Synthesizing Robust Training Data for Machine Learning

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

x

x

Abstract

Developing machine learning-based signal classifiers that generalize well requires training data that capture 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, low bias, low decay framework that can support arbitrary baseband signals and channel conditions is required. Furthermore, domain adaptation can allow for powerful generalized training via transforms on unlabeled evaluation-time signals. Together, domain transforms and a robust training set can train a deep neural net to perform well in many real-world scenarios.

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 Kaveh Pahlavan and Dr. Travis Collins for serving on my committee and providing valuable suggestions and comments with regards to my 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 friends from Audrey Ave: Alex and the Armstrongs, Ben, Elissa, Ivan, Thomas, and Vlad, my fraternity brothers from Lambda Chi Alpha and Theta Xi, who are too many to mention here, and the Lakeside bunch: Alia, Kelly, Gene, Jack, Jon, Nate, and Nic. 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. Without their constant support I wouldn't have finished this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

what deep nets offer the communications world, why generalized training is needed to make deep nets perform well

1.2 State of the Art

current forms of communications generalized training

discuss rml

1.3 Current Issues

issues in generalized training...what happens when false assumptions made? perturbations?

1.4 Thesis Contributions

This work contains the following contributions:

- A low decay, low bias framework is presented in Chapter 3, as well as example waveforms and data sets that have low entropy.

- Ongoing work on wireless channel domain adaptation in Chapter 4, including neural net architecture and initial results.

1.5 Thesis Organization

The thesis will be organized as follows: Chapter 2 will give a survey of background knowledge learned by the author on the topics of wireless channel modeling (Section 2.1), neural networks with an emphasis on training and data sets (Section 2.2), and modulation classification (Section 2.3). Chapter 3 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 4 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 5.

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." VTC Chicago, Fall 2018.
- K. Gill, K. McClintick, N. Kanthasamy, "Experimental Test-Bed for Bumblebee-Inspired Channel Selection in an Ad-hoc Network." VTC Chicago, Fall 2018.

Chapter 2

Background

Although transmitted wave-forms begin as well defined, man-made, synthetic structures, a virtually endless number of probabilistic, and sometimes non-linear, phenomenon alter the observed wave-forms receive-side [13]. Even within a single phenomenon, there can exist again a virtually endless 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, and information theory model used to mimic the effects of many wide-band noise sources
2. Path loss reduction of signal power density due to refraction, diffraction, absorption, aperture-medium coupling loss, and free-space loss
3. Doppler shifts resulting from motion of the transmitter, receiver, or scatterers and reflectors within the wireless channel
4. Coupled noise resulting from inter-modulation, interference from same and adjacent channels, industrial noise, Cosmic and terrestrial events
5. Carrier frequency offset of both the transmitter and receiver's local oscillators, which drive each radio's mixers
6. Phase ambiguity introduced by the unknown distance between transmitter and receiver

7. Random symbol timing offset resulting from independently running sample clocks
8. IQ imbalance resulting from phase and magnitude mismatches between the sine and cosine sections of receiver and transmitter chains
9. Rounding of sampled voltages and digital filter coefficients due to Quantization
10. Electronic Noise caused by semi-conductors such as shot and flicker noise

In this chapter, background information is given on classical channel model theory in Section 2.1 to support Chapter 3. Training of Neural Networks is described in Section 2.2, and in Section 2.3 a survey of modulation classification is discussed to support Chapter 4.

2.1 Classical Channel Models

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 chain of tasks (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. The first such model that is often discussed in this field is the model for Additive White Gaussian Noise (AWGN).

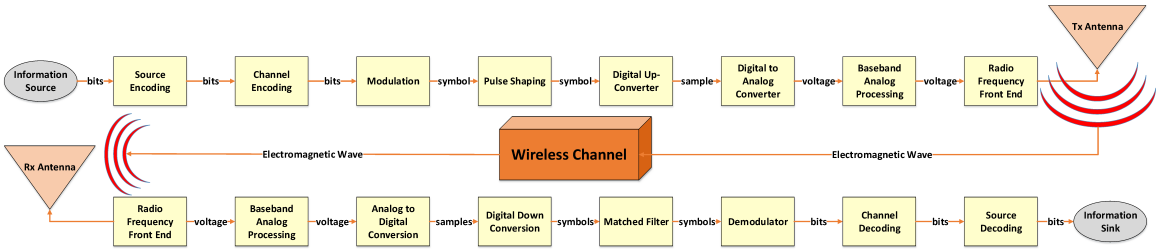


Figure 2.1: A flow chart of a transmit and receive chain of communications tasks. Arrows indicate movement of information from one block to another. The form that information takes at each step is communicated through annotations. Antennas are pictured as upside-down triangles.

2.1.1 Additive White Gaussian Noise (AWGN)

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 states that as independent random variables are added together, the more there are, the closer their joint distribution approaches a Gaussian probability density function (PDF):

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

AWGN is described as additive because it is added to any transmission, no matter its location. Additionally, there exists enough noise sources to approximate the CLT at all frequencies (see Figure 2.2) such that the noise has uniform power across the frequency band. This is why AWGN is described as white, it is an analogy for how white frequencies of optical wavelengths are the sum of all other colors of the visible light frequency band.

Claude Shannon proved in [14] that 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, a mid-1700's 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.

2.1.2 Path Loss

A key contributer to SNR is the received power, P_r. According to the Friis formula, the received power can be calculated in dBm as:

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

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

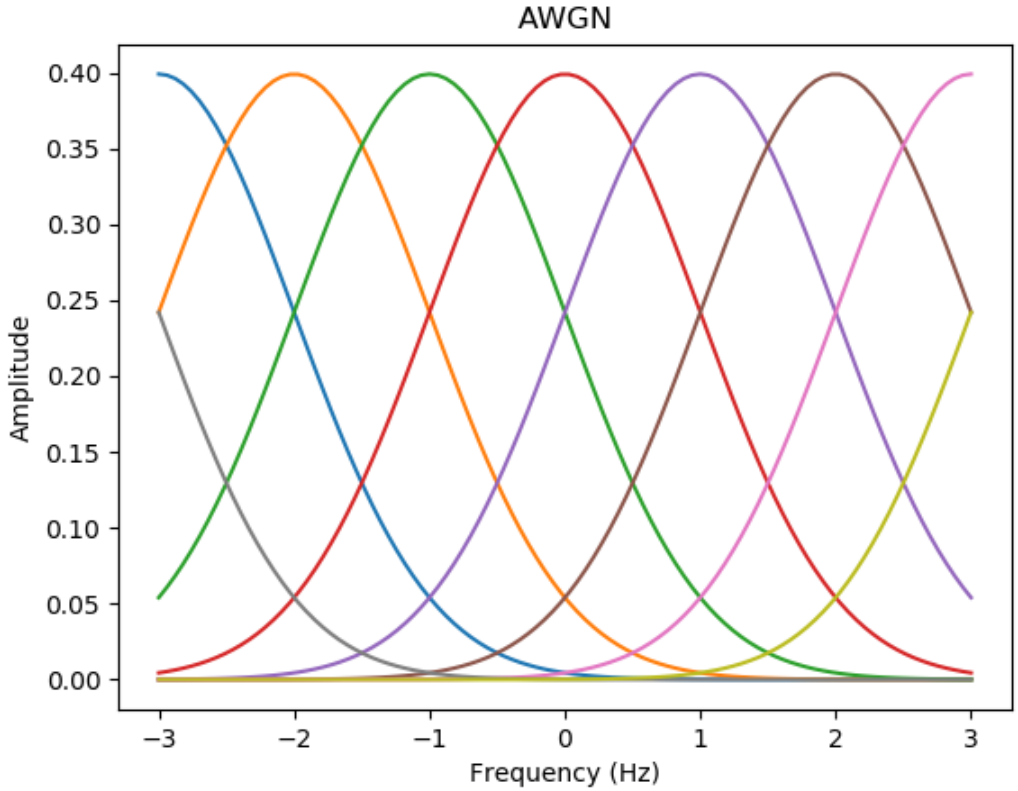


Figure 2.2: A cartoon showing the sum of several Gaussian PDFs (2.1) at varying frequencies. Amplitude and location of PDFs are arbitrary and not representative of any measurement or formal model.

formula has a few assumptions: far-field transmission ($d \gg \lambda$), the signal is narrow-band, and antennas are isotropic in the direction of transmission. Notice that power received decreases with both increased frequency and distance.

This section will discuss the relationship between received power and distance between transmitter and receiver of both wide-band and narrow-band signals. A narrow-band signal's 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}:

$$\tau_{rms} = \sqrt{\overline{\tau^2} + (\overline{\tau})^2}, \tag{2.4}$$

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

$$\frac{5}{\tau_{rms}} < BW. \tag{2.5}$$

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

$$\overline{\tau^n} = \frac{\sum_{i=1}^{L} \tau_i^n |\beta_i|^2}{\sum_{i=1}^{L} |\beta_i|^2}, \tag{2.6}$$

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:

$$a_i = \sum_{j=1}^{K_i} a_{ij}, \tag{2.7}$$

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:

$$P_0 = P_t G_r G_t (\lambda / 4\pi)^2. \tag{2.8}$$

A popular model for representation of a channel with path loss is the discrete delay channel model [1] (see Figure 2.3). However, this model behaves differently for narrow and wide-band signals.

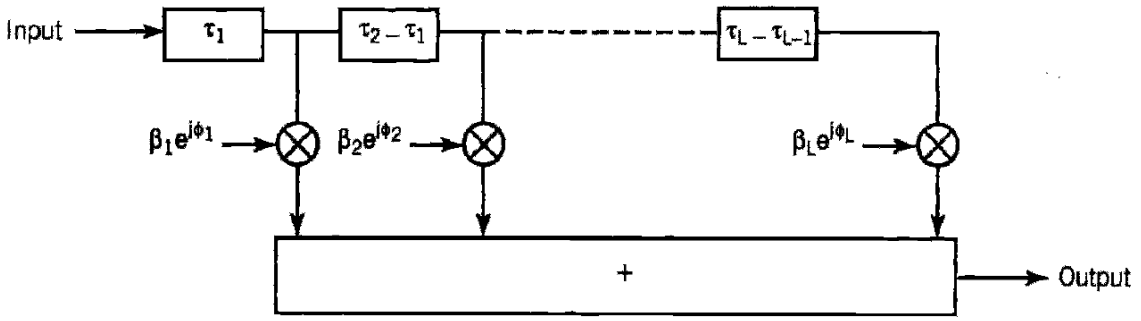


Figure 2.3: The discrete delay channel model. Inputs each have isolated time delays τ_i, ray powers $|\beta_i|^2 = A_0 a_i / d_i$, and ray phases $e^{j\phi_i}$.

Narrow-Band Signal

A signal is determined to be narrow-band using (2.5), the received power can be formulated as:

$$P_r = P_0 \left| \sum_{i=1}^{L} \frac{a_i}{d_i} e^{j\phi_i} \right| \tag{2.9}$$

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 both constructively and destructively (see Figure 2.4).

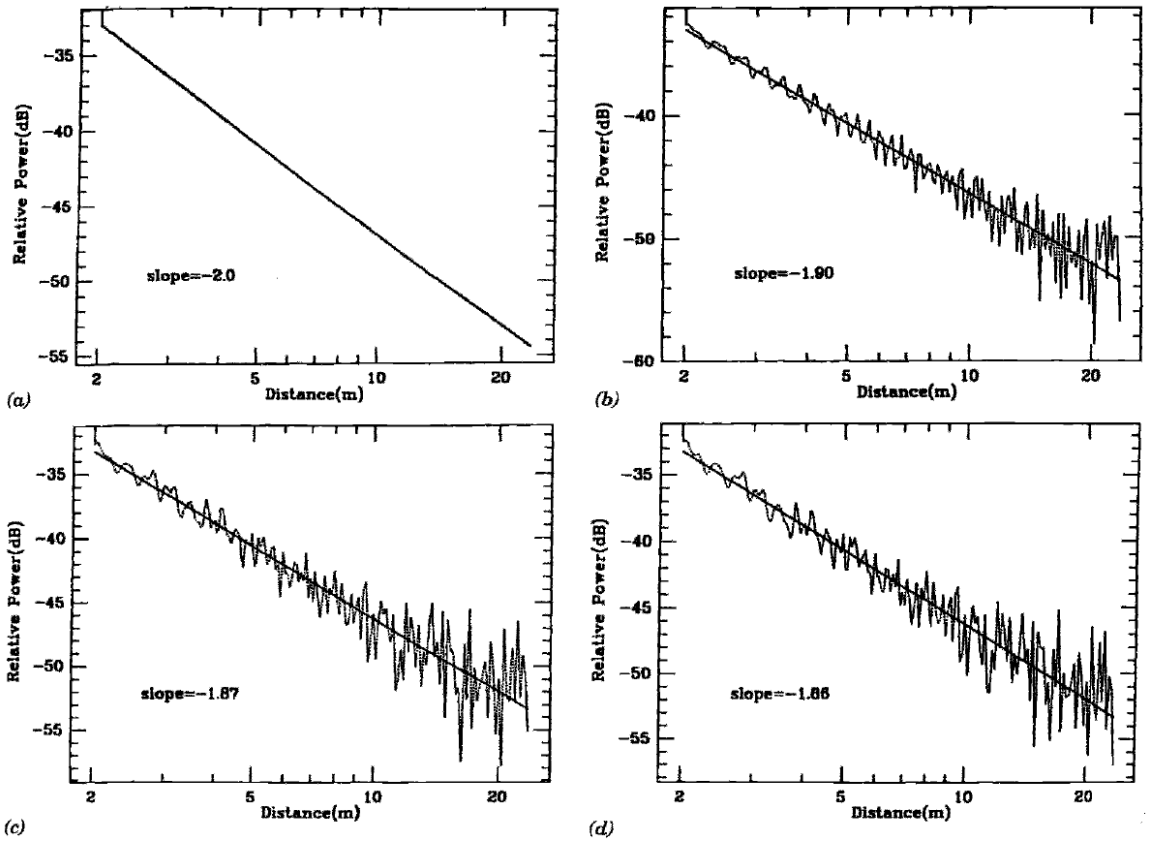


Figure 2.4: A series of narrow-band transmissions from [1], received in a room obtained by the 2D ray-tracing model (2.9). a) Line of Sight (LOS) path (no reflections). b) First-order reflection ($K = 2$ for coefficients equation (2.7)). c) Second-order reflection, $K = 3$. d) Third-order reflection, $K = 4$. Notice that higher order reflections have higher frequency changes in power received as distance increases, and that the average power (black line) decreases with distance due to (2.3).

Wide-Band Signal

A wide-band signal in the frequency domain can be shown to be of a short time duration in the time domain [2]. Often these bursts of signals are modeled as impulses, $\delta(t)$. In ideal wide-band communications, each path of arrival an impulse makes from transmitter to receiver are isolated. Additionally, since impulse's duration are instantaneous compared to time delays τ_i, the phase offset of each ray does not add constructively or destructively. Consequentially, and similarly to (2.9), the received power of a wide-band signal can be formulated as [1]:

$$P_r = P_0 \left| \sum_{i=1}^{L} \frac{a_i}{d_i} \right| = \sum_{i=1}^{L} |\beta_i|^2, \tag{2.10}$$

with the phase term $e^{j\phi_i}$ missing. The result is that wide-band path loss doesn't vary from (2.3) very much, unlike Figure 2.4. For a series of wide-band path loss experiments, see Figure 2.5.

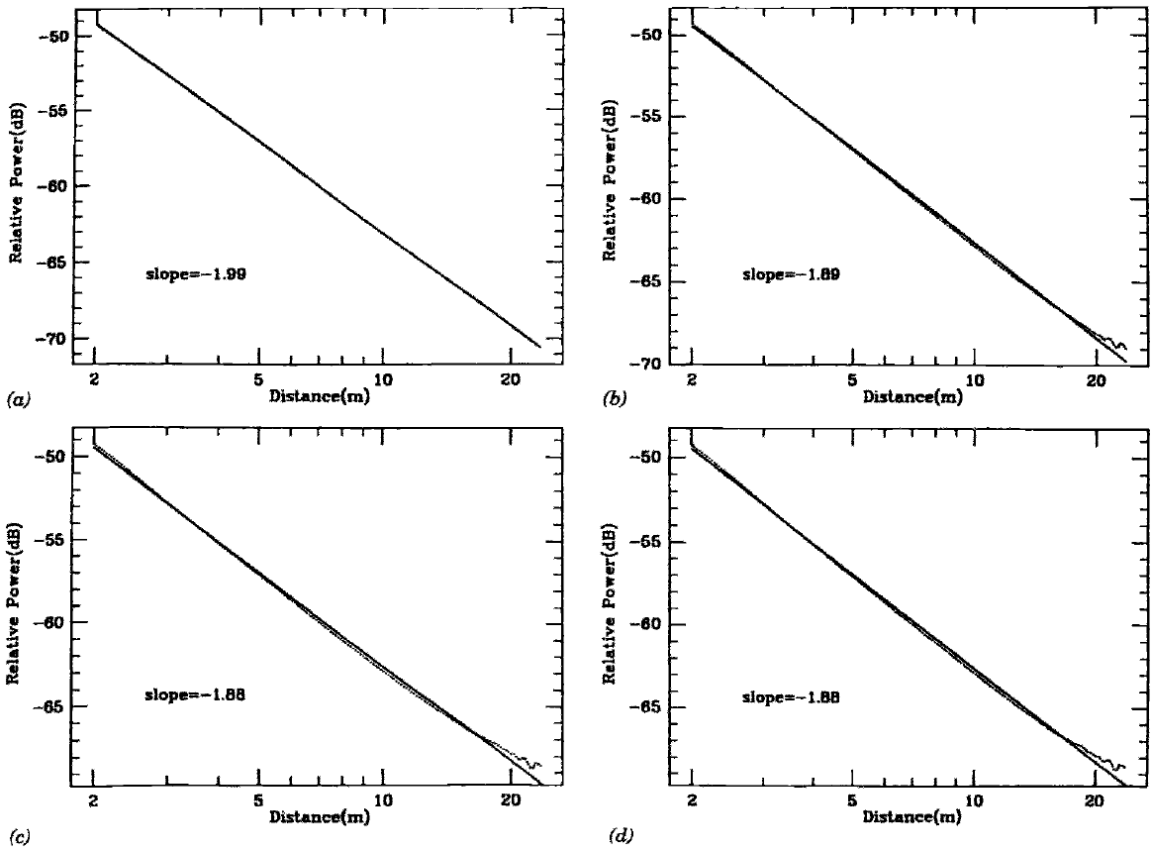


Figure 2.5: A series of wide-band transmissions from [1], received in a room obtained by the 2D ray-tracing model (2.9). a) Line of Sight (LOS) path (no reflections). b) First-order reflection ($K = 2$ for coefficients equation (2.7)). c) Second-order reflection, $K = 3$. d) Third-order reflection, $K = 4$. Average power (black line) decreases with distance due to (2.3). For higher order reflections the power received is higher because the impulse signals are not totally isolated.

Knife-Edge Diffraction Model

A popular method used to modify (2.3) is the knife-edge diffraction model [2]. 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:

$$v = h\sqrt{\frac{2(d_1 + d_2)}{\lambda d_1 d_2}}, \tag{2.11}$$

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

$$G_{dB} = 20 \log |F(v)|, \tag{2.12}$$

where the Fresnel integral can be calculated as:

$$F(v) = \frac{(1+j)}{2} \int_v^\infty e^{(-j\pi t^2)/2} dt. \tag{2.13}$$

While it is often sufficient [2] 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.

Topographical Path Loss Models

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 path loss model to be used in the free space Friis formula (2.3) is the Okumara-Hat a [15] empirical path loss model, whose behavior was collected in Tokyo, Japan using isotropic antennas, and has seen widespread use. 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 [16] have been conducted to expand the range of distances and frequencies the model can accommodate. The original Okumara model can be described [2] as use of the Friis formula (2.3) where L_{path} is instead:

$$L_{50,dB} = L_F + A_{mu}(f, d) - G(h_{te}) - G(h_{re}) - G_{AREA}, \tag{2.14}$$

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 2.8), $G(h_{te}), G(h_{re}$ are the transmitting and receiving antenna gain factors (2.15), respectively, and G_{AREA} is the gain with respect to the type of environment (see Figure 2.7).

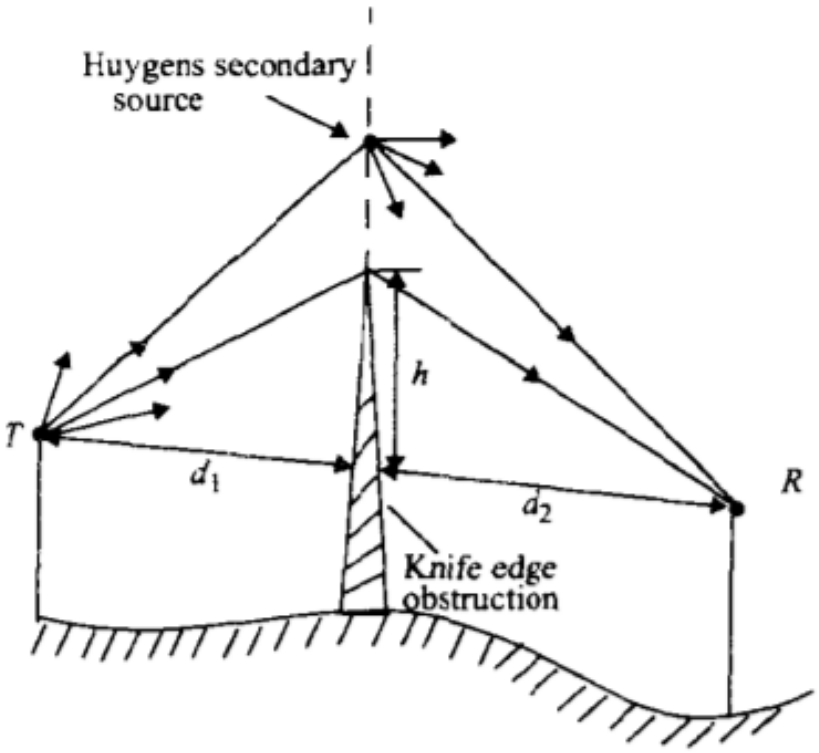


Figure 2.6: An illusration [2] of (2.12), where Υ is the transmitter, R is the receiver, h is the height of the obstruction starting from the direct path from Υ to R, and d_1, d_2 from the transmitter and receiver to the obstruction, respectively. The Huygens secondary source mimics a potentially strong reflected path, often taking the form of a reflection off a layer of the earth's ionosphere (see Section 2.1.4).

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

$$G(h_{te}) = 20\log\left(\tfrac{h_{te}}{200}\right), \quad 1000m > h_{te} > 30m \tag{2.15a}$$

$$G(h_{re}) = 10\log\left(\tfrac{h_{re}}{3}\right), \quad h_{re} \leq 3m \tag{2.15b}$$

$$G(h_{re}) = 20\log\left(\tfrac{h_{re}}{3}\right), \quad 10m > h_{re} > 3m. \tag{2.15c}$$

Just the Beginning: Fading

Path loss ideas visited in this section have just surveyed the most popular models. How do these models change in a wireless channel over time? A host of path loss and fading

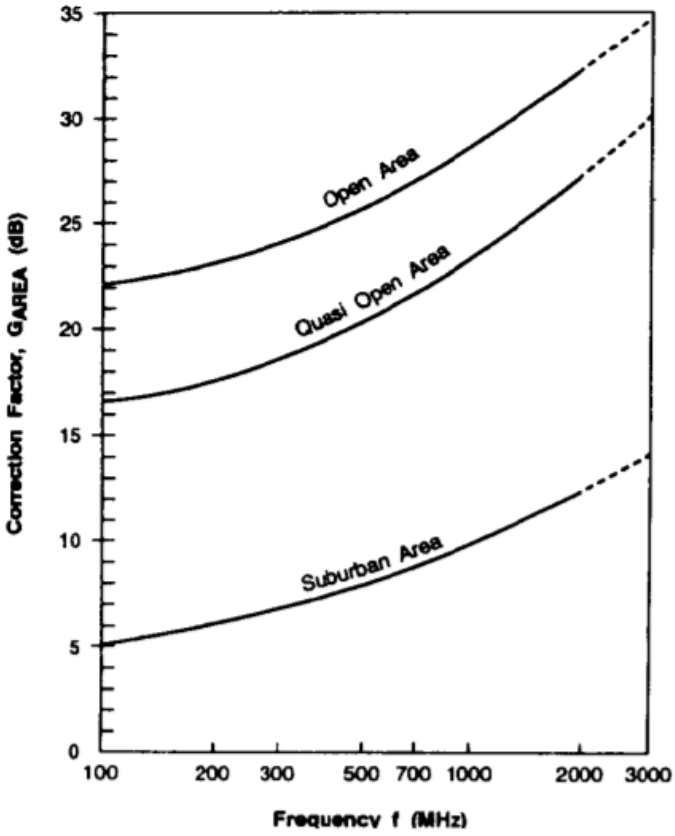


Figure 2.7: A plot [2] of correction factor, G_{AREA} to be used in (2.14) for various frequencies in open, quasi-open, and suburban areas (urban not listed).

(or shadowing) models exist for high and low altitude antennas of high and low frequency transmissions over short and large distances with narrow and wide bandwidths in both LOS and NLOS situations [1, 2, 16]. An exhaustive list of industry standard models can be found in [16]. Fading is a term used to describe variations in the path loss of a system over time, and can be broken into two categories: fast-fading ($1/D_S$ large (2.24)) and slow-fading ($1/D_S$ small (2.24)). Sources of slow-fading include large obstructions such as mountains, seasonal changes to the ionosphere (see Section 2.1.4), and any gradual change to the wireless channel such as urban construction. Fast-fading sources include weather changes, activity in the earth's ionosphere (see Section 2.1.4), solar flares, and any rapid changes to the wireless channel.

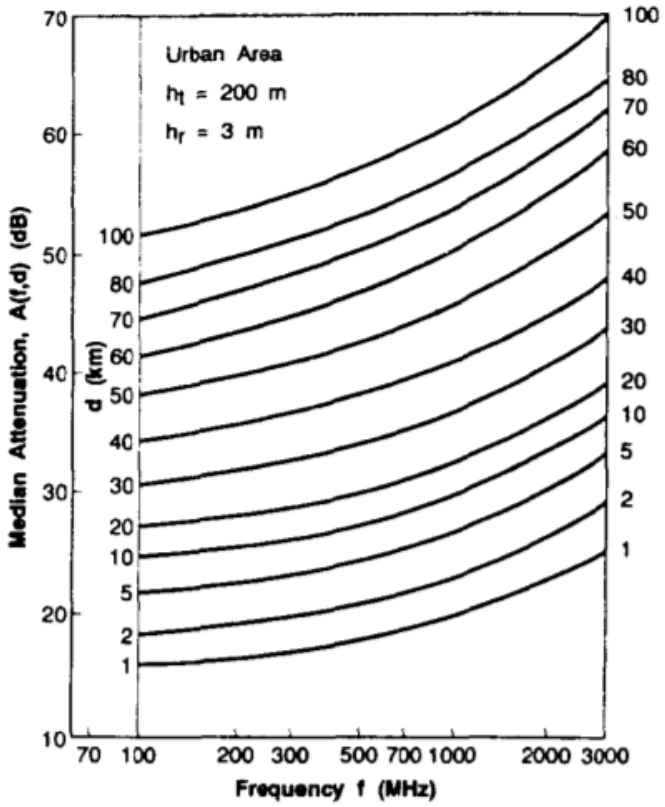


Figure 2.8: A plot [2] of $A_{mu}(f, d)$, the median attenuation relative to free space, used in (2.14). Plot assumes base-station height $h_t = 200m$, mobile receiver height $h_r = 3m$, an Urban area, and various distances and frequencies.

2.1.3 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, as in Figure 2.9.

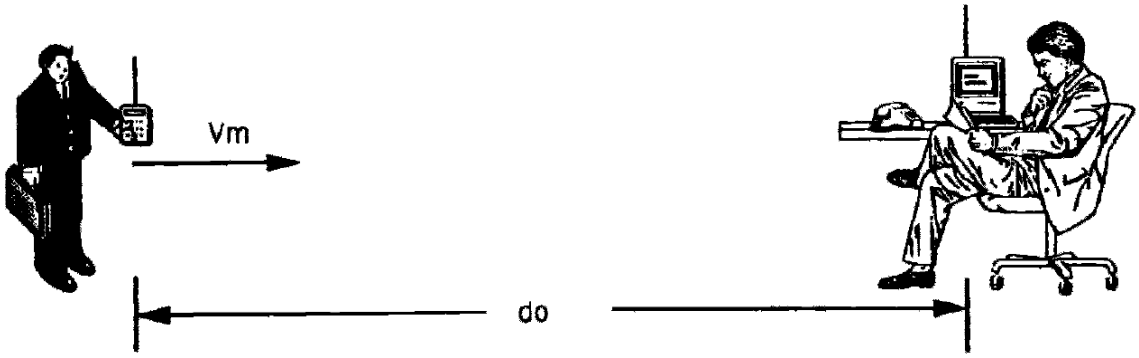


Figure 2.9: An illustration [1] of a radio link between a moving transmitter (left) and stationary receiver (right). Transmitter is moving at velocity V_m at an instantaneous distance d_0 from the receiver.

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

$$\tau(t) = \tau_0 - \frac{V_m}{c}t, \tag{2.16}$$

where c is the speed of light in free-space, 3e8 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:

$$r(t) = A_r e^{j2\pi f_c[t-\tau(t)]} = A_r e^{j[2\pi(f_c+f_d)t-\phi]}, \tag{2.17}$$

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,

$$f_d = \frac{V_m}{c} f_c \cos(\theta), \tag{2.18}$$

where θ is the direction of movement, for zero degrees being moving straight towards the receiver. This frequency shift is observed by the receiver, 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 (2.18) 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 2.1.2, rarely is there one ray (see Figure 2.3) in a wireless channel. Each ray is effected differently, and as consequence the frequency domain representation of the signal is affected by a Doppler spectrum, or spread $D(\lambda)$, derived as follows:

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 $y(t) = x(t) \circledast h(t)$.

$$h(\tau, t) = \sum_{i=1}^{L} \beta_i e^{j\phi_i} \delta(t - \tau_i), \tag{2.19}$$

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

$$R_{hh}(\tau_1, \tau_2; t_1, t_2) = R_{hh}(\tau_1; \Delta t)\delta(\tau_1 - \tau_2), \tag{2.20}$$

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:

$$R_{Hh}(f_1, f_2; \Delta t) = R_{Hh}(\Delta f; \Delta t), \tag{2.21}$$

where $\Delta f = f_2 - f_1$ and the channel is assumed to have Wide-Sense Stationary Uncorrelated Scattering (WSSUS), which is valid for most wireless channels [1], and implies that the signal variations on paths of different delays τ_i are uncorrelated, and the correlations between paths of equal delays are stationary, or time-invariant. Additionally, 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 (2.21):

$$R_{HH}(\Delta f; \lambda) = \int_{-\infty}^{\infty} R_{Hh}(\Delta f; \Delta t)e^{-j2\pi\lambda\Delta t} d(\Delta t), \tag{2.22}$$

as:

$$D(\lambda) = R_{HH}(0; \lambda). \tag{2.23}$$

The spectrum is limited by $\pm f_M$ (2.18), and the amount of variation of the spectrum over frequency is described by the Doppler spread:

$$B_{D,rms}^2 = \frac{\int_{-f_M}^{f_M} \lambda^2 D(\lambda) d\lambda}{\int_{-f_M}^{f_M} D(\lambda) d\lambda}. \tag{2.24}$$

Equations (2.20) through (2.22) are summarized in Figure 2.10.

In the most basic multi-path case, (2.23) takes the form

$$D(f) = \frac{1}{2\pi f_m} \left[1 - \left(\frac{f}{f_m}^2\right)\right]^{-1/2}, |f| \leq f_m, \tag{2.25}$$

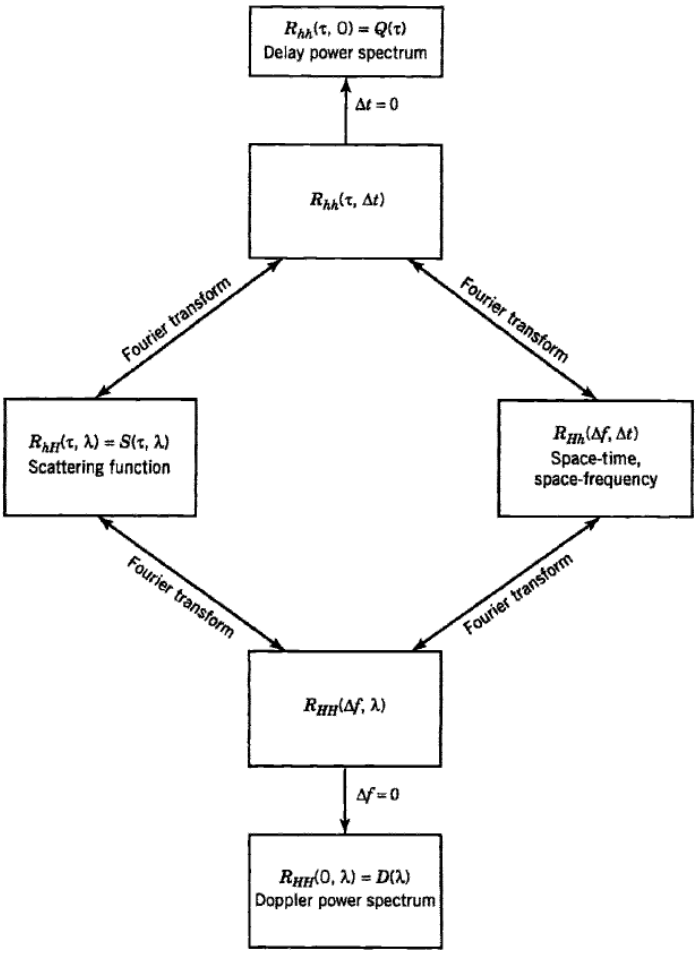


Figure 2.10: A flow chart [1] summarizing equations (2.20) through (2.22). Arrows indicate Fourier (down) and inverse Fourier (up) transforms.

or Jakes Doppler spectrum, where (2.18) is assumed and can be plotted over normalized frequency as seen in Figure 2.11.

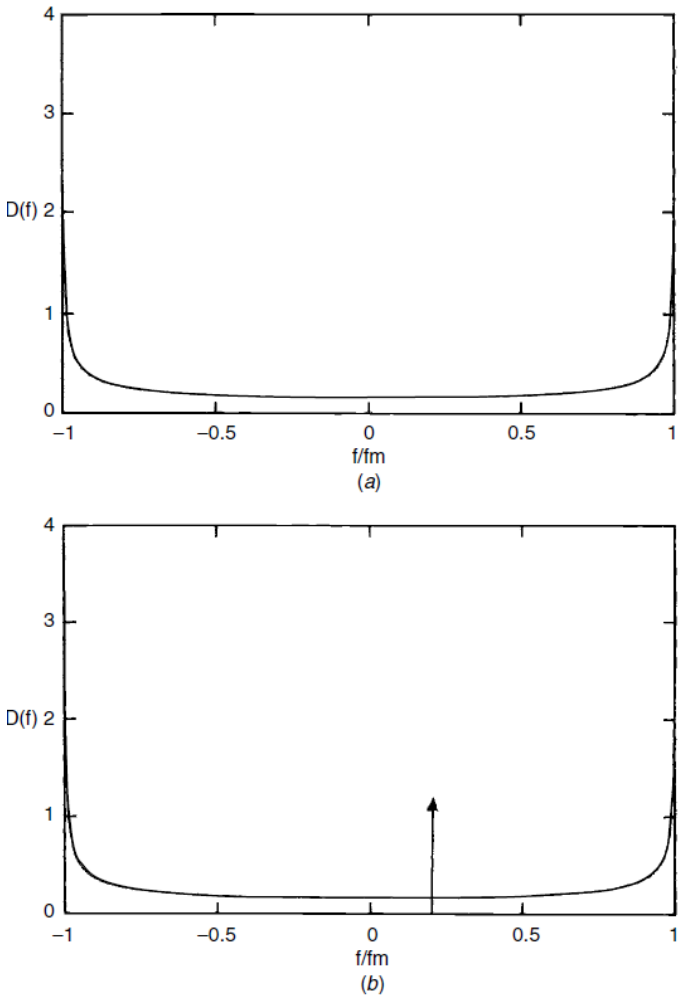


Figure 2.11: Jakes Doppler spectrum [1] for a non-line-of-sight (a) and line-of-sight (b) transmission. Horizontal axis is normalized frequency, and frequency offset is maximal at $\pm f_M$, or movement directly away from and towards the receiver. The impulse in (b) indicates the Doppler shift associated with the line-of-sight ray.

However, there is a whole field dedicated to modeling unique cases of Doppler shift. Some common cases in addition to Figure 2.11 are displayed in Figure 2.12.

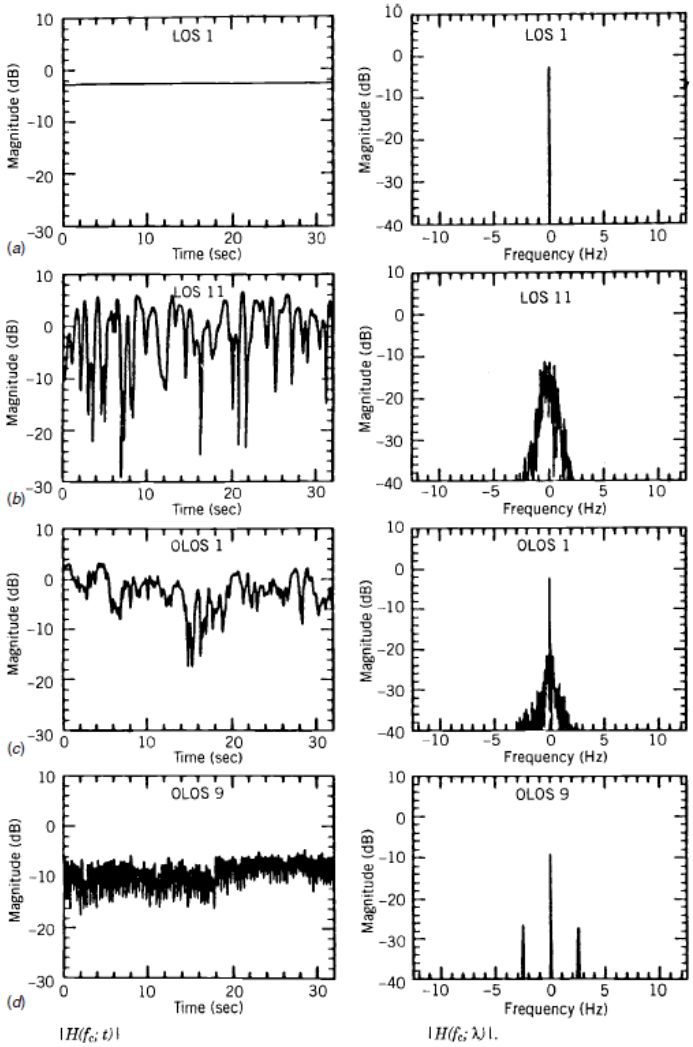


Figure 2.12: A series of impulse responses (2.19) and their Fourier transforms [1]. (a) shows a LOS experiment using a stationary radio transmit-receive pair in a stationary environment ($B_D = 0Hz$ see (2.24)). (b) display results of LOS experiment using a stationary receiver, but mobile transmitter that moved randomly within a 12 meter radius of a fixed point, simulating a mobile user pacing on their telephone ($B_D = 4.9Hz$). (c) display results of an obstructed LOS (OLOS) experiment using stationary devices 4 meters apart, but with heavy pedestrian traffic around the transmitter ($B_D = 5.7Hz$). (d) display an OLOS experiment with stationary devices, but the transmitter is rotated at a rate of 2.5 rotations per second ($B_D = 5.2Hz$).

2.1.4 Coupled Noise

test

Inter-modulation

Crosstalk

In-Band Interference

Terrestrial Interference

Industrial Noise

Cosmic Noise

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 [17]:

- 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 [17] 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 [17] as:

  $$A \sim E_p e^{-\frac{r}{r_0}}, \tag{2.26}$$

  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 2.13).

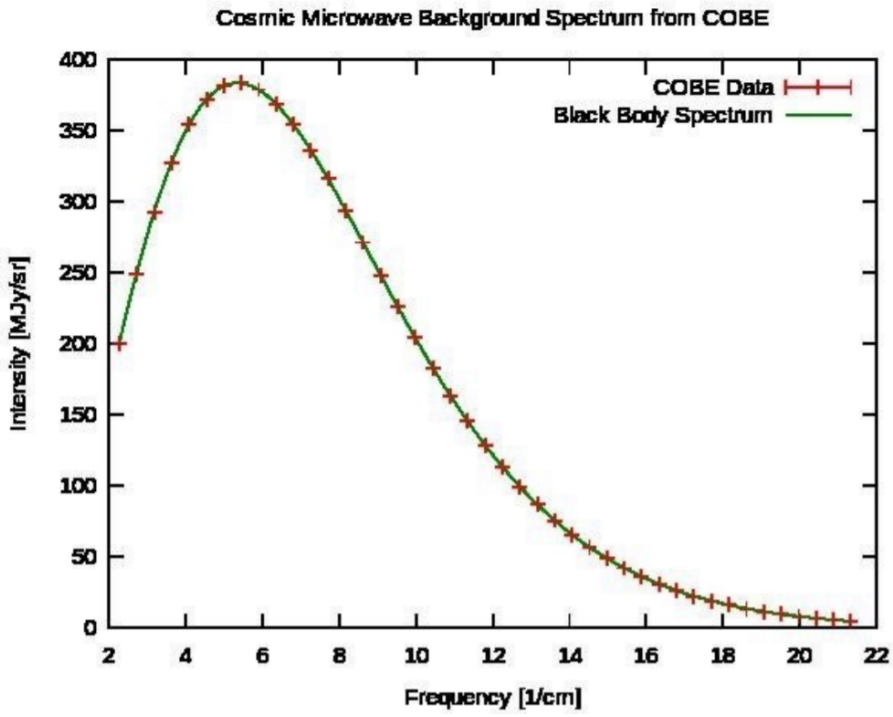


Figure 2.13: A plot [3] of the intensity of the CMBR over frequency.

2.1.5 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 2.1.5).

- 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 Local Oscillator (LO) in a RFFE drives the modulation and demodulation tasks by creating a carrier cosine wave-form. Phase noise (see Section 2.1.5) can be introduced by 1/f noise, and the frequency of the carrier can drift with time (see Section 2.1.5).
- a Mixer, which combines the carrier wave-form with the transmitted or received wave-form to form the base-band signal or the Intermediate Frequency (IF) signal.

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 2.1.5) 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 2.1.5), and in the case of pulse-shaped (see Figure 2.30) wave-forms, symbol timing offset (STO) (see Section 2.1.5) can push bit error rates (BER) to their limits, 1/M (2.76).

Furthermore, the cosine and sine paths of the RFFE (see Figure 2.20) experience phase and magnitude imbalances, resulting in stretched IQ plots (see Section 2.1.5).

Finally, various forms of electronic noise (see Section 2.1.5) can impede SNR, sometimes significantly.

Carrier Frequency Offset (CFO)

Each radio system (see Figure 2.1) has either a down-converter or an up-converter (see Figure 2.14), shifting the center frequency of the signal (see Figure 2.15) either up to the carrier frequency if transmitting or down to the base-band if receiving.

This conversion, however, is not perfect and is often affected by both a phase offset (see Section 2.1.5) and a frequency offset, the latter of which can modeled at each local oscillator as:

$$f_{o,max} = \frac{f_c \times PPM}{10^6}, \tag{2.27}$$

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

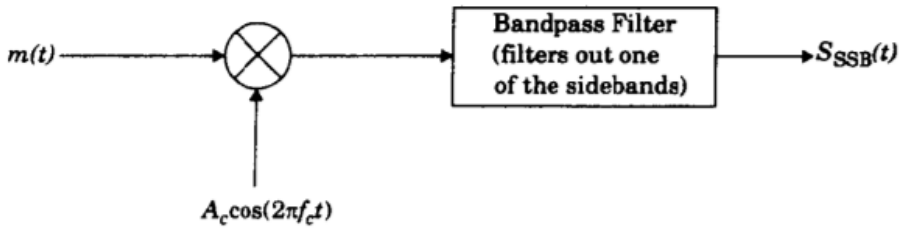


Figure 2.14: An illustration [2] of the base-band signal $m(t)$ being up-converted to the intermediate frequency f_c by a mixer, where the carrier wave-form $A_c cos(2\pi f_c t)$ is generated by a local oscillator, and in this case half of the signal's bandwidth is filtered out (see Figure 2.15 for a double side-band plots of the base-band spectrum of $m(t)$ and the up-converted spectrum of a $S_{DSB}(t)$.

the random variable no longer Gaussian. This results in a minimum possible offset of zero cycles when each offset is zero or equal and opposite, or a maximum offset of $\pm 2 \times f_{o,max}$.

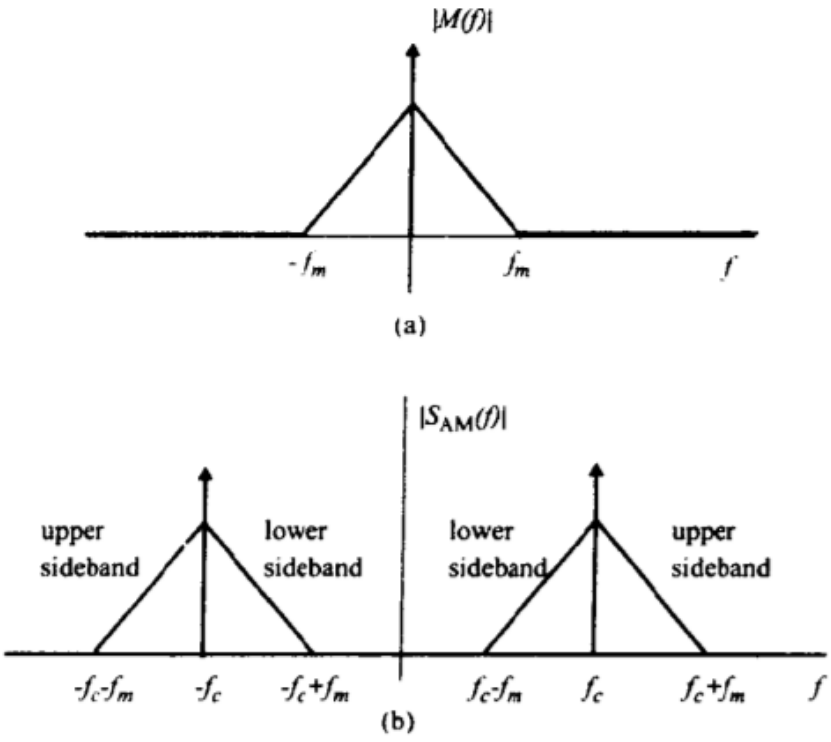


Figure 2.15: A plot [2] of (a) the base-band magnitude spectrum $|M(f)|$ of $m(t)$ (see Figure 2.14), and (b) its up-converted double side-band magnitude spectrum $|S_{AM}(f)|$.

Phase Ambiguity

As seen in Figure 2.4, 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 like Figure 2.49 can become very ambiguous, 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:

1. Equalization
2. Codewords
3. Differential Encoding

Symbol Timing Offset (STO)

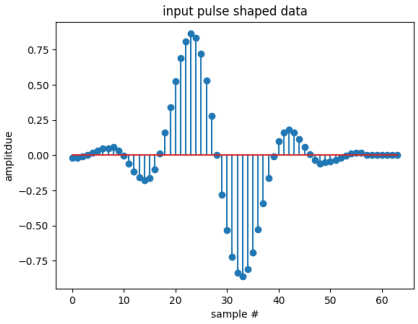


Figure 2.16

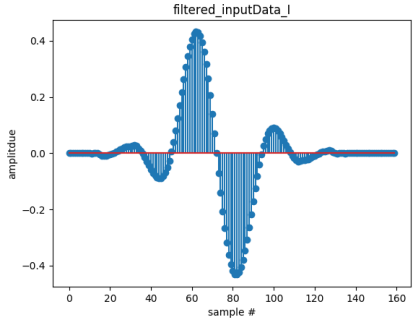


Figure 2.17

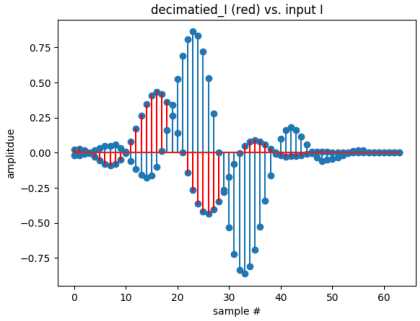


Figure 2.18

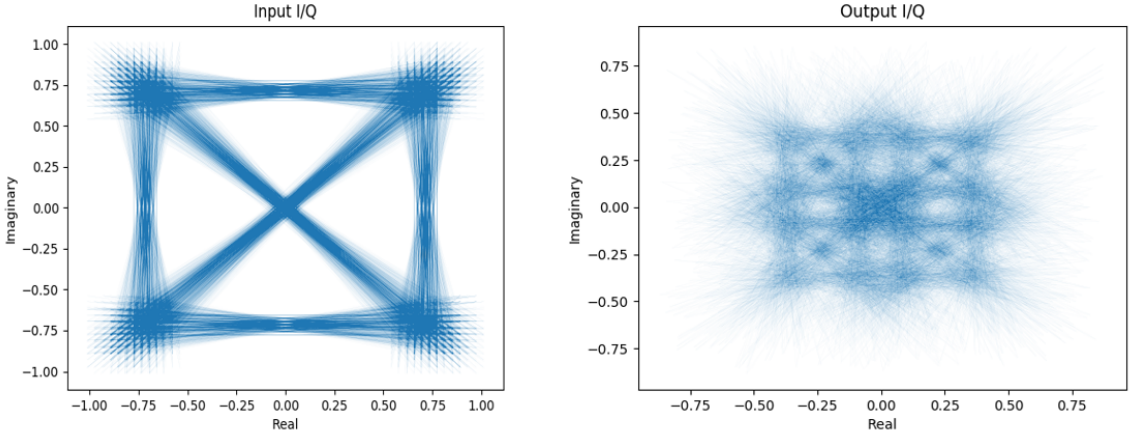


Figure 2.19

IQ Imbalance

The transmit and receive chains of a radio system (see Figure 2.1) contain a similar modulation (see Figure 2.20) 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). Quality instruments tend to keep this low, although it can vary to large amounts over frequency [18]. IQ imbalance can be modeled at mapping time for each symbol through the expression:

$$sI = kI \times sI', \tag{2.28a}$$

$$sQ = -kQsin(\phi_\epsilon) \times sI' + kQcos(\phi_\epsilon) \times sQ', \tag{2.28b}$$

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, $sI = sI', sQ = sQ'$.

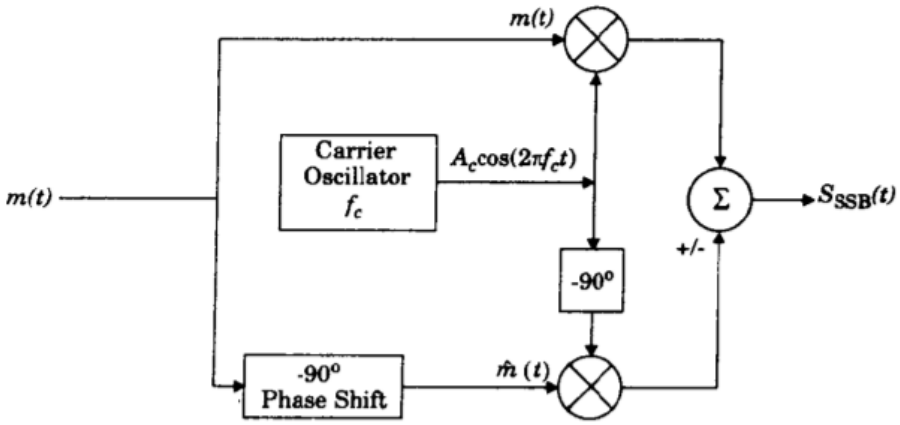


Figure 2.20: An illustration [2] of the in-phase and quadrature paths of the modulator block in a RFFE (see Figure 2.1). The signals $\dot{m}(t), m(t)$ may not experience the same gains or appropriate phases of 0 and 90 degrees (2.28).

Quantization

A radio system (see Figure 2.1) experiences numerous forms of approximations of infinite-resolution analog values [4]. Two such examples are the digital approximations of analog wave-forms by the ADC, and the floating-point approximation of IIR filter coefficients (16, 32, or 64 bit typically). While a received wave-form has an infinite-resolution amplitude value, computers only have so much computational power and must reduce the value of voltages and and filter coefficients to so many points of precision (see Figure 2.21).

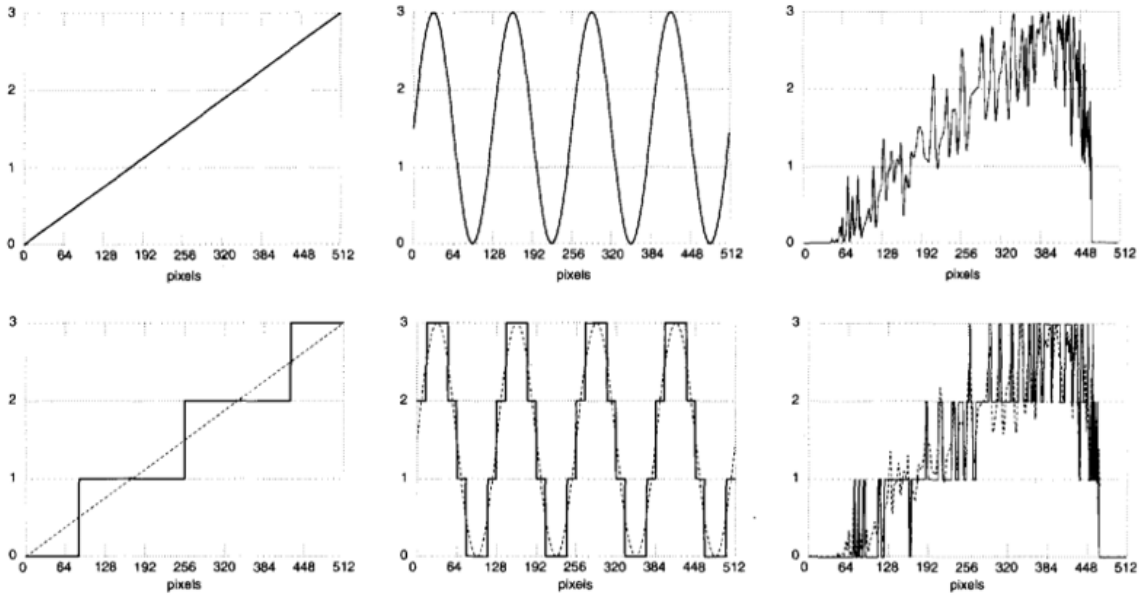


Figure 2.21: Three plots [4] describing an example of the effects of quantization error on a section of a ramp signal, (left) sine wave, (middle) and noisy wave-form (right). These plots have a digital resolution of one, all wave-forms can only be represented by a zero value, one, two, or three.

Electronic Noise

1. Thermal Noise, or Johnson-Nyquist noise, is a form of white noise (see Section 2.1.1) 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:

$$v = \sqrt{4k_B T R \Delta f}, \tag{2.29}$$

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

$$P_{dBm} = -174 + 10 \log_{10}(\Delta f), \tag{2.30}$$

assuming room temperature, $T = 298.15K$.

2. Shot Noise: 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. This phenomenon 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:

$$P = \frac{1}{2} q I \Delta f R, \tag{2.31}$$

   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.

3. 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:

$$\lambda = \sqrt{2\pi f_o 10^{L/10}}, \tag{2.32a}$$

$$\gamma_i = (i - 2.5)\frac{\gamma_{i-1}}{i-1}, \tag{2.32b}$$

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:

$$y[n] = \frac{1}{\gamma_1}(\lambda x[n] - \gamma_2 y[n-1] - ... - \gamma_i y[n-i+1]). \tag{2.33}$$

4. Burst Noise results from the summed voltage errors resulting from Gibb's phenomenon (see Figure 2.22) 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 wave-forms 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.

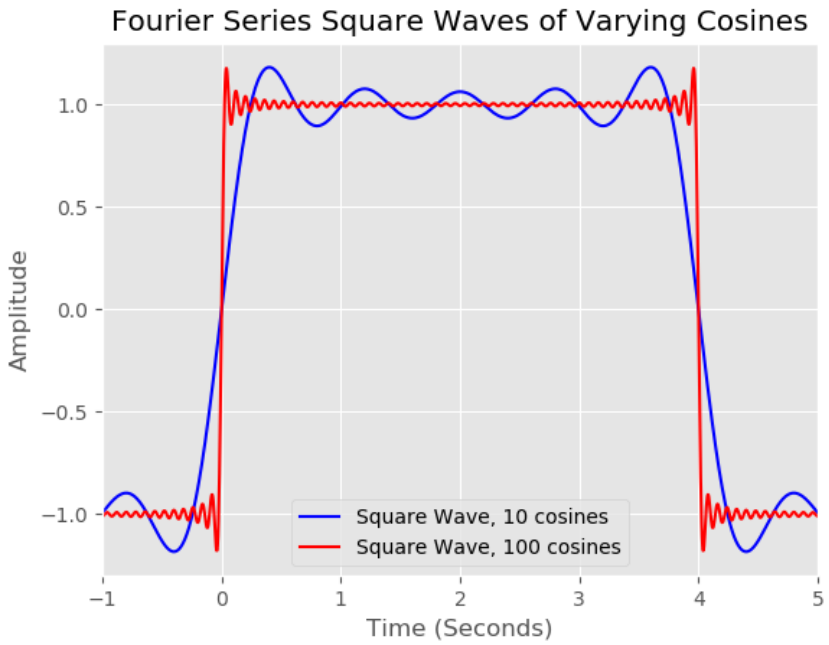


Figure 2.22: A section of a time-domain square wave-form formulated by summing cosines. While the states of the square wave-form aims to have values of negative and positive one, there are large deviations near high-definition edges, and small ripples in flat sections. If unlucky, these analog deviations can sum to mV values.

5. Transit-Time Noise is a high frequency noise that occurs in transistors. Transistors are gate-like semi-conductors used to many effects 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.

2.2 Training Deep Learning

In order to better understand what makes deep learning training data valuable, this section communicates the basic concepts of linear classifiers and convolutional neural network classifiers. Knowing how the data is used is important in knowing what good data is.

2.2.1 Linear Classification

A linear classifier is the simplest form of machine learning that classifies signals that uses weights instead of storing all training signals in memory, like the nearest-neighbor or K-nearest-neighbor classifier. 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 Neuron Analogy

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 (see Figure 2.23), and its mathematical model (see Figure 2.24) is very rough. In reality, dendrites perform non-linear, time-varying operations on signals coming in from axons [5] 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.

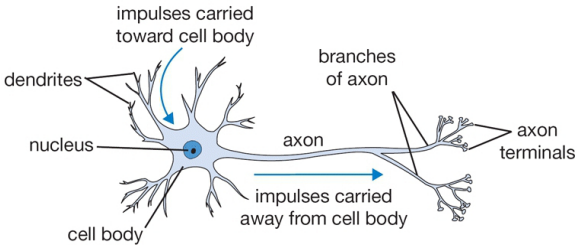


Figure 2.23: An illustration [5] of a biological neuron. A neuron cell is composed of a nucleus which receives signals from many dendrites. The amount of influence a dendrite has on a neuron is determined by synapses. When the sum of incoming signals is above a threshold, the nucleus fires a signal down its axon, which in turn splits into many dendrites, feeding into other neurons.

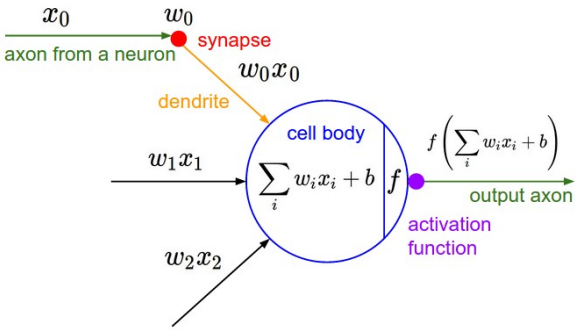


Figure 2.24: A mathematical representation [5] of Figure 2.23. The previous neurons' axons carry the signals x_0, x_1, x_2, which split into many dendrites. Synapses influence that value by a weight, (w_0, w_1, w_2). The cell body adds weights to each incoming dendrite (b_0, b_1, b_2), computes the dot 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.

Multi-class Support Vector Machine (SVM)

The 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:

$$L_i = \sum_{j \neq y_i} max(0, S_j - S_{y_i} + \Delta), \tag{2.34}$$

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 $S = [13, -7, 11]$,

$$L_i = max(0, S_1 - S_0 + \Delta) + max(0, S_2 - S_0 + \Delta), \tag{2.35a}$$

$$L_i = max(0, -7 - 13 + 10) + max(0, 11 - 13 + 10), \tag{2.35b}$$

$$L_i = max(0, -10), max(0, 8), \tag{2.35c}$$

$$L_i = 8. \tag{2.35d}$$

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. This is done by adding a unit value to the end of each flattened row vector signal x_i as shown in Figure 2.25.

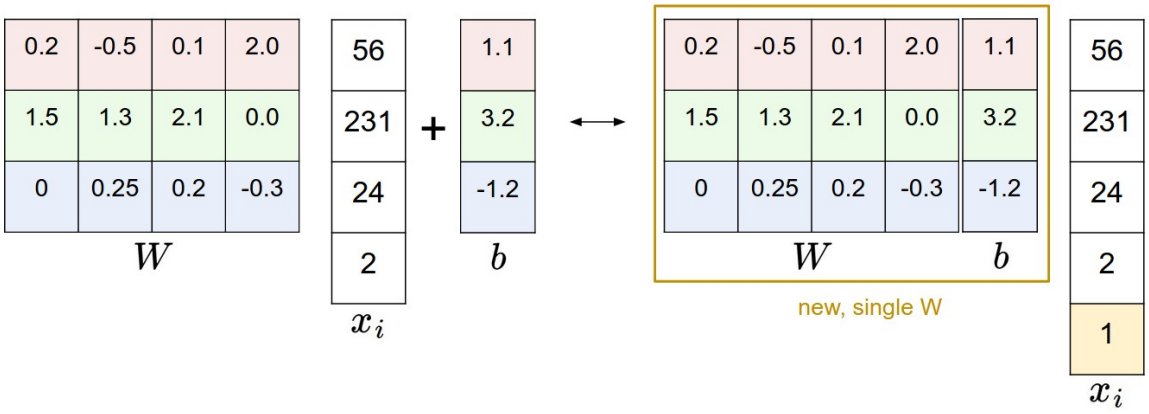


Figure 2.25: An illustration [5] of reducing $W \in [3, 4]$ and $b \in [3, 1]$ into a single matrix, $W \in [3, 5]$ by adding a unit value to the end of $x_i \in [5, 1]$.

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:

$$L_i = \sum_{j \neq y_i} max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta), \tag{2.36}$$

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 (2.35) we achieve $L_i = 0$. If we set all weights $W = \lambda W$, for $\lambda = 3$, through (2.36) we would achieve in (2.35) $L_i = max(0, (-7 - 13)\lambda + 10) + max(0, (-5 - 13)\lambda + 10) = 0$. 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, defined as:

$$R(w) = \sum_k \sum_l w_{k,l}^2, \tag{2.37}$$

such that the SVM is now defined as:

$$L_i = \sum_{j \neq y_i} max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta) + \lambda \sum_k \sum_l w_{k,l}^2. \tag{2.38}$$

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 [5], 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 next in Section 2.2.1: Validation.

Soft-Max Classifier

The other popular loss function, besides SVM, is the soft-max classifier where the soft-max function is defined as $f_j(z)$ and its loss function L_i:

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}, \tag{2.39a}$$

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right). \tag{2.39b}$$

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 (2.25):

$$P(y_i, x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}. \tag{2.40}$$

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 2.26. In practice, the performance of both loss functions is near equal [5]. However, one can train faster or classify more accurately than the other, given the right circumstances. Since the SVM is happy 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.

Validation

In the last two sections, 2.2.1: SVM and 2.2.1: Soft-Max Classifier, it was mentioned that λ, Δ are defined as hyper-parameters. Throughout Section 2.2 this term will be used to describe constants used in various machine learning tasks that might not have clear values. The way researchers typically tune hyper-parameters is through a data-driven approach known as validation [5].

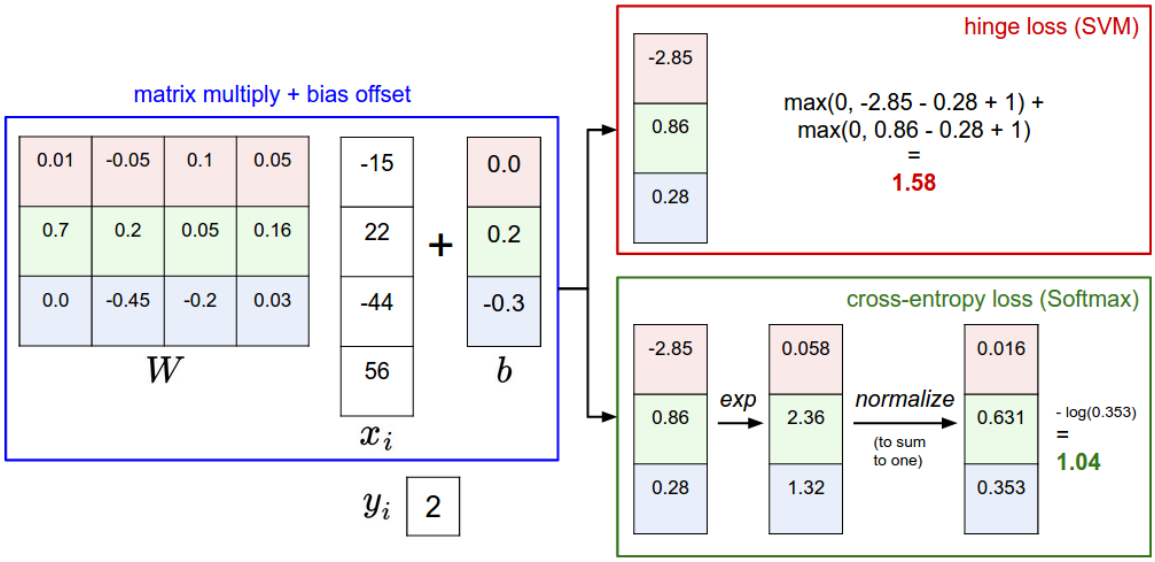


Figure 2.26: An illustration [5] of the computation of class cores f and the resulting loss score L_i using both SVM and soft-max functions. Both use the same class scores f, but have very different interpretations of their results, 1.58 and 1.04. The SVM considers each incorrect score less than a margin 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.

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 2.27). 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.

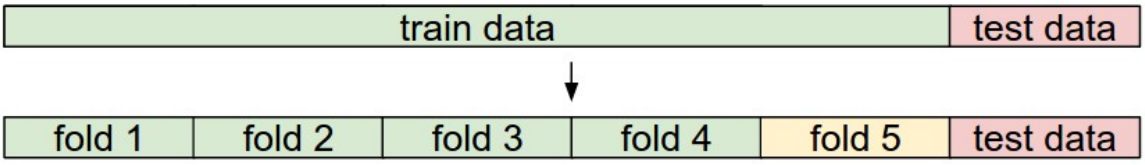


Figure 2.27: An illustration [5] of common data splits between training, validation, and testing data. In this image, validation is performed on fold 5, training on folds 1-4, and testing on the rest. Next, fold 1 would be used as validation data, and folds 2-5 as training data, and so on, until all 5 folds have been used as the validation data.

Stochastic Gradient Descent (SGD)

In the Section 2.2: SVM we discussed how the loss function evaluates the quality of the current weights used in a linear classifier. Using that information how do we improve our weights? Consider the example loss function in Figure 2.28, where dark blue represents low loss and red high loss. The axis' 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 powerfully reduce the loss function used.

In practice, due to inaccuracies introduced by numerically calculating the gradient, the centered difference formula:

$$\frac{\delta f}{\delta x} = [f(x+h) - f(x-h)]/2h, \tag{2.41}$$

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

$$\nabla f(w_0, w_1, w_2) = \frac{\delta f}{\delta w_0}\hat{\boldsymbol{w}}_\mathbf{0} + \frac{\delta f}{\delta w_1}\hat{\boldsymbol{w}}_\mathbf{1} + \frac{\delta f}{\delta w_2}\hat{\boldsymbol{w}}_\mathbf{2}, \tag{2.42}$$

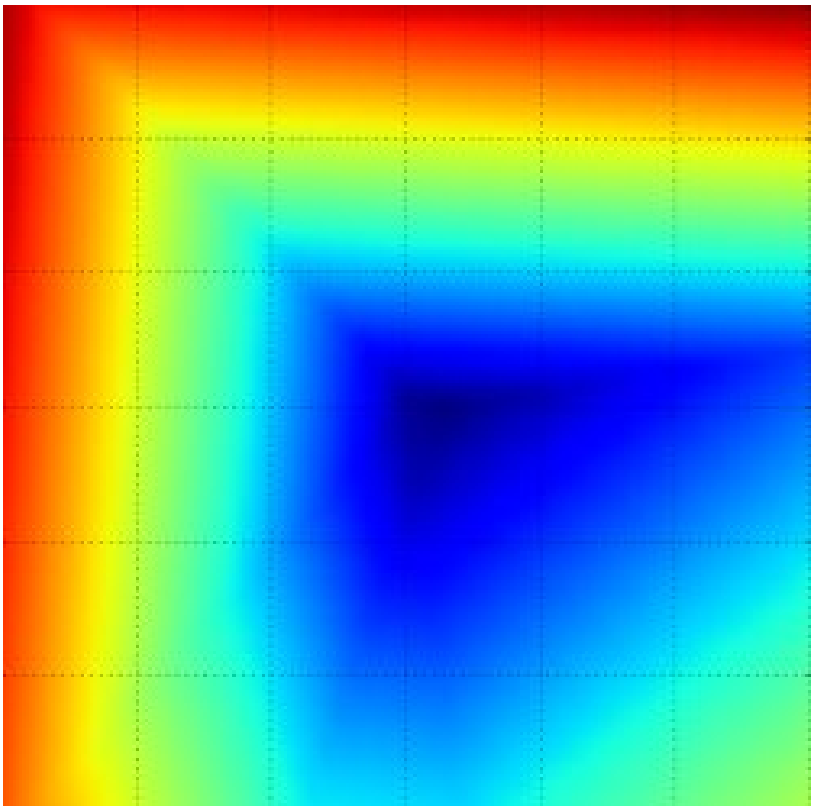


Figure 2.28: An example SVM loss function [5] plotted against two weights. Red represents high loss, blue low loss. Each of the two axis represent values assigned to a weight. In practice, loss functions have pockets of local minima/maxima, and cannot be visualized due to the number of dimensions required to represent each weight used.

where w_0, w_1, w_2 are weights used and $\hat{\boldsymbol{w_0}}, \hat{\boldsymbol{w_1}}, \hat{\boldsymbol{w_2}}$ are the weights' unit vectors. Once the descent vector has been determined, though, how does one determine how far in that direction to update the weights? Consider the consequences of updating by too much as in Figure 2.29, or updating by too little and getting stuck in a local minimum of the loss function. It turns out there are many approaches to weight updates, which are presented in Section 2.2.2.

To generalize in the extreme, a linear classifier might find that the weights corresponding to samples surrounding the max and min samples of Figure 2.30 should be increased to minimize its loss function. Giving those weights too much influence, however, may cause

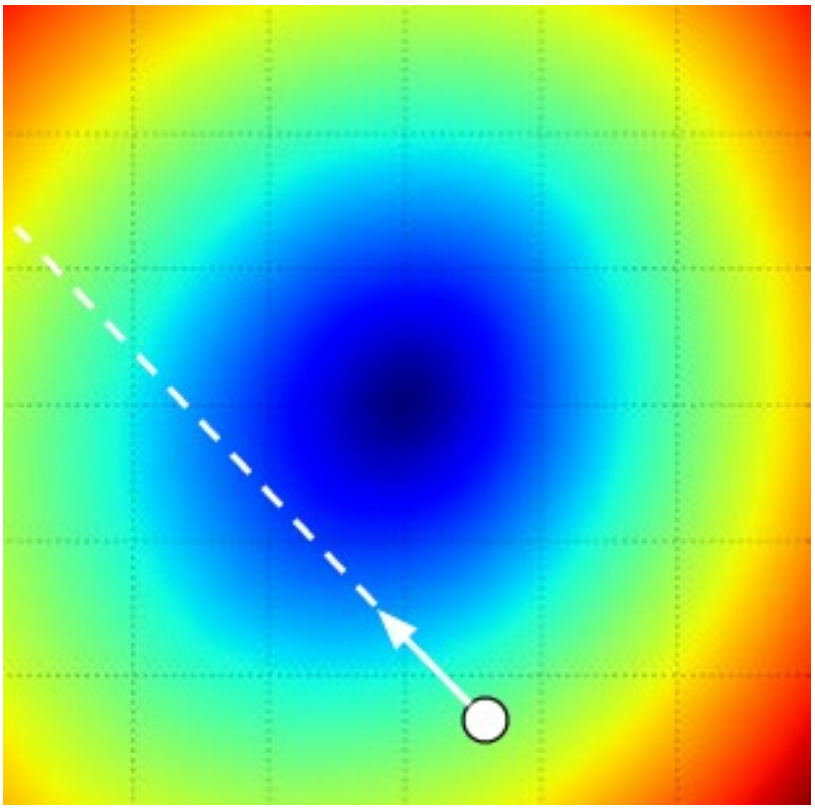


Figure 2.29: An illustration [5] of an SVM loss function's gradient vector (2.42) for a two-weight linear classifier. Red represents high loss, blue low loss. The white circle represents the current values chosen for weights w_o, w_1, the white arrow the gradients unit vector, and the dashed line an extension of that gradient. Updating the weights by too much will put the weights in perhaps a higher loss section of the graph, but updating by too little will be computationally expensive and perhaps get the SGD stuck in a local minimum of the SVM loss function.

the classifier to ignore imperfections or phenomenon 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.

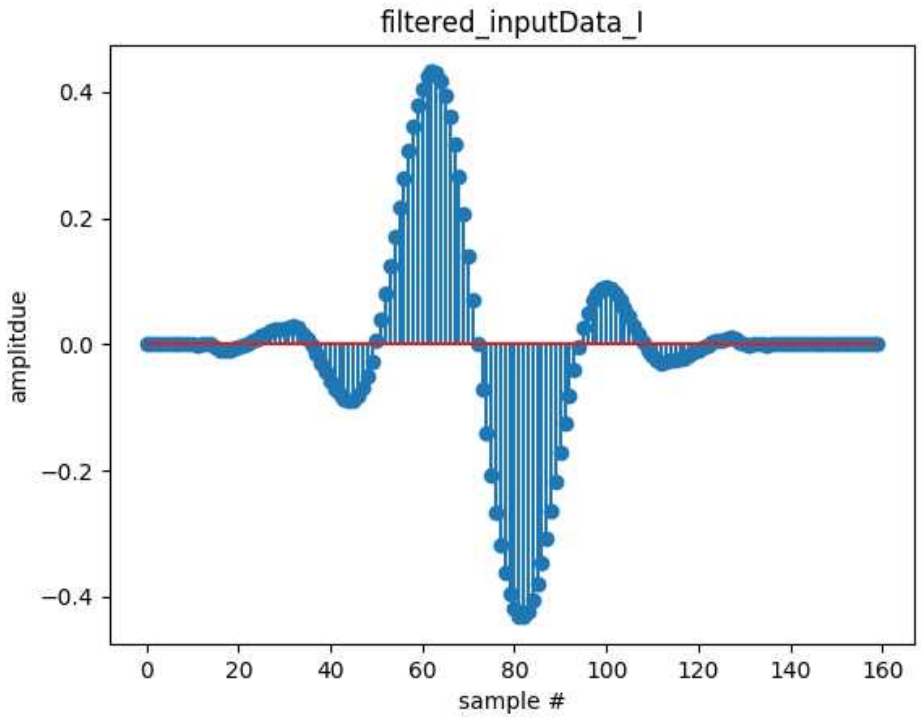


Figure 2.30: The in-phase components of one positive and one negative QPSK symbol, up-sampled to 16 SPS by a Raised-Root Cosine (RRC) filter with a roll-off coefficient of 0.35. Making up half the values of an example flattened training signal vector x_i, classification decisions of a linear classifier 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.

Parameter Updates

As mentioned in Section 2.2.1: SGD, the gradient is just a vector, and doesn't determine to what magnitude weight updates should occur. As shown in Figure 2.29, there are consequences to updating by too much or too little. How should we decide the method and amount by which to update?

- Vanilla updates, where the weights W are changed after each back pass of SGD

according to:

$$W+ = LR\nabla W, \tag{2.43}$$

where the hyper-parameter LR is the learning rate or step size, and ∇W is the current matrix of gradient values for the most recent back pass. Typically SGD parameter updates are halted by processes parameterized by patience and convergence values. SGD will halt when the classifier's loss function converges to within a certain percentage difference of the last back pass, after a patience period has passed. The purpose of the patience period is to give the SGD a chance to escape a local minimum, should it be in one, through natural noise in some weight parameter update routines.

- Momentum updates almost always result in faster convergence, 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.

$$v_{(i)} = \mu v_{(i-1)} - LR\nabla W \tag{2.44a}$$

$$W+ = v_{(i)} \tag{2.44b}$$

where the velocity is initialized at zero as $v_{-1} = 0$, and momentum μ is a tunable hyper-parameter, typically 0.9 in practice [5], $v_{(i)}$ is the current velocity value, and $v_{(i-1)}$ is the previous back pass velocity value.

- Nesterov Momentum [19] 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:

$$v_{(i)} = \mu v_{(i-1)} - LR\nabla W, \tag{2.45a}$$

$$W+ = (1+\mu)v_{(i)} - \mu v_{(i-1)}, \tag{2.45b}$$

- Adagrad [20] cached updates, where the updates are defined as:

$$cache = (\nabla W)^2 \tag{2.46a}$$

$$W+ = -\frac{LR\nabla W}{cache + h}. \tag{2.46b}$$

where $h = 0^+$ to avoid divide by zero errors. While adagrad begins fast, the aggressive update term $\frac{LR\nabla W}{cache+h}$ often stops learning too early [5] without proper annealing (covered later in this section).

- Adam [21] is a recent and recommended [5] parameter update algorithm that performs better than the other parameter updates defined in this paper. Adam updates are defined as:

$$m = beta_1 m + (1 - beta_1)\nabla W, \tag{2.47a}$$

$$mt = \frac{m}{1 - beta_1^t}, \tag{2.47b}$$

$$v = beta_2 v + (1 - beta_2)(\nabla W)^2, \tag{2.47c}$$

$$vt = \frac{v}{1 - beta_2^t}, \tag{2.47d}$$

$$W+ = \frac{-LR \times mt}{\sqrt{vt} + h}. \tag{2.47e}$$

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 [20] to avoid divide by zero errors. The constants $beta_1, beta_2$ are tunable hyper-parameters.

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

- Step decay $\alpha = c \in [0, 1]$, reduces LR by a factor every few back passes. This decay is piece-wise. A good way to tell when and how much by which to reduce LR is to look at the loss at each back pass. If the loss is not reducing, that may be a sign that your 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.

- Exponential decay is continuous, not piece-wise, and decays $LR_{(t)} = LR - \alpha$ more rapidly, where for back pass t, $\alpha = \alpha_0 e^{-kt}$. The values α_0 and k are hyper-parameters.
- Inverse decay which decays LR more slowly than exponential decay but faster than step decay, defined by $\alpha = \alpha_0/(1 + kt)$.

Back-Propagation

Computing gradients in a linear classifier can be thought of as flowing backwards, or back-propagating, through the neurons. Since neurons (see Figure 2.24) mostly interact with their inputs through multiplication and addition, circuit graphics (see Figure 2.31) are often used to visualize SGD. Beginning with a forward pass, consider the weights $x =$

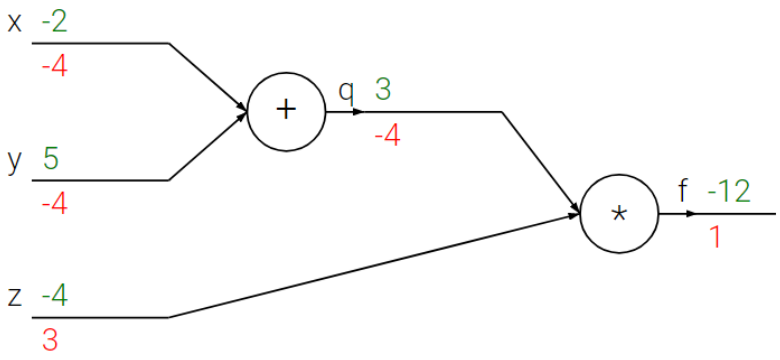


Figure 2.31: A circuit model [5] showing the forward pass (green) by applying inputs to the gates operators and backward pass (red) by applying the chain rule recursively. Gates represent a few local operations done by a linear classifier's neurons. Gates can do both passes totally independent of other gates, without knowledge of the full circuit, or classifier structure.

$-2, y = 5, z = -4$ and states $q = x + y = 5 - 2 = 3$ and $f = z \times q = -4 \times 3 = -12$. Using (2.42), 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.

Weight Initialization

In the previous section, forward passes through circuit models such as Figure 2.32 were done using weight values in green. This section aims to answer the question: what values should we use on our 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 (2.47). 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 2.24):

$$\text{Var}(s) = \text{Var}(\sum_i^n w_i x_i), \tag{2.48a}$$

$$= \sum_i^n \text{Var}(w_i x_i), \tag{2.48b}$$

$$= \sum_i^n [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i)\text{Var}(w_i). \tag{2.48c}$$

Consequently, in [5, 22] it is suggested for ReLU (2.51) classifiers to initialize each neurons' weights as $W = norms \times \sqrt{2/n}$ for $norms \sim \mathcal{N}(0,1) \in n$ where n is the number of weights in that neuron. For zero-mean activation function classifiers such as Tanh (2.50), $W = \frac{norms}{\sqrt{n}}$ is suggested [5].

The Activation Function

While addition and multiplication gates cover both operations involved in a dot product, Figure 2.24 also portrays the use of an activation function. We will now present five categories of common activation functions:

- The sigmoid non-linear function is defined as:

$$\sigma(x) = 1/(1 + e^{-x}), \tag{2.49}$$

which was historically popular because it serves as a good metaphor for the firing of a neuron, having outputs ranging from not firing at $\sigma(x) = 0$ to saturated firing at max frequency, $\sigma(x) = 1$. A significant issue with (2.49) is that during back passes of SGD, saturated regions produce a gradient of near zero using (2.41). This keeps all but the largest of signals from flowing through neurons. Another issue is that (2.41) 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 2.29).

- the Tanh non-linear function solves the all positive issue in (2.49) by zero-centering the equation.

$$\tanh(x) = 2\sigma(2x) - 1. \tag{2.50}$$

  However, (2.50) still saturates to -1 and 1, resulting in gradient-killing back passes. With the improvement of zero-centering, (2.50) is always preferred to (2.49).

- A very common activation function is the Rectified Linear Unit (ReLU) function:

$$f(x) = \max(0, x). \tag{2.51}$$

  The reasons for ReLU's popularity are that it was found to make for very fast [23] weight convergence through SGD, and is computationally cheap. However, one should take note that the gradient (2.42) of (2.51) is computed as:

$$\frac{\delta f}{\delta x}(y) = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases} \tag{2.52}$$

  Notice how this gate routes all the gradient to the larger input. The implications of this can be seen in Figure 2.32.

- Leaky ReLU attempts to fix the blackout issue that (2.51) has by giving a small negative slope α for values $x < 0$.

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases} \tag{2.53}$$

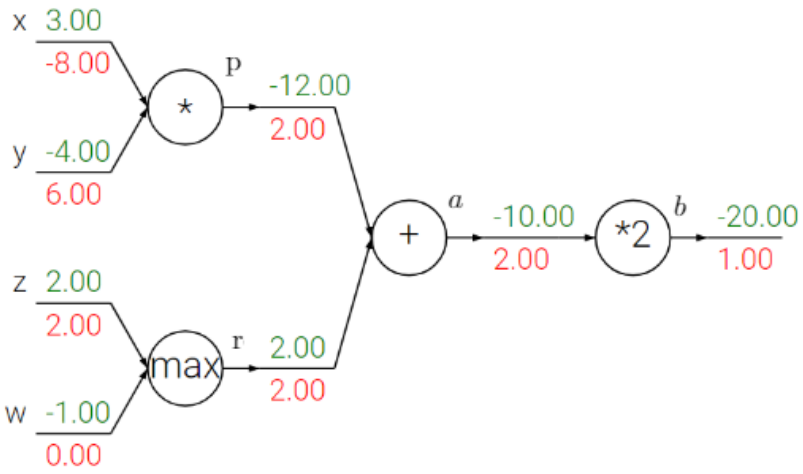


Figure 2.32: A circuit model [5] showing the forward pass (green) by applying inputs to the gates operators and backward pass (red) by applying the chain rule recursively of a circuit featuring a ReLU max gate. The blacked out w weight would cause all gates before it to have a gradient of zero, killing those neurons. Using (2.42), the back pass value for w can be shown to be $w = 1 \times \frac{\delta}{\delta a} 2a \times \frac{\delta}{\delta r}(r + p) \times \frac{\delta}{\delta w} \max(w, z) = 1 \times 2 \times 1 \times 0 = 0$

> While this sometimes prevents blackouts, other times it doesn't, and many are unsure why it works when it does and why it doesn't when it doesn't. The leaky slope α can be parametrized and tuned to optimize a classifier (called PReLU) as seen in [24], although the reason why this is of benefit is still unclear.

Most researchers use (2.51) in practice, and if a large number of neurons are dying during SGD, leaky ReLU and PReLU are often used.

Data Pre-Processing

Signal data doesn't always come in a form that is friendly for machine learning. Sometimes one or several operations must be performed on IQ data to prevent certain errors from occurring in training or testing a linear classifier. As mentioned in (2.50), 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 pre-processing that are used and

why they are important.

- Mean subtracting IQ data is the most common form of data pre-processing [5], which helps SGD avoid having back pass steps with all negative or all positive gradient updates. 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).

- Normalization is necessary if different input signals are expected to have different scales, units, or amplitudes, but are equally important. Many signals have different amplitude constants for the modulation schemes they use, and 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 [25] (PCA), which reduces the dataset X to the p highest variance signals x_i. This is meant to save time, and classifiers trained with PCA-treated data have been shown to perform well. PCA-reduced data X_{PCA} is 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}:

$$x_{i_{ZC}} = x_i - \frac{\sum x_i}{k \times D}, x_i \in X, x_{i_{ZC}} \in X_{ZC}, \tag{2.54a}$$

$$cov(X_{ZC}) = \frac{X_{ZC}^T \cdot X_{ZC}}{N}, \tag{2.54b}$$

$$USV^* = cov(X_{ZC}), \tag{2.54c}$$

$$X_{PCA} = X_{ZC} \cdot U_p, U_p \in [N, p]. \tag{2.54d}$$

  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.

2.2.2 Designing a Neural Network Architecture

In Section 2.2.1 we've 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, we will discuss what makes one classifier different from another, why neural networks can approximate any function, and what it means to train too much.

Neural Network Organization

In linear classifiers, each layer of the architecture is fully-connected, or each neuron's output is 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, 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 2.33, 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 resultant number of parameters is $w + b = 32 + 9 = 41$. Many modern architectures contain 100s of millions of parameters and 10s of layers [5].

Neural Networks: a Universal Approximator

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 [26].

This is an enormous claim, and any skeptic should reasonably meet it with caution. Consider Figure 2.34, a 2-layer NN with two neurons in its hidden layer. For large w, small b, it is shown in [6] that the output can be shaped into a step function, centered where ever desired.

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

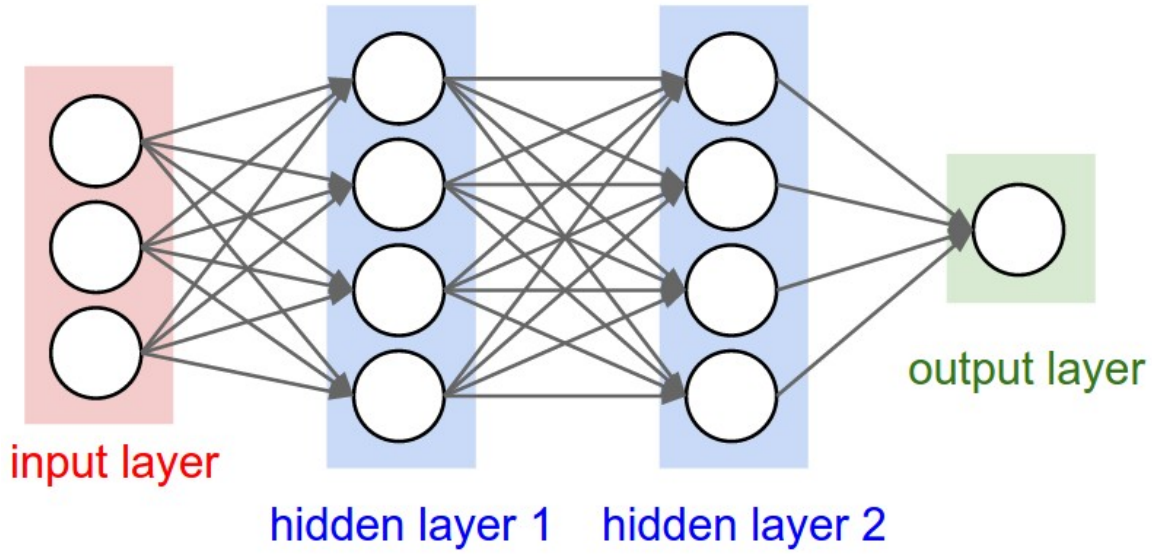


Figure 2.33: A three-layer [5] neural network with two fully-connected hidden layers. Each hidden layer has four neurons, and the input has three samples. As shown in Section 2.2.3, not all architectures use fully connected layers, and for good reason.

location and height. More formally, as shown in [26], 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 (2.50) 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 [5]. For fully connected neural networks, it has been found that classification accuracy is rarely increased for architectures beyond two hidden layers [5]. In Section 2.2.3, we will show this is not at all the case for Convolutional Neural Networks, which often benefit from having tens of hidden layers.

2.2.3 Convolutional Neural Networks

In Section 2.2.1, 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.

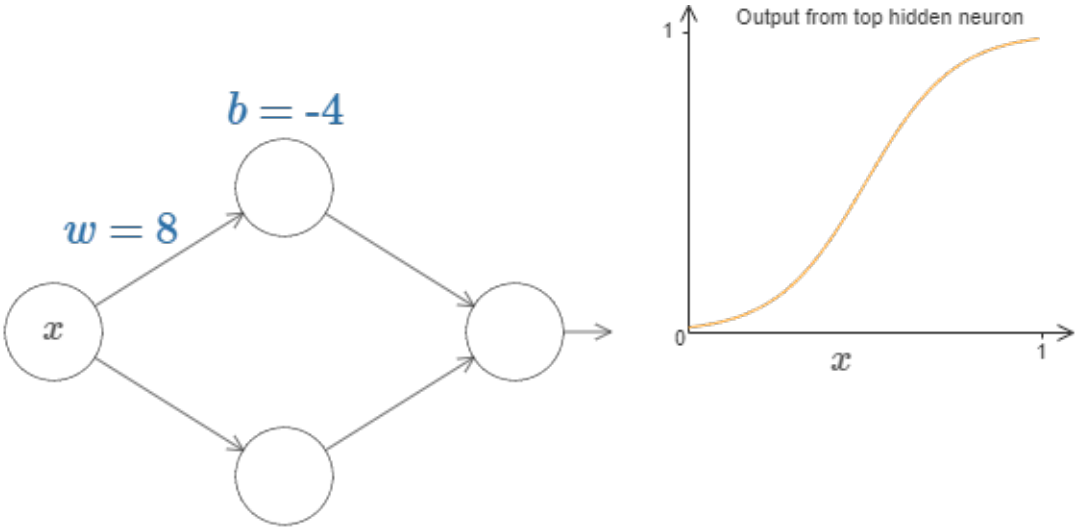


Figure 2.34: A diagram from [6]. Focusing on the top neuron, the output (shown) is computed as the sigmoid (2.49) of the dot product (see Figure 2.24) of all its inputs, in this case the one input. w shapes the transient part of the output, while b places it. The function saturates at zero and one.

This is not the case with CNNs [5], which results in tens of hidden layers. For all but the smallest signals, this translates to billions or trillions of parameters (see Figure 2.33). Fully connecting the neurons of each hidden layer would be hugely computationally expensive, and more importantly would result in significant amounts of over-fitting [5]. 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 2.36), convolutional layers in place of hidden layers, and a troupe of regularization layers that separate each convolutional layer.

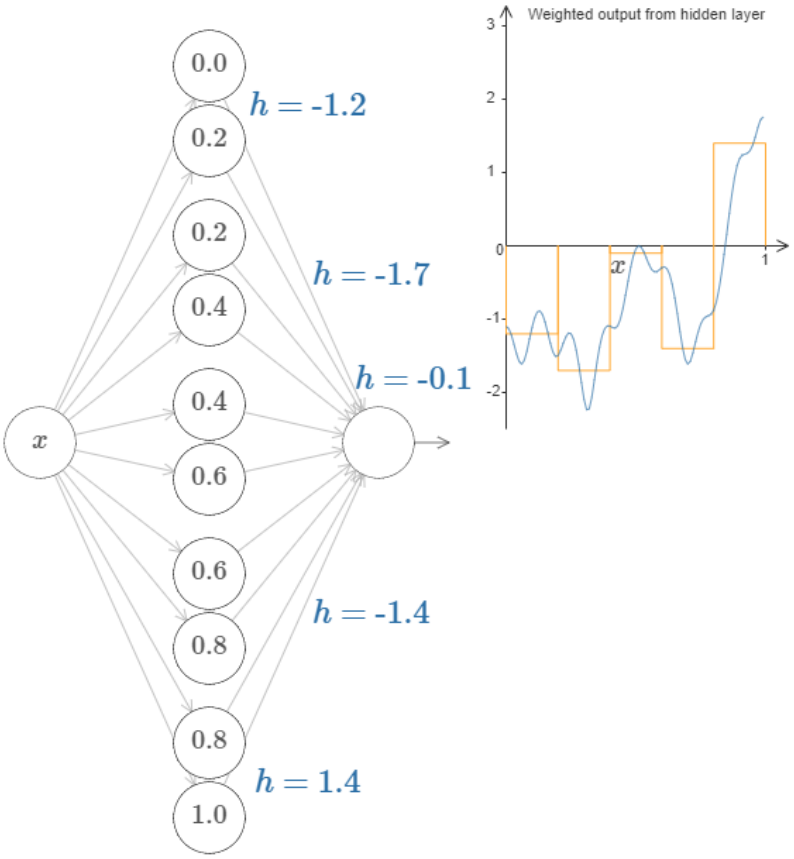


Figure 2.35: A diagram from [6]. With each additional neuron in the hidden layer, the sum of sigmoids at the neuron in the subsequent layer is a closer approximation of our cartoon of a complex function.

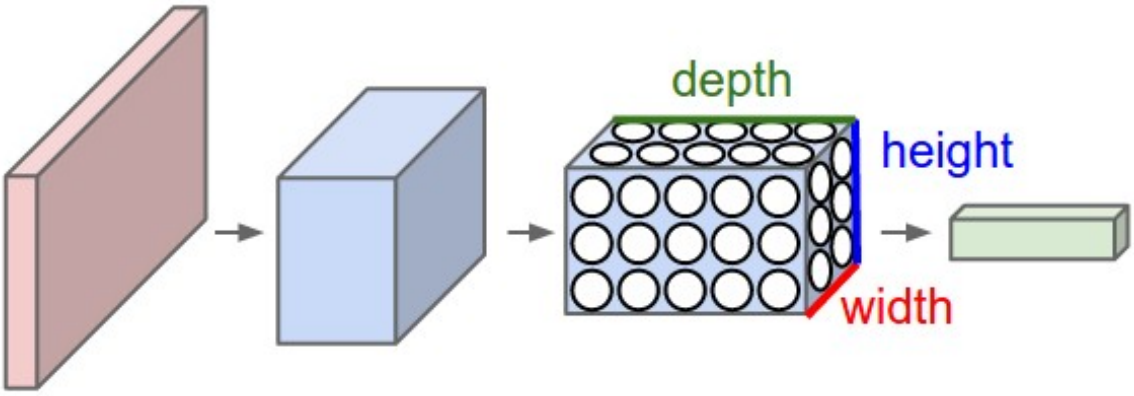


Figure 2.36: A comparable CNN [5] to the linear classifier in Figure 2.33. Convolutional layers are three-dimensional, and only the last few layers are fully connected. The rest of the CNN is much more sparsely connected in an effort to reduce over-fitting and computational cost.

The Third Dimension

In Section 2.2.1, 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 2.23). 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:

$$out\_dim = (W - F + 2P)/S + 1, \tag{2.55}$$

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 2.37 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 (2.51) 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 downsampling, 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 [5] to reduce over-fitting while maintaining the key feature information of signals.

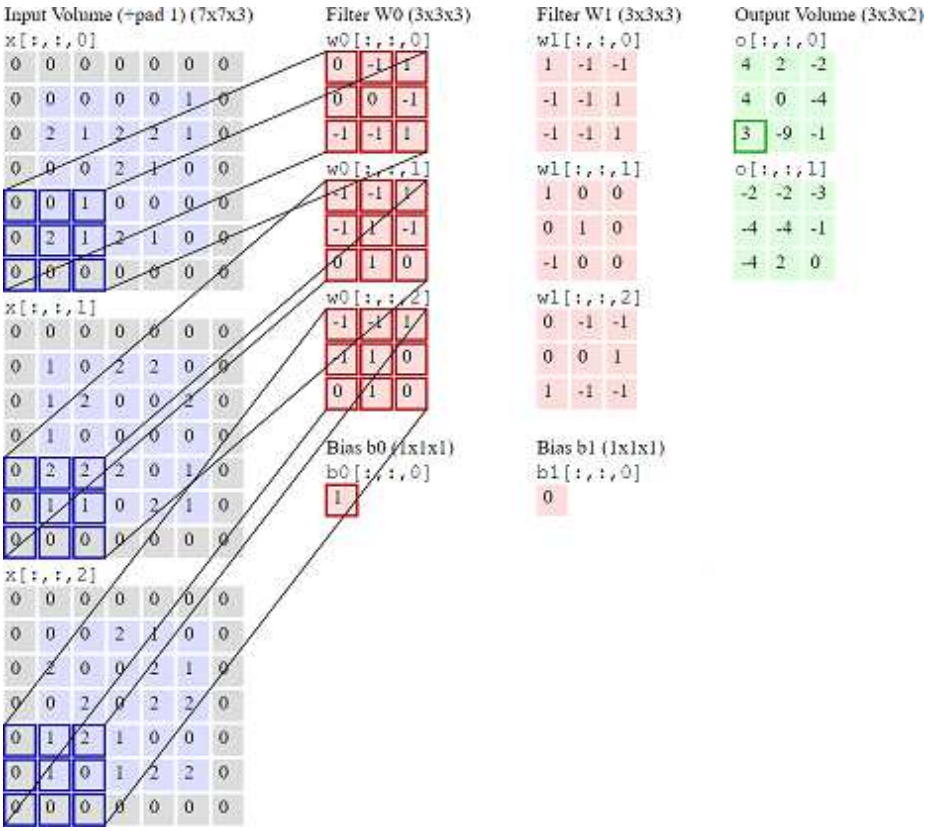


Figure 2.37: Two convolution computations [5] applied to an input (blue) of size $W = 5$, filters (red) of size $F = 3$, zero-padding (gray) $P = 1$, and stride $S = 2$. Through (2.55), we obtain the output matrix (green) height/width $(5 - 3 + 2 \times 1)/2 + 1 = 3$ of depth two due to using two filters for an output shape $\in [3, 3, 2]$. The value 3 is computed as the sum of all highlighted convolutions $x \circledast w0$, plus the bias $b0$.

It has been found in practice that repeating sequences of convolutional and down-sampling layers can extract higher dimension features in signals and images [5]. CNN layer sequences are typically terminated with a few fully connected dense layers whose elements correspond to class scores. Consider the ConvNet CNN architecture in Figure 2.39.

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 activation layers, as well as how many sequences of those placements to use. Additionally, each process is dictated by potentially dozens of hyper-parameters.

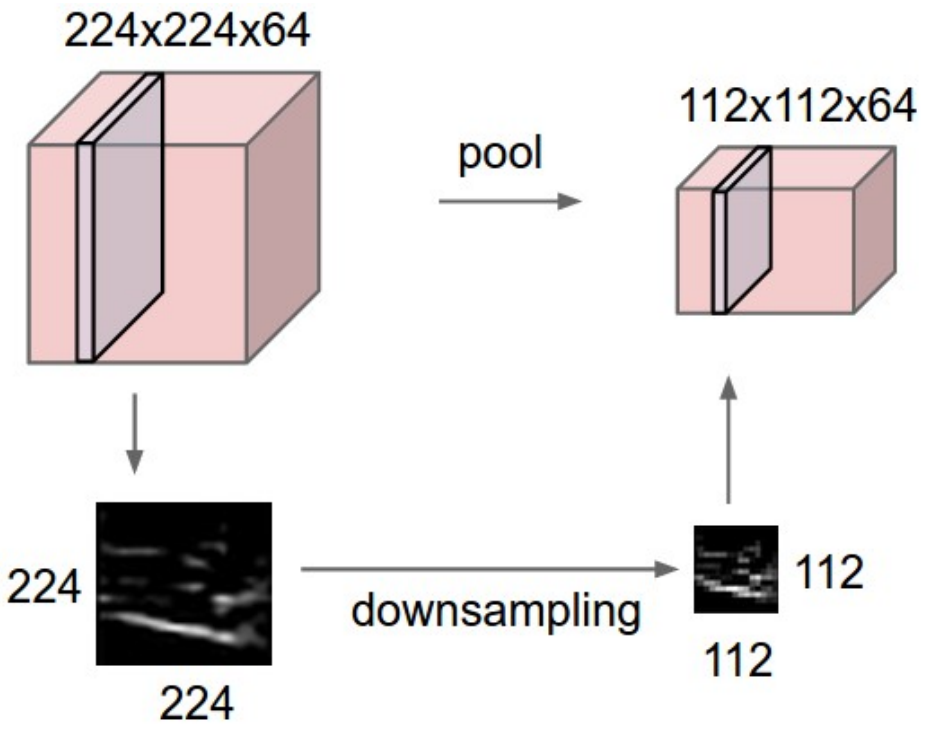


Figure 2.38: Max pooling [5] of a 244 by 244 pixel image. Input size W is 224, filter size F is two, stride S is two for an output shape (2.55) of $(224 - 2 + 2 \times 0)/2 + 1 = 112$. Depth is maintained.

Below is a brief list of powerful, popular, and recent CNN architectures:

- LeNet [27] is the first successful CNN architecture, used to read zip codes and digits.
- AlexNet [28] 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 [29] 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 [30] was designed in 2014 by a group of staff from Google. The architecture made use of an Inception Module, reducing the number of parameters by an

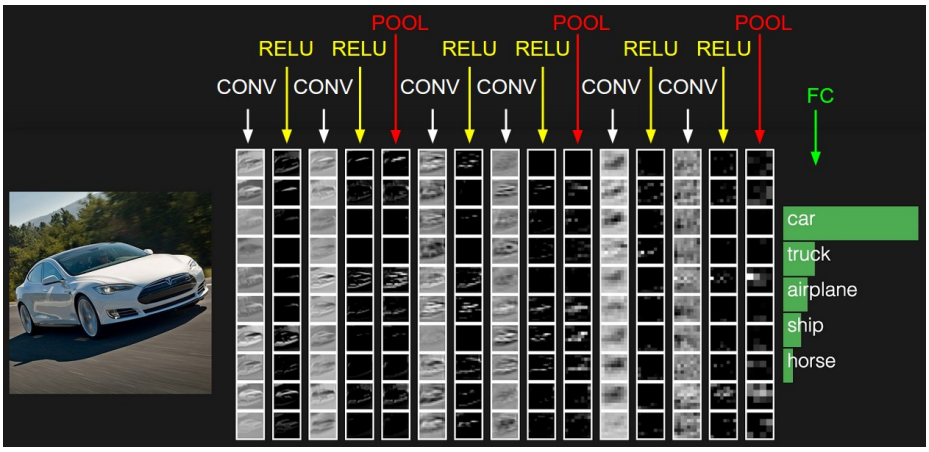


Figure 2.39: A ConvNet architecture [5] that takes raw image pixel values as input, and outputs a five-element fully-connected layer, where each value corresponds to the CNN's belief that the raw image belongs to a label. In this case, the image is likely a car.

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 [31] 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 [32] was presented in 2015, featuring skip connections, batch normalization, and a complete lack of fully-connected layers. Skip connections helped SGD training (2.42) avoid gradient saturation, while batch normalization and a lack of fully-connected layers reduced training complexity.

Regularization

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 adds the term $R(w) = \frac{1}{2}\lambda w^2$ to the loss function (2.36). 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 2.2.1: 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.

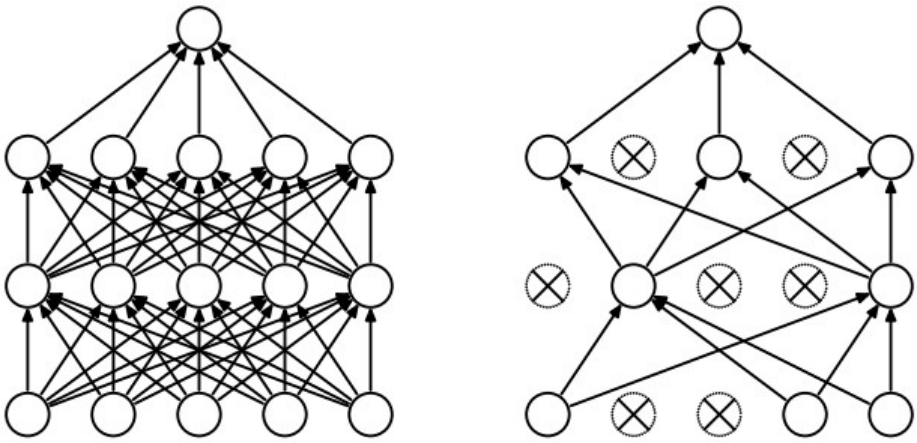
- Dropout [33] is the most effective and simple regularization method [5]. Dropout sets weights that connect any two layers to zero during training with probability p.

2.2.4 Novel Training Methods: Bayesian Optimization

Bayesian optimization [7] aims to find the global maximizer x^* of the unknown objective function f such that:

$$x^* = \arg\max_{x \in \chi} f(x), \tag{2.56}$$

where χ is the design space. Bayesian optimization has a bigger scope than neural networks. A CNN (see Section 2.2.3) can be represented by the tunable parameters (weights) $W = x$ and the observed classification accuracy (loss function) $L = f$ (*i.e.*, equation (2.36)). The reason for this is because Bayesian optimization is any sequential model-based approach to solving a problem. Updates to the parameters x are provided via Bayesian posterior updates, or our updated beliefs given data. Posterior updates are guided by acquisition functions $\alpha_n : \chi$, where the next parameters in time x_{n+1} are chosen by maximizing the



(a) Standard Neural Net (b) After applying dropout.

Figure 2.40: An illustration [5] of the last few layers of a linear classifier before (a) and after (b) dropout layers are implemented. Arrows represent connections between neurons, while neurons with x's through them represent neuron connections terminated by being dropped out.

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 fucntion f, so it is critical that the acquisition functions be simple to evaluate. See Figure 2.41 for an illustration of three time iterations of Bayesian optimization.

Parametric Models

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'

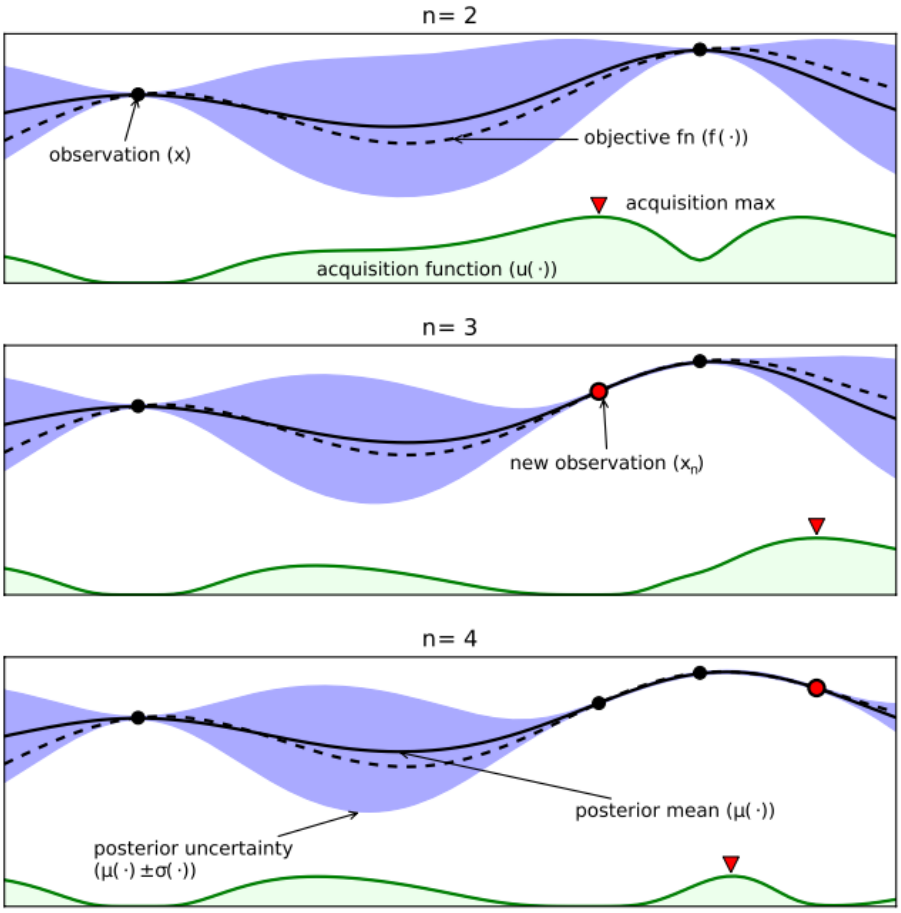


Figure 2.41: An illustration [7] of three time iterations of (2.56). The black line is the estimated objective or loss function f, while the dashed black line is the true f (unknown but visualized). The acquisition function α is in green, whose maxima are highlighted with red arrows, indicating either 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.

rule:

$$p(w|\mathcal{D}) = \frac{p(\mathcal{D}|w)p(w)}{p(\mathcal{D})}, \tag{2.57}$$

which describes our updated beliefs about w after observing data $\mathcal{D}$. The choice of statistical model used now is paramount to the effectiveness of the Bayesian optimization [7]. The simplest such model to perform Thompson Sampling in the Beta-Bernoulli Bandit Model. 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:

$$p(w|\alpha, \beta) = \prod_{a=1}^{K} beta(w_a|\alpha, \beta), \tag{2.58}$$

which is a good choice because beta distributions are the conjugate prior to the Bernoulli likelihood, or part of the same probability distribution family. For example, given the Bernoulli likelihood:

$$p(s) = \binom{n}{s} q^s (1-q)^{n-s}, \tag{2.59}$$

for s successes, n trials, and $q \in (0, 1)$ the probability of success, the beta distribution conjugate prior describing the probabilities of which values q can take can be described as:

$$p(q) = \frac{q^{\alpha-1}(1-q)^{\beta-1}}{B(\alpha, \beta)}. \tag{2.60}$$

Going back to the bandit statistical model, the a posteriori distribution can be derived using (2.58) as:

$$p(w|\alpha, \beta) = \prod_{a=1}^{K} beta(w_a|\alpha + n_{a,1}, \beta + n_{a,0}), \tag{2.61}$$

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 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 $f_{\tilde{w}}$:

$$a_{n+1} = \arg\max_{a} f_{\tilde{w}}(a), \tilde{w} \sim p(w|\mathcal{D}_n). \tag{2.62}$$

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.

Non-Parametric Models

Can we make predictions on outcomes from arm-pulling without the parameters w? Using a kernel mapping trick, 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:

$$K_{i,j} = k(x_i, x_j) = \Phi(x_i) V_0 \Phi(x_j)^T = \langle \Phi(x_i), \Phi(x_j) \rangle_{V_0}, \tag{2.63}$$

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, $x_{i,j}$ 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:

$$\mu_n(x) = \mu_0(x) + k(x)^T (K + \sigma^2 I)^{-1} (y - m), \tag{2.64a}$$

$$\sigma_n^2(x) = \mathtt{k}(x, x) - k(x)^T (K + \sigma^2 I)^{-1} k(x), \tag{2.64b}$$

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 2.41, accomplished without weights W but with kernels K. A simple and common kernel $\mathtt{k}(x, x')$ is the Matérn [34] stationary kernel covariance function:

$$\mathtt{k}(x, x') = \theta_0^2 \exp(-r), \tag{2.65}$$

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 Bayeseian 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. Finally there is a choice of acquisition functions not even covered here, although in [7] it is mentioned that hyper-parameter and acquisition function choice does not have a strong impact on performance.

2.2.5 Novel Training Methods: Distillation

Distillation is an idea [35] 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 2.2.1: Soft-Max Classifier, the soft-max function (2.39) 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:

$$f_j(z) = \frac{e^{z_j/T}}{\sum_k e^{z_k/T}}. \tag{2.66}$$

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 [35] for its increases in classification accuracy [35] and increased security against adversarial perturbations [36].

2.2.6 Novel Training Methods: Generative Adversarial Network (GAN)

Generative Adversarial Networks [9] 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 CNN into thinking the signals it is observing are not of one modulation scheme, but another. It has been shown in many works [8] (see Figure 2.42) that small changes can be made to signals, images, and voice can render brittle classifiers useless.

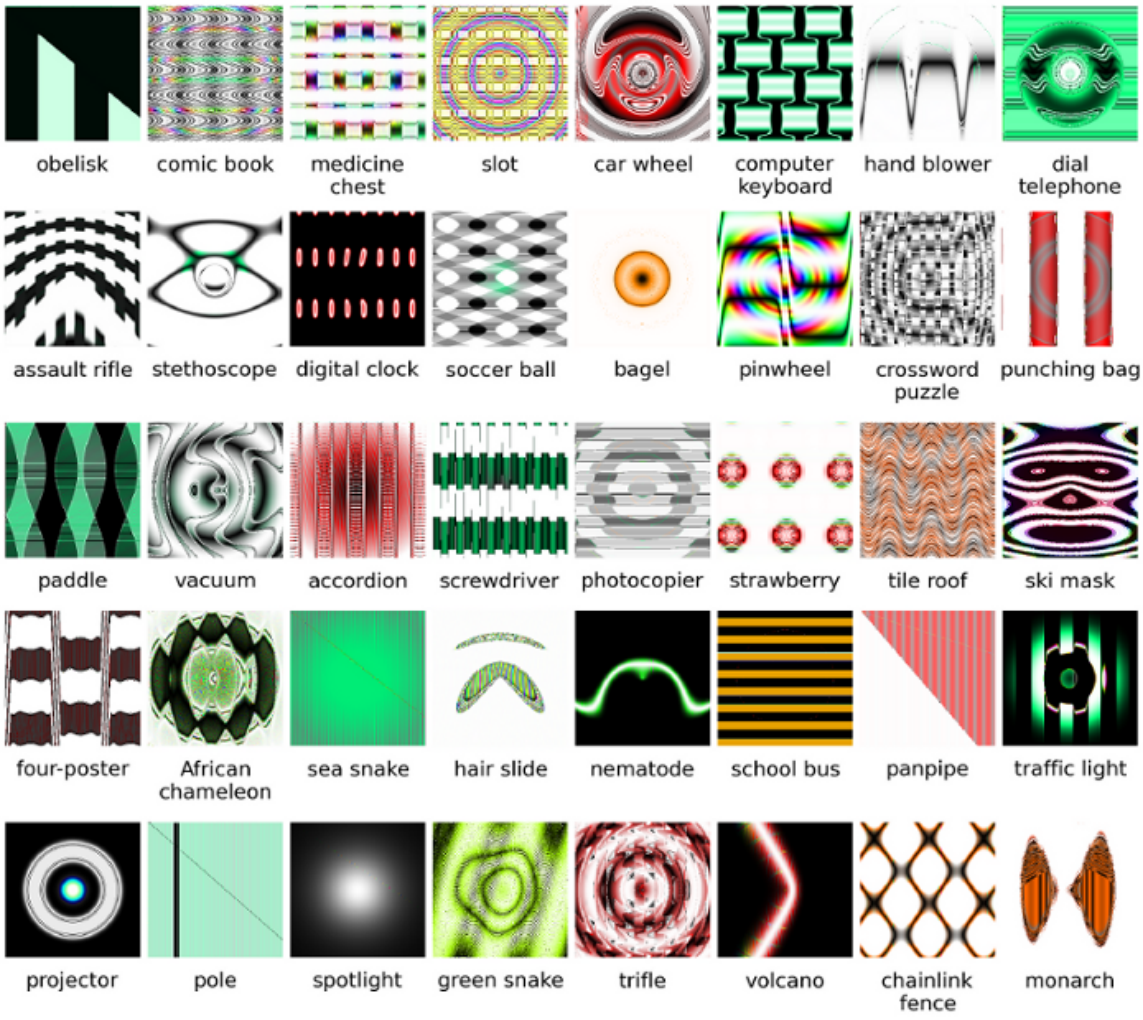


Figure 2.42: A series of decoy images [8] fooling a computer vision classifier. Most images fail to remotely resemble their target label, however due to the brittle nature of training neural networks with standard back-pass techniques, small movements in feature-space at evaluation time can have significant and catastrophic results.

In a GAN setup, the decoy network described above is called the generator, while the modulation classifier is the discriminator. See Figure 2.43 for an overview of how networks in a GAN interact with each other.

The essence 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

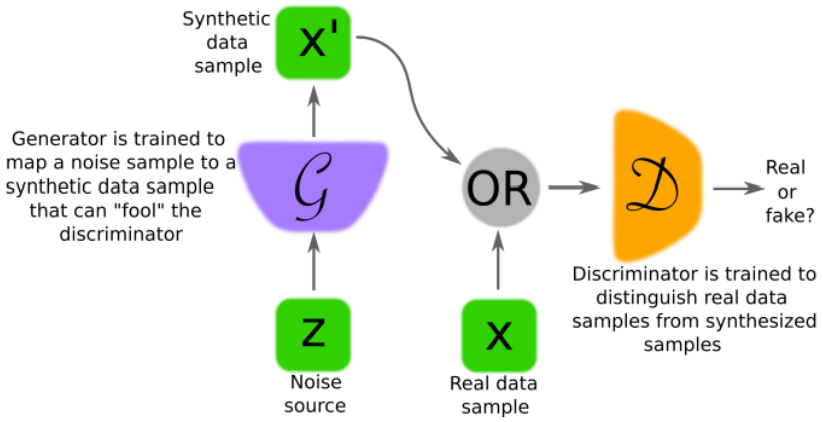


Figure 2.43: A flow diagram [9] of a GAN process. Synthetic data samples are added to data samples observed by the discriminator.

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.

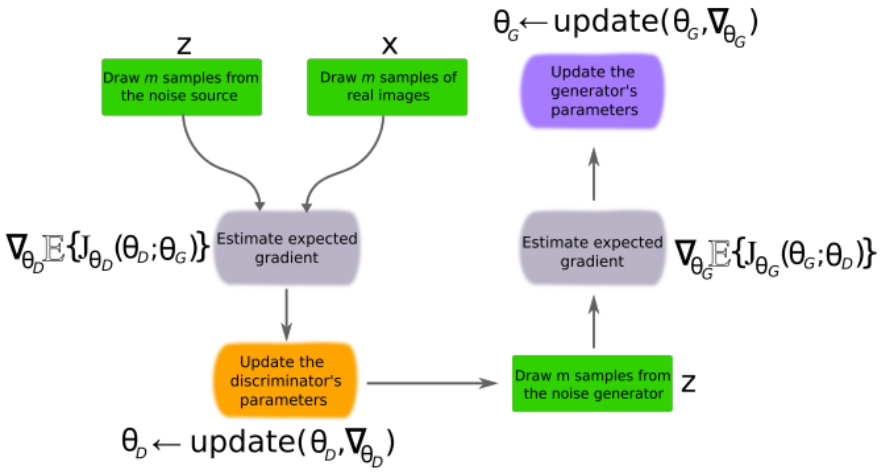


Figure 2.44: A flow diagram [9] describing the SGD (2.42) feedback loop between the generator and discriminator (see Figure 2.43). Parameter updates continue until the learning capacity of the networks are reached and classification accuracy of the discriminator reaches a steady state value $\in (0.5, 1)$.

2.2.7 Novel Training Methods: Domain Adaptation

There is a method of generalized training [10] 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 2.45).

The core idea of domain adaptation is to create a latent space Z (see Figure 2.46) 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 (see Figure 2.47) is minimized.

Domain adaptation attempts to perform SGD (2.42) on the general loss function

$$Q = \lambda_c Q_c + \lambda_{id} Q_{id} + \lambda_z Q_z + \lambda_{tr} Q_{tr} + \lambda_{cyc} Q_{cyc} + \lambda_{trc} Q_{trc}, \tag{2.67}$$

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 2.2.1): calculate some difference between each ground truth label c_i and each evaluation signal. This is a standard neural net operation, independent from domain adaptation, described as [37]:

  $$Q_c = \sum_i l_c(h(f_x(x_i)), c_i), \tag{2.68}$$

  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 2.46 from Z space to the annotations C, 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 2.46 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

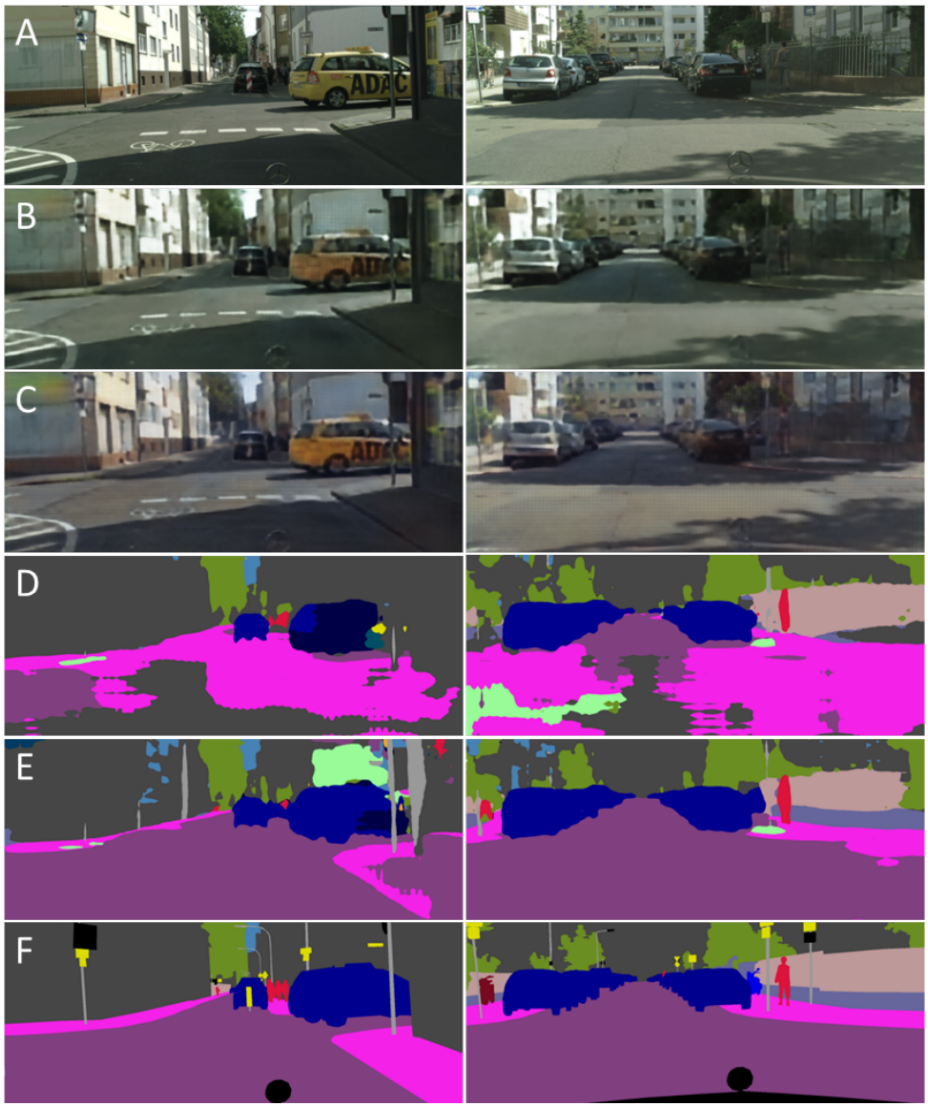


Figure 2.45: A series of testing images and classification results [10]. A) Test image collected from real Cityscapes dataset. B) Identity mapped version of the image. C) Image translated to the target domain. D) Evaluation of translated image without domain adaptation. E) Evaluation of translated image using domain adaptation [10]. F) Translated image ground truth.

as possible. That is the role of the individual loss function Q_{id} [38], which is described

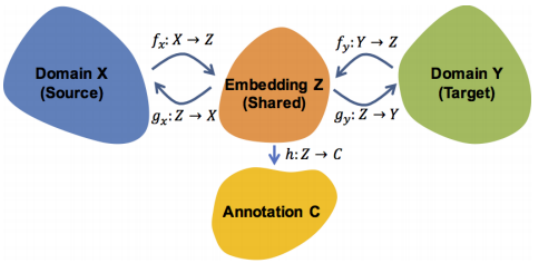


Figure 2.46: A flow diagram [10] describing the various transforms f_x, g_x, h, f_y, g_y and spaces X, Z, Y, C and their interactions at the highest level in domain adaptation. The field is motivated by scarcity of annotated real pictures, but has much wider applications. Implemented correctly, training 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.

as:

$$Q_{id} = \sum_i l_{id}(g_x(f_x(x_i)), x_i) + \sum_j l_{id}(g_y(f_y(y_j)), y_j), \tag{2.69}$$

where 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 2.2.6) 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 (2.67) can be described as [39]:

$$Q_z = \sum_i l_a(d_z) f_x(x_i)), c_x) + \sum_j l_a(d_z(f_y(y_j)), c_y), \tag{2.70}$$

where l_a is a suitable loss function for GANs.

- As an extra precaution to make sure the transforms in Figure 2.46 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 [40] 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:

$$Q_{tr} = \sum_i l_a(d_y(g_y(f_x(x_i))), c_x) + \sum_j l_a(d_x(g_x(f_y(y_j))), c_y), \tag{2.71}$$

- Similarly to (2.69), a cycle loss function [40] 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:

$$Q_{cyc} = \sum_i l_{id}(g_x(f_y(g_y(f_x(x_i)))), x_i) + \sum_j l_{id}(g_y(f_x(g_x(f_y(y_j)))), y_j), \tag{2.72}$$

- Similarly to (2.68), a translations loss function formulated in [10] 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.

$$Q_{trc} = \sum_i l_c(h(f_y(g_y(f_x(x_i)))), c_i). \tag{2.73}$$

2.3 Modulation Classification

In Section 2.2, Chapter 3, and Chapter 4, 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, and the different forms neural nets take to classify them.

2.3.1 Signal 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, $\cos(\omega_0 t)$.

$$y(t) = u(t)\cos(\omega_0 t), \tag{2.74}$$

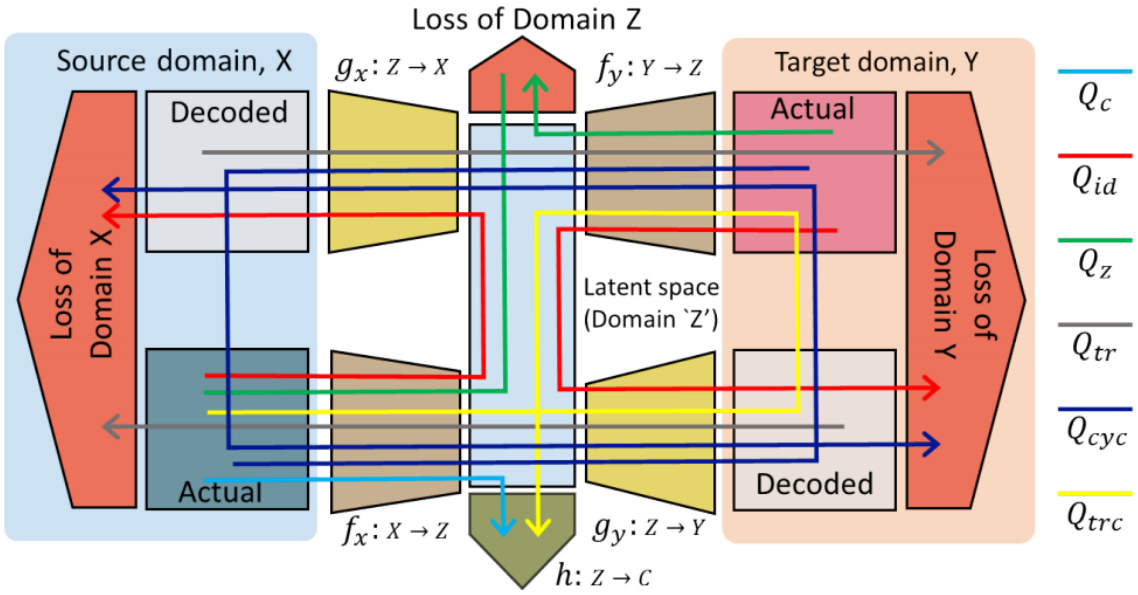


Figure 2.47: An elaborated [10] flow diagram of Figure 2.46, describing additionally the various weighted loss functions $Q_c, Q_{id}, Q_z, Q_{tr}, Q_{cyc}, Q_{trc}$ and how they interact with each domain X, Y, and Z. See equations (2.68) through (2.73).

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:

$$z(t) = u(t)\cos(\omega_0 t). \tag{2.75}$$

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 [2].

Designing a Modulation Scheme

While an exhaustive list of modulation schemes and their performance can be found in [11], the design and use of the Quadrature Phase Shift Keying (QPSK) modulation

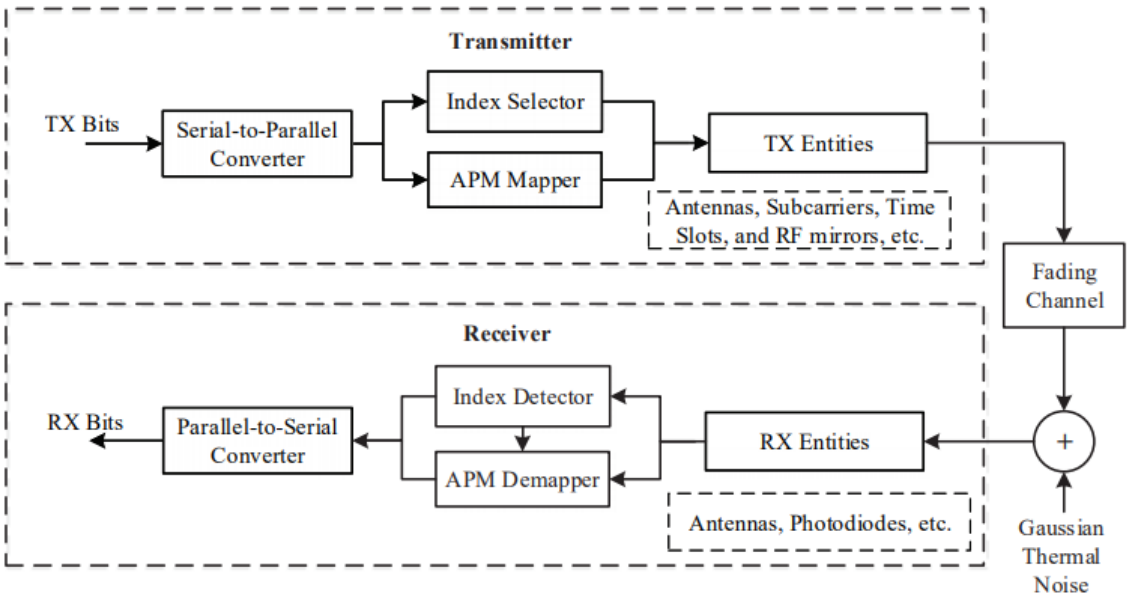


Figure 2.48: A very high-level flow diagram [11] describing the flow of information in a communications transmit-receive pair. Information begins as bits mapped to IQ points, is transformed into a voltage by a DAC, transduced into an electromagnetic wave by a transmitting antenna, travels through a noisy channel, is transduced back into a voltage by a receiving antenna, detected and transformed into IQ points with the help of a ADC, and finally mapped back to bits.

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:

$$2^b = M. \tag{2.76}$$

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:

$$s_n(t) = \sqrt{\frac{2E_s}{T_s}}\cos(2\pi f_c t + (2n-1)\frac{\pi}{4}), n = 1, 2, 3, 4, \tag{2.77}$$

where E_s is the energy in Joules per symbol $n = i : b \to (b_1, b_2), i = 1, 2, 3, 4$, 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 [2]:

$$\phi_1(t) = \sqrt{\frac{2}{T_s}}\cos(2\pi f_c t), \tag{2.78a}$$

$$\phi_2(t) = \sqrt{\frac{2}{T_s}}\sin(2\pi f_c t), \tag{2.78b}$$

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 2.49. 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.

$$\epsilon_p = \frac{d^2_{min}}{\bar{E}_b} \tag{2.79}$$

For the case of QPSK, there are two bits per symbol, so $\bar{E}_b = \frac{1}{2}\bar{E}_s$. For (2.78), 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 [2].

2.3.2 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

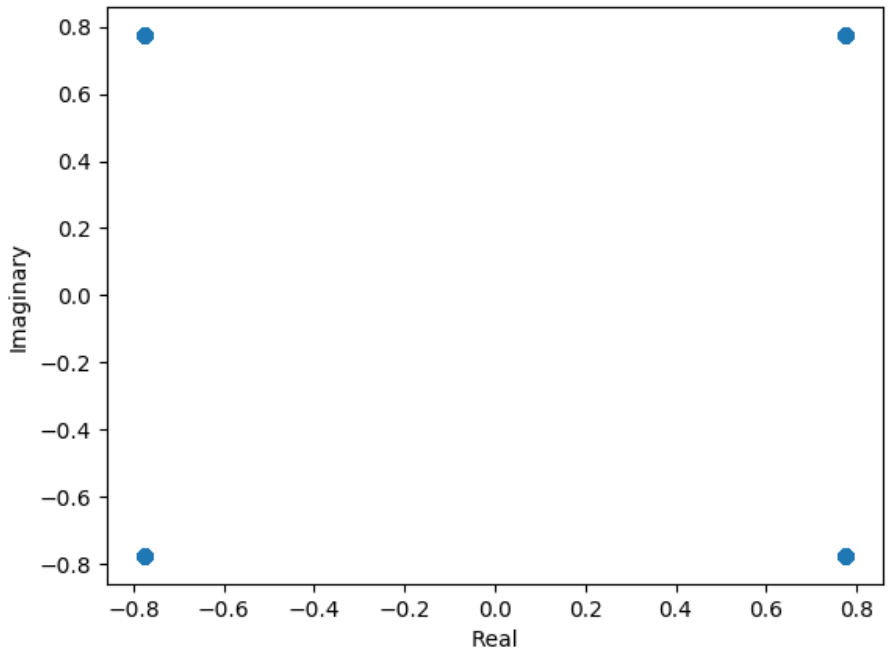


Figure 2.49: A set of QPSK constellation points (2.77) for $E_s = 4$. The horizontal axis is defined as $\phi_1(t)$ or the real valued element in a complex tuple, and the vertical axis as $\phi_2(t)$, traditionally represented as the imaginary valued element in a complex tuple. The resulting transformations 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 $n = 4 : b \to (1,1) : s \to (2/\sqrt{2}, -2/\sqrt{2})$

be properly mapped to the right message bits, bit error rate can increase significantly. Although numerous [12] 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 [12]. It is the goal of this section to investigate the architecture of a foundational [12] 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 [12], 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 2.2.3) and a single dense layer followed by a soft-max (2.39) classifier. The hidden layers are followed by ReLU (2.51) activation layers, and dropout layers (see Figure 2.40) with $p = 0.5$. In [12] figures describing their optimization process are detailed, resulting in choosing a filter size $F = 8$, stride $S = 1$, and 50 filters (see Figure 2.37 for an example convolution).

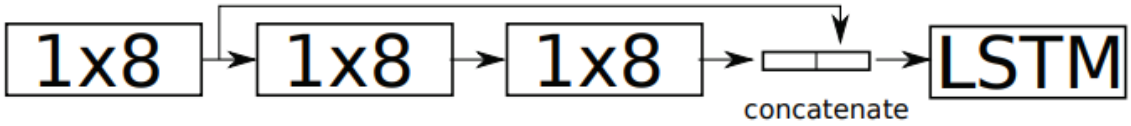


Figure 2.50: A flow chart [12] describing the forward pass (see Figure 2.32) of a set of eight input values through the CLDNN. A $[1, 8]$ input vector is concatenated with values filtered through a $[1, 8]$ filter in both the first and second convolutional layer. Each filter (see Figure 2.52) contains eight weights and one bias value (see Figure 2.37 for example filters), which are calculated during SGD (2.42). The Long Short-Term Memory (LSTM) cell holds the values for the soft-max classification layer.

A common metric in modulation classification used to evaluate, in detail, the performance of a classifier is the confusion matrix, which simply describes how correctly a function answers different questions. The CLDNN in [12] calculates a confusion matrix in Figure 2.53.

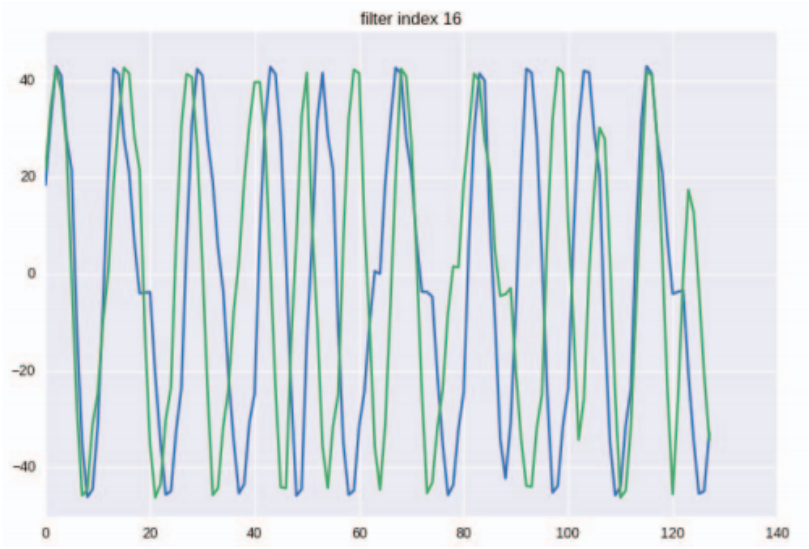


Figure 2.51: The time-domain IQ plot [12] of a $[2, 128]$ output signal from the $[1, 8]$ filter in Figure 2.52. The signal input to the filter was random, but trained to maximally activate the filters eight weights. The result is a Binary Phase Shift Keying (BPSK) waveform, indicating that this filter was trained to maximize the eventual soft-max class scores of BPSK signals.

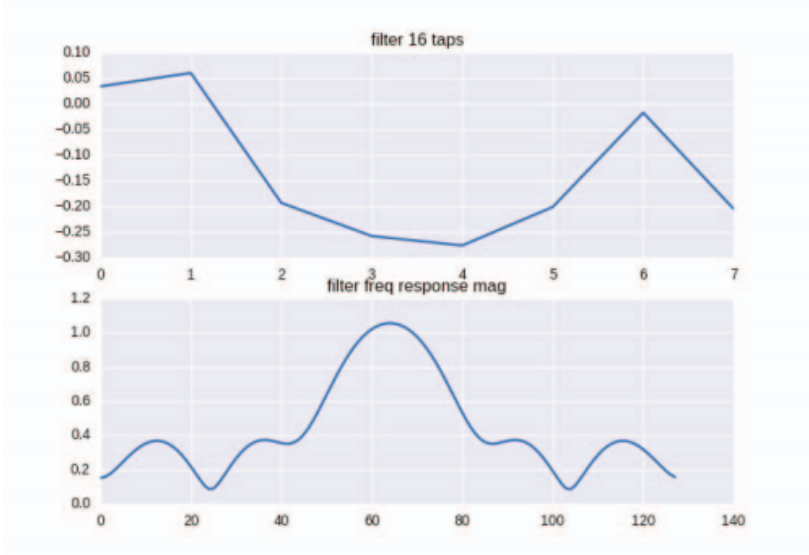


Figure 2.52: Time (top) and frequency (bottom) domain magnitude plots [12] of a trained $[1, 8]$ filter like those in Figure 2.50. This filter's first, second, and seventh weights have the most influence on classification. See Figure 2.51 for another visualization of this filter.

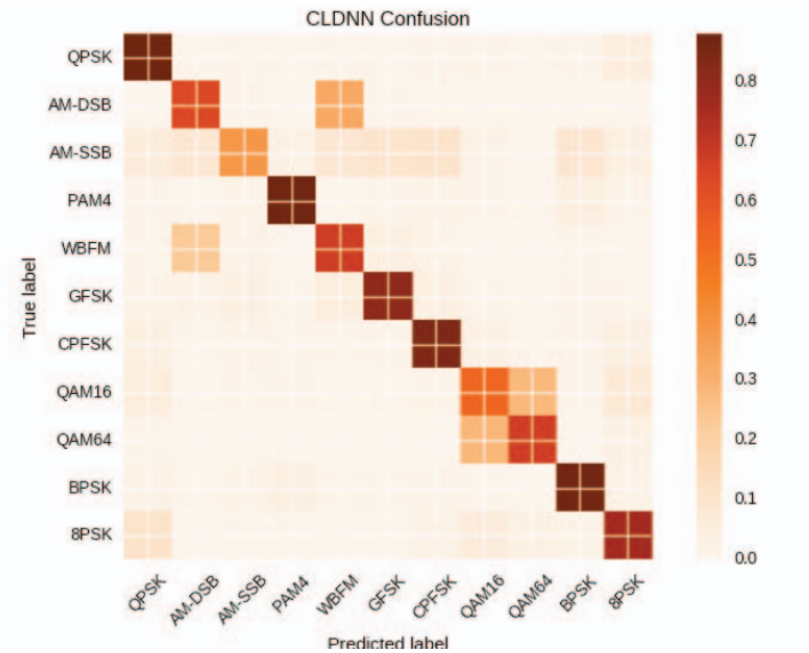


Figure 2.53: A confusion matrix [12] describing the classification accuracy of the CLDNN on all test signals at evaluation time. The color gradient communicates classification accuracy averaged over SNR values ranging from -20 dB to 20dB. The horizontal axis displays the modulation scheme that the CLDNN classifies signals by, and the vertical axis the ground truth of those signals. A perfectly performing classifier would have a deep brown 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.

2.4 Summary

In this chapter, various classical models for transmission imperfections and channel models were surveyed in Section 2.1. In Section 2.2, the fundamentals of deep learning was discussed, specifically topics concerning training. Finally, Section 2.3 presented a survey of modulation classification, the primary form of neural network evaluation discussed in this thesis.

In the next Chapter 3, a novel framework for low-decay, low-bias dataset generation is

presented.

Chapter 3

Physical Layer Neural Network Framework for Training Data Formation

3.1 Abstract

In this paper, we propose a low decay, low bias dataset synthesis framework 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. The demand for signal-domain Neural Networks (NNs) have increased significantly in recent years with respect to the classification of observed radio activity. In particular, 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.

3.2 Introduction

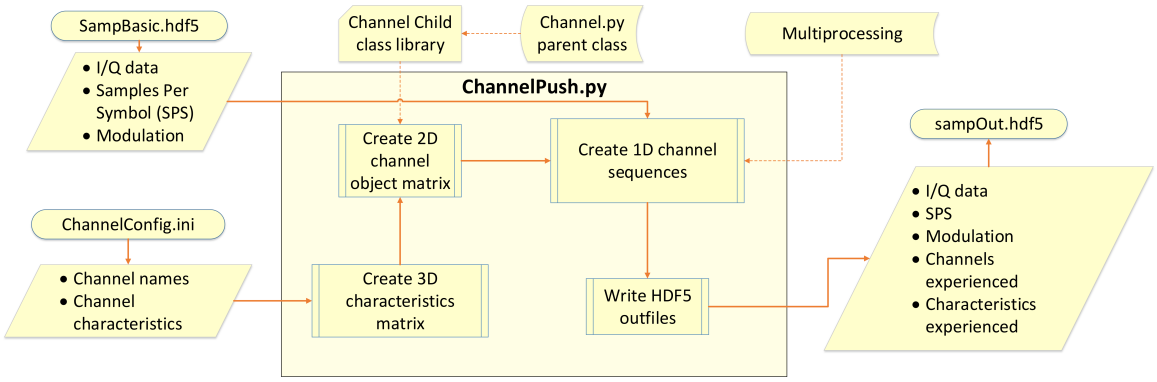


Figure 3.1: Illustration of the proposed framework and the ChannelPush.py script. SampBasic.hdf5 acts as the Dataset Under Test (DUT) while ChannelConfig.ini as the instructions file. SampOut.hdf5 files are written as outputs. The 3D matrix is formed by the instructions file, containing the 2D matrix's (see Table 3.1) instance variables. The 2D matrix objects are formed by run-time channel class imports. 1D channel sequences (see Figure 3.2) 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.

Radio Frequency (RF) Neural Networks (NN) have recently received significant attention within the wireless research community [41, 42]. However, RF NNs do not possess the openly available datasets and established benchmarks that are frequently associated with other NN applications [13, 41]. 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 [2, 16], which can negatively affect a NNs classification accuracy in highly time-varying and probabilistic environments. In recent publications [13, 43], 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 [13]. 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 [13]. While the dataset is popular due to its availability, some researchers have had difficulty in replicating the peak accuracy of their classifier [44], 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 [43] 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.

The work presented in this paper 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 paper is organized as follows: The proposed framework and dataset variations are presented in Section 3.3, and applications of the framework in Section 3.4. Section 3.5 presents KLD entropy analysis of training set approximations of PDFs and their implications on training set size, and concluding thoughts are presented in Section 3.6.

3.3 Proposed Framework

The proposed framework is described in Figure 3.1. 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 3.1: Example 2D Channel Object Matrix (refer to Figure 3.1). 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, $x = [N \times D]$ [5]. 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 [45]. A popular and simple heuristic that we consider represents the number of training examples N as the product:

$$N = K \times C \tag{3.1a}$$

$$C = (f \times F) \times (v \times V) \tag{3.1b}$$

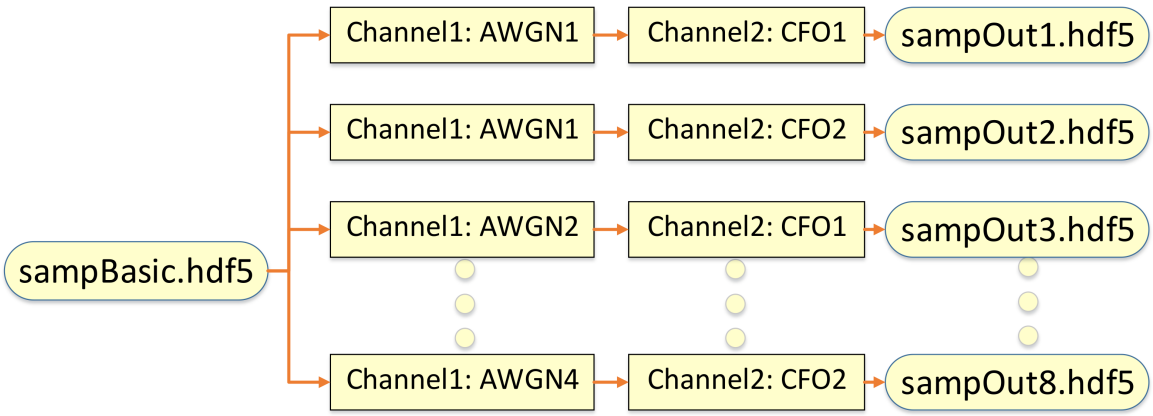


Figure 3.2: Example set of eight 1D channel sequences (refer to Figure 3.1) formed by permuting through the 2D channel object matrix. SampBasic.hdf5 is the DUT, and is pushed through each sequence sample by sample, leveraging Multiprocessing.

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 [46] choose a C value of 6,000 and 600, respectively, and the RF dataset RML2016.10A 1,000 [42].

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 3.2 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:

$$\Delta N = (\Delta f \times F) \times (\Delta v \times V). \tag{3.2}$$

While several input variations are displayed in this paper, 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 3.2.

Many features have well defined classical models that describe their behavior, excluding

certain non-linear phenomenon such as amplifier non-linearities [47]. 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.

Table 3.2: Examples of variations in computer vision image datasets, and a collection of analogies for their signal domain parallel [5].

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)

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

3.4 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 [48]. 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

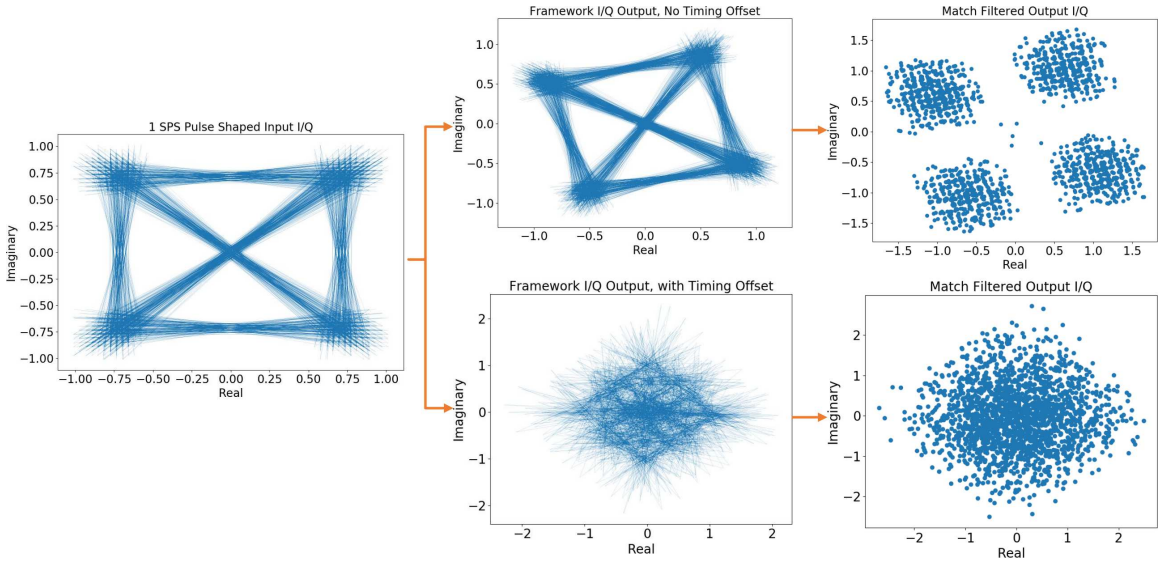


Figure 3.3: 1 SPS pulse shaped Quadrature Phase-Shift Keying (QPSK) IQ data representing the baseband data of an Ettus Research N210 transmission. For the sake of visualization, frequency offset from Local Oscillator (LO) drift has been left out. The top track displays the dataset influenced by phase 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).

Ettus N210 with an SBX daughter-board [18]. Figure 3.3 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 synthe-

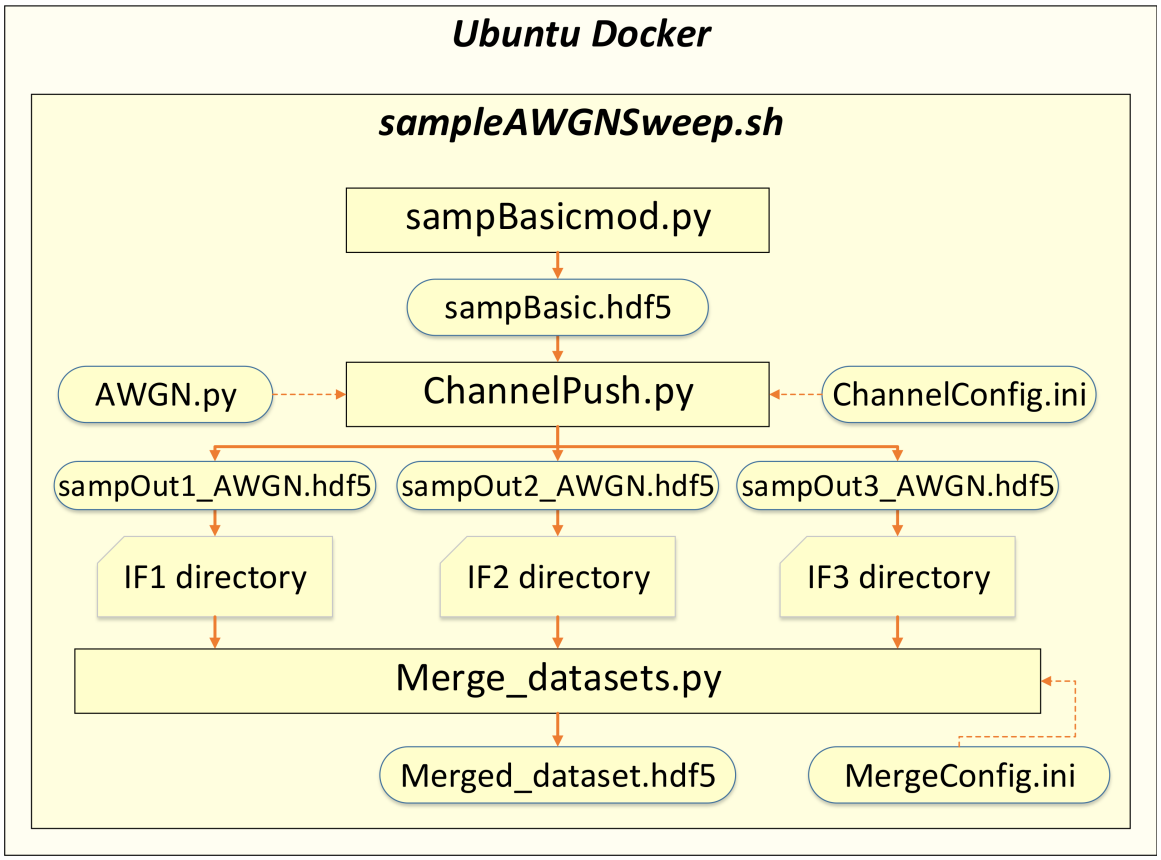


Figure 3.4: The AWGN channel effect is described by its SNR and Gaussian RV variance, σ. Three AWGN channels of varying SNR but constant σ described by ChannelConfig.ini are applied to the same infile sampBasicmod.py. The outputs of which are manually moved to Intermediate Frequency (IF) folders corresponding to a secondary instructions file, MergeConfig.ini. Merge_datasets.py (see Figure 3.5) modulates and sums the independent transmissions.

size 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. Figure 3.4 presents a set of three separate transmissions produced from the proposed framework in pursuit of this goal.

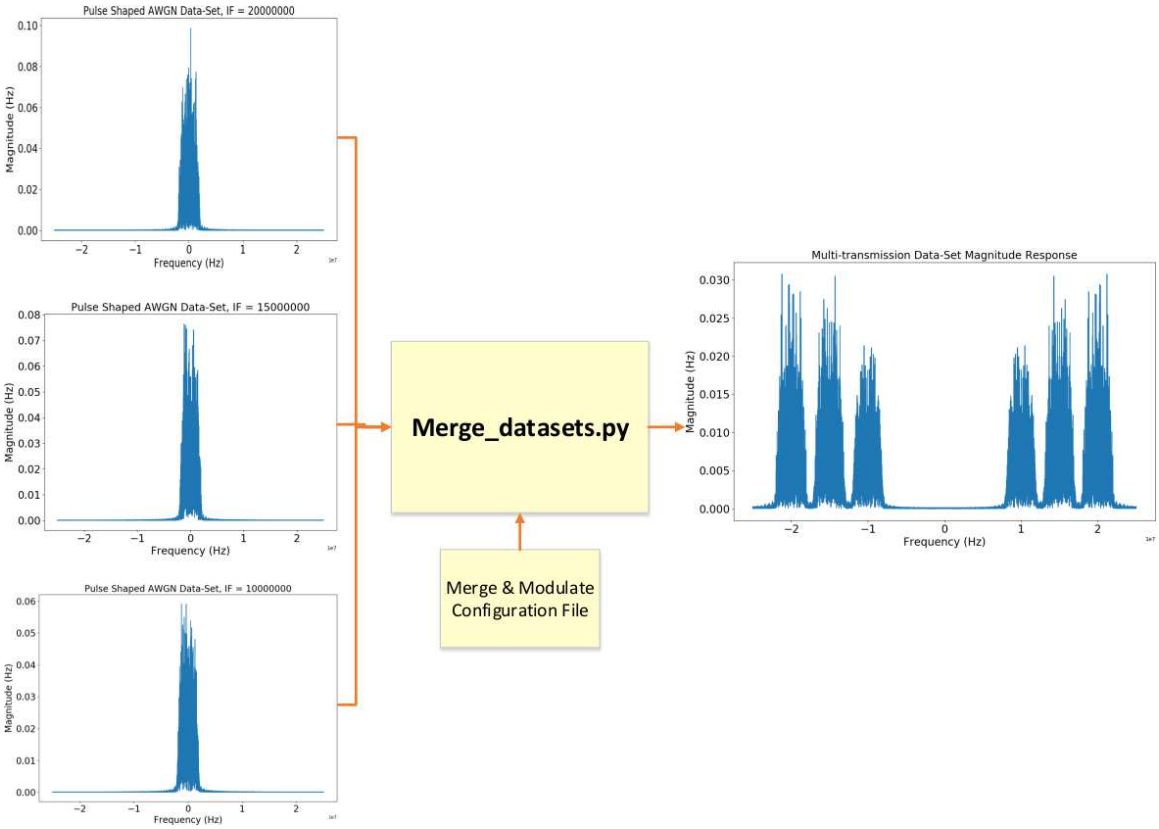


Figure 3.5: Three 16 SPS pulse shaped QPSK datasets from Figure 3.4 are modulated to intermediate frequencies 10, 15, and 20 MHz. Each dataset was pushed through the framework as a DUT and modified by a unique AWGN channel block independently, each representing a transmitted signal. Future work will implement this feature to produce MIMO and OFDM datasets.

3.5 Simulations and Results

In the previous section, we discussed the number of examples a training set should have and what information those examples should contain. As shown in [43], 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, we will investigate how many samples pulled from RVs are needed to estimate their theoretical PDFs to within a specific degree of accuracy.

It is desirable to have the number of samples be integers of base two since it is com-

putationally efficient [5] 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 3.3, we perform the KLD analysis (3.3) in Figure 3.6:

$$D_{KL}(P||Q) = \sum_i P(i) log_2\left(\frac{P(i)}{Q(i)}\right), \tag{3.3}$$

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

$$P(i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\pi\sigma^2}}, \tag{3.4}$$

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

$$q(i) = CDF_{norm}^{-1}(u) + eps, \tag{3.5}$$

where $eps \sim U(0, 0^+)$ to avoid $log_2(0)$ errors in (3.3) 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:

$$CDF_{norm}(x) = \frac{1}{2}\left[1 + \text{erf}\,\frac{x-\mu}{\sigma\sqrt{2}}\right]. \tag{3.6}$$

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 3.3) is described by a uniform RV from a phase ambiguity and STO model, two Gaussian RVs describing the CFO of a transmitting

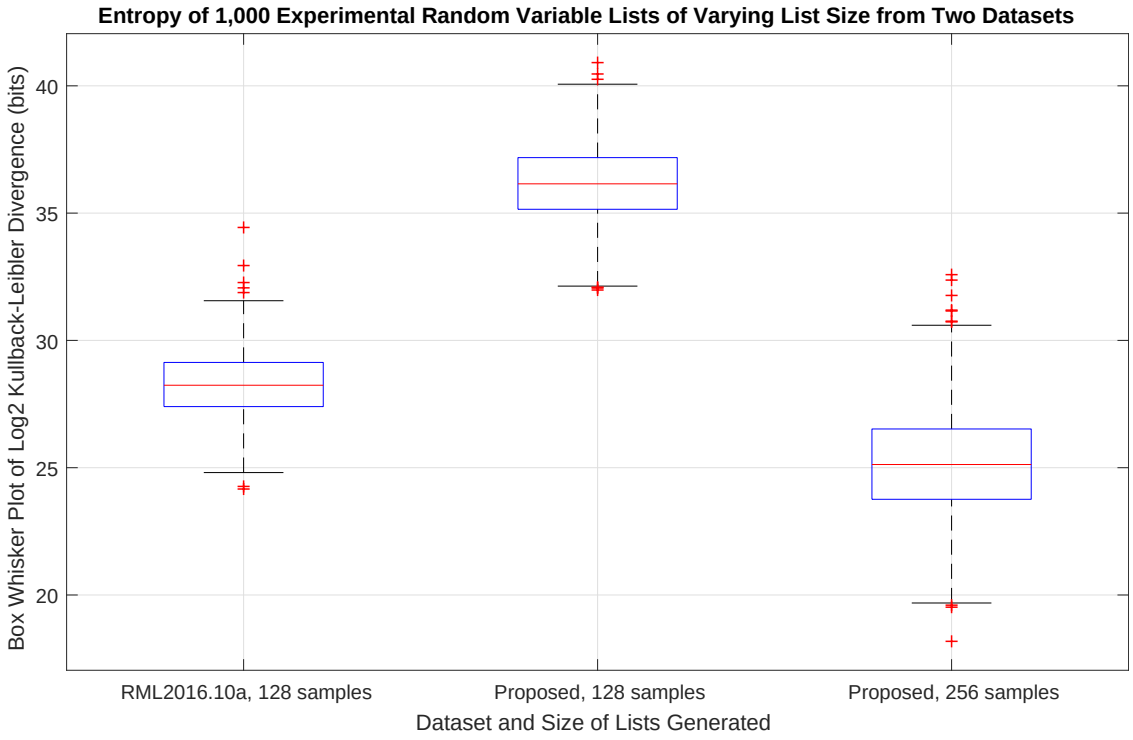


Figure 3.6: RML2016.10A is composed of 1,000 training sets containing 128 samples each per class per SNR value. Transmissions average 28.3 bits divergence from theory. The proposed application (see Figure 3.3) averages 36.2 bit divergence from theory. In order to achieve similar KLD entropy at an RF NNs evaluation time to state-of-the-art datasets, this analysis shows our 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.

and receiving LO, and a Gaussian RV contained in the AWGN model. Since KLD is additive (3.7) for independent distributions (*i.e.*, P_1, P_2, Q_1, Q_2), the values in Figure 3.6 are calculated as the sum of KLD for each RV of the dataset under consideration.

$$D_{KL}(P||Q) = D_{KL}(P_1||Q_1) + D_{KL}(P_2||Q_2). \tag{3.7}$$

3.6 Conclusion

Our 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 our proposed framework achieves 11.58% less entropy than state-of-the-art datasets at classification time.

Chapter 4

Domain Adaptation of Wireless Channels

abstract

4.1 test

test

4.2 Summary

summary

Chapter 5

Conclusion

test

5.1 Research Outcomes

test

5.2 Future Work

test

Bibliography

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

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

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

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

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

[6] M. A. Nielsen, "Neural networks and deep learning," *Determination Press*, 2015.

[7] 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.

[8] 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

[9] 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

[10] 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

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

[12] 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.

[13] 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.

[14] 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

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

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

[17] 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.

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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

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

[27] 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.

[28] 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

[29] 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

[30] 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

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

[32] 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

[33] 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

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

[35] 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

[36] 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

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

[38] 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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

[48] 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.

Appendix A

Data Synthesis Framework

A.1 Meta.py

```
#!/usr/bin/python3.5
"""
Created on Sep 22, 2017
@author: kwmcclintick

Meta class for testing ChannelPush.py, see respective classes for their
    documentation
"""

import warnings
import configparser
import ChannelTool
import sys
warnings.simplefilter(action='ignore', category=FutureWarning)
import h5py
warnings.resetwarnings()

if __name__ == '__main__':
    if len(sys.argv) < 4:
        print("Usage: ./program {HDF5 filename with data}")
```

```
    print("Exiting...")
    quit()
else:
    filename = sys.argv[1]

print('\n------------------BOOTING META FILE------------------')
# data under test
infile = h5py.File(filename, "r")
print("DUT file information:")
for item in infile.attrs.keys():
    print(item + ":", infile.attrs[item])

datapath = infile.attrs['default']
print("# of Symbols: " + str(int(len(infile[datapath]) / infile['source_data
    '].attrs['oversampling_factor'])))
print("Samples Per Symbol: " + str(infile['source_data'].attrs['
    oversampling_factor']))
print("# of Samples: " + str(len(infile[datapath])))

# channelconfig gives instructions to the ChannelTool on what to test the
    data with
# this is just printing out the contents of channelconfig
channelconfig = configparser.ConfigParser()
print("\n" + "Instructions File information:")
print(str(channelconfig.read(str(sys.argv[2]))) + " Channels and Channel
    Characteristics have been loaded")
for each_section in channelconfig.sections():
    print("[" + each_section + "]")
    for (each_key, each_val) in channelconfig.items(each_section):
        print(each_key + ":  " + each_val)

output_tag = sys.argv[3]
# ChannelTool multiplies input files by pushing them through different
    channels described by channelconfig
ChannelTool = ChannelTool.ChannelTool(channelconfig, infile, output_tag)

ChannelTool.run()
```

```
    # ChannelTool object created, file multiplication called
    print("\n" + "please wait, pushing samples through each channel series...")
    infile.close()
```

A.2 ChannelPush.py

```
"""
Created on Oct, 10 2017
author: kwmcclintick
The ChannelTool class interacts with channel objects in mass, initializes
    information from ChannelConfig.ini,
and handles file multiplication by passing inputs through a variety of typical
    wireless channels with typical design
parameters
"""

import sys
import matplotlib.pyplot as plt
import os
from multiprocessing import Process, Pool
import numpy as np
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
import h5py
warnings.resetwarnings()


class ChannelTool:
    def __init__(self, channelconfig, infile, output_tag):  # constructor
        print('\n----------INITIALIZING DATASET SYNTHESIS FRAMEWORK----------')
        self.infile = infile
        self.output_tag = output_tag
        self.channelconfig = channelconfig
```

```
    # CREATING THE CHARACTERISTICS MATRIX FROM CHANNELCONFIG.INI
    print('////// CREATING 3D CHARACTERISTICS MATRIX //////')
    characteristics = self.charMatrix()

    # SEARCH CHANNELS FOLDER AND CREATE CHANNEL OBJECTS
    print('////// CREATING 2D MATRIX OF CHANNEL OBJECTS //////')
    self.channels = self.chanMatrix(characteristics)

'''
Running the ChannelTool picks one instance from each channel type and
    generates a data flow from those channels,
modifying the input HDF5 file one channel at a time until saving the last
    channel's output as new HDF5 file with
identifying attributes as to how the data was changed.
'''
def run(self):
    print('////// CREATING 1D CHANNEL SEQUENCES //////')
    channel_combos\
        = [list(x) for x in np.array(np.meshgrid(*self.channels)).T.reshape
            (-1, len(self.channels))]

    for k in range(len(channel_combos)):
        print("Starting channel data flow process " + str(k + 1) + " of " +
            str(len(channel_combos)))
        p = Process(target=self.dataFlow, args=(k, channel_combos[k],))
        p.start()

'''
Returns the first result in the directory path for the file of name name
'''
def find(self, name, path):
    for root, dirs, files in os.walk(path):
        if name in files:
            return os.path.join(root, name)

'''
attaches labels to the hdf5 outfile and writes it to the HDF5_files
```

```
    directory
'''
def dataFlow(self, k, channel_combos):
    # writing each hdf5 data set here
    outfile = h5py.File('./HDF5_files/noisyQPSK_' + str(k + 1) +
        self.output_tag + '.hdf5', "w")
    # point to the default data to be plotted
    outfile.attrs["default"] = "/out/bbIQ"  # group name/data set name
    # give the HDF5 root some more attributes, keeping group attributes same
         as input files (mod scheme, sps)
    outfile.attrs["Outfile_Name"] = 'noisyQPSK_' + str(k + 1) +
        self.output_tag
    outfile.attrs["creator"] = sys.argv[0]
    outfile.attrs["HDF5_Version"] = h5py.version.hdf5_version
    outfile.attrs["h5py_version"] = h5py.version.version
    outfile.attrs["modulation"] = self.infile['source_data'].attrs['
        modulation']
    outfile.attrs["oversampling_factor"] = self.infile['source_data'].attrs[
        'oversampling_factor']

    # infile data is kept on hand for SER calculations and such
    source = outfile.create_group("source_data")
    source.create_dataset("bbIQ", data=self.infile[self.infile.attrs['
        default']])

    # each channels dash bigta flow computes here, attaching attributes to
        the hdf5 file as it goes
    outdata = self.labelPush(np.array(channel_combos, dtype=np.object),
        outfile)

    # initializing the hdf5 group that each data set will be written to
    grp = outfile.create_group("out")
    # file is actually written and closed
    grp.create_dataset("bbIQ", data=outdata)
    outfile.close()
    print('Closing hdf5 file: ' + str(k+1))
```

```
'''
Pushes data sample by sample through the kth sequence of channel_combos,
    attaching characteristic labels to hdf5
outfile using attrs
'''
def labelPush(self, channel_combos, outfile):
    # single channel sequences
    if len(channel_combos) == 1:
        # attach labels
        for i in range(len(channel_combos[0].getCharacteristics())):
            outfile.attrs[str(channel_combos[0].characteristic_names[i])] \
                = channel_combos[0].getCharacteristics()[i]
        channel_combos[0].compute()
        outdata = channel_combos[0].getOutput()

    # multiple channel sequences
    if len(channel_combos) > 1:
        for c in range(len(channel_combos) - 1):
            channel_combos[c].compute()
            channel_combos[c + 1].setInput(channel_combos[c].getOutput())
            # attach labels
            for i in range(len(channel_combos[c].getCharacteristics())):
                outfile.attrs[str(channel_combos[c].characteristic_names[i])
                    ] \
                    = channel_combos[c].getCharacteristics()[i]
        # compute final channel output and attach final label
        channel_combos[len(channel_combos) - 1].compute()
        for i in range(len(channel_combos[c + 1].getCharacteristics())):
            outfile.attrs[str(channel_combos[c + 1].characteristic_names[i])
                ] \
                = channel_combos[c + 1].getCharacteristics()[i]
        outdata = channel_combos[len(channel_combos) - 1].getOutput()
    return outdata

'''
Creates a 3D characteristics matrix. Each channel has a list of
    characteristics, and each characteristic
```

```
has a list of values it iterates through
'''
def charMatrix(self):
    num_channels = len(self.channelconfig.items(self.channelconfig.sections
        ()[0]))
    characteristics = [[0 for x in range(1)] for y in range(num_channels)]
    for i in range(num_channels):
        # .ini files store values as strings, so I use a delimiter to
            separate the string into characteristics
        cur_chars = self.channelconfig['CHARACTERISTICS']['characteristics'
            + str(i + 1)].split('|')
        for k in range(len(cur_chars)):
            # and here again into separate values
            cur_chars[k] = cur_chars[k].split(',')
            # now everything is organized by characteristic and value, but
                are still strings. Here I parse as floats
            for b in range(len(cur_chars[k])):
                cur_chars[k][b] = float(cur_chars[k][b])
        # finally we have our characteristics matrix for the i-th channel.
            We repeat again for each testing channel
        characteristics[i] = cur_chars
    return np.array(characteristics, dtype=np.object)

'''
Creates a 2D matrix of every possible combination of unique series of
    channel realizations from the
characteristics 3D matrix and config file instructions
'''
def chanMatrix(self, characteristics):
    path = './Channels'
    # to initialize the channels, first I create a list of the names of
        testing channels
    channelsNames = self.channelconfig.items(self.channelconfig.sections()
        [0])
    channelsPaths = [None] * len(channelsNames)
    channels = [[0 for x in range(0)] for y in range(len(channelsNames))]
    # the list of channel names is iterated through and a list of channel
```

```
            objects is initialized according to
        # the list of channel names and the list of characteristics

        for i in range(len(channelsNames)):
            channelsNames[i] = channelsNames[i][1]
            channelsPaths[i] = self.find(channelsNames[i] + '.py', path)
            if channelsPaths[i] is None:
                sys.exit('Error: channel ' + channelsNames[i] + ' not found or
                    not Python file')

        for i in range(len(channels)):
            # these channels have non-zero characteristics so a channel object
                is created for each combo of character
            self.combos = \
                [list(x) for x in
                 np.array(np.meshgrid(*characteristics[i])).T.reshape(-1, len(
                     characteristics[i]))]
            for d in range(len(self.combos)):
                mod = __import__('Channels.' + channelsNames[i], globals(),
                    locals(), channelsNames[i])
                class_ = getattr(mod, channelsNames[i])
                cur_channel = class_(self.combos[d], self.infile)
                if len(self.combos[d]) != len(
                    cur_channel.getCharacteristic_names()):
                    print(len(self.combos[d]))
                    print(len(cur_channel.getCharacteristic_names()))
                    sys.exit('Error: channel ' + channelsNames[i] + ' has wrong
                        number of characteristics')
                channels[i].insert(d, cur_channel)
        return channels
```

A.3 merge_datasets.py

```
#!/usr/bin/python3.5
"""
```

```
Created on Sep 22, 2017
@author: kwmcclintick

Meta class for testing ChannelTool.py, see respective classes for their
    documentation
"""

from pylab import *
import warnings
import math
import os
import configparser
from os import listdir
from os.path import isfile, join
import sys
warnings.simplefilter(action='ignore', category=FutureWarning)
import h5py
warnings.resetwarnings()


'''
modulate frequency shifts data to a specified Intermediate Frequency (IF)
modulated data = I*cos(2pi * fc * t) + Q*sin(2pi * fc * t)
where t = n*Ts where Ts = 1/Fs, sampling frequency, n is the nth sample
'''


def modulate(file, Fs, IF):
    print('\nModulating files...')
    data = file[file.attrs['default']]
    mod_data = [None] * len(data)

    # plt.magnitude_spectrum(data, Fs=50000000)
    # plt.title("Pulse Shaped AWGN Data-Set, IF = " + IF)
    # plt.show()

    phase_per_sample = 2*math.pi * float(IF) * 1/float(Fs)
```

```
    for i in range(len(data)):  # phase grows with each symbol, spinning the
        constellation with time
        curr_phase = phase_per_sample * (i + 1)
        new_real = data[i].real*math.cos(curr_phase)
        new_imag = data[i].imag*math.sin(curr_phase)
        new_imag = 0  # sampling must be real, not complex
        mod_data[i] = complex(new_real, new_imag)
    return mod_data


'''
sums the complex values of all files_to_sum
'''


def sum_files(data_to_sum):
    print("Summing files...")
    merged_data = [complex(0,0)] * len(data_to_sum[0][0])

    for each_dir in data_to_sum:
        for each_list in each_dir:
            for i in range(len(each_list)):
                merged_data[i] = merged_data[i] + each_list[i]
    return merged_data


if __name__ == '__main__':
    if len(sys.argv) < 5:
        print("Usage: ./program {HDF5 filename with data}")
        print("Exiting...")
        quit()
    else:
        # directory containing each IF sub directory
        dir = sys.argv[1]

    print('\n--------------MODULATING AND SUMMING DATASETS--------------')
    # list of IF sub directories IF1, IF2, etc.
```

```
sub_dirs = [x[0] for x in os.walk(dir)]
sub_dirs.pop(0)

# mergeconfig gives instructions on what frequency to modulate which
    datasets up to
mergeconfig = configparser.ConfigParser()
print(str(mergeconfig.read(str(sys.argv[2]))) + " Intermediate Frequency
    mappings have been loaded")
for each_section in mergeconfig.sections():
    print("[" + each_section + "]")
    for (each_key, each_val) in mergeconfig.items(each_section):
        print(each_key + ":  " + each_val)

# save list of IFs, Fs's
IF = [i[1] for i in mergeconfig.items(mergeconfig.sections()[0])]
IFnames = [i[0] for i in mergeconfig.items(mergeconfig.sections()[0])]
Fs = [i[1] for i in mergeconfig.items(mergeconfig.sections()[1])]

# throw error if number of IF in config file != number of IF sub directories
     in argv[1]
if len(sub_dirs) != len(mergeconfig.items(mergeconfig.sections()[0])):
    sys.exit('Error: number of IF in config file: ' + str(len(
        mergeconfig.items(mergeconfig.sections()[0]))) +
             ' != number of IF sub directories in argv[1]: ' + str(len(
                 sub_dirs)))

# import all hdf5 files from the subdirectories
merge_files = [None] * len(sub_dirs)
i = 0
for each_dir in sub_dirs:
    merged_file = None
    merge_files[i] = [f for f in listdir(each_dir) if isfile(join(each_dir,
        f))]
    for each_file in merge_files[i]:
        if not each_file.endswith('.hdf5'):
            merge_files[i].remove(each_file)
    # modulate each file to be merged
```

```
    print('\n/////////// Directory: ' + str(each_dir) + ' /////////')
    j = 0
    for each_file in merge_files[i]:
        print('\n##### Target file: ' + str(each_file) + ' #####')
        print('HDF5 labels:')
        each_file = h5py.File(str(each_dir) + '/' + str(each_file), "r")

        # the order of directories isn't always the order of config values.
            Searches for subdirectory matching
        # the name of the key string in the config file
        freq_idx = IFnames.index(each_dir.split('/')[len(each_dir.split('/')
            )-1])

        for item in each_file.attrs.keys():
            print(item + ":", each_file.attrs[item])
        sps = each_file.attrs['oversampling_factor']
        each_file = modulate(each_file, Fs[freq_idx], IF[freq_idx])
        merge_files[i][j] = each_file
        j = j + 1
    i = i + 1

# Sum all datasets from each directory after modulation
merged_data = sum_files(merge_files)

Fs = int(sys.argv[4])  # sampling frequency of receiving radio

plt.magnitude_spectrum(merged_data, Fs=Fs)
plt.title("Multi-transmission Data-Set Magnitude Response")
plt.show()

# writing each hdf5 data set here
outfile = h5py.File('./merged_datasets/merged_set' + sys.argv[3] + '.hdf5',
    "w")
# point to the default data to be plotted
outfile.attrs["default"] = "/out/bbIQ"  # group name/data set name
# give the HDF5 root some more attributes, keeping group attributes same as
    input files (mod scheme, sps)
```

```
    outfile.attrs["creator"] = sys.argv[0]
    outfile.attrs["HDF5_Version"] = h5py.version.hdf5_version
    outfile.attrs["h5py_version"] = h5py.version.version

    # initializing the hdf5 group that each data set will be written to
    grp = outfile.create_group("out")
    # file is actually written and closed
    grp.create_dataset("bbIQ", data=merged_data)
    outfile.close()
```

A.4 upsampFilt.py

```
#!/usr/bin/python3

# kwmcclintick
# wilab

from pylab import *
import sys
import filters
import matplotlib.pyplot as plt
import numpy as np
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
import h5py
warnings.resetwarnings()

if len(sys.argv) < 5:
    print("Usage: ./program {# of Symbols to make, SPS}")
    print("Exiting...")
    quit()
else:
    filename = sys.argv[1]
    infile = h5py.File(filename, "r")
    inputData = infile[infile.attrs['default']]
```

```
    rolloff_factor = float(sys.argv[2])
    span = int(sys.argv[3])
    sps = int(sys.argv[4])

print('\n----------------------UPSAMPFILT----------------------')
# separate inputData into real and imaginary parts
I = [None] * len(inputData)
Q = [None] * len(inputData)
for i in range(len(inputData)):
    I[i] = inputData[i].real
    Q[i] = inputData[i].imag

# interpolate
print("INTERPOLATING FILE BY: X" + str(sps))
N = sps  # up sampling factor, typed from float to int
interpolated_inputData_I = [0] * (len(I) * N)
interpolated_inputData_Q = [0] * (len(Q) * N)
for i in range(len(interpolated_inputData_I)):
    if i % N == 0:
        interpolated_inputData_I[i] = I[int(i / N)]
        interpolated_inputData_Q[i] = Q[int(i / N)]

# don't even ask, had to do some weird stuff to equate this filter class's
    rrcosfilter() function to MATLAB's
# wonderful rcosdesign() function
Ts = 10
Fs = N / 10
rrc_length = int(span * N + 2)  # filter length in samples, typed from float to
    int
# root raised cosine filter, h_rrc is the impulse response of the filter,
    time_idx the time values of h_rrc
time_idx, h_rrc = filters.rrcosfilter(rrc_length, rolloff_factor, Ts, Fs)
h_rrc = np.delete(h_rrc, 0)

# envelope is achieved by convolving the filters impulse response with our
    interpolated data
print("PULSE SHAPING FILE: rolloff: " + str(rolloff_factor) + ", sps: " + str(
```

```
    sps) + ", span: " + str(span))
filtered_inputData_I = np.convolve(interpolated_inputData_I, h_rrc, mode='full')
filtered_inputData_Q = np.convolve(interpolated_inputData_Q, h_rrc, mode='full')

#grpDelay = math.floor(len(h_rrc)/2)
outdata = [None] * int(len(filtered_inputData_I))  #- 2*grpDelay))
for i in range(len(outdata)):
    outdata[i] = complex(filtered_inputData_I[i], filtered_inputData_Q[i])

print(sys.argv[0][:2])
if sys.argv[0][:2] == "./":
    filestem = sys.argv[0][2:]
else:
    filestem = sys.argv[0]

filename = filestem[:-2] + "hdf5"
print(filename)

outfile = h5py.File(filename, "w")

I = [None] * len(outdata)
Q = [None] * len(outdata)
for i in range(len(outdata)):
    I[i] = outdata[i].real
    Q[i] = outdata[i].imag

plt.plot(I, Q, linewidth=0.1)
plt.title("8SPS Pulse Shaped Input I/Q")
plt.xlabel("Real")
plt.ylabel("Imaginary")
# plt.show()

# point to the default data to be plotted
outfile.attrs["default"]          = "/source_data/bbIQ"
# give the HDF5 root some more attributes
outfile.attrs["file_name"]        = filename
outfile.attrs["creator"]          = filestem
```

```
outfile.attrs["HDF5_Version"]     = h5py.version.hdf5_version
outfile.attrs["h5py_version"]     = h5py.version.version

dgroup = outfile.create_group("source_data")
dgroup.attrs["modulation"] = "QPSK"
dgroup.attrs["oversampling_factor"] = sps
dgroup.attrs["Beta"] = rolloff_factor
dgroup.create_dataset("bbIQ", data=outdata)

outfile.close()
```

A.5 Filtdownsamp.py

```
#!/usr/bin/python3

# kwmcclintick
# wilab

import sys
import filters
import math
import matplotlib.pyplot as plt
import numpy as np
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
import h5py
warnings.resetwarnings()

if len(sys.argv) < 5:
    print("Usage: ./program {# of Symbols to make, SPS}")
    print("Exiting...")
    quit()
else:
    filename = sys.argv[1]
    infile = h5py.File(filename, "r")
```

```
    inputData = infile[infile.attrs['default']]
    rolloff_factor = float(sys.argv[2])
    span = int(sys.argv[3])
    sps = int(sys.argv[4])

print('-------------------FILTDOWNSAMP-------------------')

# separate inputData into real and imaginary parts
I = [None] * len(inputData)
Q = [None] * len(inputData)
for i in range(len(inputData)):
    I[i] = inputData[i].real
    Q[i] = inputData[i].imag

# don't even ask, had to do some weird stuff to equate this filter class's
    rrcosfilter() function to MATLAB's
# wonderful rcosdesign() function
N = sps  # up sampling factor, typed from float to int
Ts = 10
Fs = N / 10
rrc_length = int(span * N + 2)  # filter length in samples, typed from float to
    int
# root raised cosine filter, h_rrc is the impulse response of the filter,
    time_idx the time values of h_rrc
time_idx, h_rrc = filters.rrcosfilter(rrc_length, rolloff_factor, Ts, Fs)
h_rrc = np.delete(h_rrc, 0)

print("MATCHED FILTERING FILE: rolloff: " + str(rolloff_factor) + ", sps: " +
    str(sps) + ", span: " + str(span))

# envelope is achieved by convolving the filters impulse response with our
    interpolated data
filtered_inputData_I = np.convolve(I, h_rrc, mode='full')
filtered_inputData_Q = np.convolve(Q, h_rrc, mode='full')

print("DECIMATING FILE BY: X" + str(sps))
# decimate
```

```
decimated_I = [0] * int((len(filtered_inputData_I) / N) - len(h_rrc))
decimated_Q = [0] * int((len(filtered_inputData_I) / N) - len(h_rrc))
for i in range(len(decimated_I)):
    decimation_index = int(math.floor(len(h_rrc) / 2) + (i * N))
    decimated_I[i] = filtered_inputData_I[decimation_index]
    decimated_Q[i] = filtered_inputData_Q[decimation_index]

outdata = [None] * len(decimated_Q)
for i in range(len(decimated_Q)):
    outdata[i] = complex(decimated_I[i], decimated_Q[i])

print(sys.argv[0][:2])
if sys.argv[0][:2] == "./":
    filestem = sys.argv[0][2:]
else:
    filestem = sys.argv[0]

filename = filestem[:-2] + "hdf5"
print(filename)

outfile = h5py.File(filename, "w")

I = [None] * len(outdata)
Q = [None] * len(outdata)
for i in range(len(outdata)):
    I[i] = outdata[i].real
    Q[i] = outdata[i].imag

plt.scatter(I,Q)
plt.title("Match Filtered, Downsampled I/Q")
plt.xlabel("Real")
plt.ylabel("Imaginary")
plt.show()

# point to the default data to be plotted
outfile.attrs["default"]          = "/source_data/bbIQ"
# give the HDF5 root some more attributes
```

```
outfile.attrs["file_name"]        = filename
outfile.attrs["creator"]          = filestem
outfile.attrs["HDF5_Version"]     = h5py.version.hdf5_version
outfile.attrs["h5py_version"]     = h5py.version.version

dgroup = outfile.create_group("source_data")
dgroup.attrs["modulation"] = "QPSK"
dgroup.attrs["oversampling_factor"] = sps
dgroup.attrs["Beta"] = rolloff_factor
dgroup.create_dataset("bbIQ", data=outdata)

outfile.close()
```

A.6 Channel.py

```
"""
Created on Sep 22, 2017
@author: kwmcclintick

The Channel parent class contains methods and instance variables that all
    wireless channel
blocks should share. Any wireless channel should of course have a name so it can
     be identified,
inputs and outputs (as any system has), a README so insight can be gained about
    the class without
diving into it's .py file, and characteristics with which the channel can apply
    to the input in
its computeOutput method.
"""

import numpy as np


class Channel:
```

```
def __init__(self, name, characteristics, infile):  # constructor
    self.characteristic_names = [None]
    self.name = name
    self.characteristics = characteristics
    self.infile = infile
    self.inputData = np.array(list(self.infile[self.infile.attrs['default'
        ]]), dtype=np.complex)
    self.outputData = [0] * len(self.inputData)

def getName(self):  # get method to return instance variable
    return self.name

def getInput(self):  # get method to return instance variable
    return self.inputData

def getOutput(self):  # get method to return instance variable
    return self.outputData

def getCharacteristics(self):  # get method to return instance variable
    return self.characteristics

def getCharacteristic_names(self):
    return self.characteristic_names

def README(self):  # get method to return instance variable
    return self.readme

def setInput(self, newInput):  # set method used in ChannelTool.py
    self.inputData = newInput

def setCharacteristics(self, characteristics):
    self.characteristics = characteristics

def setInfile(self, infile):
    self.infile = infile
```

A.7 AWGNFriis.py

```
"""
Created on Sep 22, 2017
@author: kwmcclintick

AWGNFriis is a channel child class that represents how an Additive White
    Gaussian Noise wireless
channel modifies information passed through it using the Friis formula and
    Boltzmann
"""

import warnings
warnings.filterwarnings("ignore", message="numpy.dtype size changed")
warnings.filterwarnings("ignore", message="numpy.ufunc size changed")
from random import gauss
from pylab import *
from Channels.Channel import Channel
import matplotlib.pyplot as plt
from scipy import signal
from pylab import plot, xlabel, ylabel, ylim, title, show


class AWGNFriis(Channel):

    def __init__(self, characteristics, infile):  # constructor
        # parent's constructor is called. Everything else in here should be
            specific to the AWGN child class
        Channel.__init__(self, type(self).__name__, characteristics, infile)

        # this is used in write_to_hdf5 to create attributes for the output data
             set
        self.characteristic_names = ["bandwidth", "temp", "Ptx", "Gtx", "Grx", "
            d", "lambda", "Lsys", "Fs"]
```

```
'''
compute_write is the function called when a new process is created.
    Everything in this function is done in parallel
with other combinations of channel characteristics

AWGN's compute output adds zero mean Gaussian distributed real and imaginary
     numbers to
the input QI data. The real and imaginary parts of the noise are
    statistically independent.
The variance and amplitude of the noise is user specified with variables var
     and SNR.

The more oversampling there is, the more the AWGN noise is reduced for each
    sample
'''
def compute(self):
    bandwidth = self.characteristics[0]
    temp = self.characteristics[1]
    Ptx = self.characteristics[2]
    Gtx = self.characteristics[3]
    Grx = self.characteristics[4]
    d = self.characteristics[5]
    lamb = self.characteristics[6]
    Lsys = self.characteristics[7]
    Fs = self.characteristics[8]
    # Friis eq, Power received in dB
    Lpath = 10*math.log10((4 * math.pi * d / lamb)**2)
    PrxdB = 10*math.log10(Ptx/0.001) + Gtx + Grx - Lpath - Lsys
    Prx = 10**(PrxdB / 10)

    Pnoise = 0.00000000245  # noise power in watts, measured by K. Gill with
         an N210 in our lab AK318
    # 450 MHz 7.5 kHz bin size
    #Pnoise = 10*math.log10(Pnoise)
    snr = Prx / Pnoise
    snrdB = 10*math.log10(snr)
```

```
        k = 1.38 * 10**-23  # Boltzmann constant
        h = 6.62 * 10**-34  # Planck constant
        R = 50  # resistance of SMA port, ohms
        variance = 2*(math.pi*k*temp)**2 / (3 * h) * R * bandwidth**0.5

        Asig_rms = 0
        sps = self.infile['source_data'].attrs['oversampling_factor']

        # here, input data is looped through to find the rms amplitude so that
            the noise amplitude can
        # appropriately be determined
        for b in range(len(self.inputData)):
            Asigcurrent = math.sqrt(pow(self.inputData[b].real, 2) + pow(
                self.inputData[b].imag, 2))
            Asig_rms += math.pow(Asigcurrent, 2)
        Asig_rms = math.sqrt((1 / (b + 1)) * Asig_rms)

        awgnNoise = [0] * len(self.inputData)
        for i in range(len(self.inputData)):
            # noise amplitude of each symbol in log scale
            awgnNoise[i] = Asig_rms * 10 ** (-snrdB / 20.0) * complex(gauss(0,
                math.sqrt(variance)),
                                                                      gauss(0,
                                                                          math.sqrt
                                                                          (variance
                                                                          )))
            # limit noise amplitude by SPS
            awgnNoise[i] = awgnNoise[i] / sps
        # band-limit the AWGN noise
        band_limited_noise = self.bandLimit(bandwidth, Fs, awgnNoise)
        # apply band-limited noise to input data
        for i in range(len(self.outputData)):
            self.outputData[i] = self.inputData[i] + band_limited_noise[i]

    '''
    Applies a 1001-order Blackman Harris Low Pass Filter to the AWGN list of
        values to keep all spectral noise in-band
```

```
with the signal to prevent machine learning tricks
'''
def bandLimit(self, bandwidth, Fs, noise):
    # plot incoming noise magnitude response
    # plt.magnitude_spectrum(noise, Fs=Fs)
    # plt.title("Noise Magnitude Response")
    # plt.show()

    n = 1001  # filter order
    half_n = int(math.floor(n/2))
    cutoff = bandwidth / Fs

    # Lowpass filter defined
    a = signal.firwin(n, cutoff, window='blackmanharris')
    band_limtited_AWGN = np.convolve(noise, a, mode='full')
    for i in range(half_n, len(noise) + half_n):
        band_limtited_AWGN[i - half_n] = band_limtited_AWGN[i]
    '''
    plot filter impulse response
    stem(a)
    title("Blackman Harris LPF Impulse Response")
    show()

    plot filter frequency response
    plt.magnitude_spectrum(band_limtited_AWGN, Fs=Fs)
    plt.title("band_limtited_AWGN Magnitude Response")
    plt.show()

    b = a  # coeffecients, numerator
    a = 1  # coeffecients, denominator
    w, h = signal.freqz(b, a)
    h_dB = 20 * log10(abs(h))
    subplot(211)
    plot(w / max(w), h_dB)
    ylim(-150, 5)
    ylabel('Magnitude (db)')
    xlabel(r'Normalized Frequency (x$\pi$rad/sample)')
```

```
        title(r'Frequency response')
        subplot(212)
        h_Phase = unwrap(arctan2(imag(h), real(h)))
        plot(w / max(w), h_Phase)
        ylabel('Phase (radians)')
        xlabel(r'Normalized Frequency (x$\pi$rad/sample)')
        title(r'Phase response')
        subplots_adjust(hspace=0.5)
        show()
        '''

        return band_limtited_AWGN
```

A.8 STOresolution.py

```
"""
Created on Nov 17, 2017
@author: kwmcclintick

Symbol Timing Offset (STO) is a channel child class that simulates timing error
    by interpolating data up the smallest
intermediate SPS needed to achieve a certain % of a symbol offset. A 4 sps
    signal, could, for instance, have an
offset of 0 (0% offset), 1 (25% offset), 2 (50% offset), or 3 (75% offset). If
    the resolution desired is 0.125,
the intermediate SPS would be 2, and 4 if the resolution desired was 0.0625, and
     so on.

the timing offset is random but constant for each transmission.
"""

import warnings
warnings.filterwarnings("ignore", message="numpy.dtype size changed")
warnings.filterwarnings("ignore", message="numpy.ufunc size changed")
from Channels.Channel import Channel
```

```
import matplotlib.pyplot as plt
import numpy as np
import math
import numpy
from numpy import abs
from scipy import signal


class STOresolution(Channel):
    def __init__(self, characteristics, infile):  # constructor
        # parent's constructor is called. Everything else in here should be
            specific to the AWGN child class
        Channel.__init__(self, type(self).__name__, characteristics, infile)

        # this is used in write_to_hdf5 to create attributes for the output data
             set
        self.characteristic_names = ["resolution"]

    '''
    compute_write is the function called when a new process is created.
        Everything in this function is done in parallel
    with other combinations of channel characteristics
    '''
    def compute(self):
        sps = self.infile['source_data'].attrs['oversampling_factor']
        resolution = float(self.characteristics[0])

        # interpolation filter defined
        # ~~[Filter Design with Parks-McClellan Remez]~~
        order = 32  # Filter order
        # Filter symetric around 0.25 (where .5 is pi or Fs/2)
        bands = numpy.array([0., .22, .28, .5])
        h = signal.remez(order + 1, bands, [1, 0], [1, 1])
        h[abs(h) <= 1e-4] = 0.
        # (w, H) = signal.freqz(h)  # quicker falloff, higher dB stop band

        # ~~[Filter Design with Windowed freq]~~
```

```
# b = signal.firwin(order + 1, 0.5)
# b[abs(h) <= 1e-4] = 0.
# (wb, Hb) = signal.freqz(b)  # slower falloff, lower dB stop band

if (1/sps) / resolution == 0:  # offset determined as in STOauto.py if
    sps is sufficient to achieve res
    offset = round(np.random.uniform(0, sps))
    for i in range(len(self.inputData) - offset):
        self.outputData[i] = self.inputData[i + offset]
else:
    counter = sps
    # data is temp upsampled and filtered with an interpolation filter
        then downsampled such that the new
    # sps is a multiple of the original and the resolution achieved is
        less than or equal to the desired
    while (1/counter) > resolution or counter % sps != 0:
        counter += 1

    N = int(counter/sps)  # our temp interpolation decimation rate
    iteration = N  # if N = 8, this loop will interp by 2, filter 3
        times, to get a total 2*2*2 = 8
    N = 2  # each iteration will interpolate by 2

    # separate inputData into real and imaginary parts
    I = [None] * len(self.inputData)
    Q = [None] * len(self.inputData)
    for i in range(len(self.inputData)):
        I[i] = self.inputData[i].real
        Q[i] = self.inputData[i].imag

    # plt.stem(I)
    # plt.title("input pulse shaped data")
    # plt.xlabel("sample #")
    # plt.ylabel("amplitdue")
    # plt.show()

    # interpolate
```

```
interpolated_inputData_I = [0] * (len(I) * N)
interpolated_inputData_Q = [0] * (len(Q) * N)
for i in range(len(interpolated_inputData_I)):
    if i % N == 0:
        interpolated_inputData_I[i] = I[int(i / N)]
        interpolated_inputData_Q[i] = Q[int(i / N)]

# plt.stem(interpolated_inputData_I)
# plt.title("interpolated_inputData_I")
# plt.xlabel("sample #")
# plt.ylabel("amplitdue")
# plt.show()

# envelope is achieved by convolving the filters impulse response
    with our interpolated data
filtered_inputData_I = np.convolve(interpolated_inputData_I, h, mode
    ='full')
filtered_inputData_Q = np.convolve(interpolated_inputData_Q, h, mode
    ='full')

# plt.stem(filtered_inputData_I)
# plt.title("filtered_inputData_I")
# plt.xlabel("sample #")
# plt.ylabel("amplitdue")
# plt.show()

# interpolate and filter until the desired resolution is achieved
iteration = iteration / 2
while iteration != 1:
    # interpolate again
    interpolated_inputData_I = [0] * (len(filtered_inputData_I) * N)
    interpolated_inputData_Q = [0] * (len(filtered_inputData_Q) * N)
    for i in range(len(interpolated_inputData_I)):
        if i % N == 0:
            interpolated_inputData_I[i] = filtered_inputData_I[int(i
                / N)]
            interpolated_inputData_Q[i] = filtered_inputData_Q[int(i
```

```
                / N)]

    # filter again
    filtered_inputData_I = np.convolve(interpolated_inputData_I, h,
        mode='full')
    filtered_inputData_Q = np.convolve(interpolated_inputData_Q, h,
        mode='full')

    iteration = iteration / 2  # iteration is changed to signal a 2x
         interp and filtering loop

# apply symbol timing offset
offset = np.random.randint(0, sps * int(counter/sps) - 1)
for i in range(len(filtered_inputData_I) - offset):
    filtered_inputData_I[i] = filtered_inputData_I[i + offset]
    filtered_inputData_Q[i] = filtered_inputData_Q[i + offset]

# plt.stem(filtered_inputData_I)
# plt.title("offset filtered_inputData_I by: +" + str(offset) + "
    samp")
# plt.xlabel("sample #")
# plt.ylabel("amplitdue")
# plt.show()

N = int(counter / sps)  # our temp interpolation decimation rate
for i in range(len(self.outputData)):
    decimation_index = int(math.log(N, 2)*math.floor(len(h) / 2) + (
        i * N))
    # decimated values are scaled according to the interpolation
        filter's max amplitude and the number
    # of times it was applied
    self.outputData[i] = complex((1/max(h))*math.log(N, 2)*
        filtered_inputData_I[decimation_index],
                                 (1 / max(h)) * math.log(N, 2)*
                                     filtered_inputData_Q[
                                     decimation_index])
```

```
            # plt.stem(decimatied_I, 'r-')
            # plt.stem(I)
            # plt.title("decimatied_I (red) vs. input I")
            # plt.xlabel("sample #")
            # plt.ylabel("amplitdue")
            # plt.show()
```

A.9 RandomInitialPhase.py

```
"""
Created on Nov 14, 2017
@author: kwmcclintick

RandomInitialPhase is a channel child class that represents how a wireless
    transmitter can randomize the phase of
transmitted IQ data, since its exact distance as a multiple of carrier
    wavelength from the receiver is not
always considered.

The only difference between RandomInitialPhase and ClarkeScattering is that
    RandomInitialPhase applies the same
random phase to every symbol, while ClarkeScattering applies a different random
    phase to each symbol
"""

from Channels.Channel import Channel
import numpy as np
import math

class RandomInitialPhase(Channel):
    def __init__(self, characteristics, infile):  # constructor
        # parent's constructor is called. Everything else in here should be
            specific to the AWGN child class
        Channel.__init__(self, type(self).__name__, characteristics, infile)
```

```
    '''
    compute_write is the function called when a new process is created.
        Everything in this function is done in parallel
    with other combinations of channel characteristics
    '''
    def compute(self):
        theta = np.random.uniform(0, 2 * math.pi)
        phase = complex(math.cos(theta), math.sin(theta))
        for i in range(len(self.inputData)):
            self.outputData[i] = phase * self.inputData[i]
```

A.10 IQimbalance.py

```
"""
Created on Nov 14, 2017
@author: kwmcclintick

IQimbalance is a channel child class that damages IQ symbols due to internal
    noise from an RF frontend. The idea
stems from the fact that the I and Q branches of an analog system will not have
    the same phase and magnitude as each
other.

This type of balance is assumed to be caused by a MIXER MISMATCH in a DIRECT UP
    transmit chain, and
is CONSTANT WITH RESPECT TO FREQUENCY

kI: in-phase magnitude, linear. ex: 0dB would give kI = 1
kQ: quadrature magnitude, linear.
phi_epsilon: phase difference, degrees

The channel is described by the matrix equation below

[sI] = [      kI               0      ] [sI']
[sQ]   [-kQsin(phi_ep)   kQcos(phi_ep)] [sQ']
```

```
where sI' and sQ' are the input IQ waveforms and sI and sQ are the output IQ
    waveforms after being damaged

this relationship should make it clear that a balanced system would have kI=kQ=1
     and phi_epsilon=0, as sI=sI' and sQ=sQ'
"""

from Channels.Channel import Channel
import numpy


class IQimbalance(Channel):
    def __init__(self, characteristics, infile):  # constructor
        # parent's constructor is called. Everything else in here should be
            specific to the AWGN child class
        Channel.__init__(self, type(self).__name__, characteristics, infile)

        # this is used in write_to_hdf5 to create attributes for the output data
             set
        self.characteristic_names = ["k_I", "k_Q", "phi_epsilon"]

    '''
    compute_write is the function called when a new process is created.
        Everything in this function is done in parallel
    with other combinations of channel characteristics
    '''
    def compute(self):
        self.characteristics[2] = self.characteristics[2] * numpy.pi / 180.0  #
            converts phi_epsilon to radians

        for i in range(len(self.inputData)):
            self.outputData[i] = complex(self.characteristics[0]*self.inputData[
                i].real,
                                         self.inputData[i].imag*
                                             self.characteristics[1]*numpy.cos(
                                             self.characteristics[2])
```

```
                                        - self.inputData[i].real*
                                            self.characteristics[1]*numpy.sin(
                                            self.characteristics[2]))
```

A.11 CFO.py

```
"""
Created on Nov 17, 2017
@author: kwmcclintick

Carrier Frequency Offset (CFO) is a channel child class that simulates small
    errors in the carrier frequency. The result
is each sample is rotated by a small phase value that builds with each sample,
    resulting in a spinning effect over
time of the IQ data based on the term fo/Fs

CFO.py is different than CFOSymRate.py in that CFO.py tries to model two
    independent Local Oscillators. A random
frequency offset is generated, and kept constant for each transmission.

A more accurate version of this model would include drift, varying the frequency
     offset even within each small
transmission
"""

from Channels.Channel import Channel
import math, random


class CFO(Channel):
    def __init__(self, characteristics, infile):  # constructor
        # parent's constructor is called. Everything else in here should be
            specific to the AWGN child class
        Channel.__init__(self, type(self).__name__, characteristics, infile)
```

```
    # this is used in write_to_hdf5 to create attributes for the output data
        set
    self.characteristic_names = ["Fc", "Fs", "PPM"]

'''
compute_write is the function called when a new process is created.
   Everything in this function is done in parallel
with other combinations of channel characteristics
'''
def compute(self):
    Fc = self.characteristics[0]
    Fs = self.characteristics[1]
    PPM = self.characteristics[2]
    sps = self.infile['source_data'].attrs['oversampling_factor']

    # Tx and Rx both randomly choose an offset for each transmission of mean
        fc and std dev fo,max
    fo_max = PPM * Fc / 10**6
    std_dev = fo_max
    # bounded by fo,max, offset is re-picked if too large or too small
    fo_tx = random.gauss(0, std_dev)
    fo_rx = random.gauss(0, std_dev)
    while fo_rx < -fo_max or fo_rx > fo_max:
        fo_rx = random.gauss(0, std_dev)
    while fo_tx < -fo_max or fo_tx > fo_max:
        fo_tx = random.gauss(0, std_dev)

    # sum the offsets, worst case to_rx = to_tx = fo_max or -fo_max, best
       case both equal zero or opposite signs
    fo_total = fo_tx + fo_rx

    phase_per_sample = (2 * math.pi * fo_total / Fs) / sps

    for i in range(len(self.inputData)):  # phase grows with each symbol,
       spinning the constellation with time
        phase = phase_per_sample * (i + 1)
        rotation = complex(math.cos(phase), math.sin(phase))
```

```
            self.outputData[i] = rotation * self.inputData[i]
```

A.12 filters.py

```
# Authors: Veeresh Taranalli <veeresht@gmail.com>
# License: BSD 3-Clause

"""
===============================================
Pulse Shaping Filters (:mod:`commpy.filters`)
===============================================

.. autosummary::
   :toctree: generated/

   rcosfilter          -- Raised Cosine (RC) Filter.
   rrcosfilter         -- Root Raised Cosine (RRC) Filter.
   gaussianfilter      -- Gaussian Filter.
   rectfilter          -- Rectangular Filter.

"""

import numpy as np

__all__=['rcosfilter', 'rrcosfilter', 'gaussianfilter', 'rectfilter']

def rcosfilter(N, alpha, Ts, Fs):
    """
    Generates a raised cosine (RC) filter (FIR) impulse response.

    Parameters
    ----------
    N : int
        Length of the filter in samples.
```

```
alpha : float
    Roll off factor (Valid values are [0, 1]).

Ts : float
    Symbol period in seconds.

Fs : float
    Sampling Rate in Hz.

Returns
-------

h_rc : 1-D ndarray (float)
    Impulse response of the raised cosine filter.

time_idx : 1-D ndarray (float)
    Array containing the time indices, in seconds, for the impulse response.
"""

T_delta = 1/float(Fs)
time_idx = ((np.arange(N)-N/2))*T_delta
sample_num = np.arange(N)
h_rc = np.zeros(N, dtype=float)

for x in sample_num:
    t = (x-N/2)*T_delta
    if t == 0.0:
        h_rc[x] = 1.0
    elif alpha != 0 and t == Ts/(2*alpha):
        h_rc[x] = (np.pi/4)*(np.sin(np.pi*t/Ts)/(np.pi*t/Ts))
    elif alpha != 0 and t == -Ts/(2*alpha):
        h_rc[x] = (np.pi/4)*(np.sin(np.pi*t/Ts)/(np.pi*t/Ts))
    else:
        h_rc[x] = (np.sin(np.pi*t/Ts)/(np.pi*t/Ts))* \
                (np.cos(np.pi*alpha*t/Ts)/(1-(((2*alpha*t)/Ts)*((2*alpha*t)/
                    Ts))))
```

```
    return time_idx, h_rc

def rrcosfilter(N, alpha, Ts, Fs):
    """
    Generates a root raised cosine (RRC) filter (FIR) impulse response.

    Parameters
    ----------
    N : int
        Length of the filter in samples.

    alpha : float
        Roll off factor (Valid values are [0, 1]).

    Ts : float
        Symbol period in seconds.

    Fs : float
        Sampling Rate in Hz.

    Returns
    ---------

    h_rrc : 1-D ndarray of floats
        Impulse response of the root raised cosine filter.

    time_idx : 1-D ndarray of floats
        Array containing the time indices, in seconds, for
        the impulse response.
    """

    T_delta = 1/float(Fs)
    time_idx = ((np.arange(N)-N/2))*T_delta
    sample_num = np.arange(N)
    h_rrc = np.zeros(N, dtype=float)
```

```
    for x in sample_num:
        t = (x-N/2)*T_delta
        if t == 0.0:
            h_rrc[x] = 1.0 - alpha + (4*alpha/np.pi)
        elif alpha != 0 and t == Ts/(4*alpha):
            h_rrc[x] = (alpha/np.sqrt(2))*(((1+2/np.pi)* \
                    (np.sin(np.pi/(4*alpha)))) + ((1-2/np.pi)*(np.cos(np.pi/(4*
                        alpha)))))
        elif alpha != 0 and t == -Ts/(4*alpha):
            h_rrc[x] = (alpha/np.sqrt(2))*(((1+2/np.pi)* \
                    (np.sin(np.pi/(4*alpha)))) + ((1-2/np.pi)*(np.cos(np.pi/(4*
                        alpha)))))
        else:
            h_rrc[x] = (np.sin(np.pi*t*(1-alpha)/Ts) +  \
                    4*alpha*(t/Ts)*np.cos(np.pi*t*(1+alpha)/Ts))/ \
                    (np.pi*t*(1-(4*alpha*t/Ts)*(4*alpha*t/Ts))/Ts)

    return time_idx, h_rrc

def gaussianfilter(N, alpha, Ts, Fs):
    """
    Generates a gaussian filter (FIR) impulse response.

    Parameters
    ----------

    N : int
        Length of the filter in samples.

    alpha : float
        Roll off factor (Valid values are [0, 1]).

    Ts : float
        Symbol period in seconds.

    Fs : float
        Sampling Rate in Hz.
```

```
    Returns
    -------

    h_gaussian : 1-D ndarray of floats
        Impulse response of the gaussian filter.

    time_index : 1-D ndarray of floats
        Array containing the time indices for the impulse response.
    """

    T_delta = 1/float(Fs)
    time_idx = ((np.arange(N)-N/2))*T_delta
    h_gaussian = (np.sqrt(np.pi)/alpha)*np.exp(-((np.pi*time_idx/alpha)*(np.pi*
        time_idx/alpha)))

    return time_idx, h_gaussian

def rectfilter(N, Ts, Fs):
    """
    Generates a rectangular filter (FIR) impulse response.

    Parameters
    ----------

    N : int
        Length of the filter in samples.

    Ts : float
        Symbol period in seconds.

    Fs : float
        Sampling Rate in Hz.

    Returns
    -------
```

```
    h_rect : 1-D ndarray of floats
        Impulse response of the rectangular filter.

    time_index : 1-D ndarray of floats
        Array containing the time indices for the impulse response.
    """

    h_rect = np.ones(N)
    T_delta = 1/float(Fs)
    time_idx = ((np.arange(N)-N/2))*T_delta

    return time_idx, h_rect
```

A.13 ChannelConfig_ettusN210.ini

```
[CHANNELS]
;Ettus N210 SBX daughterboard SISO Tx/Rx open field rural transmission dataset,
only themost impactful channel effects are considered for this basic example
;The following channel effects are ignored: IQ imbalance, multipath, doppler,
amplifier non-linearities, interference from weather or other users, quantization
error, floating point error in filter coeffecients
channel1 = STOresolution
channel2 = AWGNFriis
channel3 = CFO
channel4 = RandomInitialPhase

[CHARACTERISTICS]
characteristics1 = 0.0625
;offset resolution where 100% offset at N SPS means offset = N samples
characteristics2 = 4800000|298.15|0.001|5.19|5.19|500|0.33253893476|0.5|50000000
;4.8 MHz BW|temp = 77 degrees fahr, 298.15 degrees kelvin, about room temp|ptx=0 dBm|
gtx=5.19 dBi|grx=5.19 dBi, typical quarter-monopole|d=500 meters|lambda=c/f,
f = 902.15 MHz|Lsys=0.5 dB|Fs=50MHz
characteristics3 = 902150000|25000000|2.5
;center frequency=902.15 MHz|samp freq = 25 MS/s|LO PPM=2.5
```

```
characteristics4 = 0
;parameterless
```

A.14 merge_config_3IFs.ini

```
[INTERMEDIATE FREQUENCIES]

IF1=15000000
IF2=10000000
IF3=20000000

[SAMPLING FREQUENCIES]

Fs1=50000000
Fs2=50000000
Fs3=50000000
```