Learning Algorithms for Error Correction

by Loren Peter Lugosch

Department of Electrical and Computer Engineering

McGill University

Montréal, Québec, Canada

April 2018

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Engineering

© Loren Peter Lugosch, 2018

There are wavelengths that people cannot see, there are sounds that people cannot hear, and maybe computers have thoughts that people cannot think.

— Richard Hamming

ABSTRACT

Channel coding enables reliable communication over unreliable, noisy channels: by encoding messages with redundancy, it is possible to decode the messages in such a way that errors introduced by the channel are corrected. Modern channel codes achieve very low error rates at long block lengths, but long blocks are often not acceptable for low-latency applications. While there exist short block codes with excellent error-correction performance when decoded optimally, designing practical, low-complexity decoding algorithms that can achieve close-to-optimal results for short codes is still an open problem. In this thesis, we explore an approach to decoding short block codes in which the decoder is recast as a machine learning algorithm. After providing the background concepts on error-correcting codes and machine learning, we review the literature on learning algorithms for error correction, with a special emphasis on the recently introduced "neural belief propagation" algorithm. We then describe a set of modifications to neural belief propagation which improve its performance and reduce its implementation complexity. We also propose a new syndrome-based output layer for neural error-correcting decoders which takes the code structure into account during training to yield decoders with lower frame error rate. Finally, we suggest some future work.

ABRÉGÉ

Le codage de canal permet une communication fiable sur des canaux non fiables et bruyants. En ajoutant de la redondance aux messages, il est possible de décoder les messages de telle sorte que les erreurs introduites par le canal soient corrigées. Les codes de canal modernes atteignent des taux d'erreur très faibles à des longueurs de blocs longues, mais les blocs longs ne sont souvent pas acceptables pour les applications à faible latence. Bien qu'il existe des codes en blocs courts avec d'excellentes performances de correction d'erreur lorsqu'ils sont décodés de manière optimale, la conception d'algorithmes de décodage pratiques et peu complexes qui peuvent obtenir des résultats proches de l'optimum pour les codes courts reste un problème ouvert. Dans cette thèse, nous explorons une approche de décodage de codes en blocs courts dans laquelle le décodeur est appréhendé comme un algorithme d'apprentissage automatique. Après avoir fourni les concepts de base sur les codes correcteurs d'erreurs et l'apprentissage automatique, nous passons en revue la littérature sur les algorithmes d'apprentissage pour la correction d'erreurs, en mettant l'accent sur l'algorithme "neural belief propagation" récemment introduit. Nous décrivons ensuite un ensemble de modifications à cet algorithme qui améliorent ses performances et réduisent sa complexité de mise en œuvre. Nous proposons également une nouvelle couche de sortie à base de syndrome pour les décodeurs de correction d'erreur neuronaux qui prend en compte la structure du code pendant l'apprentissage, ce qui permet de réduire le taux d'erreur de trame. Enfin, nous suggérons de pistes de recherche future.

ACKNOWLEDGMENTS

This Masters degree has been the happiest time of my life thus far, partly because these were years spent "beholding the bright countenance of truth in the quiet and still air of delightful studies", but also because of the people I spent the time with. I would like to acknowledge and thank those people.

I could not have had a better supervisor than Prof. Warren Gross. He was generous with his time and counsel and had many enjoyable conversations with me on research and other topics. I thank Prof. Takahiro Hanyu and Prof. Naoya Onizawa at Tohoku University for hosting me for my research visit to Japan and warmly welcoming me into the Brainware Lab. My colleagues at Tel-Aviv University, Mr. Eliya Nachmani, Prof. David Burshtein, and Prof. Yair Be'ery, wrote the fascinating paper that sparked the question behind my research and had helpful exchanges with me over the course of our collaboration. I thank Prof. Stephan ten Brink at the University of Stuttgart for taking the time to read my thesis and giving me useful feedback. Dr. François Leduc-Primeau at the École de Technologie Supérieure kindly corrected the French translation of my abstract.

I also thank my colleagues at McGill, especially Mr. Harsh Aurora, Mr. Andrey Tolstikhin, Mr. Scott Dagondon Dickson, Mr. Dylan Watts, Mr. Adam Cavatassi, and Mr. Dirk Dubois, for their friendship, their help debugging both my code and my crackpot ideas, and their making life in the otherwise staid McConnell Engineering Building a little more snarky, cheerful, and stochastic. Above all, I thank Ms. Aimee Castro, who, despite having had her ear talked off on the subject of neural networks and error-correcting codes, has not retracted her acceptance of my marriage proposal. ("Yet!", she reminds me, as I nag her to read draft after draft of my writing.)

Finally, this degree would not have been possible without the love and support of my parents. And I suppose a shout-out to my brother (who inspired me to study electrical engineering) and sister (who inspired me to go to grad school) is also in order: HENH?!?

CONTRIBUTION OF AUTHORS

The work in Chapter 4 was published in a conference paper for which I am the primary author ([1]) and a journal paper for which I am the third author ([2]). The results for the neural BP decoders and neural mRRD decoders in Chapters 3 and 4 were provided by David Burshtein. All other data, plots, and intellectual contributions in Chapters 4 and 5 are my own original work.

LIST OF ABBREVIATIONS

- AWGN — additive white Gaussian noise
- BCH — Bose-Chaudhuri-Hocquenghem
- BER — bit error rate
- BP — belief propagation
- BPSK — binary phase-shift keying
- E_b/N_0 — energy per bit to noise power spectral density ratio
- FER — frame error rate
- FF — feedforward
- HDD — hard-decision decoding
- HDPC — high-density parity-check
- KL — Kullback-Leibler
- LDPC — low-density parity-check
- LLR — log-likelihood ratio
- MAP — maximum *a posteriori*
- mRRD — improved random redundant decoding
- ML — maximum likelihood
- NMS — normalized min-sum
- NNMS – neural normalized min-sum
- NOMS — neural offset min-sum

- OMS — offset min-sum
- RNN — recurrent neural network
- SGD — stochastic gradient descent

LIST OF FIGURES

1.1 An appropriately constructed neural network (such as the one used in the Prisma app [3]) can learn to render a photograph in an artistic style. 3

2.1 Parity-check matrix and Tanner graph for the (7,4) Hamming code. 10
2.2 A path traversing variable node 3, check node 2, variable node 4, and check node 3 forms a cycle in a Tanner graph for the Hamming code. 12
2.3 Artificial neuron. 15
2.4 Biological neuron. 15
2.5 Illustration of gradient descent. 17
2.6 Noisy training examples (left) and the pattern learned by the neuron converging to the true underlying signal (right). 19

3.1 A four iteration neural BP decoder. (Figure adapted from [4].) 28
3.2 BER for BCH (63,36) code. 29
3.3 The mRRD decoder. (Figure taken from [4]). 31
3.4 Neural mRRD BER results for BCH(63,36) code. (Figure adapted from [4].) . . 32

4.1 Plots of the output of a degree-3 check node. 35
4.2 Four iterations of a decoder with weight sharing. (Figure adapted from [4].) . . 38
4.3 Two iterations of a relaxed decoder. 39
4.4 Performance comparison of BP and min-sum decoders for BCH (63,36) code. . 42
4.5 Performance comparison of BP and min-sum decoders for BCH (63,45) code. . 43
4.6 Performance comparison of neural BP and NNMS decoders for BCH (63,45) code when inputs are not properly scaled to LLR format. 44
4.7 Performance of a min-sum decoder (i.e., $\gamma = 0$), a relaxed min-sum decoder which has learned $\gamma = 0.863$, a relaxed min-sum decoder which is constrained to using $\gamma = 0.875$, and a relaxed NOMS decoder for the BCH (63,45) code. 45

4.8 Evolution of γ as training proceeds. 46
4.9 Comparison of decoder trained using cross-entropy loss and decoder trained using hinge loss. 47
4.10 Offset histogram for BCH (63,45) decoder after 3,000 minibatches. 48
4.11 Activations and gradients for a three-input variable node. 50
4.12 Activations and gradients for multiplicative and additive synapses. 50
4.13 Sigmoid neuron in a variable node. 51
4.14 Activations and (non-zero) gradients for a three-input min-sum check node after pruning. Note that there are very few non-zero gradients. 52

5.1 Comparison of FER for decoders for the (16,8) LDPC code trained with different values of λ. 57
5.2 Comparison of FER for decoders for the BCH (63,45) code trained with different values of λ. 58
5.3 Comparison of FER for decoders for the (128,64) polar code trained with different values of λ. 59
5.4 Comparison of FER for decoders for the (200,100) LDPC code trained with different values of λ. 60

Contents

1 Introduction **1**

2 Background **6**

2.1 Inference 6

2.1.1 Error-correcting codes 7

2.1.2 Bitwise MAP decoding 8

2.1.3 Belief propagation decoding 10

2.2 Learning 13

2.2.1 Gradient-based learning algorithms 13

2.2.2 Learning non-differentiable functions 20

3 A Survey of Neural Decoding Algorithms **22**

3.1 Learning to decode from scratch 22

3.2 Augmenting existing decoders with learning 26

3.2.1 Neural belief propagation decoding 27

3.2.2 Neural mRRD decoding 30

4 Neural Min-Sum Decoding **33**

4.1 Modifications to neural BP 33

4.1.1 Min-sum check nodes 33

4.1.2 Additive correction 36

4.1.3 Weight sharing 37

4.1.4 Relaxation 37

4.1.5 Alternative loss functions 40
4.2 Experimental results 40
4.2.1 Does learning help for other codes? 47
4.3 Discussion 48
4.3.1 Analyzing the learned behaviour 48
4.3.2 Backpropagation and sparsity of gradients 49

5 Learning from the Syndrome 53

5.1 Error correction as structured prediction 53
5.2 The syndrome loss 54
5.3 Experimental results 56
5.4 Discussion 59
5.4.1 Related work 59
5.4.2 Impact on BER and FER 61
5.4.3 Unsupervised learning 61

6 Conclusion and Future Work 62

Bibliography 63

Chapter 1

Introduction

Computers often need to communicate over noisy channels. For example, cell phones communicate with base stations using radio waves, and a stray radio wave from another source, such as the Sun, can combine with a cell phone transmission to distort the signal. The problem with using a noisy channel for communicating binary data, as a cell phone does, is that the noise can cause a 0 to look like a 1, or vice versa, thus introducing errors into messages. Suppose that a transmitter sends the message $1\ 0\ 1\ 0$ over a communication channel. Due to noise in the channel, the receiver might "mishear" the message as $1\ 0\ 1\ 1$. Error-correcting codes, invented by Richard Hamming [5], make it possible to extract the original signal from the noise, allowing reliable digital communication over unreliable channels.

The basic idea behind error-correcting codes is to add redundancy to a message so that the various messages a transmitter might send are easier to distinguish from each other, thus reducing the probability of an error. To understand how redundancy helps the receiver distinguish between messages, consider the length-3 repetition code, in which the transmitter redundantly sends each bit of a message three times. To send the message $1\ 0\ 1\ 0$, the transmitter first encodes the message into the sequence $111\ 000\ 111\ 000$. If a bit is flipped by noise, the receiver may observe a sequence such as $111\ 000\ 111\ 100$, but the receiver knows that the transmitter can't have sent 100 because 100 does not consist of the same bit repeated three times. In this case, the receiver should guess that the last triple was 000 before one of the bits was flipped by noise, as 000 is closer to 100 than

111. By making this correction, the receiver can infer that the original transmission was 111 000 111 000, and therefore the intended message was 1 0 1 0.

For any error-correcting code, the optimal method for recovering the original message, called "maximum *a posteriori* decoding", is to compute the probability that each bit was a 0 or a 1 and pick whichever hypothesis has the higher probability. Except for very short codes (such as the length-3 repetition code) or codes with a special structure (such as codes with cycle-free Tanner graphs) on well understood channels, it is usually not possible to perform maximum *a posteriori* decoding; the computation may involve an intractable number of terms, or the probability distribution of the channel noise may be unknown. In these situations, communication system designers have typically resorted to using suboptimal methods, some of which are based on manually designed heuristics. Such suboptimal methods often perform much worse than maximum *a posteriori* decoding.

In this thesis, we explore an alternative approach, which is to use algorithms that automatically learn from data, i.e. machine learning algorithms. Situations in which a probability distribution is i) intractible to maximize and/or compute or ii) unknown are good candidates for the application of machine learning. Additionally, the combination of new learning algorithms with fast, parallel hardware and large amounts of training data has enabled computer applications which were unthinkable only a few years ago, such as realistically rendering a normal photograph as a painting in the style of a famous artist (Fig. 1.1). We would like to know whether the same striking improvement in other applications due to machine learning can be achieved in error correction.

The idea of improving communication systems using machine learning is not new. For a long time, the digital communications research community has had a vague intuition that, because machine learning and information theory solve similar problems using similar statistical methods, it ought to be possible to use learning algorithms in error correction applications [6]. Wiberg *et al.*, describing potential hardware implementations of iterative decoding algorithms for codes on graphs, wrote: "Such implementations would look much like 'neural networks'; in fact, the prospect of such 'neuromorphic' decoders was an important motivation for this work [...]" [7]. Similarly, MacKay wrote in "Information Theory, Inference, and Learning Algorithms" [8]: "Why unify information theory and machine

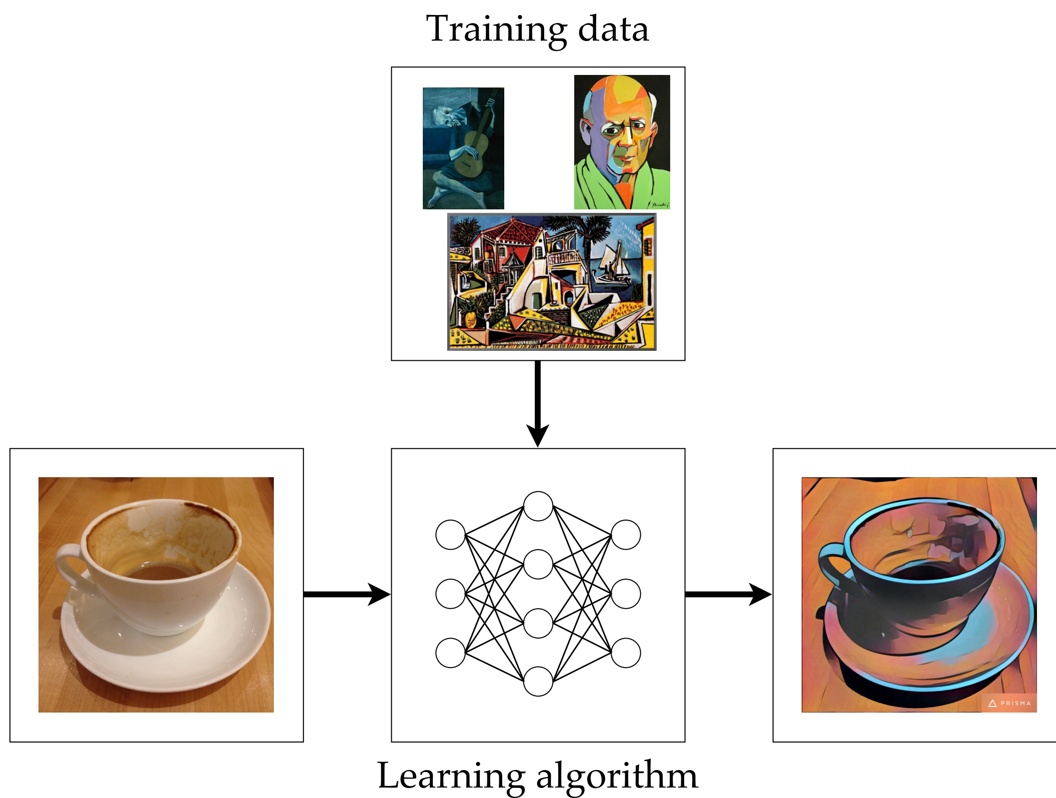


Figure 1.1: An appropriately constructed neural network (such as the one used in the Prisma app [3]) can learn to render a photograph in an artistic style.

learning? Because they are two sides of the same coin. [...] Brains are the ultimate compression and communication systems. And the state-of-the-art algorithms for both data compression and error-correcting codes use the same tools as machine learning."

While many attempts to use machine learning for error correction have been made, these attempts have mostly been thwarted by the "curse of dimensionality" for large input spaces [9]: there are 2^k possible codewords for a code which uses k bits per message, and a naïvely configured learning algorithm must see noisy examples of all 2^k codewords in order to learn how to correct errors in them, rendering these methods impractical for all but the shortest of message lengths. (When $k = 256$, 2^k is on the order of the number of atoms in the Universe; in comparison, the LDPC code used in 10 Gigabit Ethernet has $k = 1723$.)

A breakthrough came in [10], in which Nachmani *et al.* showed that the belief propagation decoding algorithm can be equipped with learnable multiplicative weights and trained as a neural network to achieve improved error correction performance. Since the performance of belief propagation does not depend on the transmitted codeword, and Nachmani *et al.*'s modification preserves this property, it suffices to train a neural belief propagation decoder using only the all-zeros codeword, thus lifting the curse of dimensionality and opening the way to applying machine learning to other problems in error correction and digital communications. Our main contribution in this thesis is to make modifications to Nachmani *et al.*'s approach that allow it to be implemented more efficiently, making it possible to use their idea in practical devices. We also describe a new gradient-based training method in which decoding is explicitly treated as a structured prediction problem rather than as a simple binary classification problem. We find that this method, which uses an unsupervised syndrome-based loss function, yields soft decoders with better frame error rates across all signal-to-noise ratios for a variety of codes.

The thesis is organized as follows. Chapter 2 covers the fundamental concepts in statistical inference and learning required for the rest of the thesis. Chapter 3 reviews the literature on decoding algorithms that use machine learning, with an emphasis on neural belief propagation. Chapter 4 presents our generalizations of and improvements to neural belief propagation. Chapter 5 describes the syndrome-based loss function and training

method. Finally, Chapter 6 concludes with a summary of the thesis and some ideas for future applications of machine learning to error correction.

Chapter 2

Background

In this chapter, we provide the mathematical background for the thesis. Specifically, we will discuss methods for statistical inference and machine learning.

2.1 Inference

Error correction is essentially an inference problem. By “inference”, we mean using probabilistic models to draw conclusions about random events. Inference is an important ingredient in communication systems because these systems have random behaviour. For example, in a speech recognition system, the objective might be to decide what words were spoken in a recording. What makes this difficult is that no two recordings of the same word sound exactly the same. We can *infer* what words were probably spoken using a model of what different words sound like and what words are likely to follow each other. Likewise, in a digital communication system, the receiver needs to decide what message was sent, given a noisy signal. We can *infer* what the original message was using a model of how transmitted signals are structured and how they are distorted by the communication channel. In this section, we discuss inference methods for communication systems that use error-correcting codes.

2.1.1 Error-correcting codes

We will first define some of the concepts mentioned in Chapter 1 more rigorously and introduce notation which will be used throughout the thesis. Generally, we will use capital letters to refer to matrices or random variables, lowercase letters to deterministic values or realizations of random variables, $p(\cdot)$ to denote a continuous probability density function, and $\Pr(\cdot)$ to denote the probability of a discrete event. We will consider communication systems which use binary linear block codes modulated using binary phase-shift keying (BPSK) and sent over memoryless channels affected by additive white Gaussian noise (AWGN).

In a communication system, the transmitter wishes to send a message $U \in \{0,1\}^k$ to the receiver, where k is the message length. To protect the message from noise, the transmitter encodes the message with some redundancy by multiplying U by a generator matrix G to yield a codeword $Z = UG$, where $Z \in \{0,1\}^n$, $G \in \{0,1\}^{k \times n}$, and n is the code length. The matrix multiplication is over the field GF(2)— in other words, $+$ maps to XOR, and $\cdot$ maps to AND.

The set of all possible codewords formed by multiplying a message by G is called the code C. Since the redundant encoding enables the receiver to locate and correct errors in a received word, C is called an error-correcting code. A matrix $H \in \{0,1\}^{(n-k) \times n}$ is a parity-check matrix for C if $GH^T = 0$ and $HZ = 0$ (again, over GF(2)) for all codewords $Z \in C$. The parity-check matrix is so named because each row represents a parity check, a check of whether a certain subset of the codeword bits has even parity.

Before transmission, Z is mapped from the unipolar format $\{0,1\}$ to a bipolar format $X = 1 - 2Z \in \{-1,+1\}^n$. (By abuse of notation, we will also call X a codeword and write $X \in C$.) X is then modulated to yield an analog signal and sent through the communication channel. As the signal passes through the channel, it is affected by noise; thus, the receiver's demodulator outputs not X but $Y = X + W$, where $W \in \mathbb{R}^n$ is a vector of Gaussian noise with mean 0 and variance σ^2.

The receiver has a component called the decoder which takes as input Y and outputs $\hat{U}$, the message it believes the transmitter sent. During the decoding process, the decoder

may maintain a “hard decision” $\hat{Z} \in \{0,1\}^n$ as the estimate of the transmitted codeword. (Likewise, $\hat{X} = 1 - 2\hat{Z}$ is the hard decision in the bipolar format.) The modulo-2 product $H\hat{Z}$ is called the syndrome. The presence of any 1s in the syndrome indicates that $\hat{Z}$ is not a codeword, and therefore one or more bits in $\hat{Z}$ must be incorrectly estimated. A soft decoder may also maintain a “soft decision” $S \in \mathbb{R}^n$, a vector with entries related to the probability of each codeword bit being a 0 or a 1.

2.1.2 Bitwise MAP decoding

Given a channel and a code, the designer of a communication system would ideally like to find a decoding algorithm which allows for messages to be recovered with as few errors as possible. One approach to this task is to look for a decoding algorithm which minimizes the probability of a bit error, that is, the probability of outputting a 0 when a 1 was sent, or vice versa.

The first decoder we will consider is the bitwise maximum *a posteriori* (MAP) decoder (see e.g. [11]). The MAP decoder is derived from the principle of MAP estimation in general inference problems, which dictates that one should guess that the event with the maximum posterior probability is the event which truly occurred. Bitwise MAP decoding applies the principle of MAP estimation to error correction by computing the posterior probability of each codeword bit being a 0 or a 1 and choosing whichever hypothesis has a higher probability. This decoder is optimal in the following sense:

Lemma 2.1.1. *Let* $\hat{X}_{i,\text{ MAP}} = \underset{x_i=\{-1,+1\}}{\operatorname{argmax}} \Pr(X_i = x_i|Y = y)$ *for* $i \in \{1,\ldots,n\}$. *A decoder which outputs* $\hat{X}_{i,\text{ MAP}}$ *as the estimate for* X_i *has the minimum probability of making a bit error among all possible decoders.*

Proof. Note that $P_{err,\text{ MAP}}$, the probability of the bitwise MAP decoder making a bit error, is equal to $1 - \Pr(X_i = \hat{X}_{i,\text{ MAP}}|Y = y)$. Let $\hat{X}_{i,\text{ other}}$ be the bit chosen by any other decoding method. Then $P_{err,\text{ other}} = 1 - \Pr(X_i = \hat{X}_{i,\text{ other}}|Y = y)$ is the probability of this other decoding method making an error. Since by definition $\Pr(X_i = \hat{X}_{i,\text{ MAP}}|Y = y) \geq \Pr(X_i = \hat{X}_{i,\text{ other}}|Y = y)$, we know that $\Big(1 - \Pr(X_i = \hat{X}_{i,\text{ MAP}}|Y = y)\Big) \leq \Big(1 - \Pr(X_i = \hat{X}_{i,\text{ other}}|Y = y)\Big)$, and therefore $P_{err,\text{ MAP}} \leq P_{err,\text{ other}}$. $\square$

By an argument similar to the one given for the bitwise MAP decoder, the codeword MAP decoder (that is, the decoder which chooses the codeword with the highest probability) has the lowest frame error rate among all possible decoders. It is often assumed that all codewords are equally likely, in which case choosing the codeword with the maximum posterior probability is equivalent to choosing the codeword with the maximum likelihood (ML). This decoder is referred to as the ML decoder. For now, we will consider only bitwise MAP decoding.

To perform bitwise MAP decoding, the decoder must compute $\Pr(X_i = +1|Y = y)$ for each $i \in \{1, \ldots, n\}$. The probability that $X_i = +1$ can be expressed as the probability that the transmitter sent one of the codewords in which the i'th bit is $+1$. Therefore:

$$\Pr(X_i = +1|Y = y) = \sum_{x' \in C} \Pr(X = x'|Y = y)\mathbb{I}(x'_i = +1). \tag{2.1}$$

where $\mathbb{I}(\cdot)$ is the indicator function.

The probability that a particular codeword was sent can be computed using Bayes' rule:

$$\Pr(X = x'|Y = y) = \frac{p(Y = y|X = x')\Pr(X = x')}{p(Y = y)}. \tag{2.2}$$

We know that $\Pr(X = x') = \frac{1}{|C|}$ since all codewords are assumed to be equally likely, and $p(Y = y)$ does not depend on the transmitted codeword, so these terms can be lumped into a single proportionality constant α:

$$\Pr(X = x'|Y = y) = \alpha \cdot p(Y = y|X = x'), \tag{2.3}$$

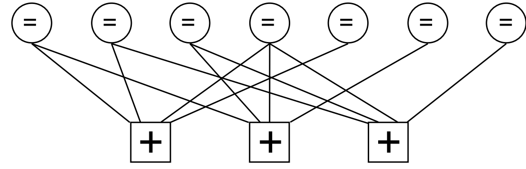
and therefore $\Pr(X_i = +1|Y = y)$ is proportional to the sum of the likelihoods of the possible codewords:

$$\Pr(X_i = +1|Y = y) = \alpha \cdot \sum_{x' \in C} p(Y = y|X = x')\mathbb{I}(x'_i = +1). \tag{2.4}$$

We have assumed that the channel is a memoryless AWGN channel, so the likelihood of the received vector is equal to the product of the likelihoods of the received values:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

(a) A parity-check matrix.



(b) Corresponding Tanner graph.

Figure 2.1: Parity-check matrix and Tanner graph for the (7,4) Hamming code.

$$p(Y = y|X = x') = \prod_{i=1}^{n} p(Y_i = y_i|X_i = x'_i) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y_i - x'_i)^2}{2\sigma^2}}. \quad (2.5)$$

Then the probability that $X_i = +1$ is proportional to the sum of the products:

$$\Pr(X_i = +1|Y = y) = \alpha \cdot \sum_{x' \in C} \prod_{i=1}^{n} p(Y_i = y_i|X_i = x'_i) \mathbb{I}(x'_i = +1) \quad (2.6)$$

Similarly, the probability that $X_i = -1$ is proportional to the sum of the likelihoods of the codewords for which the i'th bit is -1. The MAP bit estimate can be computed by picking the bit with the higher probability.

2.1.3 Belief propagation decoding

It is generally not possible to compute Equation 2.6 directly because of the sum over all possible codewords. However, for certain codes, the sum can be computed efficiently by expressing the code in a graphical form and running the belief propagation (BP) algorithm [12]. BP is a general algorithm for computing the probabilities of different events described by a probabilistic graphical model. The graphical model used in BP decoding is called a Tanner graph [13] (Fig. 2.1). A Tanner graph is a bipartite graph, isomorphic to a parity-check matrix for a certain code, composed of two types of nodes: variable nodes, which correspond to columns of the parity-check matrix, and check nodes, which correspond to rows of the parity-check matrix. If the (c,v)-th element of the parity-check matrix is a 1, then the c'th check node is connected to the v'th variable node by an edge in the Tanner graph.

BP decoding operates by passing probability messages along the edges of the Tanner graph. To simplify the implementation, the computations are performed using log-likelihood ratios (LLRs) rather than probabilities. Decoding begins by putting the noisy received values in LLR format. The LLR of the v'th received value l_v is computed using

$$l_v = \log\left(\frac{p(Y_v = y_v|X_v = +1)}{p(Y_v = y_v|X_v = -1)}\right) = \frac{2y_v}{\sigma^2}. \tag{2.7}$$

Next, probability messages are exchanged between variable nodes and check nodes to update the decoder's soft output. At each iteration t, the decoder produces a soft decision vector s^t that converges to the posterior LLRs over the iterations.

The message $\mu^t_{v\to c}$ from variable node v to check node c for iteration t is computed using

$$\mu^t_{v\to c} = l_v + \sum_{c'\in\mathcal{N}(v)\backslash c} \mu^{t-1}_{c'\to v}, \tag{2.8}$$

where $\mathcal{N}(v)$ is the set of CNs connected to VN v, and $\mu^0_{c\to v} = 0$.

The message $\mu^t_{c\to v}$ from CN c to VN v for iteration t is computed using

$$\mu^t_{c\to v} = 2\tanh^{-1}\left(\prod_{v'\in\mathcal{M}(c)\backslash v} \tanh\left(\frac{\mu^t_{v'\to c}}{2}\right)\right), \tag{2.9}$$

where $\mathcal{M}(c)$ is the set of VNs connected to CN c.

Finally, the soft output for iteration t is computed using

$$s^t_v = l_v + \sum_{c'\in\mathcal{N}(v)} \mu^t_{c'\to v}. \tag{2.10}$$

The recovered binary word can be extracted from s^t using a hard decision:

$$\hat{z}^t_v = \begin{cases} 0, & \text{if } s^t_v > 0 \\ 1, & \text{otherwise.} \end{cases}$$

It is possible that the Tanner graph for a given parity-check matrix contains cycles. A cycle is a path through the graph that starts and ends at the same node (see Fig. 2.2). If there are no cycles in the Tanner graph, then the values of the soft bit estimates s^t converge to the MAP bit estimates, and therefore a hard decision on these estimates has

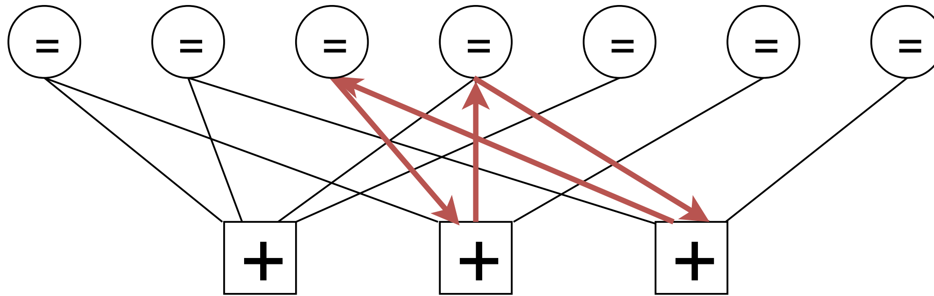


Figure 2.2: A path traversing variable node 3, check node 2, variable node 4, and check node 3 forms a cycle in a Tanner graph for the Hamming code.

the lowest possible probability of making a bit error. If there are cycles in the graph, BP is not guaranteed to produce the MAP estimates, and the decoder is suboptimal as a result.

We can explain through an analogy why "loopy BP" (BP applied to a Tanner graph with cycles) results in a suboptimal decoder. Suppose that you hear a rumor from person A, and then you tell the rumor to person B. If later you hear the same rumor from person C, your belief in the rumor may be made more confident, since you have heard the rumor not only from person A but also from person C. If person A's information and person C's information are independent, then it makes sense to become more confident in your belief in the rumor. However, if person C heard the rumor from person B (who originally heard it from you), or if person C heard the rumor from person A, it doesn't make sense to become more confident about the rumor. In the same way, a cycle in the Tanner graph causes nodes to incorrectly update their own beliefs based on beliefs that they themselves have propagated.

While loopy BP is not guaranteed to produce the MAP bit estimates, for certain codes (turbo codes and LDPC codes) the algorithm achieves nearly optimal performance [14]. For these codes, the Tanner graph is very large and sparse, and any cycles through this graph become effectively "infinitely long". (In terms of the rumor analogy, it takes a very long time for your own rumor to make it back to you, so most of the time, you are spreading mostly new information and aggregating information correctly.) For short codes and codes with dense parity-check matrices, on the other hand, loopy BP does not work well. We are concerned in this thesis with finding good decoding methods for this latter case.

Specifically, we will show that machine learning can be used to design improved decoders for these short high-density codes. The next section gives a little more background on machine learning and why it may be useful here.

2.2 Learning

Machine learning can be thought of as the automatic construction of functions from data, effectively allowing computers to learn from the data without being directly programmed by a human. In contrast to traditional algorithms, which simply evaluate a fixed function f, learning algorithms operate in two modes: a learning or training mode, in which the algorithm constructs f using training data, and an inference mode, in which the algorithm evaluates f for some input. In "offline" learning, the algorithm operates in the learning mode only once, and the model is fixed and used for all subsequent inference. In "online" learning, the learning mode is used to improve the model even as the algorithm makes predictions about incoming data.

2.2.1 Gradient-based learning algorithms

A common approach to machine learning is to assume that the optimal inference function f has a certain form, with unknown parameters, and choose the parameters so that the function performs as well as possible for its task. As an example of this approach, suppose that we would like to make an accurate detector for a signal used in an "on-off keying" system— that is, a communication system in which the presence of a certain signal indicates a 1 and the absence of that signal indicates a 0— and further suppose that we do not know the exact form of the signal. We may have a database of noisy received vectors as training examples to use when designing the detector in which the training examples are simply labeled as having the signal present or not present. In this case, we could assume that the signals were sent through an AWGN channel and use the following model:

$$p(Y = y| \text{ signal }) = \frac{1}{(2\pi\sigma^2)^{\frac{n}{2}}} e^{\frac{-(y-s)^T(y-s)}{2\sigma^2}}, \tag{2.11}$$

$$p(Y = y | \text{ no signal }) = \frac{1}{(2\pi\sigma^2)^{\frac{n}{2}}} e^{\frac{-y^T y}{2\sigma^2}}, \tag{2.12}$$

where s is the vector corresponding to the waveform of the noise-free signal. We can use MAP estimation to guess whether the signal is present or not by computing the posterior probability:

$$\begin{aligned}
\Pr(\text{ signal } | Y = y) &= \frac{p(Y = y | \text{ signal })\Pr(\text{ signal })}{p(Y = y)} \\
&= \frac{p(Y = y | \text{ signal })\Pr(\text{ signal })}{p(Y = y | \text{ signal })\Pr(\text{ signal }) + p(Y = y | \text{ no signal })\Pr(\text{ no signal })} \\
&= \frac{1}{1 + \frac{p(Y=y | \text{ no signal })\Pr(\text{ no signal })}{p(Y=y | \text{ signal })\Pr(\text{ signal })}} \\
&= \frac{1}{1 + \frac{\Pr(\text{ no signal }) \; e^{\frac{-y^T y}{2\sigma^2}}}{\Pr(\text{ signal }) \; e^{\frac{-(y-s)^T (y-s)}{2\sigma^2}}}} \\
&= \frac{1}{1 + e^{\log \frac{\Pr(\text{ no signal })}{\Pr(\text{ signal })} + \frac{-2s^T y + s^T s}{2\sigma^2}}}
\end{aligned} \tag{2.13}$$

To simplify the form of the model, let $w = \frac{s}{\sigma^2}$ and $b = -\frac{s^T s}{2\sigma^2} - \log \frac{\Pr(\text{ no signal })}{\Pr(\text{ signal })}$. The entries of w are called "weights", and b is called the "bias." Using this reformulation, we can rewrite Equation 2.13 as

$$f(y) = \Pr(\text{ signal } | Y = y) = \frac{1}{1 + e^{-(w^T y + b)}}. \tag{2.14}$$

This is known as a logistic regression model. The function $\sigma(a) = \frac{1}{1+e^{-a}}$ used by the model is the logistic sigmoid function (often referred to as *the* "sigmoid" function, although really any "S"-shaped function is a sigmoid function). The model is also known as an artificial neuron because of its resemblance to a biological neuron (cf. Fig. 2.3 and Fig. 2.4).

It is necessary to estimate w and b before the neuron can actually be used. We will choose values for the model parameters which make the model match the distribution of the data. To do this, we need a metric for how well a model probability distribution

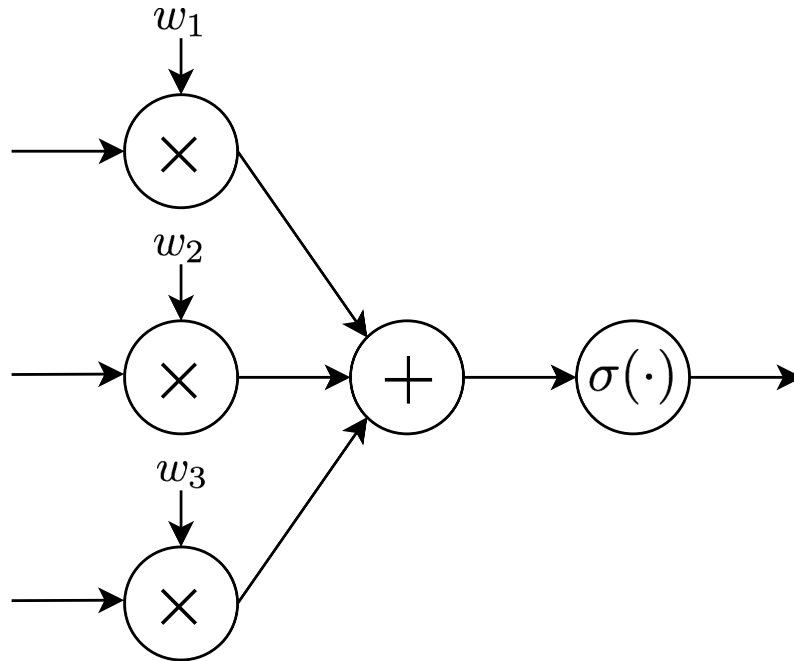


Figure 2.3: Artificial neuron.

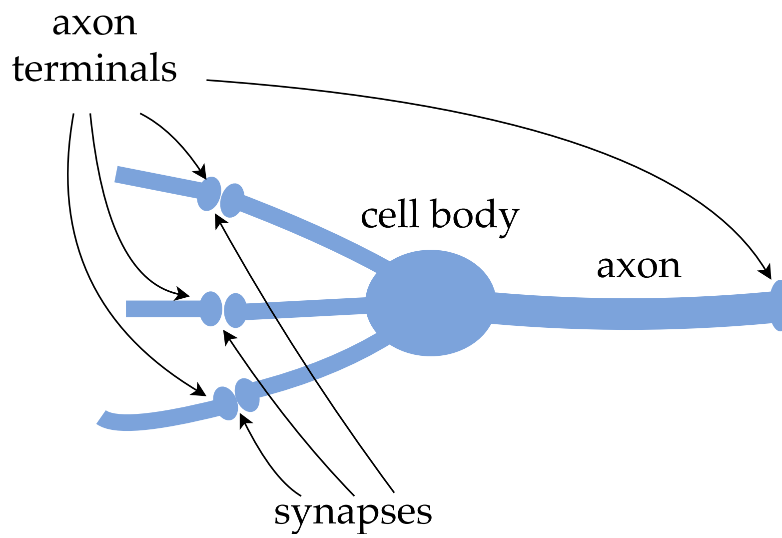


Figure 2.4: Biological neuron.

matches a true probability distribution. The Kullback-Leibler (KL) divergence, also known as information gain or relative entropy, is one such metric [15]. The KL divergence between a distribution p and another distribution q is defined as

$$D_{KL}(p||q) = \mathbb{E}_p \left[\log \frac{p(x)}{q(x)} \right]. \tag{2.15}$$

The KL divergence is always greater than or equal to 0; when it equals 0, the two distributions are identical. Therefore, we should minimize the KL divergence to cause the model distribution to match up with the true distribution. The algorithm we will use to minimize the KL divergence is called stochastic gradient descent (SGD) [16]. SGD is a simple iterative optimization algorithm which consists of repeating the following two steps:

1. Compute $\nabla J(w)$ for a random minibatch of training examples

2. Set w equal to $w - \eta \nabla J(w)$

where J is the loss function to be minimized (in this case, the KL divergence), w is the parameter vector, and η is a small step size or "learning rate". The updates continue until the value of J is satisfactorily low. An illustration of gradient descent is given in Figure 2.5.

Gradient descent is not always the best optimizer for learning: whereas gradient-based methods use only the first derivatives of the loss function, second-order methods, such as Newton's method, also take the second derivatives (and hence the curvature of the loss function) into account, resulting in faster convergence. However, gradient-based optimization is an excellent general-purpose tool, and sometimes works in scenarios where more sophisticated methods do not, such as when the Hessian of the loss function is dense and unstructured or the loss function is non-convex.

A challenging issue in gradient-based optimization is the avoidance of local minima, i.e. values of the loss function which are locally good but globally poor. Gradient descent can avoid getting stuck in a local minimum using "momentum" in the update rule: rather than using the raw gradient, the moving average of the gradients is computed and used

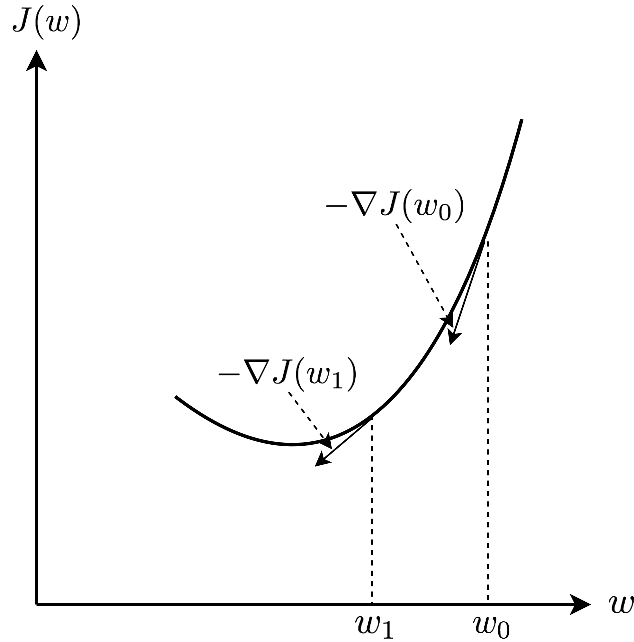


Figure 2.5: Illustration of gradient descent.

to update the parameters. Like a ball rolling down a hill, building up momentum and not settling in small dips in the hill, a parameter vector updated using momentum can avoid local minima in the loss function if sufficiently large gradients have been seen in the recent past. Some update rules, such as Adam [17], also track the variance of the gradients over time.

Note that

$$\begin{aligned} \underset{q}{\operatorname{argmin}}\, D_{KL}(p||q) &= \underset{q}{\operatorname{argmin}}\, \mathbb{E}_p \left[\log \frac{p(x)}{q(x)} \right] \\ &= \underset{q}{\operatorname{argmin}} -\mathbb{E}_p \left[\log q(x) \right] + \mathbb{E}_p \left[\log p(x) \right] \\ &= \underset{q}{\operatorname{argmin}} -\mathbb{E}_p \left[\log q(x) \right], \end{aligned} \tag{2.16}$$

where the last expression is known as the cross-entropy. Since minimizing the cross-entropy is equivalent to minimizing the KL divergence, and the form of the loss function is slightly simpler, usually the cross-entropy is considered instead of the KL divergence.

A problem arises when attempting to compute the cross-entropy: the expectation is taken with respect to the target distribution p, but this distribution is unknown— after all, if p were known, there would be no need for learning. Therefore, an approximation to the

cross-entropy is computed using the labels of the training data instead of the true posterior probability. For a training example marked "signal", the true probability is replaced with 1, and for a training example marked "no signal", the true probability is replaced with 0. This is known as the principle of empirical risk minimization.

When empirical risk minimization is used, the loss function to be minimized is:

$$J(w) = -\frac{1}{M}\sum_{i} z_i \log f(y_i) + (1 - z_i)\log(1 - f(y_i)), \tag{2.17}$$

where i indexes over the training examples in a minibatch of size M, and z_i is the label for training example i (0 if "no signal", 1 if "signal"). Fig. 2.6 shows the weights of a sigmoid neuron learning to distinguish "signal" from "no signal" using only the noisy training examples and SGD with empirical risk minimization. As the MAP weights are equal to $\frac{s}{\sigma^2}$, not surprisingly, the weights take on the shape of the noise-free signal.

The sigmoid neuron is considered a linear model because it uses an affine transform ($w^T y + b$) to compute a "score" for the input. While linear models can solve many problems, they are limited in the sense that they can only learn to distinguish patterns which are linearly separable, i.e., capable of being separated by a hyperplane. In this example, the training examples in which a signal was present were data points clustered around a certain center point in a high-dimensional space (the noise-free signal vector), and the examples with no signal were clustered around another point (the zero vector); it is easy to separate points from these two clusters by drawing a hyperplane between them. If the signal could take on different frequencies or phase shifts, it would be much more difficult to classify using a linear model. It would be nice if such a dataset could be made linearly separable somehow.

Luckily, even if a training set is not linearly separable, it is generally possible to make it linearly separable by applying some high-dimensional nonlinear transformation. Precisely which nonlinear transformation should be used, however, is unknown and depends on the dataset. Some learning algorithms, such as kernel methods, require the designer to choose a nonlinear transformation, which may require some manual tuning. It is also possible to learn the transformation automatically using a feedforward neural network. In a feedforward neural network, rather than using a single neuron, a collection of neurons

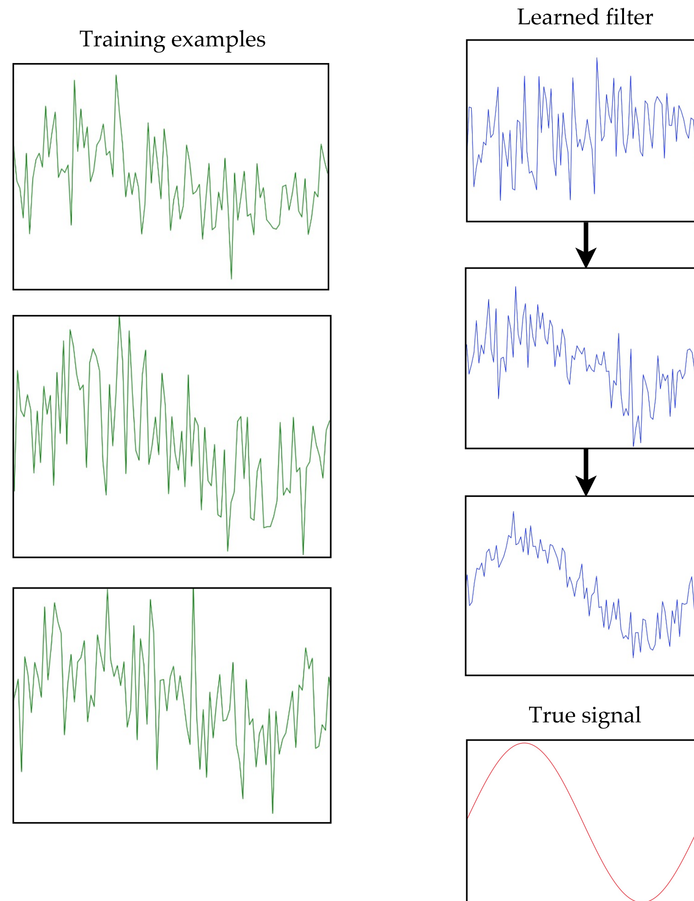


Figure 2.6: Noisy training examples (left) and the pattern learned by the neuron converging to the true underlying signal (right).

process the input, and their outputs, taken together, form the input vector to another neuron or set of neurons. Each set of neurons is called a "hidden layer", and the output of a neuron is called an "activation". The hidden layers warp the distribution of the input in such a way that the output of the final hidden layer is more suitable for use in a linear model. Neural networks are differentiable models, and hence can be trained using gradient descent. The gradient of the loss function with respect to a neural network's parameters can be computed efficiently using the "backpropagation" algorithm. Partial derivatives flow backward through the network during backpropagation in a manner similar to the way activations flow forward during inference, and are referred to as "gradients".

While individual neurons can be interpreted as feature detectors, as in the example above, many practioners of neural networks forgo the probabilistic interpretation and the notion of a "neuron" entirely and instead think of the layers of the network as performing linear and nonlinear parts of a nested function approximation. Since any continuous function can be represented using a weighted sum of sigmoidal functions, a feedforward network with a single, sufficiently large hidden layer is a universal function approximator, and can theoretically be made to learn any function [18]. (Because all that is necessary to achieve this property is a nonlinear activation function, usually a simpler activation function is used, most often the rectified linear unit function $\mathrm{ReLU}(a) = \max(0, a)$.) However, there is a growing body of theoretical and empirical evidence that deep learning architectures (neural networks with many layers) are capable of learning and representing functions more efficiently than shallow architectures [19]. The state-of-the-art results in perceptual tasks, such as computer vision, have been achieved using very deep networks. Recurrent neural networks, in which the output is fed back into the network, are, in a sense, infinitely deep, and are in fact Turing-complete [20]. This makes them very strong candidates for use in challenging inference tasks, such as error correction.

2.2.2 Learning non-differentiable functions

Suppose that the function to be learned (and, by extension, the loss function) is not differentiable with respect to its parameters at some point(s). In that case, we cannot use

a gradient-based method in the learning mode. While it is possible to optimize a non-differentiable function using random search or heuristics such as genetic algorithms, it would be preferable to use gradient-based methods, which scale well with the size of the optimization problem.

By making a small modification to the gradient descent algorithm, it is possible to learn a certain class of non-differentiable functions. The modification applies to functions with "kinks." Kinks are points at which the function is continuous but the derivative does not exist. When the function to be minimized has an input at which there is a kink, a subgradient can be used in place of the gradient during gradient descent. A subgradient for f at x_0 is any vector g such that $f(x) - f(x_0) \geq g^T(x - x_0)$, for all x. Informally, a subgradient is the slope of a hyperplane which fits below the function. (Similarly, a supergradient is the slope of a hyperplane which fits above the function. For simplicity, we will refer in this thesis to both subgradients and supergradients as "subgradients".) By using subgradients, activation functions with kinks (such as ReLU) can be used in neural networks and still allow gradient-based training, as in [21].

Chapter 3

A Survey of Neural Decoding Algorithms

An error-correcting decoder applies a certain function to a received signal in an attempt to recover the original message. How well the decoder performs depends on the function used. The optimal function in terms of bit error rate is $\underset{x_i=\{-1,+1\}}{\operatorname{argmax}} \Pr(X_i = x_i | Y = y)$, but as we have seen, this function is generally too computationally complex to implement exactly in practical devices. Designing practical decoders with close to optimal performance is a challenging task because it is unknown what functions can approximate the bitwise MAP decoder.

As we saw in Chapter 2, machine learning can be thought of as constructing a function from data. In the light of recent machine learning successes in tasks for which constructing a function by hand is difficult (such as speech and image processing), one might ask: is it possible to construct a good error-correcting decoder from data? In other words, is it possible to "learn to decode"? Here, we review several existing "neural" (learning) decoding algorithms and describe their advantages and disadvantages.

3.1 Learning to decode from scratch

We first consider approaches which use "off-the-shelf", general-purpose learning algorithms to implement decoding.

In a general survey of applications for neural networks [22], Lippmann showed that a simple neural network ("Hamming net") can be configured to output which pattern

in a set of binary patterns has the minimum Hamming distance to a given input. The network is a linear model with one output unit corresponding to each possible pattern. The network is not trained; rather, the weights are simply computed directly from the patterns to be recognized. Zeng *et al.* recognized in [23] that this configuration could be used to decode both linear and nonlinear error-correcting codes. Such an approach is necessarily impractical for large value of k, as 2^k output units are required. This is not surprising, since this decoding method is equivalent to simple minimum distance ML decoding. At around the same time, Bruck and Blaum explored the connections between neural networks and error-correcting codes in [24], and showed that ML decoding of error-correcting codes is equivalent to maximizing the energy of a certain type of neural network (a Hopfield network), but do not explore efficient neural network decoder architectures.

The first work to propose training a fully-connected feedforward neural network using backpropagation and gradient descent to decode was [25]. In this work, Caid and Means describe a network with n inputs, a single hidden layer, and k outputs, directly predicting the transmitted message from the received word. This is a significant improvement over the previous approaches, which require 2^k outputs. The authors of [25] show that their approach is effective for the (7,4) Hamming code and a small convolutional code. (There have been many such neural decoders for convolutional codes, especially turbo codes: cf. [26], [27], and [28]). However, they do not attempt to decode larger codes or note how well the decoder generalizes to codewords not seen during training. Such an analysis is provided in [29], in which Stefano *et al.* find that a neural network for decoding the Hamming code could not successfully decode received words corresponding to codewords never shown during training.

The failure of the network in [29] to generalize to new codewords highlights the crucial difficulty of using learning algorithms mentioned in Chapter 1: unless an algorithm can learn decoding concepts which apply to all codewords and not just to particular codewords, it may need to see examples of noisy received vectors corresponding to all possible codewords, which is impractical for any reasonably sized code. Esposito *et al.* showed in [30] that for some codes, the structure of the code allows for $\frac{|C|}{2}$ codewords to be used in training, rather than $|C|$ codewords. This is not much of a reduction, but it is an impor-

tant recognition that incorporating knowledge of the code into the learning algorithm can reduce the amount of training data required.

Another feedforward network with a single hidden layer trained to decode the Hamming code is described in [31]. Unlike the previous decoders, this work shows that neural networks can learn to decode in the presence of non-white Gaussian noise, that is, noise which is correlated between samples. Their network does not require any special modification to handle the correlated noise; they simply used conventional gradient descent, and the network learned to compensate for the correlations. This work shows that learning algorithms may be useful for channels with unusual noise distributions, which may not be easily accommodated in traditional decoding algorithms.

The conventional feedforward network is not the only neural network architecture which has been used; in [32], El-Khamy *et al.* use a time-delay neural network, which takes as input a window of received values and outputs an estimate, while shifting the window over the length of the block. Likewise, Hämäläinen and Henriksson used a recurrent neural network [33], and in [34], Abdelbaki *et al.* use a recurrent "random neural network" which operates using spikes of unit amplitude. In [35], Wu *et al.* use a combination of high-order polynomials and genetic algorithms to decode. All of these approaches are limited to very short codes.

In [36], Tallini and Cull trained a feedforward network with two hidden layers to take as input a syndrome and produce as output an error pattern, which is combined with the received word to correct errors. Using the syndrome as input changes the rank of the input from k to $n-k$, which for codes with rate greater than $\frac{1}{2}$ is a reduction. For codes with reasonably large values for $n-k$, this method can still require an impractically large number of training codewords. During training, they use the sigmoid activation function for all neurons; during decoding, they replace the sigmoid with a hard threshold, which allows the network to be implemented more efficiently. Another interesting aspect of their neural decoder is that it is able to correct 26% of error patterns of weight 4 for a (32, 16, 8) Reed-Muller code, even though it was only trained using codewords with error patterns of up to weight 3.

An unconventional use for neural networks in decoders was demonstrated in [37].

In this work, Buckley and Wicker propose using a neural network in a turbo decoder to monitor the decoding process and predict whether there are errors in the soft output. Their method forgoes the use of a CRC code (the conventional method for error detection in such decoders), allowing higher transmission rates.

In [9], Gruber *et al.* present an in-depth study of deep feedforward networks trained to decode linear codes with $n = 16$ and $k = 8$. They compare the performance of a network trained to decode structured codes (specifically, polar codes) and a network trained to decode random codes. They show that, when a portion of the code is withheld during training, the random code decoder completely fails when presented with codewords not used in training. In contrast, the polar code decoder can, to a certain extent, successfully decode new codewords. This suggests that the polar code decoder is making use of the code structure rather than simply memorizing the codewords, which suggests that such an approach might be used for decoding practical polar codes. Unfortunately, even for structured codes their neural decoder does not scale to moderate block lengths— a large portion of the code must be seen for the network to be able to generalize.

O'Shea *et al.* in [38] have gone beyond simply applying learning to the decoder and have recast the complete communication system as a "channel autoencoder", in which the transmitter and receiver jointly learn to communicate reliably for a given channel without the aid of a human-designed code. They perform an experiment which shows that it is possible to use a feedforward networks to implement the transmitter and receiver, which jointly learn a soft coding scheme. The idea of an end-to-end learning communication system is intriguing, but for this thesis, we restrict ourselves to considering the common case when the code and modulation scheme are already decided upon, and focus on improving the performance of the decoder.

All of the approaches described above are essentially limited by the "curse of dimensionality" of needing to learn all possible codewords as described in [9]. Even if the off-the-shelf neural network approaches manage to avoid this curse of dimensionality, there are two reasons to believe that an off-the-shelf architecture is not the right choice for the problem of error correction.

First, the XOR function, which is an essential ingredient in binary linear codes, is dif-

ficult for neural networks to learn. Computing the XOR of several inputs using a shallow feedforward network is difficult because the XOR function requires many logic gates to be implemented in a circuit of short depth [39]— indeed, this fact was presented as one of the original theoretical arguments for using deep learning architectures rather than shallow ones [40]. Therefore, a significant amount of a neural network decoder's computational resources may be dedicated to simply computing XOR.

Second, the problem domain of channel decoding is fairly different from the problem domains where conventional deep networks have been found to work well, namely, processing "natural" signals (images, sounds, language, etc.). Natural signals typically have a complex, multifaceted surface structure, but the underlying structure of these signals is often actually very low-dimensional. (For example, speech is produced through movement of the tongue and the vocal cords, which can be described using a small number of parameters [41].) Coded messages, on the other hand, are known to have dimension k, where $\frac{k}{n}$ is typically a reasonably large fraction such as $\frac{1}{4}$, $\frac{1}{2}$, or $\frac{3}{4}$, and all 2^k possibilities are assumed to be equally likely.

On the other hand, there are reasons to believe that learning to decode from scratch may eventually outperform traditional decoding algorithms. Neural networks are (within a single layer) very parallelizable— for codes with highly sequential decoding algorithms (such as polar codes), a neural network could theoretically enable lower latency decoding. Also, because the function learned by a fully-connected neural network is not "biased" by the designer towards existing decoders, it is possible that by not relying on our assumptions, a neural network could learn a high-performance decoding function too complicated for a human to invent.

3.2 Augmenting existing decoders with learning

An alternative method for learning to decode is to augment an existing suboptimal error correction algorithm with learning capability in the hope that it will learn to perform more like an optimal algorithm. In this section, we focus on two such methods: neural BP and neural mRRD.

3.2.1 Neural belief propagation decoding

The decoder described in this section was developed as an attempt to solve the problem of soft decoding of codes with short block lengths. For certain applications, short blocks are desirable (for instance, in communication systems where very low latency is required). Modern codes, such as turbo codes, LDPC codes, and polar codes, typically only attain good performance at relatively long block lengths. At short to moderate block lengths, high-density parity-check (HDPC) codes such as BCH codes and Reed-Solomon codes may be a better choice due to their high minimum distance property [42]. While there exist low-complexity hard decision algorithms for such codes, these algorithms do not achieve the full error-correction power that short HDPC codes have when decoded optimally. There have been several soft decoding algorithms for HDPC codes with high performance (such as adaptive BP [43]), but these algorithms have correspondingly high complexity. Simple conventional BP can be used for HDPC codes, but it yields very poor results.

Nachmani *et al.* suggest a modification to BP which significantly improves its performance for HDPC codes. In their approach, which we refer to as "neural belief propagation" or "neural BP", the messages are computed in the same manner as in traditional LLR-based BP and subsequently multiplied by a set of weights. Thus, to compute messages and outputs, the BP update equations (see Chapter 2) (2.8), (2.9), and (2.10) are replaced with (3.1), (3.2), and (3.3), respectively:

$$\mu^t_{v\to c} = \tanh\left(\frac{1}{2}\left(w^t_{v,in} l_v + \sum_{c' \in \mathcal{N}(v)\setminus c} w^t_{c'\to v,c}\,\mu^{t-1}_{c'\to v}\right)\right) \tag{3.1}$$

$$\mu^t_{c\to v} = 2\tanh^{-1}\left(\prod_{v' \in \mathcal{M}(c)\setminus v} \mu^t_{v'\to c}\right) \tag{3.2}$$

$$s^t_v = w_{v,out} l_v + \sum_{c' \in \mathcal{N}(v)} w_{c'\to v,c,out}\,\mu^t_{c'\to v}, \tag{3.3}$$

where $w^t_{v,in}$, $w_{c'\to v,c}$, $w_{v,out}$, and $w_{c'\to v,c,out}$ are the aforementioned multiplicative weights. The weights are unique for each iteration; hence, as a composition of affine transformations and nonlinearities, the neural BP decoder can be considered a kind of non-fully-

connected feedforward neural network, as shown in Fig. 3.1. Each layer of the network is composed of a learnable message passing operation.

As pointed out by [44], neural BP is an instance of “deep unfolding” [45], a methodology in which an iterative algorithm (which may or may not be a learning algorithm) is unfolded and learnable parameters are assigned to each iteration to yield a deep architecture. Deep unfolding was previously applied to BP in works including [45], [46], [47], and [48] for problems outside of error correction.

A key property of the neural BP decoder is that, because the network connectivity is based on the Tanner graph, the learning algorithm does not need to learn the structure of the code, and it suffices to train the decoder using noisy versions of a single codeword, such as the all-zeros codeword. This makes it possible to train a decoder for codes with reasonably large values for k, which is not possible for the other neural decoders described above.

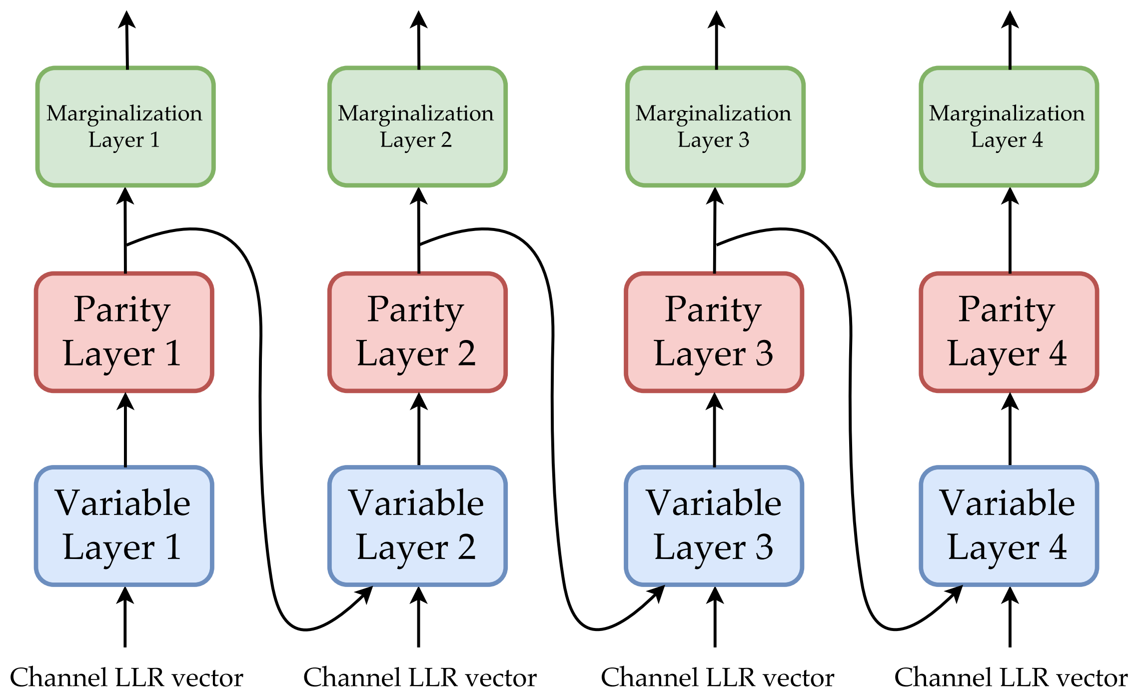


Figure 3.1: A four iteration neural BP decoder. (Figure adapted from [4].)

Fig. 3.2 shows the improvement in BER for a BCH (63,36) code unrolled to 5 iterations when decoded using neural BP. (The standard BP decoder could only attain the same

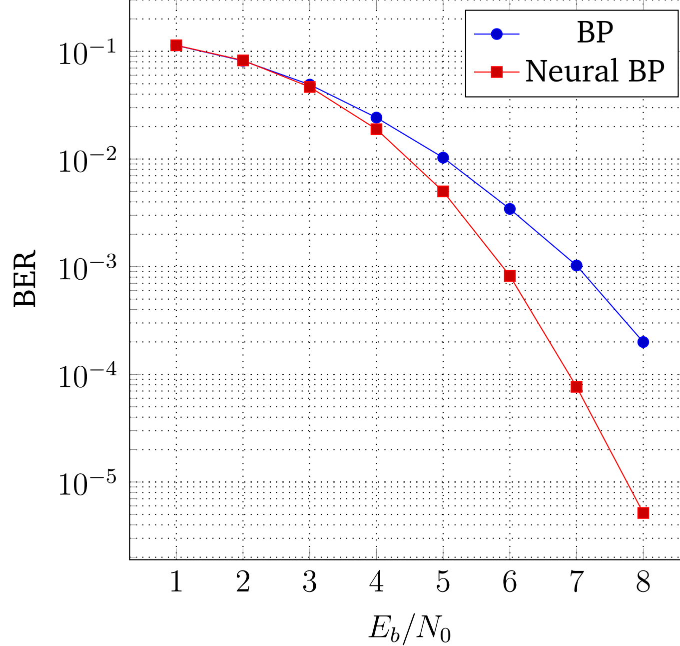


Figure 3.2: BER for BCH (63,36) code.

performance after 50 iterations.) Similar improvements were obtained for the (15,11), (63,45), (127,106), (127,64), (127,99) BCH codes. The parity-check matrices used for these codes were their high-density cyclic versions (found online at [49]). Nachmani *et al.* later showed in [4] that neural BP still significantly outperformed normal BP even when an improved, sparsified version of these matrices was used with short cycles removed.

To train the neural BP decoders, Nachmani *et al.* used SGD to minimize the average cross-entropy between each decoder soft output (passed through the sigmoid function to yield values in the range $[0, 1]$) and each corresponding bit of the transmitted codeword:

$$\ell_{\text{cross-entropy}}(z, s) = -\frac{1}{n} \sum_{v=1}^{N} z_v \log \sigma(-s_v^T) + (1 - z_v) \log (1 - \sigma(-s_v^T)), \tag{3.4}$$

where T is the index of the final decoding iteration. (Note that we multiply the soft output by -1 so that $\sigma(\cdot)$ maps large positive numbers to 0 and large negative numbers to 1, to be consistent with the bipolar encoding described in Chapter 2.) Each training step used a minibatch of size 120, 80, or 40, and used the RMSProp update rule [50] with learning rates of 0.001, 0.0003, and 0.003 to compute parameter updates, for the codes with n=63 or (n, k)=(127,106), (n, k) = (127,99), and (n, k) = (127,64), respectively. They found

that using a “multiloss” approach, in which the cross-entropy loss is computed for the soft output of each iteration rather than the last iteration alone, improved results. This improvement may be due to the gradients encountering fewer intervening multiplications along the path between the loss function and the parameters.

3.2.2 Neural mRRD decoding

While the neural BP decoder improves greatly upon the simple BP decoder for HDPC codes, at a small cost in arithmetic complexity, there is still a gap between its performance and the performance of the optimal decoder. One decoder which does achieve close to optimal performance is the improved random redundant decoding (mRRD) algorithm [51]. In mRRD decoding, a set of parallel BP decoders with random permutations attempt to decode a received word in parallel, stopping when their soft output results in a valid codeword (Fig. 3.3). Of the parallel decoders' outputs, the codeword with the maximum likelihood with respect to the received word is selected. Running multiple BP decoders in parallel naturally results in an increase in complexity, but the mRRD decoder achieves close to ML performance as a result of the modification.

Nachmani *et al.* suggested modifying mRRD to yield “neural mRRD” in [4]. Neural mRRD improves upon mRRD by replacing the BP blocks with neural BP blocks. Fig. 3.4 shows the improvement in BER due to this modification when 1, 3, and 5 parallel BP or neural BP decoders are used in mRRD. It can been seen that the performance is improved for each case, and indeed the neural decoder with 5 parallel BP blocks achieves close to ML performance for this code. It is also possible to trade performance for complexity by running the neural mRRD for a smaller number of maximum iterations, thus yielding the same results as normal mRRD but using fewer BP iterations. As many other HDPC decoders also use BP as a building block ([43], [52], [53], [54]), it is very likely that neural BP would also improve their performance, although this has not yet been explored.

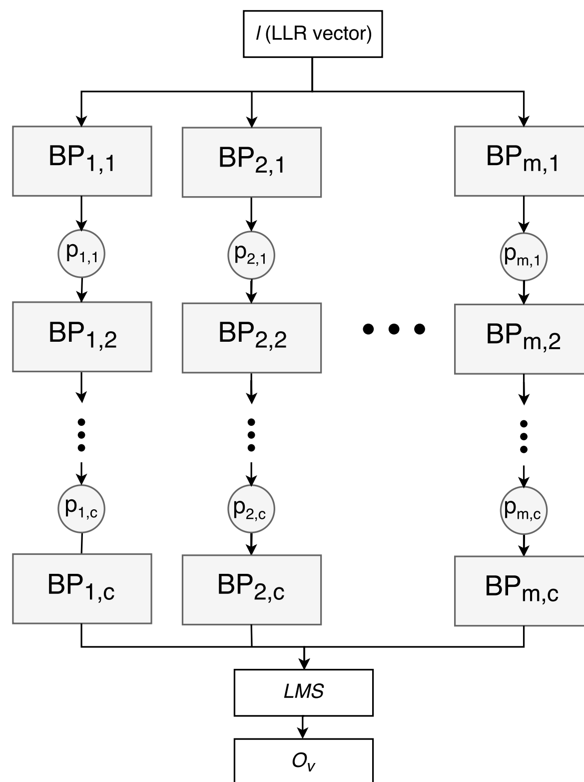


Figure 3.3: The mRRD decoder. (Figure taken from [4]).

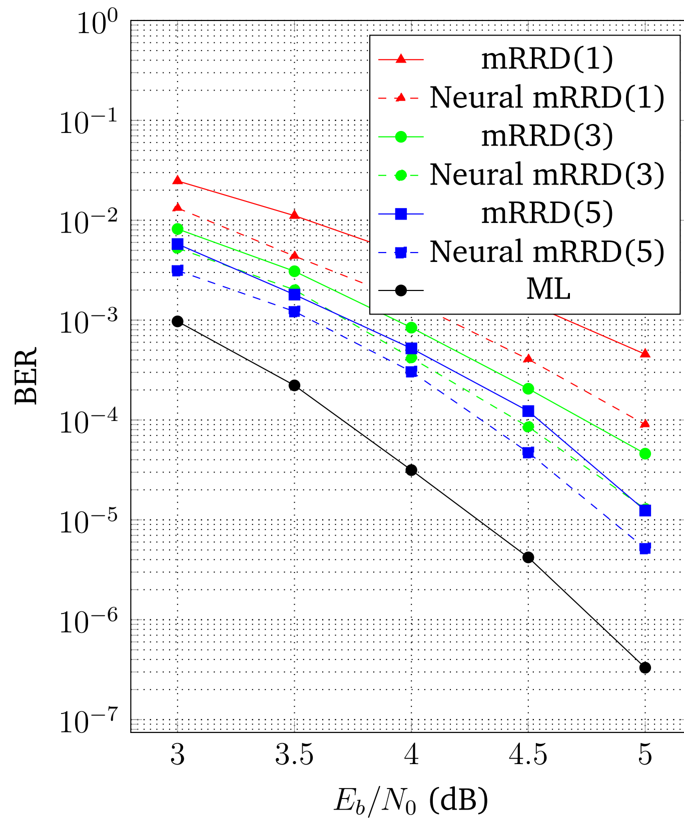


Figure 3.4: Neural mRRD BER results for BCH(63,36) code. (Figure adapted from [4].)

Chapter 4

Neural Min-Sum Decoding

The neural BP algorithm, combined with mRRD decoding, can achieve close to optimal performance for short HDPC codes, at low computational complexity. However, there are obstacles to using neural BP in practice: the algorithm requires numerous real-valued multiplications and storage for parameters. In this chapter, we suggest some modifications inspired by min-sum decoding algorithms which reduce the implementation complexity of both learning and inference for neural BP and analyze the performance and behaviour of these modifications.

4.1 Modifications to neural BP

4.1.1 Min-sum check nodes

The first modification is for the check nodes to use the so-called "min-sum" approximation [55]. In min-sum decoding, Equations 2.8 and 2.10 are used as before, and Equation 2.9 is replaced with

$$\mu^{t}_{c \to v} = \min_{v' \in \mathcal{M}(c) \setminus v} (|\mu^{t}_{v' \to c}|) \prod_{v' \in \mathcal{M}(c) \setminus v} \operatorname{sign}(\mu^{t}_{v' \to c}). \tag{4.1}$$

This approximation significantly simplifies the hardware required for the check node computation because it does not require any real-valued multiplications or hyperbolic functions. However, the min-sum approximation can, for certain codes, cause a large degra-

dation in bit error rate. This is in part because the approximation "overshoots" the values which the BP decoder would have computed. Since the message magnitudes can be interpreted as reliabilities or "confidence" metrics, messages with overly large magnitudes may seem to be more reliable than they actually are, causing the decoder to make mistakes when updating its beliefs.

A simple way to correct for the min-sum approximation is to shrink the magnitude of a check-to-variable message before sending it. This is implemented in normalized min-sum (NMS) decoding by multiplying the message by a small weight $w \in [0, 1]$ [56]. In NMS decoding, the check-to-variable message is thus computed using

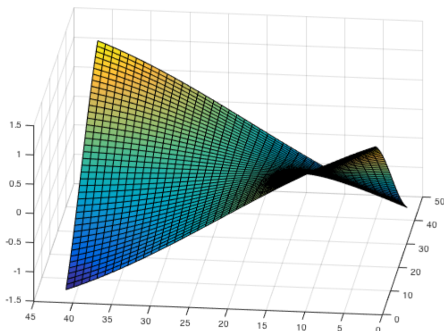
$$\mu_{c\to v}^t = w \left(\min_{v' \in \mathcal{M}(c) \setminus v} (|\mu_{v' \to c}^t|) \prod_{v' \in \mathcal{M}(c) \setminus v} \operatorname{sign}(\mu_{v' \to c}^t) \right). \tag{4.2}$$

In the spirit of neural BP and other "deep unfolding" algorithms, we proposed in [2] to generalize the NMS decoder by assigning a learnable weight to each edge for each iteration, rather than using a single global value for w. We call this approach neural normalized min-sum (NNMS) decoding. In NNMS decoding, the check node computation becomes:

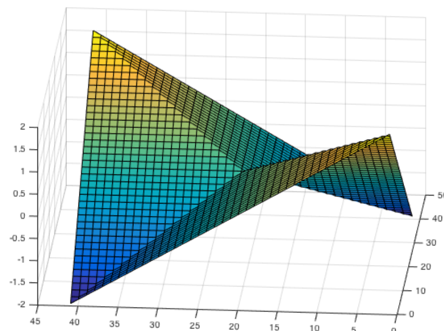
$$\mu_{c\to v}^t = w_{c\to v}^t \left(\min_{v' \in \mathcal{M}(c) \setminus v} (|\mu_{v' \to c}^t|) \prod_{v' \in \mathcal{M}(c) \setminus v} \operatorname{sign}(\mu_{v' \to c}^t) \right), \tag{4.3}$$

where $w_{c\to v}^t$ is the learnable weight for the message from check node c to variable node v during iteration t.

Due to the presence of the kinked $\min$ and abs functions, the function computed by a check node implementing Equation 4.3 is not differentiable with respect to the input messages at certain points (note the kinks in Fig. 4.1), which impedes the use of gradient descent. As discussed in Section 2.2.2, it is possible to work around this issue by using subgradients during backpropagation. While $\min$ and abs have subgradients defined everywhere, the sign function does not; the derivative of $\operatorname{sign}(x)$ is undefined at $x = 0$, and is equal to 0 everywhere else. Our approach has been to treat sign as though its derivative were simply equal to 0 everywhere, including at $x = 0$, thus routing the gradient entirely through $\min$ and abs. We have not encountered any difficulties by treating sign in this



(a) BP check node.



(b) Min-sum check node.

Figure 4.1: Plots of the output of a degree-3 check node.

way. With these modifications, the NNMS decoder can be trained using gradient-based optimization.

NNMS requires many fewer multiplications than neural BP due to the min-sum check nodes. NNMS also uses many fewer weights than the original version of neural BP presented in [10], but we have found that constraining neural BP to using the same number of parameters as NNMS does not significantly impact its performance, so this is not an essential difference. Another nice property of NNMS is that, like NMS, the algorithm does not suffer when the channel noise variance is is estimated imperfectly, since the computation is scale-invariant. By scale-invariant, we mean the following: Let a be a positive scalar, x be a vector in $\mathbb{R}^n$, and f_{NNMS} be the function computed by an NNMS decoder (that is, the soft output of the T'th iteration). Then $f_{\text{NNMS}}(ax) = af_{\text{NNMS}}(x)$. Therefore, if the received values are scaled by $a \neq \frac{2}{\sigma^2}$ (such as $a = 1$), the soft output of the NNMS decoder will simply be a scaled version of what the soft output would have been if the inputs had been properly scaled by $\frac{2}{\sigma^2}$, and the hard decision will produce the same bit estimates. This is not true of the neural BP decoder. Finally, an NNMS decoder may be easier to optimize than a neural BP decoder because, as a piecewise linear function, it is "almost linear", and linear functions are easy to optimize (a hypothesis used to justify the success of ReLU neural networks [57]).

4.1.2 Additive correction

NNMS is more hardware-friendly than neural BP but still requires multiplications. It would be preferable not to use any multiplications at all in the decoder. Practical NMS decoders often constrain the value of w to a power of two so that the multiplication can be implemented using a simple shift operation. However, better performance can be achieved using weights which are not powers of two. Another corrected min-sum algorithm which avoids the use of multiplications yet allows fine control over the possible parameter values is the offset min-sum (OMS) algorithm [58]. In OMS, the check-to-variable message is computed using:

$$\mu_{c\to v}^{t} = \max\left(\min_{v'\in\mathcal{M}(c)\backslash v}(|\mu_{v'\to c}^{t}|) - \beta, 0\right)\prod_{v'\in\mathcal{M}(c)\backslash v}\text{sign}(\mu_{v'\to c}^{t}), \tag{4.4}$$

where β is an offset that reduces the message magnitude. The inclusion of $\max(\ldots, 0)$ in the message computation prevents the offset subtraction from flipping the sign of the outgoing message, in the case when the minimum input message magnitude is very small.

In [1], we proposed generalizing OMS decoding to yield neural offset min-sum (NOMS) decoding. Like the other neural decoders described above, NOMS assigns a separate, learnable parameter to each edge of the graph, except the parameters are additive offsets rather than multiplicative weights. In NOMS, the check-to-variable message is computed using:

$$\mu_{c\to v}^{t} = \max\left(\min_{v'\in\mathcal{M}(c)\backslash v}(|\mu_{v'\to c}^{t}|) - \beta_{c\to v}^{t}, 0\right)\prod_{v'\in\mathcal{M}(c)\backslash v}\text{sign}(\mu_{v'\to c}^{t}), \tag{4.5}$$

where $\beta_{c\to v}^{t}$ is the learnable offset for the message from check node c to variable node v during iteration t.

Unlike NNMS and neural BP, NOMS requires no multiplications whatsoever during inference. The only overhead compared to OMS is additional read-only memory for parameter storage and indexing logic to look up the parameter used to compute a given message.

4.1.3 Weight sharing

A shortcoming of the neural decoding algorithms presented above is that they are feedforward networks and require a separate set of parameters for each iteration. As the number of iterations and edges increases, so does the amount of storage required for parameters, which reduces implementation efficiency. In [1], we suggested that the number of parameters could be reduced by constraining multiple edges to using the same parameter, a technique known as "weight sharing" in the deep learning literature. (We also noted that, in the extreme case when all edges are constrained to using the same parameter, NOMS reverts to OMS and NNMS reverts to NMS, thus yielding a new method for optimizing the parameters for those algorithms, as well as algorithms which use a separate parameter value for each iteration, as suggested in [58]). Weight sharing was later implemented by Nachmani *et al.* in [4], who showed that the same set of edge weights could be used for each iteration. This constraint effectively transforms the decoder into a kind of recurrent neural network (RNN), as shown in Fig. 4.2, and greatly reduces the number of parameters required.

4.1.4 Relaxation

Successive relaxation (or simply "relaxation") is another technique commonly used for improving the performance of iterative algorithms. The application of relaxation to belief propagation was first proposed in [59] by Murphy *et al.*, who refer to it as "momentum". Murphy *et al.* showed that relaxation can improve inference in certain loopy probabilistic graphical models, but they did not apply the technique to decoding. Hemati *et al.* showed in [60] that relaxation may arise naturally in analog decoders, in which the parasitic capacitances and resistances of the decoder circuit have a low-pass filter effect on the messages, and that relaxation improves the performance of both analog and digital decoders for LDPC codes.

In a relaxed system, the update μ^t in iteration t is a combination of the update μ^{t-1} in the previous iteration and some other vector ν^t, that is:

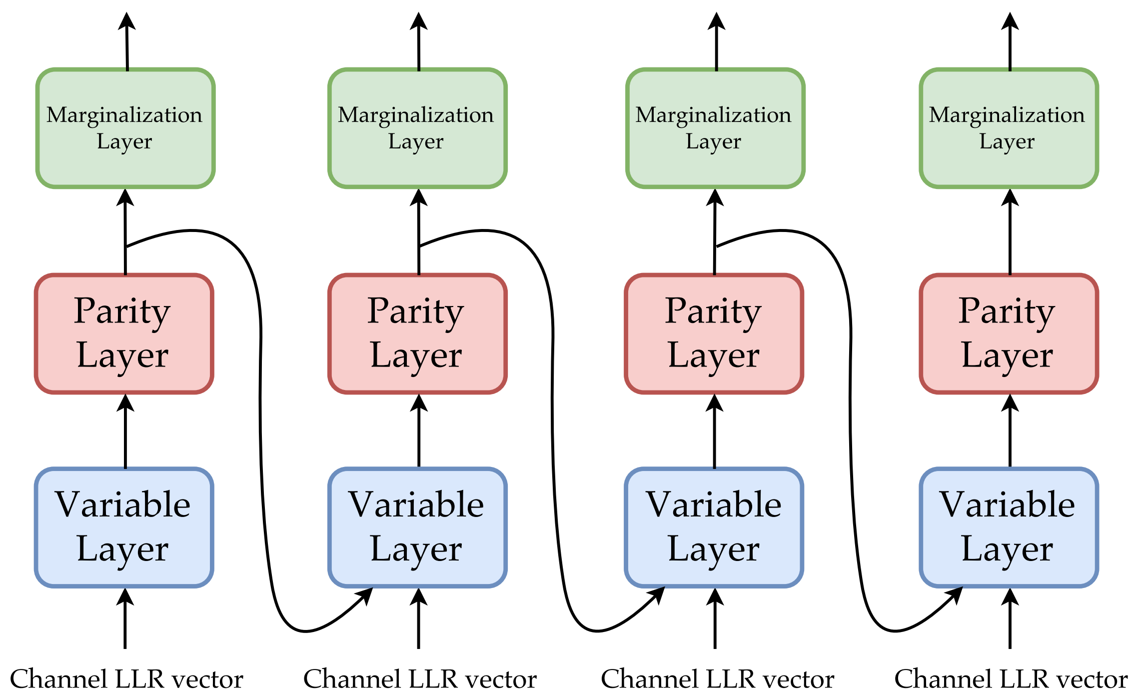


Figure 4.2: Four iterations of a decoder with weight sharing. (Figure adapted from [4].)

$$\mu^t = \gamma\mu^{t-1} + (1-\gamma)\nu^t, \tag{4.6}$$

where γ is the relaxation factor. When relaxation is applied to iterative decoding, ν^t is simply the message which would have been computed by a non-relaxed decoder, and μ^t is the relaxed message.

Analyzing the behavior of relaxed decoders is difficult because the edges of the Tanner graph have memory in such decoders. While there has been work in modeling relaxed decoders analytically [61], other researchers ([60], [62]) have chosen the relaxation factor using simple trial-and-error or by choosing a value which is a power of two. We proposed instead in [2] to allow the decoder to learn the relaxation factor using gradient descent, in conjunction with any other learnable decoder parameters. (It is also possible to use a separate learnable relaxation parameter for each edge of the Tanner graph, but we have found that this does not yield any improvement over using a single parameter.) The advantage of this approach, as we will show, is that a good value for γ can be computed much more

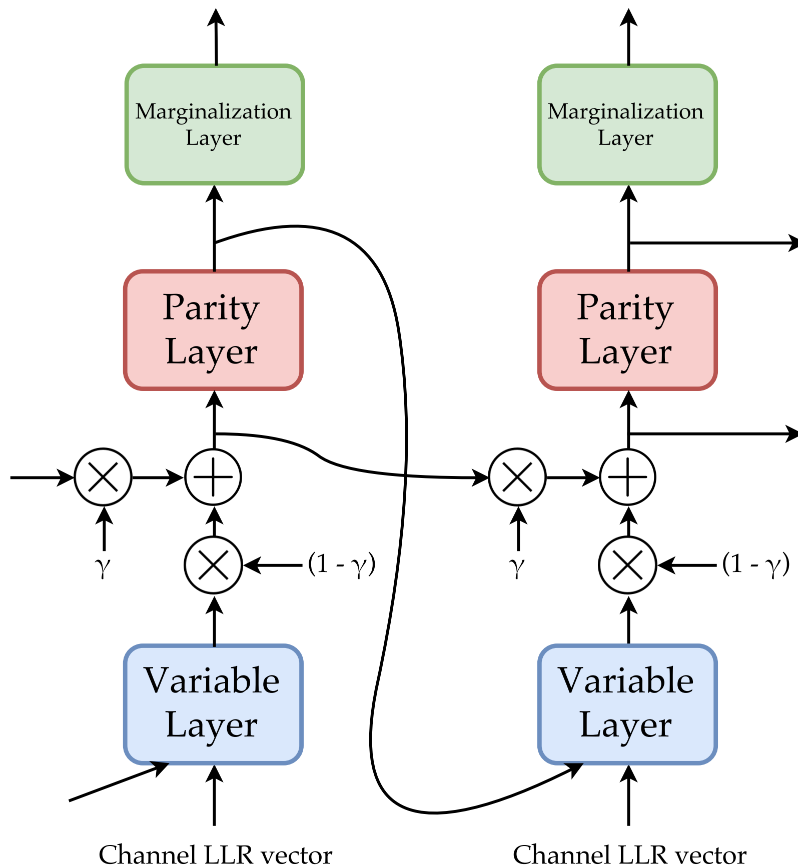


Figure 4.3: Two iterations of a relaxed decoder.

efficiently than using trial-and-error, even when only a single relaxation factor is used.

Fig. 4.3 shows one possible version of a relaxed decoder, in which the variable-to-check messages are relaxed. In this figure, $+$ indicates elementwise addition, $\times$ indicates elementwise multiplication, and $\gamma = g(\gamma')$, where γ' is the unconstrained relaxation parameter(s) in the range $(-\infty, +\infty)$, and g is the sigmoid function. The sigmoid function differentiably squashes γ' into the range (0,1), where $\gamma = 0$ corresponds to a non-relaxed decoder, and $\gamma \to 1$ corresponds to a completely relaxed decoder. (During inference, γ can be precomputed, so that $g(\gamma')$ need not be computed redundantly.)

4.1.5 Alternative loss functions

An aspect of training which may be improved is the loss function itself. The cross-entropy loss is the correct loss to use, in the sense that truly minimizing it would make the decoder equivalent to the optimal bitwise MAP decoder. However, there is no guarantee that the decoder has the capability to learn weights which minimize the cross-entropy loss.

Additionally, the cross-entropy loss is not hardware-friendly. Consider the v'th entry of the gradient of the cross-entropy with respect to the decoder's soft output:

$$\begin{aligned}\frac{\partial}{\partial s_v}\ell_{\text{cross-entropy}}(z,s) &= \frac{\partial}{\partial s_v}\left(-\frac{1}{n}\sum_{v'=1}^{n} z_{v'}\ \log\,\sigma(-s_{v'}^T) + (1-z_{v'})\ \log\,(1-\sigma(-s_{v'}^T))\right) \\ &= \frac{1}{n}\left(z_v - \sigma(-s_v^T)\right).\end{aligned} \tag{4.7}$$

Computing the final expression in 4.7 requires computing $\sigma(\cdot)$, which is slow and expensive. It may be sensible to consider using a different classification loss, such as the hinge loss. The hinge loss is defined as:

$$\ell_{\text{hinge}}(x,s) = \frac{1}{n}\sum_{v=1}^{n}\max(0, 1 - x_v s_v^T), \tag{4.8}$$

where the 1 represents a safety margin that ensures that outputs are "far" from being incorrect. The (sub)gradient of the hinge loss is

$$\frac{\partial}{\partial s_v}\ell_{\text{hinge}}(x,s) = -\frac{1}{n}\mathbb{I}(1 > x_v s_v^T)\cdot x_v. \tag{4.9}$$

This gradient is slightly easier to compute than that of the cross-entropy loss, since it does not require a sigmoid. The difference is not crucial for GPU-based training, but one could imagine a training setup using specialized hardware— such as a decoder that learns online— in which the hinge loss would be a better choice.

4.2 Experimental results

We now provide several experimental results to validate the techniques proposed above.

We trained several neural decoders for two BCH codes and compared their performance to the non-neural BP and min-sum decoders. We used the Adam optimizer [17] with multiloss and a learning rate of 0.1 for the NOMS decoders and 0.01 for the NNMS decoders, and reused the training hyperparameters for the neural BP decoders described in the previous chapter. The parity-check matrices for these codes were taken from [49].

Fig. 4.4 and Fig. 4.5 show the performance of the various decoders for the BCH (63,36) code and BCH (63,45) code, respectively. Three main conclusions may be drawn from these plots.

1. The min-sum approximation, while degraded compared to BP without additional correction, has little impact on the performance of the neural decoders– in other words, the neural BP decoder and NNMS decoders have roughly equivalent performance.

2. Weight sharing slightly degrades decoder performance.

3. Additive offsets, while yielding performance improvements on the order of 1 dB compared to the non-neural decoder, are not quite as effective in improving performance as multiplicative weights.

We also demonstrate the scale-invariance of NNMS mentioned above. We trained an NNMS decoder and a neural BP decoder, and at test time fed the raw received values into the decoders, rather than scaling them by $\frac{2}{\sigma^2}$. Fig. 4.6 shows the resulting performance gap between the decoders. The performance of neural BP is sharply degraded, whereas NNMS is unaffected.

To measure the performance of relaxed decoders, we trained a relaxed decoder with a single relaxation factor and no additional learnable parameters, a constrained version of that decoder, and a relaxed NOMS decoder for the BCH (63,45) code. Fig. 4.7 shows that these decoders achieve excellent performance, on par with the other decoders described here. In fact, the relaxed NOMS decoder achieves the best performance among all the decoders for this code, showing that relaxation can be effectively combined with other trainable decoder building blocks.

Fig. 4.8 shows the evolution of the value of the relaxation factor γ as training proceeds. The value of γ converges to around 0.863. Note, however, that 0.863 is close to 0.875— if

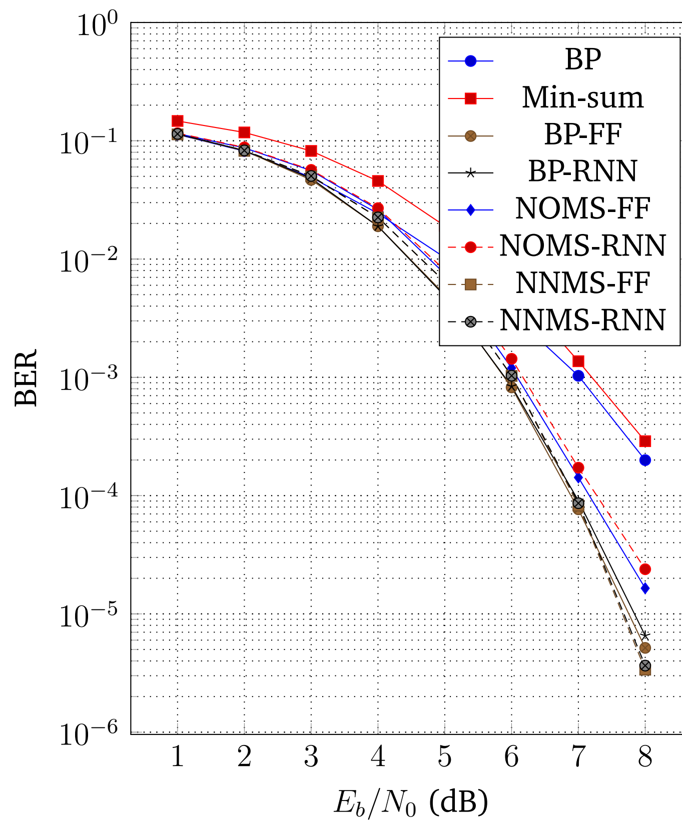


Figure 4.4: Performance comparison of BP and min-sum decoders for BCH (63,36) code.

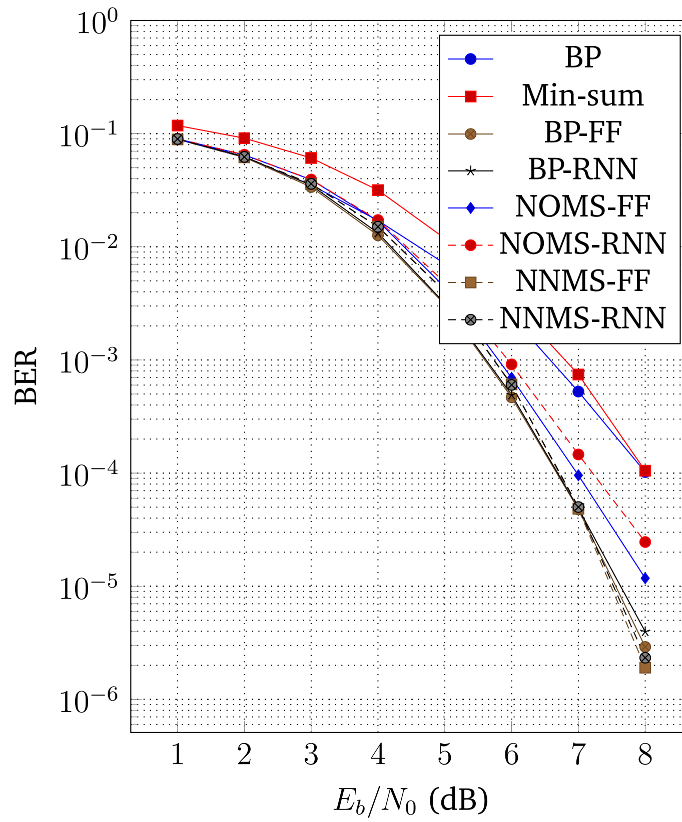


Figure 4.5: Performance comparison of BP and min-sum decoders for BCH (63,45) code.

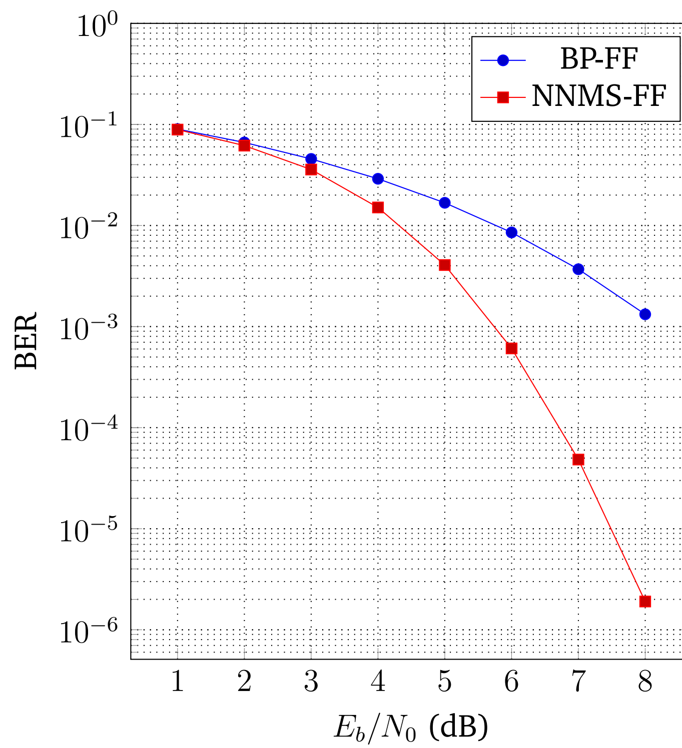


Figure 4.6: Performance comparison of neural BP and NNMS decoders for BCH (63,45) code when inputs are not properly scaled to LLR format.

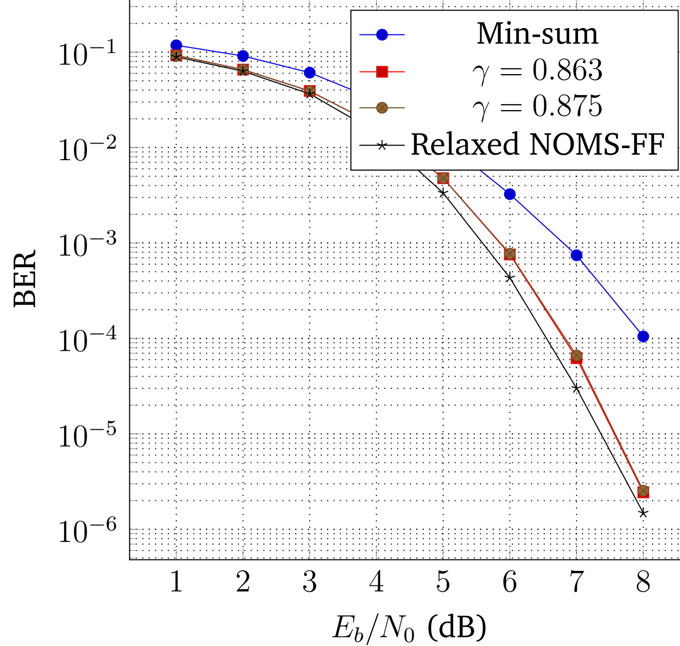


Figure 4.7: Performance of a min-sum decoder (i.e., $\gamma = 0$), a relaxed min-sum decoder which has learned $\gamma = 0.863$, a relaxed min-sum decoder which is constrained to using $\gamma = 0.875$, and a relaxed NOMS decoder for the BCH (63,45) code.

we the learned value of γ is replaced with 0.875, the relaxation equation can be rewritten as follows:

$$\mu^t = \mu^{t-1} + 0.125(\nu^t - \mu^{t-1}), \tag{4.10}$$

which can be implemented very efficiently in hardware, since 0.125 is a power of two. As Fig. 4.7 shows, constraining the learned relaxation factor in this case causes a nearly imperceptible increase in BER.

The trajectory of γ reveals another useful property of gradient-based decoder learning. The relaxation factor attains its final value after approximately 400 minibatches have been processed in training. In a non-optimized implementation of backpropagation (if the algorithm does not take advantage of sparsity, as we will describe later), the number of operations used during backpropagation is approximately the same as the number of operations used during the forward pass. Thus, processing a received vector in training is roughly equivalent to decoding two received vectors, in terms of the number of operations required. Hence, processing 400 minibatches is roughly equivalent to simulating

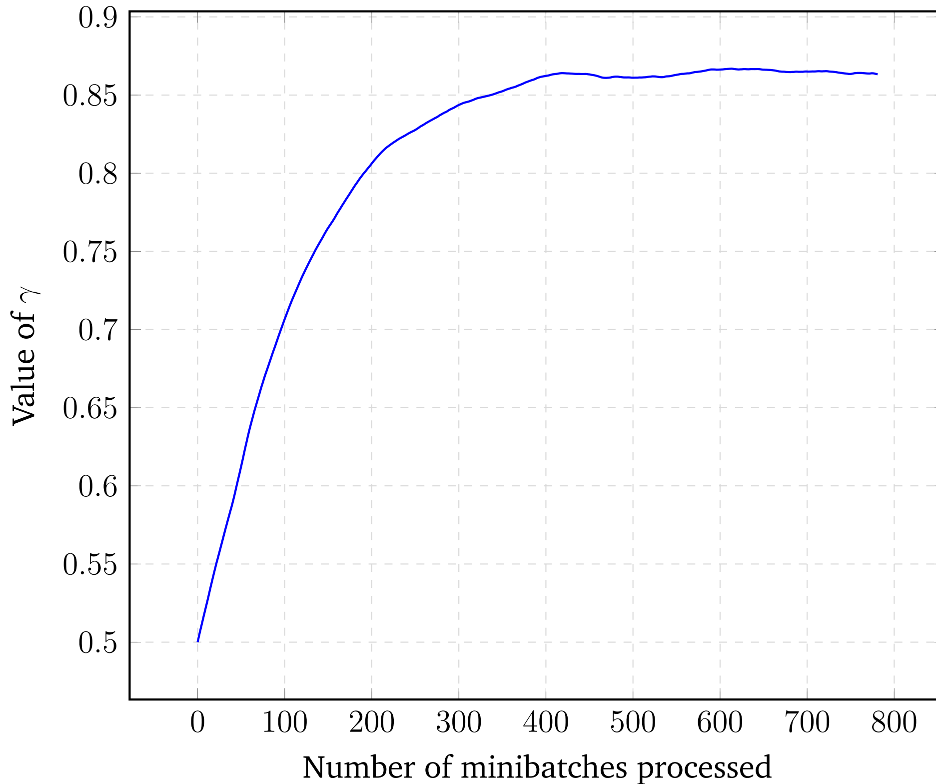


Figure 4.8: Evolution of γ as training proceeds.

$400 \times 120 \times 2 = 96{,}000$ frames. In contrast, in one simulation, running the decoder until 100 frame errors had occurred at an SNR of 8 dB required processing $1{,}064{,}040$ frames. In order to test more values for γ to choose a good value in the trial-and-error approach, even more frames would need to be processed. For this reason, it would seem that gradient-based learning is a more efficient way to optimize a decoder, even when the number of parameters is small.

Finally, we compared the performance of decoders trained using the cross-entropy loss with those trained using the hinge loss. Fig. 4.9 shows the performance comparison for a recurrent NNMS decoder for the BCH (63,45) code. The performance is virtually the same; the hinge loss could, therefore, potentially be used to speed up training without affecting the quality of results.

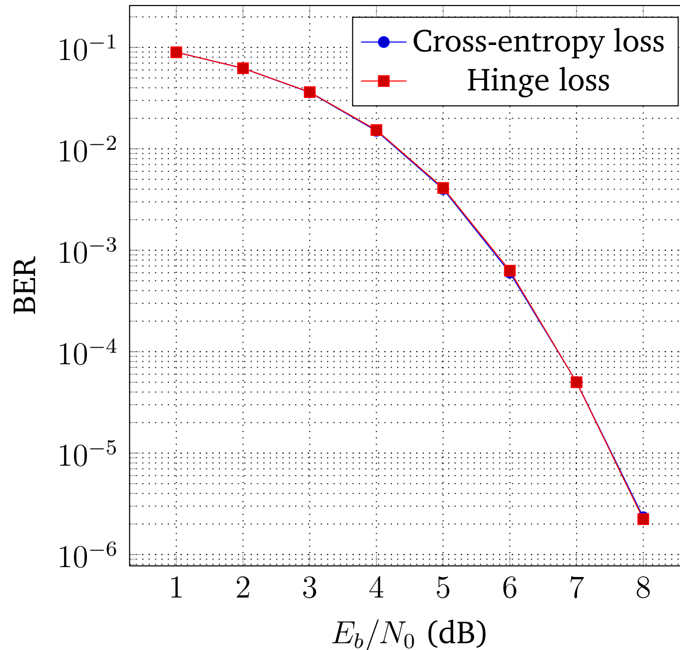


Figure 4.9: Comparison of decoder trained using cross-entropy loss and decoder trained using hinge loss.

4.2.1 Does learning help for other codes?

The results presented above are only for short, high-density codes. The reader may wonder whether learnable parameters can further improve the performance of decoders for codes for which iterative decoding already performs well, such as long, well-designed LDPC codes and turbo codes. We have also run an experiment with a (200,100) LDPC code, and found that learning does improve performance slightly (see Fig. 5.4 in Chapter 5). Unfortunately, we were unable to run simulations for longer codes; constructing the computational graph for these decoders in our rather inefficient simulation framework used too much memory, even on a server with 48 GB of RAM. At the end of this chapter, we discuss some optimizations that could be applied to our simulation framework to allow it to handle larger codes.

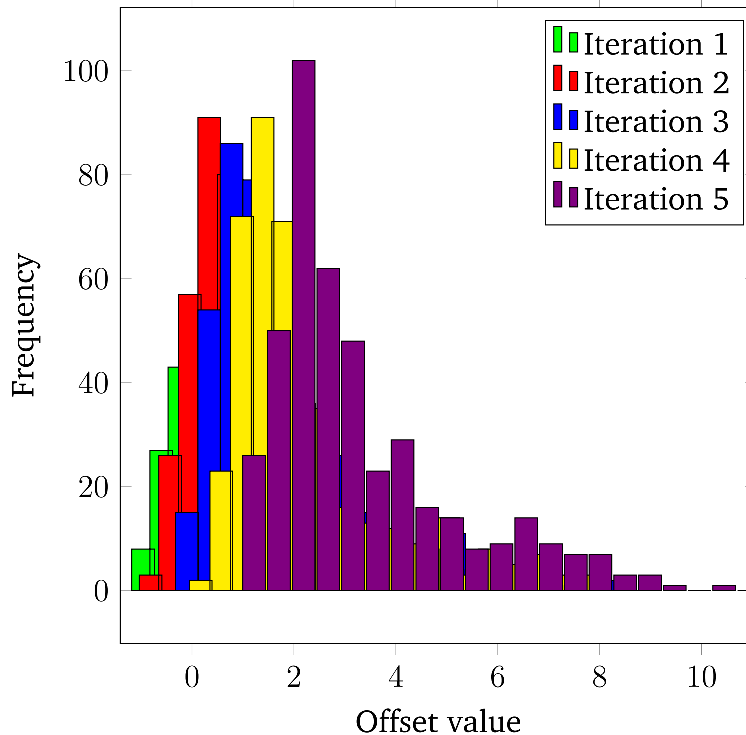


Figure 4.10: Offset histogram for BCH (63,45) decoder after 3,000 minibatches.

4.3 Discussion

4.3.1 Analyzing the learned behaviour

The complex, nonlinear nature of neural networks makes them difficult to analyze, which can be frustrating to those who would like to understand how exactly deep networks are able to generalize so well to new unseen test data. In lieu of theoretical results, it is sometimes possible to visualize what a network has learned, as in [63] or the signal detector example given in Chapter 2. While we do not yet have a precise understanding for where the improvement in neural decoders comes from, visualizing the parameter distribution may provide a clue [1].

Fig. 4.10 shows the offset distribution in a feedforward NOMS decoder for the BCH (63, 45) code after 3,000 minibatches have been processed during training. For each iteration, the offsets have a bimodal distribution: one mode centered at a lower value, consistent with the values typically chosen for β in the OMS literature, and a second, much higher

mode. It may be that the offsets clustered around the lower mode serve mostly to correct for the min-sum approximation, and the offsets clustered around the higher mode serve more to reduce the effect of cycles. A high value for an offset means that a larger message is required to activate the output of a NOMS check node, so perhaps the higher-valued offsets serve to zero out unhelpful messages. More work is required to understand the operation of neural decoders.

4.3.2 Backpropagation and sparsity of gradients

It is often helpful when designing a new deep learning architecture to represent the behaviour of activations and gradients graphically, as in [64]. Here, we present a graphical analysis of the behaviour of neural min-sum decoders during backpropagation and reveal some interesting properties.

Fig. 4.11 shows a three-input variable node. From the multivariable chain rule, we know that gradients sum at a node, and therefore the gradient with respect to one of the input messages is equal to the sum of the gradients of the output messages. Thus, the variable node operation during the backward pass of training is identical to the variable node operation during the forward pass, except that the inputs and outputs are gradients with respect to the messages rather than the messages themselves.

Examples of decoder "synapses" (sites where a parameter interacts with a message) are shown in Fig. 4.12. The gradients in the case of non-shared parameters are straightforward. If a synapse parameter is shared (e.g. between two iterations), each instance of the parameter can be considered a separate "dummy" parameter, and the gradients of these dummy parameters are summed to yield the actual gradient for the parameter, a procedure known as "backpropagation through time" [65] (another application of the multivariable chain rule).

Whereas the marginalization in a non-neural decoder "unthinkingly" adds up the messages to compute the soft output, the NNMS decoder learns to weight the incoming messages before adding them up. Since the sigmoid function is used during training with the cross-entropy loss, this "smart" marginalization can be interpreted as the action of a

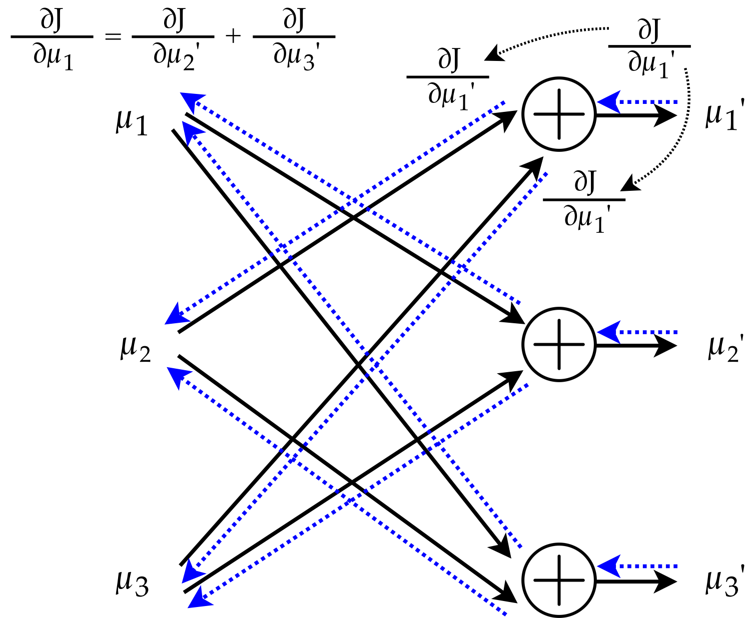


Figure 4.11: Activations and gradients for a three-input variable node.

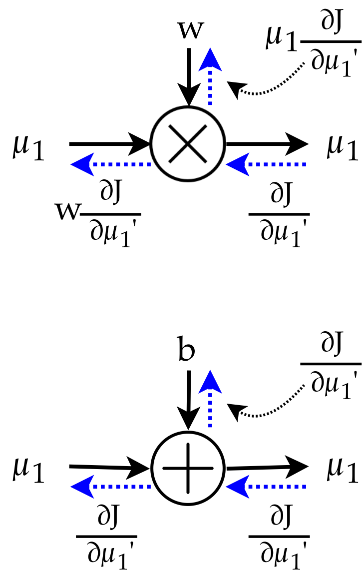


Figure 4.12: Activations and gradients for multiplicative and additive synapses.

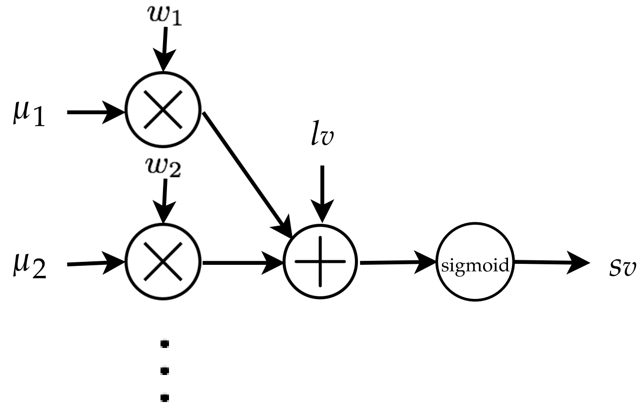


Figure 4.13: Sigmoid neuron in a variable node.

sigmoid neuron, in which the bias input is the received value, l_v (Fig. 4.13).

Fig. 4.14 shows the behaviour of a three-input check node. Notice that many of the forward propagating signals do not have a corresponding backward propagating signal. There are two reasons for this. First, as we have been treating the derivative of sign as equal to 0 everywhere, the paths from sign to the output can be pruned from the backpropagation computation. Second, note that the derivative of the $\min$ function is:

$$\frac{\partial}{\partial x_i} \min_j x_j = \begin{cases} 1, & \text{if } x_i = \min_j x_j \\ 0, & \text{otherwise.} \end{cases}$$

In other words, for the input which was the minimum, the derivative is equal to 1, and for every other input, the derivative is equal to 0. Due to the presence of the $\min$ function, all but two of the inputs (the first minimum and the second minimum) will have a gradient equal to 0 for a given received vector. For parity-check matrices such as the one used for the BCH (63,45) code, which has 24 inputs for each check node, the gradients will be quite sparse as a result. It may be possible to train neural min-sum decoders using fewer arithmetic operations and less memory by taking advantage of the sparsity in the gradients, simply by keeping track of the first minima and second minima during the forward pass, and setting the gradient equal to zero for every other input to the check node.

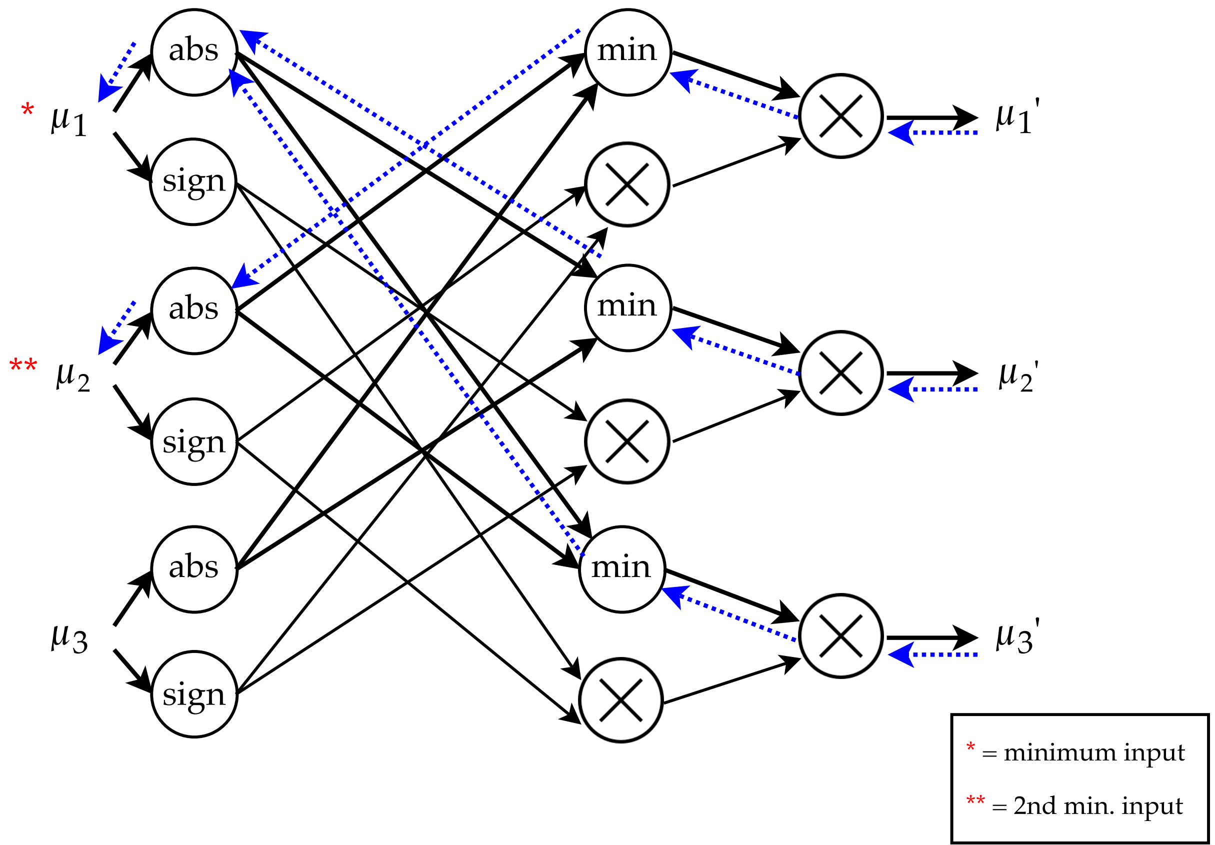


Figure 4.14: Activations and (non-zero) gradients for a three-input min-sum check node after pruning. Note that there are very few non-zero gradients.

Chapter 5

Learning from the Syndrome

All of the approaches described thus far have a common flaw: they operate by applying a loss function, such as the cross-entropy loss or hinge loss, to each of the neural network outputs *separately* to minimize the distance between the neural network's predictions and the actual transmitted codewords. Implicitly, this treats the network as though it were solving a set of unrelated binary classification problems (classifying each bit of the received word as a 0 or a 1), when in fact the task of the network is to predict a single structured object (a codeword) with certain known constraints (the parity). In this chapter, we propose to instead train the decoder using structured output learning, in which the outputs are penalized not only for their distance to the transmitted codeword but also for having invalid structure, measured using a loss function based on the syndrome. We show that using this loss function yields decoders with lower frame error rate than when the classification loss is used.

5.1 Error correction as structured prediction

A structured prediction problem is a problem in which several items must be predicted, and there are some known constraints on the relationship between the items. For instance, in part-of-speech tagging, the predictor must guess a part-of-speech (e.g., noun, verb, adjective) for each word in a sentence, and we know that language has certain constraints (e.g., in English, an adjective typically does not come after a noun). Error correction

is another perfect example of a structured prediction problem: given a noisy signal, the decoder must guess a value for each codeword bit, and the guesses must satisfy the parity checks.

"Structured output learning" refers to machine learning techniques that take the known structure into account while training a predictor, thus improving the final accuracy or speeding up the learning process. One approach to structured output learning is to use a differentiable loss function that encodes "how violated" the known constraints are and to incorporate this loss function into gradient-based learning.

5.2 The syndrome loss

Since error correction is a structured prediction problem, we would like to formulate a structured loss function for soft decoders. We will base this loss function on the syndrome, since the syndrome is a measure of whether code constraints are violated.

As described in Chapter 2, the syndrome is defined as Hz, but we can define it in a slightly different way that may be more illuminating for the current discussion. For a soft estimate s of the codeword, the c'th entry of the syndrome can be defined as

$$\text{synd}(s)_c = \prod_{v' \in \mathcal{M}(c)} \text{sign}(s_{v'}), \tag{5.1}$$

where $\mathcal{M}(c)$ is the set of codeword bits which belong to parity-check c. If one or more entries of synd(s) are equal to -1, then the hard decision is not a codeword. Thus, the training objective should be to minimize the average number of negative entries in the syndrome that the decoder causes.

Unfortunately, the raw syndrome is not well suited to be used in a loss function for gradient-based learning, since the derivative of $\text{sign}(\cdot)$ is 0 almost everywhere. Therefore, we introduce an alternative form of the syndrome. Let softsynd(s) be the "soft syndrome", the c'th entry of which is computed using

$$\text{softsynd}(s)_c = \min_{v' \in \mathcal{M}(c)} (|s_{v'}|) \prod_{v' \in \mathcal{M}(c)} \text{sign}(s_{v'}). \tag{5.2}$$

Note that the c'th entry of the soft syndrome is simply the min-sum check node computation applied to all bit estimates belonging to the c'th parity check, rather than a subset of those messages. Additionally, the soft syndrome has a nontrivial subgradient. The derivative of the c'th entry of the soft syndrome with respect to the v'th soft bit estimate is:

$$\begin{aligned}
\frac{\partial}{\partial s_v}\text{softsynd}(s)_c &= \frac{\partial}{\partial s_v}\left(\min_{v'\in\mathcal{M}(c)}(|s_{v'}|)\prod_{v'\in\mathcal{M}(c)}\text{sign}(s_{v'})\right) \\
&= \frac{\partial}{\partial s_v}\left(\min_{v'\in\mathcal{M}(c)}(|s_{v'}|)\right)\prod_{v'\in\mathcal{M}(c)}\text{sign}(s_{v'}) \\
&+ \min_{v'\in\mathcal{M}(c)}(|s_{v'}|)\frac{\partial}{\partial s_v}\left(\prod_{v'\in\mathcal{M}(c)}\text{sign}(s_{v'})\right) \\
&= \frac{\partial}{\partial s_v}\left(\min_{v'\in\mathcal{M}(c)}(|s_{v'}|)\right)\prod_{v'\in\mathcal{M}(c)}\text{sign}(s_{v'}) + 0 \\
&= \mathbb{I}(|s_v| = \min_{v'\in\mathcal{M}(c)}(|s_{v'}|) \wedge v \in \mathcal{M}(c)) \cdot \frac{\partial}{\partial s_v}(|s_v|) \cdot \prod_{v'\in\mathcal{M}(c)}\text{sign}(s_{v'}) \\
&= \mathbb{I}(|s_v| = \min_{v'\in\mathcal{M}(c)}(|s_{v'}|) \wedge v \in \mathcal{M}(c)) \cdot \text{sign}(s_v) \cdot \prod_{v'\in\mathcal{M}(c)}\text{sign}(s_{v'}) \\
&= \mathbb{I}(|s_v| = \min_{v'\in\mathcal{M}(c)}(|s_{v'}|) \wedge v \in \mathcal{M}(c)) \prod_{v'\in\mathcal{M}(c)\backslash v}\text{sign}(s_{v'}).
\end{aligned} \tag{5.3}$$

In words, the derivative is equal to the product of the signs of the other bit estimates if s_v has the minimum magnitude and bit v is involved in parity check c, and 0 otherwise. This is similar to the hinge loss in that the derivative is always one of $\{-1, 0, +1\}$.

The soft syndrome behaves similarly to the conventional hard syndrome, but has real-valued entries. For example, suppose that a transmitter in a communication system using the (7,4) Hamming code (see Fig. 2.1) sends the all-zeros codeword,

$$x = \{+1, +1, +1, +1, +1, +1, +1\}, \tag{5.4}$$

and the receiver observes the sequence

$$y = \{+1.67, +1.42, \ \mathbf{-0.03}, +1.03, +0.88, +1.98, +0.44\}, \tag{5.5}$$

which has one error. Whereas the hard syndrome for the received vector evaluates to

$$\text{synd}(y) = \{+1, \ \mathbf{-1}, \ \mathbf{-1}\}, \tag{5.6}$$

the soft syndrome evaluates to

$$\text{softsynd}(y) = \{+0.88,\ \textbf{-0.03},\ \textbf{-0.03}\}. \tag{5.7}$$

We can construct a loss function based on the soft syndrome by penalizing all the entries that are negative (or non-negative but of smaller magnitude than a safety margin of 1) as follows:

$$\ell_{\text{syndrome}}(s) = \frac{1}{n-k}\sum_{c=1}^{n-k} \max\left(1 - \text{softsynd}(s)_c, 0\right). \tag{5.8}$$

We propose to use the syndrome loss as part of a loss function in gradient-based learning. The complete loss function is a convex combination of the usual "binary classification" loss and the syndrome loss, and it can be written as:

$$\ell(z,s) = \lambda \cdot \ell_{\text{cross-entropy}}(z,s) + (1-\lambda)\cdot \ell_{\text{syndrome}}(s), \tag{5.9}$$

where $\lambda \in [0,1]$ controls the relative importance of the two terms.

5.3 Experimental results

We trained FNNMS decoders for a variety of codes with and without the syndrome loss. For each experiment, we used 10,000 minibatches with $\lambda = 0.5$, using the same settings as were used for the experiments in the previous chapter. Better results could probably be obtained by tuning the hyperparameters; however, we want to show here that even without careful tuning, the syndrome loss leads to improvement.

Figures 5.1, 5.2, 5.3, and 5.4 compare the FER of the decoders for a (16,8) LDPC code, a (63,45) BCH code, a (128, 64) polar code, and a (200, 100) LDPC code, respectively. Across all SNRs, for all codes, the FER of the decoders trained with the syndrome loss is lower than that of the decoders trained without it.

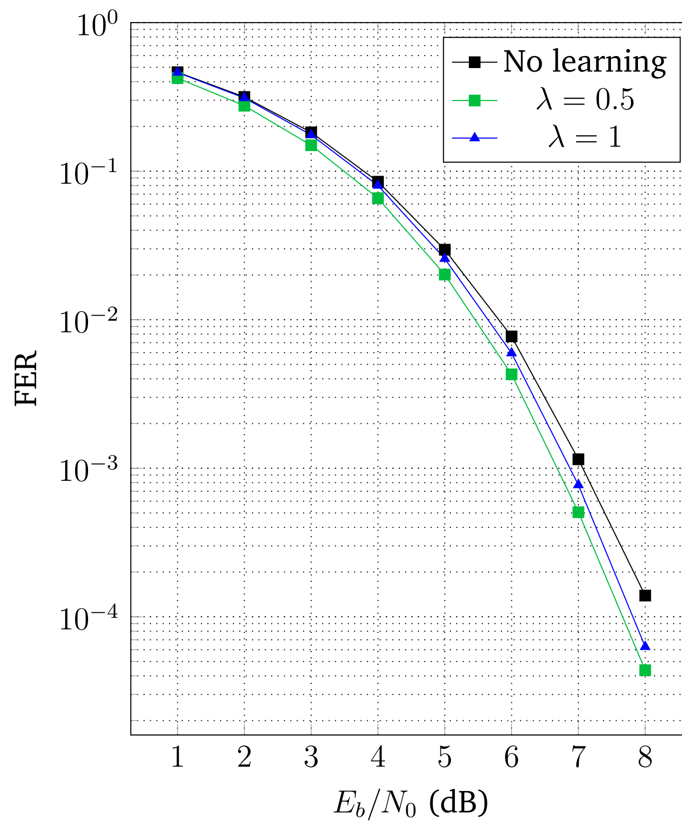


Figure 5.1: Comparison of FER for decoders for the (16,8) LDPC code trained with different values of λ.

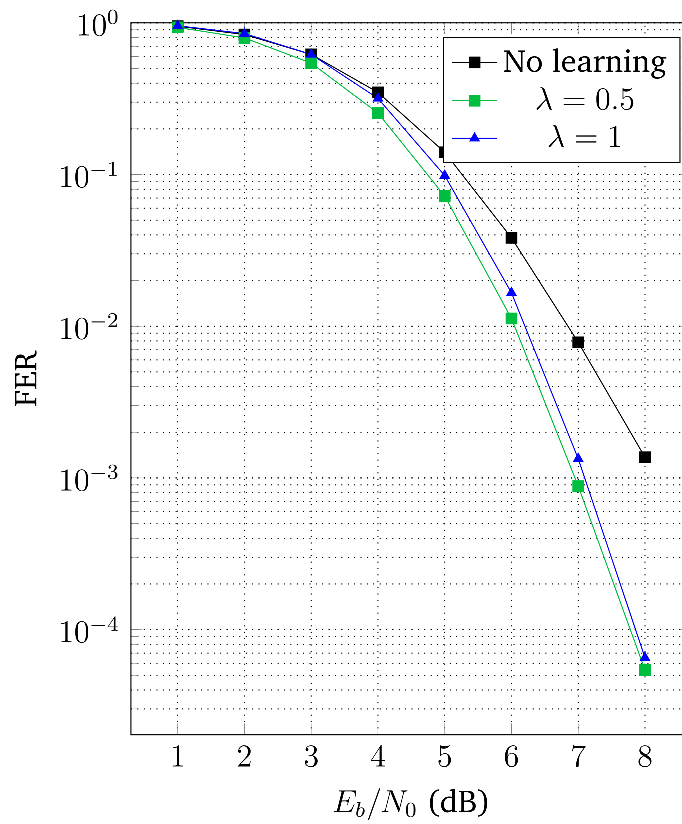


Figure 5.2: Comparison of FER for decoders for the BCH (63,45) code trained with different values of λ.

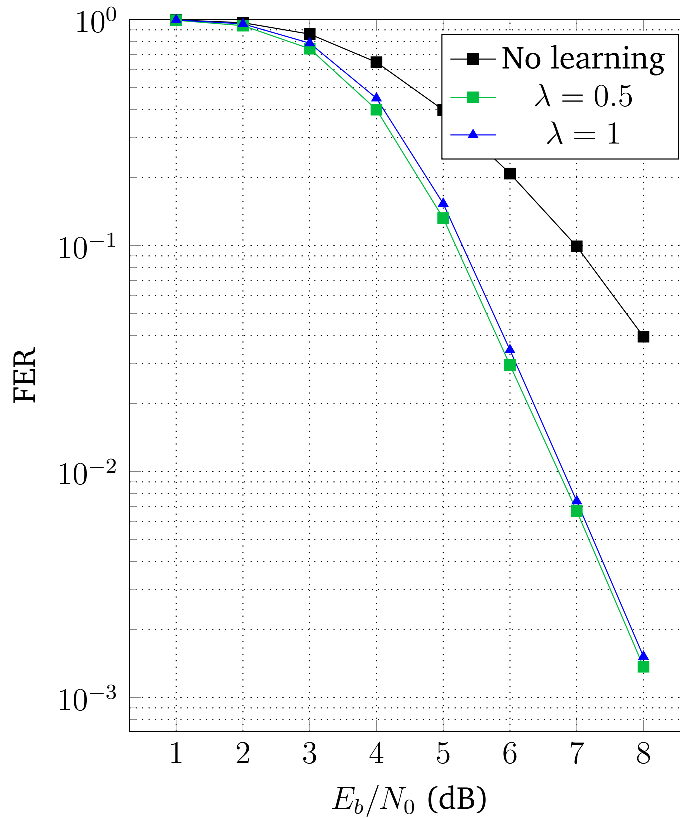


Figure 5.3: Comparison of FER for decoders for the (128,64) polar code trained with different values of λ.

5.4 Discussion

5.4.1 Related work

The syndrome loss is similar in spirit to the generalized syndrome weight in [66]. In this work, the authors used the generalized syndrome weight in a gradient descent-based decoding algorithm. This generalized syndrome weight was also used in [43] and [51]. In a similar vein, [67] and [68] approached the problem of blind detection and identification of LDPC codes using "syndrome LLRs" for each candidate code. However, none of these works considered using a soft syndrome to optimize the parameters of a decoder.

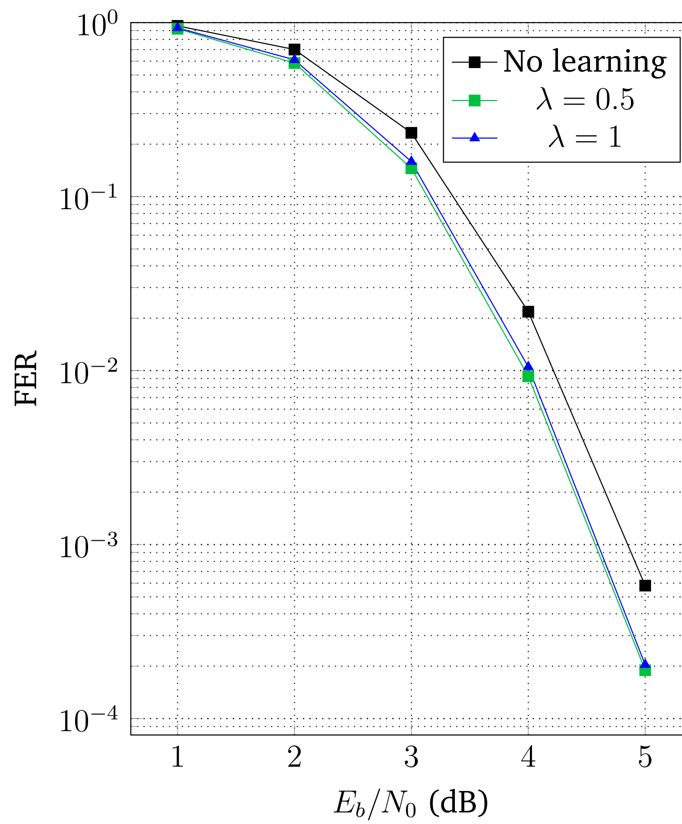


Figure 5.4: Comparison of FER for decoders for the (200,100) LDPC code trained with different values of λ.

5.4.2 Impact on BER and FER

Unlike the impact on FER, the impact of the syndrome loss on BER is inconsistent. At some SNRs BER improves, and at others it becomes worse. The difference is only very slight.

We hypothesize that the inconsistency occurs because the syndrome loss is a good proxy for FER but not necessarily for BER. The syndrome loss teaches the decoder to prefer to output a valid codeword, but in attempting to do so it may incorrectly estimate some bits.

5.4.3 Unsupervised learning

The syndrome loss is unsupervised in the sense that it does not depend on the transmitted codeword. To the best of our knowledge, ours is the first use of unsupervised learning to improve error-correcting decoders.

Is it possible to train a decoder in a fully unsupervised fashion (i.e., $\lambda = 0$)? It is, but with a caveat: the weights must be constrained to being non-negative. Otherwise, the decoder can "overfit" to the syndrome loss by finding a configuration of positive and negative weights which, when multiplied with the input, successfully pass the parity checks (in which case FER $= 1$ across all SNRs). It is simple to constrain the weights to being positive in our setup by passing them through the "softplus" nonlinearity ($w' = \log(1+e^w)$), just as we passed the relaxation parameters through the sigmoid function to constrain them to the range $[0, 1]$. The non-negativity constraint is not such an unreasonable constraint: we have found that the NNMS decoder weights always eventually end up positive when training with $\lambda > 0$, anyways. When we constrain the weights to be positive and set $\lambda = 0$, the resulting FER is similar to when $\lambda = 0.5$.

Although the capability to do unsupervised learning is theoretically interesting, the reader may wonder whether this capability is at all useful for the decoder, since it is possible to train the decoder offline using unlimited amounts of labelled training data. We believe that the unsupervised learning capability could potentially be used in an online learning setting in which the channel noise is non-Gaussian and the transmitted codewords are unknown. However, more work needs to be done to see if this actually works.

Chapter 6

Conclusion and Future Work

In this thesis, we reviewed— and improved upon— the state-of-the-art learning algorithms for error correction. We briefly conclude with some ideas that we did not fully explore during this research, which may be reasonable starting points for others interested in this topic.

There are different schedules for message passing in iterative decoding. In this thesis, we have considered only the flooding schedule, in which every variable node and check node sends a message during every iteration. It would be interesting to apply the learning techniques described here to other message passing schedules, such as the row-layered schedule. Indeed, it may even be possible to design better schedules using machine learning.

We also conjecture that learning can be used to improve the performance of LDPC decoders in the error-floor region. However, both the training of neural decoders for larger codes and Monte Carlo simulation in the high SNR region are computationally intensive. Implementing inference and learning more efficiently (perhaps using the sparsified training method described at the end of Chapter 4) is necessary before neural decoders for these scenarios will be possible.

Finally, we believe that neural decoding algorithms are well-suited for channels which are difficult to model analytically, such as optical communication channels. Currently, high-complexity digital signal processing algorithms are used in receivers for these channels [69]; learning algorithms may offer a lower complexity alternative.

Bibliography

[1] L. Lugosch and W. J. Gross, "Neural offset min-sum decoding," in *2017 IEEE International Symposium on Information Theory*, June 2017, pp. 1361–1365.

[2] E. Nachmani, E. Marciano, L. Lugosch, W. J. Gross, D. Burshtein, and Y. Be'ery, "Deep learning methods for improved decoding of linear codes," *IEEE Journal of Selected Topics in Signal Processing - Special Issue on Machine Learning for Cognition in Radio Communications and Radar*, 2018, to appear.

[3] "Prisma," https://www.prisma-ai.com, accessed: 2017-06-25.

[4] E. Nachmani, E. Marciano, D. Burshtein, and Y. Be'ery, "RNN decoding of linear block codes," *arXiv preprint arXiv:1702.07560*, 2017.

[5] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[6] M. Ibnkahla, "Applications of neural networks to digital communications - a survey," *Signal Processing*, vol. 80, no. 7, pp. 1185–1215, 2000.

[7] N. Wiberg, H.-A. Loeliger, and R. Kotter, "Codes and iterative decoding on general graphs," *Transactions on Emerging Telecommunications Technologies*, vol. 6, no. 5, pp. 513–525, 1995.

[8] D. J. MacKay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[9] T. Gruber, S. Cammerer, J. Hoydis, and S. t. Brink, "On deep learning-based channel decoding," *CISS 2017*, 2017.

[10] E. Nachmani, Y. Be'ery, and D. Burshtein, "Learning to decode linear codes using deep learning," *54th Annual Allerton Conf. on Communication, Control and Computing*, 2016.

[11] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate (corresp.)," *IEEE Transactions on Information Theory*, vol. 20, no. 2, pp. 284–287, Mar 1974.

[12] J. Pearl, "Reverend Bayes on inference engines: A distributed hierarchical approach," *Second National Conference on Artificial Intelligence. AAAI-82*, 1982.

[13] R. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, Sep 1981.

[14] B. J. Frey and D. J. MacKay, "A revolution: Belief propagation in graphs with cycles," in *Advances in Neural Information Processing Systems*, 1998, pp. 479–485.

[15] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[16] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[17] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 2015.

[18] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[20] H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets," *Journal of computer and system sciences*, vol. 50, no. 1, pp. 132–150, 1995.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[22] R. Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, vol. 4, no. 2, pp. 4–22, Apr 1987.

[23] G. Zeng, D. Hush, and N. Ahmed, "An application of neural net in decoding error-correcting codes," in *IEEE International Symposium on Circuits and Systems,*, May 1989, pp. 782–785 vol.2.

[24] J. Bruck and M. Blaum, "Neural networks, error-correcting codes, and polynomials over the binary n-cube," *IEEE Transactions on Information Theory*, vol. 35, no. 5, pp. 976–987, Sep 1989.

[25] W. R. Caid and R. W. Means, "Neural network error correcting decoders for block and convolutional codes," in *Global Telecommunications Conference, 1990, and Exhibition. 'Communications: Connecting the Future', GLOBECOM '90., IEEE,* Dec 1990, pp. 1028–1031 vol.2.

[26] G. Marcone, E. Zincolini, and G. Orlandi, "An efficient neural decoder for convolutional codes," *European transactions on telecommunications and related technologies*, vol. 6, no. 4, pp. 439–445, 1995.

[27] R. Annauth and H. C. S. Rughooputh, "Neural network decoding of turbo codes," in *Neural Networks, 1999. IJCNN '99. International Joint Conference on*, vol. 5, 1999, pp. 3336–3341 vol.5.

[28] W. G. Teich and J. Lindner, "A novel decoder structure for convolutional codes based on a multilayer perceptron," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 1, Nov 1995, pp. 449–454 vol.1.

[29] A. D. Stefano, O. Mirabella, G. D. Cataldo, and G. Palumbo, "On the use of neural networks for Hamming coding," in *1991., IEEE International Sympoisum on Circuits and Systems*, Jun 1991, pp. 1601–1604 vol.3.

[30] A. Esposito, S. Rampone, and R. Tagliaferri, "A neural network for error correcting decoding of binary linear codes," *Neural networks*, vol. 7, no. 1, pp. 195–202, 1994.

[31] I. Ortuno, M. Ortuno, and J. A. Delgado, "Error correcting neural networks for channels with Gaussian noise," in *Proceedings of the IJCNN International Joint Conference on Neural Networks*, vol. 4, Jun 1992, pp. 295–300 vol.4.

[32] S. E. El-Khamy, E. A. Youssef, and H. M. Abdou, "Soft decision decoding of block codes using artificial neural network," in *Proceedings IEEE Symposium on Computers and Communications*, July 1995, pp. 234–240.

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

[34] H. Abdelbaki, E. Gelenbe, and S. E. El-Khamy, "Random neural network decoder for error correcting codes," in *Neural Networks, 1999. IJCNN'99. International Joint Conference on*, vol. 5. IEEE, 1999, pp. 3241–3245.

[35] J.-L. Wu, Y.-H. Tseng, and Y.-M. Huang, "Neural network decoders for linear block codes," *International Journal of Computational Engineering Science*, vol. 3, no. 03, pp. 235–255, 2002.

[36] L. G. Tallini and P. Cull, "Neural nets for decoding error-correcting codes," in *IEEE Technical Applications Conference and Workshops. Northcon/95. Conference Record*, Oct 1995, pp. 89–.

[37] M. E. Buckley and S. B. Wicker, "The design and performance of a neural network for predicting turbo decoding error with application to hybrid ARQ protocols," *IEEE Transactions on Communications*, vol. 48, no. 4, pp. 566–576, Apr 2000.

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

[39] J. Håstad, "Computational limitations of small-depth circuits," Ph.D. dissertation, MIT, 1987.

[40] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in Neural Information Processing Systems*, 2007, pp. 153–160.

[41] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[42] G. Liva, L. Gaudio, T. Ninacs, and T. Jerkovits, "Code design for short blocks: A survey," *arXiv preprint arXiv:1610.00873*, 2016.

[43] J. Jiang and K. R. Narayanan, "Iterative soft-input soft-output decoding of Reed-Solomon codes by adapting the parity-check matrix," *IEEE Transactions on Information Theory*, vol. 52, no. 8, pp. 3746–3756, 2006.

[44] 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.

[45] J. R. Hershey, J. Le Roux, and F. Weninger, "Deep unfolding: Model-based inspiration of novel deep architectures," Sep. 2014, Mitsubishi Electric Research Laboratories, Tech. Rep. TR2014-117.

[46] V. Stoyanov, A. Ropson, and J. Eisner, "Empirical risk minimization of graphical model parameters given approximate inference, decoding, and model structure," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 725–733.

[47] J. Domke, "Parameter learning with truncated message-passing," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2011, pp. 2937–2943.

[48] ——, “Learning graphical model parameters with approximate marginal inference,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 10, pp. 2454–2467, 2013.

[49] M. Helmling, S. Scholl, F. Gensheimer, T. Dietz, K. Kraft, S. Ruzika, and N. Wehn, “Database of Channel Codes and ML Simulation Results,” www.uni-kl.de/channel-codes, 2017.

[50] T. Tieleman and G. Hinton, “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude,” *COURSERA: Neural Networks for Machine Learning*, vol. 4, no. 2, 2012.

[51] I. Dimnik and Y. Be’ery, “Improved random redundant iterative HDPC decoding,” *IEEE Transactions on Communications*, vol. 57, no. 7, pp. 1982–1985, 2009.

[52] T. R. Halford and K. M. Chugg, “Random redundant soft-in soft-out decoding of linear block codes,” in *Information Theory, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 2230–2234.

[53] T. Hehn, J. B. Huber, O. Milenkovic, and S. Laendner, “Multiple-bases belief-propagation decoding of high-density cyclic codes,” *IEEE Transactions on Communications*, vol. 58, no. 1, pp. 1–8, January 2010.

[54] C. Jego and W. J. Gross, “Turbo decoding of product codes based on the modified adaptive belief propagation algorithm,” in *2007 IEEE International Symposium on Information Theory*, June 2007, pp. 641–644.

[55] N. Wiberg, “Codes and decoding on general graphs,” Ph.D. dissertation, Linköping University, 1996.

[56] J. Chen and M. P. Fossorier, “Near optimum universal belief propagation based decoding of low-density parity check codes,” *IEEE Transactions on Communications*, vol. 50, no. 3, pp. 406–414, 2002.

[57] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean *et al.*, "On rectified linear units for speech processing," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 3517–3521.

[58] J. Chen and M. P. Fossorier, "Density evolution for two improved BP-based decoding algorithms of LDPC codes," *IEEE Communications Letters*, vol. 6, no. 5, pp. 208–210, 2002.

[59] K. P. Murphy, Y. Weiss, and M. I. Jordan, "Loopy belief propagation for approximate inference: An empirical study," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1999, pp. 467–475.

[60] S. Hemati and A. H. Banihashemi, "Dynamics and performance analysis of analog iterative decoding for low-density parity-check (LDPC) codes," *IEEE Transactions on Communications*, vol. 54, no. 1, pp. 61–70, 2006.

[61] E. Janulewicz and A. H. Banihashemi, "Performance analysis of iterative decoding algorithms with memory," in *2010 IEEE Information Theory Workshop on Information Theory (ITW 2010, Cairo)*, Jan 2010, pp. 1–5.

[62] S. Hemati, F. Leduc-Primeau, and W. J. Gross, "A relaxed min-sum LDPC decoder with simplified check nodes," *IEEE Communications Letters*, vol. 20, no. 3, pp. 422–425, March 2016.

[63] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.

[64] A. Karpathy, "Connecting images and natural language," Ph.D. dissertation, Stanford University, 2016.

[65] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural networks*, vol. 1, no. 4, pp. 339–356, 1988.

[66] R. Lucas, M. Bossert, and M. Breitbach, “On iterative soft-decision decoding of linear binary block codes and product codes,” *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 2, pp. 276–296, Feb 1998.

[67] T. Xia and H. C. Wu, “Blind identification of nonbinary LDPC codes using average LLR of syndrome a posteriori probability,” *IEEE Communications Letters*, vol. 17, no. 7, pp. 1301–1304, July 2013.

[68] ——, “Novel blind identification of LDPC codes using average LLR of syndrome a posteriori probability,” *IEEE Transactions on Signal Processing*, vol. 62, no. 3, pp. 632–640, Feb 2014.

[69] J. C. Cartledge, F. P. Guiomar, F. R. Kschischang, G. Liga, and M. P. Yankov, “Digital signal processing for fiber nonlinearities,” *Optics Express*, vol. 25, no. 3, pp. 1916–1936, 2017.