THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

Learning an End-to-End Physical Layer with Computational Graphs

by
Caleb Zulawski

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

September 12, 2017

Professor Sam Keene, Advisor

THE COOPER UNION FOR THE
ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

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

Dean, School of Engineering Date

Prof. Sam Keene, Advisor Date

Abstract

This thesis presents a novel method for designing a modulation scheme using artificial neural networks. This method models the communication system as a low complexity autoencoder which learns a modulation scheme. The learned modulation scheme is compared to communication systems using Hamming and convolutional codes with binary phase-shift keying (BPSK) and is shown to be competitive with these traditional communication systems in both bit error rate and computational complexity. Proving the capability of the autoencoder modem reveals a promising future of deep learning communication systems that do not require error correction coding.

Foreword

I would first like to thank Sam Keene, whose support and guidance has extended far beyond his advisement on this thesis and allowed my interest in these topics to flourish.

I would next like to thank the rest of the faculty for their dedication to this institution and its students.

I would like to thank Chris Curro, as a friend and fellow classmate, but more recently as an educator and additional source of support and guidance for this thesis.

I would also like to thank all of my other friends and classmates at The Cooper Union for making these six years memorable (and often entertaining).

Finally, I would like to thank my family for their steadfast love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The search for faster and more reliable telecommunications is a major field of research, driven by both commercial and governmental interests. With the relatively recent widespread adoption of turbo codes [1], telecommunications is able to reach bandwidth near the theoretical maximum. With these improvements, however, has come increased complexity and further abstraction between the physical layer modulation and the techniques for guaranteeing reliable data delivery, such as error-correcting codes.

The need for bandwidth has risen simultaneously with, and perhaps due to, the computing power available with modern hardware. This availability of computing power has led to an explosion in the capability of machine learning, specifically neural networks. The bulk of research into artificial neural networks in the last decade has been focused primarily on convolutional neural networks for image-related tasks such as object recognition and image generation. Though other topics do see some research, surprisingly little work has been done with telecommunications and machine learning, and even less on the front of designing new communication systems with neural networks.

In this thesis, I will explore the design of modems with neural networks. The goal is to create an autoencoder modem that is competitive with modern communication systems. This thesis begins with an explanation of relevant aspects of both telecommunications and machine learning that are necessary for the following chapters. With this foundation in place, I then introduce the autoencoder modem, a previous approach to the problem. Finally, I present novel improvements to the design and usage of the autoencoder modem and evaluate its performance.

Chapter 2

Background

2.1 Telecommunications

Telecommunication is the transmission of information in any form through the use of technology. The field of telecommunications has traditionally been focused on radio technologies but optical, acoustic, and many other technologies have seen use.

2.1.1 Digital communication systems

Digital communication systems, which transmit digital information rather than analog, have become the norm for communication systems. Digital telecommunication allows a wider variety of information to be transmitted with the same system, since text, voice, video, and many other forms of information can be represented digitally. Digital communication systems are

Layer	Role
Application	sharing data between its end use and the network
Presentation	converting data between its transmitted and original form
Session	maintaining state between two nodes that are in continuous communication
Transport	reliable transmission between nodes that may not be adjacent in the network
Network	handling connections between many nodes
Data link	reliable transmission between two adjacent nodes
Physical	bit transmission over the physical medium

Table 2.1: The OSI model, in order from its highest to lowest layer

also generally very complex, so they are often abstracted into several *layers*. Each layer has a unique role in the communication system; in theory any particular implementation of a layer can be replaced with a different implementation without effecting the other layers. The layers most commonly used are defined in the OSI model [2]. Table 2.1 lists each layer of the OSI model and its role.

The application layer is considered the highest layer, or most abstracted from the communication system. A web browser or mobile phone calling application interfaces with the application layer. The physical layer, on the other hand, is the lowest and least abstracted layer. The physical layer is responsible for transmitting the data as a bit stream over the physical medium, such as a radio frequency or fiber optic cable. In this thesis we are primarily concerned with optimizing the physical layer.

2.1.2 Modulation

Most physical layer implementations modulate, or modify, a carrier signal to transmit information [3]. The carrier signal is a periodic, usually sinusoidal, waveform. Its periodic property allows a demodulator to efficiently recover the information added to the carrier with modulation. Early forms of modulation include amplitude modulation (AM) and frequency modulation (FM), which modulate the amplitude and frequency, respectively, of the carrier signal. Most modulation schemes today modify the phase and amplitude of the carrier signal. The modulating signal, called the *baseband* signal, is often represented as a complex signal which includes the phase and amplitude information. The real component of the baseband signal is also called the *in-phase*, or *I*, component, and the imaginary component is called the *quadrature*, or *Q*, component. To modulate the carrier, the baseband signal is multiplied with the carrier signal in a process known as frequency mixing, or just mixing.

Modern communication systems use digital modulation, which modulates the analog carrier with a sampled digital baseband signal which represents a bit stream. The simplest method of digital modulation is with a modulation scheme known as binary phase-shift keying. The premise is simple; when the carrier is untouched, a 0 bit is transmitted. When the phase is shifted by 180°, a 1 bit is transmitted. The associated baseband signal for BPSK is therefore 1 for transmitting a 0 bit, and -1 for transmitting a 1 bit.

Higher order modulation can transmit more bits per baseband sample.

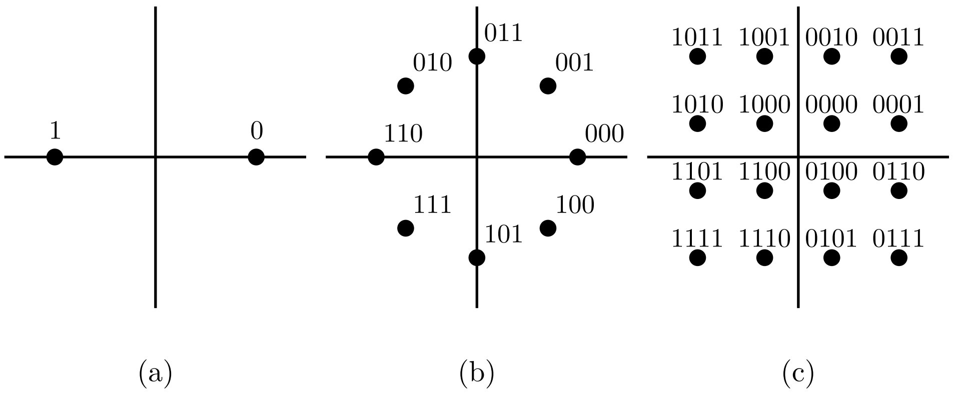


Figure 2.1: Constellation diagrams for (a) BPSK, (b) 8-PSK, and (c) 16-QAM.

Phase-shift keying (PSK) can be adapted to use more phase shifts to include more information. Another popular modulation scheme, quadrature amplitude modulation (QAM), modulates the amplitude of the in-phase and quadrature components separately. These unique transmitted waveforms containing some number of bits is called a *symbol*. A convenient way of visualizing modulation schemes is by plotting their symbols on a complex graph, which is called a constellation diagram. Figure 2.1 shows constellation diagrams for several modulation schemes.

2.1.3 The noisy channel coding theorem

Demodulation is usually accomplished by determining which symbol is closest to the received waveform. Communication systems are not expected to work in ideal, noise-free conditions, however, and must account for corrupted received symbols. The noise present in communication channels is typically modeled as *additive white Gaussian noise* (AWGN). AWGN is added to the transmitted waveform (additive), has constant power across all frequencies (white), and is normally distributed (Gaussian).

Claude Shannon devised a simple theorem to describe reliable communication in the presence of noise [3]:

> Reliable communication over a discrete memoryless channel is possible if the communication rate R satisfies $R < C$, where C is the channel capacity. At rates higher than capacity, reliable communication is impossible. This threshold is known as the *Shannon limit*.

A discrete memoryless channel refers to a channel for digital information that does not have any memory of past transmitted information, leaving noise as the primary corrupting factor. Channel capacity for a channel subject to AWGN is defined as:

$$C = B \log_2 \left(1 + \frac{S}{N}\right) \tag{2.1}$$

where B is the bandwidth of the channel in Hz and S/N is the SNR as a ratio of powers, leaving the channel capacity C in the form of bits/second. Error-free communication is possible at any rate that is less than this channel capacity, but impossible otherwise.

No communication system has ever reached this theoretical limit. Only recently have systems approached this threshold by utilizing error-correcting codes such as turbo codes. Turbo codes are capable of performance within

1 dB SNR of the Shannon limit [1].

2.1.4 Avoiding bit errors with codes

In general, codes manipulate the bit stream before modulation and transmission to improve the performance of the communication system. Error-correcting codes add redundant data in order to recover from bit errors. From the perspective of the noisy channel coding theorem, error-correcting codes reduce the rate of information transmission to below the Shannon limit.

Gray code

Due to the noise's Gaussian distribution, it is more likely for a symbol to be corrupted to an adjacent symbol. The first step in avoiding bit errors is by numbering symbols with the *Gray code* [4]. In the Gray code, each adjacent number changes by only a single bit. Ordering symbols by the Gray code makes it more likely that a corrupted symbol will result in only a single bit error, since adjacent symbols differ by only one bit. The constellation diagrams in Figure 2.1 have symbols that are ordered in this manner. The Gray code does not change the data rate and is not an error-correcting code.

Repetition code

The repetition code is the simplest error-correcting code, though not very efficient. A rate $1/3$ repetition code, for example, transmits each bit a total of three times. The decoder uses a simple voting algorithm on each encoded block to determine which bit is most likely. The rate 1/3 repetition code is capable of correcting one bit error in every three bit block.

Hamming code

The rate 1/3 repetition code is a member of a generalized class of codes called Hamming codes [5]. Hamming codes are a type of *linear block code* that can correct a single bit error per block at the maximum possible rate for the block size [3]. A linear block code is encoded block by block, rather than on a continuous bit stream. Additionally, every linear combination of encoded blocks, or *codewords*, is also a valid codeword. Linear block codes are simple to encode and decode using a *generator matrix* $\boldsymbol{G}$ and *parity check matrix* $\boldsymbol{H}$.

The Hamming(7,4) code, for example, encodes 4 bits of data into 7 bits which can be decoded with a single bit error. The code can be described by its generator and parity check matrices [3]:

$$\boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{2.2}$$

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

All operations using these matrices must be done in a finite field known as the Galois field with two elements, or GF(2), such that $1+1=0$ and $1\cdot 1=0$. To generate a codeword, multiply the data by the generator matrix:

$$\boldsymbol{c}_m = \boldsymbol{u}_m \boldsymbol{G} \tag{2.4}$$

where $\boldsymbol{u}_m$ is the message and and $\boldsymbol{c}_m$ is the codeword. The original design of the code was to append the transmitted data with three parity bits, which is clear from the generator matrix design of the identity with three columns that compute parity bits.

To decode a received message, the *syndrome* is computed:

$$\boldsymbol{s} = \boldsymbol{y}\boldsymbol{H}^{\top} \tag{2.5}$$

where $\boldsymbol{s}$ is the syndrome and $\boldsymbol{y}$ is the received message. The syndrome contains the error pattern—a particular syndrome corresponds to a particular list of bits that are flipped. Hamming codes were designed to have a very simple syndrome; it is a binary number that corresponds to which single bit is incorrect. A syndrome of 0 indicates that there is no bit error, otherwise the syndrome indicates the position of the flipped bit.

In addition to Hamming codes, there are other linear block codes with different values for $\boldsymbol{G}$ and $\boldsymbol{H}$ in common use. Reed-Solomon codes [6] are *concatenated* with, or used along with, convolutional codes to avoid burst errors [7]. Low-density parity check (LDPC) codes use very large block sizes and a sparse parity check matrix, resulting in a Shannon-limit approaching code [8].

Convolutional code

Convolutional codes [3] are another type of error-correcting code that generally outperform Hamming codes. They are capable of correcting more bit errors than Hamming codes, which are limited to correcting a single bit error per block. Convolutional codes were the best performing code before the invention of turbo codes.

Convolutional codes generate parity bits by convolving the input data with a set of generator polynomials in GF(2). This results in a code that is continuous, rather than operating on a block by block basis. The rate of the convolutional code is determined by the number of generator polynomials used as each polynomial produces a parity bit stream. The length of the generator polynomials, called the *constraint length k*, determines how many bits are used to compute each parity bit. A convolutional encoder can be implemented with a shift register, where each parity bit is generated from the current state of the shift register. An example convolutional encoder is shown in Figure 2.2.

Convolutional codes can be optimally decoded with the Viterbi algorithm [3]. To understand the Viterbi algorithm, we first consider the *trellis*, a graph that indicates the possible state transitions of the convolutional encoder. Figure 2.3 illustrates the trellis for the convolutional encoder shown in Figure

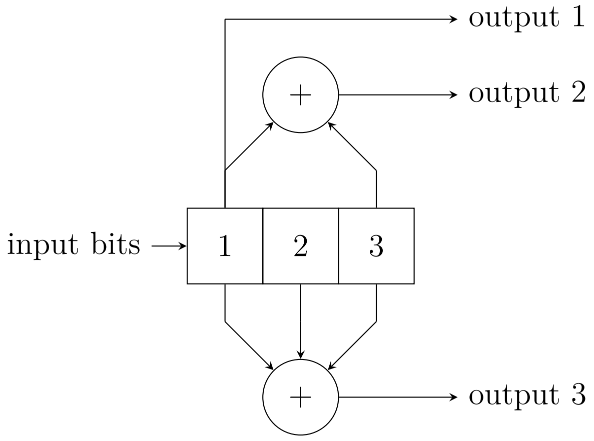


Figure 2.2: Convolutional encoder implemented with a shift register. This encoder is for a rate 1/3, $k = 3$ convolutional code with generator polynomials 100, 101, and 111.

2.2, starting at the all zero state. Each node in the trellis indicates the possible states s_1 through s_4 of the convolutional encoder at time t. The state is equivalent to the state of the last $k-1$ bits of the shift register after a new bit is shifted in. The input bit is indicated by the transition edge between nodes, with a solid line indicating a 0 and dashed indicating 1. The emitted parity bits are also shown for each transition.

Decoding a convolutional code is accomplished by finding the path from the beginning to the end of the trellis with the fewest parity bit errors. The Viterbi algorithm presents a maximum-likelihood solution to decoding a convolutional code. The Viterbi algorithm starts at $t = 0$ and determines the most likely transition by minimizing the parity bit errors. The most likely input bit and number of bit errors are stored for each state. This process is repeated sequentially for every time t, storing each transition bit and ac-

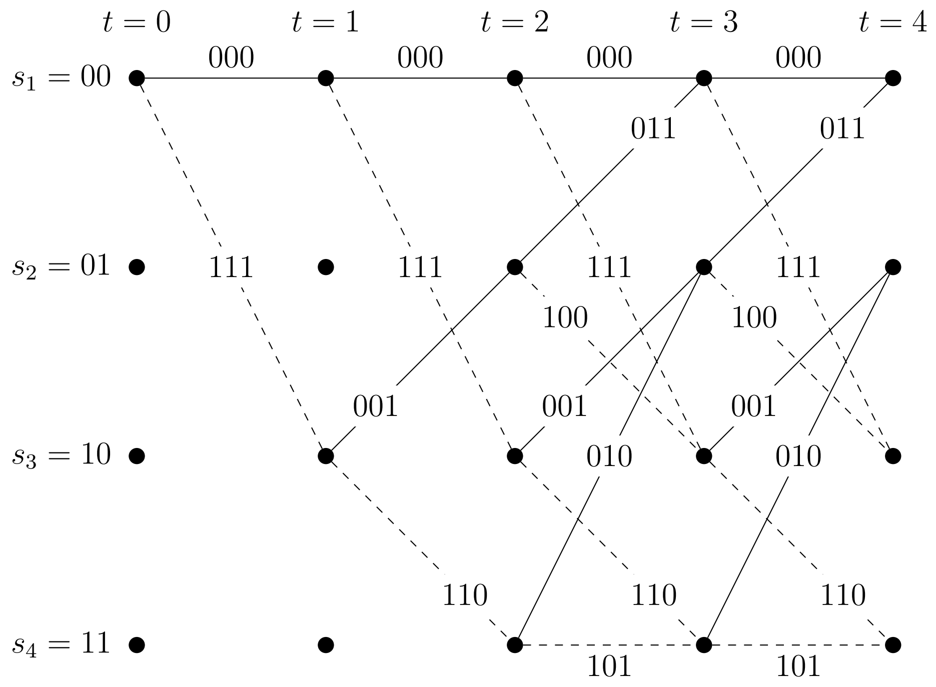


Figure 2.3: Trellis diagram for the rate 1/3, $k = 3$ convolutional code from Figure 2.2. Diagram adapted from [3].

cumulating the bit errors at each state. At each decision, the total number of parity bit errors since $t = 0$ is minimized which results in a maximum-likelihood path through the trellis. The decoded bits are determined by selecting the final state (by choosing the minimum error path in the case of truncated termination, or s_1 in the case of zero termination) and tracing through the stored input bit decisions.

2.2 Machine learning and artificial neural networks

2.2.1 Machine learning

Machine learning is a broad field of research encompassing many techniques. A machine learning algorithm is usually designed to complete a particular task, and is said to be learning if it improves its performance at that task [9]. Machine learning is often used to complete tasks that are difficult to manually design a method for, such as determining the species of a flower from its physical measurements [10] or applying a painter's artistic style to a photograph [11]. The performance metric is often both algorithm- and task-dependent. In turn, there are many machine learning algorithms and many tasks can be formulated for multiple algorithms, resulting in a nebulous field. Most machine learning algorithms, however, rely on some form of feedback to improve performance. There are three types of algorithms for each type of feedback: *supervised*, *unsupervised*, and *reinforcement* learning [9].

Supervised learning is accomplished by providing the algorithm its desired output as feedback. This is useful in scenarios where a pattern is known and a method for producing the desired output exists, but is impractical or relies on human recognition. Unsupervised learning, on the other hand, does not provide any feedback to the algorithm. This is often used for discovering patterns without any prior knowledge. The final technique, reinforcement learning, provides “rewards” or “punishments” to the algorithm as feedback.

2.2.2 Artificial neural networks

Artificial neural networks, or just neural networks, are a class of machine learning algorithms that mimic the structure of the brain in some senses [9]. The fundamental unit of a neural network is the *neuron*. Each neuron is connected to many other neurons within the neural network. More specifically, each neuron has many inputs and a single output which is used as input to other neurons. The output *activation* of a neuron j is defined as

$$a_j = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right) \tag{2.6}$$

where n is the number of input neurons, a_i is an input activation, $w_{i,j}$ is a weight which can be optimized, and g is the *activation function* [9]. It is common for one of the input activations, the *bias*, to be fixed at 1. The activation function g was traditionally the sigmoid function

$$g(x) = \frac{1}{1+e^{-x}} \tag{2.7}$$

but has largely been replaced with the rectified linear unit (ReLU)

$$g(x) = \max(0, x) \tag{2.8}$$

or variants of the ReLU such as the exponential linear unit (ELU) [12]

$$g(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha\left(e^{x} - 1\right) & \text{otherwise} \end{cases} \tag{2.9}$$

where α is a *hyperparameter*, a value that corresponds to a prior distribution and is tunable and not learned. It is also possible to have a linear activation with $g(x) = x$, which is commonly used on output neurons.

Neurons are generally organized into *layers*, where the neurons of each layer share the same inputs. The layers are then stacked to produce the entire neural network. The first and last layers are given the input data and produce the output data and are called the *input* and *output layers*, respectively. The layers between the input and output layers are not used externally to the algorithm and are called *hidden layers*. An artificial neural network with a single hidden layer is illustrated in Figure 2.4.

The calculation of the output of a layer with n inputs and m neurons can be framed as a matrix operation

$$\boldsymbol{y} = g\left(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}\right) \tag{2.10}$$

where $\boldsymbol{x}$ is a $1 \times n$ input matrix, $\boldsymbol{y}$ is a $1 \times m$ output matrix, $\boldsymbol{W}$ is a $n \times m$ matrix, $\boldsymbol{b}$ is a $1 \times m$ bias matrix, and g is a function that independently applies the activation function to each matrix element.

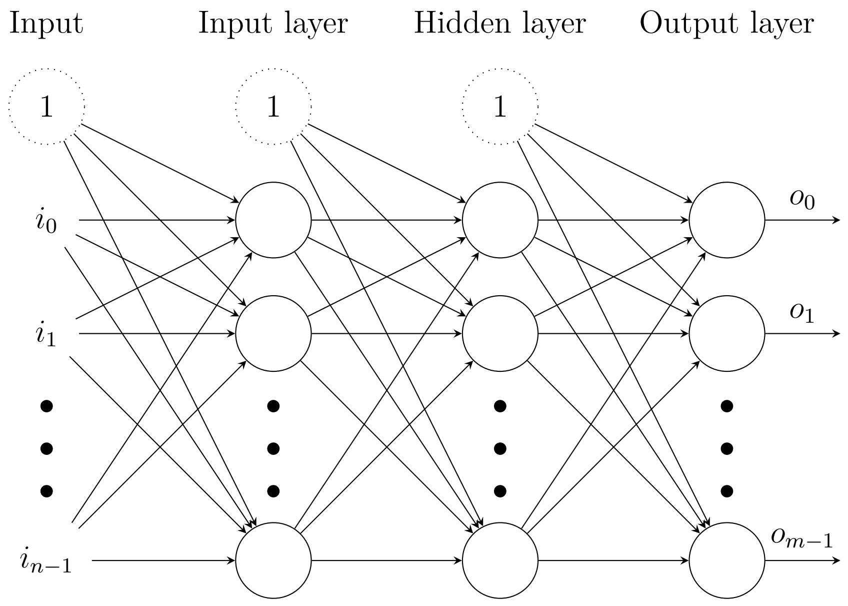


Figure 2.4: Diagram of an artificial neural network with a single hidden layer, n inputs, and m outputs. The solid circles represent neurons and the dashed circles represent the bias input to each layer.

2.2.3 Backpropagation

The previous section described how to propagate the inputs forward to produce the outputs of the neural network. To update the weights and train the neural network we must *backpropagate* the error from the outputs [9]. Backpropagation is accomplished by defining an error (or *loss*) function and performing gradient descent or similar optimization. A fraction of the error is attributed to each neuron. This fraction of the error is used to update that neuron's weights.

2.2.4 Single layer perceptrons

A single layer perceptron is a very simple neural network that distinguishes between two linearly separable classes [9]. It is equivalent to a single neuron with a unit step [13] activation function:

$$g(x) = \begin{cases} 1 & x > 0 \\ \frac{1}{2} & x = 0 \\ 0 & x < 0 \end{cases} \tag{2.11}$$

The perceptron is guaranteed to learn a decision boundary between linearly separable classes, but cannot distinguish between classes that are not [14].

2.2.5 Autoencoders

An autoencoder is a particular type of artificial neural network that is used to learn a different representation, or *coding*, of the input data [15]. This representation is also sometimes called a *latent* representation because it is hidden in the input data. An autoencoder is made of two parts, the encoder and decoder. The encoder takes an input $\boldsymbol{x}$ and produces the coded representation $\boldsymbol{z}$, while the decoder produces an estimate of the input $\hat{\boldsymbol{x}}$ from $\boldsymbol{z}$. The autoencoder is trained to minimize the error between $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$. Figure 2.5 illustrates an autoencoder with a single layer encoder and decoder.

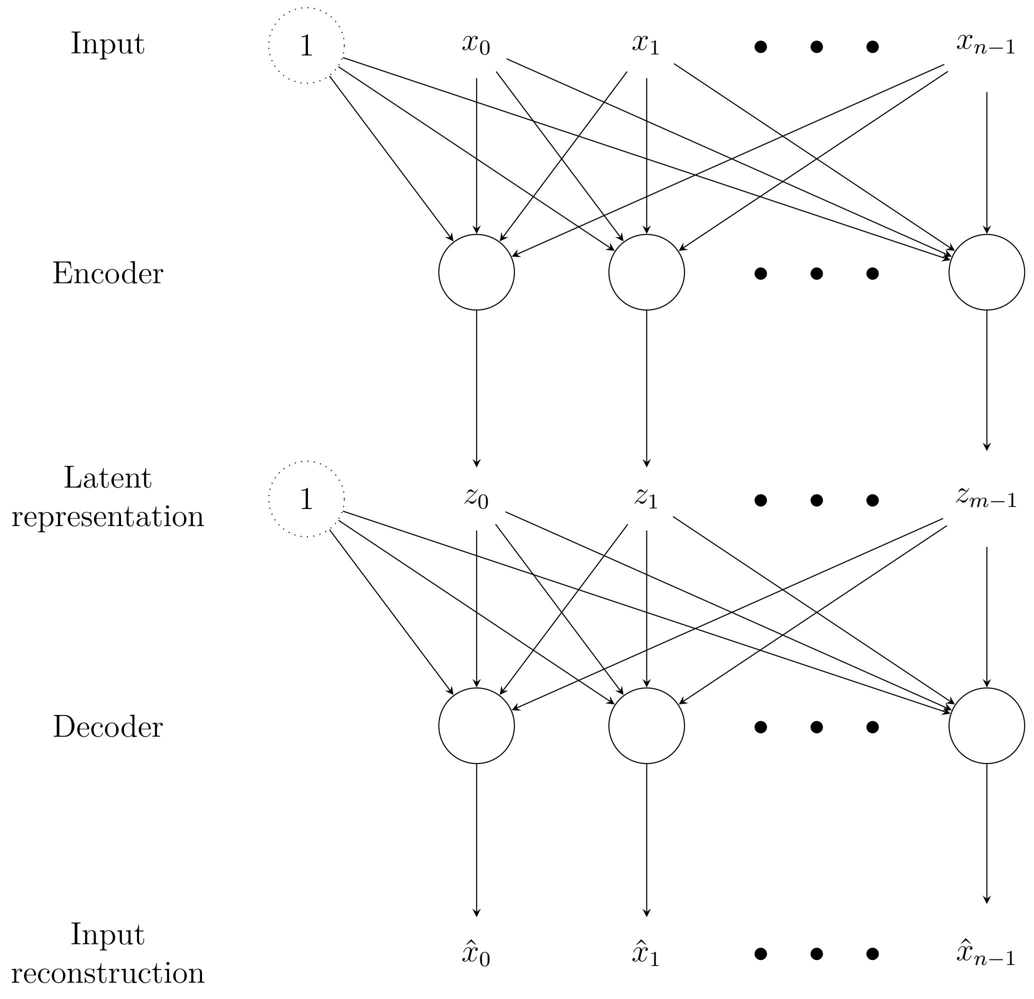


Figure 2.5: Diagram of an autoencoder with an n-dimensional input space and an m-dimensional latent space.

Chapter 3

Problem statement and related works

3.1 Communication systems as an optimization problem

The design process of a communication system can be framed as an optimization problem [16]. A memoryless modulator can be thought of as a function $f\left(s\right)$ that maps from a message s to transmitted samples $\boldsymbol{x}$. Similarly, the demodulator is a function $g\left(\boldsymbol{y}\right)$ that returns the most likely message $\hat{s}$ from the received samples $\boldsymbol{y}$, where $\boldsymbol{y}$ is the transmitted samples $\boldsymbol{x}$ corrupted by the channel. The functions f and g are then selected to minimize a metric that describes the performance of the modem, such as bit error rate.

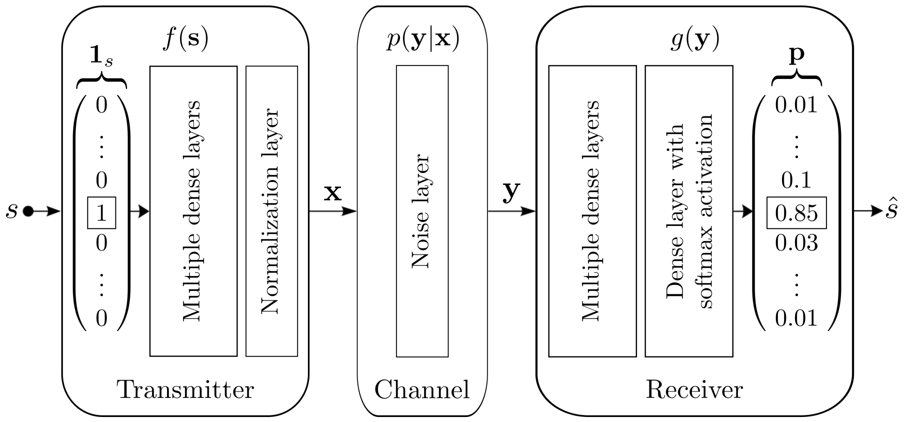


Figure 3.1: Block diagram of the autoencoder introduced by O'Shea and Hoydis. Figure reproduced from [16].

3.2 Previous work

In [16], an autoencoder is used to solve the optimization problem. The encoder and decoder correspond to the functions f and g and the latent representation corresponds to the modulated samples $\boldsymbol{x}$. The layout of the autoencoder is shown in Figure 3.1.

The transmitted message s is a binary sequence, which can be represented by an integer. The input to the autoencoder, 1_s, is one-hot encoded, meaning all elements of the input vector are 0 except the element with an index corresponding to the message, which is 1. A 4-bit message, for example, has 16 possible values and can be represented by an integer in the range $[0, 16)$ or a one-hot encoded vector with 16 elements.

The output of the autoencoder is a probability distribution $\boldsymbol{p}$ with the same shape as the one-hot encoded input 1_s and contains the likelihoods of each possible transmitted message being the received message. The prediction of the transmitted message, $\hat{s}$, is found by determining the message with the highest probability in $\boldsymbol{p}$.

Producing a probability distribution over the possible messages allows the use of cross-entropy loss. Unlike bit error rate, cross-entropy loss can still be minimized when there are no bit errors. The layout of the autoencoder modem proposed in [16] is shown in Table 3.1. Very recently, similar autoencoder modems have been shown to work in real systems over the air [17].

A unique feature of this autoencoder is the use of AWGN and dropout between the encoder and decoder layers. In most autoencoders, the latent representation is chosen to have a lower dimensionality than the input. This prevents the autoencoder from learning the identity, $f(s) = s$, since the input could not be represented in the smaller latent space. In this autoencoder there is no such limitation on the dimensionality of the latent representation. Using dropout prevents the autoencoder from learning the identity by forcing the decoder to reconstruct the input from only a random subspace of the latent space. Additionally, the use of AWGN encourages generation of a modulation scheme that is resistant to the kind of noise frequently seen in telecommunications. Corrupting the latent representation is similar to denoising autoencoders [18], which produce robust latent representations by corrupting the input of the encoder.

layer	output dimension
input	M
fully connected + ReLU	M
fully connected + linear	n
power normalization	n
AWGN and dropout	n
fully connected + ReLU	M
fully connected + softmax	M

Table 3.1: Layout of the autoencoder modem proposed in [16], where M is the number of possible symbols and n is twice the number of samples per symbol

Chapter 4

Redesigning the autoencoder modem

4.1 Architecture

Several optimizations can be made to the original autoencoder modem in [16] to reduce its complexity significantly. Recall that the encoder (which learns the modulator) is made up of two fully connected layers and a normalization layer, as shown in Figure 3.1. The modulator takes the message input as a one-hot encoded binary vector, and returns the modulated IQ samples associated with that message. Recall that a fully connected layer takes the form of:

$$\boldsymbol{y} = g\left(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}\right) \tag{4.1}$$

Consider that the input, $\boldsymbol{x}$, is a one-hot encoded binary vector, meaning that every element is 0, except the selected element which is 1. When the matrix multiply is performed between the weights, $\boldsymbol{W}$, and one-hot encoded vector, the output of the layer is $g(\boldsymbol{w}_i + \boldsymbol{b})$, where $\boldsymbol{w}_i$ is the weight vector corresponding to $\boldsymbol{x} = 1_i$. This first layer is simply operating as a lookup table of values commonly known as an *embedding*. For each embedding output, the second layer of the encoder will produce output only dependent on the value of i. The entire encoder can be replaced with a lookup table that learns a mapping from i to the associated latent representation.

The samples generated by the modulator must have some constraint applied to them, otherwise they will simply learn to increase power in the presence of noise. The two constraints used in [16] are both constraints on power—one forcing each sample's power to a particular value and one forcing the average sample power to a value. The first constraint results in a signal with constant power, or a constant envelope modulation scheme. Such schemes have been designed and used in real systems, such as FSK or GMSK. These schemes were useful in the past as they did not require the power amplifier in a hardware modulator to be a linear amplifier, simplifying the design. With the proliferation of modulation schemes such as OFDM and higher order QAM, non-constant envelope modulation schemes are now the norm; requiring constant envelope modulation adds an unnecessary constraint to the neural network. Therefore, the selected normalization function rescales the total power of the output to unit power:

$$f\left(x\right) = \frac{x}{\sqrt{p}} \text{ where } p = \frac{1}{n}\sum_{i=1}^{n}\sqrt{\operatorname{Re}\left(x\right)^2 + \operatorname{Im}\left(x\right)^2} \tag{4.2}$$

Like the modulator, it is possible to simplify the demodulator by applying a constraint. The autoencoder modem in [16] was designed to work at a rate of less than 1 bit per sample, specifically $4/7$ of a bit per sample. If the rate is constrained to less than 1 bit per sample, this results in the space of the IQ samples having equal or larger dimensionality than the space of the message bits. This guarantees that there is a trivial modulation scheme where each symbol is linearly separable from the collection of every other sample in the IQ samples space. If the modulation scheme is then constrained to this type, it is possible for the demodulator to be solved by a single layer. A single layer is equivalent to a set of single layer perceptrons, each determining whether the input samples belongs to the class of its symbol, or the class of every other symbol. It is therefore possible for the modulator to learn these linearly separable embeddings with the guarantee that the demodulator can find a solution.

It is still possible for the modulator to learn a modulation scheme with a rate greater than 1 bit per symbol, but this constraint effectively becomes a constraint on the power. Phase-shift keying always results in a linearly separable modulation scheme since the decision line can tend arbitrarily close to the tangent. Extending this idea, high rate modulation schemes with linearly separable symbols are possible but result in a constellation that tends toward

layer	output dimension
input	1
embedding lookup table	n
power normalization	n
AWGN and dropout	n
fully connected + softmax	M

Table 4.1: Layout of the proposed autoencoder modem, where M is the number of possible symbols and n is twice the number of samples per symbol

a hypersphere in the IQ samples space. This may result in a suboptimal modem.

The layout of the entire proposed autoencoder is shown in Table 4.1.

4.2 Evaluating complexity

In this section, the complexity of the forward propagation step of the autoencoder is considered. The forward propagation step is the inference step, when a symbol is modulated and demodulated. The focus is on the inference complexity since it will be used later in this thesis to compare to existing methods. Additionally, forward propagation is the first half of a training step, allowing it to be a good indicator of training complexity.

The autoencoder proposed above is significantly less complex than the one proposed in [16]. Not only does this result in a network that is easier and faster to train, but this reduced complexity also results in a modem that is more reasonable to implement in hardware and use in a real communication

	original	proposed
modulator	$M^2 + M + Mn + n$	Mn
demodulator	$M^2 + 2M + Mn$	$Mn + M$
total	$2M^2 + 3M + 2Mn + n$	$2Mn + M$
for large M	$O(M^2)$	$O(M)$
for large N	$O(n)$	$O(n)$

Table 4.2: Space complexities of the original and proposed autoencoders, where M is the number of possible symbols and n is twice the number of samples per symbol.

system.

To compute the complexity of each autoencoder, we must first understand the complexity of each individual operation. A fully connected layer with input size N and output size M has a space complexity of $M(N+1)$, due to the $N \times M$ weights matrix and M-length biases vector. Its computational complexity is $M(N+1)$ multiplies and MN adds.

The space complexities of the original and proposed autoencoders are shown in Table 4.2. The space and computational complexities are reduced from a quadratic to linear relationship with the number of possible symbols. Additionally, the proposed modulator is nearly "free" computationally, due to its implementation as a simple lookup table.

4.3 Comments on design decisions

Much of the complexity of the autoencoder is due to the exponential relationship between the number of bits and the number of possible symbols.

The number of possible symbols is equal to 2^k where k is the number of bits per symbol. If it were possible for the outputs of the demodulator to be the individual bits of the symbol, rather than the probability of each possible symbol, the complexity would be reduced significantly. Experimentation yielded several difficulties with this, however.

The first issue was that it was difficult to get the autoencoder to converge; in other words, the trained autoencoder could not distinguish between different symbols, even without the presence of AWGN. This was solved by switching to the Adam optimizer [19] and by returning the demodulator to two layers, with ELU [12] activation functions rather than the ReLU used in [16].

Even after the autoencoder could converge to a valid modem, its bit error rate performance did not come close to matching the Hamming(7,4) code which was matched by [16]. This is possibly due to the loss function. When the outputs are chosen to be a probability distribution, the loss function used is binary cross-entropy:

$$L = \sum_i y_i \log \hat{y}_i \tag{4.3}$$

where $\hat{y}_i$ is the estimated probability for symbol i and y_i is the actual probablity of symbol i (which is 0 for all symbols except the transmitted symbol which has probability 1). The loss function used with bit output was sum-of-squared error:

$$L = \sum_i (b_i - \hat{b}_i)^2 \tag{4.4}$$

where b_i and $\hat{b}_i$ are the value and the estimated value of bit i, respectively. Using binary cross-entropy explicitly maximizes the probability of receiving the correct symbol, whereas using sum-of-squared error only minimizes each bit's error individually. It is possible that this explicit maximization of symbol discrimination is necessary for learning a modulation scheme that performs well.

Due to these issues, more research into using bit output is necessary to produce that type of modem. An autoencoder modem with bit output, rather than a probability distribution over the possible symbols, would open up the possibility of using very large block sizes and potentially coming very close to the Shannon limit.

Chapter 5

Exploring additional use cases

The autoencoder modem in [16] was compared to Hamming codes, specifically the Hamming(7,4) code. Hamming codes are considered the first error-correcting codes invented, which is likely why they were chosen. They do have several limitations, however, and other codes are generally preferred in most applications—they are only capable of correcting a single bit error per block and only work at particular rates.

5.1 Comparison to Hamming codes

Hamming codes only provide enough redundancy to correct a single bit error via the added parity bits. In order to correct additional bit errors at the same rate, the amount of redundancy must be increased (in other words, the instantaneous data rate must be reduced). With Hamming codes this is not possible, as they are defined by their block size, but with the autoencoder modem the block and embedding sizes can be increased while maintaining the same data rate. The autoencoder modem was trained with double and triple the block sizes used to match Hamming(7,4). This resulted in several autoencoder modems with the same rate but varying bit error rate performance. Spreading the information across more samples by increasing the number of input bits and output samples resulted in lower bit error rates without changing the data rate. The curvature of the bit error rate curve steepened, improving performance in the operating range of approximately -2dB or greater signal-to-noise ratio. The bit error rate performance of each autoencoder is shown in Figure 5.1.

This promising result indicates that Shannon-approaching performance may be possible by increasing the block size, but this does not come without a cost. Testing was halted at the 12 bits, 21 samples autoencoder modem because of the exponential increase in complexity. The next largest autoencoder, with 16 bits and 28 samples per block, could not be trained due to the exponential space complexity resulting in an autoencoder that could not fit in memory. It would be interesting to see if continuing to increase the block size results in a Shannon-approaching modulation scheme, but without reducing the complexity further this is not feasible.

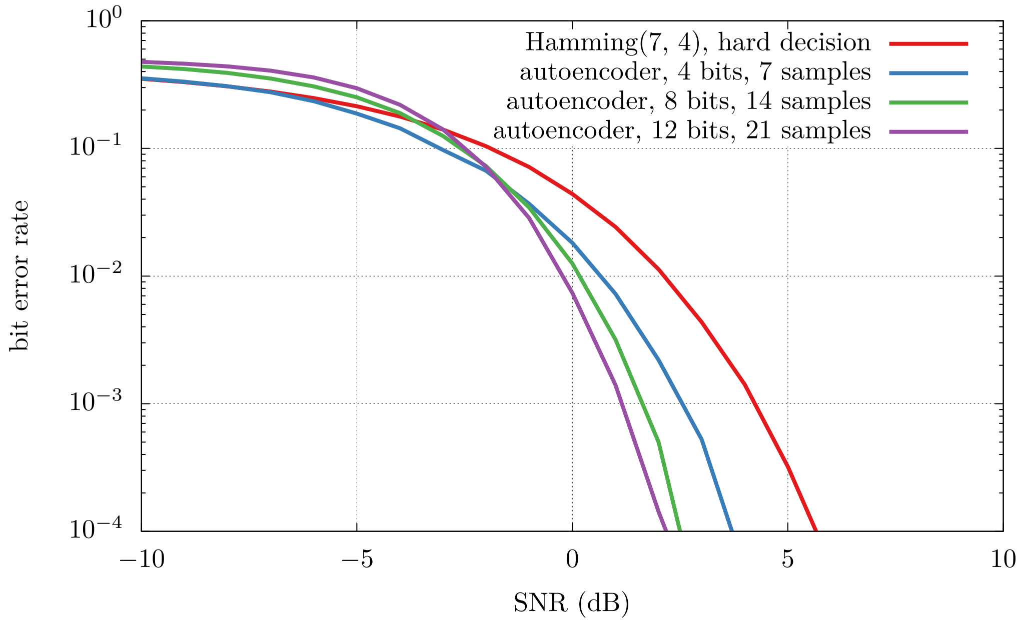


Figure 5.1: Comparison of rate $4/7$ autoencoders with varying block sizes

5.2 Comparison to convolutional codes

Before the invention of turbo codes in the early 1990s, convolutional codes most closely approached the Shannon limit. In addition to generally outperforming Hamming codes, convolutional codes can be used at a wider variety of rates, though rate $1/2$ and $1/3$ convolutional codes are the most common. Using lower rates is sometimes desirable since the lowest usable SNR, the Shannon limit, decreases as the rate decreases allowing bandwidth to be sacrificed to push data through at lower SNRs. Convolutional codes also have a variable constraint length, k, which gives some control over the level of error correction capable by the code at a particular rate.

5.2.1 Competing with a common convolutional code

One of the most common convolutional codes in use today is the rate $1/3$, $k = 7$ code used in the Long-Term Evolution (LTE) wireless communication standard [20]. If the autoencoder modem is capable of competing with this commonly used code, it may indicate a promising future for the technique. The autoencoder modem was configured with 7 bits per symbol and 21 samples per symbol, and trained with a 10% sample dropout probability and 0dB SNR.

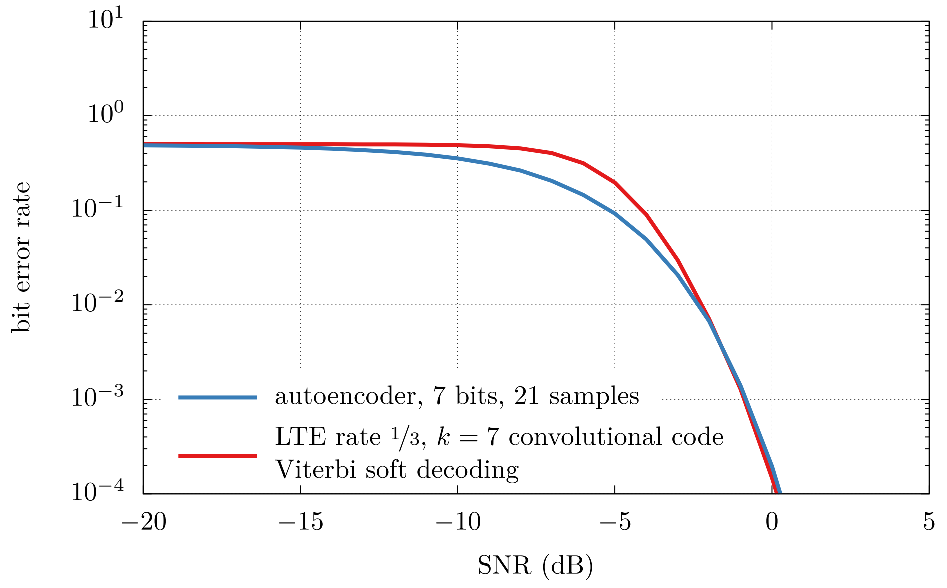
The autoencoder modem performs very similarly to convolutionally coded BPSK, as shown in Figure 5.2. The autoencoder modem appears to have slight degradation in bit error rate above approximately -2dB of SNR, however the difference from the convolutional code is minimal. The autoencoder has a notable advantage below that point, though most communication systems would not be designed to work with such high bit error rates.

5.2.2 Comparing complexity with convolutional codes

The autoencoder modem may be competitive with a similar convolutional code, but it is necessary to see how the computational complexities compare as well. The common way to decode convolutional codes is with a soft-decision Viterbi decoder, a variety of the Viterbi algorithm that uses bit likelihoods as input, rather than "hard" decision bits.

For a rate a/b convolutional code with constraint length k and n coded

(a)



(b)

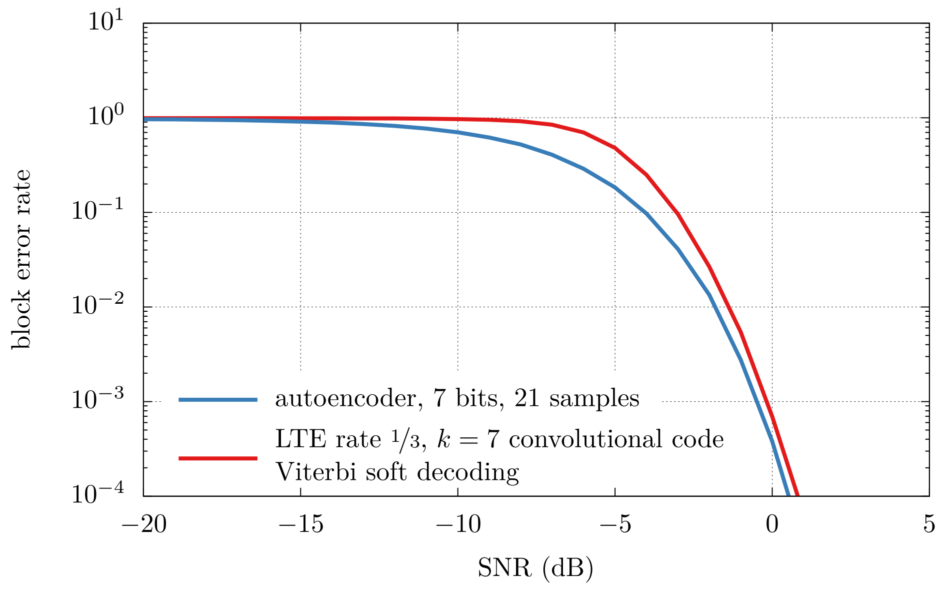


Figure 5.2: (a) Bit and (b) block error rate comparisons of an autoencoder modem to BPSK with a similar convolutional code. Since convolutional codes do not block encode, every 7 bits are considered a block.

input bits, 2^a paths are compared at every state. This computation is done for each of 2^{k-1} states over n sets of parity bits, with b soft decision likelihoods added to each path cost at each step.

A similar autoencoder modem uses a block size k, with $\frac{bk}{a}$ samples per block. This results in $\frac{2bk}{a}(2^k+1)n$ multiplications and $\frac{2bk}{a}2^k n$ additions. For every single computation step done in the Viterbi decoder, the autoencoder demodulator performs $\frac{4bk}{a}$ additions and multiplications, plus an extra $\frac{4bk}{a}$ multiplications per block. For a rate 1/3, $k = 7$ code, this is 84 multiplications and additions for every Viterbi calculation, which is approximately $2^a b$, or 6, additions along with some parity and max calculations.

It is clear that the autoencoder demodulator requires more computations than the Viterbi decoder, but the algorithms have similar time complexities. The number of operations of the autoencoder demodulator grows slightly faster than the Viterbi decoder in terms of k, with a complexity of $O\left(2^k kn\right)$ compared to the Viterbi algorithm's $O\left(2^k n\right)$. This is a significant improvement over the autoencoder proposed in [16], which as shown earlier has a quadratic computational complexity with the number of symbols, resulting in a complexity of $O\left(2^{2k} n\right)$.

5.2.3 Advantages of the autoencoder modem

The autoencoder modem has two particular advantages over convolutional codes. First, since it block encodes data, it does not require any special form of termination. Recall that a convolutional encoder is a state machine—when the entire message has been passed into the convolutional encoder, there is still some information remaining in the states. The most common method of dealing with the extra state is to flush it out by inserting zeros into the encoder in a process known as zero termination. These extra states are then transmitted appended to the end of the data, resulting in a rate loss. For transmitting very small blocks of data, such as control messaging or packet headers, this rate loss is unacceptable and leads to the other two forms of termination: truncation and tail-biting [21]. Truncation is accomplished by simply not transmitting the extra state, and taking a guess at which state was final, which results in increased bit error rates. Tail-biting is a more reasonable solution, where the termination state is guaranteed to be the same as the starting state. This makes constructing the data more complicated, and requires double the computation at the decoder, while still incurring a slight performance degradation over zero termination. The autencoder modem does not require any special termination, and even has better performance for small blocks, as shown in Figure 5.2b.

The second advantage of the autoencoder modem is that its outputs are not just the decoded block, but its likelihood. In some cases, the likelihood values can be used in later stages of processing. For convolutional codes, the per-bit likelihoods are computed using the BCJR algorithm, which is more computationally expensive than the Viterbi algorithm. The autoencoder modem does not generate likelihoods per bit, but the per block likelihoods may be used similarly.

Chapter 6

Factored autoencoder with reduced complexity

The autoencoder modem was shown to be competitive with convolutional codes, but still needs improvement to come close to the Shannon limit and compete with turbo codes. The limiting factor appears to be the exponential space and time complexity associated with increasing the block size. In the following sections, attempts to reduce the complexity by factoring the blocks has mixed success. Efforts to create a reduced complexity modulator showed early signs of success, but more research must be done to reduce the complexity of the demodulator.

6.1 Factored modulator

Consider an autoencoder with rate r and block size k. The number of modulated samples per block is k/r, resulting in an embedding lookup table of $\frac{2^{k+1}k}{r}$ parameters[1]. It is possible to reduce the size of this lookup table with factorization, combining c smaller blocks of k/c bits with an operation with lower complexity than exponential. We define this embedding block size as $k_e = k/c$.

The most straightforward solution is to concatenate c embeddings of size $\frac{2k_e}{r}$ and use a single fully connected layer with $\frac{2k}{r}$ inputs and outputs to combine the embeddings into a single block. This block contains the information of k bits, but with only $\frac{2^{k_e+1}k_e}{r} + \frac{2k}{r}\left(\frac{2k}{r}+1\right)$ trainable parameters. With a sufficiently small k_e, the exponential relationship in k is reduced to the quadratic relationship resulting from the fully connected layer.

For example, consider a rate $1/2$ autoencoder with $k = 16$ and $k_e = 4$. The original formulation will contain just over 4 million trainable parameters in the lookup table, whereas the factored implementation only requires a lookup table with 256 parameters and an additional 72 parameters due to the fully connected layer. This method does involve additional computation from the fully connected layer, as well as an additional hyperparameter k_e, but allows modulators with block sizes of thousands or more for traditional code rates. The reduction in memory required to store the modulator is visualized in

[1]An embedding contains 2 values per complex sample, hence the $k+1$ in the exponent.

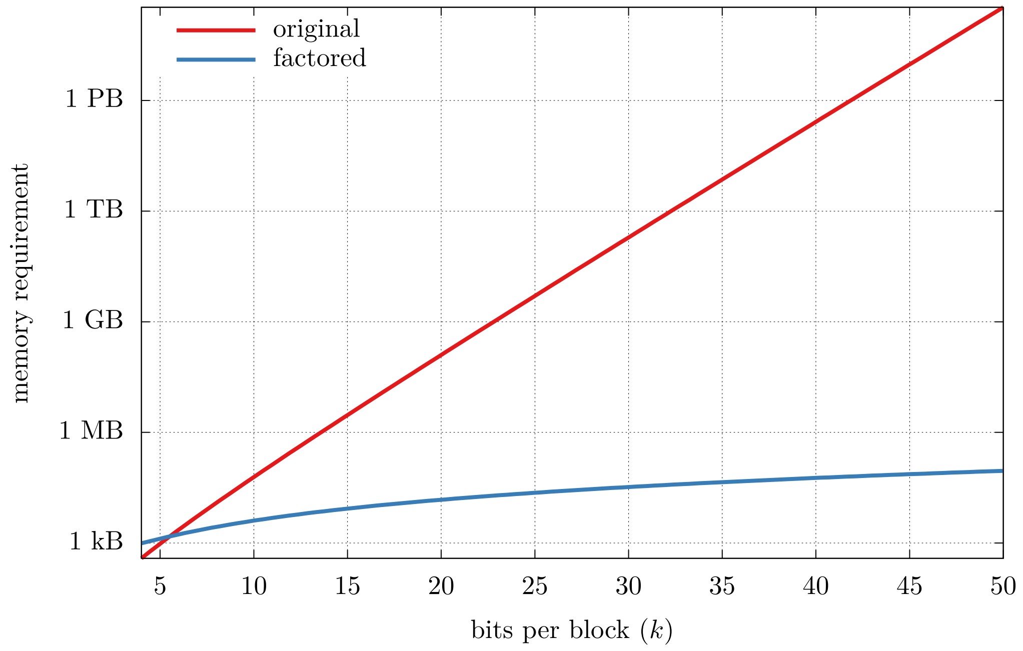


Figure 6.1: Amount of memory required to store the single-precision trainable parameters of a rate $^1/_3$ autoencoder modulator. The factored modulator has embedding block size $k_e = 4$.

Figure 6.1

The factored modulator was tested at a rate of $^4/_7$ with $k_e = 4$ and $k = 4$, $k = 8$, and $k = 12$. For each of these scenarios, the BER and BLER performance was substantially the same as the equivalent autoencoder with the full block size. Due to the inability to create a reduced complexity demodulator, the factored modulator was only tested with the original demodulator, limiting the testable block size significantly. The success at the tested block sizes, however, indicates that the layout may work for larger block sizes.

6.2 Factored demodulator

As shown earlier, space and time complexity of the demodulator also scales exponentially with the block size k. In an attempt to reduce the complexity of the demodulator, factorization was again used.

The output of the demodulator is a predicted probability distribution across the 2^k possible transmitted blocks. The output layer can be factored into d distributions where each distribution predicts $2^{k/d}$ bits of the total block. We define this subblock size as $k_f = k/d$. There is no requirement for the value of c in the factored modulator to match d in the demodulator.

This factored demodulator is only useful if it can be proved that the new set of probability distributions are capable of representing the probability of the entire message and results in the same loss calculation. Recall that the loss function is cross entropy error, which takes the form:

$$L = \sum_i y_i \log \hat{y}_i \tag{6.1}$$

where y_i is the "true" probability and $\hat{y}_i$ is the predicted probability. In this situation there is only one target class, or message, at a time, resulting in a "true" probability distribution with a value of 1 for a particular i and 0 everywhere else. This reduces the loss to the form:

$$L\left(t\right) = \log \hat{y}_t \tag{6.2}$$

where $\hat{y}_t$ is the predicted probability of the target message t.

Now consider several probability distributions for each subblock. The joint probability of all of the subblocks can be written as:

$$\hat{y}_t = \prod_i \hat{y}_{t,i} \tag{6.3}$$

where $\hat{y}_{t,i}$ is the predicted probability of the target for subblock i. This is only strictly true if the predicted subblocks are independent of each other, which is true for a valid demodulator since the transmitted bits are independent. This allows us to compute the loss of the factored demodulator as:

$$\begin{aligned} L\left(t\right) &= \log \prod_i \hat{y}_{t,i} && (6.4)\\ &= \sum_i \log \hat{y}_{t,i} && (6.5)\end{aligned}$$

Several designs utilizing the factored probability distribution were tested, shown in Table 6.1. Each demodulator, regardless of the values of k, performed substantially similarly to an autoencoder modem with block size k_f, instead of k. An illustration of a factored demodulator compared to the regular demodulator is shown in Figure 6.2.

One possible explanation for this behavior is that the demodulation is being deferred until the final fully connected layer and softmax. The final layer produces a quantity for each subblock that is high for the target block

layer	output dimension
sample input	n
fully connected	2^k
split + $d\times$ softmax	$d \times 2^{k_f}$

(a)

layer	output dimension
sample input	n
fully connected + ELU	2^k
split	$d \times 2^{k_f}$
$d\times$ fully connected + softmax	$d \times 2^{k_f}$

(b)

layer	output dimension
sample input	n
fully connected + ELU	2^k
fully connected	2^k
split + $d\times$ softmax	$d \times 2^{k_f}$

(c)

layer	output dimension
sample input	n
fully connected + ELU	2^k
fully connected + ELU	2^k
split	$d \times 2^{k_f}$
$d\times$ fully connected + softmax	$d \times 2^{k_f}$

(d)

Table 6.1: Layouts of the tested factored demodulators, with n inputs, block size k, and factored block size k_f, in order of increasing complexity. Each layout was tested with $k_f = 4$, $k = 8$, $n = 14$. Other sizes were also tested with no success.

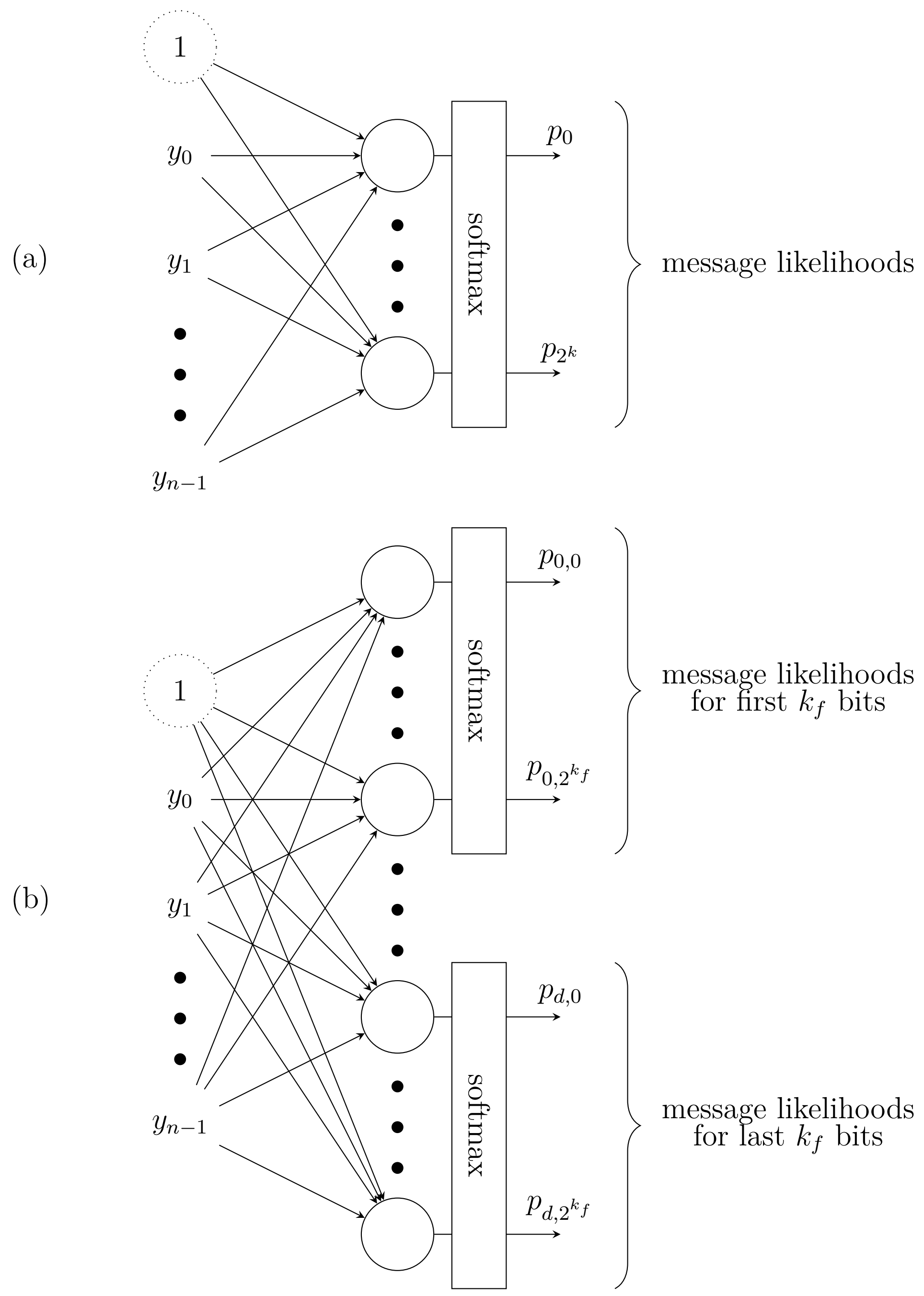


Figure 6.2: Diagram of (a) the normal demodulator and (b) the factored demodulator shown in Table 6.1a.

and low for the other blocks. Each of these quantities is the result of a dot product of the input with the weights. Since the dot product is a measure of the magnitude of the input projected onto the weights, it is very likely that the demodulator simply learns to project the input onto the possible messages and pick the one with the largest magnitude. When the final output layer is factored into smaller output layers, it may be forcing the demodulation to be done only on a smaller set of messages due to the reduced number of output projections. This could result in the observed behavior. This problem requires more research to determine if this is in fact the issue and to find the solution.

Chapter 7

Conclusion

In this thesis, a method of designing the physical layer of a communication system using an autoencoder was proposed. The autoencoder modem was shown to reach bit error rate performance similar to Hamming and convolutional codes with similar parameters. Additionally, this autoencoder architecture has reduced complexity compared to previous implementations and is near the computational complexity of the Viterbi algorithm.

The primary limitation with this architecture was shown to be the exponential increase in computational complexity when the input block size is increased. Increasing the input block size was shown to reduce the bit error rate, but results in an intractable autoencoder with large block sizes. Methods for creating a factored autoencoder by splitting the encoder (or modulator) and decoder (or demodulator) into smaller encoders and decoders were proposed. The factored modulator was shown to work as expected and reduce space complexity, but the factored demodulator incurred a bit error rate penalty.

Overall, the autoencoder modem was shown to be competitive with Hamming and convolutional codes, and was shown to be a suitable alternative to the convolutional code used in the LTE standard. Future work is necessary, however, to compete with codes that near the Shannon limit, such as turbo codes.

Chapter 8

Future work

There are several avenues for future research into using autoencoders to design communication systems. An obvious direction for future work is to reach near the Shannon limit. Turbo [1] and low-density parity-check (LDPC) [8] codes perform well because the information from each bit is spread over many output bits. An improved implementation may draw from the reason behind the name of turbo codes. Turbo codes are named after turbocharged engines, which feed exhaust into the air intake; similarly, turbo decoders use feedback in the decision process, but the encoder does not [22]. Recurrent neural networks (RNN), such as the long short-term memory (LSTM) architecture [23], use feedback to remember previous inputs and have proven to be good solutions for tasks using time-series data. An autoencoder modem using LSTM or other RNN architectures may be able to spread the information over many samples without requiring a very large neural network.

A strength of the autoencoder modem is the ease at which it can be modified for related tasks. Systematic codes, which embed the original information bits in the output bits, are common in modern communication systems because it is possible to skip decoding and avoid additional computation when the original bits are received intact. The autoencoder modem could potentially be modified to use multiple demodulators with varying complexity for use in different environments.

There are many possibilities for experimentation in applying various constraints to the input samples to the demodulator during training. The power constraints could be expanded to be frequency selective, resulting a method for shaping the bandwidth of the modulation scheme. The samples could also be corrupted using channels with memory, such as multipath channels, to produce modulation schemes that are more resilient to them.

Another potential modification of the autoencoder modem is to use channel state information (CSI) as an additional input. CSI contains a description of the channel, such as an impulse response, and the modulation scheme could adapt to particular channels. A more complex implementation could learn the CSI in the demodulator and backpropagate that information across the network to train the autoencoder during operation.

Chapter 9

Bibliography

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes.," in *Proc. IEEE International Conference on Communications.*, vol. 2, pp. 1064–1070 vol.2, May 1993.

[2] "Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model," ISO/IEC 7498-1:1994(E), International Organization for Standardization, 1996.

[3] J. G. Proakis and M. Salehi, *Digital communications.* McGraw-Hill, 2008.

[4] F. Gray, "Pulse code communication," Mar. 17 1953. US Patent 2,632,058.

[5] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, pp. 147–160, April 1950.

[6] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[7] G. D. Forney, "Concatenated codes," *MIT Press*, 1965.

[8] D. J. C. MacKay and R. M. Neal, "Near shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 33, pp. 457–458, Mar 1997.

[9] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach (3rd edition).* Prentice Hall, 2009.

[10] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[11] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *CoRR*, vol. abs/1508.06576, 2015.

[12] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *CoRR*, vol. abs/1511.07289, 2015.

[13] M. Abramowitz and I. A. Stegun, *Handbook of mathematical functions.* Dover, 1972.

[14] C. M. Bishop, *Pattern recognition and machine learning.* Springer, 2009.

[15] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1* (D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, eds.), pp. 318–362, Cambridge, MA, USA: MIT Press, 1986.

[16] T. J. O’Shea and J. Hoydis, “An introduction to machine learning communications systems,” *CoRR*, vol. abs/1702.00832, 2017.

[17] S. Dörner, S. Cammerer, J. Hoydis, and S. ten Brink, “Deep Learning-Based Communication Over the Air,” *ArXiv e-prints*, July 2017.

[18] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, “Extracting and composing robust features with denoising autoencoders,” in *ICML*, 2008.

[19] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014.

[20] “Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding,” TS 36.212, 3rd Generation Partnership Project (3GPP), Sept. 2008.

[21] H. Ma and J. Wolf, “On tail biting convolutional codes,” *IEEE Transactions on Communications*, vol. 34, pp. 104–111, Feb 1986.

[22] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Transactions on Information Theory*, vol. 42, pp. 429–445, Mar 1996.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–80, 12 1997.

Appendix A

Code

```
#!/usr/bin/env python3
import tensorflow as tf
import tensorflow.contrib.slim as slim
import numpy as np


def db2pow(v):
    "convert dB to power"
    return np.power(10, v / 10)


def count_nparams():
    "returns the number of trainable parameters"
    nparams = 0
    for trainable_variable in tf.trainable_variables():
        shape = trainable_variable.get_shape()
        this_nparams = 1
        for dim in shape:
            this_nparams = this_nparams * int(dim)
        nparams += this_nparams
    return nparams


def scale_power(samples):
    "scales the input to unit power"
    samples_per_symbol = samples.get_shape().as_list()[1]
    power = tf.reduce_sum(tf.square(samples), axis=[1, 2])/samples_per_symbol
    power = tf.stack([power for i in range(samples_per_symbol)], axis=1)
    power = tf.stack([power for i in range(2)], axis=2)
    return tf.div(samples, tf.sqrt(power))


def add_channel(samples, dropout_prob, snr):
    "adds a channel with the supplied dropout probability and SNR"
    out = samples
    out = tf.nn.dropout(out, 1 - dropout_prob)
    out = out + tf.random_normal(out.get_shape().as_list(),
                                 stddev=np.sqrt(1 / db2pow(snr)))
    return out


class Model():
    def __init__(self, sess, bits_per_symbol, samples_per_symbol, batch_size,
                 is_training, save_path, require_save=False,
                 train_dropout=None, train_snr=None):
        self.sess = sess
        self.M = 2 ** bits_per_symbol
        self.n = samples_per_symbol
        self.batch_size = batch_size
        self.is_training = is_training
        self.save_path = save_path
        self.train_dropout = train_dropout
        self.train_snr = train_snr
        self.build_model()
        self.saver = tf.train.Saver(tf.trainable_variables())
        try:
            self.saver.restore(sess, self.save_path)
            print("Save file restored")
        except:
            if require_save:
                print("No save file found.")
                raise
            else:
                print("No save file found.  Creating at {}".format(
                    self.save_path))

    def build_model(self):
        "Initialize the TensorFlow model"
        with slim.arg_scope([slim.fully_connected],
                            weights_regularizer=slim.l2_regularizer(0.0005)):

            # MODULATOR
```

```
self.mod_input = tf.placeholder(tf.int64, shape=(self.batch_size,))

# Get symbol embeddings
embedding = tf.get_variable(
    'embedding',
    shape=(self.M, 2 * self.n),
    dtype=tf.float32,
    initializer=tf.random_normal_initializer(stddev=1))
mod = tf.nn.embedding_lookup(embedding, self.mod_input)
mod = tf.reshape(mod, [self.batch_size, 2 * self.n])

# reshape to I/Q
mod = tf.reshape(mod, [self.batch_size, self.n, 2])

# Power scale
mod = scale_power(mod)
self.mod_output = mod

# CHANNEL (training only)
if self.is_training:
    self.demod_input = add_channel(self.mod_output,
                                   self.train_dropout,
                                   self.train_snr)
else:
    self.demod_input = tf.placeholder(
        tf.float32,
        self.mod_output.get_shape().as_list())
        # DEMODULATOR
        demod = self.demod_input

        # reshape from I/Q
        demod = tf.reshape(demod, [self.batch_size, 2 * self.n])

        # output layer with linear activation
        demod = slim.fully_connected(demod, self.M, scope='demod/fc',
                                     activation_fn=None)

    # loss function
    self.loss = tf.reduce_sum(
        tf.nn.sparse_softmax_cross_entropy_with_logits(
            labels=self.mod_input, logits=demod))

    # demodulator output
    self.demod_output = tf.argmax(demod, axis=1)

    # block error rate
    block_successes = tf.reduce_sum(tf.cast(
        tf.equal(self.mod_input, self.demod_output), tf.float32))
    self.bler = 1 - block_successes / self.batch_size

    # Adam optimizer
    if self.is_training:
        self.optimizer = tf.train.AdamOptimizer(
            learning_rate=0.0001).minimize(self.loss)
```

```
def train_step(self, optimizer, validate=False):
    "Train or validate over a single batch"
    symbols = self.random_symbols()
    if validate:
        loss, bler = self.sess.run([self.loss, self.bler],
                                   feed_dict={self.mod_input: symbols})
        print('Loss: {:10.4f}\tBLER: {:1.6f}'.format(loss, bler))
        self.saver.save(self.sess, self.save_path)
    else:
        self.sess.run(optimizer, feed_dict={self.mod_input: symbols})

def train(self, iterations):
    "Run many training steps, occasionally validating"
    for i in range(iterations):
        self.train_step(self.optimizer, (i % 10000 == 0))

def modulate(self, message):
    """Modulate the supplied batch of symbols and return the modulated
    samples"""
    samples = self.sess.run(self.mod_output,
                            feed_dict={self.mod_input: message})
    return np.array([[x[0] + x[1] * 1j for x in s] for s in samples])

def demodulate(self, samples):
    """Demodulate the supplied batch of samples and return the demodulated
    symbols"""
    split_samples = np.stack([np.real(samples), np.imag(samples)], axis=-1)
    return self.sess.run(self.demod_output,
                                    feed_dict={self.demod_input: split_samples})

    def demodulate_bler(self, samples, truth):
        """Demodulate the supplied batch of samples and return the block
        (or symbol) error rate"""
        split_samples = np.stack([np.real(samples), np.imag(samples)], axis=-1)
        out = self.sess.run(self.demod_output,
                                   feed_dict={self.demod_input: split_samples})
        return 1 - float(np.sum(out == truth)) / self.batch_size

    def random_symbols(self):
        "Generate a batch of random symbols"
        return np.random.randint(self.M, size=(self.batch_size,))


def main_train(bits_per_symbol, samples_per_symbol, train_snr, train_dropout,
               save_path, iterations, batch_size):
    "Train the model"
    with tf.Graph().as_default(), tf.Session() as sess:
        m = Model(sess, bits_per_symbol=bits_per_symbol,
                  samples_per_symbol=samples_per_symbol,
                  batch_size=batch_size, is_training=True, train_snr=train_snr,
                  train_dropout=train_dropout, save_path=save_path)
        sess.run(tf.global_variables_initializer())
        print("Number of trainable parameters: {}".format(count_nparams()))
        m.train(iterations)
```

```
def main_test(bits_per_symbol, samples_per_symbol, snr, matfile, save_path,
              iterations):
    "Test a trained model, producing a MATLAB/Octave file containing results"
    import scipy.io as sio
    import sys
    with tf.Graph().as_default(), tf.Session() as sess:
        # Open a new model not in training mode, and load from file
        try:
            m = Model(sess, bits_per_symbol=bits_per_symbol,
                      samples_per_symbol=samples_per_symbol, batch_size=1,
                      is_training=False, require_save=True,
                      save_path=save_path)
        except:
            sys.exit("The TensorFlow model could not be initialized. "
                     "This is likely due to a missing save file")

        samples_tx = np.zeros((len(snr), iterations, samples_per_symbol),
                              dtype=np.complex)
        samples_rx = np.zeros((len(snr), iterations, samples_per_symbol),
                              dtype=np.complex)
        symbols_tx = np.zeros((len(snr), iterations))
        symbols_rx = np.zeros((len(snr), iterations))
        snr_rx = np.zeros((len(snr)))
        snr_pow = db2pow(snr)

        # Iterate over each snr
        for i in range(snr.size):
            snr_rx[i] = snr[i]
            for iteration in range(iterations):
                # Generate a random symbol
                msg = m.random_symbols()
                symbols_tx[i, iteration, ] = msg

                # Modulate the symbol
                samples = m.modulate(msg)
                samples_tx[i, iteration, ] = samples

                # Apply AWGN channel
                stdev = np.sqrt(1/(2 * snr_pow[i]))
                samples = samples \
                    + np.random.normal(0, stdev, size=samples.shape) \
                    + np.random.normal(0, stdev, size=samples.shape) * 1j
                samples_rx[i, iteration, ] = samples

                # Demodulate the received samples
                symbols_rx[i, iteration, ] = m.demodulate(samples)

        # Save the results
        sio.savemat(matfile, {'snr': snr_rx,
                              'samples_tx': samples_tx,
                              'samples_rx': samples_rx,
                              'symbols_tx': symbols_tx,
                              'symbols_rx': symbols_rx})


if __name__ == "__main__":
```

```python
import sys
import argparse

# common options
parser = argparse.ArgumentParser(add_help=False)
parser.add_argument("--samples_per_symbol", type=int,
                    help="Number of samples per symbol", default=21)
parser.add_argument("--bits_per_symbol", type=int,
                    help="Number of bits per symbol", default=7)
parser.add_argument("--model", help="Path to TensorFlow model",
                    default="saves/model.ckpt")

subparsers = parser.add_subparsers(dest="mode")

# training options
parser_train = subparsers.add_parser("train", help="Train the model")
parser_train.add_argument("--snr", type=float, help="Signal-to-noise ratio"
                          " to use during training (dB)", default=3)
parser_train.add_argument("--dropout", type=float, help="Dropout "
                          "probability to use during training",
                          default=0.1)
parser_train.add_argument("--iterations", help="Number of training "
                          "iterations", type=int, default=100000)
parser_train.add_argument("--batch-size", help="Batch size", type=int,
                          default=100)

# testing options
parser_test = subparsers.add_parser("test", help="Test the model")
```

```
parser_test.add_argument("--snrs",
                         metavar=("MIN", "MAX", "COUNT"),
                         type=float, nargs=3, default=[0, 10, 10],
                         help="Test COUNT signal-to-noise ratios from MIN "
                         "to MAX")
parser_test.add_argument("--results", help="Path to write results to in "
                         "the MATLAB/Octave MAT format",
                         default="debug.mat")
parser_test.add_argument("--iterations", help="Number of test iterations",
                         type=int, default=100000)

args = parser.parse_args()
if args.mode == "train":
    main_train(args.bits_per_symbol, args.samples_per_symbol, args.snr,
               args.dropout, args.model, args.iterations, args.batch_size)
elif args.mode == "test":
    snr = np.linspace(args.snrs[0], args.snrs[1], args.snrs[2])
    main_test(args.bits_per_symbol, args.samples_per_symbol, snr,
              args.results, args.model, args.iterations)
else:
    parser.print_help()
    parser.exit()
```