1 may be too different for reconstruction to be learned by the chosen architecture. Finally, it may be the case that signals are too low dimension for this computer vision technique, and no choice of architecture or dataset will result in successful reconstruction.

Chapter 6

Conclusion

In Chapter 2, a survey of background knowledge learned by the author on the topics of wireless channel modeling. Chapter 3 surveyed neural networks with an emphasis on training and data sets, and modulation classification. Chapter 4 showcased work on generalized training through the development and use of a low bias, low decay framework that synthesizes low-entropy data sets modeling state-of-the-art waveforms. Finally, Chapter 5 presented ongoing work on generalized training through the use of the domain adaptation technique.

6.1 Research Outcomes

The work presented in this thesis has resulted in the following research outcomes:

- A survey of wireless channel environments was given in Chapter 2 that can be used in the proposed framework for dataset generation.
- A survey of machine-learning based signal classification was given in Chapter 3 that can be used to implement an unsupervised domain adaptation architecture from Chapter 5.
- A framework for wireless transmission dataset synthesis, implementing arbitrary channel environments and baseband waveforms.
- A dataset generated using the framework from Section 4 was proposed, and was shown to be properly sized, having 11% lower entropy than state-of-the-art datasets.
- A Deep Reconstruction-Classification Network (DRCN) was proposed in Chapter 5, which attempts to maintain peak classification accuracy despite heavy data bias resulting in a 16% peak testing accuracy drop compared to an experiment with all else equal but no data bias. These contributions

show both data-side (pre-training) and testing-phase manipulations to increase NN generalization and avoid retraining.

6.2 Future Work

Given the outcomes of this work, the following future tasks are of interest to the authors:

- Improving the generalization of the DRCN method presented in Chapter 5. This could be caused by a low learning capacity or over-fitting in the classification/decoder architecture. Similarly, the two wireless channels may be too different for reconstruction to be learned by the chosen architecture. Finally, it may be the case that signals are too low dimension for this computer vision technique, and no choice of architecture or dataset will result in successful reconstruction.

- Investigating Ghost Batch Normalization [18] (GBN) as a method of increasing generalizaiton. None of the contributions in this thesis are meant to totally solve the problem of dataset bias. It is of the interest of the author to see how much GBN can contribute.

- Developing a method of using Jacobian Norm and Transition Matricies [19] to manipulate generalization sensitivity, making communications NNs robust to perturbations. Changes to a classifier's architecture can change the number of activations given different data samples and the way class scores are changed by data perturbations.

Bibliography

[1] K. Pahlavan and A. H. Levesque, *Wireless information networks*. John Wiley & Sons, 2005, vol. 93.

[2] S. Y. Fei-Fei Li, Justin Johnson, "Cs231n convolutional neural networks for visual recognition," *http://cs231n.github.io/*, Stanford University, April 2018.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, 1998, pp. 2278–2324. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665

[4] N. E. West and T. O'Shea, "Deep architectures for modulation recognition," in *2017 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, March 2017, pp. 1–6.

[5] T. J. O'Shea and N. West, "Radio machine learning dataset generation with gnu radio," in *Proceedings of the GNU Radio Conference*, vol. 1, no. 1, 2016.

[6] M. Ghifary, W. B. Kleijn, M. Zhang, D. Balduzzi, and W. Li, "Deep reconstruction-classification networks for unsupervised domain adaptation," *CoRR*, vol. abs/1607.03516, 2016. [Online]. Available: http://arxiv.org/abs/1607.03516

[7] A. Patnaik, D. E. Anagnostou, R. K. Mishra, ChristodoulouCG, and J. C. Lyke, "Applications of neural networks in wireless communications," *IEEE Antennas and Propagation Magazine*, vol. 46, no. 3, pp. 130–137, June 2004.

[8] M. Ibnkahla, "Applications of neural networks to digital communications – a survey," *Signal Processing*, vol. 80, no. 7, pp. 1185 – 1215, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016516840000030X

[9] S. Chen, X. Hong, Y. Gong, and C. J. Harris, "Digital predistorter design using b-spline neural network and inverse of de boor algorithm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 6, pp. 1584–1594, June 2013.

[10] D. Mascharka and E. Manley, "Machine learning for indoor localization using mobile phone-based sensors," 05 2015.

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

[12] A. Hamalainen and J. Henriksson, "A recurrent neural decoder for convolutional codes," in *1999 IEEE International Conference on Communications (Cat. No. 99CH36311)*, vol. 2, June 1999, pp. 1305–1309 vol.2.

[13] N. Samuel, T. Diskin, and A. Wiesel, "Deep mimo detection," in *2017 IEEE 18th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, July 2017, pp. 1–5.

[14] T. J. O'Shea, K. Karra, and T. C. Clancy, "Learning to communicate: Channel auto-encoders, domain specific regularizers, and attention," in *Signal Processing and Information Technology (ISSPIT), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 223–228.

[15] J. Stebel, M. Krumnikl, P. Moravec, P. Olivka, and D. Seidl, "Neural network classification of sdr signal modulation," pp. 160–171, 09 2016.

[16] T. J. O'Shea, T. Roy, and N. West, "Approximating the void: Learning stochastic channel models from observation with variational generative adversarial networks," *CoRR*, vol. abs/1805.06350, 2018. [Online]. Available: http://arxiv.org/abs/1805.06350

[17] T. J. O'Shea, T. Roy, and T. C. Clancy, "Over-the-air deep learning based radio signal classification," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 168–179, Feb 2018.

[18] E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: closing the generalization gap in large batch training of neural networks," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 1731–1741. [Online]. Available: http://papers.nips.cc/paper/6770-train-longer-generalize-better-closing-the-generalization-gap-in-large-batch-training-of-neural-networks.pdf

[19] R. Novak, Y. Bahri, D. A. Abolafia, J. Pennington, and J. Sohl-Dickstein, "Sensitivity and generalization in neural networks: an empirical study," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=HJC2SzZCW

[20] Y. Sun, M. Peng, Y. Zhou, Y. Huang, and S. Mao, "Application of Machine Learning in Wireless Networks: Key Techniques and Open Issues," *ArXiv e-prints*, p. arXiv:1809.08707, Sep. 2018.

[21] N. Papernot, P. D. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," *CoRR*, vol. abs/1511.04508, 2015. [Online]. Available: http://arxiv.org/abs/1511.04508

[22] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against deep learning systems using adversarial examples," *CoRR*, vol. abs/1602.02697, 2016. [Online]. Available: http://arxiv.org/abs/1602.02697

[23] C. E. Shannon, "A mathematical theory of communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, no. 1, pp. 3–55, Jan. 2001. [Online]. Available: http://doi.acm.org/10.1145/584091.584093

[24] J. Gleick, "The information: A history, a theory, a flood (gleick, j.; 2011) [book review]," *IEEE Transactions on Information Theory*, vol. 57, no. 9, pp. 6332–6333, Sept 2011.

[25] Y. Singh, "Comparison of okumura, hata and cost-231 models on the basis of path loss and signal strength."

[26] A. Medeisis and A. Kajackas, "On the use of the universal okumura-hata propagation prediction model in rural areas," in *VTC2000-Spring. 2000 IEEE 51st Vehicular Technology Conference Proceedings (Cat. No.00CH37026)*, vol. 3, May 2000, pp. 1815–1818 vol.3.

[27] TIA, *Wireless Communications Systems Performance in Noise and Interference Limited Situations Part 2: Propagation and Noise*. Telecommuncations Industry Association, 2016, vol. 2, revision E.

[28] T. S. Rappaport *et al.*, *Wireless communications: principles and practice*. Prentice Hall PTR New Jersey, 1996, vol. 2.

[29] E. Labs, "Sbx without uhd corrections," in *Performance Data*. Ettus Research, October 2014.

[30] J. F. Blinn, "Quantization error and dithering," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 78–82, July 1994.

[31] J. H. D.L. Lucas, *A numerical representation of CCIR report 322 high frequency (3-30 MC/S) atmospheric radio noise data*. NBS Technical Note 318, 1965.

[32] R. Barry and W. Kinsner, "Multifractal characterization for classification of telecommunications traffic," in *IEEE CCECE2002. Canadian Conference on Electrical and Computer Engineering. Conference Proceedings (Cat. No.02CH37373)*, vol. 3, May 2002, pp. 1538–1544 vol.3.

[33] M. Stephan, "Antennas, filters and preamplifiers designed for the radio detection of ultra-high-energy cosmic rays," in *2010 Asia-Pacific Microwave Conference*, Dec 2010, pp. 1455–1458.

[34] D. Sahoo and A. Gupta, "Analysis of cmbr using nano-satellite," in *2018 IEEE Aerospace Conference*, March 2018, pp. 1–7.

[35] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in optimizing recurrent networks," *CoRR*, vol. abs/1212.0901, 2012. [Online]. Available: http://arxiv.org/abs/1212.0901

[36] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-24, Mar 2010. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-24.html

[37] T. Tieleman and G. Hinton, "RMSprop Gradient Optimization." [Online]. Available: http://www.cs.toronto.edu/~{}tijmen/csc321/slides/lecture_slides_lec6.pdf

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

[39] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[42] K. P. F.R.S., "On lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901. [Online]. Available: https://doi.org/10.1080/14786440109462720

[43] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *CoRR*, vol. abs/1803.04311, 2018. [Online]. Available: http://arxiv.org/abs/1803.04311

[44] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[46] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *CoRR*, vol. abs/1311.2901, 2013. [Online]. Available: http://arxiv.org/abs/1311.2901

[47] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[48] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[49] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. [Online]. Available: http://arxiv.org/abs/1512.00567

[50] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning," *CoRR*, vol. abs/1602.07261, 2016. [Online]. Available: http://arxiv.org/abs/1602.07261

[51] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[52] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[53] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[54] G. Cybenko, "Approximation by superpositions of a sigmoidal function," 1989.

[55] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, Jan 2016.

[56] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[57] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *CoRR*, vol. abs/1503.02531, 2015. [Online]. Available: http://arxiv.org/abs/1503.02531

[58] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *CoRR*, vol. abs/1710.07035, 2017. [Online]. Available: http://arxiv.org/abs/1710.07035

[59] A. M. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images." in *CVPR*. IEEE Computer Society, 2015, pp. 427–436. [Online]. Available: http://dblp.uni-trier.de/db/conf/cvpr/cvpr2015.html#NguyenYC15

[60] Z. Murez, S. Kolouri, D. J. Kriegman, R. Ramamoorthi, and K. Kim, "Image to image translation for domain adaptation," *CoRR*, vol. abs/1712.00479, 2017. [Online]. Available: http://arxiv.org/abs/1712.00479

[61] E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell, "Adversarial discriminative domain adaptation," *CoRR*, vol. abs/1702.05464, 2017.

[62] J. Hoffman, D. Wang, F. Yu, and T. Darrell, "Fcns in the wild: Pixel-level adversarial and constraint-based adaptation," *CoRR*, vol. abs/1612.02649, 2016.

[63] J. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," *CoRR*, vol. abs/1703.10593, 2017.

[64] E. Alberty, S. Defever, C. Moreau, R. D. Gaudenzi, A. Ginesi, R. Rinaldo, G. Gallinaro, and A. Vernucci, "Adaptive coding and modulation for the dvb-s2 standard interactive applications: Capacity assessment and key system issues," *IEEE Wireless Communications*, vol. 14, no. 4, pp. 61–69, August 2007.

[65] T. Mao, Q. Wang, Z. Wang, and S. Chen, "Novel index modulation techniques: A survey," *IEEE Communications Surveys Tutorials*, pp. 1–1, 2018.

[66] T. Wang, C.-K. Wen, H. Wang, F. Gao, T. Jiang, and S. Jin, "Deep learning for wireless physical layer: Opportunities and challenges," *China Communications*, vol. 14, no. 11, pp. 92–111, 2017.

[67] T. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer," *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, no. 4, pp. 563–575, Dec 2017.

[68] S. C. Hauser, W. C. Headley, and A. J. Michaels, "Signal detection effects on deep neural networks utilizing raw iq for modulation classification," in *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, Oct 2017, pp. 121–127.

[69] T. J. O'Shea, J. Corgan, and T. C. Clancy, "Convolutional radio modulation recognition networks," in *International Conference on Engineering Applications of Neural Networks*. Springer, 2016, pp. 213–226.

[70] S. J. Raudys and A. K. Jain, "Small sample size effects in statistical pattern recognition: recommendations for practitioners," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 3, pp. 252–264, Mar 1991.

[71] A. Krizhevsky, "Learning multiple layers of features from tiny images," 05 2012.

[72] H. Wymeersch, *Iterative receiver design*. Cambridge University Press Cambridge, 2007, vol. 234.

[73] S. G. Bilen, A. M. Wyglinski, C. R. Anderson, T. Cooklev, C. Dietrich, B. Farhang-Boroujeny, J. V. Urbina, S. H. Edwards, and J. H. Reed, "Software-defined radio: a new paradigm for integrated curriculum delivery," *IEEE Communications Magazine*, vol. 52, no. 5, pp. 184–193, May 2014.

[74] A. Torralba and A. A. Efros, "Unbiased look at dataset bias," in *CVPR 2011*, June 2011, pp. 1521–1528.

[75] N. Japkowicz and S. Stephen, "The class imbalance problem: A systematic study," *Intell. Data Anal.*, vol. 6, no. 5, pp. 429–449, Oct. 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1293951.1293954

[76] H. Shimodaira, "Improving predictive inference under covariate shift by weighting the log-likelihood function," *Journal of Statistical Planning and Inference*, vol. 90, no. 2, pp. 227–244, Oct. 2000. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0M-4136355-5/1/6432c256e0be03b1503bbf79e4e91d1a

[77] B. Zadrozny, "Learning and evaluating classifiers under sample selection bias," in *Proceedings of the Twenty-first International Conference on Machine Learning*, ser. ICML '04. New York, NY, USA: ACM, 2004, pp. 114–. [Online]. Available: http://doi.acm.org/10.1145/1015330.1015425

Appendix A

Channel Modeling MATLAB

A.1 Road channel.m

```
% AUTHOR: Kyle McClintick
% DATE: 9/17/18
% DESCRIPTION: Calculates the received power for a narrow-band signal
% as a function of distance given a 20x50x5 meter room, 0.7 wall and
% ceiling attenuation, 5.2GHz tone transmitted at 1mW, only considering
% first order reflections, and antenna height of 1.5m, a transmitter
% at (25,10), and a receiver at locations ranging from (1,5) to (49,5).

clc;
close all;
clear;

% seven first order paths (direct, walls, floor, ceiling)
r = 5000; % resolution, number of positions plotted

% 26.822 m/s is 60 MPH, to move 48 meters takes 1.79 seconds
t = linspace(0, 1.79, r);
phasors = zeros(r,5);
wb_phasors = zeros(r,5);
path_lengths = zeros(r,5);
PNB = zeros(1,r);
a = [1 0.38 0.34 0.8 0.8]; % mean dry asphalt attenuation coeff for floor, metal for front
    and back (cars), concrete for sides
```

```
c = 3*10^8; % speed of light
f = 2.42*10^9; % TPMS is 315 MHz (434MHz in Europe), bluetooth 2.42 GHz
xm = 50;
Pt = 0.00631; % -30dBm power transmitted for tpms, +8dBm for bluetooth
ym = 3.7; % width of a one lane road
xt = 25;
yt = 4.7; % listening station is a 1 meter off road
xr = sort(linspace(1,xm-1, r)+normrnd(0, 3, [1,r])); % position vector
yr = smoothdata(ym/2 + normrnd(0, 1, [1,r]),'gaussian',100); % car is somewhere near middle
    of lane
freq = 12.5; % avg tire has 12.5 RPS at 60 MPH
ht = 5; % listening station height
hr = 1.5*ones(1,r); % bluetooth height
% TPMS height changes as the tire rotates, wheel radius 19.05cm (where tpms
% is attached), whole wheel plus tire radius 34.42462cm
%hr = 0.3442462 + 0.1905*cos(2*pi*freq*t);
x1 = sort(linspace(1-2, xm-1-2, r)-normrnd(5, 3, [1, r])); % trailing car path
x2 = sort(linspace(1+2, xm-1+2, r)+normrnd(5, 3, [1,r])); % leading car path
y1 = smoothdata(ym/2 + normrnd(0, 1, [1,r]),'gaussian',100);
y2 = smoothdata(ym/2 + normrnd(0, 1, [1,r]),'gaussian',100);

for i=1:r
    % direct path
    path_lengths(i,1) = sqrt((xt-xr(i))^2+(yt-yr(i))^2);
    % bottom wall
    path_lengths(i,2) = sqrt((xt-xr(i))^2+(-yt-yr(i))^2);
    % top wall, ignored for RSU on top
    % path_lengths(i,3) = sqrt((xt-xr(i))^2+(-(ym-yt(i))-(ym-yr(i)))^2);
    % floor reflection
    path_lengths(i,3) = sqrt((xt-xr(i))^2+(yt-yr(i))^2+(-ht-hr(i))^2);
    if xt > x1(i)
        % path off first car behind
        path_lengths(i,4) = sqrt((-(xt-x1(i))-(xr(i)-x1(i)))^2+(yt-y1(i))^2);
    end
    if xt < x2(i)
        % path off first car ahead
        path_lengths(i,5) = sqrt((-(x2(i)-xt)-(x2(i)-xr(i)))^2+(yt-y2(i))^2);
    end
    % calculate phasors using path lengths
    phasors(i,:) = a * sqrt(Pt) .* exp(sqrt(-1)*2*pi*path_lengths(i,:)/(c/f)) ./ path_lengths
        (i,:);
```

```
    % wide_band phasors have no phase term. Used to evaluate effectiveness
    % of averaging narrow band phasors
    wb_phasors(i,:) = a * sqrt(Pt) ./ path_lengths(i,:);
    if xt < x1(i)
        % diffraction occurs on lagging car, modeled later
        phasors(i,4) = 0;
        wb_phasors(i,4) = 0;
    end
    if xt > x2(i)
        % diffraction occurs on leading car, modeled later
        phasors(i,5) = 0;
        wb_phasors(i,5) = 0;
    end

    % final received power
    PNB(i) = 20*log10(abs(sum(phasors(i,:))));
    wb_PNB(i) = 20*log10(abs(sum(wb_phasors(i,:))));
end

fixed_fading_low = smoothdata(PNB,'movmedian',30);
fixed_fading_high = smoothdata(PNB,'sgolay',500);
figure(1)
% subplot(1,2,1)
% plot(linspace(1,49,r), PNB, linspace(1,49,r), wb_PNB,'LineWidth',2);
plot(linspace(1,49,r), wb_PNB,'LineWidth',2);
grid on; title('Bluetooth Wide-Band Shadow Fading');
xlabel('Xcar (m) where Xrsu = 25');
ylabel('Pr (dB)');
% lgd = legend('Narrow Band Bluetooth', 'Bluetooth Power Profile with No Phase');
lgd.FontSize = 20;

% subplot(1,2,2)
% plot(linspace(1,49,r), fixed_fading_low, linspace(1,49,r), fixed_fading_high,'LineWidth',2)
% grid on;
% xlabel('Xcar (m) where Xrsu = 25','FontSize', 20);
% ylabel('SG PNB (dB)','FontSize', 20);
% lgd = legend('n=30 median','n=500 SG');
% lgd.FontSize = 20;


figure(2)
```

```
plot(linspace(1,49,r), PNB); grid on;
xlabel('Xcar (m) where Xrsu = 25');
ylabel('PNB (dB)','FontSize', 24);

% Doppler spectrum and spread
Tau = zeros(r,5); % travel time for each path for each receiver position
for i=1:r
    Tau(i,:) = path_lengths(i,:) / c;
end

% taps of middle location of center beacon node
figure(3)
stem(Tau(r/2,:), abs(phasors(r/2,:))); grid on;
title('Taps as Beacon Passes Sensor'); xlabel('Tau (seconds)');
ylabel('Amplitude');

% plot doppler spectrum
figure(4)
semilogy(linspace(-2*pi*c/f,2*pi*c/f,r), sum(abs(fft(abs(phasors))).^2,2)); grid on;
title('Doppler Spectrum'); xlabel('Frequency w (Hz)');
ylabel('D (dB)');

% calculate rms delay spread, tau_rms at taps passing by RSU
tau = abs(phasors(r/2,:)).*Tau(r/2,:) / abs(phasors(r/2,:));
tau2 = abs(phasors(r/2,:)).^2.*Tau(r/2,:) / abs(phasors(r/2,:));
tau_rms = sqrt(tau2 - tau^2)
tau_max = max(Tau(r/2,:))

% plot car swerving over time
figure(5)
plot(linspace(1,49, r), yr); ylim([0 3.7]);
grid on;
title('Ycar movement from 0m to 50m');
xlabel('Xcar (m)'); ylabel('Ycar (m)');


% lets get the power gradient, no more random variation in position
xr = linspace(xt, xm-1, r);
yr = ym/2;
x1 = linspace(xt-2-5, xm-1-2-5, r);
y1 = ym/2;
```

```
x2 = linspace(xt+2+5, xm-1-2+5, r);
y2 = ym/2;
for i=1:r
    % direct path
    path_lengths(i,1) = sqrt((xt-xr(i))^2+(yt-yr)^2);
    % bottom wall
    path_lengths(i,2) = sqrt((xt-xr(i))^2+(-yt-yr)^2);
    % top wall, ignored for RSU on top
    % path_lengths(i,3) = sqrt((xt-xr(i))^2+(-(ym-yt(i))-(ym-yr(i)))^2);
    % floor reflection
    path_lengths(i,3) = sqrt((xt-xr(i))^2+(yt-yr)^2+(-ht-hr(i))^2);
    if xt > x1(i)
        % path off first car behind
        path_lengths(i,4) = sqrt((-(xt-x1(i))-(xr(i)-x1(i)))^2+(yt-y1)^2);
    end
    if xt < x2(i)
        % path off first car ahead
        path_lengths(i,5) = sqrt((-(x2(i)-xt)-(x2(i)-xr(i)))^2+(yt-y2)^2);
    end
    % calculate phasors using path lengths
    phasors(i,:) = a * sqrt(Pt) .* exp(sqrt(-1)*2*pi*path_lengths(i,:)/(c/f)) ./ path_lengths
        (i,:);

    if xt < x1(i)
        % diffraction occurs on lagging car, modeled later
        phasors(i,4) = 0;
    end
    if xt > x2(i)
        % diffraction occurs on leading car, modeled later
        phasors(i,5) = 0;
    end

    % final received power
    PNB(i) = 20*log10(abs(sum(phasors(i,:))));
end

fit = polyfit(10*log10(linspace(1,xm-(xt+1), r)), PNB ,1);
gradient_narrowband = fit(1)

figure(6)
plot(linspace(1,xm-(xt+1),r), PNB); grid on; title('Total TPMS Prx Over Road Distance');
```

```
xlabel('distance (m)'); ylabel('PNB (dB)');
```

A.2 intermod.m

```
% AUTHOR: KWM

rng default
fi1 = 10e3; % sinusoid 1
fi2 = 11e3; % sinusoid 2
Fs = 48e3;
N = 1000; % num time domain samples
x = sin(2*pi*fi1/Fs*(1:N))+sin(2*pi*fi2/Fs*(1:N)); % sum sinusoids
% make sum non-linear by evaluating polynomial. Also add gaussian noise
% y = 0.0005x^3 + 0.0000001x^2 + 0.1x + 0.003
y = polyval([0.5e-3 1e-7 0.1 3e-3],x)+1e-5*randn(1,N);
% evaluate periodogram (PSD) using kaiser window, plot
w = kaiser(numel(y),38);

[Sxx, F] = periodogram(y,w,N,Fs,'psd');
[myTOI,Pfund,Ffund,Pim3,Fim3] = toi(Sxx,F,'psd')
toi(Sxx,F,'psd');
```

A.3 crosstalk.m

```
% AUTHOR: KWM
% simulates two neighboring gaussian pulses interfering with each other

% time domain
Fs = 100e6;
tc = gauspuls('cutoff',50e2,0.8,[],-40);
t = -tc : 1/Fs : tc;
y1 = gauspuls(t,200e3,0.6)+1e-5*randn(1,length(t));
y2 = gauspuls(t,100e3,0.6)+1e-5*randn(1,length(t));
```

```
% freq domain
Y = fft(y1);
L = length(t);
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = Fs*(0:(L/2))/L;

Y2 = fft(y2);
P2_2 = abs(Y2/L);
P1_2 = P2_2(1:L/2+1);
P1_2(2:end-1) = 2*P1_2(2:end-1);

% plotting
plot(f,P1,f,P1_2); xlim([0 300000]);
title('Single-Sided Amplitude Spectrum')
xlabel('f (Hz)');
ylabel('|P(f)|');
%Add vertical line
y1=get(gca,'ylim'); hold on; plot([75e3 75e3],y1,'r-','linewidth',2); hold off;
y2=get(gca,'ylim'); hold on; plot([125e3 125e3],y1,'r-','linewidth',2,'HandleVisibility','off
    '); hold off;
y3=get(gca,'ylim'); hold on; plot([175e3 175e3],y1,'b-','linewidth',2); hold off;
y4=get(gca,'ylim'); hold on; plot([225e3 225e3],y1,'b-','linewidth',2,'HandleVisibility','off
    '); hold off;
legend('200 kHz Gaussian Pulse','100 kHz Gaussian Pulse');
```

A.4 industrialnoise.m

```
% Author: Kyle McClintick

white = dsp.ColoredNoise('Color','white');
brown = dsp.ColoredNoise('Color','brown');
pink = dsp.ColoredNoise('Color','pink');
black = dsp.ColoredNoise('InverseFrequencyPower',2);

out_w = white();
```

```
out_brwn = brown();
out_p = pink();
out_blk = smoothdata(black());

out_w1 = white();
out_brwn1 = brown();
out_p1 = pink();
out_blk1 = smoothdata(black());

class_4 = [out_w; out_brwn; out_blk; out_p];
class_7 = [out_brwn1; out_w1; out_p1; out_blk1];

figure(1)
plot(class_4); title('Class 4 Industrial Noise');
xlabel('Discrete Time (n)'); ylabel('Amplitdue');

figure(2)
plot(class_7); title('Class 7 Industrial Noise');
xlabel('Discrete Time (n)'); ylabel('Amplitdue');
```

A.5 iqoffset.m

```
% AUTHOR: kyle mcclintick

% Gen AWGN QPSK data
I = sqrt(2)/2 * (2*randi(2,1000,1) - 3) + 1e-1*randn(1000,1);
Q = sqrt(2)/2 * (2*randi(2,1000,1) - 3) + 1e-1*randn(1000,1);


% apply IQ offest
kI = 1;
kQ = 1;
phi = 20*pi/180;

I_o = kI*I;
Q_o = kQ*sin(phi)*Q - kQ*sin(phi)*I;

scatter(I,Q); hold on;
```

```
scatter(I_o,Q_o); title('QPSK Waveform');
xlabel('In-phase'); ylabel('Quadriture');
legend('No IQ Offset','IQ Offset'); xlim([-1.5 1.5]); ylim([-1.5 1.5]);
hold off;
```

A.6 pcm.m

```
% AUTHOR: Alex Wyglinski, Kyle McClintick and Gigi Dong
% DATE: 10/3/18
close all;
clear;
clc;

% Define constants
L = 100; % Length of the overall transmission
M = 10; % Pulse duration for rectangular pulse train
N = 10; % Upsampling factor for generating analog waveform

% Generate a discrete version of a random continuous analog
% waveform using a Uniform Random Number Generator and
% an interpolation function to smooth out the result
analog_wavefm = interp((2*rand(1,(L/M))-1),M);
% Generate a rectangular pulse train of samples
impulsetrain_wavefm = reshape(ones(N,1)*rem(1:1:(L/N),2),[1,L]);

% Plotting Base signals
% figure(1)
% subplot(2,1,1)
% plot(analog_wavefm); title('Random Analog Waveform')
% ylim([-2 2]); ylabel('Signal Amplitude'); xlabel('Discrete Time (n)');
% subplot(2,1,2)
% stem(impulsetrain_wavefm); ylabel('Signal Amplitude');
% xlabel('Discrete Time (n)'); title('Impulse Train');

% PAM
natural_PAM = analog_wavefm .* impulsetrain_wavefm;
flattop_PAM = repelem(natural_PAM(1:N:end), N);
```

```
% plotting PAM
% figure(2)
% subplot(2,1,1)
% stem(natural_PAM); title('Naturally Sampled PAM');
% ylim([-2 2]); ylabel('Signal Amplitude'); xlabel('Discrete Time (n)');
% subplot(2,1,2)
% stem(flattop_PAM); ylabel('Signal Amplitude');
% xlabel('Discrete Time (n)'); title('Flat-Top PAM');

% Quantizing
q = zeros(1,L);
[ind,quantv] = quantiz(downsample(analog_wavefm,N), ...
    [-0.8 -0.6 -0.4 -0.2 0 0.2 0.4 0.6 0.8], ...
    [-0.9 -0.7 -0.5 -0.3 -0.1 0.1 0.3 0.5 0.7 0.9]);
for i=1:N-1
    [d, ix] = min(abs(quantv -   flattop_PAM(1+((i-1)*N))  ));
    q(1+((i-1)*N):i*N) = quantv(ix);
end
q = q .* impulsetrain_wavefm;
residual = flattop_PAM - q;

%
figure(3)
subplot(5,1,1)
plot(analog_wavefm); title('Random Analog Waveform'); xlabel('Time');
ylabel('Signal Amplitude');
subplot(5,1,2)
stem(natural_PAM); title('Natural PAM'); xlabel('Discrete Time (n)');
ylabel('Signal Amplitude');
subplot(5,1,3)
stem(flattop_PAM); title('Flat-Top PAM'); xlabel('Discrete Time (n)');
ylabel('Signal Amplitude');
subplot(5,1,4)
stem(q); title('PCM'); xlabel('Discrete Time (n)');
ylabel('Signal Amplitude');
subplot(5,1,5)
stem(residual); ylabel('Residual Amplitdue'); xlabel('Discrete Time (n)');
title('Quantization Error');
```

A.7 universal approximator.m

```
% AUTHOR: KYLE MCCLINTICK
% DATE: Oct 31st 2018

% simple two neuron case showcasing step ability
x = linspace(0,1);
w1 = 999;
b1 = -400;
b2 = -650;

sigmoid1 = 1./(1+exp(-(w1*x+b1)));
sigmoid2 = 1./(1+exp(-(w1*x+b2)));

figure(1)
plot(x,sigmoid1+sigmoid2); title('Two Neuron Hidden Layer Output');
xlabel('x'); ylabel('Sum of Dendrites at next layers neuron');

% try to approximate an actual analog function
L = 100; % Length of the overall transmission
M = 10; % Pulse duration for rectangular pulse train

% Generate a discrete version of a random continuous analog
% waveform using a Uniform Random Number Generator and
% an interpolation function to smooth out the result
analog_wavefm = interp((2*rand(1,(L/M))-1),M);

approx = zeros(1,L);
num_neurons = 10;
for i=1:num_neurons
    b = -10 * (num_neurons*(i-1) + 1);
    w = 999;
    weight = analog_wavefm((i-1)*num_neurons + 1);
    approx = approx + weight * 1./(1+exp(-(w*x+b))); % add sigmoid for this segment
    approx = approx - weight * 1./(1+exp(-(w*x+(b-num_neurons*10))));% remove it from future
        segments
end

figure(2)
```

```
plot(x, analog_wavefm,x,approx); title('Approximation of Waveform Using 10 Neurons');
xlabel('x'); ylabel('Sum of Dendrites at next layers neuron');
```

Appendix B

High-Performance Computing Cluster Bash

B.1 bash job submission.sh

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 2
#SBATCH -t 10:00:00
#SBATCH -o commdrcn_kwm.out
#SBATCH --gres=gpu:2

module load cuda90/toolkit/9.0.176
module load cudnn/7.1

source /home/kwmcclintick/tf_keras/bin/activate
python main_sm.py
```

B.2 1convnet.out

```
Using TensorFlow backend.
```

```
2018-12-03 16:58:48.659439: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found
    device 0 with properties:
name: Tesla K20Xm major: 3 minor: 5 memoryClockRate(GHz): 0.732
pciBusID: 0000:03:00.0
totalMemory: 5.57GiB freeMemory: 5.49GiB
2018-12-03 16:58:48.828053: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found
    device 1 with properties:
name: Tesla K20Xm major: 3 minor: 5 memoryClockRate(GHz): 0.732
pciBusID: 0000:83:00.0
totalMemory: 5.57GiB freeMemory: 5.49GiB
2018-12-03 16:58:48.828195: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1511] Adding
    visible gpu devices: 0, 1
2018-12-03 16:58:49.682739: I tensorflow/core/common_runtime/gpu/gpu_device.cc:982] Device
    interconnect StreamExecutor with strength 1 edge matrix:
2018-12-03 16:58:49.682842: I tensorflow/core/common_runtime/gpu/gpu_device.cc:988]  0 1
2018-12-03 16:58:49.682855: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 0:   N N
2018-12-03 16:58:49.682862: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 1:   N N
2018-12-03 16:58:49.683293: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created
    TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 5279 MB memory) ->
    physical GPU (device: 0, name: Tesla K20Xm, pci bus id: 0000:03:00.0, compute capability:
     3.5)
2018-12-03 16:58:49.683991: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created
    TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:1 with 5279 MB memory) ->
    physical GPU (device: 1, name: Tesla K20Xm, pci bus id: 0000:83:00.0, compute capability:
     3.5)
source dataset shape:
(220000, 2, 128)
Source domain partition summary:
X_train: (132000, 2, 128, 1)
Y_train: (132000, 11)
X_val: (44000, 2, 128, 1)
Y_val: (44000, 11)
X_test: (44000, 2, 128, 1)
Y_test: (44000, 11)
target dataset shape:
(220000, 2, 128)
Target domain partition summary:
X_tgt_train: (132000, 2, 128, 1)
Y_tgt_train: (132000, 11)
X_tgt_val: (44000, 2, 128, 1)
Y_tgt_val: (44000, 11)
```

```
X_tgt_test: (44000, 2, 128, 1)
Y_tgt_test: (44000, 11)
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2, 128, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 2, 128, 100)       1000
_________________________________________________________________
activation_1 (Activation)    (None, 2, 128, 100)       0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 1, 128, 100)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 1, 128, 150)       135150
_________________________________________________________________
activation_2 (Activation)    (None, 1, 128, 150)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 128, 150)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 128, 200)       270200
_________________________________________________________________
activation_3 (Activation)    (None, 1, 128, 200)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25600)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              26215424
_________________________________________________________________
activation_4 (Activation)    (None, 1024)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              1049600
_________________________________________________________________
activation_5 (Activation)    (None, 1024)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_3 (Dense)              (None, 11)                11275
=================================================================
Total params: 27,682,649
Trainable params: 27,682,649
```

```
Non-trainable params: 0
----------------------------------------------------------------
None
Train Convnet source/source
Epoch-1: (loss: 2.271, acc: 0.202), (val_loss: 2.131, val_acc: 0.202), (test_Loss: 2.135,
    test_acc: 0.196) -- 65.37 sec
Epoch-2: (loss: 2.050, acc: 0.297), (val_loss: 1.901, val_acc: 0.296), (test_Loss: 1.900,
    test_acc: 0.291) -- 52.84 sec
Epoch-3: (loss: 1.839, acc: 0.371), (val_loss: 1.719, val_acc: 0.367), (test_Loss: 1.716,
    test_acc: 0.365) -- 52.95 sec
Epoch-4: (loss: 1.746, acc: 0.393), (val_loss: 1.657, val_acc: 0.387), (test_Loss: 1.653,
    test_acc: 0.389) -- 52.92 sec
Epoch-5: (loss: 1.666, acc: 0.408), (val_loss: 1.591, val_acc: 0.404), (test_Loss: 1.588,
    test_acc: 0.405) -- 52.84 sec
Epoch-6: (loss: 1.595, acc: 0.425), (val_loss: 1.544, val_acc: 0.424), (test_Loss: 1.543,
    test_acc: 0.423) -- 52.81 sec
Epoch-7: (loss: 1.526, acc: 0.456), (val_loss: 1.460, val_acc: 0.452), (test_Loss: 1.458,
    test_acc: 0.446) -- 52.80 sec
Epoch-8: (loss: 1.483, acc: 0.465), (val_loss: 1.423, val_acc: 0.457), (test_Loss: 1.419,
    test_acc: 0.455) -- 52.85 sec
Epoch-9: (loss: 1.448, acc: 0.474), (val_loss: 1.405, val_acc: 0.468), (test_Loss: 1.401,
    test_acc: 0.464) -- 52.96 sec
Epoch-10: (loss: 1.420, acc: 0.483), (val_loss: 1.379, val_acc: 0.478), (test_Loss: 1.377,
    test_acc: 0.471) -- 52.86 sec
Epoch-11: (loss: 1.404, acc: 0.498), (val_loss: 1.360, val_acc: 0.492), (test_Loss: 1.357,
    test_acc: 0.485) -- 52.97 sec
Epoch-12: (loss: 1.385, acc: 0.502), (val_loss: 1.351, val_acc: 0.493), (test_Loss: 1.348,
    test_acc: 0.488) -- 52.91 sec
Epoch-13: (loss: 1.368, acc: 0.497), (val_loss: 1.365, val_acc: 0.489), (test_Loss: 1.362,
    test_acc: 0.484) -- 52.86 sec
Epoch-14: (loss: 1.354, acc: 0.511), (val_loss: 1.330, val_acc: 0.500), (test_Loss: 1.326,
    test_acc: 0.495) -- 53.04 sec
Epoch-15: (loss: 1.346, acc: 0.514), (val_loss: 1.322, val_acc: 0.502), (test_Loss: 1.317,
    test_acc: 0.497) -- 52.82 sec
Epoch-16: (loss: 1.334, acc: 0.524), (val_loss: 1.310, val_acc: 0.512), (test_Loss: 1.306,
    test_acc: 0.508) -- 52.86 sec
Epoch-17: (loss: 1.327, acc: 0.520), (val_loss: 1.319, val_acc: 0.507), (test_Loss: 1.315,
    test_acc: 0.502) -- 52.82 sec
Epoch-18: (loss: 1.317, acc: 0.530), (val_loss: 1.299, val_acc: 0.514), (test_Loss: 1.295,
    test_acc: 0.509) -- 52.85 sec
```

```
Epoch-19: (loss: 1.306, acc: 0.530), (val_loss: 1.301, val_acc: 0.513), (test_Loss: 1.298,
    test_acc: 0.506) -- 52.88 sec
Epoch-20: (loss: 1.302, acc: 0.532), (val_loss: 1.301, val_acc: 0.514), (test_Loss: 1.297,
    test_acc: 0.507) -- 52.80 sec
Epoch-21: (loss: 1.298, acc: 0.537), (val_loss: 1.293, val_acc: 0.515), (test_Loss: 1.289,
    test_acc: 0.511) -- 52.83 sec
Epoch-22: (loss: 1.297, acc: 0.534), (val_loss: 1.305, val_acc: 0.513), (test_Loss: 1.300,
    test_acc: 0.510) -- 52.91 sec
Epoch-23: (loss: 1.294, acc: 0.542), (val_loss: 1.291, val_acc: 0.517), (test_Loss: 1.288,
    test_acc: 0.514) -- 52.81 sec
Epoch-24: (loss: 1.281, acc: 0.538), (val_loss: 1.299, val_acc: 0.515), (test_Loss: 1.296,
    test_acc: 0.511) -- 52.82 sec
Epoch-25: (loss: 1.277, acc: 0.542), (val_loss: 1.289, val_acc: 0.519), (test_Loss: 1.287,
    test_acc: 0.515) -- 52.83 sec
Epoch-26: (loss: 1.278, acc: 0.540), (val_loss: 1.308, val_acc: 0.513), (test_Loss: 1.304,
    test_acc: 0.509) -- 52.94 sec
Epoch-27: (loss: 1.268, acc: 0.547), (val_loss: 1.288, val_acc: 0.517), (test_Loss: 1.285,
    test_acc: 0.515) -- 52.87 sec
Epoch-28: (loss: 1.263, acc: 0.548), (val_loss: 1.295, val_acc: 0.517), (test_Loss: 1.289,
    test_acc: 0.514) -- 52.79 sec
Epoch-29: (loss: 1.261, acc: 0.554), (val_loss: 1.286, val_acc: 0.520), (test_Loss: 1.282,
    test_acc: 0.517) -- 52.89 sec
Epoch-30: (loss: 1.252, acc: 0.557), (val_loss: 1.286, val_acc: 0.519), (test_Loss: 1.282,
    test_acc: 0.515) -- 52.84 sec
```

B.3 2conv.out

```
Using TensorFlow backend.
2018-12-03 13:39:16.727641: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found
    device 0 with properties:
name: Tesla K20Xm major: 3 minor: 5 memoryClockRate(GHz): 0.732
pciBusID: 0000:03:00.0
totalMemory: 5.57GiB freeMemory: 5.49GiB
2018-12-03 13:39:16.860210: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found
    device 1 with properties:
name: Tesla K20Xm major: 3 minor: 5 memoryClockRate(GHz): 0.732
pciBusID: 0000:83:00.0
totalMemory: 5.57GiB freeMemory: 5.49GiB
```

```
2018-12-03 13:39:16.860296: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1511] Adding
    visible gpu devices: 0, 1
2018-12-03 13:39:17.420655: I tensorflow/core/common_runtime/gpu/gpu_device.cc:982] Device
    interconnect StreamExecutor with strength 1 edge matrix:
2018-12-03 13:39:17.420727: I tensorflow/core/common_runtime/gpu/gpu_device.cc:988]  0 1
2018-12-03 13:39:17.420738: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 0:   N N
2018-12-03 13:39:17.420744: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 1:   N N
2018-12-03 13:39:17.421083: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created
    TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 5279 MB memory) ->
    physical GPU (device: 0, name: Tesla K20Xm, pci bus id: 0000:03:00.0, compute capability:
     3.5)
2018-12-03 13:39:17.421456: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created
    TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:1 with 5279 MB memory) ->
    physical GPU (device: 1, name: Tesla K20Xm, pci bus id: 0000:83:00.0, compute capability:
     3.5)
source dataset shape:
(220000, 2, 128)
Source domain partition summary:
X_train: (132000, 2, 128, 1)
Y_train: (132000, 11)
X_val: (44000, 2, 128, 1)
Y_val: (44000, 11)
X_test: (44000, 2, 128, 1)
Y_test: (44000, 11)
target dataset shape:
(220000, 2, 128)
Target domain partition summary:
X_tgt_train: (132000, 2, 128, 1)
Y_tgt_train: (132000, 11)
X_tgt_val: (44000, 2, 128, 1)
Y_tgt_val: (44000, 11)
X_tgt_test: (44000, 2, 128, 1)
Y_tgt_test: (44000, 11)
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2, 128, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 2, 128, 100)       1000
_________________________________________________________________
activation_1 (Activation)    (None, 2, 128, 100)       0
```

```
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 1, 128, 100)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 1, 128, 150)       135150
_________________________________________________________________
activation_2 (Activation)    (None, 1, 128, 150)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 128, 150)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 128, 200)       270200
_________________________________________________________________
activation_3 (Activation)    (None, 1, 128, 200)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25600)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              26215424
_________________________________________________________________
activation_4 (Activation)    (None, 1024)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              1049600
_________________________________________________________________
activation_5 (Activation)    (None, 1024)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_3 (Dense)              (None, 11)                11275
=================================================================
Total params: 27,682,649
Trainable params: 27,682,649
Non-trainable params: 0
_________________________________________________________________
None
Train Convnet target/target
Epoch-1: (loss: 2.398, acc: 0.119), (val_loss: 2.388, val_acc: 0.120), (test_Loss: 2.389,
    test_acc: 0.118) -- 63.67 sec
Epoch-2: (loss: 2.351, acc: 0.160), (val_loss: 2.310, val_acc: 0.161), (test_Loss: 2.311,
    test_acc: 0.157) -- 52.43 sec
Epoch-3: (loss: 2.258, acc: 0.208), (val_loss: 2.174, val_acc: 0.207), (test_Loss: 2.178,
    test_acc: 0.207) -- 52.48 sec
```

```
Epoch-4: (loss: 2.095, acc: 0.262), (val_loss: 1.941, val_acc: 0.260), (test_Loss: 1.939,
    test_acc: 0.256) -- 52.42 sec
Epoch-5: (loss: 1.953, acc: 0.274), (val_loss: 1.891, val_acc: 0.275), (test_Loss: 1.887,
    test_acc: 0.274) -- 52.43 sec
Epoch-6: (loss: 1.915, acc: 0.287), (val_loss: 1.875, val_acc: 0.290), (test_Loss: 1.872,
    test_acc: 0.286) -- 52.44 sec
Epoch-7: (loss: 1.904, acc: 0.299), (val_loss: 1.852, val_acc: 0.298), (test_Loss: 1.848,
    test_acc: 0.297) -- 52.43 sec
Epoch-8: (loss: 1.892, acc: 0.291), (val_loss: 1.854, val_acc: 0.288), (test_Loss: 1.851,
    test_acc: 0.291) -- 52.41 sec
Epoch-9: (loss: 1.875, acc: 0.301), (val_loss: 1.849, val_acc: 0.300), (test_Loss: 1.846,
    test_acc: 0.299) -- 52.41 sec
Epoch-10: (loss: 1.891, acc: 0.295), (val_loss: 1.862, val_acc: 0.296), (test_Loss: 1.860,
    test_acc: 0.295) -- 52.40 sec
Epoch-11: (loss: 1.860, acc: 0.306), (val_loss: 1.832, val_acc: 0.306), (test_Loss: 1.830,
    test_acc: 0.306) -- 52.40 sec
Epoch-12: (loss: 1.851, acc: 0.296), (val_loss: 1.836, val_acc: 0.296), (test_Loss: 1.834,
    test_acc: 0.294) -- 52.42 sec
Epoch-13: (loss: 1.834, acc: 0.319), (val_loss: 1.801, val_acc: 0.317), (test_Loss: 1.800,
    test_acc: 0.317) -- 52.40 sec
Epoch-14: (loss: 1.831, acc: 0.322), (val_loss: 1.804, val_acc: 0.323), (test_Loss: 1.802,
    test_acc: 0.322) -- 52.39 sec
Epoch-15: (loss: 1.819, acc: 0.333), (val_loss: 1.780, val_acc: 0.330), (test_Loss: 1.779,
    test_acc: 0.329) -- 52.40 sec
Epoch-16: (loss: 1.796, acc: 0.296), (val_loss: 1.837, val_acc: 0.295), (test_Loss: 1.834,
    test_acc: 0.293) -- 52.41 sec
Epoch-17: (loss: 1.788, acc: 0.330), (val_loss: 1.771, val_acc: 0.329), (test_Loss: 1.771,
    test_acc: 0.327) -- 52.40 sec
Epoch-18: (loss: 1.772, acc: 0.347), (val_loss: 1.733, val_acc: 0.346), (test_Loss: 1.732,
    test_acc: 0.344) -- 52.40 sec
Epoch-19: (loss: 1.769, acc: 0.335), (val_loss: 1.748, val_acc: 0.331), (test_Loss: 1.747,
    test_acc: 0.329) -- 52.42 sec
Epoch-20: (loss: 1.757, acc: 0.338), (val_loss: 1.744, val_acc: 0.333), (test_Loss: 1.742,
    test_acc: 0.332) -- 52.42 sec
Epoch-21: (loss: 1.748, acc: 0.352), (val_loss: 1.714, val_acc: 0.349), (test_Loss: 1.713,
    test_acc: 0.348) -- 52.40 sec
Epoch-22: (loss: 1.745, acc: 0.347), (val_loss: 1.721, val_acc: 0.344), (test_Loss: 1.719,
    test_acc: 0.344) -- 52.41 sec
Epoch-23: (loss: 1.744, acc: 0.355), (val_loss: 1.710, val_acc: 0.351), (test_Loss: 1.708,
    test_acc: 0.350) -- 52.39 sec
```

```
Epoch-24: (loss: 1.729, acc: 0.338), (val_loss: 1.732, val_acc: 0.335), (test_Loss: 1.729,
    test_acc: 0.331) -- 52.40 sec
Epoch-25: (loss: 1.725, acc: 0.353), (val_loss: 1.701, val_acc: 0.347), (test_Loss: 1.700,
    test_acc: 0.347) -- 52.34 sec
Epoch-26: (loss: 1.729, acc: 0.333), (val_loss: 1.750, val_acc: 0.331), (test_Loss: 1.747,
    test_acc: 0.330) -- 52.35 sec
Epoch-27: (loss: 1.720, acc: 0.353), (val_loss: 1.708, val_acc: 0.349), (test_Loss: 1.705,
    test_acc: 0.348) -- 52.46 sec
Epoch-28: (loss: 1.709, acc: 0.351), (val_loss: 1.706, val_acc: 0.346), (test_Loss: 1.704,
    test_acc: 0.347) -- 52.35 sec
Epoch-29: (loss: 1.707, acc: 0.354), (val_loss: 1.700, val_acc: 0.349), (test_Loss: 1.699,
    test_acc: 0.346) -- 52.43 sec
Epoch-30: (loss: 1.707, acc: 0.360), (val_loss: 1.697, val_acc: 0.354), (test_Loss: 1.696,
    test_acc: 0.351) -- 52.41 sec
```

B.4 3commdrcn.out

```
Using TensorFlow backend.
2018-12-03 13:38:38.863491: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found
    device 0 with properties:
name: Tesla K20Xm major: 3 minor: 5 memoryClockRate(GHz): 0.732
pciBusID: 0000:03:00.0
totalMemory: 5.57GiB freeMemory: 5.49GiB
2018-12-03 13:38:38.991373: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found
    device 1 with properties:
name: Tesla K20Xm major: 3 minor: 5 memoryClockRate(GHz): 0.732
pciBusID: 0000:83:00.0
totalMemory: 5.57GiB freeMemory: 5.49GiB
2018-12-03 13:38:38.991469: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1511] Adding
    visible gpu devices: 0, 1
2018-12-03 13:38:39.615807: I tensorflow/core/common_runtime/gpu/gpu_device.cc:982] Device
    interconnect StreamExecutor with strength 1 edge matrix:
2018-12-03 13:38:39.615958: I tensorflow/core/common_runtime/gpu/gpu_device.cc:988]  0 1
2018-12-03 13:38:39.615974: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 0:   N N
2018-12-03 13:38:39.615981: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 1:   N N
2018-12-03 13:38:39.616589: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created
    TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 5279 MB memory) ->
    physical GPU (device: 0, name: Tesla K20Xm, pci bus id: 0000:03:00.0, compute capability:
```

```
     3.5)
2018-12-03 13:38:39.617574: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created
    TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:1 with 5279 MB memory) ->
    physical GPU (device: 1, name: Tesla K20Xm, pci bus id: 0000:83:00.0, compute capability:
     3.5)
2018-12-03 13:38:45.593172: W tensorflow/core/common_runtime/bfc_allocator.cc:211] Allocator
    (GPU_0_bfc) ran out of memory trying to allocate 3.48GiB. The caller indicates that this
    is not a failure, but may mean that there could be performance gains if more memory were
    available.
2018-12-03 13:38:46.356328: W tensorflow/core/common_runtime/bfc_allocator.cc:211] Allocator
    (GPU_0_bfc) ran out of memory trying to allocate 3.03GiB. The caller indicates that this
    is not a failure, but may mean that there could be performance gains if more memory were
    available.
2018-12-03 13:38:49.108255: W tensorflow/core/common_runtime/bfc_allocator.cc:211] Allocator
    (GPU_0_bfc) ran out of memory trying to allocate 3.03GiB. The caller indicates that this
    is not a failure, but may mean that there could be performance gains if more memory were
    available.
2018-12-03 13:38:50.368324: W tensorflow/core/common_runtime/bfc_allocator.cc:211] Allocator
    (GPU_0_bfc) ran out of memory trying to allocate 3.48GiB. The caller indicates that this
    is not a failure, but may mean that there could be performance gains if more memory were
    available.
source dataset shape:
(220000, 2, 128)
Source domain partition summary:
X_train: (132000, 2, 128, 1)
Y_train: (132000, 11)
X_val: (44000, 2, 128, 1)
Y_val: (44000, 11)
X_test: (44000, 2, 128, 1)
Y_test: (44000, 11)
target dataset shape:
(220000, 2, 128)
Target domain partition summary:
X_tgt_train: (132000, 2, 128, 1)
Y_tgt_train: (132000, 11)
X_tgt_val: (44000, 2, 128, 1)
Y_tgt_val: (44000, 11)
X_tgt_test: (44000, 2, 128, 1)
Y_tgt_test: (44000, 11)
Create DRCN Model
----------------------------------------------------------------
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2, 128, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 2, 128, 100)       1000
_________________________________________________________________
activation_1 (Activation)    (None, 2, 128, 100)       0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 1, 128, 100)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 1, 128, 150)       135150
_________________________________________________________________
activation_2 (Activation)    (None, 1, 128, 150)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 128, 150)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 128, 200)       270200
_________________________________________________________________
activation_3 (Activation)    (None, 1, 128, 200)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25600)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              26215424
_________________________________________________________________
activation_4 (Activation)    (None, 1024)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              1049600
_________________________________________________________________
activation_5 (Activation)    (None, 1024)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_3 (Dense)              (None, 11)                11275
=================================================================
Total params: 27,682,649
Trainable params: 27,682,649
Non-trainable params: 0
_________________________________________________________________
None
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2, 128, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 2, 128, 100)       1000
_________________________________________________________________
activation_6 (Activation)    (None, 2, 128, 100)       0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 1, 128, 100)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 1, 128, 150)       135150
_________________________________________________________________
activation_7 (Activation)    (None, 1, 128, 150)       0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 1, 128, 150)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 128, 200)       270200
_________________________________________________________________
activation_8 (Activation)    (None, 1, 128, 200)       0
_________________________________________________________________
flatten_2 (Flatten)          (None, 25600)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              26215424
_________________________________________________________________
activation_9 (Activation)    (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              1049600
_________________________________________________________________
activation_10 (Activation)   (None, 1024)              0
_________________________________________________________________
dense_4 (Dense)              (None, 1024)              1049600
_________________________________________________________________
activation_11 (Activation)   (None, 1024)              0
_________________________________________________________________
dense_5 (Dense)              (None, 25600)             26240000
_________________________________________________________________
activation_12 (Activation)   (None, 25600)             0
_________________________________________________________________
reshape_1 (Reshape)          (None, 1, 128, 200)       0
_________________________________________________________________
```

```
conv2d_4 (Conv2D)            (None, 1, 128, 200)       360200
_________________________________________________________________
activation_13 (Activation)   (None, 1, 128, 200)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 128, 150)       270150
_________________________________________________________________
activation_14 (Activation)   (None, 1, 128, 150)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 1, 128, 100)       135100
_________________________________________________________________
activation_15 (Activation)   (None, 1, 128, 100)       0
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 2, 128, 100)       0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 2, 128, 1)         901
=================================================================
Total params: 55,727,325
Trainable params: 55,727,325
Non-trainable params: 0
_________________________________________________________________
None
Epoch-1: (loss: 2.272, acc: 0.188, gen_loss: 0.000), (val_loss: 2.135, val_acc: 0.188), (
    test_Loss: 2.333, test_acc: 0.139) -- 145.27 sec
Epoch-2: (loss: 2.058, acc: 0.285, gen_loss: 0.000), (val_loss: 1.910, val_acc: 0.283), (
    test_Loss: 2.105, test_acc: 0.223) -- 128.05 sec
Epoch-3: (loss: 1.854, acc: 0.366, gen_loss: 0.000), (val_loss: 1.730, val_acc: 0.361), (
    test_Loss: 1.942, test_acc: 0.279) -- 128.14 sec
Epoch-4: (loss: 2.259, acc: 0.315, gen_loss: 0.000), (val_loss: 1.847, val_acc: 0.316), (
    test_Loss: 2.060, test_acc: 0.244) -- 127.94 sec
Epoch-5: (loss: 1.865, acc: 0.329, gen_loss: 0.000), (val_loss: 1.802, val_acc: 0.329), (
    test_Loss: 2.006, test_acc: 0.256) -- 128.03 sec
Epoch-6: (loss: 1.829, acc: 0.335, gen_loss: 0.000), (val_loss: 1.780, val_acc: 0.333), (
    test_Loss: 1.980, test_acc: 0.261) -- 127.95 sec
Epoch-7: (loss: 1.800, acc: 0.357, gen_loss: 0.000), (val_loss: 1.746, val_acc: 0.355), (
    test_Loss: 1.930, test_acc: 0.280) -- 126.86 sec
Epoch-8: (loss: 1.765, acc: 0.374, gen_loss: 0.000), (val_loss: 1.698, val_acc: 0.372), (
    test_Loss: 1.899, test_acc: 0.291) -- 126.97 sec
Epoch-9: (loss: 1.730, acc: 0.356, gen_loss: 0.000), (val_loss: 1.751, val_acc: 0.353), (
    test_Loss: 1.913, test_acc: 0.297) -- 127.07 sec
Epoch-10: (loss: 1.674, acc: 0.408, gen_loss: 0.000), (val_loss: 1.587, val_acc: 0.405), (
    test_Loss: 1.792, test_acc: 0.335) -- 127.17 sec
```

```
Epoch-11: (loss: 1.603, acc: 0.427, gen_loss: 0.000), (val_loss: 1.533, val_acc: 0.426), (
    test_Loss: 1.743, test_acc: 0.346) -- 126.82 sec
Epoch-12: (loss: 1.558, acc: 0.440, gen_loss: 0.000), (val_loss: 1.491, val_acc: 0.434), (
    test_Loss: 1.706, test_acc: 0.357) -- 126.98 sec
Epoch-13: (loss: 1.565, acc: 0.452, gen_loss: 0.000), (val_loss: 1.463, val_acc: 0.444), (
    test_Loss: 1.687, test_acc: 0.363) -- 127.03 sec
Epoch-14: (loss: 1.598, acc: 0.455, gen_loss: 0.000), (val_loss: 1.445, val_acc: 0.450), (
    test_Loss: 1.665, test_acc: 0.370) -- 127.05 sec
Epoch-15: (loss: 1.781, acc: 0.440, gen_loss: 0.000), (val_loss: 1.495, val_acc: 0.433), (
    test_Loss: 1.716, test_acc: 0.351) -- 127.07 sec
Epoch-16: (loss: 1.569, acc: 0.459, gen_loss: 0.000), (val_loss: 1.443, val_acc: 0.452), (
    test_Loss: 1.658, test_acc: 0.374) -- 126.88 sec
Epoch-17: (loss: 1.490, acc: 0.466, gen_loss: 0.000), (val_loss: 1.419, val_acc: 0.460), (
    test_Loss: 1.637, test_acc: 0.379) -- 127.10 sec
Epoch-18: (loss: 1.459, acc: 0.467, gen_loss: 0.000), (val_loss: 1.405, val_acc: 0.459), (
    test_Loss: 1.629, test_acc: 0.380) -- 126.91 sec
Epoch-19: (loss: 1.444, acc: 0.476, gen_loss: 0.000), (val_loss: 1.396, val_acc: 0.466), (
    test_Loss: 1.617, test_acc: 0.384) -- 126.92 sec
Epoch-20: (loss: 1.427, acc: 0.477, gen_loss: 0.000), (val_loss: 1.381, val_acc: 0.468), (
    test_Loss: 1.602, test_acc: 0.387) -- 127.02 sec
Epoch-21: (loss: 1.424, acc: 0.485, gen_loss: 0.000), (val_loss: 1.371, val_acc: 0.475), (
    test_Loss: 1.597, test_acc: 0.392) -- 127.02 sec
Epoch-22: (loss: 1.407, acc: 0.487, gen_loss: 0.000), (val_loss: 1.365, val_acc: 0.478), (
    test_Loss: 1.585, test_acc: 0.396) -- 127.04 sec
Epoch-23: (loss: 1.397, acc: 0.495, gen_loss: 0.000), (val_loss: 1.355, val_acc: 0.484), (
    test_Loss: 1.578, test_acc: 0.400) -- 126.98 sec
Epoch-24: (loss: 1.387, acc: 0.497, gen_loss: 0.000), (val_loss: 1.351, val_acc: 0.485), (
    test_Loss: 1.581, test_acc: 0.399) -- 127.02 sec
Epoch-25: (loss: 1.380, acc: 0.489, gen_loss: 0.000), (val_loss: 1.358, val_acc: 0.479), (
    test_Loss: 1.594, test_acc: 0.392) -- 126.93 sec
Epoch-26: (loss: 1.372, acc: 0.501, gen_loss: 0.000), (val_loss: 1.344, val_acc: 0.491), (
    test_Loss: 1.575, test_acc: 0.405) -- 127.01 sec
Epoch-27: (loss: 1.364, acc: 0.505, gen_loss: 0.000), (val_loss: 1.339, val_acc: 0.494), (
    test_Loss: 1.569, test_acc: 0.406) -- 129.98 sec
Epoch-28: (loss: 1.358, acc: 0.504, gen_loss: 0.000), (val_loss: 1.334, val_acc: 0.491), (
    test_Loss: 1.562, test_acc: 0.406) -- 132.31 sec
Epoch-29: (loss: 1.353, acc: 0.510, gen_loss: 0.000), (val_loss: 1.331, val_acc: 0.497), (
    test_Loss: 1.558, test_acc: 0.413) -- 132.06 sec
Epoch-30: (loss: 1.344, acc: 0.517, gen_loss: 0.000), (val_loss: 1.317, val_acc: 0.501), (
    test_Loss: 1.550, test_acc: 0.417) -- 132.13 sec
```

B.5 4conv.out

```
Using TensorFlow backend.
2018-12-03 20:33:11.621082: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found
    device 0 with properties:
name: Tesla K20Xm major: 3 minor: 5 memoryClockRate(GHz): 0.732
pciBusID: 0000:03:00.0
totalMemory: 5.57GiB freeMemory: 5.49GiB
2018-12-03 20:33:11.783667: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found
    device 1 with properties:
name: Tesla K20Xm major: 3 minor: 5 memoryClockRate(GHz): 0.732
pciBusID: 0000:83:00.0
totalMemory: 5.57GiB freeMemory: 5.49GiB
2018-12-03 20:33:11.783761: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1511] Adding
    visible gpu devices: 0, 1
2018-12-03 20:33:19.157765: I tensorflow/core/common_runtime/gpu/gpu_device.cc:982] Device
    interconnect StreamExecutor with strength 1 edge matrix:
2018-12-03 20:33:19.158315: I tensorflow/core/common_runtime/gpu/gpu_device.cc:988]  0 1
2018-12-03 20:33:19.158336: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 0:   N N
2018-12-03 20:33:19.158344: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 1:   N N
2018-12-03 20:33:19.158820: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created
    TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 5279 MB memory) ->
    physical GPU (device: 0, name: Tesla K20Xm, pci bus id: 0000:03:00.0, compute capability:
     3.5)
2018-12-03 20:33:19.162065: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created
    TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:1 with 5279 MB memory) ->
    physical GPU (device: 1, name: Tesla K20Xm, pci bus id: 0000:83:00.0, compute capability:
     3.5)
source dataset shape:
(220000, 2, 128)
Source domain partition summary:
X_train: (132000, 2, 128, 1)
Y_train: (132000, 11)
X_val: (44000, 2, 128, 1)
Y_val: (44000, 11)
X_test: (44000, 2, 128, 1)
Y_test: (44000, 11)
target dataset shape:
(220000, 2, 128)
```

```
Target domain partition summary:
X_tgt_train: (132000, 2, 128, 1)
Y_tgt_train: (132000, 11)
X_tgt_val: (44000, 2, 128, 1)
Y_tgt_val: (44000, 11)
X_tgt_test: (44000, 2, 128, 1)
Y_tgt_test: (44000, 11)
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2, 128, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 2, 128, 100)       1000
_________________________________________________________________
activation_1 (Activation)    (None, 2, 128, 100)       0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 1, 128, 100)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 1, 128, 150)       135150
_________________________________________________________________
activation_2 (Activation)    (None, 1, 128, 150)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 128, 150)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 128, 200)       270200
_________________________________________________________________
activation_3 (Activation)    (None, 1, 128, 200)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25600)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              26215424
_________________________________________________________________
activation_4 (Activation)    (None, 1024)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              1049600
_________________________________________________________________
activation_5 (Activation)    (None, 1024)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 1024)              0
```

```
_________________________________________________________________
dense_3 (Dense)              (None, 11)                11275
=================================================================
Total params: 27,682,649
Trainable params: 27,682,649
Non-trainable params: 0
_________________________________________________________________
None
Train Convnet source/target
Epoch-1: (loss: 2.267, acc: 0.199), (val_loss: 2.133, val_acc: 0.199), (test_Loss: 2.341,
    test_acc: 0.146) -- 73.69 sec
Epoch-2: (loss: 2.047, acc: 0.307), (val_loss: 1.893, val_acc: 0.306), (test_Loss: 2.088,
    test_acc: 0.235) -- 52.47 sec
Epoch-3: (loss: 1.834, acc: 0.368), (val_loss: 1.719, val_acc: 0.366), (test_Loss: 1.929,
    test_acc: 0.282) -- 52.48 sec
Epoch-4: (loss: 1.746, acc: 0.387), (val_loss: 1.673, val_acc: 0.382), (test_Loss: 1.896,
    test_acc: 0.296) -- 52.51 sec
Epoch-5: (loss: 1.699, acc: 0.393), (val_loss: 1.634, val_acc: 0.390), (test_Loss: 1.840,
    test_acc: 0.309) -- 52.48 sec
Epoch-6: (loss: 1.640, acc: 0.422), (val_loss: 1.563, val_acc: 0.418), (test_Loss: 1.771,
    test_acc: 0.340) -- 52.47 sec
Epoch-7: (loss: 1.577, acc: 0.438), (val_loss: 1.522, val_acc: 0.433), (test_Loss: 1.751,
    test_acc: 0.349) -- 52.46 sec
Epoch-8: (loss: 1.525, acc: 0.447), (val_loss: 1.473, val_acc: 0.440), (test_Loss: 1.682,
    test_acc: 0.367) -- 52.48 sec
Epoch-9: (loss: 1.484, acc: 0.463), (val_loss: 1.428, val_acc: 0.455), (test_Loss: 1.651,
    test_acc: 0.373) -- 52.48 sec
Epoch-10: (loss: 1.453, acc: 0.470), (val_loss: 1.411, val_acc: 0.464), (test_Loss: 1.637,
    test_acc: 0.379) -- 52.51 sec
Epoch-11: (loss: 1.433, acc: 0.476), (val_loss: 1.401, val_acc: 0.469), (test_Loss: 1.627,
    test_acc: 0.383) -- 52.49 sec
Epoch-12: (loss: 1.410, acc: 0.489), (val_loss: 1.368, val_acc: 0.481), (test_Loss: 1.597,
    test_acc: 0.393) -- 52.50 sec
Epoch-13: (loss: 1.396, acc: 0.488), (val_loss: 1.373, val_acc: 0.480), (test_Loss: 1.596,
    test_acc: 0.395) -- 52.49 sec
Epoch-14: (loss: 1.383, acc: 0.500), (val_loss: 1.348, val_acc: 0.492), (test_Loss: 1.581,
    test_acc: 0.403) -- 52.49 sec
Epoch-15: (loss: 1.376, acc: 0.499), (val_loss: 1.351, val_acc: 0.488), (test_Loss: 1.578,
    test_acc: 0.402) -- 52.49 sec
Epoch-16: (loss: 1.364, acc: 0.508), (val_loss: 1.331, val_acc: 0.498), (test_Loss: 1.559,
    test_acc: 0.410) -- 52.48 sec
```

```
Epoch-17: (loss: 1.356, acc: 0.497), (val_loss: 1.364, val_acc: 0.486), (test_Loss: 1.606,
    test_acc: 0.398) -- 52.47 sec
Epoch-18: (loss: 1.346, acc: 0.516), (val_loss: 1.321, val_acc: 0.502), (test_Loss: 1.556,
    test_acc: 0.412) -- 52.47 sec
Epoch-19: (loss: 1.332, acc: 0.515), (val_loss: 1.322, val_acc: 0.501), (test_Loss: 1.557,
    test_acc: 0.414) -- 52.48 sec
Epoch-20: (loss: 1.326, acc: 0.522), (val_loss: 1.310, val_acc: 0.507), (test_Loss: 1.543,
    test_acc: 0.419) -- 52.46 sec
Epoch-21: (loss: 1.318, acc: 0.526), (val_loss: 1.311, val_acc: 0.509), (test_Loss: 1.549,
    test_acc: 0.423) -- 52.49 sec
Epoch-22: (loss: 1.314, acc: 0.523), (val_loss: 1.317, val_acc: 0.508), (test_Loss: 1.554,
    test_acc: 0.420) -- 52.48 sec
Epoch-23: (loss: 1.310, acc: 0.534), (val_loss: 1.301, val_acc: 0.514), (test_Loss: 1.528,
    test_acc: 0.430) -- 52.49 sec
Epoch-24: (loss: 1.298, acc: 0.535), (val_loss: 1.299, val_acc: 0.515), (test_Loss: 1.534,
    test_acc: 0.430) -- 52.50 sec
Epoch-25: (loss: 1.295, acc: 0.533), (val_loss: 1.294, val_acc: 0.511), (test_Loss: 1.534,
    test_acc: 0.430) -- 52.47 sec
Epoch-26: (loss: 1.290, acc: 0.534), (val_loss: 1.314, val_acc: 0.510), (test_Loss: 1.546,
    test_acc: 0.427) -- 52.48 sec
Epoch-27: (loss: 1.282, acc: 0.542), (val_loss: 1.291, val_acc: 0.516), (test_Loss: 1.526,
    test_acc: 0.435) -- 52.45 sec
Epoch-28: (loss: 1.277, acc: 0.546), (val_loss: 1.290, val_acc: 0.518), (test_Loss: 1.529,
    test_acc: 0.434) -- 52.47 sec
Epoch-29: (loss: 1.276, acc: 0.546), (val_loss: 1.291, val_acc: 0.518), (test_Loss: 1.532,
    test_acc: 0.432) -- 52.47 sec
Epoch-30: (loss: 1.272, acc: 0.550), (val_loss: 1.291, val_acc: 0.517), (test_Loss: 1.524,
    test_acc: 0.435) -- 52.48 sec
```

Appendix C

Commdrcn Python

C.1 main sm.py

```
"""
Run DRCN on SVHN (source) --> MNIST (target)
Author: Muhammad Ghifary (mghifary@gmail.com)

Revisions made by: Kyle McClintick (kwmcclintick@wpi.edu)
Run DRCN on RML2016.10A (source) --> modified RML2016.10A (target)

This is the meta file. Datasets are loaded and partitioned by dataset.py
The DRCN model is then made, which both classifies and transforms a dataset as per the
    publication by Ghifary et al
The model is then trained using mini-batches
"""


from keras.utils import np_utils

from drcn import *
from myutils import *
from dataset import *

# Load datasets
print('Load datasets (* partitions used in DRCN)')
```

```
(X_train, y_train), (x_val, y_val) ,(X_test, y_test) = load_rmlsource(dataset='
    RML2016.10a.dat')  # source, train and validation data
(_, _), (_, _), (X_tgt_test, y_tgt_test) = load_rmltarget(dataset='RML2016.10a.targetd.dat')
     # target, test data

# Convert class vectors to binary class matrices
nb_classes = 11

print('Create DRCN Model')
drcn = DRCN()

input_shape = (X_train.shape[1], X_train.shape[2], X_train.shape[3])
# model has 3 conv layers with 100, 150, and 200 filters, of dimensions 3x3.
# model prevents overfitting via 2x2 maxpooling and 50% dropout layers
# activation function is ReLu, classification layers are softmax layers with 1024 nodes
drcn.create_model(input_shape=input_shape, dense_dim=1024, dy=nb_classes, nb_filters=[100,
    150, 200], kernel_size=(3, 3), pool_size=(2, 1),
                dropout=0.5, bn=False, output_activation='softmax', opt='adam')

print('Train DRCN...')
PARAMDIR = ''
CONF = 'svhn-mnist_drcn_v2'
drcn.fit_drcn(X_train, y_train, X_tgt_test, validation_data=(X_test, y_test), test_data=(
    X_tgt_test, y_tgt_test),
                          nb_epoch=30, batch_size=1024, PARAMDIR=PARAMDIR, CONF=CONF)

print('Train Convnet...')
# create convnet for benchmarking against the drcn
conv = DRCN()
conv.create_convnet(input_shape, dense_dim=1024, dy=11, nb_filters=[100,150,200], kernel_size
    =(3, 3), pool_size=(2, 1),
                        dropout=0.5, bn=False, output_activation='softmax', opt='adam')
# benchmark drcn against naked convnet trained and tested on source domain data
conv.fit_convnet(X_train, y_train, nb_epoch=30, batch_size=1024, shuffle=True,
                     validation_data=(x_val, y_val), test_data=(X_tgt_test, y_tgt_test),
                         PARAMDIR=PARAMDIR, CONF=CONF)
```

C.2 dataset.py

```
import gzip
import pickle
import numpy as np

'''
Load and partition the target domain dataset for main_sm.py
'''
def load_rmltarget(dataset='RML2016.10a.targetd.dat'):
    # import dataset
    Xd = pickle.load(open(dataset, 'rb'))

    # Split labels from data
    [snrs, mods] = map(lambda j: sorted(list(set(map(lambda x: x[j], Xd.keys())))), [1, 0])
    X = []
    lbl = []
    for mod in mods:
        for snr in snrs:
            X.append(Xd[(mod, snr)])
            for i in range(Xd[(mod, snr)].shape[0]):
                lbl.append((mod, snr))
    X = np.vstack(X)
    print('target dataset shape:')
    print(np.shape(X))

    # partition data into train, validation, test sets
    np.random.seed(2016)  # set seed for consistent partitioning
    n_examples = X.shape[0]
    n_train = int(n_examples * 0.8)  # ratio of training data
    train_idx = np.random.choice(range(0, n_examples), size=n_train, replace=False)  # take
        subset from dataset for train/val
    n_val = int(n_examples * 0.1)  # ratio of validation data, rest will be test data
    val_idx = np.random.choice(list(set(train_idx)), size=n_val, replace=False)  # val idxs
        picked from training subset
    test_idx = list(set(range(0, n_examples)) - set(train_idx))  # test idxs taken from
        remaining index values
    train_idx = list(set(train_idx) - set(val_idx))  # subtract validation idxs from training
         idxs
    X_train = X[train_idx]  # training data (not to be used in drcn)
    X_val = X[val_idx]  # validation data (not used in the drcn)
    X_test = X[test_idx]  # test data (used in drcn)
```

```
    Y_train = to_onehot(list(map(lambda x: mods.index(lbl[x][0]), train_idx)))  # training
        labels
    Y_val = to_onehot(list(map(lambda x: mods.index(lbl[x][0]), val_idx)))  # validation
        labels
    Y_test = to_onehot(list(map(lambda x: mods.index(lbl[x][0]), test_idx)))  # validation
        labels

    # reshape X dims for batch normalization later
    X_train = np.expand_dims(X_train, axis=3)
    X_val = np.expand_dims(X_val, axis=3)
    X_test = np.expand_dims(X_test, axis=3)
    print('target domain partition summary:')
    print('X_train: ' + str(np.shape(X_train)))
    print('Y_train: ' + str(np.shape(Y_train)))
    print('X_val: ' + str(np.shape(X_val)))
    print('Y_val: ' + str(np.shape(Y_val)))
    print('X_test*: ' + str(np.shape(X_test)))
    print('Y_test*: ' + str(np.shape(Y_test)))
    return (X_train, Y_train), (X_val, Y_val), (X_test, Y_test)

'''
Load and partition the source domain dataset for main_sm.py
'''
def load_rmlsource(dataset='RML2016.10a.dat'):
    # import dataset
    Xd = pickle.load(open(dataset, 'rb'))

    # Split labels from data
    [snrs, mods] = map(lambda j: sorted(list(set(map(lambda x: x[j], Xd.keys())))), [1, 0])
    X = []
    lbl = []
    for mod in mods:
        for snr in snrs:
            X.append(Xd[(mod, snr)])
            for i in range(Xd[(mod, snr)].shape[0]):
                lbl.append((mod, snr))
    X = np.vstack(X)
    print('source dataset shape:')
    print(np.shape(X))

    # partition data into train, validation, test sets
```

```
    np.random.seed(2016)  # set seed for consistent partitioning
    n_examples = X.shape[0]
    n_train = int(n_examples * 0.9)  # ratio of training data set here
    train_idx = np.random.choice(range(0, n_examples), size=n_train, replace=False)
    test_idx = list(set(range(0, n_examples)) - set(train_idx))
    X_train = X[train_idx]  # training data
    X_test = X[test_idx]  # drcn validation data

    Y_train = to_onehot(list(map(lambda x: mods.index(lbl[x][0]), train_idx)))  # training
        labels
    Y_test = to_onehot(list(map(lambda x: mods.index(lbl[x][0]), test_idx)))  # validation
        labels

    # reshape dims for batch normalization later
    X_train = np.expand_dims(X_train, axis=3)
    X_test = np.expand_dims(X_test, axis=3)
    print('source domain partition summary:')
    print('X_train*: ' + str(np.shape(X_train)))
    print('Y_train*: ' + str(np.shape(Y_train)))
    print('X_test*: ' + str(np.shape(X_test)))
    print('Y_test*: ' + str(np.shape(Y_test)))

    return (X_train, Y_train), (X_test, Y_test)
```

C.3 myutils.py

```
"""
Contains all helpers for DRCN
"""

from PIL import Image, ImageDraw
import numpy as np
from keras import backend as K
import os

'''
zero centers data. Not needed for RML
'''
```

```
def preprocess_images(X, tmin=-1, tmax=1):
    V = X * (tmax - tmin) / 255.
    V += tmin
    return V

'''
removes zero-centering from output data. Again, not needed for RML, its already zero centered
'''
def postprocess_images(V, omin=-1, omax=1):
    X = V - omin
    X = X * 255. / (omax - omin)
    return X


def show_images(Xo, padsize=1, padval=0, filename=None, title=None):
    # data format : channel_first
    X = np.copy(Xo)
    [n, c, d1, d2] = X.shape
    if c == 1:
        X = np.reshape(X, (n, d1, d2))

    n = int(np.ceil(np.sqrt(X.shape[0])))

    padding = ((0, n ** 2 - X.shape[0]), (0, padsize), (0, padsize)) + ((0, 0),) * (X.ndim -
        3)
    canvas = np.pad(X, padding, mode='constant', constant_values=(padval, padval))

    canvas = canvas.reshape((n, n) + canvas.shape[1:]).transpose((0, 2, 1, 3) + tuple(range
        (4, canvas.ndim + 1)))
    canvas = canvas.reshape((n * canvas.shape[1], n * canvas.shape[3]) + canvas.shape[4:])

    if title is not None:
        title_canv = np.zeros((50, canvas.shape[1]))
        title_canv = title_canv.astype('uint8')
        canvas = np.vstack((title_canv, canvas)).astype('uint8')

        I = Image.fromarray(canvas)
        d = ImageDraw.Draw(I)
        fill = 255
        d.text((10, 10), title, fill=fill, font=fnt)
    else:
```

```
        canvas = canvas.astype('uint8')
        I = Image.fromarray(canvas)

    if filename is None:
        I.show()
    else:
        I.save(filename)

    return I


def get_impulse_noise(X, level):
    p = 1. - level
    Y = X * np.random.binomial(1, p, size=X.shape)
    return Y


def get_gaussian_noise(X, std):
    # X: [n, c, d1, d2] images in [0, 1]
    Y = np.random.normal(X, scale=std)
    Y = np.clip(Y, 0., 1.)
    return Y


def get_flipped_pixels(X):
    # X: [n, c, d1, d2] images in [0, 1]
    Y = 1. - X
    Y = np.clip(Y, 0., 1.)
    return Y


def iterate_minibatches(inputs, targets, batchsize, shuffle=True):
    assert len(inputs) == len(targets)

    if shuffle:
        indices = np.arange(len(inputs))
        np.random.shuffle(indices)

    for start_idx in range(0, len(inputs), batchsize):
        end_idx = start_idx + batchsize
        if end_idx > len(inputs):
```

```
            end_idx = start_idx + (len(inputs) % batchsize)

        if shuffle:
            excerpt = indices[start_idx:end_idx]

        else:
            excerpt = slice(start_idx, end_idx)

        yield inputs[excerpt], targets[excerpt]


def accuracy(Y1, Y2):
    n = Y1.shape[0]
    ntrue = np.count_nonzero(np.argmax(Y1, axis=1) == np.argmax(Y2, axis=1))
    return ntrue * 1.0 / n

'''
saves weights and hyperparams
'''
def save_weights(model, PARAMDIR, CONF):
    # model: keras model
    print(' == save weights == ')

    # save weights
    PARAMPATH = os.path.join(PARAMDIR, '%s_weights.h5') % CONF
    model.save(PARAMPATH)

    # save architecture
    CONFPATH = os.path.join(PARAMDIR, '%s_conf.json') % CONF
    archjson = model.to_json()

    open(CONFPATH, 'wb').write(archjson)

'''
ReLU but capped at a value of one
'''
def clip_relu(x):
    y = K.maximum(x, 0)
    return K.minimum(y, 1)
```

```
def augment_dynamic(X, ratio_i=0.2, ratio_g=0.2, ratio_f=0.2):
    batch_size = X.shape[0]

    ratio_n = ratio_i + ratio_g + ratio_f

    num_noise = int(batch_size * ratio_n)
    idx_noise = np.random.choice(range(batch_size), num_noise, replace=False)

    ratio_i2 = ratio_i / ratio_n
    num_impulse = int(num_noise * ratio_i2)
    i1 = 0
    i2 = num_impulse
    idx_impulse = idx_noise[i1:i2]

    ratio_g2 = ratio_g / ratio_n
    num_gaussian = int(num_noise * ratio_g2)
    i1 = i2
    i2 = i1 + num_gaussian
    idx_gaussian = idx_noise[i1:i2]

    ratio_f2 = ratio_f / ratio_n
    num_flip = int(num_noise * ratio_f2)
    i1 = i2
    i2 = i1 + num_flip
    idx_flip = idx_noise[i1:i2]

    Xn = np.copy(X)

    # impulse noise
    Xn[idx_impulse] = get_impulse_noise(Xn[idx_impulse], 0.5)
    Xn[idx_gaussian] = get_gaussian_noise(Xn[idx_gaussian], 0.5)
    Xn[idx_flip] = get_flipped_pixels(Xn[idx_flip])
    return Xn
```

C.4 drcn.py

```
"""
DRCN main class
```

```
Dependency:
keras 2.2.4
Python 2.7
Tensorflow-gpu 1.12.0

Author: Muhammad Ghifary (mghifary@gmail.com)
Revisions by: Kyle McClintick (kwmcclintick@wpi.edu)

creates the convnet classifier and convae NN to transform data from source to the target
    domain
"""

from keras.models import Model
from keras.layers import Input, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D, UpSampling2D
from keras.layers.pooling import MaxPooling2D
from keras.layers.core import Activation, Dropout, Dense, Reshape
from keras.layers.normalization import BatchNormalization
from keras.optimizers import RMSprop, Adam
from keras.preprocessing.image import ImageDataGenerator

import os
import numpy as np
import time

from myutils import *  # contains all helpers for DRCN


class DRCN(object):
    def __init__(self, name='svhn-mnist'):
        """
        Class constructor
        """
        self.name = name

    '''
    Convnet breaks off half way through the CNN structure to classify the data given to it
    '''
    def create_convnet(self, _input, dense_dim=1024, dy=11, nb_filters=[100,150,200],
        kernel_size=(3, 3), pool_size=(2, 2),
                       dropout=0.5, bn=True, output_activation='softmax', opt='adam'):
```

```
"""
Create convnet model / encoder of DRCN

Args:
    _input (Tensor)        : input layer
    dense_dim (int)            : dimensionality of the final dense layers
    dy (int)           : output dimensionality
    nb_filter (list)       : list of #Conv2D filters
    kernel_size (tuple)    : Conv2D kernel size
    pool_size (tuple)      : MaxPool kernel size
    dropout (float)        : dropout rate
    bn (boolean)           : batch normalization mode
    output_activation (string) : act. function for output layer
    opt (string)           : optimizer

Store the shared layers into self.enc_functions list
"""

_h = _input

self.enc_functions = []  # to store the shared layers, will be used later for
    constructing conv. autoencoder
for i, nf in enumerate(nb_filters):
    enc_f = Conv2D(padding='same', filters=nf, kernel_size=kernel_size)
    _h = enc_f(_h)
    self.enc_functions.append(enc_f)

    _h = Activation(activation='relu')(_h)

    if i < 2:
        _h = MaxPooling2D(pool_size=pool_size, padding='same')(_h)

_h = Flatten()(_h)

enc_f = Dense(units=dense_dim)
_h = enc_f(_h)
self.enc_functions.append(enc_f)
if bn:
    _h = BatchNormalization()(_h)
_h = Activation(activation='relu')(_h)
```

```
    _h = Dropout(rate=dropout)(_h)

    enc_f = Dense(units=dense_dim)
    _h = enc_f(_h)
    self.enc_functions.append(enc_f)
    if bn:
        _h = BatchNormalization()(_h)
    _feat = Activation(activation='relu')(_h)
    _h = Dropout(rate=dropout)(_feat)
    # classification layer is a fully connected dense layer of output size equal to # of
        classes
    _y = Dense(units=dy, activation=output_activation)(_h)

    # convnet
    self.convnet_model = Model(inputs=_input, outputs=_y)
    self.convnet_model.compile(loss='categorical_crossentropy', optimizer=opt)
    print(self.convnet_model.summary())

    self.feat_model = Model(inputs=_input, outputs=_feat)

'''
This NN recreates the training data as a set from the other domain
'''
def create_model(self, input_shape=(1, 32, 32), dense_dim=1024, dy=11, nb_filters
    =[100,150,200], kernel_size=(3, 3),
                 pool_size=(2, 2), dropout=0.5, bn=True, output_activation='softmax', opt
                     ='adam'):
    """
    Create DRCN model: convnet model followed by conv. autoencoder

    Args:
        _input (Tensor)        : input layer
        dense_dim (int)            : dimensionality of the final dense layers
        dy (int)           : output dimensionality
        nb_filter (list)       : list of #Conv2D filters
        kernel_size (tuple)    : Conv2D kernel size
        pool_size (tuple)      : MaxPool kernel size
        dropout (float)        : dropout rate
        bn (boolean)           : batch normalization mode
        output_activation (string) : act. function for output layer
        opt (string)           : optimizer
```

```
"""

d1 = input_shape[0]
d2 = input_shape[1]
c = input_shape[2]

if opt == 'adam':
    opt = Adam(lr=3e-4)
elif opt == 'rmsprop':
    opt = RMSprop(lr=1e-4)

_input = Input(shape=input_shape)

# Create ConvNet
self.create_convnet(_input, dense_dim=dense_dim, dy=dy, nb_filters=nb_filters,
                    kernel_size=kernel_size, pool_size=pool_size, dropout=dropout,
                    bn=bn, output_activation=output_activation, opt=opt)

# Create ConvAE, encoder functions are shared with ConvNet
_h = _input

# Reconstruct Conv2D layers
for i, nf in enumerate(nb_filters):
    _h = self.enc_functions[i](_h)
    _h = Activation(activation='relu')(_h)
    if i < 2:
        _h = MaxPooling2D(pool_size=pool_size, padding='same')(_h)

[_, wflat, hflat, cflat] = _h.get_shape().as_list()
_h = Flatten()(_h)

# Dense layers
for i in range(len(nb_filters), len(self.enc_functions)):
    _h = self.enc_functions[i](_h)
    _h = Activation(activation='relu')(_h)

# Decoder
_h = Dense(units=dense_dim)(_h)
_h = Activation(activation='relu')(_h)

_xdec = Dense(units=wflat * hflat * cflat)(_h)
```

```
    _xdec = Activation(activation='relu')(_xdec)
    _xdec = Reshape((wflat, hflat, nb_filters[-1]))(_xdec)
    i = 0
    for nf in reversed(nb_filters):
        _xdec = Conv2D(filters=nf, kernel_size=kernel_size, padding='same')(_xdec)
        _xdec = Activation(activation='relu')(_xdec)

        if i > 1:
            _xdec = UpSampling2D(size=pool_size)(_xdec)
        i += 1

    _xdec = Conv2D(c, kernel_size=kernel_size, padding='same', activation=clip_relu)(
        _xdec)

    self.convae_model = Model(inputs=_input, outputs=_xdec)
    self.convae_model.compile(loss='mse', optimizer=opt)
    print(self.convae_model.summary())

'''
Trains the DRCN NN by performing SGD on the batch normalized weights
'''
def fit_drcn(self, X, Y, Xu, nb_epoch=50, batch_size=128, shuffle=True,
             validation_data=None, test_data=None, PARAMDIR=None, CONF=None):
    """
    DRCN algorithm:
        - i) train convnet on labeled source data, ii) train convae on unlabeled target
            data
        - include data augmentation and denoising

    Args:
        X (np.array)        : [n, d1, d2, c] array of source images
        Y (np.array)        : [n, dy] array of source labels
        Xu (np.array)       : [n, d1, d2, c] array of target images
        nb_epoch (int)      : #iteration of gradient descent
        batch_size (int)    : # data per batch
        shuffle (boolean)   : shuffle the data in a batch if True
        validation_data (tuple) : tuple of (Xval, Yval) array
        test_data       : tuple of (Xtest, Ytest) array
        PARAMDIR (string)   : directory to store the learned weights
        CONF (string)       : for naming purposes
```

```
"""
history = {}
history['losses'] = []
history['accs'] = []
history['gen_losses'] = []
history['val_losses'] = []
history['val_accs'] = []
history['test_losses'] = []
history['test_accs'] = []
history['elapsed_times'] = []

best_ep = 1

# begin training in mini batches
for e in range(nb_epoch):
    start_t = time.time()
    # convae training
    gen_loss = 0.
    n_batch = 0
    total_batches = Xu.shape[0] / batch_size

    for Xu_batch, Yu_batch in iterate_minibatches(Xu, np.copy(Xu), batch_size,
        shuffle=shuffle):
        Xu_batch = get_impulse_noise(Xu_batch, 0.5)
        l = self.convae_model.train_on_batch(x=Xu_batch, y=Yu_batch)
        gen_loss += l
        n_batch += 1

        if n_batch >= total_batches:  # break out, happens if batch size isn't exact
            multiple of n
            break

    gen_loss /= n_batch
    history['gen_losses'].append(gen_loss)

    # convnet training, using mini batches
    loss = 0.
    n_batch = 0
    total_batches = X.shape[0] / batch_size
```

```
for X_batch, Y_batch in iterate_minibatches(X, Y, batch_size, shuffle=shuffle):
    l = self.convnet_model.train_on_batch(X_batch, Y_batch)
    loss += l
    n_batch += 1

    if n_batch >= total_batches:  # break out, happens if batch size isn't exact
        multiple of n
        break

loss /= n_batch
history['losses'].append(loss)

# calculate accuracy
acc = accuracy(self.convnet_model.predict(X), Y)
history['accs'].append(acc)

elapsed_t = time.time() - start_t
history['elapsed_times'].append(elapsed_t)

val_loss = -1
val_acc = -1
best_val_acc = -1
# begin optimizing hyperparameters using validation on val data taken from source
     domain dataset
if validation_data is not None:
    (X_val, Y_val) = validation_data
    val_loss = 0.
    n_batch = 0
    for Xv, Yv in iterate_minibatches(X_val, Y_val, batch_size, shuffle=False):
        l = self.convnet_model.test_on_batch(Xv, Yv)
        val_loss += l
        n_batch += 1
    val_loss /= n_batch
    history['val_losses'].append(val_loss)

    val_acc = accuracy(self.convnet_model.predict(X_val), Y_val)
    history['val_accs'].append(val_acc)

test_loss = -1
test_acc = -1
# evaluate on test data taken from target domain dataset
```

```
if test_data is not None:
    (X_test, Y_test) = test_data
    test_loss = 0.
    n_batch = 0
    for Xt, Yt in iterate_minibatches(X_test, Y_test, batch_size, shuffle=False):
        l = self.convnet_model.test_on_batch(Xt, Yt)
        test_loss += l
        n_batch += 1

    test_loss /= n_batch
    history['test_losses'].append(test_loss)

    test_acc = accuracy(self.convnet_model.predict(X_test), Y_test)
    history['test_accs'].append(test_acc)

print('Epoch-%d: (loss: %.3f, acc: %.3f, gen_loss: %.3f), (val_loss: %.3f,
    val_acc: %.3f), (test_Loss: %.3f, test_acc: %.3f) -- %.2f sec' %
((e + 1), loss, acc, gen_loss, val_loss, val_acc, test_loss, test_acc, elapsed_t)
    )

# save weights at the highest accuracy
if PARAMDIR is not None:
    if (acc + val_acc) > best_val_acc:
        best_val_acc = (acc + val_acc)
        best_ep = e + 1
        CONFCNN = '%s_cnn' % CONF
        save_weights(self.convnet_model, PARAMDIR, CONFCNN)

        CONFCAE = '%s_cae' % CONF
        save_weights(self.convae_model, PARAMDIR, CONFCAE)
    else:
        print('do not save, best val_acc: %.3f at %d' % (best_val_acc, best_ep))

# store history
HISTPATH = '%s_hist.npy' % CONF
np.save(HISTPATH, history)

# visualization
if validation_data is not None:
    (X_val, Y_val) = validation_data
    Xsv = X_val[:100]
```

```
            Xs = postprocess_images(Xsv, omin=0, omax=1)
            imgfile = '%s_src.png' % CONF
            Xs = np.reshape(Xs, (len(Xs), Xs.shape[3], Xs.shape[1], Xs.shape[2]))
            show_images(Xs, filename=imgfile)

            Xs_pred = self.convae_model.predict(Xsv)
            Xs_pred = postprocess_images(Xs_pred, omin=0, omax=1)
            imgfile = '%s_src_pred.png' % CONF
            Xs_pred = np.reshape(Xs_pred, (len(Xs_pred), Xs_pred.shape[3], Xs_pred.shape
                [1], Xs_pred.shape[2]))
            show_images(Xs_pred, filename=imgfile)

        if test_data is not None:
            (X_test, Y_test) = test_data
            Xtv = X_test[:100]
            Xt = postprocess_images(Xtv, omin=0, omax=1)
            imgfile = '%s_tgt.png' % CONF
            Xt = np.reshape(Xt, (len(Xt), Xt.shape[3], Xt.shape[1], Xt.shape[2]))
            show_images(Xt, filename=imgfile)

            Xt_pred = self.convae_model.predict(Xtv)
            Xt_pred = postprocess_images(Xt_pred, omin=0, omax=1)
            imgfile = '%s_tgt_pred.png' % CONF
            Xt_pred = np.reshape(Xt_pred, (len(Xt_pred), Xt_pred.shape[3], Xt_pred.shape
                [1], Xt_pred.shape[2]))
            show_images(Xt_pred, filename=imgfile)

###  just in case want to run convnet and convae separately, below are the training
    modules  ###
def fit_convnet(self, X, Y, nb_epoch=50, batch_size=128, shuffle=True,
                validation_data=None, test_data=None, PARAMDIR=None, CONF=None):

    history = {}
    history['losses'] = []
    history['accs'] = []
    history['val_losses'] = []
    history['val_accs'] = []
    history['test_losses'] = []
    history['test_accs'] = []
    history['elapsed_times'] = []
```

```
best_ep = 1
for e in range(nb_epoch):
    loss = 0.
    n_batch = 0
    start_t = time.time()
    for X_batch, Y_batch in iterate_minibatches(X, Y, batch_size, shuffle=shuffle):
        l = self.convnet_model.train_on_batch(X_batch, Y_batch)
        loss += l
        n_batch += 1

    elapsed_t = time.time() - start_t
    history['elapsed_times'].append(elapsed_t)

    loss /= n_batch
    history['losses'].append(loss)

    # calculate accuracy
    acc = accuracy(self.convnet_model.predict(X), Y)
    history['accs'].append(acc)

    val_loss = -1
    val_acc = -1
    best_val_acc = -1
    if validation_data is not None:
        (X_val, Y_val) = validation_data
        val_loss = 0.
        n_batch = 0
        for Xv, Yv in iterate_minibatches(X_val, Y_val, batch_size, shuffle=False):
            l = self.convnet_model.test_on_batch(Xv, Yv)
            val_loss += l
            n_batch += 1
        val_loss /= n_batch
        history['val_losses'].append(val_loss)

        val_acc = accuracy(self.convnet_model.predict(X_val), Y_val)
        history['val_accs'].append(val_acc)

    test_loss = -1
    test_acc = -1
    if test_data is not None:
```

```
            (X_test, Y_test) = test_data
            test_loss = 0.
            n_batch = 0
            for Xt, Yt in iterate_minibatches(X_test, Y_test, batch_size, shuffle=False):
                l = self.convnet_model.test_on_batch(Xt, Yt)
                test_loss += l
                n_batch += 1

            test_loss /= n_batch
            history['test_losses'].append(test_loss)

            test_acc = accuracy(self.convnet_model.predict(X_test), Y_test)
            history['test_accs'].append(test_acc)

        print(
        'Epoch-%d: (loss: %.3f, acc: %.3f), (val_loss: %.3f, val_acc: %.3f), (test_Loss:
            %.3f, test_acc: %.3f) -- %.2f sec' % \
        ((e + 1), loss, acc, val_loss, val_acc, test_loss, test_acc, elapsed_t))

        if PARAMDIR is not None:
            if (acc + val_acc) > best_val_acc:
                best_val_acc = (acc + val_acc)
                best_ep = e + 1
                save_weights(self.convnet_model, PARAMDIR, CONF)
            else:
                print('do not save, best val_acc: %.3f at %d' % (best_val_acc, best_ep))

        # store history
        HISTPATH = '%s_hist.npy' % CONF
        np.save(HISTPATH, history)

def fit_convae(self, X, nb_epoch=50, batch_size=128, shuffle=True,
               validation_data=None, test_data=None, PARAMDIR=None, CONF=None):

    history = {}
    history['losses'] = []
    history['val_losses'] = []
    history['test_losses'] = []
    history['elapsed_times'] = []

    best_ep = 1
```

```
for e in range(nb_epoch):
    loss = 0.
    n_batch = 0
    start_t = time.time()
    for X_batch, Y_batch in iterate_minibatches(X, np.copy(X), batch_size, shuffle=
        shuffle):
        l = self.convae_model.train_on_batch(X_batch, Y_batch)
        loss += l
        n_batch += 1

    elapsed_t = time.time() - start_t
    history['elapsed_times'].append(elapsed_t)

    loss /= n_batch
    history['losses'].append(loss)

    val_loss = -1
    best_val_loss = 100000

    test_loss = -1

    print('Epoch-%d: (loss: %.3f), (val_loss: %.3f), (test_Loss: %.3f) -- %.2f sec' %
         \
          ((e + 1), loss, val_loss, test_loss, elapsed_t))

    if PARAMDIR is not None:
        if loss < best_val_loss:
            best_val_loss = loss
            best_ep = e + 1
            save_weights(self.convae_model, PARAMDIR, CONF)
        else:
            print('do not save, best val loss: %.3f at %d' % (best_val_loss, best_ep)
                )

    # store history
    HISTPATH = '%s_hist.npy' % CONF
    np.save(HISTPATH, history)

    # visualization
    if validation_data is not None:
        Xtv = validation_data
```

```
        Xt = postprocess_images(Xtv, omin=0, omax=1)
        imgfile = '%s_tgt.png' % CONF
        Xt = np.reshape(Xt, (len(Xt), Xt.shape[3], Xt.shape[1], Xt.shape[2]))

        show_images(Xt, filename=imgfile)

        Xt_pred = self.convae_model.predict(Xtv)
        Xt_pred = postprocess_images(Xt_pred, omin=0, omax=1)
        imgfile = '%s_tgt_pred.png' % CONF

        Xt_pred = np.reshape(Xt_pred, (len(Xt_pred), Xt_pred.shape[3], Xt_pred.shape
            [1], Xt_pred.shape[2]))
        show_images(Xt_pred, filename=imgfile)

    if test_data is not None:
        Xsv = test_data
        Xs = postprocess_images(Xsv, omin=0, omax=1)
        imgfile = '%s_src.png' % CONF
        Xs = np.reshape(Xs, (len(Xs), Xs.shape[3], Xs.shape[1], Xs.shape[2]))
        show_images(Xs, filename=imgfile)

        Xs_pred = self.convae_model.predict(Xsv)
        Xs_pred = postprocess_images(Xs_pred, omin=0, omax=1)
        imgfile = '%s_src_pred.png' % CONF
        Xs_pred = np.reshape(Xs_pred, (len(Xs_pred), Xs_pred.shape[3], Xs_pred.shape
            [1], Xs_pred.shape[2]))
        show_images(Xs_pred, filename=imgfile)
```