Deep Learning for Channel Coding

Yihan Jiang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Sreeram Kannan, Chair

Sewoong Oh

Jenq-Neng Hwang

Program Authorized to Offer Degree:
Electrical and Computer Engineering

University of Washington

Abstract

Deep Learning for Channel Coding

Yihan Jiang

Chair of the Supervisory Committee:
Professor Sreeram Kannan
Electrical and Computer Engineering

Wireless Communication has become a critical backbone of the information economy in the past few decades. In this rapidly improving telecommunications landscape, a crucial role is played by channel codes. Channel coding refers to the coding of information in such a way that the transmission can be robustly decoded even under noisy conditions.

Progress in channel coding research has been powered by sophisticated mathematics and driven solely by human ingenuity, and therefore, progress is necessarily sporadic. For example, from convolutional code (2G) to Polar code (5G), it took several decades of research efforts to develop a new generation of channel codes. Deep learning has revolutionized a wide variety of fields and modern AI systems built using deep learning techniques are now able to surpass humans as well as human-designed algorithms. Motivated by the success of deep learning in other fields, in this thesis, we study the role of deep learning in tackling telecommunication system design.

The first part of this thesis shows that three major channel coding problems: (a) decoder design, (b) code design, and (c) feedback code design, can be automated by applying end-to-end deep supervised learning with near-optimal performance. We show surprisingly in several of these scenarios that even when the communication channels follow well studied and canonical (text-book) models, there is a significant performance improvement from deep-learning. This can be attributed not only to the ability of the learning-based methods

to adapt to the channel statistics (because it is a well-known channel), but also to its ability to design sophisticated non-linear algorithms for both encoding and decoding.

The second part of this thesis studies other learning paradigms such as meta learning and federated learning (FL), which can be applied to channel coding problems to further demonstrate the versatility of neural networks. Meta learning based neural decoder shows significant efficiency on data and computation, compared to naive fine-tuning. Finally, inspired by the strong algorithmic connection between FL personalization and meta learning, we propose a personalized FL algorithm which improves personalization significantly.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I consider myself fortunate to be able to get the opportunity to pursue my Ph.D. at the University of Washington (UW). As the Elder says: "One's destiny, of course, depends on self-perfection, but it also needs to take the historical progress into account." Standing on the shoulder of giants and inspired by my mentors, I am fortunate to make tiny contribution to the research community.

First of all, I would like to thank my advisor Professor Sreeram Kannan, who allowed me to work under his guidance. Professor Kannan is passionate, patient, supportive and open to new directions. Working under his guidance provided me with ample flexibility to explore diverse areas of research. Whenever I faced any hurdle in research, he was always there beside me. I can recollect his smiling face providing me assurance that we will walk the troubled, uncharted lands together. He was always full of ideas and eager to remove my obstacles in research. I consider him my role model in research. I would also like to thank my other committee members, Professor Sewoong Oh, Professor Jenq-Neng Hwang, and Professor Arvind Krishnamurthy, for agreeing to be on my dissertation committee. Their insight and feedback helped me improve my research.

I convey my sincere thanks to Professor Hyeji Kim, who has been my collaborator and mentor since the early days of my Ph.D. Her insights played an indispensable role in all my research projects. No words are sufficient to express my appreciation for the guidance I received from her. I extend my gratitude to my collaborators Professor Sewoong Oh, Professor Pramod Viswanath, and Professor Himanshu Asnani. Under their supervision, our research achieved new heights, far more than what I had initially anticipated. Many thanks to Yixing Lao, who introduced me to deep learning. I still remember that afternoon in 2014 when I

was awestruck by his presentation on AlexNet.

I also want to thank Dr. Jakub Konečnỳ and Dr. Keith Rush under whom I completed my summer internship in Google Beijing. They introduced me to the brand new field of Federated Learning, which has fascinated me ever since. Thanks to Aira Technology, Anand Chandrasekher, Dr. RaviKiran Gopalan, Sandeep KesiReddy, Sun Kang, and Arman Rahimzamani for giving me the opportunity to extend my academic research ideas into industrial applications. I want to thank Kuppan Premlatha, Reevan Jankeel, Dr. Paolo Minero, Professor Fan-gang Zeng, Professor Truong Nguyen for their mentorship and support.

I am grateful to all of my labmates and friends: I have randomly sampled some names here from an infinite list without order. UW: Dr. Sudipto Mukherjee, Dr. Shunfu Mao, Arman Rahimzamani, Bowen Xue, we have shared the same cubicle and engaged in interesting discussions every day for years. Other labmates: Soubhik Deb, Viswa Virinchi Muppirala, Robert Raynor, Nishant Relan, Edwin Mathew, Namit Chauhan, Jianxiong Zhou, Saket Mangalam; Other collaborators: Ranvir Rana, Ashwin Hebbar. Peer, seniors, and friends: Yue Sun, Yize Chen, Baicen Xiao, Lvtianyang Zhang, Hao Yin, Cunzhi Ren, Liyuan Zheng, Xun Sun, Yueyang Chen, Zhihang Dong, Ting Liu, Zhengde Zhao, Chun-Wei Chen, Ying-Hsiu Chou, Tony Tung, Yu-Chia Chen, Naoco Oguri, Dr. Kevin Lin, Dr. Huazeng Deng, Dr. Yuanyuan Shi, Dr. Xuhang Ying, Dr. Zhipeng Liu, Professor Hao Wang, Tianyi Zhou. Dr.Zhengli Zhao, Dr. Weihao Gao, Dr. Wenhao Wu. I am also grateful to my long-time friends known in San Diego: Professor Chicheng Zhang, Dr. Hao Zhuang, Dr. Yacong Ding, Professor Jinxing Li, Dr. Yunqi Zhang, Dr. Yuxiang Zhang, Dr. Ning Ma, Dr. Jiangcheng Zhu. Miao Li, Hao Jiang, Boyan Zhang, Liming Wei, Xichang Wu, Jiechun Sun, Andi Zhao, Shiyang Gao, Ning Liu, Guobi Zhao, Jiance Tong, Bochao Yue, Wei Zhao, Ang Gao, Sisi Xi, Weihe Wang, Yue Fan, Shengqiong Xie, Zhujun Wang, Hongpeng Wang, Han Li, Xiang Xu, Boneng Zhang, Jing Yan, Han Liu, Yichi Zhang, Xu Han, Dr. Phillip Kyriakakis. Kudos to my long-time friends before coming to the U.S: Zhongen Tao, Xi Han, Chenlin Du, Dacheng

Liu, Shuo Mi, Chenguang Guo, Haiyan Yu, Yufei Ji, Jiuling Li, Hanqing Zhang, Ji'ao Zhang, Yi Zhen, Jiajun Liu, Dr. Jianbo Yuan, Zhilun Li, Yiming Chu, Shaocheng Cui, Di Bao, Dr. Changjiang Liu, Dr. Honggang Zhang, Dr. Zhenhong Chen, Dr. Kun Yang, Jialin Liu, Dongsheng Han, Dr. Lei Shen, Fushun Su, Kai Liu, and, always, Sida Kang.

Words are not enough to convey my gratitude to Shenghui Shao. Without her love and company, I could not have survived the brutal pressure of Ph.D. Finally, thanks to my family. I am proud to get my "Dr." title, following the footsteps of my father, Professor Shouda Jiang, and my mother, Dr. Limei Yu. Without their unconditional love and support, I could have never reached here.

DEDICATION

to Kobe Bryant

Chapter 1

INTRODUCTION

> To carve a native jade, one needs a non-native stone.
>
> Shijing, Classic of Poetry.

1.1 Background

Deep learning methods have been remarkably successful in an extensive range of engineering fields, ranging from computer vision [68], and natural language processing [28], to gaming [111]. This is because, they are able to effectively utilize the large volumes of data to learn as well as leverage the high computational power to iteratively improve the models. The tasks where deep learning significantly outperforms other methods can be divided into two categories: **model deficiency** and **algorithm deficiency**. Real-world data from tasks such as image classification [22] cannot be modeled accurately using clean mathematical models, we term this "model deficiency". In the absence of simple mathematical models, algorithms utilizing handcrafted features have been traditionally used. However, deep learning algorithms can significantly outperform prior art by effecitiently learning novel non-linear representations exploiting the large amount of data. "Algorithm deficiency" can arise even in environments with well-defined models and simple rules (like Go and Chess), when the design space of algorithms is astronomically large. This makes it difficult to hand-design optimal algorithms. On the other hand, deep-learning based algorithms such as AlphaGo significantly outperforms both humans and human-designed algorithms [111, 112, 107]. Therefore, we expect that deep learning algorithms can work well on an even broader range of applications where either

model-deficiency or algorithm-deficiency holds, as long as there is enough data (i.e, data suffiency).

The smartphone has revolutionized modern society in unprecedented ways. Underlying the ubiquity of the smartphone is a critical technological infrastructure, powered by wireless communication (cellular, Bluetooth, Wifi, and satellite) and wireline communication (DSL modems, cable, and ethernet) technologies.

Underlying these communication systems, channel coding is a mission-critical module, which allows the transmission to be decoded at the receiver in a robust and computationally-efficient manner even under noisy conditions. Since the birth of information theory [109], the discipline of coding theory has made much progress in designing near-optimal codes. In the past few decades, near-optimal codes such as turbo codes, low-density parity-check (LDPC) codes, and, recently, polar codes significantly impacted every cellular phone, which features global cellular standards ranging from the 2nd generation to the 5th generation respectively [118]. Although achieving near-capacity performance, the traditional channel coding approach has the following caveats:

(a) **model deficiency:** Channel code design heavily relies on handcrafted optimal algorithms for the canonical Additive White Gaussian Noise (AWGN) channel, where the signal is corrupted by i.i.d. Gaussian noise. Practical channels deviate significantly from AWGN settings, where often sub-optimal heuristics are used.

(b) **algorithm deficiency:** Channel codes are designed for a finite block length K, which denotes the number of encoded bits. Channel codes are guaranteed to be optimal only when the block-length approaches infinity and are near-optimal in practice only when the block-length is large. On the other hand, under short and moderate block length regimes, there is much room for improvement. Moreover, even under some simple canonical channels like the feedback channel, designing optimal codes remains an elusive open problem.

The development of channel coding has been driven primarily by human ingenuity, and therefore progress has been necessarily sporadic. For example, from convolutional code (2G) to Polar code (5G), it took several decades to develop a new generation of channel codes.

As pointed out earlier, deep learning can potentially handle problems with model-deficiency and algorithm-deficiency. The channel coding problem is an excellent fit not only because it displays these two criteria, but also additionaly, there is plenty of data available from both real-word and simulated models. We therefore envision that deep learning can play a critical role in tackling channel coding problems.

1.2 Main Contributions

The first half of this thesis designs point-to-point channel codes via deep learning in the supervised learning paradigm. This thesis firstly studies **neural decoder**, as shown in Figure 1.1 top. We study neural decoder for sequential channel codes, namely, Convolutional code and Turbo code. Sequential code decoders have natural recurrent structures, which bears structural similarity to recurrent neural networks (RNN). Inspired by this similarity, we build a RNN-based neural decoder for both these codes. The decoder achieves the optimal performance on AWGN channel (matching state-of-the-art decoders), as well as outperforming them under non-AWGN channels. Moreover, the RNN-based decoder achieves good performance generalizing over a wide range of block lengths and signal-to-noise ratios (SNR), indicating that the neural decoder learns the sophisticated decoding algorithm rather than just memorizing the training data.

The problem of designing **neural code**, as shown in Figure 1.1 middle, is studied by jointly designing the encoder and decoder via deep learning. We find that naively applying general-purpose neural networks in this context does not produce codes whose performance improves with blocklength (i.e., there is no coding gain). By designing a novel neural structure combining deep learning and structural knowledge from decades of coding theory, our proposed Turbo Autoencoder (TurboAE) matches canonical capacity-approaching code on the AWGN channel. On non-AWGN channels, TurboAE outperforms the canonical approach. TurboAE automates the search for codes, showing its effectiveness as an alternative method for tackling channel coding problems.

The problem of **neural feedback code** is an elusive open problem, as shown in Figure

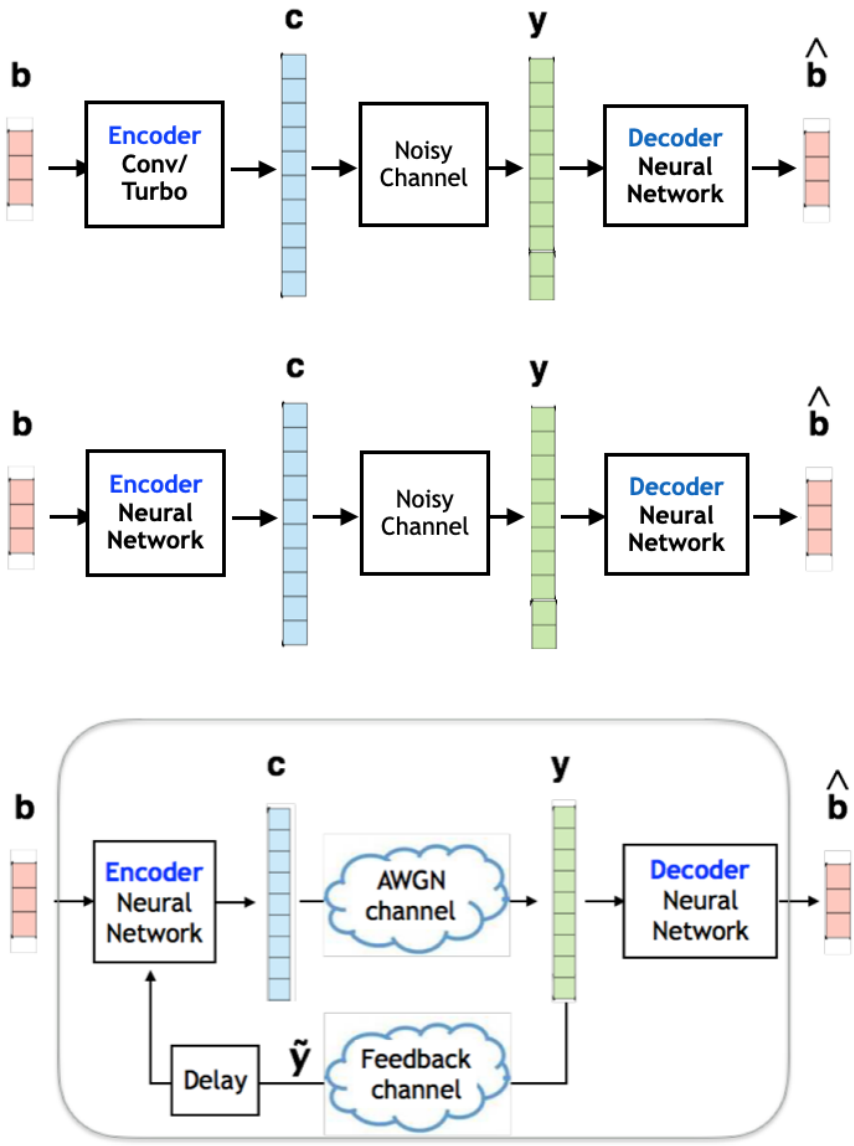


Figure 1.1: Neural channel coding problems: (top) neural decoder design, (middle) neural code design and (bottom) neural feedback code design

1.1 bottom. Since all practical codes are linear and linear codes are known to achieve capacity without feedback, a first attempt at code design will involve linear codes. However, theoretical results have shown that no linear code incorporating the noisy output feedback can perform better than simply neglecting feedback [62]. Our proposed neural feedback code learns

sophisticated non-linear encoding and decoding algorithm, outperforming known codes in reliability by a large margin even in the AWGN scenario. This surprising result indicates that deep learning can succeed even when there is no model deficiency (since we work with simple well-stuied AWGN models), due to the presence of algorithm-deficiency.

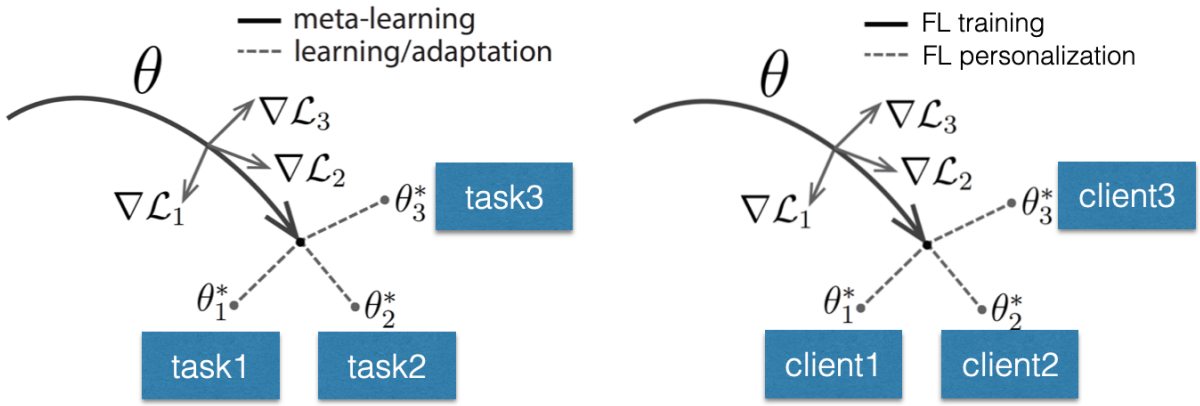


Figure 1.2: Model Agnostic Meta Learning (left) and connection to FL personalization (right). Figure adapted from [33]

.

Beyond supervised learning, the second half of this thesis studies two paradigms: **meta learning** and **federated learning (FL)**, in the context of applying neural decoder on heterogeneous environments.

Meta learning is a machine learning paradigm aimed at conducting fast adaptation of a general purpose model to a scenario of interest with only a few training examples. The adaptation of a channel decoder to varying channel conditions can be interpreted as a meta-learning problem. Trained with gradient-based model agnostic meta learning (MAML) [33] method as shown in Fig 1.2 left, the adapted neural decoder can achieve near-optimal performance for a wide range of channel models, with only few-shot preamble bits. MAML-based neural decoder shows significant efficiency on data and computation compared to naive

fine-tuning approach.

While the rest of the thesis focused on applications of deep learning to communications, the final topic focuses on communication-limited learning, taking the interface in the other direction. We consider the problem of federated learning (FL), where a server tries to learn a statistcial model from data distributed in different nodes, while preserving user privacy. The distributed nature of the problem requires communication-efficient algorithms. Furthermore, since different nodes may have data with (slightly) different distributions, this necessiates personalization of the jointly learnt model locally. We study this problem of personalized federated learning.

Existing FL personalization methods disconnect training and personalization, and therefore, are sub-optimal when dealing with heterogeneous clients. The strong connection between FL and MAML, as shown in Figure 1.2, suggests the usage of meta learning algorithms to improve FL personalization. We propose an algorithmic framework to connect FL personalization and meta learning, which shows how existing FL algorithm can be simply configured to improve its personalization performance. We demonstrate excellent peformance empirically on multiple FL benchmarks.

1.3 Orgainization

This thesis is organized as shown in Fig 1.3. Each chapter is designed to be self-contained.

Chapter 2 studies the **neural decoder**. These methods were previously published as in [61] and [51].

Chapter 3 studies the problem of **neural code**, to automate the code design via deep learning. These methods were previously published in [55], [53] and [54].

Chapter 4 studies the **neural feedback code** problem. This method was previously published in [60].

Section 5 studies the fast adaptation of neural decoders via MAML, which builds on the context of neural decoder. This method was previously published as in [52].

Section 6 studies FL personalization. A strong algorithmic connection between FL

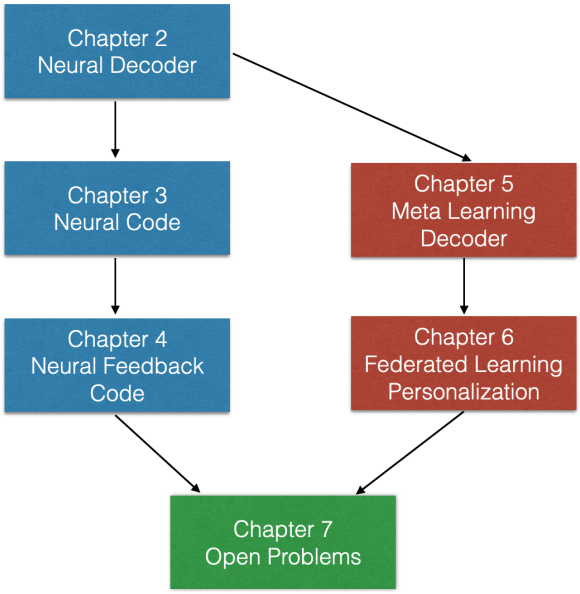


Figure 1.3: Paper Structure

personalization and MAML is studied, which leads to the proposed personalized FL algorithm that improves performance under multiple benchmarks. This method was previously published as in [56].

Finally, in section 7, we conclude the thesis.

Chapter 2

NEURAL DECODERS

In this chapter [1], we study whether it is possible to **automate** the discovery of decoding algorithms via deep learning. We study a family of sequential codes parameterized by RNN architectures. We show that novelly designed and trained RNN architectures can decode well-known sequential codes such as the convolutional and turbo codes with close to optimal performance on the additive white Gaussian noise (AWGN) channel. Canonical results are achieved by breakthrough algorithms of our times (Viterbi [122] and BCJR [8] decoders, representing dynamic programing and forward-backward algorithms). Moreover, we propose a fully end-to-end trained neural Turbo decoder [11] without BCJR knowledge, which shows improved performance comparing to existing decoders. We show strong generalizations, i.e., we train at a specific signal to noise ratio and block length but test at a wide range of these quantities and test the robustness and adaptivity to deviations from the AWGN setting. This chapter is previously published in [61], and [51].

2.1 Channel Coding Decoder Problem Formation

The channel coding decoder design problem is shown in Figure 2.1, with input message $b \in \{0,1\}^k$, where the message has i.i.d Bernoulli distribution with equal probability; given encoder $c = g(b)$, where $c \in \{-1,+1\}^n$; and channel $y = channel(c)$, where $y \in R$ depends on the channel model. Decoder aims at reconstructing the original message $\hat{b} = f_W(y)$, where $\hat{b} \in \{0,1\}^k$.

While the learning framework is clear and there is virtually unlimited training data

[1] The material in this chapter is based on joint work with Hyeji Kim, Ranvir Rana, Himanshu Asnani, Sreeram Kannan, Sewoong Oh, Pramod Viswanath

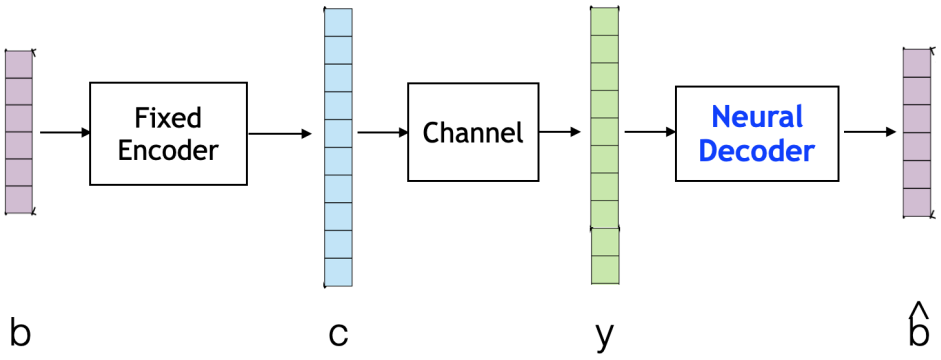


Figure 2.1: Neural Decoder Problem

available (via simulating the channel), there are two main challenges: (a) The space of codes is vast and the sizes astronomical; for instance, a rate $1/2$ code over 100 information bits involves designing 2^{100} codewords in a 200 dimensional space. Computationally efficient encoding and decoding procedures are a must, apart from high reliability over the AWGN channel. (b) Generalization is highly desirable across block lengths and data rates that each work very well over a wide range of channel signal to noise ratios (SNR). In other words, one is looking to design a family of codes (parametrized by data rate and the number of information bits), and their performance is evaluated over a range of channel SNRs. For example, it is shown that when a neural decoder is exposed to nearly 90% of the codewords of a rate $1/2$ polar code over 8 information bits, its performance on the unseen codewords is poor [37].

In part due to these challenges, recent deep learning works on decoding known codes using data-driven neural decoders have been limited to short or moderate block lengths [37, 16, 29, 89]. Other deep learning works on coding theory focus on decoding known codes by training a neural decoder that is initialized with the existing decoding algorithm but is more general than the existing algorithm [85, 128].

The main challenge is to restrict oneself to a class of codes that neural networks can naturally encode and decode. In this chapter, we restrict ourselves to a class of *sequential*

encoding and decoding schemes, to which convolutional and turbo codes belong. These sequential coding schemes naturally meld with the family of recurrent neural network (RNN) architectures, which have recently seen immense success in a wide variety of time-series tasks.

Working within sequential codes parametrized by RNN architectures, we make the following contributions.

(1) Focusing on convolutional codes, we aim to decode them on the AWGN channel using RNN architectures. Efficient optimal decoding of convolutional codes has historically represented fundamental progress in the broad arena of algorithms. Optimal bit error decoding is achieved by the Viterbi decoder [122] which is dynamic programming or Dijkstra's algorithm on a specific graph (the 'trellis') induced by the convolutional code. Optimal block error decoding is the BCJR decoder [8] which is part of a family of forward-backward algorithms.

Early work had shown that vanilla-RNNs are capable in principle of emulating both Viterbi and BCJR decoders [124, 105]. We show empirically, through careful construction of RNN architectures and training methodology, that neural network decoding is possible at very near-optimal performances (both bit error rate (BER) and block error rate (BLER)). The critical point is that we train an RNN decoder at a specific SNR and over short information bit lengths (100 bits) and show strong generalization capabilities by testing over a wide range of SNR and block lengths (up to 10,000 bits). The specific training SNR is closely related to the Shannon limit of the AWGN channel at the code's rate and provides vital information-theoretic collateral to our empirical results.

(2) Turbo codes are naturally built on top of convolutional codes, both in encoding and decoding. A natural generalization of our RNN convolutional decoders allows us to decode turbo codes at BER comparable to and at specific regimes, even better than the state-of-the-art turbo decoders on the AWGN channel. Moreover, we propose a fully end-to-end trained neural decoder: DeepTurbo [51], without requiring the knowledge of the BCJR algorithm. That data-driven, SGD-learnt, RNN-based DeepTurbo can decode comparably is relatively remarkable since Turbo codes already operate near the Shannon limit of reliable communication over the AWGN channel.

(3) We show the afore-described neural network decoders for both convolutional and turbo codes are *robust* to variations to the AWGN channel model. We consider a problem of real-world interest: communication over a "bursty" AWGN channel. Bursty AWGN channel means a small fraction of noise has much higher variance than usual, which is used in modeling inter-cell interference in OFDM cellular systems used in 4G and 5G cellular standards and co-channel radar interference. We empirically demonstrate the neural network architectures can adapt to such variations and beat the state-of-the-art heuristics comfortably (despite evidence elsewhere that neural networks are sensitive to models they are trained on [115]). Via an innovative local perturbation analysis (akin to [97]), we demonstrate the neural network to have learned sophisticated preprocessing heuristics in the engineering of real-world systems [73].

2.2 Neural Decoder for Convolutional Code

2.2.1 Convolutional Code Introduction

Convolutional Code is widely used in communication systems nowadays, including digital video, radio, Bluetooth, mobile communication, despite its long history. Before Turbo codes, convolutional code concatenated Reed-Solomon code was the most efficient solution.

A standard example of a convolutional code is the *rate-1/2 Recursive Systematic Convolutional (RSC) code*. The encoder performs a forward pass on a recurrent network shown in Figure 2.2. on binary input sequence $\mathbf{b} = (b_1, \ldots, b_K)$, which we call *message bits*, with binary vector states $(s_1, ..., s_K)$ and binary vector outputs $(c_1, ..., c_K)$, which we call *transmitted bits* or a *codeword*. At time k with binary input $b_k \in \{0, 1\}$ and the state of a two-dimensional binary vector $s_k = (s_{k1}, s_{k2})$, the output is a two-dimensional binary vector $c_k = (c_{k1}, c_{k2}) = (2b_k - 1, 2(b_k \oplus s_{k1}) - 1) \in \{-1, 1\}^2$, where $x \oplus y = |x - y|$. The state of the next cell is updated as $s_{k+1} = (b_k \oplus s_{k1} \oplus s_{k2}, s_{k1})$. Initial state is assumed to be 0, i.e., $s_1 = (0, 0)$.

The $2K$ output bits are sent over a noisy channel, with the canonical one being the

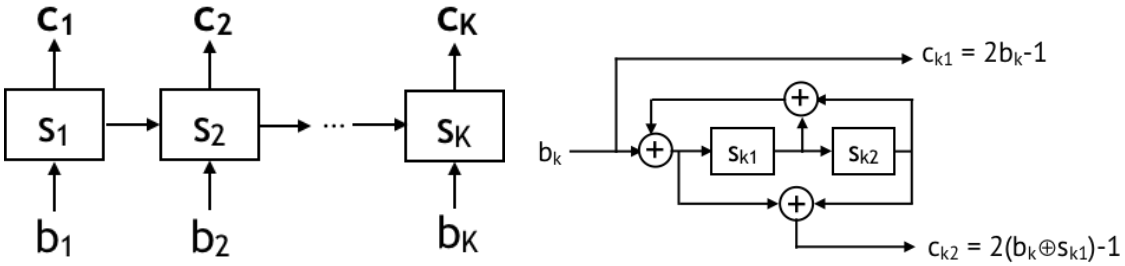


Figure 2.2: (Left) Sequential encoder is a recurrent network, (Right) One cell for a rate 1/2 RSC code

AWGN channel: the received binary vectors $\mathbf{y} = (y_1, \ldots, y_K)$, which are called the *received bits*, are $y_{ki} = c_{ki} + z_{ki}$ for all $k \in [K]$ and $i \in \{1, 2\}$, where z_{ki}'s are i.i.d. Gaussian with zero mean and variance σ^2. Decoding a received signal y refers to (attempting to) finding the maximum a posteriori (MAP) estimate. Due to the simple recurrent structure, efficient iterative schemes are available for finding the MAP estimate for convolutional codes [122, 8].

There are two types of MAP decoders depending on the error criterion in evaluating the performance: bit error rate (BER) or block error rate (BLER).

BLER counts the fraction of blocks that are wrongly decoded (assuming many such length-K blocks have been transmitted), and matching optimal MAP estimator is $\hat{\mathbf{b}} = \arg\max_{\mathbf{b}} \Pr(\mathbf{b}|\mathbf{y})$. Using dynamic programming, one can find the optimal MAP estimate in time linear in the block length K, which is called the *Viterbi algorithm*.

BER counts the fraction of bits that are wrong, and matching optimal MAP estimator is $\hat{b}_k = \arg\max_{b_k} \Pr(b_k|\mathbf{y})$, for all $k = 1, \cdots, K$. Again using dynamic programming, the optimal estimate can be computed in $O(K)$ time, which is called the *BCJR algorithm*, as shown in Figure 2.3 left.

In both cases, the linear time optimal decoder crucially depends on the recurrent structure of the encoder. This structure can be represented as a hidden Markov chain (HMM), and both decoders are special cases of general efficient methods to solve inference problems on HMM using the principle of dynamic programming (e.g. belief propagation). These methods

efficiently compute the exact posterior distributions in two passes through the network: the forward pass and the backward pass.

2.2.2 Neural Convolutional Code Decoder via RNN

Our first aim is to train a recurrent neural network from samples only, without explicitly specifying the underlying probabilistic model and still recovering the matching optimal decoders' accuracy. At a high level, we want to prove by a constructive example that highly engineered dynamic programming can be matched by a neural network that only has access to the samples. The challenge lies in finding the right architecture and design the right training algorithm in a data-driven manner.

As shown in Figure 2.3, bi-GRU and BCJR algorithm are very similar in structure, and the bi-GRU neural decoder has the structural ability to learn the BCJR algorithm.

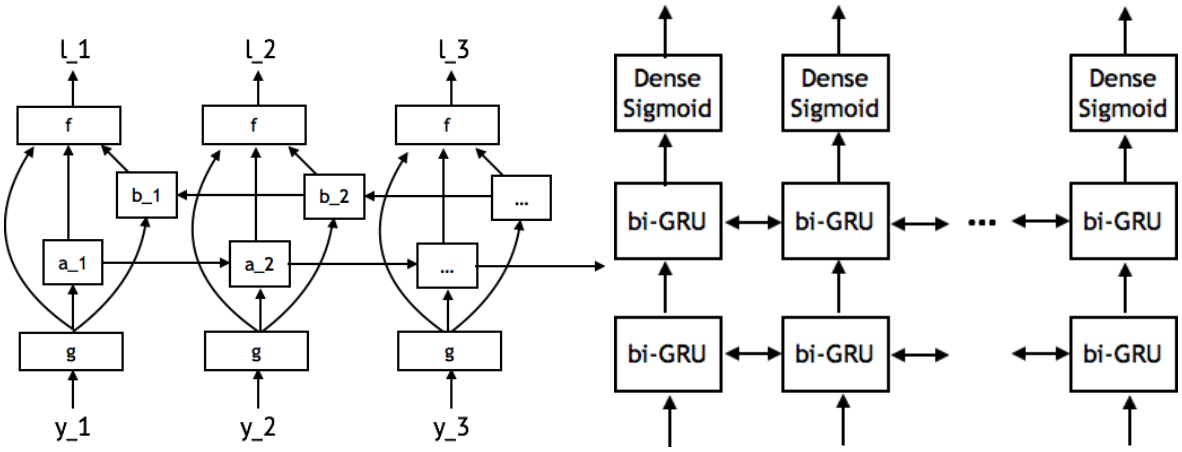


Figure 2.3: BCJR algorithm (left) and N-RSC: Neural decoder for RSC codes by bi-GRU(right)

We treat the decoding problem as a K-dimensional binary classification problem for each of the *message bits* b_k. The input to the decoder is a length-$2K$ sequence of received bits $\mathbf{y} \in R^{2K}$ each associated with its length-K sequence of "true classes" $\mathbf{b} \in \{0,1\}^K$. The goal is to train a model to find an accurate sequence-to-sequence classifier. The input $\mathbf{y}$ is a noisy version of the class $\mathbf{b}$ according to the rate-1/2 RSC code defined earlier in this section. We

generate N training examples $(\mathbf{y}^{(i)}, \mathbf{b}^{(i)})$ for $i \in [N]$ according to this joint distribution to train our model.

We introduce a novel neural decoder for rate-1/2 RSC codes, which we call NRSC. It is two layers of bi-direction Gated Recurrent Units (bi-GRU), each followed by batch normalization units, and the output layer is a single fully-connected sigmoid unit. Let W denote all the parameters in the model whose dimensions are shown in Figure 2.4, and $f_W(\mathbf{y}) \in [0,1]^K$ denote the output sequence. The k-th output $f_W(\mathbf{y})_k$ estimates the posterior probability $\Pr(b_k = 1|\mathbf{y})$, and we train the weights W to minimize the L_w error with respect to a choice of a loss function $\ell(\cdot,\cdot)$ specified below:

$$L \;=\; \sum_{i=1}^{N}\sum_{k=1}^{K} \ell(f_W(\mathbf{y}^{(i)})_k, b_k^{(i)}) \;. \tag{2.1}$$

We use RNNs as a building block. Among several options of designing RNNs, we make three specific choices that are crucial in achieving the target accuracy: (i) bidirectional GRU as a building block instead of unidirectional GRU; (ii) 2-layer architecture instead of a single layer; and (iii) using batch normalization. Unidirectional GRU fails because the underlying dynamic program requires bi-directional recursion of both forward pass and backward pass through the received sequence. A single layer bi-GRU fails to give the desired performance, and two layers are sufficient [2]. Batch Normalization (BatchNorm) is also critical in achieving the target accuracy. Without BatchNorm, the training becomes more unstable, and accuracy drops slightly.

2.2.3 Performance on AWGN channel

In Figure 2.5, for various test SNR, we train our NRSC on randomly generated training data for rate-1/2 RSC code of block length 100 over AWGN channel with proposed training SNR of $\min\{\mathrm{SNR}_{\mathrm{test}}, 1\}$. We trained the decoder with Adam optimizer with learning rate 1e-3, batch size 200, and a total number of examples is 12,000, and we use clip norm. On the

[2] more details refer to [61] appendix for more details

Layer	Output dimension
Input	(K, 2)
bi-GRU	(K, 400)
Batch Normalization	(K, 400)
bi-GRU	(K, 400)
Batch Normalization	(K, 400)
Dense (sigmoid)	(K, 1)

Figure 2.4: N-RSC structure and parameters

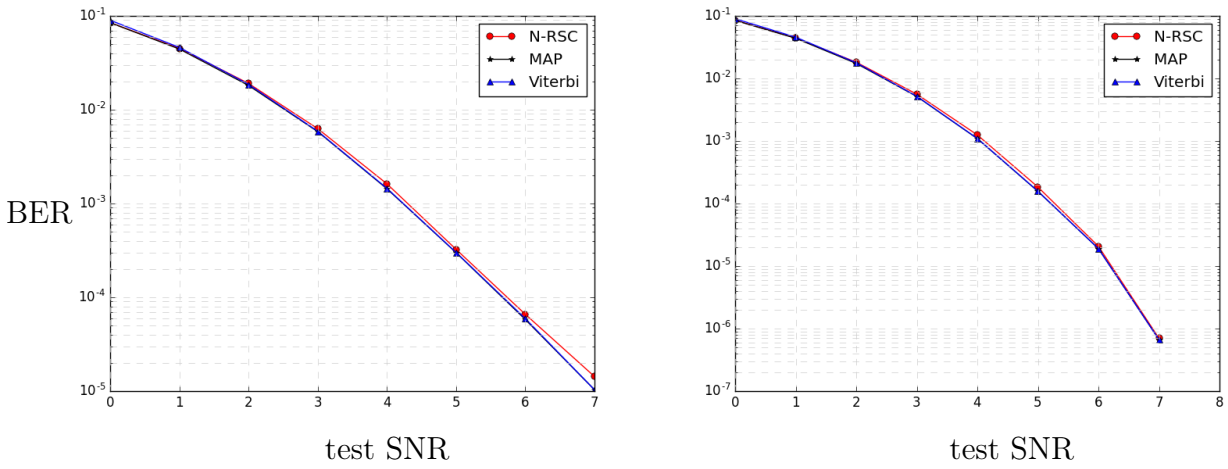


Figure 2.5: Rate-1/2 RSC code on AWGN. BER vs. SNR for block length (left) 100 and (right) 10,000

left, we show bit-error-rate when tested with length 100 RSC encoder, matching the training data. We show that NRSC can learn to decode and achieve the optimal performance of the optimal dynamic programming (MAP decoder) almost everywhere. Perhaps surprisingly, we show on the right figure that we can use the neural decoder trained on length 100 codes, and apply it directly to codes of length $10,000$ and still meet the optimal performance. Note that we only give $12,000$ training examples, while the number of unique codewords is $2^{10,000}$.

This shows that the proposed neural decoder (a) can generalize to unseen codeword; and (b) seamlessly generalizes to block lengths significantly longer. We also note that training with $b_k^{(i)}$ in decoding convolutional codes also gives the same final BER performance as training with the posterior $\Pr(b_k = 1|\mathbf{y}^{(i)})$.

2.2.4 Optimal Training SNR for Neural Decoders

we provide a guideline for choosing the training examples that improve the accuracy. As it is natural to sample the training data and test data from the same distribution, one might use the same noise level for testing and training. However, this is not reliable as shown in Figure 2.6.

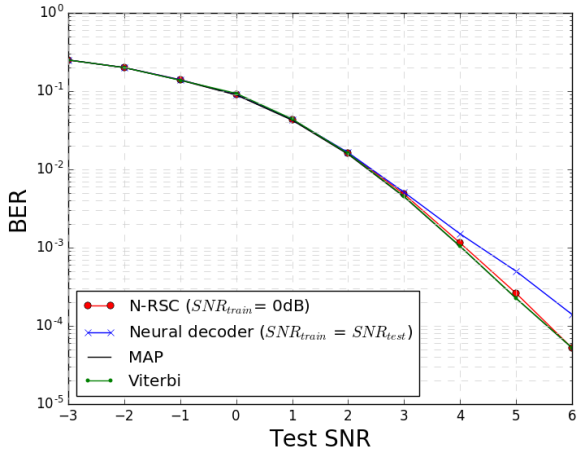


Figure 2.6: BER vs. test SNR for 0dB training and mismatched SNR training in decoding rate-1/2 RSC code of block length 100

Channel noise is measured by Signal-to-Noise Ratio (SNR) defined as $-10\log_{10}\sigma^2$ where σ^2 is the variance of the Gaussian noise in the channel. For rate-1/2 RSC code, we propose using training data with noise level $\mathrm{SNR}_{\mathrm{train}} = \min\{\mathrm{SNR}_{\mathrm{test}}, 0\}$. Namely, we propose using training SNR matched to test SNR if test SNR is below $0dB$, and otherwise fix training SNR at $0dB$ independent of the test SNR.

As it is natural to sample the training data and test data from the same distribution, one might use the same noise level for testing and training. However, this matched SNR is not reliable, as shown in Figure 2.6. We give an analysis that predicts the appropriate choice of training SNR that might be different from testing SNR and justify our choice via comparisons over various pairs of training and testing SNRs.

We conjecture that the optimal training SNR that gives the best BER for a target testing SNR depends on the coding rate. A coding rate is defined as the ratio between the length of the message bit sequence K and the length of the transmitted codeword sequence $\mathbf{c}$. The example we use in this chapter is a rate $r = 1/2$ code with length of $\mathbf{c}$ equal to $2K$. For a rate r code, we propose using training SNR according to

$$\mathrm{SNR}_{\mathrm{train}} = \min\{\mathrm{SNR}_{\mathrm{test}}, 10\log_{10}(2^{2r} - 1)\}\ , \tag{2.2}$$

and call the knee of this curve $f(r) = 10\log_{10}(2^{2r} - 1)$ a threshold. In particular, this gives $\mathrm{SNR}_{\mathrm{train}} = \min\{\mathrm{SNR}_{\mathrm{test}}, 0\}$ for rate 1/2 codes. In Figure 2.7 left, we train our neural decoder for RSC encoders of varying rates of $r \in \{1/2, 1/3, 1/4, 1/5, 1/6, 1/7\}$ whose corresponding $f(r) = \{0, -2.31, -3.82, -4.95, -5.85, -6.59\}$. $f(r)$ is plotted as a function of the rate r in Figure 2.7 right panel. Compared to the grey shaded region of empirically observed region of training SNR that achieves the best performance, we see that it follows the theoretical prediction up to a small shift. The figure on the left shows empirically observed the best SNR for training at each testing SNR for various rate r codes. We can observe that it follows the trend of the theoretical prediction of a curve with a knee. Before the threshold, it closely aligns with the 45-degree line $\mathrm{SNR}_{\mathrm{train}} = \mathrm{SNR}_{\mathrm{test}}$. Around the threshold, the curves become constant functions.

We derive the formula in equation 2.2 in two parts. When the test SNR is below the threshold, then we are targeting for bit error rate (and similarly the block error rate) of around $10^{-1} \sim 10^{-2}$. This implies that a significant portion of the testing examples lies near the decision boundary of this problem. Hence, it makes sense to show matching training

examples, as a significant portion of the training examples will also be at the boundary, which is what we want to maximize the use of the samples. On the other hand, when we are above the threshold, our target bit-error-rate can be significantly smaller, say 10^{-6}. In this case, most of the testing examples are *easy*, and only a tiny proportion of the testing examples lie at the decision boundary. Hence, if we match training SNR, most of the examples will be wasted. Hence, we need to show those examples at the decision boundary, and we propose that the training examples from SNR $10\log_{10}(2^{2r}-1)$ should lie near the boundary. This is a crude estimate but effective, and can be computed using the capacity-achieving random codes for AWGN channels and the distances between the codes words at capacity. Capacity is a fundamental limit on what rate can be used at a given test SNR to achieve small errors. In other words, for a given test SNR over AWGN channel, Gaussian capacity gives how closely we can pack the codewords (the classes in our classification problem) to be as densely packed as possible. This gives us a sense of how decision boundaries (as measured by the test SNR) depend on the rate. It is given by the Gaussian capacity rate $= 1/2log(1+SNR)$. Translating this into our setting, we set the desired threshold that we seek.

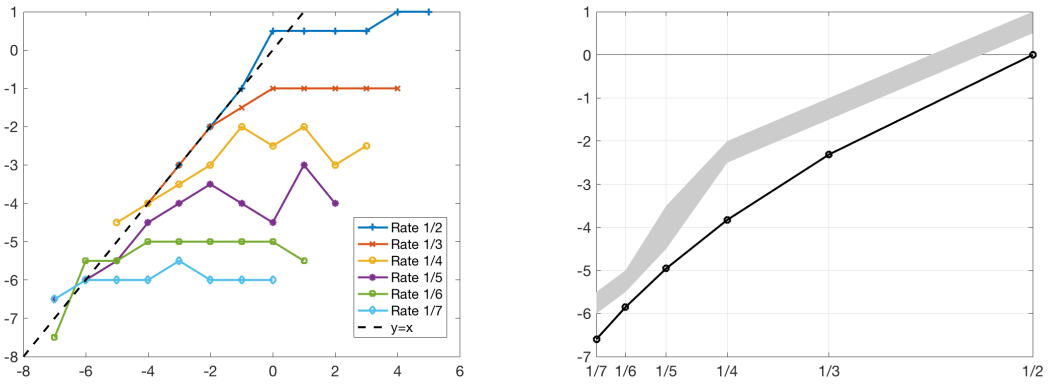


Figure 2.7: (Left) Best training SNR vs. Test SNR (Right) Best training SNR vs. code rate

2.3 Neural Turbo Decoders

2.3.1 Turbo Code

Turbo codes are naturally built out of convolutional codes (both encoder and decoder) and represent some of the most successful codes for the AWGN channel [11]. A corresponding stacking of multiple layers of the neural convolutional decoders leads to a natural neural turbo decoder that we show to match the standard state-of-the-art turbo decoders on the AWGN channel. Unlike the convolutional codes, state-of-the-art (message-passing) decoders for turbo codes are not the corresponding MAP decoders, so there is no contradiction that our neural decoder would beat the message-passing ones.

Turbo encoder is composed of an interleaver and recursive systematic convolutional (RSC) encoders as shown in Figure 2.8, where the interleaver (π) shuffles the input string with a given order; while the deinterleaver (π^{-1}) undos the interleaving in a known order. RSC code with generating function $(1, \frac{f_1(x)}{f_2(x)})$ servers as the encoding block for Turbo code. Two commonly used configurations of RSC are used in this chapter:

- code rate $R = 1/3$, with $f_1(x) = 1 + x^2$ and $f_2(x) = 1 + x + x^2$, which is denoted as turbo-757.

- code rate $R = 1/3$, with $f_1(x) = 1 + x^2 + x^3$ and $f_2(x) = 1 + x + x^3$, which is standard Turbo code used in LTE system, denoted as turbo-LTE.

Code rate $R = 1/3$ is used since both 3GPP LTE and LTE-A systems take $R = 1/3$ as the original code and achieve different code rates with external rate matching.

Notation. A rate 1/3 turbo encoder generates three coded bits (x_1, x_2, x_3) per each message bit. As illustrated in Figure 2.8 (up), among the three coded bits, first output x_1 is the systematic bit, and x_2 is a coded bit generated through an RSC, and x_3 is a coded bit generated through an RSC for the interleaved bit stream. We let (y_1, y_2, y_3) denote the noise corrupted versions of (x_1, x_2, x_3) that the decoder receives.

2.3.2 Turbo Decoders

Turbo decoders are designed under the 'Turbo Principle' [38], which is iteratively refining posterior information by interleaved/de-interleaved received signals with soft-in soft-output (SISO) decoders. SISO decoder takes received signals and prior and produces posterior as prior for later SISO blocks. The general Turbo decoder with rate 1/3 structure is shown in Figure 2.8 (down). Each decoding iteration uses two SISO decoders to decode. The first stage takes de-interleaved posterior p from the last stage as prior and received signal y_1 and y_2 as inputs; while the second stage takes interleaved posterior $\pi(q)$ as prior and received signal $\pi(y_1)$ and y_3. SISO outputs posterior q to be fed to the next stage. At the end of the iterative decoding procedure, decoding is done according to the estimated posterior.

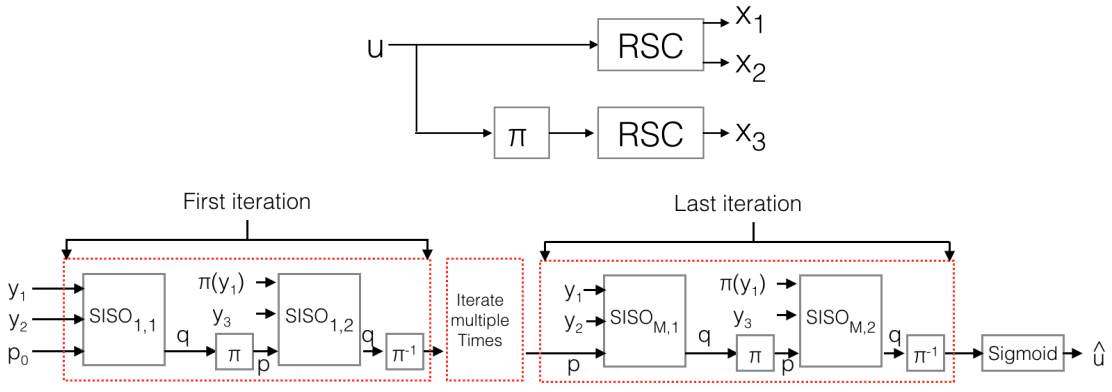


Figure 2.8: Turbo Encoder (up) and Decoder (down)

2.3.3 Design of SISO Algorithm

Different Turbo decoding algorithms differ in the design of the SISO algorithm. We compare the SISO design of the TURBO, NEURALBCJR, and our proposed DEEPTURBO:

*Standard Turbo Decoder (*TURBO*)*

TURBO [11] uses the BCJR algorithm for all the SISO algorithm with extrinsic information and noise weighted systematic bits, cf. Figure 2.9 (left). The extrinsic information takes the difference between the prior and the posterior to be fed to the next stage. TURBO also requires estimating the channel noise variance to compensate noise. The posterior is compensated with extrinsic information and compensated systematic bits before it is sent to the next stage as prior. While TURBO is designed to operate reliably under AWGN settings, it is sensitive to non-AWGN noises as in a non-AWGN setting. For example, a bursty noise corrupted bit leads to severely degraded performance (shown in Figure 2.13), since the iterative decoding scheme propagates corrupted posterior to other code bits via interleaving. Even under AWGN channel, error floor, and error propagation lead TURBO with BCJR algorithm to sub-optimality.

*Iterative Neural BCJR Decoder (*NEURALBCJR*) [61]*

NEURALBCJR replaces the BCJR algorithm with Bidirectional Gated Recurrent Unit (Bi-GRU). The SISO block of NEURALBCJR is shown in Figure 2.9 (right), where it is Initialized by pre-trained Bi-GRU with BCJR input and output to imitate BCJR algorithm followed by an end-to-end training till convergence. All SISO blocks share the same model weights. NEURALBCJR SISO block removes the link of compensating systematic bits and does not require estimating the channel noise to decode. NEURALBCJR avoids producing inaccurate noise weighted systematic bits by implicitly estimating the channel, which improves the robustness against unexpected non-AWGN channels. NEURALBCJR shows matched performance on AWGN channel compared to TURBO under AWGN channels, while it shows better robustness and adaptivity comparing to TURBO with heuristics. There are two major caveats in this design: (1) NEURALBCJR requires to have BCJR knowledge to initialize the SISO block. Without BCJR-initialization, directly training NEURALBCJR from scratch is not stable, and (2) NEURALBCJR simply replaces the BCJR block by weight-sharing

Bi-GRUs, with the same input and output relationship, which limits its potential capacity of NeuralBCJR.

*Deep Turbo Decoder (*DeepTurbo*)*

To ameliorate the caveats of NeuralBCJR, we propose DeepTurbo where each SISO block (cf. Figure 2.9 (right)) still uses Bi-GRU as the building block, while keeping the extrinsic connection as a short-cut for gradient inspired by ResNet [42].

Two major structural differences between NeuralBCJR and DeepTurbo:

- Non-shared weights. Unlike NeuralBCJR uses the same Bi-GRU for all SISO blocks, DeepTurbo doesn't share weight across different iterations, which allows each iteration to deal with posterior differently. Furthermore, non-shared weights improve the training stability.

- More information passed to the next stage. Both Turbo and NeuralBCJR represent the posterior of each code bit by a single value log-likelihood (LLR). A single value for each code bit might not be sufficient to convey enough information. Inspired by resolving calibration issues by ensemble methods [92], for each code bit position, we take length K bits instead of 1 bit. For example, for block length $L = 100$, NeuralBCJR posterior LLR has shape $(L, 1)$, while DeepTurbo transmits a posterior of shape (L, K) to next stage.

A significant advantage is that DeepTurbo does not utilize BCJR knowledge at all, which allows DeepTurbo to learn a better decoding algorithm in a data-driven end-to-end approach. The training dataset is randomly generated for each epoch which avoids overfitting to the fixed training dataset. The hyper-parameters for DeepTurbo decoder are shown in Table 2.1[3].

[3]Further details refer to [51]

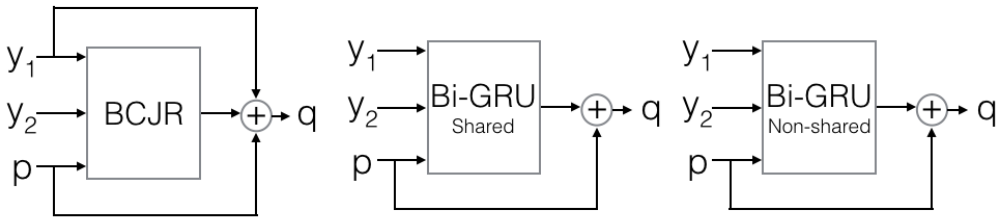


Figure 2.9: Different SISOs: TURBO (left), NEURALBCJR (middle) and DEEPTURBO (right)

DEEPTURBO SISO	2-layer Bi-GRU with 100 units
Learning rate	0.001, decay by 10 when saturate
Num epoch	200
Block length	100/1000
Batch per epoch	100
Optimizer	Adam
Loss	BCE
Train SNR	-1.5dB
Batch size	500
Posterior feature size K	5
Decode iterations M	6

Table 2.1: DEEPTURBO hyperparameters

2.3.4 Performance on AWGN channel

We compare the performance of Deep Turbo Decoder (DEEPTURBO) with the baseline decoders, the standard turbo decoder (TURBO), and Iterative Neural BCJR Decoder (NEURALBCJR).

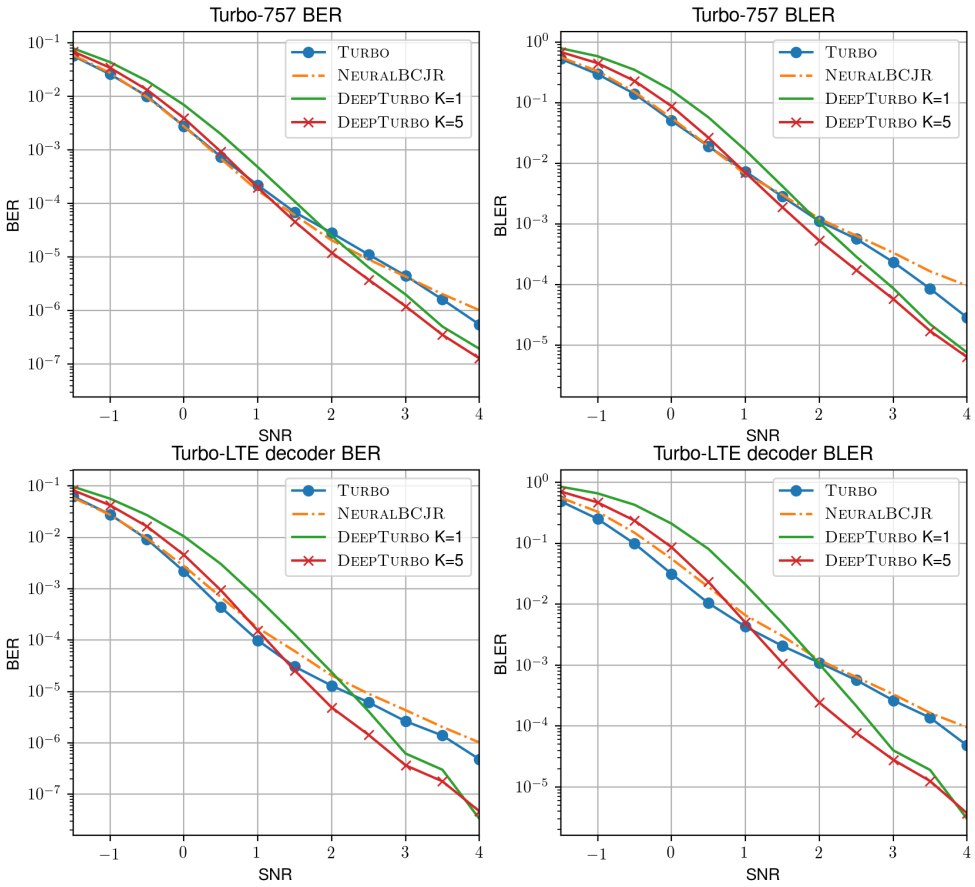


Figure 2.10: The proposed Deep Turbo Decoder (DEEPTURBO) improves upon the standard Turbo decoder in the large SNR regime for Turbo-757 (up) and Turbo-LTE (down)

2.3.5 AWGN with a block length 100

In Figure 2.10, we compare the decoder performances for Turbo codes with turbo-757 and turbo-LTE, trained under block length 100, and decoding iteration 6. NEURALBCJR matches the performance of TURBO as expected. In all scenarios, DEEPTURBO outperforms both TURBO and NEURALBCJR on high SNR cases (SNR $\geq$ 0.5 dB), which implies lowered error floor. To achieve this performance, it is critical to use the appropriate choice of the posterior information dimension K. Empirically we find $K = 5$ trains faster and achieve the best

performance among all $K < 10$. Note that when applying DeepTurbo with $K = 1$, the performance is not better than NeuralBCJR. With limited information flow due to feature size $K = 1$, imitating BCJR is the optimal algorithm. When allowing more information flow with feature size $K = 5$, DeepTurbo learns a better method than BCJR to pass more information.

In Figure 2.11, we compare the decoder performances of DeepTurbo and Turbo with 2 and 6 decoding iterations. Compared to Turbo with 2 decoding iterations, DeepTurbo ($i = 2$) shows significant improvement. This implies that the latent representations at lower layers (iterations) of DeepTurbo extracts the information faster than iteratively applying BCJR. Hence, DeepTurbo can achieve a desired level of accuracy with a smaller number of iterations.

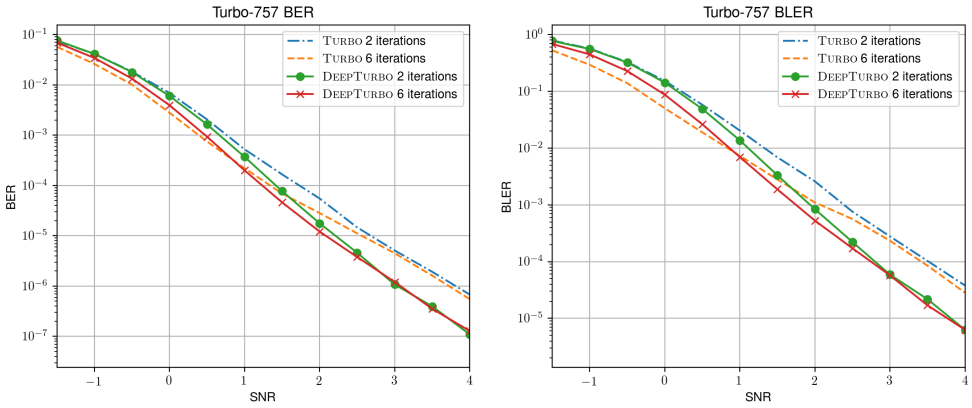


Figure 2.11: An intermediate layer of DeepTurbo already achieves improved performance

2.3.6 Generalization to Longer Block Lengths

Turbo uses the BCJR algorithm, which is independent of the block length. Hence, Turbo is generalizable to any block lengths. On the other hand, the proposed DeepTurbo trained on a short block length ($L = 100$) turbo code does not perform well enough on a larger block length ($L = 1000$) turbo code when applied directly. This indicates that the gain of

DeepTurbo in the high SNR regime is due to customizing the decoder to the specific block length it is trained on. We can partly recover the desired performance on larger block lengths by initializing DeepTurbo with the model trained on shorter block lengths and then further training it for a small number of epochs. In Figure 2.12, we plot the BER and BLER of the DeepTurbo after re-training under block length 1000.

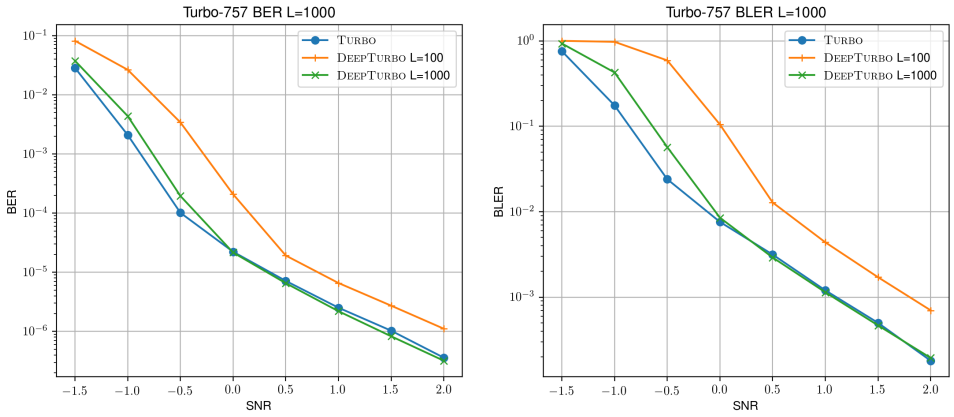


Figure 2.12: Further training of DeepTurbo is required to achieve the desired performance on larger block lengths: BER (left), and BLER (right)

Without relying on the code's mathematical structure (as exploited in BCJR), generalizing to longer block lengths remains a challenging task. Ideally, we want a decoder trained on short blocks, which can be used on longer blocks. This will eliminate the bottleneck of several challenges in training longer blocks. For example, training under a large block length requires a large amount of GPU memory. Hence, we cannot train the decoder with large batch sizes, resulting in unstable training. Furthermore, exploding and diminishing gradient of RNN makes learning unstable. It would be an exciting research direction to design sound decoders that generalize to longer block lengths.

2.4 Neural Decoder's Robustness and Adaptivity

In this section we test DEEPTURBO[4] on the following non-AWGN channels with block length 100:

- Additive T-distribution Noise (ATN) channel: $y = x + z$, where $z \sim T(\nu, \sigma^2)$.
- Radar Channel: $y = x + z + w$. where $z \sim N(0, \sigma_1^2)$ is a background AWGN noise, and $w \sim N(0, \sigma_2^2)$, with probability p is the radar noise with high variance and low probability. $\sigma_1 << \sigma_2$.

NEURALBCJR shows improved robustness and adaptivity compared to existing heuristics [61]. We train DEEPTURBO on ATN and Radar end-to-end. Figure 2.13 shows that DEEPTURBO significantly improves upon NEURALBCJR. This is due to the non-shared parameters of DEEPTURBO, which can perform different decoding functions at different decoding stages. Hence, DEEPTURBO has better adaptivity compared to NEURALBCJR. All neural decoder shows better robustness and adaptivity compared to canonical AWGN-designed decoder.

[4]Neural Convolutional Code Decoder robustness and adaptivity are shown in [61] for more details

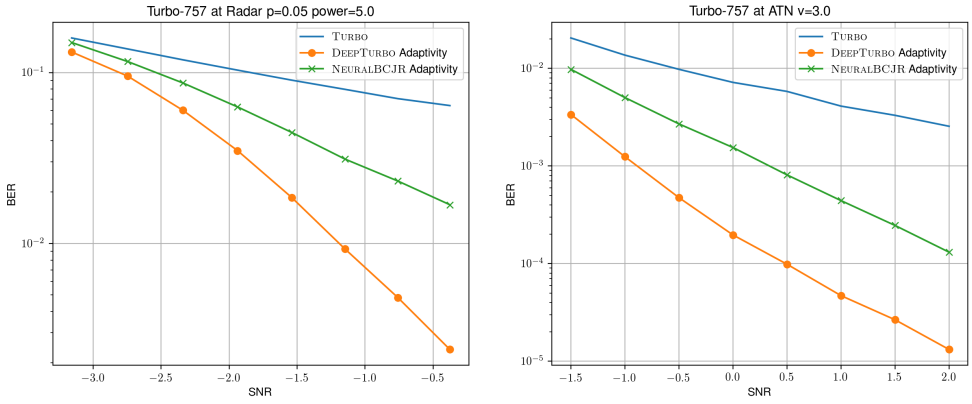


Figure 2.13: DEEPTURBO adapts to non Gaussian channels: Radar channel with $p = 0.05$ and $\sigma_2 = 5.0$ (left), and ATN with $\nu = 3.0$ (right)

Chapter 3

NEURAL CODE

In this chapter [1], we study whether it is possible to **automate** code design via deep learning. Designing codes that combat the noise in a communication medium has remained a significant research area in information theory and wireless communications. Mathematicians have developed asymptotically optimal channel codes for communicating under canonical models after over 60 years of research. On the other hand, in many non-canonical channel settings, optimal codes do not exist, and the codes designed for canonical models are adapted via heuristics to these channels. They are thus not guaranteed to be optimal. In this work, we make significant progress on this problem by designing a fully end-to-end trained neural code, namely, Turbo Autoencoder (TurboAE), with encoder and decoder jointly learned. TurboAE has the following contributions: (a) under moderate block lengths, TurboAE approaches state-of-the-art performance under canonical channels; (b) moreover, TurboAE outperforms the state-of-the-art codes under non-canonical settings in terms of reliability. TurboAE shows that channel coding design development can be automated via deep learning, with near-optimal performance. This chapter were previously published as in [55], [53] and [54].

3.1 Introduction

Autoencoder is a powerful unsupervised learning framework to learn latent representations by minimizing reconstruction loss of the input data [120]. Autoencoders have been widely used in unsupervised learning tasks such as representation learning [120] [67], denoising [80], and generative model [19] [63]. Most autoencoders are under-complete autoencoders, for

[1] The material in this chapter is based on joint work with Hyeji Kim, Himanshu Asnani, Sreeram Kannan, Sewoong Oh, Pramod Viswanath

which the latent space is smaller than the input data [67]. Over-complete autoencoders have latent space larger than input data. While the under-complete autoencoder's goal is to find a low dimensional representation of input data, the over-complete autoencoder's goal is to find a higher-dimensional representation of input data so that from a noisy version of the higher dimensional representation, original data can be reliably recovered. Over-complete autoencoders are used in sparse representation learning [80] [27] and robust representation learning [71].

Channel coding aims at communicating a message over a noisy random channel [109]. As shown in Figure 3.1, the transmitter maps a message to a codeword via adding redundancy (this mapping is called encoding). A channel between the transmitter and the receiver randomly corrupts the codeword. The receiver observes a noisy version to estimate the transmitted message (this process is called decoding). The encoder and the decoder together can be naturally viewed as an over-complete autoencoder, where the noisy channel in the middle corrupts the hidden representation (codeword). Therefore, designing a reliable autoencoder can have a strong bearing on alternative ways of designing new encoding and decoding schemes for wireless communication systems.

Traditionally, the design of communication algorithms first involves designing a 'code' (i.e., the encoder) via optimizing the encoder's specific mathematical properties such as minimum code distance [99]. The associated decoder that minimizes the bit-error-rate is then derived based on the maximum a posteriori (MAP) principle. However, while the optimal MAP decoder is computationally simple for some simple codes (e.g., convolutional codes), the MAP decoder is not computationally efficient for known capacity-achieving codes. Alternative decoding principles such as belief propagation are employed (e.g., for decoding turbo codes). The progress in optimal channel codes with computationally efficient decoders has been quite sporadic due to its reliance on human ingenuity. Since Shannon's seminal work in 1948 [109], it took several decades of research to reach the current state-of-the-art codes [5] finally.

Near-optimal channel codes such as Turbo [11], Low-Density Parity Check (LDPC) [79], and Polar codes [5] show Shannon capacity-approaching [109] performance on AWGN channels,

and they have had a tremendous impact on the Long Term Evolution (LTE) and 5G standards. The traditional approach has the following caveats:

(a) Decoder design heavily relies on handcrafted optimal decoding algorithms for the canonical Additive White Gaussian Noise (AWGN) channels, where the signal is corrupted by i.i.d. Gaussian noise. In practical channels, when the channel deviates from AWGN settings, heuristics are frequently used to compensate for the noise's non-Gaussian properties, which leaves room for the potential improvement in reliability of a decoder [99] [73].

(b) Channel codes are designed for a finite block length K. Channel codes are guaranteed to be optimal only when the block-length approaches infinity and thus are near-optimal in practice only when the block-length is large. On the other hand, under short and moderate block length regimes, there is room for improvement [94].

(c) The encoder designed for the AWGN channel is used across a large family of channels, while the decoder is adapted. This design methodology fails to utilize the flexibility of the encoder.

Related work. Deep learning has pushed the state-of-the-art computer vision performance and natural language processing to a new level far beyond handcrafted algorithms in a data-driven fashion [35]. There also has been a recent movement in applying deep learning to wireless communications. Deep learning based channel decoder design has been studied since [91] [89], where the encoder is fixed as a near-optimal code. It is shown that belief propagation decoders for LDPC and Polar codes can be imitated by neural networks [85] [86] [128] [37] [16]. It is also shown that convolutional code and Turbo codes can be decoded optimally via Recurrent Neural Networks (RNN) [61] and Convolutional Neural Networks (CNN) [51]. Equipping a decoder with a learnable neural network also allows fast adaptation via meta-learning [52]. Recent works also extend deep learning to multiple-input and multiple-output (MIMO) settings [41]. While *neural decoders* show improved performance on various communication channels, there has been limited success in inventing novel codes using this paradigm. Training methods for improving both modulation and channel coding are introduced in [91] [89], where a (7,4) neural code mapping a 4-bit message to a length-7

codeword can match (7,4) Hamming code performance. Current research includes training an encoder and a decoder with noisy feedback [3], improving modulation gain [31], as well as extensions to multi-terminal settings [90]. Joint source-channel coding shows improved results combining source coding (compression) along with channel coding (noise mitigation) [20]. Neural codes were shown to outperform existing state-of-the-art codes on the feedback channel [60]. However, in the canonical setting of the AWGN channel, neural codes are still far from capacity-approaching performance due to the following challenges.

(Challenge A) Encoding with randomness is critical to harvest coding gain on long block lengths [109]. However, existing sequential neural models, both CNN and even RNN can only learn limited local dependency [21]. Hence, the neural encoder cannot sufficiently utilize the benefits of even moderate block length.

(Challenge B) Training neural encoder and decoder jointly (with a random channel in between) introduces optimization issues where the algorithm gets stuck at local optima. Hence, a novel training algorithm is needed.

Contributions. This chapter confronts the above challenges by introducing Turbo Autoencoder (henceforth, TurboAE) – the first channel coding scheme with both encoder and decoder powered by neural networks. TurboAE achieves reliability close to the state-of-the-art channel codes under AWGN channels for a moderate block length.

We demonstrate that channel coding, which has been a focus of study by mathematicians for several decades [99], can be learned in an end-to-end fashion from data alone. Our major contributions are:

- We introduce TurboAE, a neural network based over-complete autoencoder parameterized as Convolutional Neural Networks (CNN) along with interleavers (permutation) and de-interleavers (de-permutation) inspired by turbo codes (Section 3.3). We introduce TurboAE-binary, which binarizes the codewords via straight-through estimator (Section 3.3.1).

- We propose critical training techniques for training TurboAE which includes mechanisms

of alternate training of encoder and decoder and strategies to choose good training examples. Our training methodology ensures stable training of TurboAE without getting trapped at locally optimal encoder-decoder solutions.

- Compared to multiple capacity-approaching codes on AWGN channels, TurboAE shows superior performance in the low to middle SNR range when the block length is of moderate size ($K \sim 100$). To the best of our knowledge, this is the first result demonstrating the deep learning powered discovered neural codes can outperform traditional codes in the canonical AWGN setting (Section 3.4.1).

- On a non-AWGN channel, fine-tuned TurboAE shows significant improvements over state-of-the-art coding schemes due to the flexibility of encoder design, which shows that TurboAE has advantages in designing codes where handcrafted solutions fail (Section 3.4.2).

3.2 Problem Formation

The channel coding problem is illustrated in Figure 3.1 left, which consists of three blocks – an encoder $f_\theta(\cdot)$, a channel $c(\cdot)$, and a decoder $g_\phi(.)$. A channel $c(\cdot)$ randomly corrupts an input x and is represented as a probability transition function $p_{y|x}$. A canonical example of channel $c(\cdot)$ is an identically and independently distributed (i.i.d.) AWGN channel, which generates $y_i = x_i + z_i$ for $z_i \sim N(0, \sigma^2)$, $i = 1, \cdots, K$. The encoder $x = f_\theta(u)$ maps a random binary message sequence $u = (u_1, \cdots, u_K) \in \{0,1\}^K$ of block length K to a codeword $x = (x_1, \cdots, x_N)$ of length N, where x must satisfy either soft power constraint where $E(x) = 0$ and $E(x^2) = 1$, or hard power constraint $x \in \{-1, +1\}$. Code rate is defined as $R = \frac{K}{N}$, where $N > K$. The decoder $g_\phi(y)$ maps a real valued received sequence $y = (y_1, \cdots, y_N) \in \mathcal{R}^N$ to an estimate of the transmitted message sequence $\hat{u} = (\hat{u}_1, \cdots, \hat{u}_K) \in \{0,1\}^K$.

AWGN channel allows closed-form mathematical analysis, which has remained the major playground for channel coding researchers. The noise level is defined as signal-to-noise ratio,

$SNR = -10\log_{10}\sigma^2$. The decoder recovers the original message as $\hat{u} = g_\phi(y)$ using the received signal y.

Channel coding aims to minimize the error rate of recovered message $\hat{u}$. The standard metrics are bit error rate (BER), defined as $BER = \frac{1}{K}\sum_1^K \Pr(\hat{u}_i \neq u_i)$, and block error rate (BLER), defined as $BLER = \Pr(\hat{u} \neq u)$.

While canonical capacity-approaching channel codes work well as block length goes to infinity, they are not guaranteed optimal when the block length is short. We show the benchmarks on block length 100 in Figure 3.1 right with widely-used LDPC, Turbo, Polar, and Tail-bitting Convolutional Code (TBCC), generated with code rate 1/3.

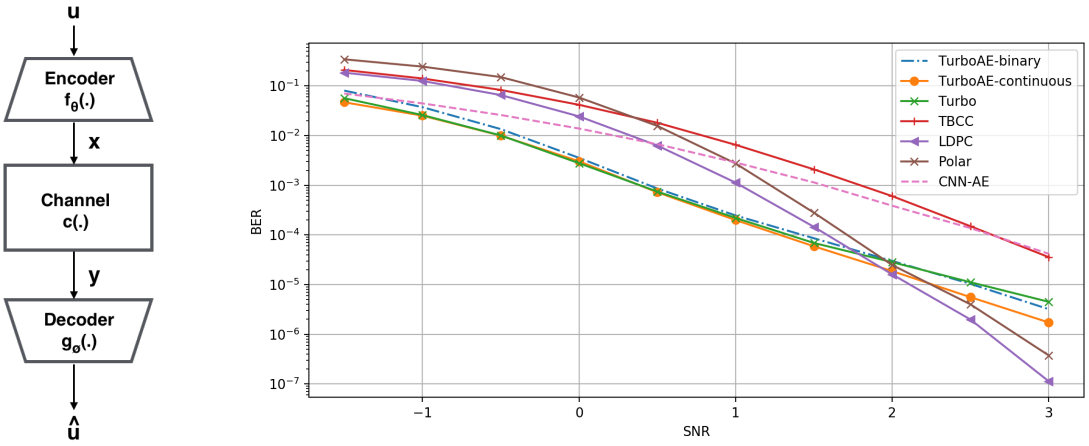


Figure 3.1: Channel coding can be viewed as an over-complete autoencoder with channel in the middle (left). TurboAE performs well under moderate block length in low and middle SNR (right).

Naively applying deep learning models by replacing encoder and decoder with general-purpose neural network does not perform well. Direct applications of the fully-connected neural network (FCNN) cannot scale to a longer block length; the performance of FCNN-AE is even worse than repetition code [55]. Direct applications where both the encoder and the decoder are Convolutional Autoencoder (termed as CNN-AE [133]) shows better performance than TBCC but are far from capacity-approaching codes such as LDPC, Polar, and Turbo.

Bidirectional RNN and LSTM [55] has similar performance as CNN-AE and is not shown in the figure for clarity. Thus neither CNN nor RNN based auto-encoders can directly approach state-of-the-art performance. A key reason for their shortcoming is that they have only local memory. The encoder only remembers information locally. To have high protection against channel noise, it is necessary to have long-term memory.

We propose TurboAE with interleaved encoding and iterative decoding that creates long-term memory in the code and shows a significant improvement compared to CNN-AE. TurboAE has two versions: (a)TurboAE-continuous, which faces soft power constraint (i.e., the total power across a codeword is bounded); and (b) TurboAE-binary, which faces hard power constraint (i.e., each transmitted symbol has a power constraint - and is thus forced to be binary). Both TurboAE-binary and TurboAE-continuous perform comparable or better than all other capacity-approaching codes at a low SNR. At the same time, at a high SNR (over 2 dB with $BER < 10^{-5}$), the performance is only worse than LDPC and Polar code.

3.3 Turbo Autoencoder

Turbo code and turbo principle: Turbo code is the first capacity-approaching code ever designed [11]. Turbo code has two novel components that lead to its success: an interleaved encoder and an iterative decoder. The starting point of the Turbo code is a recursive systematic convolutional (RSC) code that has an optimal decoding algorithm (the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm [8]). A key disadvantage in the RSC code is that it lacks long-range memory (since the convolutional code operates on a sliding window). The key insight of Berrou was to introduce long-range memory by creating two copies of the input bits - the first goes through the RSC code, and the second copy goes through an interleaver (which is a permutation of the bits) before going through the same code. Such a code can be decoded by iteratively alternating between soft-decoding based on the first copy's signal and then using the de-interleaved version as prior to decoding the second copy. The 'Turbo principle' [38] refers to the iterative decoding with successively refining the posterior distribution on the transmitted bits across decoding stages with original and interleaved order.

This code is known to have excellent performance. Inspired by this, we design TurboAE featuring both learnable interleaved encoder and iterative decoder.

Interleaved Encoding Structure: Interleaving is widely used in communication systems to mitigate bursty noise [100]. Formally, interleaver $x^\pi = \pi(x)$ and de-interleaver $x = \pi^{-1}(x^\pi)$ shuffle and shuffle back the input sequence x with the a pseudo random interleaving array known to both encoder and decoder, respectively, as shown in Figure 3.2 left. In the context of Turbo code and TurboAE, the interleaving is not used to mitigate bursty errors (since we are mainly concerned with i.i.d. channels) but rather to add long-range memory in the structure of the code.

We take code rate 1/3 as an example for interleaved encoder f_θ, which consists of three learnable encoding blocks $f_{i,\theta}(.)$ for $i \in \{1, 2, 3\}$, where $f_{i,\theta}(.)$ encodes $b_i = f_\theta(u), i \in \{1, 2\}$ and $b_3 = f_{3,\theta}(\pi(u))$, where b_i is a continuous value. The power constraint of channel coding is enforced via power constraint block $x_i = h(b_i)$.

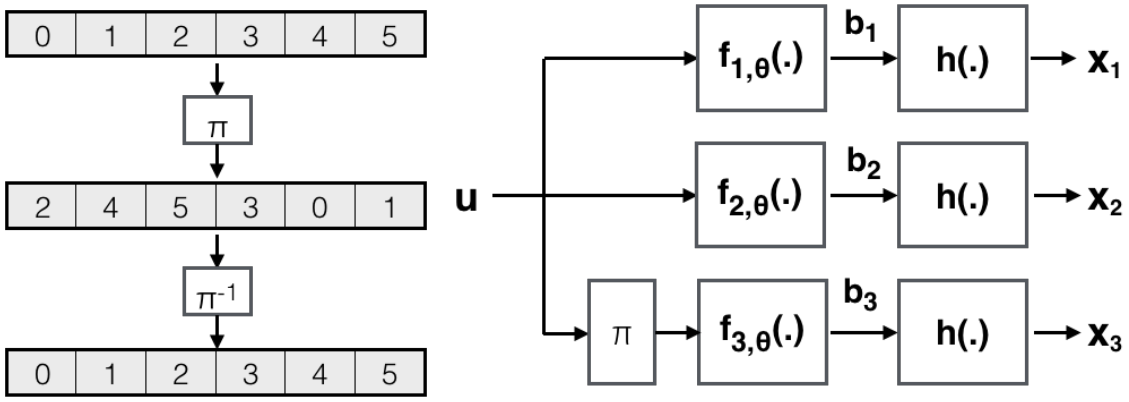


Figure 3.2: Visualization of Interleaver (π) and De-interleaver (π^{-1}) (left); TurboAE encoder on code rate 1/3 (right)

Iterative Decoding Structure: As received codewords are encoded from original message u and interleaved message $\pi(u)$, decoding interleaved code requires iterative decoding on both interleaved and de-interleaved order shown in Figure 3.3. Let y_1, y_2, y_3 denote noisy versions of x_1, x_2, x_3, respectively. The decoder runs multiple iterations, with each iteration

contains two decoders $g_{\phi_{i,1}}$ and $g_{\phi_{i,2}}$ for interleaved and de-interleaved order on the i-th iteration.

The first decoder $g_{\phi_{i,1}}$ takes received signal y_1, y_2 and de-interleaved prior p with shape (K, F), where as F is the information feature size for each code bit, to produce the posterior q with same shape (K, F). The second decoder $g_{\phi_{i,2}}$ takes interleaved signal $\pi(y_1)$, y_3 and interleaved prior p to produce posterior q. The posterior of the previous stage q serves as the prior of the next stage p. The first iteration takes 0 as a prior, and at last iteration the posterior is of shape $(K, 1)$, are decoded as by sigmoid function as $\hat{u} = sigmoid(q)$.

Both encoder and decoder structure can be considered as a parametrization of Turbo code. Once we parametrize the encoder and the decoder, since the encoder, channel, and decoder are differentiable, TurboAE can be trained end-to-end via gradient descent and its variants.

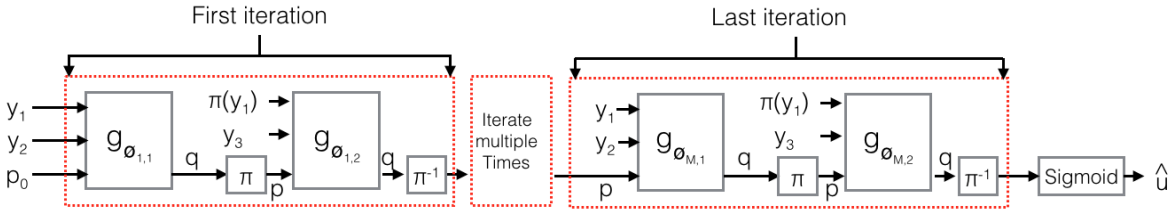


Figure 3.3: TurboAE iterative decoder on code rate 1/3

Encoder and Decoder Design: The space of messages and codewords are exponential (For a length-K binary sequence, there are 2^K distinct messages). Hence, the encoder and decoder must have some structural restrictions to ensure generalization to messages unseen during the training [29]. Applying parameter-sharing sequential neural models such as CNN and RNN are natural parametrization methods for both the encoding and the decoding blocks.

RNN models such as Gated Recurrent Unit (GRU) and Long-Short Term Memory (LSTM) are commonly used for sequential modeling problems [7]. RNN is widely used in deep learning based communications systems [61] [60] [51] [55], as RNN has a natural connection to sequential encoding and decoding algorithms such as convolutional code and BCJR algorithm [61].

However, RNN models are (1) of higher complexity than CNN models, (2) more challenging to train due to gradient explosion, and (3) more challenging to run in parallel [21]. In this chapter, we use one-dimensional CNN (1D-CNN) as the alternative encoding and decoding model. Although the longest dependency length is fixed, 1D-CNN has lower complexity, better trainability [131], and easier implementation in parallel via AI-chips [93]. The learning curve comparison between CNN and RNN is shown in Figure 3.4 left. Training CNN-based model converges faster and more stable than RNN-based GRU model.

Power Constraint Block: The operation of power constraint blocks (i.e., $h(\cdot)$ in $x = h(b)$) depends on the requirement of power constraint.

Soft power constraint normalize the power of code, as $E(x) = 0$ and $E(x^2) = 1$. TurboAE-continuous with soft power constraint allows the code x to be continuous. Addressing the statistical estimation issue given a limited batch size, we use normalization method as:$x_i = \frac{b_i - \mu(b)}{\sigma(b)}$, where $\mu(b) = \frac{1}{K}\sum_{i=1}^{K} b_i$ and $\sigma(b) = \sqrt{\frac{1}{K}\sum_{i=1}^{K}(b_i - \mu(b))^2}$ are scalar mean and standard deviation estimation of the whole block. During the training phase, $\mu(b)$ and $\sigma(b)$ are estimated from the whole batch. On the other hand, during the testing phase, $\mu(b)$ and $\sigma(b)$ are pre-computed with multiple batches. The normalization layer can be also considered as BatchNorm without affine projection, which is critical to stabilize the training of the encoder [103].

3.3.1 Design of TurboAE-binary – Binarization via Straight-Through Estimator

To overcome the differentiablity, multiple algorithms have been applied in the context of binary neural networks. Some wireless communication system requires a hard power constraint, where the encoder output is binary as $x \in \{-1, +1\}$ [118] - so that every symbol has exactly the same power and the information is conveyed in the sign. Hard power constraint is not differentiable, since restricting $x \in \{-1, +1\}$ via $x = \text{sign}(b)$ has zero gradient almost everywhere. We combine normalization and Straight-Through Estimator (STE) [10] [48] to bypass this differentiability issue. STE passes the gradient of $x = \text{sign}(b)$ as $\frac{\partial x}{\partial b} = \mathbb{1}(|b| \leq 1)$ and enables training of an encoder by passing estimated gradients to the encoder, while

enforcing hard power constraint.

Simply training with STE cannot learn a good encoder as shown in Figure 3.4 right. To mitigate the trainability issue, we apply pre-training, which pre-trains TurboAE-continuous firstly, and then add the hard power constraint on top of soft power constraint as $x = \text{sign}(\frac{b-\mu(b)}{\sigma(b)})$, whereas the gradient is estimated via STE. Figure 3.4 right shows that with pre-training, TurboAE-binary reaches Turbo performance within 100 epochs of fine-tuning.

TurboAE-binary is slightly worse than TurboAE-continuous as shown in Figure 3.1, especially at high SNR, since: (a) TurboAE-continuous can be considered as a joint coding, and high order modulation scheme, which has a larger capacity than binary coding at high SNR [118], and (b) STE is an estimated gradient, which makes training encoder noisier and less stable.

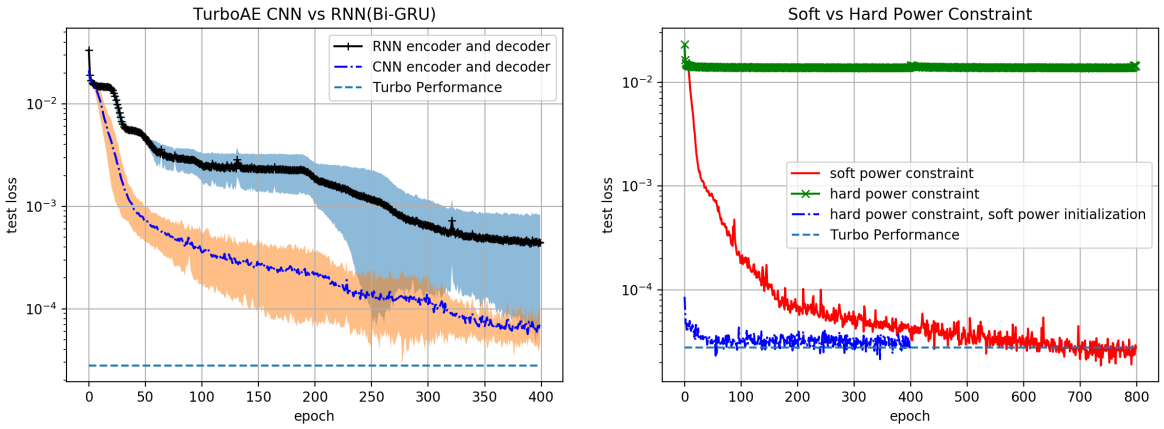


Figure 3.4: Learning Curves on CNN vs GRU: CNN shows faster training convergence (left); Training with STE requires soft-constraint pre-training (right)

3.3.2 Neural Trainability Design

The training algorithms for training TurboAE are shown in Algorithm 1. Compared to the conventional deep learning model training, training TurboAE has the following differences:

Algorithm 1 Training Algorithm for TurboAE

Require: Batch Size B, Train Encoder Steps T_{enc}, Train Decoder Steps T_{dec}, Number of Epoch M Encoder Training SNR σ_{enc}, Decoder Training SNR σ_{dec}

for $i \leq M$ **do**

  for $j \leq T_{enc}$ **do**

    Generate random training example u, and random noise $z \sim N(0, \sigma_{enc})$.

    Train encoder f_θ with decoder fixed, with u and z.

  end for

  for $j \leq T_{dec}$ **do**

    Generate random training example u, and random noise $z \sim N(0, \sigma_{dec})$.

    Train decoder g_ϕ with encoder fixed, with u and z.

  end for

end for

- **Very Large Batch Size** Large batch size is critical to average the channel noise effects. Empirically, TurboAE reaches Turbo performance only when the batch size is greater than 500.

- **Train Encoder and Decoder Separately** We train encoder and decoder separately as shown in Algorithm 1, to avoid getting stuck in local optimum [3] [55].

- **Different Training Noise Level for Encoder and Decoder** Empirically, while it is best to train a decoder at a low training SNR as discussed in [61], it is best to train an encoder at a training SNR that matches the testing SNR, e.g., training encoder at 2dB results in good encoder when testing at 2dB [55]. In this work, we use a random selection of -1.5 to 2 dB for training the decoder and test and train the encoder at the same SNR.

We do a detailed analysis of training algorithms in the supplementary materials. The

hyper-parameters are shown in Table 1.

Loss	Binary Cross-Entropy (BCE)
Encoder	2 layers 1D-CNN, kernel size 5, 100 filters for each $f_{i,\theta}(.)$ block
Decoder	5 layers 1D-CNN, kernel size 5, 100 filters for each $g_{\phi_{i,j}(.)}$ block
Decoder Iterations	6
Info Feature Size F	5
Batch Size	500 when start, double when saturates for 20 epochs, till reaches 2000
Optimizer	Adam with initial learning rate 0.0001
Training Schedule for Each Epoch	Train encoder $T_{enc} = 100$ times, then train decoder $T_{dec} = 500$ times
Block Length K	100
Number of Epochs M	800

Table 3.1: Hyper-parameters of TurboAE

3.4 Experiment Results

3.4.1 Block length coding gain of TurboAE

As block length increases, better reliability can be achieved via channel coding, referred to as *blocklength gain* [11]. We compare TurboAE (only TurboAE-continuous is shown in this section) with the Turbo code and CNN-AE, tested at BER at 2dB on different block lengths, shown in Figure 3.5 left. Both CNN-AE and TurboAE are trained with block length 100, and tested on various block lengths. As the block length increases, CNN-AE shows saturating blocklength gain, while TurboAE and Turbo code reduce the error rate as the block length increases. Naively applying general-purpose neural networks such as CNN to channel coding

cannot gain performance on long block lengths.

Note that TurboAE is still worse than Turbo when the block length is large since long block length requires large memory usage and a more complicated structure to train. Improving TurboAE on very long block length remains open as an interesting future direction.

The BER performance boosted by neural architecture design is shown in Figure 3.5 right. We compare the fine-tuned performance among CNN-AE, TurboAE, and TurboAE without interleaving as $x^{\pi} = \pi(x)$. TurboAE with interleaving significantly outperforms TurboAE without interleaving and CNN-AE.

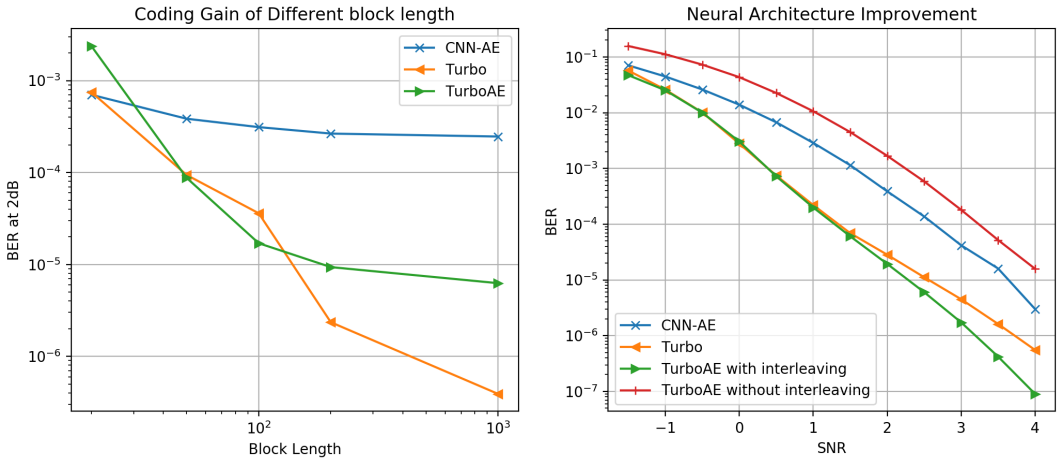


Figure 3.5: Interleaving improves blocklength gain (left); Neural Architecture improves BER performance (right).

3.4.2 Performance on non-AWGN channels

Typically there are no closed-form solutions under non-AWGN and non-iid channels. We compare two benchmarks: (1) canonical Turbo code and (2) DeepTurbo Decoder [51], a neural decoder fine-tuned at the given channel. We test the performance on both iid channels and non-iid channels in settings as follows:

(a) iid Additive T-distribution Noise (ATN) Channel, with $y_i = x_i + z_i$, where iid

$z_i \sim T(\nu, \sigma^2)$ is heavy-tail (tail weight controlled based on the parameter $\nu = 3.0$) T-distribution noise with variance σ^2. The performance is shown in Figure 6.2 left.

(b) non-iid Markovian-AWGN channel is a special AWGN channel with two states, {good, bad}. At bad state, the noise is worse by 1dB than the SNR, and at good state, the noise is better by 1dB than the SNR. The state transition probability between good and bad states are symmetric as $p_{bg} = p_{gb} = 0.8$. The performance is shown in Figure 6.2 right.

For both ATN and Markovian-AWGN channels, DeepTurbo outperforms canonical Turbo code. TurboAE-continuous with learnable encoder outperforms DeepTurbo in both cases. TurboAE-binary outperforms DeepTurbo on the ATN channel. On the Markovian-AWGN channel, TurboAE-binary does not perform better than DeepTurbo at high SNR regimes (but still outperforms canonical Turbo). With the flexibility of designing an encoder, TurboAE designs better code than handcrafted Turbo code for channels without a closed-form mathematical solution.

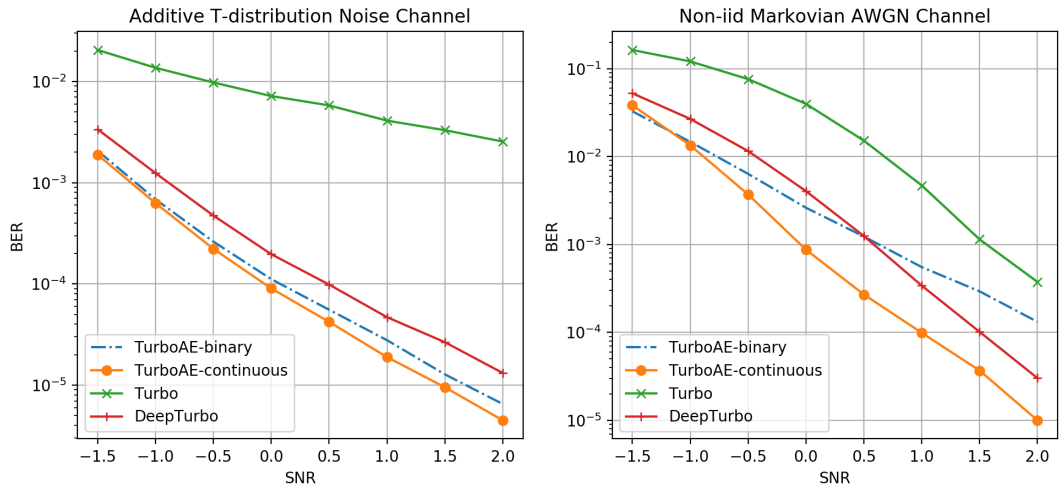


Figure 3.6: TurboAE on iid ATN channel (left) and on-iid Markovian-AWGN channel (right)

3.5 TurboAE Design Analysis

3.5.1 Neural Architecture Design

Supporting code rates beyond 1/3

Previously, only neural code for code rate $R = 1/3$ is shown. The TurboAE encoder and decoder for code rate 1/2 is shown in Figure 3.7. Still designed under 'Turbo principle', TurboAE with code rate 1/2 shows impressive performance under low to moderate SNR, within block length 100. To generate code rates beyond 1/2, we can utilize puncturing.

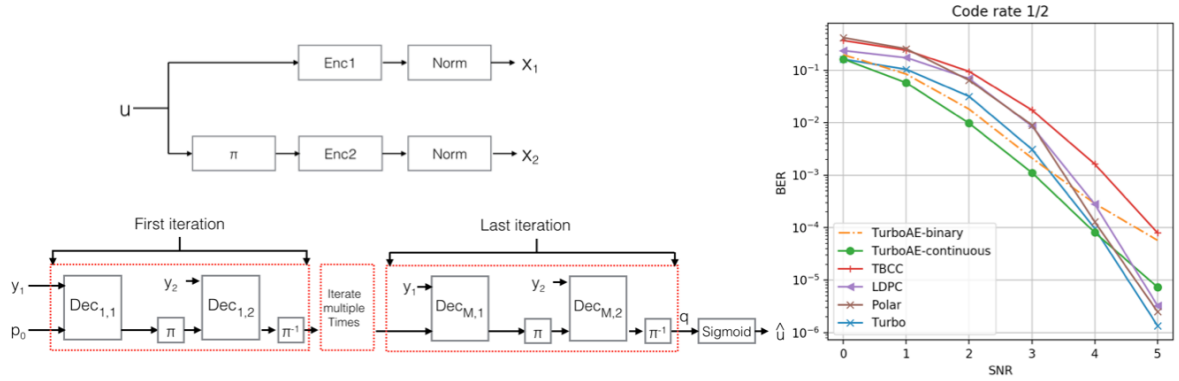


Figure 3.7: The encoder structure (up left), decoder structure (down left), and BER performance (right) of code rate 1/2

CNN with Residual Connection

The same shape property of 1D-CNN is preserved by setting odd kernel size k equals twice the zero-padding length minus one, as shown in Figure 3.8 left. The encoder simply uses 1D-CNN as encoder blocks. In contrast, the decoder uses the residual connection to bypass gradient on iterative decoding procedure to improve trainability [42], and also inspired by extrinsic information from Turbo code [38], shown in Figure 3.8 right. Adding residual connection improves training speed and improves final BER performance [51].

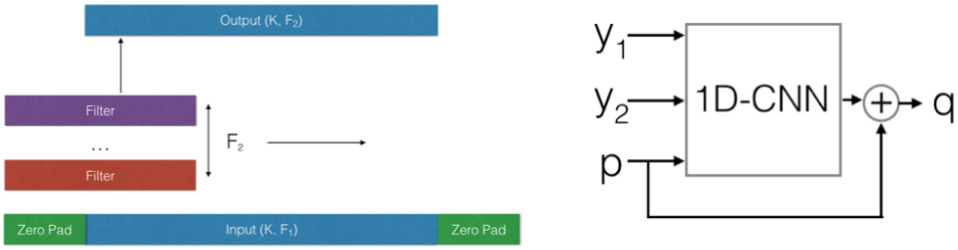


Figure 3.8: 1D CNN visualization on 1 layer (left); CNN with residual connection (right).

Network Size

In figure 3.9 left, we show the test loss trajectory of TurboAE with different network sizes. We keep both encoder and decoder with the same number of filters. A larger network leads to faster training and better performance, with the cost of larger computation and memory usage. We take encoder and decoder with 100 filters, which trains fast given limited computational resources (e.g., training 400 epochs takes 1 day on one Nvidia 1080Ti.)

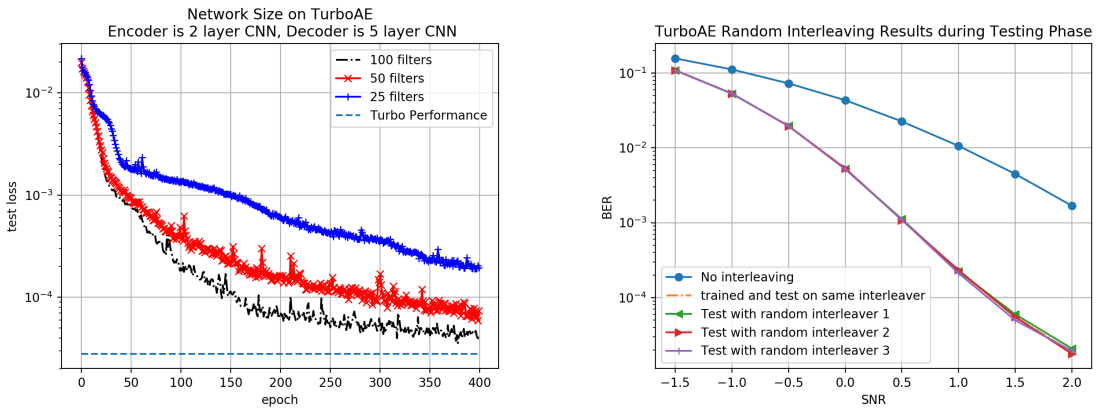


Figure 3.9: Larger Network has better performance (left); Random interleaving array shows same performance (right).

Random Interleaving Array During Testing Phase

Given a fixed pseudo-random interleaving array, one concern is that TurboAE could overfit to a specific interleaving array. When both encoder and decoder change the interleaving array, TurboAE will have a degraded performance. However, empirically, we observe that TurboAE doesn't overfit to the training fixed pseudo-random interleaving array, as shown in Figure 3.9 right. The TurboAE is trained on one specific interleaving array and tested on 3 randomly generated interleaving arrays. For TurboAE, whenever the interleaving array is pseudo-random, the neural encoder and decoder still learn without overfitting.

However, when the interleaving array is not random, e.g., not applying interleaving as $y = \pi(x)$, termed as 'no interleaving', the performance degrades significantly.

3.5.2 Training Algorithms

Joint Training vs Separate Training

Empirically training the encoder and decoder simultaneously is easier to get stuck in the local optimum as shown in Figure 3.10 left. Training encoder and decoder separately is less likely to get stuck in local optimum [3] [55]. Training decoder more times than encoder, on the other hand, makes decoder better approximates optimal decoding algorithm of the encoder, which offers more accurate estimated gradient and stabilizes the training process [55]. We train the encoder and decoder separately, with each epoch trains encoder 100 times and decoder 500 times.

Large Batch Size Improves Training Significantly

Large batch size helps training deep generative models such as Generative Adversarial Networks (GAN) [36] and Variational Autoencoder (VAE) [63], and is also critical to training TurboAE. Figure 3.10 right shows that large batch size leads to significantly lower test BER.

The analysis is on AWGN channel by using the 1st order Taylor expansion on decoder $g_\phi(.)$ as: $\hat{u} = g_\phi(x+z) = g_\phi(x) + z g'_\phi(x) + O(z^2)$.

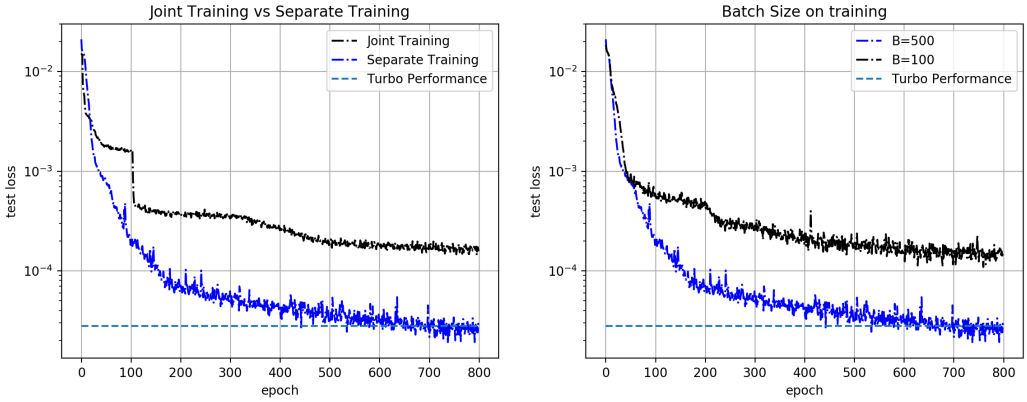


Figure 3.10: Training encoder and decoder jointly gets stuck as local optimum (left). Large batch size improves training (right).

Taking gradient of both sides becomes: $\frac{\partial \hat{u}}{\partial x} \approx g'_\phi(x) + z g''_\phi(x)$ and $\frac{\partial \hat{u}}{\partial \phi} \approx \frac{\partial g_\phi(x)}{\partial \phi} + z \frac{\partial g'_\phi(x)}{\partial \phi}$

AWGN channel has $\frac{\partial y}{\partial x} = 1$ with iid noise. Consider the normalization layer $x = h(b)$, the gradient pass through normalization layer with batch size B is [103]:

$$\frac{\partial x_i}{\partial b_j} = \frac{1}{\sigma(b)}(\mathbb{1}(i=j) - \frac{1}{B}(1 + b_i b_j)) \tag{3.1}$$

Known $\hat{u} = sigmoid(q)$, as $q = g_\phi(h(f_\theta(u)) + z)$, the gradient of BCE loss with respect to logit q is $\frac{\partial BCE(u,\hat{u})}{\partial q} = \hat{u} - u$, the gradient of encoder is:

$$\frac{\partial BCE(u,\hat{u})}{\partial \theta} = \frac{\partial BCE(u,\hat{u})}{\partial q}\frac{\partial q}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial b}\frac{\partial b}{\partial \theta} = (\hat{u} - u)(g'_\phi(x) + z g''_\phi(x))\frac{\partial x}{\partial b}\frac{\partial f_\theta(u)}{\partial \theta} \tag{3.2}$$

The gradient of decoder is:

$$\frac{\partial L}{\partial \phi} = \frac{\partial BCE(u,\hat{u})}{\partial q}\frac{\partial q}{\partial \phi} = (\hat{u} - u)(\frac{\partial g_\phi(x)}{\partial \phi} + z\frac{\partial g'_\phi(x)}{\partial \phi}) \tag{3.3}$$

The benefits of large batch size are as follows:

- **Less noisy gradient for encoder.** The gradient passes through normalization layer is as shown in Equation (3.1). With a large batch size B, the gradient passes through normalization reduces the noise introduced by $\frac{1}{B}(1 + b_i b_j)$, making the gradient passed to the encoder less noisy.

- **Larger batch size reduces gradient noise.** Large batch size makes gradient estimation for both encoder and decoder more accurate, as the error term $zg''_\phi(x)$ in Equation (3.2) and $z\frac{\partial g'_\phi(x)}{\partial \phi}$ in Equation (3.3) can be reduced with large batch size with expectation $E[z] = 0$. Better gradients for both encoder and decoder improve training.

- **More accurate statistics for normalization.** With a larger batch size, the mean and the standard deviation for normalization used in power normalization are more accurate, introducing less noise.

Training SNR

Training noise level (SNR) is a critical parameter for training TurboAE. The training SNR analysis can be derived by the gradient analysis of section 3.5.2. The training noise has a different effect on the encoder and decoder. The training noise affects decoder with noise term $z\frac{\partial g'_\phi(x)}{\partial \phi}$in Equation (3.3). Given a fixed encoder, training decoder with different SNR results in different levels of regularization. For encoder there are two source of noise regularizations: (a) $zg''_\phi(x)$ in Equation (3.2,) and (b) noise introduced by normalization layer in Equation (3.1). Training encoder with different SNR also results in different levels of regularization, which differs from training decoders.

As the effect of decoder training noise has been studied in [61], in this section, we study the training SNR of the encoder, with fixing decoder training SNR to be 0dB as shown in Figure 3.11 left. We see that the most reliable code can be learned when training SNR matches testing SNR. Throughout the chapter, we make encoder training SNR equals to the testing SNR, e.g., we testing TurboAE performance at 2dB, we train TurboAE with encoder SNR at 2dB, and decoder at 0dB. The BER curve shown in the main context is the lower envelope of all curves.

When encoder training SNR is larger than 1dB (e.g., 1dB, 2dB, and 3dB), the BER curves remain nearly the same. Thus encoder training noise level for high SNR region shows diminishing effects on high SNR, which creates an error floor for TurboAE. In the main

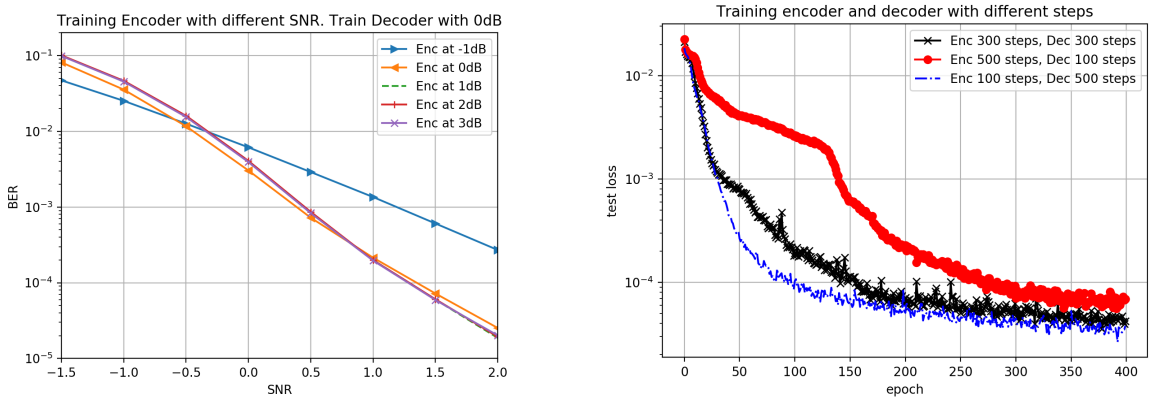


Figure 3.11: Encoder Training SNR has different coding gain effects (left); Training decoder more lead to faster convergence (right).

context, we state that neural codes are suboptimal in high SNR regions since the error is hard to encounter (with probability less than 10^{-4}), making it hard to gather negative examples to train the encoder. Improving high SNR region coding gain with data imbalance is an interesting future research direction.

Train decoder more than encoder

We argue that when the decoder is well-paired to the fixed encoder, the encoder's gradient is more accurate. Training decoder more times will improve performance, as shown in Figure 3.11 right. Training decoder more times leads to faster convergence.

Learning Rate and Batch Size Scheduling

Increasing batch size improves generalization rather than reducing learning rate [35]. To reduce computational expense, we start with batch size $B = 500$, and double the batch size when the test loss saturates for 20 epochs till $B = 2000$, which is our GPU memory limit. Figure 3.4 shows that there exist long 'fake saturating' points where the test loss saturates for over 20 epochs and then continues to drop. When $B = 2000$, when saturates for longer

than 20 epochs, the learning rate lr is reduced by 10 times till the learning rate reaches $lr = 0.000001$.

Block Error Rate Performance comparison

The loss function used is Binary Cross-Entropy (BCE), which minimizes average cross-entropy for all bits along the block, aiming at minimizing BER. Optimizing BER doesn't necessarily result in optimizing block error rate (BLER), as shown in Figure 3.12. TurboAE-binary shows better performance than Turbo code in BER sense under all SNR points, the BLER performance is worse than Turbo code. Improving BLER performance remains an open problem.

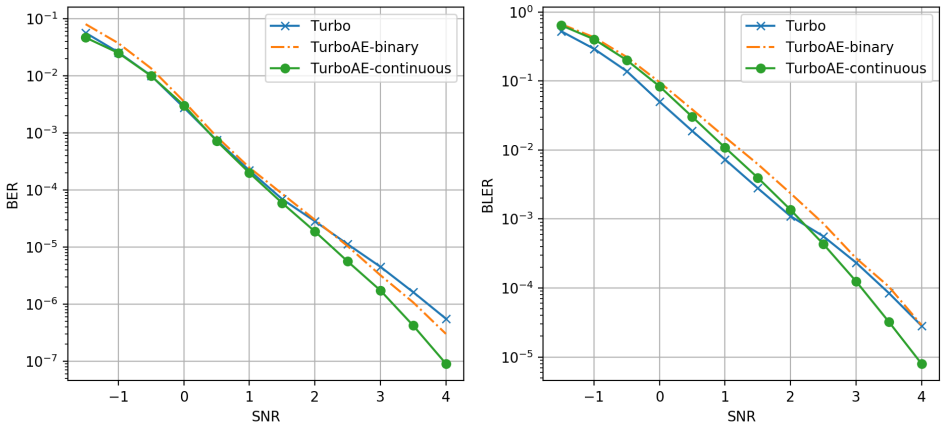


Figure 3.12: TurboAE BER (left) and BLER (right) performance

Chapter 4

NEURAL FEEDBACK CODE

The design of codes for communicating reliably over a statistically well-defined channel is an important endeavor involving in-depth mathematical research and wide-ranging practical applications. This chapter presents the first family of codes obtained via deep learning, which significantly beats state-of-the-art codes designed over several decades of research. The communication channel under consideration is the Gaussian noise channel with feedback, whose study was initiated by Shannon; feedback is known theoretically to improve communication reliability, but no practical feedback codes have ever been successfully constructed. We break this logjam by integrating information-theoretic insights harmoniously with RNN based encoders and decoders to create novel codes that outperform known codes by three orders of magnitude in reliability. We also demonstrate several desirable properties of the codes: (a) generalization to larger block lengths, (b) composability with known codes, (c) adaptation to practical constraints. This result also has broader ramifications for coding theory: even when the channel has an exact mathematical model, deep learning methodologies, when combined with channel-specific information-theoretic insights, can potentially beat state-of-the-art codes constructed over decades of mathematical research. These methods were previously published as in [60][1].

4.1 Problem Settings

The most canonical channel studied in the literature (example: textbook material [24]) and also used in modeling practical scenarios (example: 5G LTE standards) is the AWGN

[1]The material in this chapter is based on joint work with Hyeji Kim, Sreeram Kannan, Sewoong Oh, Pramod Viswanath

channel *without* feedback. Concretely, the encoder takes in K information bits jointly, $\mathbf{b} = (b_1, \cdots, b_K) \in \{0,1\}^K$, and outputs n real valued signals to be transmitted over a noisy channel (sequentially). At the i-th transmission for each $i \in \{1, \ldots, n\}$, a transmitted symbol $x_i \in \mathbb{R}$ is corrupted by an independent Gaussian noise $n_i \sim \mathcal{N}(0, \sigma^2)$, and the decoder receives $y_i = x_i + n_i \in \mathbb{R}$. After receiving the n received symbols, the decoder makes a decision on which information bit sequence $\mathbf{b}$ was sent, out of 2^K possible choices. The goal is to maximize the probability of correctly decoding the received symbols and recover $\mathbf{b}$.

Both the encoder and the decoder are functions, mapping $\mathbf{b} \in \{0,1\}^K$ to $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^n$ to $\hat{\mathbf{b}} \in \{0,1\}^K$, respectively. The design of a good code (an encoder and a corresponding decoder) addresses both (i) the statistical challenge of achieving a small error rate; and (ii) the computational challenge of achieving the desired error rate with efficient encoder and decoder. Almost a century of progress in this domain of coding theory has produced several innovative codes that efficiently achieve small error rate, including convolutional codes, Turbo codes, LDPC codes, and polar codes. These codes are known to perform close to the fundamental limits of reliable communication [94].

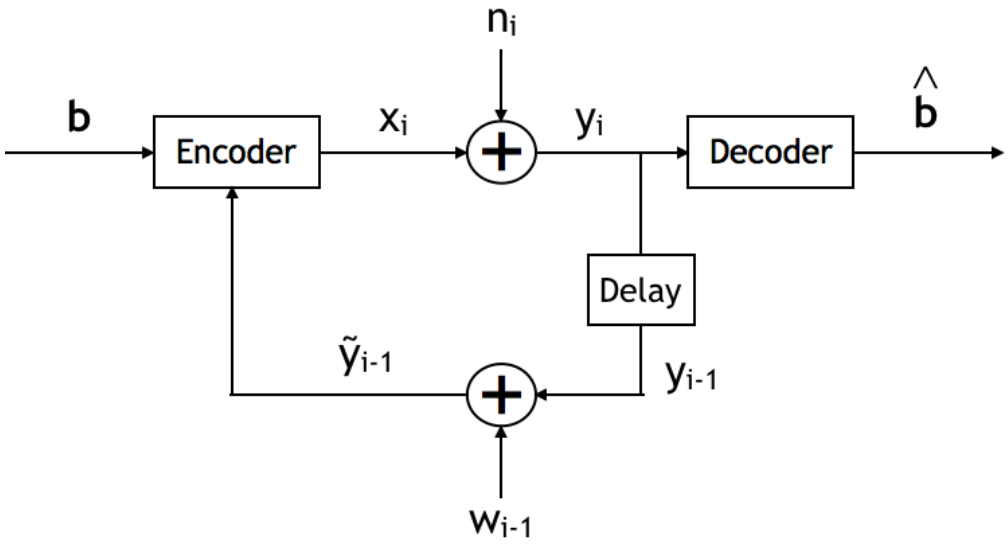


Figure 4.1: AWGN channel with noisy output feedback

In a canonical AWGN channel *with* noisy feedback, the received symbol y_i is transmitted back to the encoder after one unit time of delay and via another additive white Gaussian noise *feedback channel* (Figure 4.1). The encoder can use this feedback symbol to sequentially and adaptively decide what symbol to transmit next. At time i the encoder receives a noisy view of what was received at the receiver (in the past by one unit time), $\tilde{y}_{i-1} = y_{i-1} + w_{i-1} \in \mathbb{R}$, where the noise is independent and distributed as $w_{i-1} \sim \mathcal{N}(0, \sigma_F^2)$. Formally, an *encoder* is now a function that sequentially maps the information bit vector $\mathbf{b}$ and the feedback symbols $\tilde{y}_1^{i-1} = (\tilde{y}_1, \cdots, \tilde{y}_{i-1})$ received thus far to a transmit symbol x_i:

$f_i : \quad (\mathbf{b}, \tilde{y}_1^{i-1}) \mapsto x_i, \quad i \in \{1, \cdots, n\}$ and a *decoder* is a function that maps the received sequence $y_1^n = (y_1, \cdots, y_n)$ into estimated information bits: $g : \quad y_1^n \mapsto \hat{\mathbf{b}} \in \{0,1\}^K$.

The standard measures of performance are the average bit error rate (BER) defined as $\text{BER} \equiv (1/K) \sum_{i=1}^{K} \mathbb{P}(b_i \neq \hat{b}_i)$ and the block error rate (BLER) defined as $\text{BLER} \equiv \mathbb{P}(\mathbf{b} \neq \hat{\mathbf{b}})$,

where the randomness comes from the forward and feedback channels and any other sources of randomness that might be used in the encoding and decoding processes. It is standard (both theoretically and practically) to have an average power constraint, i.e., $(1/n)\mathbb{E}[\|\mathbf{x}\|^2] \leq 1$, where $\mathbf{x} = (x_1, \cdots, x_n)$ and the expectation is over the randomness in choosing the information bits $\mathbf{b}$ uniformly at random, the randomness in the noisy feedback symbols $(\tilde{y}_1, \cdots, \tilde{y}_n)$ and any other randomness used in the encoder.

4.1.1 Results preview

While the capacity of the channel remains the same in the presence of feedback [108], the reliability can increase significantly, as demonstrated by the celebrated result of Schalkwijk and Kailath (S-K) [106]. Although the S-K scheme meets the optimal theoretical performance, critical drawbacks make it fragile. Theoretically, the scheme critically relies on exactly noiseless feedback (i.e. $\sigma_F^2 = 0$) and does not extend to channels with an even arbitrarily small amount of noise in the feedback (i.e. $\sigma_F^2 > 0$). The scheme is also very sensitive to numerical precisions; we see this in Figure 4.2, where the numerical errors dominate the performance of the S-K scheme, with a practical choice of MATLAB implementation with a precision of 64

bits to represent floating-point numbers.

Even with a noiseless feedback channel with $\sigma_F^2 = 0$, which the S-K scheme is designed for, it is outperformed significantly by our proposed Deepcode (described in detail in Section 4.2). At moderate SNR of 2 dB, Deepcode can outperform S-K scheme by three orders of magnitude in BER. In Figure 4.2 (top), the resulting BER is shown as a function of the Signal-to-Noise Ratio (SNR) defined as $-10\log_{10}\sigma^2$, where we consider the setting of rate 1/3 and information block length of $K = 50$ (hence, $n = 150$). Also shown as a baseline is an LTE turbo code which does not use any feedback. Deepcode exploits the feedback symbols to achieve a significant gain of two orders of magnitude consistently over the Turbo code for all SNR. In Figure 4.2 (bottom), BLER of Deepcode is shown as a function of the Signal-to-Noise Ratio (SNR), together with state-of-the-art polar, LDPC, and convolutional codes in a 3GPP document for 5G.

Deepcode significantly improves overall state-of-the-art codes of the similar block-length and the same rate. Also plotted as a baseline are the theoretically estimated performance of the best code with no efficient decoding schemes. This impractical baseline lies between approximate achievable BLER (labeled Normapx in the figure) and a converse to the BLER (labeled Converse in the figure) from [94]. We note that there are schemes proposed more recently that address the sensitivity to noise in the output feedback, a major drawback of the S-K scheme (e.g., [62] and [17]). However, these schemes either still suffer from similar sensitivity to numerical precisions at the decoder due to the uniform message constellation as in the S-K scheme [62], or are often incapable of exploiting the feedback information [17] as we illustrate in Figure 4.12 in experiments with noisy feedback.

4.2 neural encoder and decoder

A natural strategy to create a feedback code is to utilize a recurrent neural network (RNN) as an encoder since (*i*) communication with feedback is naturally a sequential process and (*ii*) we can exploit the sequential structure for efficient decoding. We propose representing

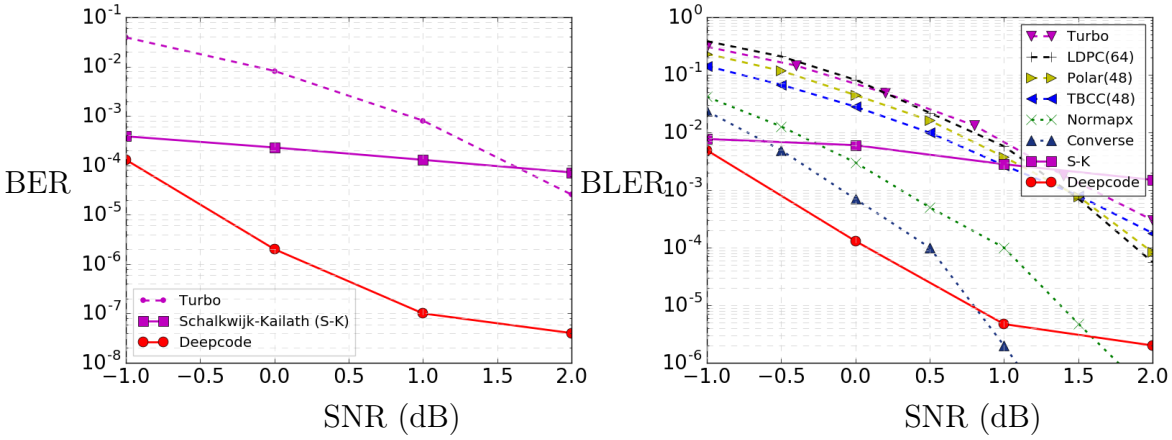


Figure 4.2: Deepcode significantly outperforms the baseline of S-K and Turbo code when information block length is 50 and noiseless feedback is available in BER (left) and BLER (right). Deepcode also outperforms all state-of-the art codes (without feedback) in BLER (right).

the encoder and the decoder as RNNs, training them jointly under AWGN channels with noisy feedback, and minimizing the error in decoding the information bits. However, in our experiments, we find that this strategy by itself is insufficient to achieve any performance improvement with feedback; several design elements need to be carefully chosen in constructing and training an RNN based code.

We exploit information-theoretic insights to enable improved performance by considering the coding scheme for erasure channels with feedback: here transmitted bits are either received perfectly or erased, and whether the previous bit was erased or received perfectly is fed back to the transmitter. In such a channel, the following two-phase scheme can be used: transmit a block of symbols and then transmit whichever symbols were erased in the first block (and ad infinitum). This motivates a two-phase scheme, where uncoded bits are sent in the first phase, and then based on the feedback in the first phase, coded bits are sent in the second phase; thus, the code only needs to be designed for the second phase. In this section, we show that these intuitions can be critically employed to innovate neural network architectures for coding on AWGN channels with feedback. Even within this two-phase paradigm, several

architectural choices need to be made. In the following, we show the baseline neural network architectures (Scheme A) and a series of improvements (Schemes B,C,D) made based on the typical error analysis.

Our experiments focus on the setting of rate $1/3$ and information block length of 50 for concreteness That is, the encoder maps $K = 50$ message bits to a codeword of length $n = 150$. We discuss generalizations to longer block lengths in Section 4.3.

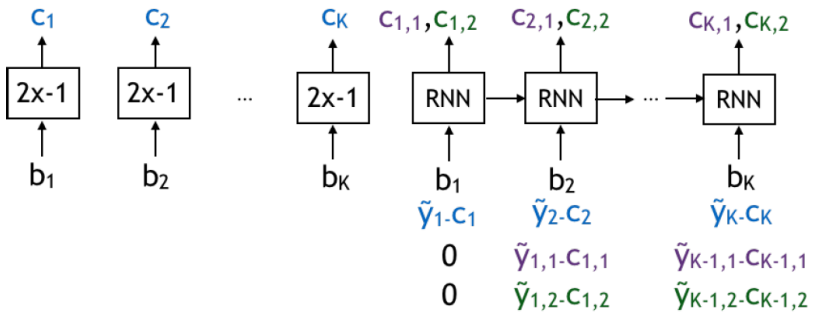
Scheme A. RNN based feedback encoder/decoder (RNN (linear) and RNN (tanh)).

We propose a baseline encoding scheme that progresses in two phases. In the first phase, the K raw information bits are sent (uncoded) over the AWGN channel. In the second phase, $2K$ coded bits are generated based on the information bits $\mathbf{b}$ and (delayed) output feedback and sequentially transmitted (so that the total rate is fixed as $1/3$).

Encoding. The architecture of the encoder is shown in Figure 4.3. The architectures for RNN (tanh) and RNN (linear) feedback codes are equivalent except the activation function in RNN; RNN (tanh) encoder uses a tanh activation while RNN (linear) encoder uses a linear activation (for both the recurrent and output activation).

In the first phase of the encoding process, the encoder simply transmits the K raw message bits via binary phase shift keying (BPSK). That is, the encoder maps b_k to $c_k = 2b_k - 1$ for $k \in \{1, \cdots, K\}$, and stores the feedback $\tilde{y}_1, \cdots, \tilde{y}_K$ for later use. In the second phase, the encoder generates a coded sequence of length $2K$ (length $(1/r - 1)K$ for general rate r code) through a single directional RNN. In particular, each k-th RNN cell generates two coded bits $c_{k,1}, c_{k,2}$ for $k \in \{1, \ldots, K\}$, which uses both the information bits and (delayed) output feedback from the earlier raw information bit transmissions. The input to the k-th RNN cell is of size four: b_k, $\tilde{y}_k - c_k$ (the estimated noise added to the k-th message bit in phase 1) and the most recent two noisy feedbacks from phase 2: $\tilde{y}_{k-1,1} - c_{k-1,1}$ and $\tilde{y}_{k-1,2} - c_{k-1,2}$. Note that we use $\tilde{y}_{k,j} = c_{k,j} + n_{k,j} + w_{k,j}$ to denote the feedback received from the transmission of $c_{k,j}$ for $k \in \{1, \cdots, K\}$ and $j \in \{1, 2\}$, and $n_{k,j}$ and $w_{k,j}$ are corresponding forward and

feedback channel noises, respectively.



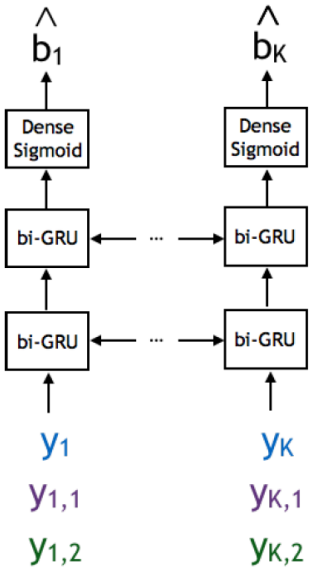
Layer	Output dimension
Input	(K, 4)
RNN (linear or tanh)	(K, 50)
Dense (sigmoid)	(K, 2)
Normalization	(K, 2)

Figure 4.3: RNN encoder for Scheme A.

To generate codewords that satisfy the power constraint, we put a normalization layer to the RNN outputs so that each coded bit has a mean 0 and a variance 1. During training, the normalization layer subtracts the batch mean from the output of RNN and divide by the standard deviation of the batch. After training, we compute the mean and the variance of the RNN outputs over 10^6 examples. In testing, we use the precomputed means and variances.

Decoding. We propose a decoding scheme using two layers of bidirectional Gated Recurrent Units (GRU). The architecture of the decoder is shown in Figure 4.4. Based on the received sequence $\mathbf{y} = (y_1, \cdots, y_k, y_{1,1}, y_{1,2}, y_{2,1}, y_{2,2}, \cdots, y_{K,1}, y_{K,2})$ of length $3K$, the decoder estimates K information bits. For the decoder, we use a two-layered bidirectional Gated Recurrent Unit (GRU), where the input to the k-th GRU cell is a tuple of three received symbols, $(y_k, y_{k,1}, y_{k,2})$.

Training. As illustrated in Figure 4.5, both the encoder and decoder are trained *jointly* as in



Layer	Output dim.
Input	(K, 3)
bi-GRU	(K, 100)
Batch Norm.	(K, 100)
bi-GRU	(K, 100)
Batch Norm.	(K, 100)
Dense (sigmoid)	(K, 1)

Figure 4.4: RNN decoder for Scheme A.

the autoencoder training. We model the whole communication system including the encoder and the channels and the decoder as a large neural network, where the input is a random sequence of message bits and the output is the estimate of the message bit sequence.

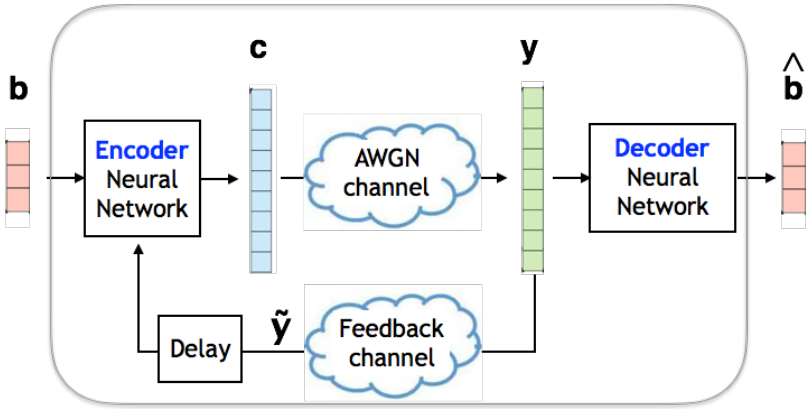


Figure 4.5: Autoencoder framework for the joint training of encoder–decoder (illustrated for information block length 3, rate 1/3)

For training examples, we generate a random message bit sequence $\mathbf{b} = (b_1, \cdots, b_K)$

and a random noise sequence for the forward channel (and a random noise sequence for the feedback channel if we consider an AWGN feedback channel). We train the encoder and the decoder jointly via backpropagation through time (on the entire input sequence), where the goal of training is to minimize the binary cross-entropy loss function $\mathcal{L}(\mathbf{b}, \hat{\mathbf{b}}) = \sum_{i=1}^{K}(-b_i \log \hat{b}_i - (1-b_i)\log(1-\hat{b}_i))$. We do the backpropagation through time over a 4×10^6 examples via an Adam optimizer (β_1=0.9, β_2=0.999). We fix the batch size as 200. We randomly initialize weights of the encoder and the decoder. We observe that training with a random initialization of the encoder and the decoder gives a better code compared to initializing with a pre-trained encoder/decoder by sequential channel codes for non-feedback AWGN channels (e.g. convolutional codes). We also use a decaying learning rate and gradient clipping; we reduce the learning rate by 10 times after training with 10^6 examples, starting from 0.02. Gradients are clipped to 1 if L_2 norm of the gradient exceeds 1 so that we prevent the gradients from getting too large. We do not use any dropout. We find that the choices of training examples are important. Empirically we find that if the length of the training input sequence is too small (e.g., 50), we cannot learn a good structured code. As we use the RNN based encoder and decoder, the learned code generalizes to arbitrary block lengths (e.g., as opposed to a feedforward neural network which only applies to a fixed input length). We set the length of the training input sequence to 100 (and test with input sequence length 50).

In generating two sets of noise sequence for the AWGN channels used during the training, we find that it works best to set the SNRs equal to the SNRs to be used in testing. For example, if we would like to learn a code to be used under the 1dB forward channel with 1dB feedback channel, it is best to train with noise sequences generated under those SNRs.

Result.

When jointly trained, as shown in Figure 4.6, a linear RNN encoder achieves performance close to Turbo code that does not use the feedback information at all (To generate plots in Figure 4.6, we take an average bit error rate over 10^8 bits for SNR $= -1, 0$dB and 10^9 bits for SNR$= 1, 2$dB.) With a *non-linear* activation function of $\tanh(\cdot)$, the performance improves, achieving BER close to the existing S-K scheme. Such a gain of non-linear codes

over linear ones is in-line with theory [62]. In order to further improve the reliability, we perform typical error analysis and propose modifications to the RNN encoder and decoder architectures (Schemes B,C,D). The improved reliabilities of modified architectures are also shown in Figure 4.6.

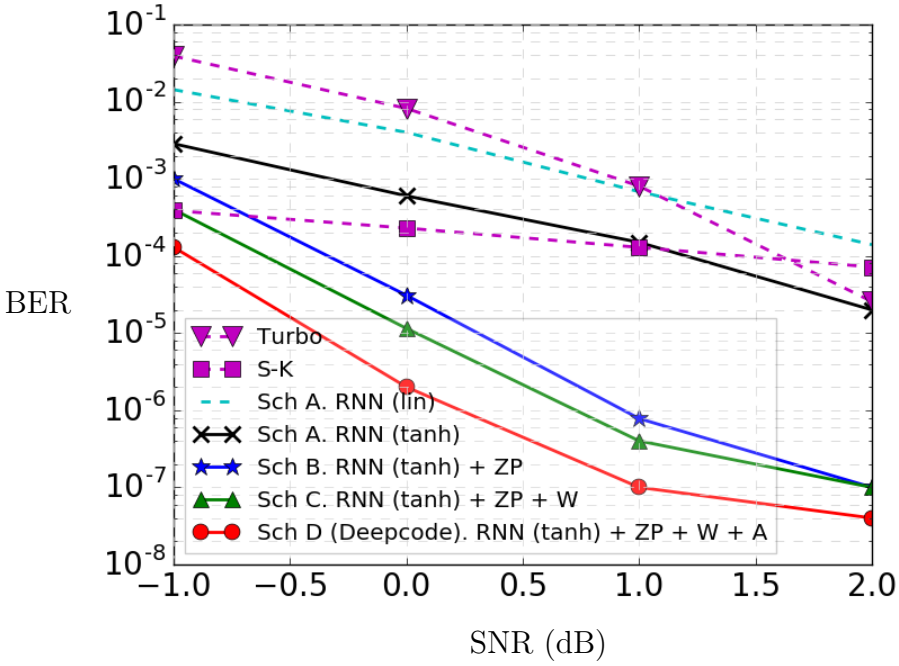


Figure 4.6: Building upon a simple linear RNN encoder (Figure 4.3), we progressively improve the architecture. Eventually with RNN(tanh)+ZP+W+A architecture formally described in Section 4.2, we significantly outperform the baseline of S-K scheme and Turbo code, by several orders of magnitude in the bit error rate, when information block length is 50 and noiseless feedback is available ($\sigma_F^2 = 0$ and forward channel is AWGN).

Typical error analysis. Due to the recurrent structure in generating coded bits $(c_{k,1}, c_{k,2})$, the coded bit stream carries more information on the first few bits than last few bits (e.g. b_1 than b_K). This results in more errors in the last information bits, as shown in Figure 4.7, where we plot the average BER of b_k for $k = \{1, \cdots, K\}$. In the following, we propose modifications to resolve this issue.

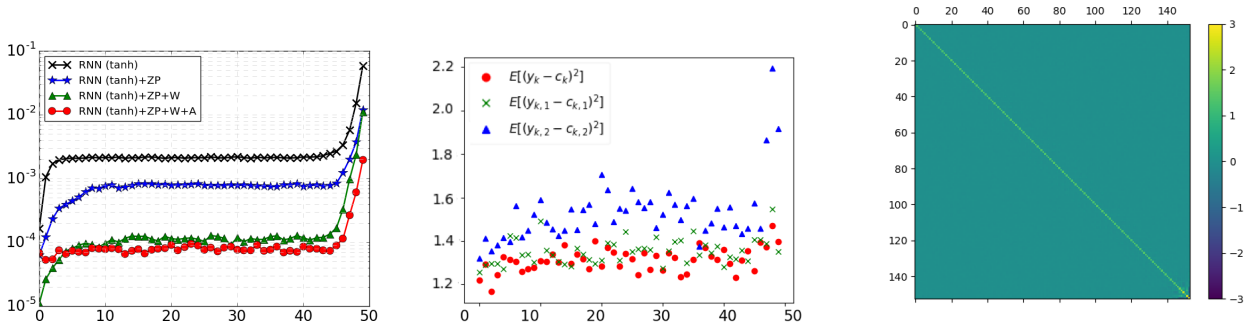


Figure 4.7: (Left) A naive RNN(tanh) code gives a high BER in the last few information bits. With the idea of zero padding and power allocation, the RNN(tanh)+ZP+W+A architecture gives a BER that varies less across the bit position, and overall BER is significantly improved over the naive RNN(tanh) code. (Middle) Noise variances across bit position which result in a block error: high noise variance on the second parity bit stream $(c_{1,2}, \cdots, c_{K,2})$ causes a block error. (Right) Noise covariance: Noise sequence which results in a block error does not have a significant correlation across position.

Scheme B. RNN feedback code with zero padding (RNN (tanh) + ZP).

In order to reduce high errors in the last information bits, as shown in Figure 4.7, we apply the zero padding (ZP) technique; we pad a zero in the end of information bits, and transmit a codeword for the padded information bits.

Encoding and decoding. The encoder and decoder structures with zero padding are shown in Figure 4.8 and Figure 4.9, respectively. We maintain the encoder and decoder architecture same as Scheme A (RNN (tanh)) and simply replace the input information bits by information bits padded by a zero; hence, we use $K + 1$ RNN cells in Phase 2 instead of K.

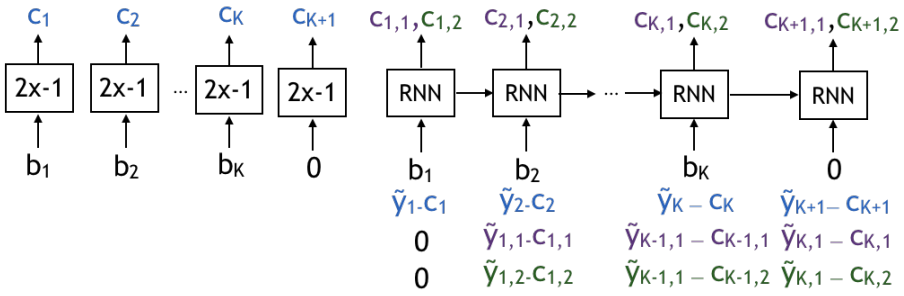


Figure 4.8: Encoder for Scheme B.

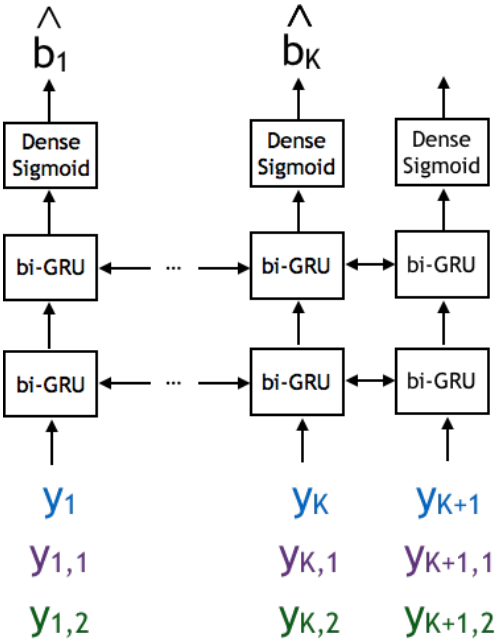


Figure 4.9: Decoder for Schemes B,C,D.

Training. As in training Scheme A, we use back-propagation with binary cross entropy loss. We measure binary crossentropy loss on the information bits of length K only (i.e., ignore the loss on the last bit which corresponds to a zero padding).

Result. By applying zero padding, the BER of the last information bits, as well as other bits, drops significantly, as shown in Figure 4.7. Zero padding requires a few extra channel

usages (e.g. with one zero padding, we map 50 information bits to a codeword of length 153. Actual transmission requires 152 channel usages because the padded zero itself does not need to be transmitted.). However, due to the significant improvement in BER, it is widely used in sequential codes (e.g. convolutional codes and turbo codes).

Typical error analysis. To see if there is a pattern in the noise sequence which makes the decoder fail, we simulate instances of the code and the channel (noise sequence) and collect the corresponding decoding results (whether each instance is decoded correct or wrong). We then look at the first and second order noise statistics which result in the decoding error. In Figure 4.7 (Middle), we plot the average variance of noise added to b_k in first phase and $c_{k,1}$ and $c_{k,2}$ in the second phase, as a function of k, which results in the (block) error in decoding. From the figure, we make two observations; (i) large noise in the last bits causes an error, and (ii) large noise in $c_{k,2}$ is likely to cause an error, which implies that the raw bit stream and the coded bit streams are not equally robust to the noise – an observation that will be exploited next. In Figure 4.7 (Right), we plot noise covariances that result in a decoding error. From Figure 4.7 (Right), we see that there is no particular correlation within the noise sequence that makes the decoder fail.

Scheme C. RNN feedback code with power allocation (RNN(tanh) + ZP + W).

Based on the observation that the raw bit c_k and coded bits $c_{k,1}, c_{k,2}$ are not equally robust, as shown in Figure 4.7 (Middle), we introduce trainable weights which allow allocating different amount of power to the raw bit stream and coded bit streams.

Encoding and decoding. The encoder and decoder architectures for scheme C are shown in Figure 4.10 and Figure 4.9, respectively. Specifically, we introduce three trainable weights (w_0, w_1, w_2) which represent the power allocated to $c_k, c_{k,1}, c_{k,2}$ for all $k \in \{1, \cdots, K\}$, respectively. The weights (w_0, w_1, w_2) satisfies $w_0^2 + w_1^2 + w_2^2 = 3$ so that the average power is preserved (c.f. in Encoder B, we let $\mathbb{E}[c_k^2] = \mathbb{E}[c_{k,1}^2] = \mathbb{E}[c_{k,2}^2] = 1$ and $w_1 = w_2 = w_3 = 1$).

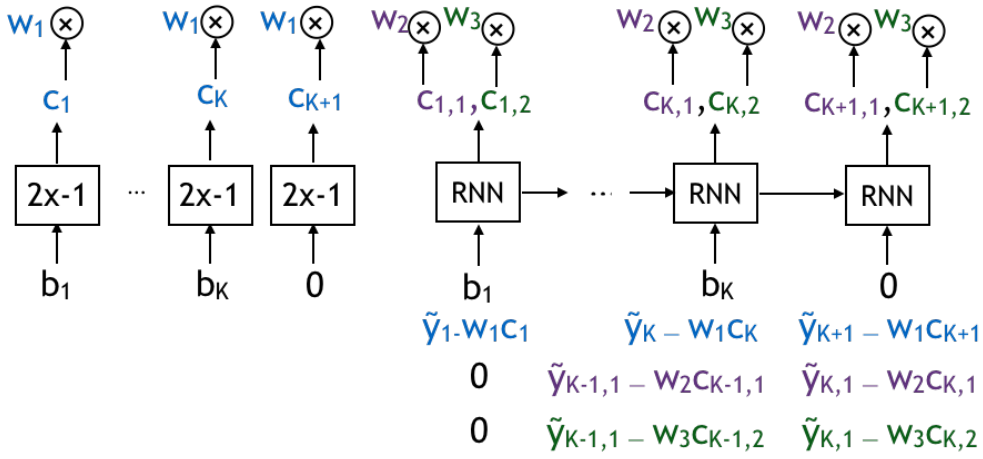


Figure 4.10: Encoder for Scheme C.

Training. We initialize w_is by 1 and train the encoder and decoder jointly as we trained Schemes A and B. The trained weights are $(w_1, w_2, w_3) = (1.13, 0.90, 0.96)$ (trained at -1dB). This implies that the encoder uses more power in Phase I, to transmit (raw) information bits. In Phase II, the encoder uses more power on the second parity bits than in the first parity bits.

result and typical error analysis. By introducing and training these weights, we achieve the improvement in BER as shown in Figures 4.6 and 4.7. While the average BER is improved by about an order of magnitude for most bit positions as shown in Figure 4.7 (Left), the BER of the last bit remains about the same. On the other hand, the BER of first few bits are now smaller, suggesting the following bit-specific power allocation method.

Scheme D. Deepcode: RNN feedback code with bit power allocation (RNN(tanh) + ZP + W + A).

One way to resolve the unbalanced error according to bit position is to use power allocation. Ideally, we would like to reduce the power for the first information bits and increase the power for the last information bits so that we help transmission of last few information bits more than first information bits. However, it is not clear how much power to allow for the

first few information bits and the last few information bits. Hence, we introduce a weight vector allowing the power of bits in different position to be different.

Encoding and decoding. The encoder and decoder architectures for scheme D are shown in Figure 4.11 and Figure 4.9, respectively. We introduce trainable weights $a_1, a_2, \cdots, a_K, a_{K+1}$ for power allocation in each transmission. To the full generality, we can train all these $K+1$ weights. However, we let $a_5, \cdots, a_{K-4} = 1$ and only train first 4 weights and the last 5 weights, a_1, a_2, a_3, a_4 and $a_{K-3}, a_{K-2}, a_{K-1}, a_K, a_{K+1}$, for two reasons. Firstly, this way we can generalize the encoder to longer block lengths by maintaining the weights for first four and last five weights and fixing the rest of weights as 1s, no matter how many rest weights we have. For example, if we test our code for length 1000 information bits, we can let $a_5, \cdots, c_{996} = 1$. Secondly, the BERs of middle bits do not depend much on the bit position; hence, power control is not as much needed as the first and last few bits.

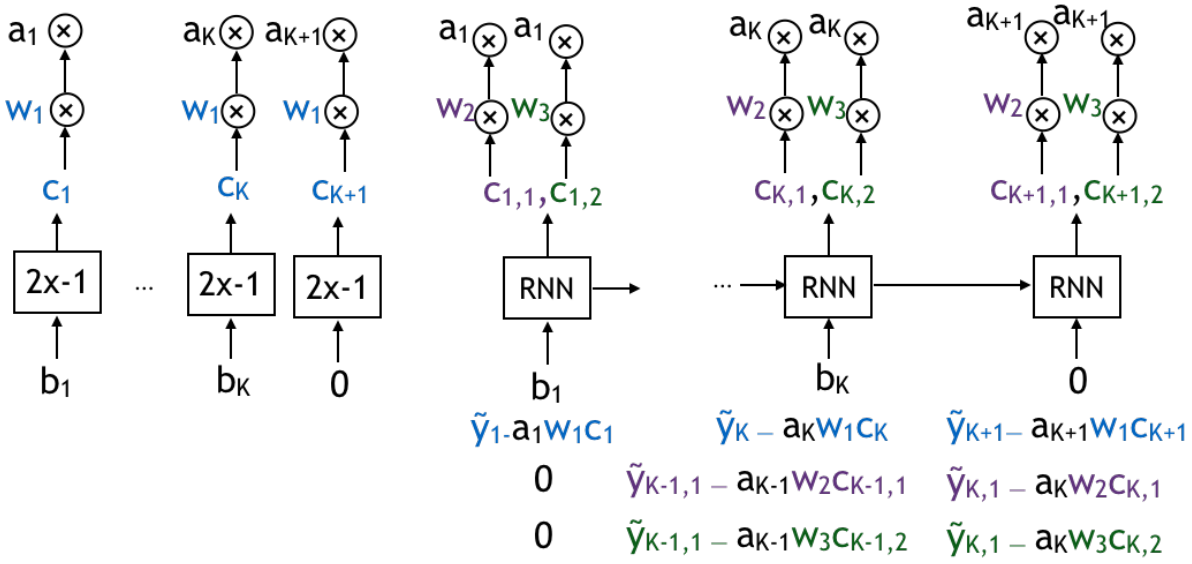


Figure 4.11: Encoder for Scheme D: Deepcode.

Training. In training scheme D, we initialize the encoder and decoder as the ones in Scheme C, and then additionally train the weight vectors **a** on top of the trained model, while allowing

all weights in the encoder and decoder to change as well. After training, we see that the trained weights are $(a_1, a_2, a_3, a_4) = (0.87, 0.93, 0.96, 0.98)$ and $(a_{K-3}, a_{K-2}, a_{K-1}, a_K, a_{K+1}) = (1.009, 1.013, 1.056, 1.199, 0.935)$ (for $-1dB$ trained model). As we expected, the trained weights in the later bits are larger. Also, the weight at the $K+1$th bit position is small because last bit is always zero and does not convey any information. On the other hand, trained weights in the beginning positions are small because these bits are naturally more robust to noise due to the sequential structure in Phase 2.

Result. The resulting BER curve is shown in Figure 4.6(-o-). We can see that the BER is noticeably decreased. In Figure 4.7(-o-), we can see that the BER in the last bits are reduced, and we can also see that the BER in the first bits are increased, as expected. Our use of unequal power allocation across information bits is in-line with other approaches from information/coding theory [30], [95]. We call this neural code Deepcode.

Typical error analysis. As shown in Figure 4.7, the BER at each position remains about the same except for the last few bits. This suggests a symmetry in our code and nearest-neighbor-like decoder. For an AWGN channel without feedback, it is known that the optimal decoder (nearest neighbor decoder) under a symmetric code (in particular, each coded bit follows a Gaussian distribution) is robust to the distribution of noise [70]; the BER does not increase if we keep the power of noise and only change the distribution. As an experiment demonstrating the robustness of Deepcode, in [60] appendix, to show that BER of Deepcode does not increase if we keep the power of noise and change the distribution from i.i.d. Gaussian to bursty Gaussian noise.

Complexity. Complexity and latency, as well as reliability, are important metrics in practice, as the encoder and the decoder need to run in real time on mobile devices. Deepcode has linear encoding and decoding complexity $O(K)$, where K denotes the information block length. S-K scheme and sequential forward error correcting codes, such as turbo codes and convolutional codes, also have linear encoding and decoding complexity. On the other hand, polar codes have encoding and decoding complexity $O(K \log K)$ [116]. General LDPC codes have encoding complexity $O(K^2)$ and decoding complexity $O(K)$, where as some optimized

LDPC codes have encoding time complexity $O(K)$ [98].

Actual latencies are very hard to compare because the latency critically depends on how operations are implemented in the hardware. Turbo decoder, for example, is a belief-propagation decoder with multiple (e.g., 10 – 20) *iterations* of a component decoder, and each iteration is followed by a permutation. On the other hand, the decoder for Deepcode is a 2-layered bi-directional GRU decoder, each with 50 hidden units, all of which are matrix multiplications that can be parallelized.

Whereas we can not compare the runtimes of Deepcode decoder and turbo decoder, we can compare the number of multiplications required per each bit. The bottleneck in the Deepcode decoder is the update of the write gate, forget gate, and the hidden state in 4 GRUs (forward GRU and backward GRU in layers 1 and 2); hence, in total, it includes 12 matrix (dimension 50×50) – vector (length 50) multiplications per bit. On the other hand, turbo decoder is a 10 – 20 iteration of BCJR algorithms, each of which includes between 10 to 100 multiplications per bit depending on the trellis used for the turbo code, followed by a permutation. We can similarly compare the encoder complexity. The bottleneck in the Deepcode encoder is an update of the hidden state in the RNN; which requires a matrix (dimension 50×50) – vector (length 50) multiplication per bit. Turbo encoder generates two recursive systematic convolutional codes, each requires 10s of boolean XORs per bit depending on the trellis, and a permutation of the message bit sequence. In the current form, Deepcode requires more computation than turbo code. Ideas such as knowledge distillation [44] and network binarization [48] can be used to potentially further reduce the complexity of the network. The optimization of Deepcode is beyond the scope of this chapter and is left as a future work. We again note that the runtime comparison is open (e.g., matrix-vector multiplications can be highly parallelized).

4.3 Practical considerations: noise and delay in feedback, finite precision, and blocklength

We considered so far the AWGN channel with noiseless output feedback with a unit time-step delay. In this section, we demonstrate the robustness of Deepcode (and its variants) under two variations on the feedback channel, *noise* and *delay*, as well as *finite precision*. We also present a generalization to longer block lengths. We show that (a) Deepcode and its variant that allows a K-step delayed feedback are more reliable than the state-of-the-art schemes in channels with *noisy* feedback, and (b) Deepcode concatenated with turbo code achieves superior error rate decay as block length increases with noisy feedback.

Noisy feedback. We show that Deepcode, trained on AWGN channels with *noisy* output feedback, achieves a significantly smaller BER than both S-K and C-L schemes [17]. In Figure 4.12 (Left), we plot the BER as a function of the feedback SNR for S-K scheme, C-L scheme, and Deepcode for a rate 1/3 code with 50 information bits, where we fix the forward channel SNR to be 0dB. As feedback SNR increases, we expect the BER to decrease. However, as shown in Figure 4.12 (Left), both C-L scheme, designed for channels with noisy feedback, and S-K scheme are sensitive to even a small amount of noise in the feedback, and reliability is almost independent of feedback quality. For C-L scheme, we take the experimental results for rate 1/3 (blocklength 5144) shown in [17].

Deepcode outperforms these two baseline (linear) codes by a large margin, with decaying error as feedback SNR increases, showing that Deepcode harnesses *noisy* feedback information to make communication more reliable. This is highly promising as the performance with noisy feedback is directly related to the practical communication channels. To achieve the performance shown in Figure 4.12, for example the line in red, training with matched SNR is required. For each datapoint, we use different neural codes specifically trained at the same SNR at the test noise. The neural encoder takes the feedback signal (as well as the message) as an input and includes power normalization tailored to the SNR of forward and feedback channels; hence, if trained with a mismatched SNR, the output of the neural code does not

satisfy the power constraint. In Section **??**, we discuss how Deepcode differs depending on what SNR it was trained on, hence it is not universal.

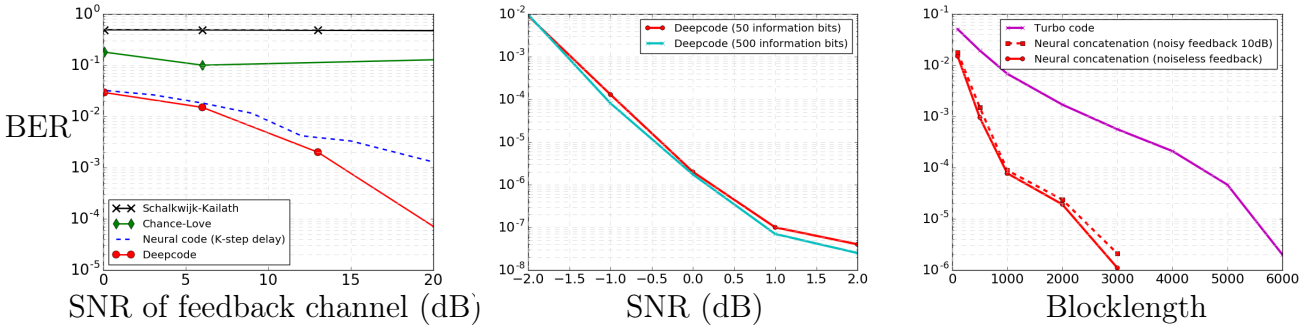


Figure 4.12: (Left) Deepcode (introduced in Section 4.2) and its variant code that allows K time-step delay significantly outperform the two baseline schemes in noisy feedback scenarios. (Middle) By unrolling the RNN cells of Deepcode, the BER of Deepcode remains unchanged for block lengths 50 to 500. (Right) Concatenation of Deepcode and turbo code (with and without noise in the feedback) achieves BER that decays exponentially as block length increases, faster than turbo codes (without feedback) at the same rate.

Noise feedback with delay. We model the practical constraint of *delay* in the feedback, by introducing a variant of Deepcode that works with a K time-step delayed feedback. for the details); recall K is the number of information bits and this code tolerates a large delay in the feedback. We see from Figure 4.12 (Left), that these neural codes are robust against delay in the feedback for noisy feedback channels of SNR up to 12dB.

Finite precision. We evaluate the sensitivity of Deepcode to the finite machine precision (without any re-training). In Figure 4.13, we plot the BER as a function of SNR for Deepcode implemented with a finite precision. We notice that under the 8 bit codeword quantization, Deepcode has almost no reliability loss. Re-training can potentially bring down the required precision even further. On the other hand, S-K scheme is very sensitive to the finite machine precision. In Figure 4.13, S-K scheme is implemented with 64-bit precision.

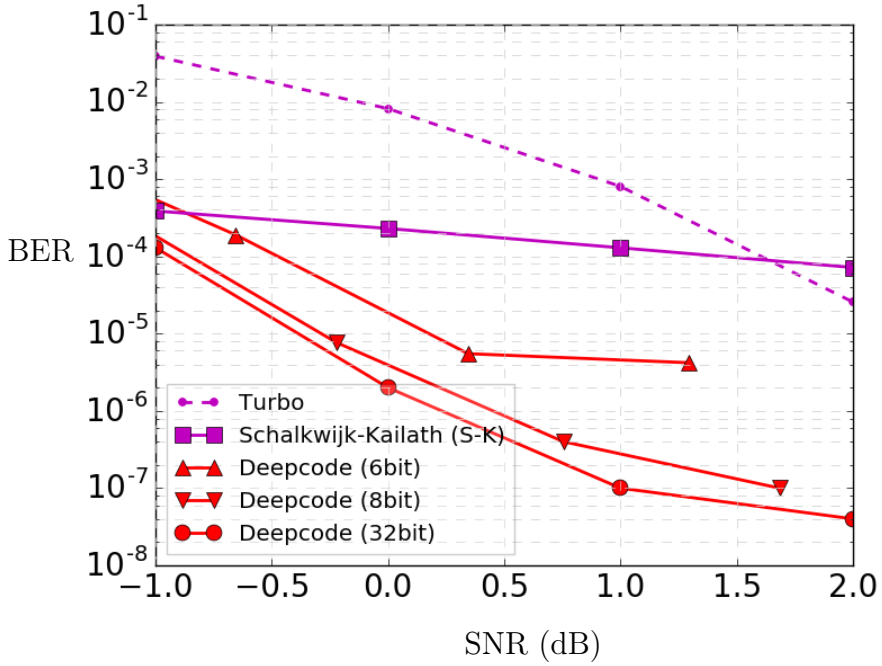


Figure 4.13: Performance of Deepcode under the scenarios where codewords are quantized to 8 bits and 6 bits.

One of the reasons for the sensitivity is that its first transmission is a M-ary PAM where M represents the number of total messages (e.g., if information block length is 50, first transmission has to be done via a 2^{50}-PAM modulation). As a means to overcome this effect, one can consider using multiple blocks each of which has a smaller block length (e.g., 5 blocks of length-10 codewords all together represent a length-50 codeword.) In Figure 4.14, we show the effect of precision (y-axis) and the length of each coding block (x-axis) on the relative performance of S-K and Deepcode for noiseless feedback (Left) and noisy feedback (Right). We let the forward SNR be 0dB for both cases and feedback SNR be 40dB (hence, very small noise) for the noisy feedback scenario. Figure 4.14 (Left) demonstrates that the S-K scheme can outperform Deepcode for noiseless feedback by reducing the coding block length as long as precision value is large enough; however, for a smaller precision (e.g., 8 bit), Deepcode always outperforms the S-K scheme. Figure 4.14 (Right) demonstrates that regardless of the

precision and the length of each coding block, Deepcode always outperforms the S-K scheme.

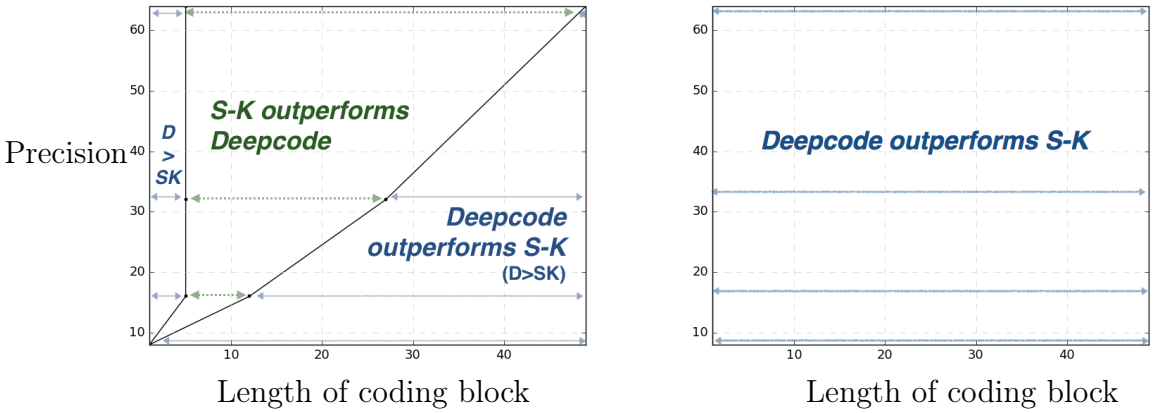


Figure 4.14: Relative performance of S-K and Deepcode as a function of machine precision (y-axis) and the length of each coding block (x-axis) for noiseless feedback (Left) and noisy feedback (Right). When precision is small (e.g., 8 bit) or feedback is noisy (even a small amount of noise; e.g., feedback SNR is 40dB), Deepcode outperforms the S-K scheme. When precision is large enough and feedback channel is noiseless, S-K can outperform Deepcode by reducing the coding block length.

Generalization to longer block lengths. In wireless communications, a wide range of blocklengths are of interest (e.g., 40 to 6144 information bits in LTE standards). In previous sections, we considered block length of 50 information bits. Here we show how to generalize Deepcode to longer block lengths and achieve an improved reliability as we increase the block length.

A natural generalization of the RNN-based Deepcode is to unroll the RNN cells. In Figure 4.12 (Middle), we plot the BER as a function of the SNR, for 50 information bits and length 500 information bits (with noiseless feedback) when we unroll the RNN cells. We can see that the BER remains the same as we increase block lengths. This is not an entirely satisfying generalization because, typically, it is possible to design a code for which error rate decays faster as block length increases. For example, turbo codes have error rate decaying exponentially (log BER decays linearly) in the block length as shown in Figure 4.12

(Right). This critically relies on the interleaver, which creates long range dependencies between information bits that are far apart in the block. Given that Deepcode is a sequential code, there is no strong long range dependence. Each transmitted bit depends on only a few past information bits and their feedback (we refer to Section **??** for a detailed discussion).

To resolve this problem, we propose a new concatenated code which concatenates Deepcode (as inner code) and turbo code as an outer code. The outer code is not restricted to a turbo code, and we refer to [60] appendix for more details. In Figure 4.12 (Right), we plot the BERs of the concatenated code, in channels with both noiseless and noisy feedback (of feedback SNR 10dB), and turbo code, both at rate 1/9 at (forward) SNR -6.5dB. From the figure, we see that even with noisy feedback, BER drops almost exponentially (log BER drops linearly) as block length increases, and the slope is sharper than the one for turbo codes. We also note that in this setting, C-L scheme suggests not using the feedback.

4.4 System and implementation issues

We began with the idealized Shannon model of feedback and have progressively considered practical variants (delay, noise and active feedback). In this section we extend this progression by studying design decisions in real-world implementations of Deepcode (our neural-network feedback-enabled codes). We do this in the context of cellular wireless systems, with specific relevance to the upcoming 5G LTE standard.

LTE cellular standards prescribe separate uplink and downlink transmissions (usually in frequency division duplex mode). Further, these transmissions are scheduled in a centralized manner by the base station associated with the cell. In many scenarios, the traffic flowing across uplink and downlink could be asymmetric (example: more "downloads" than "uploads" leads to higher downlink traffic than the combined uplink ones). In such cases, there could be more channel resources in the uplink than the traffic demand. Given the sharp inflexible division among uplink and downlink, these resources go unused. We propose to link, *opportunistically*, unused resources in one direction to aid the reliability of transmission in the opposite direction – this is done via using the feedback codes developed in this chapter.

Note that the availability of such unused channel resources is known in advance to the base station which makes scheduling decisions on *both* directions of uplink and downlink – thus such a synchronized cross uplink-downlink scheduling is readily possible.

The availability of the feedback traffic channel enables the usage of the codes designed in this chapter – leading to much stronger reliability than the feedforward codes alone. Combined with automatic repeat request (ARQ), this leads to fewer retransmissions and smaller average transmission time than the traditional scheme of feedforward codes combined with ARQ would achieve. In order to numerically evaluate the expected benefits of such a system design, in Figure 4.15, we plot BLER as a function of number of (re)transmissions for Deepcode under noiseless and noisy feedback and feedforward codes (for a rate 1/3 code with 50 information bits). From this figure, we can see that combining Deepcode with ARQ allows fewer block transmissions to achieve the target BLER compared to the state-of-the-art codes. The performance of Deepcode depends on the quality of the feedback channel. As feedback channel becomes less noisy, Deepcode requires fewer retransmissions. We note that in measuring the BLER of neural code under noisy feedback (10dB), we used a variant of Deepcode, shown as Act-Deepcode, which allows an active feedback of rate 3/4; in Phase 1, the decoder sends back RNN encoded bits at rate 1/2. Phase 2 works as in Deepcode. Hence, for a rate 1/3 code with 50 information bits, the decoder makes 200 usages of the feedback channel (204 with zero padding). Improving further the performance of (active) Deepcode at realistic feedback SNRs (such as 10dB or lower) is an important open problem. The improvements could come from architectural or learning methodology innovations or a combination of both.

We propose using Deepcode when the feedback SNR is high. Practically, a user may not always have a high SNR feedback channel, but when there are multiple users, it is possible that some of the users have high SNR feedback channels. For example, in scenarios where a base station communicates with multiple users, we propose scheduling users based on their feedback as well as forward channel qualities, utilizing multiuser diversity. In Internet-of-Things (IoT) applications, feedback channel SNR can be much higher than forward SNR;

e.g., a small device with limited power communicates a message to the router connected to the power source.

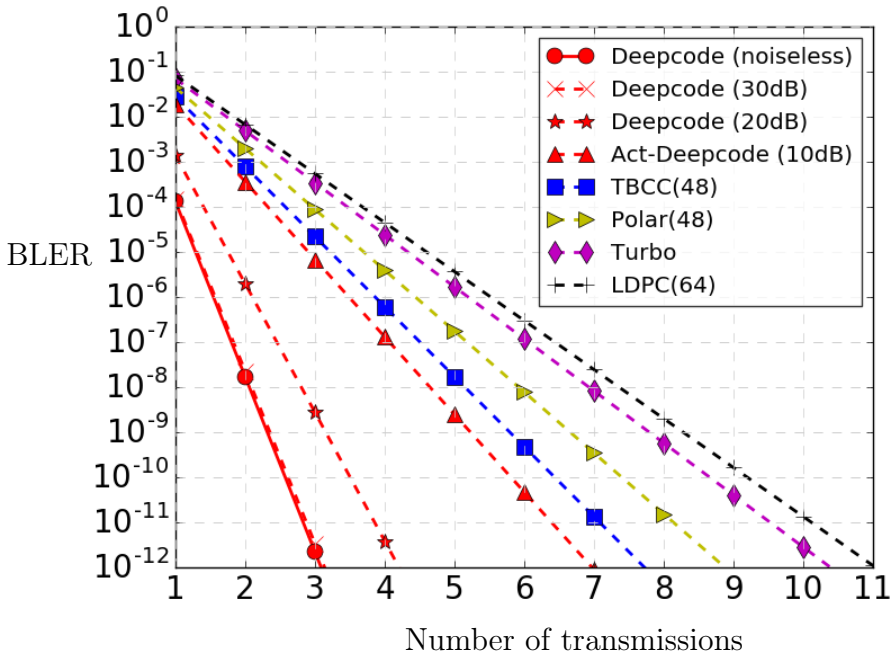


Figure 4.15: BLER as a function of number of transmissions for a rate 1/3 code with 50 information bits where forward SNR is 0dB. Deepcode allows fewer transmissions than feedforward codes to achieve the target BLER.

Chapter 5

META LEARNING FOR CHANNEL CODES

Standard decoding approaches rely on model-based channel estimation methods to compensate for varying channel effects, which degrade in performance whenever there is a model mismatch. Recently proposed Deep learning based neural decoders address this problem by leveraging a model-free approach via gradient-based training. However, they require large amounts of data to re-train to achieve the desired adaptivity, which becomes intractable in practical systems. This chapter proposes a new decoder: Model Independent Neural Decoder (MIND), which builds on the top of neural decoders and equips them with a fast adaptation capability to varying channels. This feature is achieved via the methodology of Model-Agnostic MetaLearning (MAML). Here the decoder: (a) learns a "good" parameter initialization in the meta-training stage where the model is exposed to a set of archetypal channels and (b) updates the parameter with respect to the observed channel in the meta-testing phase using minimal adaptation data and pilot bits. Building on top of existing state-of-the-art neural Convolutional and Turbo decoders, MIND outperforms the static benchmarks by a large margin. It shows a minimal performance gap compared to the neural (Convolutional or Turbo) decoders designed for that particular channel. Besides, MIND also shows strong learning capability for channels not exposed during the meta training phase. MIND was previously published in [52][1].

[1]The material in this chapter is based on joint work with Hyeji Kim, Himanshu Asnani, Sreeram Kannan, Sewoong Oh, Pramod Viswanath

5.1 Problem Settings

5.1.1 Motivation

Efficient decoding methods are known for the capacity-approaching codes, and they exhibit near-optimal performance on the Gaussian noise (AWGN) channel. However, the performance on non-AWGN channels is not uniformly optimal. Designing the corresponding decoders to deal with non-Gaussianity is challenging, primarily owing to a two-fold deficit: (a) **model-deficit**, which implies the inability of accurately expressing the observed data by a clean mathematical model, and (b) **algorithm deficit**, which implies even under a clean abstraction, the optimal decoding algorithm is not known [55]. Thus, while using the optimal codes designed under simplified models such as the AWGN channel, designing a decoder that can adapt to the non-AWGN channel effects faces challenges on both these fronts: there is a model mismatch, and most non-AWGN channels do not permit closed-form optimal decoders.

A tremendous amount of effort has been invested in developing a suite of handcrafted algorithms to circumvent these deficits. These comprise model-based methods in channel estimation, signal preprocessing, and robust decoding under unexpected channel effects [118], to make the AWGN-designed capacity-approaching decoders operate with minimal degradation [99]. Few pilot bits known by both the transmitter and the receiver are used to estimate the channel effects to compensate for their varying nature. At the same time, handcrafted decoding algorithms have been applied to improve the decoder's robustness [99]. However, they lack in two respects: (1) Channel estimation and channel-effect equalizing algorithms are model-based, hence when the underlying mathematical abstraction suffers from model-deficit, there is a suboptimal performance. (2) AWGN-designed decoders are not robust to unexpected and uncompensated noises.

5.1.2 Prior Art : Neural Decoding

In the past decade, data-driven deep learning based methods have changed the landscape of several engineering fields such as computer vision and natural language processing, with

revolutionary performance benchmarks [26] [28]. Applying general-purpose deep learning models to channel coding design has received intensive attention recently [91] [89]. Designing such neural decoders naturally fits well with the data-driven supervised learning approaches since both the received signals and the target messages can be simulated from the underlying encoder and channel models. In this way, both the model-deficit and algorithm-deficit are navigated by directly training a neural decoder on the sampled data.

Designing neural decoder for several classes of codes such as LDPC codes, Polar codes, and Turbo codes with versatile deep neural networks has seen growing interest within the channel coding community. Imitating Belief Propagation (BP) algorithm via learnable neural networks shows promising performance for High-Density Parity-Check (HDPC) codes and LDPC codes [85] [86] and Polar codes [37] [16]. Near-optimal performance of Convolutional Code and Turbo Code under AWGN channel is achieved via Recurrent Neural Networks (RNN) for arbitrary block lengths [61], which also shows robust and adaptive performance under non-AWGN setups. A further extension of RNN encoders (and decoders) reveal state-of-the-art performance for feedback channels [60] and low latency schemes [55]. Thus while neural decoders show the promise of alleviating model and algorithm deficits, compared to the traditional decoding methods, which utilize a limited amount of pilot bits to adapt, neural decoders require a large amount of data (*information complexity*) and long computation time (*computational complexity*) to adapt to the new channel. This severe drawback renders them quite intractable and far from practical deployment. The relevant question we ask here is the following: ***Can we design neural decoders that strengthen their adaptive property so that only minimal re-training is necessary?*** In what follows, this question is investigated and answered in the affirmative.

5.1.3 Our Contribution

We introduce meta-learning to navigate the data-hungry nature of the neural decoder. Meta-learning operates in two steps: (a) it firstly performs **meta training phase** by learning on a wide range of archetypal tasks, and then (b) during the **meta testing phase** enables learning

new tasks faster while consuming fewer adaptation data than learning from scratch [119]. Supervised meta-learning has a natural connection to adaptive decoder design, as we can consider different channels as different tasks in our meta-learning framework.

RNN-based meta-learning considers the whole meta-learning approach as a large-scale RNN with tasks as inputs [102]. However, this requires complex modeling and thus shows degradation in performance with respect to scalability. Model Agnostic Meta-Learning (MAML) [33] is a gradient-based meta-learning algorithm that learns a sensitive initialization for fast adaptation. MAML trained model performs well on new tasks with limited gradient update steps and few-shot adaptation data. Compared to other meta-learning methods, MAML has much less complexity. Moreover, theoretically, MAML is shown to approximate any meta-learning algorithm [34]. When faced with out-of-domain tasks, MAML shows the fast capability to adapt, although the out-of-domain tasks may not be close to the meta-trained tasks [33].

This work presents a MAML-based neural decoder: Model Independent Neural Decoder (MIND), which admits fast adaptation with few shot adaptation data utilizing the gradient-based training. Compared to the adaptive neural decoders, which require large amounts of gradient training steps and data to adapt to new channel settings, MIND can adapt to a new channel with a small number of pilot bits and few gradient descent steps. Compared to the traditional adaptive decoding method, MIND offers a model-free gradient-based meta-learning approach built on the top of neural decoders, resolving both the model and the algorithm deficit. Thus, MIND enhances the advantages of neural decoders with data and computational efficiency.

The chapter is organized as follows: Section 5.2 discusses the details of MAML, which builds on the top of neural decoders to results in our proposed decoder: MIND. Section 5.3 analyzes the performance of MIND, which shows very near-optimal performance with few shot adaptation data, under both trained and untrained channels. Section 5.4 concludes with the scope and limitations of MIND and discussion on the future directions.

5.2 Meta Learning for Neural Decoder

We consider the two neural decoders for Convolutional Code and Turbo Code respectively [61] to develop MIND[2]. Both these neural decoders have a larger number of parameters compared to the traditional algorithms [122] [11] to deal with the issues of model deficit and algorithm deficit. However, training neural decoders till convergence requires large amounts of data. This leads to a slow adaptation with costly computations. In what follows, we propose the remedy through MAML, which are described below along with the choice of the Loss function and the hyper-parameters:

Loss Function:

For neural decoders, the loss function is Binary Cross-Entropy (BCE) since decoder is a classification task. f_θ is the neural decoder with parameter θ. Formally speaking, we are given a collection of M training channels $\{T\} = \{T_i, i \in 1, ..., M\}$. For a specific channel T_i with sampled received signal $x_i^{(j)}$ and target message $y_i^{(j)}$, as (j) indicates the datapoint index in specific channel, the loss function associated with a particular channel T_i can be represented as:

$$L_{T_i}(f_\theta) = \sum_{x^{(j)}, y^{(j)} \sim T_i} BCE(f_\theta(x^{(j)}), y^{(j)}). \tag{5.1}$$

Meta Training Phase:

The meta training objective is to learn a sensitive initial weight for all the training channels. This operates as per the following two sub-steps:

- **Task Update:** For each channel T_i, MIND updates the model weights θ to $\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$ with adaptation learning rate α. This is called task update as the update for the parameter is done for each task, here channel. The updated weights θ' should learn themselves to be close to the optimal decoder for each channel T_i.

[2]More details check the appendix of [52]

- **Meta Update:** Here, the goal is to do a meta update or to minimize the following loss for all training channels with respect to θ:

$$\min_{\theta} \sum_{i} L_{T_i}(\theta_i') = \min_{\theta} \sum_{i} L_{T_i}(\theta - \alpha \nabla_{\theta} L_{T_i}(f_{\theta})) \tag{5.2}$$

which via gradient descent with meta learning rate β, is equivalent to the following update:

$$\theta \leftarrow \theta - \beta \nabla_{\theta} \sum_{T_i \in \{T\}} L_{T_i}(f_{\theta_i'}) \tag{5.3}$$

Computing the above gradient is equivalent to computing the gradient of gradient of the BCE loss. Second order gradients as in Eq. (5.3) are expensive. In this chapter, we use First-Order MAML (FOMAML) [87], which treats higher order gradients as constant and operates in two steps: (1) compute the update $\theta_i' = \theta - \alpha \nabla_{\theta} L_{T_i}(f_{\theta})$, (2) compute the gradient $\nabla_{\theta} \sum_{T_i \in \{T\}} L_{T_i}(f_{\theta_i'})$ by directly computing the gradient using updated θ_i to ignore the second-order terms. FOMAML has comparable performance, and needs far less computation compared to MAML.

Note that training phase with the vanilla average learning, known as Multi-task Learning (MTL) [32], use the following assignment via the average of gradients on all the channels instead of Eq. (5.3):

$$\theta \leftarrow \theta - \beta \nabla_{\theta} \sum_{T_i \in \{T\}} L_{T_i}(f_{\theta}). \tag{5.4}$$

Meta Testing Phase:

During the meta testing phase, firstly pilot bits from the new channel T_i are collected. Then the θ is updated via gradient descent $\theta' = \theta - \alpha \nabla_{\theta} L_{T_i}(f_{\theta})$. MIND's meta training and testing phase is depicted in detail in [52] appendix.

Note: During the meta training phase, the data to compute task update $\nabla_{\theta} L_{T_i}(f_{\theta_i'})$ and the data for computing meta update $\theta \leftarrow \theta - \beta \nabla_{\theta} \sum_{T_i \in \{T\}} L_{T_i}(f_{\theta_i'})$ are different. Using the

same data for both the task update and the meta update leads to meta-overfitting [34]. It is due to this reason for training each T_i, we need to sample twice for meta training, while during the meta testing phase each step only requires to sample once.

Parameters	Convolutional Code	Turbo Code
Neural Decoder	2 layer bi-GRU	2 layer bi-GRU
Number of Neural Units	200	200
Batch Size B	100	100
Meta Batch Size P	10	10
Meta Learning Rate β	0.00001	0.00001
Adaptation Learning Rate α	0.001	0.0001
Number of Meta Update Steps	50000	50000
Block Length L	100	100
Train SNR	0 to 4dB	-1.5 to 2dB
Code Rate	1/2	1/3

Figure 5.1: MIND Hyperparameters

Hyperparameters:

The MIND trained Neural Decoders for Convolutional and Turbo Code are trained with the following hyper-parameters as shown in Figure 5.1. Batch size B refers to the number of blocks sampled from one specific channel for training (also referred to as mini-batch size), which is the same for both meta training and meta testing phase. Meta batch size P refers to the number of random channels utilized for each meta training update step. Meta training is expensive, which uses 50000 training steps to conduct Meta Update. The adaptation rate α in the task update of meta training phase is larger than the meta-learning rate β of the meta update, which allows MIND to adapt faster. We use a smaller adaptation learning rate α

Testing Method	Adaptation Data	Task Update Steps K
Fine-tune	1000000	10000
MIND-1 Meta Testing	100	1
MIND-10 Meta Testing	1000	10

Figure 5.2: Adaptation cost between MIND and full adaptation

for neural Turbo decoder due to its sensitive iterative decoding structure with shared model weights [61].

The data and computation cost for the meta testing phase is shown in Figure 5.2. The task update step refers to the number of gradient steps K required before testing on the new channel. Here we use the trained batch size $B = 100$. Fine-tuning neural decoder without MIND to adapt to new channel requires $K = 10000$ steps (each step needs $B = 100$ blocks) to converge. Compared to the fine-tuning, MIND only requires $K = 1$ or $K = 10$ gradient steps to conduct fast adaptation, with far fewer pilot data during the meta test phase. In what follows for the evaluation of MIND's performance, MIND-K refers to MIND with K gradient update steps in the meta testing phase. A detailed discussion on hyper-parameters' effects is deferred the appendix of [52].

5.3 Performance

This section investigates the performance of MIND-K for convolution code and turbo code against several benchmarks.

5.3.1 Channel Settings and Benchmarks

The channels used in this chapter are:

- AWGN channel: $y = x + z$, $z \sim N(0, \sigma^2)$.

- Additive T-distribution Noise (ATN) channel: $y = x + z$, where $z \sim T(\nu, \sigma^2)$.

- Radar Channel: $y = x + z + w$. where $z \sim N(0, \sigma_1^2)$ is a background AWGN noise, and $w \sim N(0, \sigma_2^2)$, with probability p is the radar noise with high variance and low probability. $\sigma_1 << \sigma_2$.

5.3.2 Benchmarks

For both the convolutional code and turbo code, we compare the MIND-K decoder against the following benchmarking decoders:

- **Canonical Optimal Decoders for AWGN Channel:** For convolutional code, the Viterbi algorithm has optimal BER performance for AWGN channels [122]. For Turbo code, iterative Turbo decoder based on BCJR [11] shows capacity-approaching performance. When decoding on AWGN channels, the above two decoders serve as useful benchmarks to compare. When testing non-AWGN channels, the canonical optimal decoders are treated as static benchmarks since the AWGN-designed optimal method without adaptation is sub-optimal on non-AWGN channels.

- **Adaptive Neural Decoders:** Under non-AWGN channels, generally there doesn't exist a close-form optimal decoding scheme. On the other hand, in these cases, neural decoders outperform most state-of-the-art heuristic decoders [61]. Adaptive Neural Decoders are trained with nearly unlimited data and computing resources on a particular channel and thus provide another useful benchmark, especially for the non-AWGN channels.

- **Multi-task Learning (MTL) based Decoders:** This is a benchmark for naive adaptation, termed as **MTL-K**, which updates weights via K-step gradient descent directly from MTL trained weight (Eq. 5.4), with the same adaptation data batch size and learning rate as MIND-K.

5.3.3 MIND-K for Convolutional Code

We evaluate the fast adaptation ability under 4 different channels shown in Figure 5.3: (1) AWGN channel, (2) Radar Channel ($\sigma_2 = 2.0$ and $p = 0.05$), (3) ATN ($\nu = 3.0$), and (4) untrained Radar ($\sigma_2 = 100.0, p = 0.05$). The first three channels aim to test the fast adaptation ability on meta-trained channels, where the fourth channel aims to test learning ability on an unexpected channel with dramatically different parameters.

The MIND performance on Convolutional Code shows on trained channels:

- Among static methods without adaptation ability, MIND-0 and MTL-0 show similar performance. MIND without adaptation still performs well.

- MIND-1 performs better than MTL-1, MIND-0, and MTL-0. MTL-1 shows a degradation indicating that naive learning via average performance on all channels is not stable.

To show the continued learning property on the untrained channel, we also consider MIND-10 to compare. Here we observe:

- MIND-1 outperforms MTL-1, MTL-0, MIND-0. On the untrained channel, MIND still shows improvement with solely gradient.

- MIND-10 outperforms MTL-1.On the untrained channel, apply more gradient steps can further improve performance.

5.3.4 MIND-K for Turbo Code

As MTL-1 performs poorly, in this section we ignore MTL-1. On Turbo code, the channels tested shown below in Figure 5.4 are: (1) trained Radar channel ($\sigma_2 = 2.0, p = 0.05$), and (2) untrained Radar channel ($\sigma_2 = 100.0, p = 0.01$).

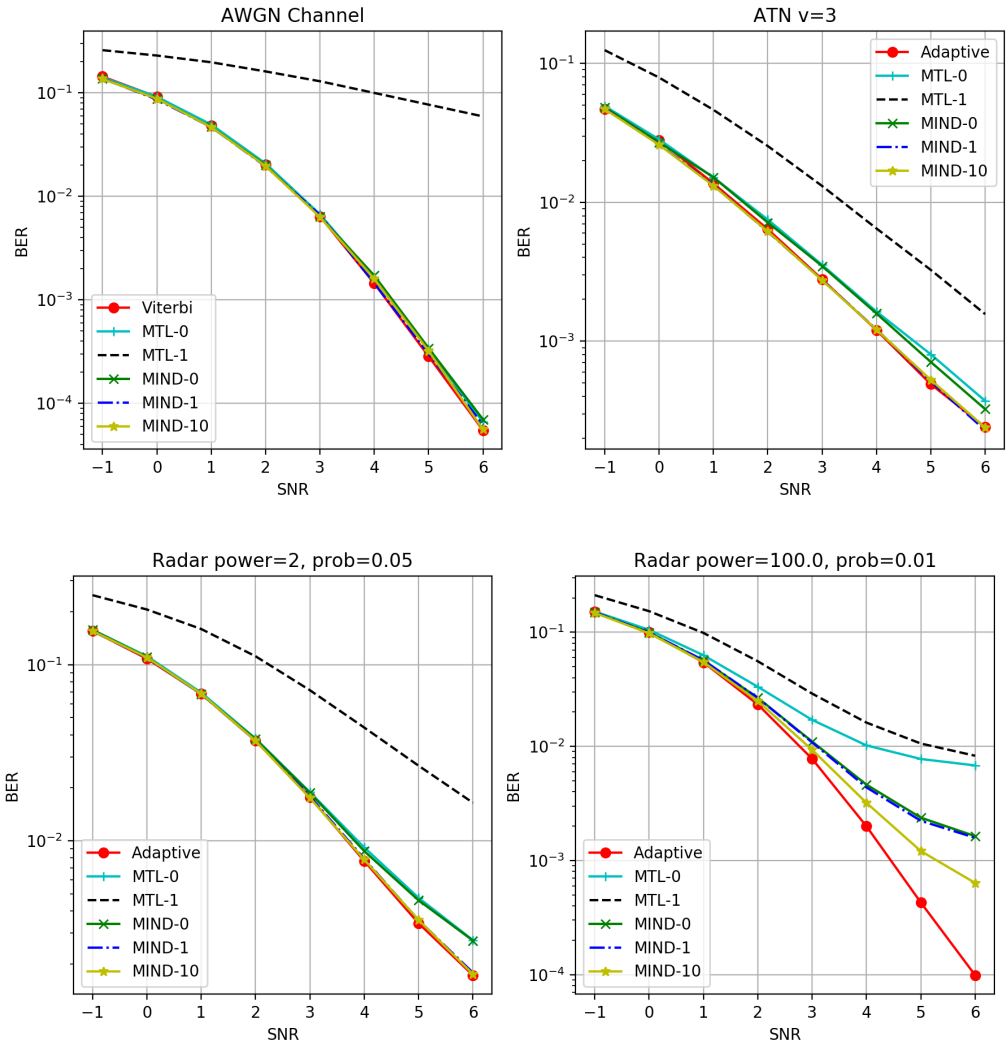


Figure 5.3: MIND for Convolutional Code: Trained AWGN (up left);Trained ATN ($\nu = 3$) (up right);Trained Radar($\sigma_2 = 2.0, p = 0.05$) (down left). and untrained Radar($\sigma_2 = 100.0, p = 0.05$)(down right).

The performance on MIND with neural Turbo decoder shows the same trend as with Convolutional Code. The performance of MIND is consistent for both neural decoders as follows:

- Without adaptation ability, MIND-0 shows robust performance, comparable to neural

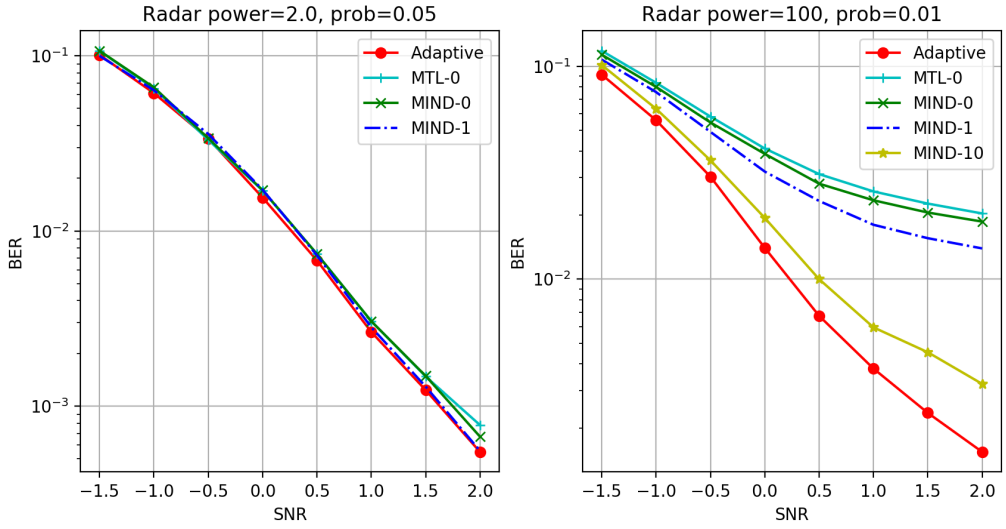


Figure 5.4: Neural Turbo Decoder with MIND. Trained Radar($\sigma_2 = 2.0, p = 0.05$) (left), and untrained Radar($\sigma_2 = 100.0, p = 0.01$)(right)

decoder trained on multiple channels.

- With limited data and computation, MIND-1 outperforms static methods and shows performance close to optimal or adaptive algorithms.

- On untrained channels, applying MIND with more gradient steps continually improves accuracy.

Note that on trained channels shown in Figure 5.3 and Figure 5.4, MIND-1 performs very close to optimal algorithms. Comparing to deploying MTL-trained neural decoders, MIND shows comparable performance without adaptation ability, and can conduct fast adaptation with minimal re-training on both trained and untrained channels.

5.4 Discussion

While we have designed MIND particularly for convolutional and Turbo codes, the methodology is not limited to these codes. In fact, the overall methodology is independent on

the code structure or the neural network architecture, and thus can be adapted with equal felicity to other neural-based decoding problems. We note that MIND is not expected to be a universal decoder for **all** channels, rather that the learnt initialization is good for a class of channels which are related to the archetypal channels, with experiments shown in [52] appendix. A precise characterization of this class is an interesting direction for future research. Furthermore, MIND still requires more samples than maybe available in a typical training channel. We expect neural method for joint channel estimation and data detection to perform better - this is left for future work.

Among future directions, it is worth considering to combine other neural decodes with MIND, such as neural LDPC [85] [86] and Polar [16] decoders. Beyond neural decoder design, MAML can also be applied to Channel Autoencoder [91] design, which deals with designing adaptive encoder and decoder. These can usher new directions of remarkable improvements in decoder design.

Chapter 6

FEDERATED LEARNING PERSONALIZATION

Federated Learning (FL) refers to learning a high-quality global model based on decentralized data storage without ever copying the raw data. A natural scenario arises with data created on mobile phones by the activity of their users. Given the typical data heterogeneity in such situations, it is natural to ask how the global model can be personalized for every such device, individually. In this work, we point out that the setting of Model Agnostic Meta Learning (MAML), where one optimizes for a fast, gradient-based, few-shot adaptation to a heterogeneous distribution of tasks, has several similarities with the objective of personalization for FL. We present FL as a natural source of practical applications for MAML algorithms and make the following observations. (a) The popular FL algorithm, Federated Averaging, can be interpreted as a meta learning algorithm. (b) Careful fine-tuning can yield a global model with higher accuracy, which is at the same time easier to personalize. However, solely optimizing for the global model accuracy yields a weaker personalization result. (c) A model trained using a standard datacenter optimization method is much harder to personalize than one trained using Federated Averaging, supporting the first claim. These results raise new questions for FL, MAML, and broader ML research. This method was previously published as [56][1].

6.1 Problem Settings

In recent years, the growth of machine learning applications was driven by the aggregation of large amounts of data in a data center, where a model can be trained using a large-scale

[1]The material in this chapter is based on joint work with Jakub Konečnỳ, Keith Rush and Sreeram Kannan, when interning in Google Research, Beijing

distributed system [25, 72]. Both the research community and the general public are becoming increasingly aware that there are various scenarios where this kind of data collection comes with significant risks, mainly related to notions of privacy and trust.

In the presence of user-generated data, such as activity on mobile phones, Federated Learning (FL) [82] proposes an alternative approach for training a high-quality global model without ever sending raw data to the cloud. The FL system proposed by Google [12] selects a sample of available devices and sends them a model to be trained. The devices compute an update to the model based on an optimization procedure with locally available data, and the central system aggregates the updates from different devices. Such iteration is repeated many times until the model has converged. The users' training data does not leave their devices. The basic FL algorithm, Federated Averaging (FedAvg) [82], has been used in production applications, for instance, for next word prediction in mobile keyboard [40], which shows that Federated Learning can outperform the best model trained in a datacenter. Successful algorithmic extensions to the central idea include training a differential private model [84], compression [66, 14], secure aggregation [13], and a smaller number of always-participating nodes [129].

FL applications generally face non-i.i.d and unbalanced data available to devices, making it challenging to ensure good performance across different devices with an FL-trained global model. Theoretical guarantees are only available under restrictive assumptions and for convex objectives, cf. [75]. In this work, we are interested in *personalization* methods that adapt the model for data available on each device, individually. We refer to a trained global model as the *initial model* and the locally adapted model as the *personalized model*. Existing FL personalization work directly takes a converged initial model and conducts personalization evaluation via gradient descent [9]. However, in this approach, the training and personalization procedures are entirely disconnected, which results in potentially suboptimal personalized models.

Meta Learning optimizes the performance after adaptation given few-shot adaptation examples on heterogeneous tasks and has increasing applications in Supervised Learning

and Reinforcement Learning. Model Agnostic Meta Learning (MAML) introduced by [33] is a solely gradient-based Meta Learning algorithm, which runs in two connected stages; meta-training and meta-testing. Meta-training learns a sensitive initial model which can conduct fast adaptation on a range of tasks, and meta-testing adapts the initial model for a particular task.

Both tasks for MAML, and clients for FL, are heterogeneous. For each task in MAML and client in FL, existing algorithms use a variant of gradient descent locally and send an overall update to a coordinator to update the global model. If we present the FL training process as meta-training in the MAML language, and the FL personalization via gradient descent as meta-testing, we show in Section 6.2 that FedAvg [82] and Reptile [87], two popular FL and MAML algorithms, are very similar to each other; see also [59].

In order to make FL personalization useful in practice, we propose that the following objectives must *all* be addressed, simultaneously.

(1) **Improved Personalized Model** – for a large majority of the clients.
(2) **Solid Initial Model** – some clients have limited or even no data for personalization.
(3) **Fast Convergence** – reach a high quality model in small number of training rounds.

Typically, the MAML algorithms only focus on objective (1); that was the original motivation in [33]. Existing FL works usually focus on objectives (2) and (3) and take the personalized performance secondary. This is mostly since it was not apparent that getting a solid initial model is feasible or practical if devices are available occasionally and with limited resources.

In this work, we study these three objectives jointly, and our main contributions are:

- We point out the connection between two widely used FL and MAML algorithms and interpret existing FL algorithms in the light of existing MAML algorithms.
- We propose a novel modification of FedAvg, with two stages of training and fine-tuning, for optimizing the three above objectives.

- We empirically demonstrate that FedAvg is already a meta learning algorithm, optimizing for personalized performance, as opposed to the global model's quality. Furthermore, we show that the fine-tuning stage enables better and more stable personalized performance.
- We observe that different global models with the same accuracy can exhibit a very different capacity for personalization.
- We highlight that these results challenge the existing objectives in the FL literature and motivate new problems for the broader Machine Learning research community.

6.2 Methods

This section highlights the similarities between the FL and MAML algorithms and interprets FedAvg as a linear combination of a naive baseline and a collection of existing MAML methods.

Algorithm 1 presents a conceptual algorithm with nested structure (left column), of which the MAML meta-training algorithm, Reptile (middle column), and FL-training algorithm, FedAvg (right column), are particular instances. We assume that L is a loss function common to all of the following arguments. In each iteration, a MAML algorithm trains across a random batch of tasks $\{T_i\}$. For each task T_i, it conducts an inner-loop update and aggregates gradients from each sampled task with an outer-loop update. In each training round, FL uses a random selection of clients $\{T_i\}$. For each client T_i and its weight w_i, it runs an optimization procedure for several epochs over the local data and sends the update to the server, which aggregates the updates to form a new global model. If we simplify the setting and assume all clients have the same amount of data, causing the weights w_i to be identical, Reptile and FedAvg, become the same algorithms. Several other MAML algorithms [33, 2], or other non-MAML/FL methods [132], can also be viewed as instances of the conceptual method in the left column of Figure 6.1.

In the following, we rearrange the summands comprising the update formula of FedAvg/Reptile algorithm to reveal the connection with other existing methods – a linear

Algorithm 1 Connects FL and MAML (left), Reptile Batch Version(middle), and FedAvg (right).

OuterLoop/Server learning rate α
InnerLoop/Client learning rate β
Initial model parameters θ
while not done **do**
  Sample batch of tasks/clients $\{T_i\}$
  for Sampled task/client T_i **do**
    if FL **then**
      $g_i, w_i = ClientUpdate(\theta, T_i, \beta)$
    else if MAML **then**
      $g_i = InnerLoop(\theta, T_i, \beta)$
    end if
  end for
  if FL **then**
    $\theta = ServerUpdate(\theta, \{g_i, w_i\}, \alpha)$
  else if MAML **then**
    $\theta = OuterLoop(\theta, \{g_i\}, \alpha)$
  end if
end while

Require: : Reptile Step K.
  function $InnerLoop(\theta, T_i, \beta)$
    Sample K-shot data $D_{i,k}$ from T_i.
    $\theta_i = \theta$
    for each local step i from 1 to K **do**
      $\theta_i = \theta_i - \beta \nabla_\theta L(\theta_i, D_{i,k})$
    end for
    Return $g_i = \theta_i - \theta$
  end function
Require: : Meta Batch Size M.
  function $OuterLoop(\theta, \{g_i\}, \alpha)$
    $\theta = \theta + \alpha \frac{1}{M} \sum_{i=1}^{M} g_i$
    Return θ
  end function

Require: FedAvg Local Epoch E.
  function $ClientUpdate(\theta, T_i, \beta)$
    Split local dataset into batches B
    $\theta_i = \theta$
    for each local epoch i from 1 to E **do**
      for batch $b \in B$ **do**
        $\theta_i = \theta_i - \beta \nabla_\theta L(\theta_i, b)$
      end for
    end for
    Return $g_i = \theta_i - \theta$
  end function
Require: Clients per training round M.
  function $ServerUpdate(\theta, \{g_i, w_i\}, \alpha)$
    $\theta = \theta + \alpha \sum_{i=1}^{M} w_i g_i / \sum_{i=1}^{M} w_i$
    Return θ
  end function

Figure 6.1: Both FedAvg and Reptile can be formed in left, and both inner and outer loop are nearly the same.

combination of the Federated SGD (FedSGD) [82] and First Order MAML (FOMAML) algorithms [33] with a different number of steps. For clarity, we assume identical weights w_i in FedAvg.

Consider T participating clients and let θ be parameters of the relevant model. For each client i, define its local loss function as $L_i(\theta)$, and let g_j^i be the gradient computed in j^{th} iteration during a local gradient-based optimization process.

FedSGD was proposed as a naive baseline against which to compare FL algorithms. For each client, it simply takes a single gradient step based on the local data, which is sent back to the server. It is a sensible baseline because it is a variant of what a traditional optimization method would do if we were to collect all the data in a central location, albeit inefficient in the FL setting. That means that FedSGD optimizes the performance of the *initial* model, as is the usual objective in datacenter training. The local update produced by FedSGD, g_{FedSGD}, can be written as

$$g_{FedSGD} = \frac{-\beta}{T}\sum_{i=1}^{T}\frac{\partial L_i(\theta)}{\partial \theta} = \frac{1}{T}\sum_{i=1}^{T} g_1^i. \tag{6.1}$$

Next, we derive the update of FOMAML in similar terms. Assuming client learning rate β, the personalized model of client i, obtained after K-step gradient update is $\theta_K^i = U_K^i(\theta) = \theta - \beta\sum_{j=1}^{K} g_j^i = \theta - \beta\sum_{j=1}^{K}\frac{\partial L_i(\theta_j)}{\partial \theta}$. Differentiating the client update formula, we get

$$\frac{\partial U_K^i(\theta)}{\partial \theta} = I - \beta\frac{\partial\sum_{j=1}^{K} g_j^i}{\partial \theta} = I - \beta\sum_{j=1}^{K}\frac{\partial^2 L_i(\theta_j)}{\partial \theta^2}. \tag{6.2}$$

Directly optimizing the current model for the personalized performance after locally adapting K gradient steps results in the general MAML update proposed by [33].

$$g_{MAML} = \frac{\partial L_{MAML}}{\partial \theta} = \frac{1}{T}\sum_{i=1}^{T}\frac{\partial L_i(U_K^i(\theta))}{\partial \theta} = \frac{1}{T}\sum_{i=1}^{T} L_i'(U_K^i(\theta))(I - \beta\sum_{j=1}^{K}\frac{\partial^2 L_i(\theta_j)}{\partial \theta^2}). \tag{6.3}$$

MAML requires to compute 2nd-order gradients, which can be computationally expensive and creates potentially infeasible memory requirements. To avoid computing the 2nd-order term, FOMAML simply ignores it, resulting in a first-order approximation of the objective [33]. $FOMAML(K)$ then uses the $(K+1)^{th}$ gradient as the local update, after K gradient steps.

$$g_{FOMAML}(K) = \frac{1}{T}\sum_{i=1}^{T} L_i'(U_K^i(\theta))I = \frac{1}{T}\sum_{i=1}^{T} L_i'(\theta_K^i) = \frac{1}{T}\sum_{i=1}^{T} g_{K+1}^i. \tag{6.4}$$

Now we have derived the building blocks of the FedAvg. As presented in Algorithm 1, the update of FedAvg, g_{FedAvg}, is the average of client updates, which are the sums of local gradient updates. Rearranging the terms presents its interpretation as a linear combination of the above ideas.

$$g_{FedAvg} = \frac{1}{T}\sum_{i=1}^{T}\sum_{j=1}^{K} g_j^i = \frac{1}{T}\sum_{i=1}^{T} g_1^i + \sum_{j=1}^{K-1}\frac{1}{T}\sum_{i=1}^{T} g_{j+1}^i = g_{FedSGD} + \sum_{j=1}^{K-1} g_{FOMAML}(j) \tag{6.5}$$

Note that interpolating to special case, g_{FedSGD} can be seen as $g_{FOMAML}(0)$ – optimizing the performance after 0 local updates, i.e., the current model. This sheds light on the

existing Federated Averaging algorithm, as the linear combination of algorithms optimizing personalized performance after a range of local updates. Note, however, this does *not* mean that FedAvg optimizes for the linear combination of the respective algorithms' objectives. Nevertheless, in the following section, we show that using $K = 1$ results in a model hard to personalize, and increasing K significantly improves the personalization performance, up until a certain point where the performance of the initial model becomes unstable.

6.3 Performance Analysis

In this section, we present the Personalized FedAvg algorithm, which results from experimental adaptation of the core FedAvg algorithm to improve the three objectives proposed in the introduction.

We denote $FedAvg(E)$ the Federated Averaging method from Algorithm 1, right, run for E local epochs, weighting the updates proportionally to the amount of data available locally. We denote $Reptile(K)$ the method from Algorithm 1, middle, run in the FL setting for K local steps, irrespective of the amount of data available locally. Based on various experiments we explored, we propose **Personalized FedAvg** in Algorithm 2.

Algorithm 2 Personalized FedAvg

1: Run $FedAvg(E)$ with momentum SGD as server optimizer and a relatively larger E.
2: Switch to $Reptile(K)$ with Adam as server optimizer to fine-tune the initial model.
3: Conduct personalization with the same client optimizer used during training.

In general, FedAvg training with several local epochs ensures reasonably fast convergence in terms of the number of communication rounds. Due to the complexity of the production system, this measure was studied as the proxy for FL algorithms' convergence speed. We find that this method with momentum SGD as the server optimizer already optimizes for the *personalized model* – objective (1) form the introduction – while the *initial model* – objective (2) – is relatively unstable. Based on prior work, the recommendation to address this problem

would be to decrease E or the local learning rate, stabilizing the initial model at the cost of slowing down convergence [82, 123] – objective (3).

We propose a fine-tuning stage using $Reptile(K)$ with small K and Adam as the server optimizer to improve the initial model while preserving and stabilizing the personalized model. We observed that Adam yields better results than other optimizers, and makes the best-personalized performance achievable with a broader set of hyperparameters, see Figure 6.3. The subsequent deployment and personalization is conducted using the same client optimizer as used for training, as we observe that this choice yields the best results for FedAvg/Reptile-trained models.

Experimental setup. We use the EMNIST-62 dataset as our primary benchmark [15]. It is the original source of the MNIST dataset, which comes with author id, and noticeable variations in style, pen width, size, etc., making it a suitable source for simulated FL experiments. The dataset contains 3400 users, each with a train/test data split, with a total of $671,585$ train and $77,483$ test images. We choose the first $2,500$ users as the initial training clients, leaving the remaining 900 clients to evaluate personalization; these clients are not touched during training. The evaluation metrics are the initial and personalized accuracy, uniformly averaged among all of the FL-personalization clients. This is preferred to a weighted average, as in a production system, we care about each device's future performance, regardless of the amount of data available for personalization. Unless specified otherwise, we use the baseline convolutional model available in TensorFlow Federated [49][2], using SGD with learning rate 0.02 and batch size of 20 as the client optimizer, and SGD with momentum of 0.9 and learning rate 1.0 as the server optimizer. Each experiment was repeated 9 times with random initialization, and the mean and standard deviation of initial and personalized accuracies are reported. We also use the shakespeare dataset for next-character prediction, split similarly with the first 500 clients used for training and the remaining 215 for personalization evaluation.

[2]Presented experiments were implemented in TFF, and we will make the code publicly available.

6.3.1 Convergence of FedAvg

In Figure 6.2, left, we present the convergence of both initial and personalized models during training using the Federated Averaging algorithm. The results correspond to training with E being 2 and 10, with visualization of the empirical mean and variance observed in the 9 replicas of the experiment. Detailed values about the performance after 500 rounds of training, and the number of rounds to reach 80% accuracy, are provided in Table 6.1. These results provide several valuable insights.

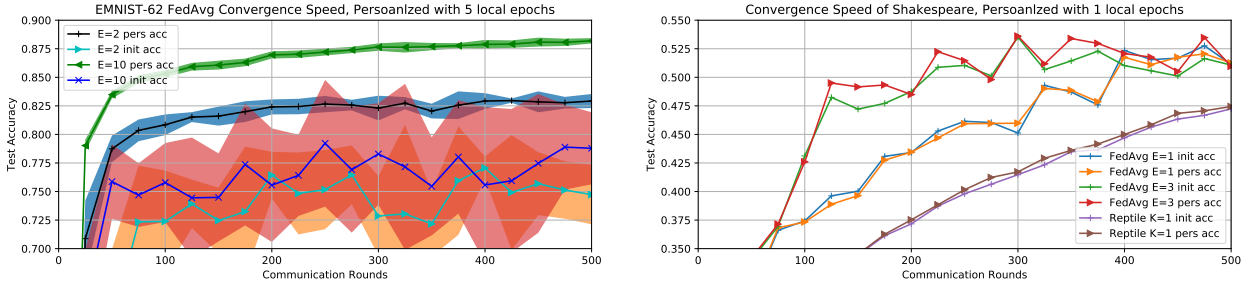


Figure 6.2: Training Convergence on EMNIST-62 (left) and Shakespeare (right).

First, the personalized accuracy converges significantly higher than the initial accuracy. This clearly validates the EMNIST-62 as an interesting simulated dataset to study Federated Learning, with significantly non-i.i.d. data available to each client.

Second, it provides empirical support to the claim made in Section 6.2, that *Federated Averaging is already a Meta Learning algorithm.* The personalized accuracy not only converges faster and higher, but the results are also of much smaller variance than those of the initial accuracy.

Third, the personalized accuracy after training with $E = 10$ is significantly higher than the personalized accuracy after training with $E = 2$. This is despite the fact that the gap in the initial accuracy between these two variants is somewhat smaller. Moreover, the variance of personalized accuracy is near 3-times smaller for training with $E = 10$, compared to $E = 2$, despite the variance of the initial accuracy being smaller for the $E = 2$ case. This supports

the insight from Equation 6.5, that *Federated Averaging with more gradient steps locally should emphasize the personalized accuracy* more, potentially at the cost of initial accuracy.

Finally, this challenges the objectives in the existing literature focusing on the setting of Federated Learning. FedAvg was presented as optimizing the performance of a shared, global model, while in fact it might achieve this only as a necessary partial step towards optimizing the personalized performance. We argue that *in the presence of non-i.i.d. data available to different clients, the objective of Federated Learning should also be the personalized performance*. Consequently, the recommendations that in order to stabilize the convergence, one might decrease the number of local epochs or the local learning rate [82, 123], are in some scenarios misguided. In this experiment, even though the initial accuracy is very noisy and roughly constant at a suboptimal level, the personalized accuracy keeps increasing.

EMNIST-62 5 clients	Initial Acc	Personalized Acc	Epochs to 0.8 (init/pers)
FedAvg E=2	0.7473(0.0260)	0.8292 (0.0061)	310.0/63.6
FedAvg E=5	**0.8028 (0.0512)**	0.8712 (0.0049)	**111.1**/33.9
FedAvg E=10	0.7879(0.0316)	**0.8820 (0.0023)**	137.5/**30.0**
FedAvg E=20	0.7430(0.0309)	0.8782 (0.0021)	152.5/32.2
EMNIST-62 20 clients			
FedAvg E=2	0.8403 (0.0173)	0.8957 (0.0011)	82.5/50.0
FedAvg E=5	0.8471 (0.0084)	**0.9057 (0.0017)**	**65.6**/31.25
FedAvg E=10	**0.8480 (0.0036)**	0.9032 (0.0017)	68.7/**25.0**
FedAvg E=20	0.8391 (0.0081)	0.8953 (0.0022)	82.1/46.4

Table 6.1: EMNIST-62 performance for 5 and 20 clients per communication round.

To evaluate the convergence speed more closely, Table 6.1 measures the accuracies after 500 rounds of training and the average number of communication rounds where the initial and personalized accuracy first time reaches 80%. While Figure 6.2 shows results using 5 clients per round, the table below also shows the same experiment with 20 clients per round,

which in general provides even better and more stable personalized accuracy. The common pattern is that increasing E initially helps, until a certain threshold. From this experiment, E in the range of $5 - 10$ seems to be the best.

We conduct a similar experiment with the Shakespeare data. The result is in Figure 6.2, right, but does not provide any interesting insights, as the personalized performance shows only a small positive improvement. We conjecture that this is due to the nature of the objective – even though the data is non-i.i.d., the next-character prediction is mostly focused on a local structure of the language in general, and is similar across all users.[3] We thus do not study this problem further in this chapter. It is likely that for a next-word prediction task, personalization would make a more significant difference.

6.3.2 Behavior of Personalization

In this section, we study the ability of models to personalize more closely. In particular, we look at the personalization performance as a function of the number of local epochs spent personalizing and the effect of both the local personalization optimizer and the optimizer used to train the initial model.

In Figure 6.3, left, we study the performance of three different models, personalized using different local optimizers. The models we test here are: one is randomly chosen from the models trained with $E = 10$ in the previous section, with initial accuracy of 74.41%. The other two models are the results of fine-tuning that specific model with $Reptile(1)$ and $Reptile(10)$ and Adam as the server optimizer for further 200 communication rounds, again using 5 clients per round. For all three models, we show the results of personalization using Adam with default parameters and using SGD with learning rate of 0.02 and batch size of 100.

In all three cases, Adam produces reasonable personalized result, but inferior to the result of SGD. We tried SGD with a range of other learning rates, and in all cases we observed

[3]However, we did not try to replace the default model suggested in TensorFlow Federated.

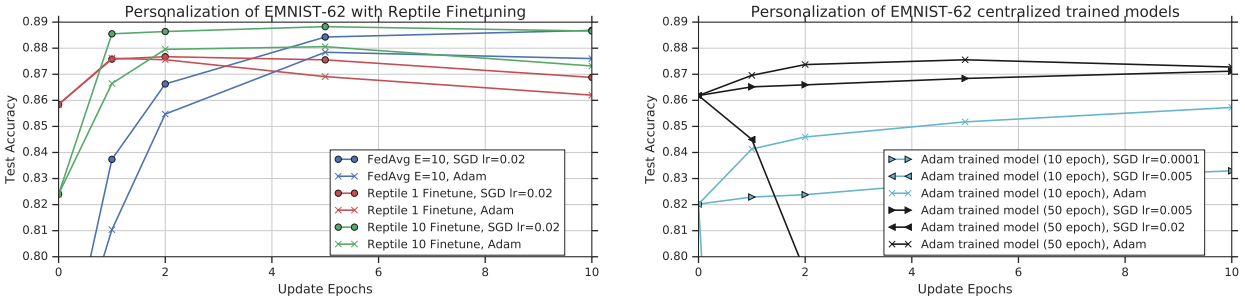


Figure 6.3: Personalized accuracy as a function of local update epochs. Federated initial models (left), initial models trained by centralizing data and using default Adam optimizer (right).

this value to work the best. Note that this is the same local optimizer that was used during training and fine tuning, which is similar to the MAML works, where the same algorithm is user for meta-training and meta-testing.

The effect of fine-tuning is very encouraging. Using *Reptile*(10), we get a model with better initial accuracy, which also gets to a slightly improved personalized accuracy. Most importantly, it gets to roughly the same performance for a wide range of local personalization epochs. Such property is of immense value for practical deployment, as only limited tools for model quality validation can be available on the devices where personalization happens. Using *Reptile*(1) significantly improves the initial accuracy, but the personalized accuracy actually drops! Even though the Equation 6.5 suggests that this algorithm should not take into account the personalized performance, it is not clear that such result should be intuitively expected – it does not mean it actively suppresses personalization, either, and it is only fine-tuning of a model already trained for the personalized performance. We note that this can be seen as an analogous observation to those of [87], where *Reptile*(1) yields a model with clearly inferior personalized performance.

Motivated by this somewhat surprising result, we ask the following question: *If the training data were available in a central location, and an initial model was trained using standard optimization algorithms, how would the personalization change?* We train such "centralized

initial model" using Adam with default settings and evaluate the personalization performance of a model snapshot after 10 and 50 training epochs. These two models have similar initial accuracies as the two models fine-tuned with *Reptile*(10) and *Reptile*(1), respectively. The results are in Figure 6.3, right.

The main message is that it is significantly harder to personalize these centralized initial models. Using SGD with learning rate of 0.02 is not good – a significantly smaller learning rate is needed to prevent the models from diverging, and then the personalization improvement is relatively small. In this case, using Adam does provide a better result, but still below the fine tuning performance in Figure 6.3, left. It is worth noting that the personalized performance of this converged model is similar to that of the model we get after fine tuning with *Reptile*(1), although using different personalization optimizer. At the moment, we are unable to suggest a sound explanation for this similarity.

We recommended using Adam as the server optimizer for fine-tuning, and here we only presented such results. We did try different optimizers, and found them to yield worse results with higher variance, especially in terms of the initial accuracy. We see that all of the optimizers can deliver higher initial accuracy at the cost of slightly lower personalized accuracy[4].

To strengthen the above observations, we look at the distribution of the initial and personalized accuracies of fine-tuned models over multiple experiments. In Figure 6.4, we look at the same three models as in Figure 6.3. It is clear that the initial model has a large variance in the initial accuracy, the results are in the range of 12%, but the personalized accuracies are only within the range of 1%. We chose one model, as indicated by the arrows[5], to be fine-tuned with *Reptile*(10) and *Reptile*(1). In both cases, fine-tuning results in more consistent results in both the initial and personalized accuracy. Moreover, the best-personalized accuracy with *Reptile*(1) is worse than the worst personalized accuracy with *Reptile*(10).

In Appendix **??**, we look at a similar visualization on a per-client basis for a given model.

[4]full experimental results refer to [56] appendix

[5]The same model was used as the starting point for fine-tuning in Figure 6.3.

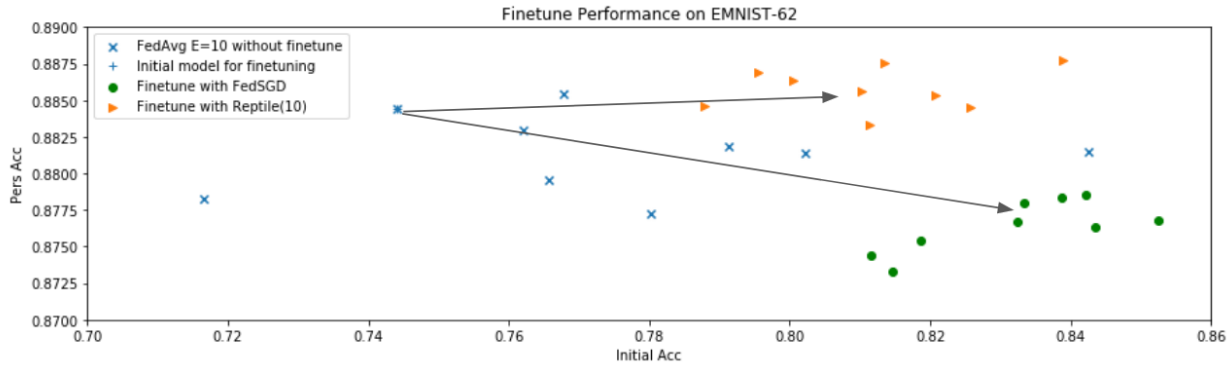


Figure 6.4: Distribution of initial and personalized accuracies of fine tuned models.

Studying this distribution is of great importance, as in practical deployment, even a small degradation in a user's experience might incur a disproportionate cost, relative to the benefit of a comparable improvement in the model quality. We do not study this question deeper in this work, though.

	Initial Acc	Personalized Acc
Reptile(1) Finetuned test clients	0.8320 (0.0133)	0.8764 (0.0017)
Reptile(1) Finetuned train clients	0.8577 (0.0019)	0.8927 (0.0015)
Reptile(10) Finetuned test clients	0.8116 (0.0148)	0.8858 (0.0014)
Reptile(10) Finetuned train clients	0.8612 (0.0020)	0.9028(0.0009)

Table 6.2: Test performance on clients seen and unseen during FL-training

Finally, we look at the performance of fine-tuned models discussed above on both train and test clients. Table 6.2 shows that if we only looked at the initial accuracy, basic ML principles would suggest the *Reptile*(10) models are over-fitting, due to the larger gap between the train and test clients. However, the personalized accuracy tells a different story - the gap is roughly the same for both model types, and for both train and test clients, *Reptile*(10)

provides significantly better personalized accuracy, suggesting we need a novel way to predict the generalization of personalized models.

6.4 Discussion

It this work, we argue that in the context of Federated Learning, the accuracy of the global model after personalization should be of much greater interest than it has been. Investigation of the topic reveals close similarities between the fields of Federated Learning and Model Agnostic Meta Learning, and raises new questions for these areas, as well as for the broader Machine Learning community.

Challenges for Federated Learning. Framing papers in the area of Federated Learning [82, 65, 74], formulate the objective as training of a shared global model, based on a decentralized data storage where each node / client has access to a non-i.i.d sample from the overall distribution. The objective is identical to one the broader ML community would optimize for, had all the data been available in a centralized location.

We argue that in this setting, the primary objective should be the adaptation to the statistical heterogeneity present at different data nodes, and demonstrate that the popular FL algorithm, Federated Averaging, does in fact optimize the personalized performance, and while doing so, also improves the performance of the global model. Experiments we perform demonstrate that the algorithm used to train the model has a major influence on its capacity to personalize. Moreover, solely optimizing the accuracy of the global model tends to have negative impact on its capacity to personalize, which further questions the correctness of the commonly presented objectives of Federated Learning.

Challenges for Model Agnostic Meta Learning. The objectives in the Model Agnostic Meta Learning literature are usually only the model performance after adaptation to given task [33]. In this work, we present the setting of Federated Learning as a good source of practical applications for MAML algorithms. However, to have impact in FL, these methods

need to also consider the performance of the initial model,[6] as in practice there will be many clients without data available for personalization. In addition, the connectivity constraints in a production deployment emphasize the importance of fast convergence in terms of number of communication rounds. We suggest these objectives become the subject of MAML works, in addition to the performance after adaptation, and to consider the datasets with a natural user/client structure being established for Federated Learning [15] as the source of experiments for supervised learning.

Challenges for broader Machine Learning. The empirical evaluation in this work raises a number of questions of relevance to Machine Learning research in general. In particular, Figure 6.3 clearly shows that models with similar initial accuracy can have very different capacity to personalize to a task of the same type as it was trained on. This observation raises obvious questions for which we currently cannot provide an answer. How does the training algorithm impact personalization ability of the trained model? Is there something we can measure that will predict the adaptability of the model? Is it something we can directly optimize for, potentially leading to novel optimization methods? These questions can relate to a gap highlighted in Table 6.2. While the common measures could suggest the global model is overfitting the training data, this is not true of the personalized model.

Transfer Learning is another technique for which our result could inspire a novel solution. It is very common for machine learning practitioners to take a trained model from the research community, replace the final layer with a different output class of interest, and retrain for the new task [88]. We conjecture that the algorithms proposed in the FL and MAML communities could yield base models for which this kind of domain adaptation would yield better results.

Finally, we believe that a systematic analysis of optimization algorithms of the inner-outer structure presented in Algorithm 1 could provide novel insights into the connections between optimization and generalization. Apart from the FL and MAML algorithms, [132] recently proposed a method that can be interpreted as an outer optimizer in the general algorithm,

[6]We recognize that some MAML datasets [69] do not admit a good notion of initial accuracy.

which improves the stability of various existing optimization methods used as the inner optimizer.

Chapter 7

CONCLUSION

In this thesis, we study three types of canonical channel coding problems via deep learning. We show that deep learning can achieve optimal performance on AWGN settings against closed-form solutions and outperform other settings. Moreover, deep learning enables other paradigms such as meta learning and federated learning, further demonstrate the versatility of neural channel codes. We hope this thesis could serve as the starting point to promote applying deep learning for the next generation communication systems.

BIBLIOGRAPHY

[1] Maruan Al-Shedivat, Liam Li, Eric Xing, and Ameet Talwalkar. On data efficiency of meta-learning. *arXiv preprint arXiv:2102.00127*, 2021.

[2] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml. *arXiv preprint arXiv:1810.09502*, 2018.

[3] Fayccal Ait Aoudia and Jakob Hoydis. End-to-end learning of communications systems without a channel model. In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, pages 298–303. IEEE, 2018.

[4] Fayccal Ait Aoudia and Jakob Hoydis. Model-free training of end-to-end communication systems. *IEEE Journal on Selected Areas in Communications*, 37(11):2503–2516, 2019.

[5] Erdal Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Transactions on information Theory*, 55(7):3051–3073, 2009.

[6] Manoj Ghuhan Arivazhagan, Vinay Aggarwal, Aaditya Kumar Singh, and Sunav Choudhary. Federated learning with personalization layers. *arXiv preprint arXiv:1912.00818*, 2019.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[8] Lalit Bahl, John Cocke, Frederick Jelinek, and Josef Raviv. Optimal decoding of linear codes for minimizing symbol error rate (corresp.). *IEEE Transactions on information theory*, 20(2):284–287, 1974.

[9] Francoise Beaufays, Khe Chai Sim, and Petr Zadrazil. An investigation into on-device personalization of end-to-end automatic speech recognition models. In *Interspeech*, 2019.

[10] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[11] Claude Berrou, Alain Glavieux, and Punya Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. 1. In *Proceedings of ICC'93-IEEE International Conference on Communications*, volume 2, pages 1064–1070. IEEE, 1993.

[12] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, H Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. 2019.

[13] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191. ACM, 2017.

[14] Sebastian Caldas, Jakub Konečný, H Brendan McMahan, and Ameet Talwalkar. Expanding the reach of federated learning by reducing client resource requirements. *arXiv preprint arXiv:1812.07210*, 2018.

[15] Sebastian Caldas, Peter Wu, Tian Li, Jakub Konečný, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097*, 2018.

[16] Sebastian Cammerer, Tobias Gruber, Jakob Hoydis, and Stephan Ten Brink. Scaling deep learning-based decoding of polar codes via partitioning. In *GLOBECOM 2017-2017 IEEE global communications conference*, pages 1–6. IEEE, 2017.

[17] Zachary Chance and David J Love. Concatenated coding for the awgn channel with noisy feedback. *IEEE Transactions on Information Theory*, 57(10):6633–6649, 2011.

[18] Fei Chen, Zhenhua Dong, Zhenguo Li, and Xiuqiang He. Federated meta-learning for recommendation. *arXiv preprint arXiv:1802.07876*, 2018.

[19] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. *arXiv preprint arXiv:1606.03657*, 2016.

[20] Kristy Choi, Kedar Tatwawadi, Aditya Grover, Tsachy Weissman, and Stefano Ermon. Neural joint source-channel coding. In *International Conference on Machine Learning*, pages 1182–1192. PMLR, 2019.

[21] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[22] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 215–223. JMLR Workshop and Conference Proceedings, 2011.

[23] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[24] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[25] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[26] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[27] Li Deng, Michael L Seltzer, Dong Yu, Alex Acero, Abdel-rahman Mohamed, and Geoff Hinton. Binary coding of speech spectrograms using a deep auto-encoder. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[29] Sebastian Dörner, Sebastian Cammerer, Jakob Hoydis, and Stephan Ten Brink. Deep learning based communication over the air. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):132–143, 2017.

[30] Tolga M Duman and Masoud Salehi. On optimal power allocation for turbo codes. In *Proceedings of IEEE International Symposium on Information Theory*, page 104. IEEE, 1997.

[31] Alexander Felix, Sebastian Cammerer, Sebastian Dörner, Jakob Hoydis, and Stephan Ten Brink. Ofdm-autoencoder for end-to-end learning of communications systems. In

2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC), pages 1–5. IEEE, 2018.

[32] Chelsea Finn. *Learning to learn with gradients*. PhD thesis, UC Berkeley, 2018.

[33] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017.

[34] Chelsea Finn and Sergey Levine. Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. *arXiv preprint arXiv:1710.11622*, 2017.

[35] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[36] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.

[37] Tobias Gruber, Sebastian Cammerer, Jakob Hoydis, and Stephan ten Brink. On deep learning-based channel decoding. In *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2017.

[38] Joachim Hagenauer. The turbo principle: Tutorial introduction and state of the art. In *Proc. International Symposium on Turbo Codes and Related Topics*, pages 1–11, 1997.

[39] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[40] Andrew Hard, Kanishka Rao, Rajiv Mathews, Franccoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*, 2018.

[41] Hengtao He, Chao-Kai Wen, Shi Jin, and Geoffrey Ye Li. Model-driven deep learning for mimo detection. *IEEE Transactions on Signal Processing*, 68:1702–1715, 2020.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[43] Yunfeng He, Jing Zhang, Chao-Kai Wen, and Shi Jin. Turbonet: A model-driven dnn decoder based on max-log-map algorithm for turbo code. In *2019 IEEE VTS Asia Pacific Wireless Communications Symposium (APWCS)*, pages 1–5. IEEE, 2019.

[44] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[45] https://openmined.org. Pysyft. *https://github.com/OpenMined/PySyft*, 2019.

[46] Lingchen Huang, Huazi Zhang, Rong Li, Yiqun Ge, and Jun Wang. Ai coding: Learning to construct error correction codes. *IEEE Transactions on Communications*, 68(1):26–39, 2019.

[47] Lingchen Huang, Huazi Zhang, Rong Li, Yiqun Ge, and Jun Wang. Reinforcement learning for nested polar code construction. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[48] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Proceedings of the 30th international conference on neural information processing systems*, pages 4114–4122. Citeseer, 2016.

[49] Alex Ingerman and Krzys Ostrowski. Introducing tensorflow federated, 2019.

[50] Mohammad Vahid Jamali, Xiyang Liu, Ashok Vardhan Makkuva, Hessam Mahdavifar, Sewoong Oh, and Pramod Viswanath. Reed-muller subcodes: Machine learning-aided design of efficient soft recursive decoding. *arXiv preprint arXiv:2102.01671*, 2021.

[51] Yihan Jiang, Sreeram Kannan, Hyeji Kim, Sewoong Oh, Himanshu Asnani, and Pramod Viswanath. Deepturbo: Deep turbo decoder. In *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5. IEEE, 2019.

[52] Yihan Jiang, Hyeji Kim, Himanshu Asnani, and Sreeram Kannan. Mind: Model independent neural decoder. In *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5. IEEE, 2019.

[53] Yihan Jiang, Hyeji Kim, Himanshu Asnani, Sreeram Kannan, Sewoong Oh, and Pramod Viswanath. Turbo autoencoder: Deep learning based channel codes for point-to-point communication channels. In *Advances in Neural Information Processing Systems*, pages 2754–2764, 2019.

[54] Yihan Jiang, Hyeji Kim, Himanshu Asnani, Sreeram Kannan, Sewoong Oh, and Pramod Viswanath. Joint channel coding and modulation via deep learning. In *2020 IEEE 21st International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5. IEEE, 2020.

[55] Yihan Jiang, Hyeji Kim, Himanshu Asnani, Sreeram Kannan, Sewoong Oh, and Pramod Viswanath. Learn codes: Inventing low-latency codes via recurrent neural networks. *IEEE Journal on Selected Areas in Information Theory*, 1(1):207–216, 2020.

[56] Yihan Jiang, Jakub Konečnỳ, Keith Rush, and Sreeram Kannan. Improving federated learning personalization via model agnostic meta learning. *NeurIPS 2019 FL workshop, arXiv preprint arXiv:1909.12488*, 2019.

[57] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[58] Chloe Kiddon Hubert Eichner Franccoise Beaufays Daniel Ramage Kangkang Wang, Rajiv Mathews. Federated evaluation of language model personalization. *Internal abstract: not published yet*, 2019.

[59] Mikhail Khodak, Maria Florina-Balcan, and Ameet Talwalkar. Adaptive gradient-based meta-learning methods. *arXiv preprint arXiv:1906.02717*, 2019.

[60] Hyeji Kim, Yihan Jiang, Sreeram Kannan, Sewoong Oh, and Pramod Viswanath. Deepcode: feedback codes via deep learning. *IEEE Journal on Selected Areas in Information Theory*, 1(1):194–206, 2020.

[61] Hyeji Kim, Yihan Jiang, Ranvir Rana, Sreeram Kannan, Sewoong Oh, and Pramod Viswanath. Communication algorithms via deep learning. *Sixth International Conference on Learning Representations (ICLR)*, 2018.

[62] Young-Han Kim, Amos Lapidoth, and Tsachy Weissman. The gaussian channel with noisy feedback. In *2007 IEEE International Symposium on Information Theory*, pages 1416–1420. IEEE, 2007.

[63] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[64] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2, 2015.

[65] Jakub Konečný, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527*, 2016.

[66] Jakub Konečný, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[67] Alex Krizhevsky and Geoffrey E Hinton. Using very deep autoencoders for content-based image retrieval. In *ESANN*, volume 1, page 2. Citeseer, 2011.

[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[69] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[70] Amos Lapidoth. Nearest neighbor decoding for additive non-gaussian noise channels. *IEEE Transactions on Information Theory*, 42(5):1520–1529, 1996.

[71] Quoc Le, Alexandre Karpenko, Jiquan Ngiam, and Andrew Ng. Ica with reconstruction cost for efficient overcomplete feature learning. *Advances in neural information processing systems*, 24:1017–1025, 2011.

[72] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[73] Junyi Li, Xinzhou Wu, and Rajiv Laroia. *OFDMA mobile broadband communications: A systems approach*. Cambridge University Press, 2013.

[74] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *arXiv preprint arXiv:1908.07873*, 2019.

[75] Xiang Li, Kaixuan Huang, Wenhao Yang, Shusen Wang, and Zhihua Zhang. On the convergence of fedavg on non-iid data. *arXiv preprint arXiv:1907.02189*, 2019.

[76] Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. Meta-sgd: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835*, 2017.

[77] Fei Liang, Cong Shen, and Feng Wu. An iterative bp-cnn architecture for channel decoding. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):144–159, 2018.

[78] Paul Pu Liang, Terrance Liu, Liu Ziyin, Nicholas B Allen, Randy P Auerbach, David Brent, Ruslan Salakhutdinov, and Louis-Philippe Morency. Think locally, act globally: Federated learning with local and global representations. *arXiv preprint arXiv:2001.01523*, 2020.

[79] David JC MacKay and Radford M Neal. Near shannon limit performance of low density parity check codes. *Electronics letters*, 32(18):1645–1646, 1996.

[80] Alireza Makhzani and Brendan Frey. K-sparse autoencoders. *arXiv preprint arXiv:1312.5663*, 2013.

[81] Yishay Mansour, Mehryar Mohri, Jae Ro, and Ananda Theertha Suresh. Three approaches for personalization with applications to federated learning. *arXiv preprint arXiv:2002.10619*, 2020.

[82] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282, 2017.

[83] H Brendan McMahan and Daniel Ramage. Federated learning: Collaborative machine learning without centralized training data, 2017. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[84] H Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models. In *International Conference on Learning Representations*, 2018.

[85] Eliya Nachmani, Yair Be'ery, and David Burshtein. Learning to decode linear codes using deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 341–346. IEEE, 2016.

[86] Eliya Nachmani, Elad Marciano, Loren Lugosch, Warren J Gross, David Burshtein, and Yair Be?ery. Deep learning methods for improved decoding of linear codes. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):119–131, 2018.

[87] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[88] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.

[89] Timothy O?shea and Jakob Hoydis. An introduction to deep learning for the physical layer. *IEEE Transactions on Cognitive Communications and Networking*, 3(4):563–575, 2017.

[90] Timothy J O'Shea, Tugba Erpek, and T Charles Clancy. Deep learning based mimo communications. *arXiv preprint arXiv:1707.07980*, 2017.

[91] Timothy J O'Shea, Kiran Karra, and T Charles Clancy. Learning to communicate: Channel auto-encoders, domain specific regularizers, and attention. In *2016 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 223–228. IEEE, 2016.

[92] Yaniv Ovadia, Emily Fertig, Jie Ren, Zachary Nado, David Sculley, Sebastian Nowozin, Joshua V Dillon, Balaji Lakshminarayanan, and Jasper Snoek. Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift. *arXiv preprint arXiv:1906.02530*, 2019.

[93] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11):1–4, 2015.

[94] Yury Polyanskiy, H Vincent Poor, and Sergio Verdú. Channel coding rate in the finite blocklength regime. *IEEE Transactions on Information Theory*, 56(5):2307–2359, 2010.

[95] Hanghang Qi, David Malone, and Vijay Subramanian. Does every bit need the same power? an investigation on unequal power allocation for irregular ldpc codes. In *2009 International Conference on Wireless Communications & Signal Processing*, pages 1–5. IEEE, 2009.

[96] Aravind Rajeswaran, Chelsea Finn, Sham Kakade, and Sergey Levine. Meta-learning with implicit gradients. *arXiv preprint arXiv:1909.04630*, 2019.

[97] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[98] Thomas J Richardson and Rüdiger L Urbanke. The capacity of low-density parity-check codes under message-passing decoding. *IEEE Transactions on information theory*, 47(2):599–618, 2001.

[99] Tom Richardson and Ruediger Urbanke. *Modern coding theory*. Cambridge university press, 2008.

[100] Hamid R Sadjadpour, Neil JA Sloane, Masoud Salehi, and Gabriele Nebe. Interleaver design for turbo codes. *IEEE Journal on Selected Areas in Communications*, 19(5):831–837, 2001.

[101] Anant Sahai, Joshua Sanz, Vignesh Subramanian, Caryn Tran, and Kailas Vodrahalli. Learning to communicate with limited co-design. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 184–191. IEEE, 2019.

[102] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850. PMLR, 2016.

[103] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? *arXiv preprint arXiv:1805.11604*, 2018.

[104] Victor Garcia Satorras and Max Welling. Neural enhanced belief propagation on factor graphs. *arXiv preprint arXiv:2003.01998*, 2020.

[105] Murat Hüsnü Sazlı and Can Icsık. Neural network implementation of the bcjr algorithm. *Digital Signal Processing*, 17(1):353–359, 2007.

[106] J Schalkwijk and Thomas Kailath. A coding scheme for additive noise channels with feedback–i: No bandwidth constraint. *IEEE Transactions on Information Theory*, 12(2):172–182, 1966.

[107] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[108] Claude Shannon. The zero error capacity of a noisy channel. *IRE Transactions on Information Theory*, 2(3):8–19, 1956.

[109] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[110] Nir Shlezinger, Nariman Farsad, Yonina C Eldar, and Andrea J Goldsmith. Viterbinet: A deep learning based viterbi algorithm for symbol detection. *IEEE Transactions on Wireless Communications*, 19(5):3319–3331, 2020.

[111] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[112] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[113] Karan Singhal, Hakim Sidahmed, Zachary Garrett, Shanshan Wu, Keith Rush, and Sushant Prakash. Federated reconstruction: Partially local federated learning. *arXiv preprint arXiv:2102.03448*, 2021.

[114] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*, pages 4077–4087, 2017.

[115] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[116] Ido Tal and Alexander Vardy. List decoding of polar codes. *IEEE Transactions on Information Theory*, 61(5):2213–2226, 2015.

[117] Eleni Triantafillou, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Kelvin Xu, Ross Goroshin, Carles Gelada, Kevin Swersky, Pierre-Antoine Manzagol, and Hugo Larochelle. Meta-dataset: A dataset of datasets for learning to learn from few examples. *arXiv preprint arXiv:1903.03096*, 2019.

[118] David Tse and Pramod Viswanath. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[119] Joaquin Vanschoren. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548*, 2018.

[120] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[121] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in neural information processing systems*, pages 3630–3638, 2016.

[122] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.

[123] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. *arXiv preprint arXiv:1810.08313*, 2018.

[124] Xiao-An Wang and Stephen B Wicker. An artificial neural net viterbi decoder. *IEEE Transactions on communications*, 44(2):165–171, 1996.

[125] Pete Warden and Daniel Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. " O'Reilly Media, Inc.", 2019.

[126] Lilian Weng. Meta-learning: Learning to learn fast. *lilianweng.github.io/lil-log*, 2018.

[127] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[128] Weihong Xu, Zhizhen Wu, Yeong-Luh Ueng, Xiaohu You, and Chuan Zhang. Improved polar decoder based on deep learning. In *2017 IEEE International workshop on signal processing systems (SiPS)*, pages 1–6. IEEE, 2017.

[129] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):12, 2019.

[130] Hao Ye, Le Liang, Geoffrey Ye Li, and Biing-Hwang Juang. Deep learning-based end-to-end wireless communication systems with conditional gans as unknown channels. *IEEE Transactions on Wireless Communications*, 19(5):3133–3143, 2020.

[131] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of cnn and rnn for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.

[132] Michael R Zhang, James Lucas, Geoffrey Hinton, and Jimmy Ba. Lookahead optimizer: k steps forward, 1 step back. *arXiv preprint arXiv:1907.08610*, 2019.

[133] Banghua Zhu, Jintao Wang, Longzhuang He, and Jian Song. Joint transceiver optimization for wireless communication phy using neural network. *IEEE Journal on Selected Areas in Communications*, 37(6):1364–1373, 2019.

[134] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.