الجمهورية الجزائرية الديمقراطية الشعبية

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
وزارة التعليم العالي والبحث العلمي

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Electronique

End-of-Studies Project Dissertation

for obtaining the State Engineer's degree in Electronics

Machine Learning Techniques for Turbo Decoding in Wireless Communication Systems

---

Mehdi BENKIRAT & **Mehdi Chames Eddinne LAYES**

Under the supervision of: **Mr. Mohamed Oussaid TAGHI.** ENP, Algiers.

Presented and publicly defended on July $1st, 2024$ to the members of the jury:

President:	Mr. Cherif LARBES	Prof	ENP, Algiers
Supervisor:	Mr. Oussaid Mohamed TAGH	MAA	ENP, Algiers
Examinator:	Mrs. Nesrine BOUADJENEK	Dr	ENP, Algiers

ENP 2024
10 Avenue des Frères OUDEK, Hassen Badi, BP. 182, El Harrach 16200 Alger Algérie.
www.enp.edu.dz

الجمهورية الجزائرية الديمقراطية الشعبية

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
وزارة التعليم العالي والبحث العلمي

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Electronique

End-of-Studies Project Dissertation

for obtaining the State Engineer's degree in Electronics

Machine Learning Techniques for Turbo Decoding in Wireless Communication Systems

---

Mehdi BENKIRAT & **Mehdi Chames Eddinne LAYES**

Under the supervision of: **Mr. Oussaid Mohamed TAGHI.** ENP, Algiers.

Presented and publicly defended on July $1st, 2024$ to the members of the jury:

President:	Mr. Cherif LARBES	Prof	ENP, Algiers
Supervisor:	Mr. Oussaid Mohamed TAGHI	MAA	ENP, Algiers
Examinator:	Mrs. Nesrine BOUADJENEK	Dr	ENP, Algiers

ENP 2024
10 Avenue des Frères OUDEK, Hassen Badi, BP. 182, El Harrach 16200 Alger Algérie.
www.enp.edu.dz

الجمهورية الجزائرية الديمقراطية الشعبية

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
وزارة التعليم العالي والبحث العلمي

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Électronique

Mémoire de projet de fin d'études

Pour l'obtention du diplôme d'Ingénieur d'État en Électronique

---

Techniques d'Apprentissage Automatique pour le Turbo Décodage dans les Systèmes de Communication Sans fil

---

Mehdi BENKIRAT & **Mehdi Chames Eddinne LAYES**

Sous la direction de : **M. Oussaid Mohamed TAGHI** ENP

Présenté et soutenu publiquement le 01/07/2024 auprès des membres du jury :

Président:	M. Cherif LARBES	Prof	ENP, Alger
Promoteur:	M. Oussaid Mohamed TAGHI	MAA	ENP, Alger
Examinatrice:	Mme. Nesrine BOUADJENEK	Dr	ENP, Alger

ENP 2024
10 Avenue des Frères OUDEK, Hassen Badi, BP. 182, El Harrach 16200 Alger Algérie.
www.enp.edu.dz

ملخص

تستكشف هذه الدراسة تقنيات مبتكرة للتعلم الآلي تهدف إلى تحسين فك الترميز التوربيني في الاتصالات اللاسلكية. غالبًا ما تواجه أجهزة فك الترميز التوربيني التقليدية صعوبات مثل القابلية للتأثر بالتشويش النبضي وارتفاع معدلات الخطأ عند نسب الإشارة إلى التشويش **SNR** العالية. لمعالجة هذه المشكلات، تفحص الدراسة نماذج الانتباه من نوع تسلسل إلى تسلسل وهياكل المحولات ، وتكييفها لفك الترميز التوربيني لتحسين الدقة والموثوقية في ظروف تشويش مختلفة. تشمل الأبحاث مناقشات أساسية حول الأكواد الالتفافية والتوربينية، باستخدام خوارزمية **SOVA** ، ومراجعات للشبكات العصبية في تطبيقات فك الترميز التوربيني، وتقديم النماذج الفعالة **TurboAttention** و **TurboTransformer** . تُظهر هذه النماذج نتائج واعدة من حيث معدل الخطأ **BER** عبر مجموعة واسعة من قيم **SNR** ، مع أداء مشجع تم ملاحظته في اختبارات مادية.

الكلمات المفتاحية: ، الأكواد التوربينية ، فك الترميز التوربيني ، نسب الإشارة إلى التشويش **SNR** ، التعلم الآلي ، نماذج الانتباه، المحول ، **SOVA** ، معدل الخطأ **BER**

Résumé

Cette étude explore des techniques d'apprentissage automatique visant à améliorer le décodage de turbo codes dans la communication sans fil. Les décodeurs turbo traditionnels rencontrent des difficultés telles que la susceptibilité au bruit impulsif et des taux d'erreur élevés à des rapports signal/bruit (SNR) élevés. Pour résoudre ces problèmes, l'étude examine des modèles d'attention Séquence-à-Séquence et des architectures Transformer, en les adaptant au décodage turbo pour potentiellement améliorer la précision et la robustesse dans diverses conditions de bruit de canal. L'étude inclut des discussions fondamentales sur les turbo codes, des simulations utilisant l'algorithme SOVA, des revues de réseaux neuronaux dans les applications de décodage turbo, et introduit les modèles *TurboAttention* et *TurboTransformer*. Ces modèles montrent des résultats prometteurs en termes de taux d'erreur binaire sur une large gamme de valeurs de SNR, avec des performances encourageantes observées lors des tests d'inférence matérielle.

Mots-clés : Codes turbo, Décodage turbo, SNR, Apprentissage automatique, Modèles d'attention, Transformer, SOVA, Taux d'erreur (BER)

Abstract

This study investigates machine-learning techniques aimed at enhancing turbo decoding in wireless communication. Traditional turbo decoders often struggle with challenges such as susceptibility to burst noise and high error rates at high Signal-to-Noise Ratios (SNRs). To tackle these issues, the study explores Sequence-to-Sequence attention models and Transformer architectures, adapting them for turbo decoding to potentially enhance accuracy and robustness across various channel noise conditions. The research includes foundational discussions on convolutional and turbo codes, simulations using the SOVA algorithm, reviews of neural networks in turbo decoding applications, and introduces the effective models *TurboAttention* and *TurboTransformer*. These models demonstrate promising results in terms of Bit Error Rate (BER) across a wide range of SNR values, with encouraging performance observed in hardware inference tests.

Keywords: Turbo codes, Turbo decoding, SNR, Machine learning, Attention models, Transformer, SOVA, Bit Error Rate (BER)

Dedication

At this moment, I find myself in a state of mind where I could leave this page blank. Yet, I have so much to tell, so much to give, and I have learned that it is always better to speak. Here I am, in this office, writing the final piece, wrapping up a chapter with the profound satisfaction of having given my all. With the promise of more to come, I eagerly anticipate continuing this life, this experience, and this journey.

To my parents, your education, care, and love have made me who I am today. I hope to pursue this life knowing that I have you by my side. To Mamina for being Mamina, and to Papou for being Papou.

To my dear sister, my only sister and the only one I could hope for. You surpass every expectation. You bring joy to my days, and even though I do not express it often, please know that your presence is invaluable. Be you and keep it that way.

To Fares, knowing you has shaped my life, and I'd love to keep it that way.

To Lotfi, knowing you is a never-ending blessing, and I'd love to keep it that way.

To Lilia, knowing you means the world, and I'd love to keep it that way.

To Mehdi, not me, knowing you and working with you is a pleasure. And to my classmates and all the friendships I made along this journey, even if your names are not mentioned, please know that you have a place in my heart.

Note to myself: love fully, appreciate the moments life has given you and take care of the people around you.

Mehdi.

Dedication

Water only obeys itself, just like my tears as I write this. Five years filled with work, joy, friends, battles, and travels are ending here. This is the day you always asked me about, Grandma ***Yami****; I am graduating, yet you are not here to witness it. Such is life. From childhood, you always loved me, soulful and pure-hearted. You taught me contentment in life, one of the most special principles I gained from you. May ALLAH The Most Merciful accept you with his mercy.*

To my lovely parents, your unconditional support has shaped me into this Mehdi Chames Eddinne; your love has always been my primary motivation in life. Mother, you molded me into the man I am. You made me someone who can feel and sense others, just as you do. Just seeing you fills me with life. Father, your strong principles have always made me proud to be your son. Since I was a little boy, talking to you about everything opened my eyes and given me an open view of this world. Our conversations have never ceased to interest me.

To my brother Mohamed, I hope you become the person I've always envisioned, the big brother I can always rely on and who can rely on me. I hope you find your true path in life.

To my sisters, being with you is always a source of joy and comfort. You have always listened to me and supported me in every possible way.

My dear family, may ALLAH protect you.

To Zaki, from a friend to a second Brother, I am forever grateful to have known you. Your motivated and initiative spirit has guided me throughout this journey and shaped the person I am today. Whether in my lowest moments or during my successes, you remain one of the most unique individuals I know. Thank you for always being there, Zak !

Mehdi ! Not me, to my partner in this work, stay well. Working with you has been another experience altogether. You are a special and talented individual, and I hope you find your path in life that brings out the best in you. What a good friend you are.

Friends are life's greatest blessings. To my ELN mates Abderezak, Baki, Said, Foutia, Nadjib and Mahmoud, the moments we've shared are unforgettable. I hope our bond remains strong because I never tire of being with you. I am grateful to have met you all. And of course, outside of my cohort, there have always been friends, good people with whom I've forgotten how we became so close. Imed, Said Guerzouma, and Rosso, your kindness and pure souls are exceptional. Thank you.

To my FamilIEEE, the journey we embarked on together, the adventure that taught and made me into the person I am today, was filled with laughter, hard work, and tears, ending with me knowing some of the most interesting people I've ever encountered.

It is a Laugh Tale; reaching its end makes you understand why you should both cry and laugh about it at the same time. That's how adventures should be. Now, I realize that the scale of our lives is negligible compared to the universe ALLAH created, so we should always be grateful for the ability to perceive the unseen and live a satisfying life with its little details.

Mehdi Chames Eddinne.

Acknowledgments

Above all, we thank God, the Almighty, for having given us the courage, patience, willpower and strength to face all the difficulties and obstacles that have stood in our way throughout this work.

First and foremost, we would like to extend our heartfelt appreciation to our supervisor, Mr. TAGHI Mohamed, for his continuous guidance and support throughout the entire journey of this thesis. His availability and advice played a crucial role in the successful completion of this work.

We extend our sincere thanks to the President of the jury, Mr. LARBES Cherif, and to the Examiner, Mrs. BOUADJENEK Nesrine, for accepting our work.

Mehdi and Mehdi.

Contents

List of Tables

List of Figures

List of Acronyms

Adam Adaptive Moment Estimation

AdamW Adaptive Moment Estimation with Weight Decay

AWGN Additive White Gaussian Noise

BCJR Bahl, Cocke, Jelinek, and Raviv

BER Bit Error Rate

BLER Block Error Rate

BPSK Binary Phase-Shift Keying

BSC Binary Symmetric Channel

FFNN Feed Forward Neural Network

GRU Gated Recurrent Unit

LSTM Long Short-Term Memory

LTE Long-Term Evolution

MAE Mean Average Error

OFDM Orthogonal Frequency-Division Multiplexing

QPP Quadratic Permutation Polynomial

RNN Recurrent Neural Network

RSC Recursive Systematic Convolutional

seq2seq Sequence-to-Sequence

SGD Stochastic Gradient Descent

SNR Signal-to-Noise Ratio

SOVA Soft-Output Viterbi Algorithm

TER Token Error Rate

General Introduction

Wireless communications necessitate ongoing development and enhancement to meet the demands for secure and reliable information transmission. Channel coding algorithms play a critical role in addressing the challenges posed by wireless communication channels, where multiple of sources of noise and artifacts can degrade the quality of transmitted signals and lose of information. By introducing redundancy into the transmitted messages, these algorithms significantly reduce the error rate at the receiver, thereby enhancing the overall integrity and reliability of the communication system.

Convolutional codes are widely used in practical communication systems for real-time error correction, converting entire data streams into single codewords. These codes rely on both current and past input bits, with the Viterbi algorithm being the primary decoding strategy. Advances in convolutional coding have led to turbo codes. Turbo codes, approach channel capacity by concatenating two convolutional codes with a random interleaver, achieving performance within 1 dB of channel capacity [1][2]. The random interleaver's structure contributes to efficient decoding while maintaining near-zero bit error rates.

Iterative turbo decoders often lack robustness and adaptability in certain channels, with burst noise significantly impacting performance. Traditional methods to enhance adaptability frequently fail under unexpected conditions [3] [4]. Turbo codes also exhibit an error floor at high SNRs, making them less suitable for high-reliability applications such as secure communications [5] [6]. Various techniques have been proposed to reduce the error floor, but with limited success. As a result, traditional turbo designs struggle to consistently achieve high reliability, robustness, adaptability, and low error floors [7].

Machine learning techniques hold significant promise for turbo decoding applications due to their diverse architectures and capabilities, which can address the limitations of conventional turbo decoders. By leveraging provided data, machine learning models can learn and identify patterns in erroneous data through iterative learning processes. This enables them to correct errors in transmitted messages more effectively than traditional methods, achieving higher accuracy across various configurations and communication channel conditions.

The objective of this project is to explore and implement novel machine learning approaches for turbo decoding, specifically focusing on sequence processing techniques used in Natural Language Processing (NLP). We will investigate the application of sequence-to-sequence attention models and transformer architectures to enhance turbo decoding performance. The goal is to leverage the advanced capabilities of these models to address the limitations of conventional turbo decoders, achieving higher accuracy and robustness in various communication channel conditions.

Outline

Chapter 1 focuses on laying down the theoretical groundwork of wireless communication. It comprehensively covers the stages of the communication channel and provides detailed insights into convolutional codes, turbo codes, and iterative algorithms used in turbo decoding. The chapter also includes simulation results using the Soft Input Soft Output (SOVA) algorithm for turbo decoding implemented in MATLAB.

Chapter 2 is divided into two parts. The first part explores the fundamentals of neural networks, sequence models such as RNNs, LSTMs, and sequence-to-sequence models. The second part reviews prior literature related to the use of machine learning architectures for turbo decoding, highlighting seminal works from the 1990s that pioneered these approaches.

Chapter 3 introduces the ***TurboAttention*** model, the first proposed approach in this study. This model applies a sequence-to-sequence attention mechanism originally developed for machine translation to the task of turbo decoding. Comparative evaluations against baseline models and iterative turbo decoders are conducted to assess their efficacy. In addition, hardware implementation of these models to test in inference and explore their functioning on real-time

Chapter 4 presents the ***TurboTransformer*** model, which adopts the Transformer architecture known for its advanced attention mechanisms. Initially designed for high-performance language translation tasks, this chapter explores its adaptation for turbo decoding applications, investigating its potential to enhance decoding accuracy and efficiency. Also, hardware inference tests have been done for ***TurboTransformer***.

The **Conclusion** summarizes the findings and contributions of the study, discussing implications for future research directions in the field of machine learning-assisted turbo decoding.

Chapter 1

Theoretical Foundations

Intoduction

In this chapter, our goal is to lay the groundwork for understanding the basics of channel coding to delving into advanced coding methods. We will explore the intricacies of using different decoding strategies to achieve reliable communication.

We begin by examining digital communication systems and understanding the pivotal role of channel coding in ensuring reliable data transmission. Next, we will provide an overview of two major types of error-correcting codes: block codes and convolutional codes. Building on this foundation, we will delve into convolutional turbo codes, offering a comprehensive understanding from both encoding and decoding perspectives. We will introduce a well-known iterative method for decoding turbo codes, analyze its performance, and finally explore the neural network architectures that can be employed to achieve similar decoding capabilities for turbo codes.

1.1 Overview of channel coding

Channel coding is a fundamental technique used in digital communication systems to ensure the integrity and reliability of data transmission. By introducing redundancy into the transmitted information, channel coding allows the detection and correction of errors that may occur due to noise or other impairments in the communication channel. Its critical role in digital communication systems and the major types of error-correcting codes are highlighted in the following subsections.

1.1.1 Digital communication systems

In order to understand the role of channel coding, we present the general structure of a digital communication system, as shown in Figure 1.1.

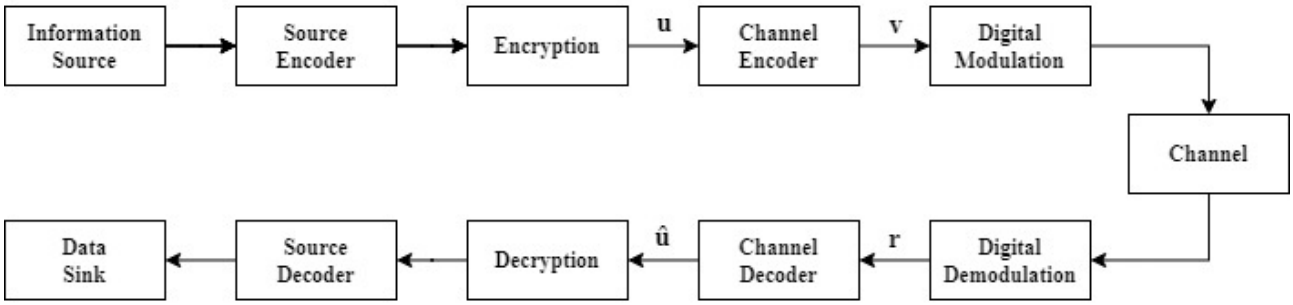


Figure 1.1: General model of a digital communication system.

The digital communication system model can be divided into three primary components: the transmitter, which handles the preparation of the information-carrying signal; the channel, which serves as the medium through which the signal is transmitted; and the receiver, which is responsible for reconstructing the original information.

1.1.1.1 Source encoding and encryption

The transmitter's role is to convert information from a source into a format that increases resilience to noise during transmission through the channel. The information source provides the messages to communicate, which are transformed into sequences of symbols, often binary. To enhance transmission efficiency, the source encoder compresses the input source sequence by minimizing redundancy. This enables the possibility of conveying larger amounts of information with the same data rate. We define the data rate or bit rate r_b, as the speed in bits-per-second at which the source encoder generates bits. To make source transmission secure, the encryption process converts source bits into a source stream that resembles meaningless random bits of data.

1.1.1.2 Channel encoding

Channel impairments introduce errors in the received signal, prompting the inclusion of a channel encoder to add redundancy and minimize errors. The process of introducing some redundant bits to a sequence of information bits in a controlled manner to detect and correct transmission errors is known as channel coding or error control coding. The channel encoder assigns codewords or code sequences to the information sequences, aiming to maximize their dissimilarity. The main idea is to encode the source data in such a way as to introduce dependency among the data, thus enabling the receiver to make a more accurate detection of the information to reconstruct.

The redundancy introduced by channel encoding usually follows a controlled format. As the channel encoder assigns to each information sequence of k bits a code sequence of n bits. This characteristic of the channel encoder is called the code rate, denoted R and expressed as:

$$R = \frac{k}{n} \tag{1.1}$$

When considering the rate r_b at which the data is fed to the channel encoder, the coded data rate at the output of the channel encoder, denoted r_c, is given by:

$$r_c = \frac{r_b}{R} \quad \text{Hz} \tag{1.2}$$

1.1.1.3 Modulation

The output of the channel encoder undergoes modulation before transmission over a channel. Modulation entails the mapping of encoded digital sequences into analog waveforms. During this process, carrier waves are combined with the encoded information, varying the amplitude, phase, or frequency of the waveforms to encode the transmitted information. This mapping can occur individually, where each bit is modulated separately, or through M-ary modulation, which involves modulating multiple bits at a time. This type of modulation introduces the notion of symbol rate, out of an M-ary modulator that maps a block of l bits coming from the channel encoder at a rate r_c into one of the $M = 2^l$ possible waveforms, is denoted r_s and defined as:

$$r_s = \frac{r_c}{l} = \frac{r_b}{Rl} \quad \text{Hz} \tag{1.3}$$

1.1.1.4 Channel

The channel constitutes a critical component in digital communication systems, serving as the medium through which signals are transmitted from the transmitter to the receiver. Channels introduce various impairments and characteristics that affect the fidelity and reliability of transmitted signals. Understanding these channel characteristics is essential for designing effective communication systems.

Channels can exhibit several limitations and properties, including:

- Thermal Noise: Random noise inherent in all electronic systems due to the thermal agitation of electrons.
- Finite Bandwidth: Limits the maximum rate at which signals can be transmitted through the channel.
- Multipath Propagation: Common in wireless communications, where signals reach the receiver via multiple paths, causing reflections and delays.
- Time Dispersion and Flat Fading: Causes frequency-selective attenuation and distortion of signals over time.

Regarding the noise, its impact cannot be totally removed since we do not have complete knowledge of the noise. However, in a simulation environment where the communication is studied and the noise is defined and modeled, we quantify the reliability of information transmission by the signal-to-noise ratio metric. The signal-to-noise ratio (SNR) is defined as the ratio of signal power to noise power, often expressed in decibels. Another important parameter is the bit error probability. Power efficiency is captured by the required bit energy to one-sided noise power

spectral density ratio, E_b/N_o, to achieve a specified bit error probability [8]. The signal-to-noise ratio (SNR), denoted by S/N, is related to E_b/N_o as:

$$\frac{S}{N} = lR\frac{E_b}{N_o} \tag{1.4}$$

Regarding the bandwidth, the channel bandwidth B limits the speed of signal variations. The signal bandwidth is a measure of its speed. The signals that change quickly in time have a large bandwidth. Bandwidth limitation of a system is quantified by the spectral efficiency [8], denoted by η, defined as:

$$\eta = \frac{r_b}{B} = \frac{r_s lR}{B}[\text{bits/sec/Hz}] \tag{1.5}$$

where r_s is the symbol rate. As the minimum required bandwidth for a modulated signal is r_s, the maximum spectral efficiency, denoted by η_{max}, is given by:

$$\eta_{max} = lR \tag{1.6}$$

For a given channel, there is an upper limit on the data rate related to the signal-to-noise ratio and the system bandwidth. Shannon introduced the concept of channel capacity, C, as the maximum rate at which information can be transmitted over a noisy channel.

Shannon's channel coding theorem guarantees the existence of codes that can achieve an arbitrarily small probability of error if the data transmission rate r_s is smaller than the channel capacity. Conversely, for a data rate $r_b > C$, it is not possible to design a code that can achieve an arbitrarily small error probability. This fundamental result shows that noise sets a limit on the transmission rate but not on the error probability as widely believed before. Though the theorem does not indicate how to design specific codes to achieve the maximum possible data rate at arbitrarily small error probabilities, it motivated the development of several error control techniques [8]. The capacity of a channel depends significantly on the type of channel and its specific properties. Binary symmetric channels and additive white gaussian noise channels are the two of the most used models.

1.1.1.4.1 Binary symmetric channel: (BSC) is a common communications channel model used in coding theory and information theory. In this model, a transmitter wishes to send a bit (a zero or a one), and the receiver will receive a bit. The bit will be flipped with a crossover probability of p, and otherwise is received correctly.

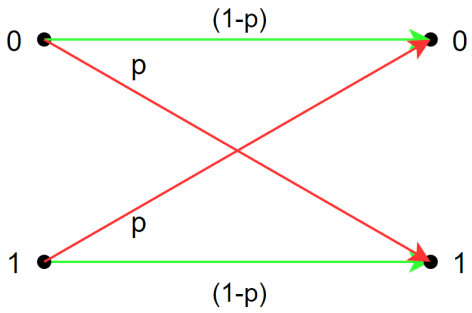


Figure 1.2: Binary symmetric channel model.

The noisy-channel coding theorem applies to BSC, saying that information can be transmitted with arbitrarily low error at any rate up to the channel capacity C, given by:

$$C_{BSC} = plog_2(p) + (1-p)log_2(1-p)[\text{bits/sec}] \tag{1.7}$$

1.1.1.4.2 Additive white gaussian noise (AWGN) is a noise model used in information theory to mimic the effect of random processes that occur in nature. It serves as a baseline model for evaluating system performance in various scenarios. While useful for theoretical analysis, it may not fully capture real-world channel characteristics. The model does not account for more complex channel impairments such as fading or multipath propagation. In the AWGN notation, the term additive implies that it is added to any noise that might be intrinsic to the information system. White refers to the uniformity of the power spectral density across the frequency band for the information system. Gaussian because it has a normal distribution in the time domain with an average time domain value of zero, $\mathcal{N}(0, \sigma^2)$.

The received signal $y(t)$ in an AWGN channel can be described as:

$$y(t) = x(t) + n(t) \tag{1.8}$$

where $x(t)$ is the transmitted signal and $n(t)$ is the additive white Gaussian noise.

The noisy-channel coding theorem applies to AWGN channels, saying that information can be transmitted with arbitrarily low error at any rate up to the channel capacity C, given by:

$$C_{AWGN} = Blog_2(1 + \frac{S}{N})[\text{bits/sec}] \tag{1.9}$$

1.1.1.5 Reception

In the receiver, the demodulator typically generates a binary or analog sequence at its output as the best estimates of the transmitted codeword or the modulated sequence, respectively [8].

Subsequently, the channel decoder utilizes the encoding scheme and channel characteristics to estimate the original message transmitted over the channel. The decoder's primary objective is to minimize the effects of channel-induced noise [8].

Following the rules of source encoding and encryption, the input sequence transforms to reconstruct an estimate of the source output sequence, which is then delivered to the end user [8].

1.1.2 Types of error correcting codes

The two most frequently used types of codes are block and convolutional codes. The main difference between the two of them is the memory of the encoder. In block codes, each encoding operation depends on the current input message and is independent of previous encodings. That is, the encoder has no memory of history of past encodings. In contrast, for a convolutional code, each encoder output sequence depends not only on the current input message but also on a number of past message blocks.

1.1.3 Linear block codes

In block coding, the information sequence is segmented into *message blocks* of fixed length; each message block, denoted by $\mathbf{u}$, consists of k information bits. There are a total of 2^k distinct messages. The block encoder, according to its structure, transforms each input message $\mathbf{u}$ into a binary $n-$tuple, denoted $\mathbf{v}$ with $n > k$. This binary $n-$tuple $\mathbf{v}$ is referred to as the *codeword* of the message $\mathbf{u}$.

The set of the 2^k codewords is called a *block code*. The codewords in this block code must be all distinct. Therefore, there must be a one-to-one correspondence between a message $\mathbf{u}$ and its codeword $\mathbf{v}$. Also a binary block code is linear if and only if the modulo-2 sum of two codewords is another a codeword. The resulting block code of length n and 2^k codewords is called a *linear (n, k)* code, and forms a $k-$dimensional subspace of the vector space of all the $n-$tuples.

1.1.4 Convolutional codes

Convolutional codes are widely used in applications like satellite communications, cellular mobile, and digital video broadcasting. Its widespread adoption is attributed to its simple structure and the existence of easily implementable maximum likelihood soft decision methods [8].

Elias [9] first introduced convolutional codes, while Fourney [10] worked on the foundation of

the algebraic theory of convolutional codes.

1.1.4.1 Structure of convolutional encoder

The convolutional encoder processes the sequence of information continuously. The output of the n-bit encoder at any given time depends on the k-bit information sequence and m previous input blocks, i.e., a convolutional encoder possesses a memory order of m [11][12]. The (n,k,m) convolutional code refers to the set of sequences generated by k-input, n-output encoder of memory order m. If ν_i is the length of the i^{th} shift register in a convolutional encoder with k *input sequences*, i = *1,2,...,k* then

$$m = \max_{1 \leq i \leq k} \nu_i \tag{1.10}$$

And the overall *constraint length* ν of the encoder is defined as

$$\nu = \sum_{1 \leq i \leq k} \nu_i \tag{1.11}$$

The code rate R for a convolutional code is defined as

$$R = \frac{k}{n} \tag{1.12}$$

The information sequence $\mathbf{u}$ is fed into the encoder one bit at a time.

$$\mathbf{u} = (u_0, u_1, \cdots, u_l, \cdots) \tag{1.13}$$

The encoder operates as a linear system, thus n *output sequences* of the encoder are generated, denoted as

$$\begin{aligned} \mathbf{v}^{(1)} &= \left(v_0^{(1)}, v_1^{(1)}, \cdots, v_l^{(1)}, \cdots\right) \\ \mathbf{v}^{(2)} &= \left(v_0^{(2)}, v_1^{(2)}, \cdots, v_l^{(2)}, \cdots\right) \\ &\vdots \\ \mathbf{v}^{(n)} &= \left(v_0^{(n)}, v_1^{(n)}, \cdots, v_l^{(n)}, \cdots\right) \end{aligned} \tag{1.14}$$

These output sequences are interleaved to represent a single code sequence $\mathbf{v} = (\mathbf{v_0}, \mathbf{v_1}, \cdots, \mathbf{v_l}, \cdots)$, where $\mathbf{v_l} = (v_l^1, v_l^2, \cdots, v_l^n)$

The code consists of n *generator sequences*, each of them with length *m+1*, they are generated by examining the resulting outputs after setting $\mathbf{u}$ = (1 0 0 ...), and they are denoted as

$$\begin{aligned}\mathbf{g}^{(1)} &= \left(g_0^{(1)}, g_1^{(1)}, \cdots, g_m^{(1)}\right)\\ \mathbf{g}^{(2)} &= \left(g_0^{(2)}, g_1^{(2)}, \cdots, g_m^{(2)}\right)\\ &\vdots\\ \mathbf{g}^{(n)} &= \left(g_0^{(n)}, g_1^{(n)}, \cdots, g_m^{(n)}\right)\end{aligned} \tag{1.15}$$

The *output sequences* **v** are the result of the convolution of the **input sequence u** and each of the*m+1 generator sequences* that specifies the code, i.e.

$$v^{(i)} = u * g^{(i)}, 1 \leq i \leq n \tag{1.16}$$

and

$$\begin{aligned}\mathbf{v}_l^{(i)} &= u_l g_0^{(i)} + u_{l-1} g_1^{(i)} + \cdots + u_{l-m} g_m^{(i)}\\ &= \sum_{i=0}^{m} u_{l-j} g_j^{(i)}\end{aligned} \tag{1.17}$$

Let's consider the block diagram shown in Fig 1.3 of a binary rate R=1/2 nonsystematic feedforward convolutional encoder, which consists k=1 shift register with m=2 delay elements (Flip Flop), and n=2 modulo-2 adders, which generates a (2,1,2) convolutional code. The following sequences specify this code [12]

$$\begin{aligned}\mathbf{g^{(1)}} &= (101),\\ \mathbf{g^{(2)}} &= (111)\end{aligned}$$

Thus, for input u_l, the outputs are given as

$$\begin{aligned}v_l^{(1)} &= u_l + u_{l-2}\\ v_l^{(2)} &= u_l + u_{l-1} + u_{l-2}\end{aligned}$$

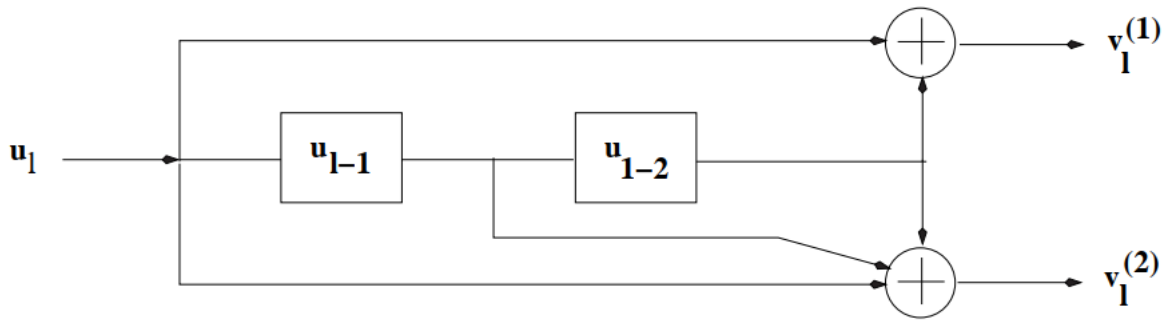


Figure 1.3: A rate R = 1/2 binary nonsystematic feedforward convolutional encoder with memory order m = 2 [12].

Let the information sequence $\mathbf{u} = (1011100)$. The output sequences are given by

$$\mathbf{v}^{(1)} = (1001011 \cdots)$$
$$\mathbf{v}^{(2)} = (1100101 \cdots)$$

The codeword can be written as

$$\mathbf{v} = (11, 01, 00, 10, 01, 10, 11, \cdots)$$

1.1.4.2 Convolutional encoder eepresentations

Convolutional encoders can be represented in various ways, and many properties can be derived from these representations. These different perspectives on convolutional encoders provide insights into their operation and efficiency.

1.1.4.2.1 Matrix representation

For any (n,k,m) convolutional encoder, the *generator matrix* is given by [12] [11]

$$\mathbf{G} = \begin{bmatrix} \mathbf{G}_0 & \mathbf{G}_1 & \mathbf{G}_2 & \cdots & \mathbf{G}_m & & \\ & \mathbf{G}_0 & \mathbf{G}_1 & \cdots & \mathbf{G}_{m-1} & \mathbf{G}_m & \\ & & \mathbf{G}_0 & \cdots & \mathbf{G}_{m-2} & \mathbf{G}_{m-1} & \mathbf{G}_m \\ & & \ddots & & & & \end{bmatrix} \tag{1.18}$$

G_l is a $k \times n$ sub-matrix

$$\mathbf{G}_l = \begin{bmatrix} g_{1,l}^{(1)} & g_{1,l}^{(2)} & \cdots & g_{1,l}^{(n)} \\ g_{2,l}^{(1)} & g_{2,l}^{(2)} & \cdots & g_{2,l}^{(n)} \\ \vdots & \vdots & & \vdots \\ g_{k,l}^{(1)} & g_{k,l}^{(2)} & \cdots & g_{k,l}^{(n)} \end{bmatrix} = \left[g_l^{(1)} g_l^{(2)} \cdots g_l^{(n)} \right], 0 \leq l \leq m \tag{1.19}$$

Each group of rows k of the matrix $\mathbf{G}$ is identical to the previous group of rows but shifted n places to the right. For an *information sequence* $\mathbf{u}$, the *codeword* $\mathbf{v}$ is given by [12] [11]

$$\mathbf{v} = \mathbf{u}\mathbf{G} \tag{1.20}$$

Take the example of (2,1,2) convolutional encoder in Fig1.3 with $\mathbf{g^{(1)}} = (101)$ and $\mathbf{g^{(2)}} = (111)$, the generator matrix is

$$\mathbf{G} = \begin{bmatrix} 11 & 01 & 11 & & & \\ & 11 & 01 & 11 & & \\ & & 11 & 01 & 11 & \\ & & \ddots & & & \ddots \end{bmatrix} \tag{1.21}$$

For $\mathbf{u} = (1\ 0\ 1\ 1\ 1\ 0\ 0\ \cdots)$, $\mathbf{v}$= $\mathbf{uG}$=(11,01,00,10,01,10,11,00,00, $\cdots$)

1.1.4.2.2 Polynomial representation

In linear systems, time-domain operations such as convolution can be substituted by more manageable transform-domain operations, specifically polynomial multiplication. Since a convolutional encoder is considered a linear system, each sequence in the encoding equations can be substituted with an equivalent polynomial multiplication [12] [11]. Thus, in the polynomial representation of the encoded binary sequence, the sequence is represented by the coefficients of the polynomial, and the encoding equations become

$$\mathbf{v}^{(i)}(D) = \mathbf{u}(D)\mathbf{g}^{(i)}(D), 1 \leq i \leq n \tag{1.22}$$

$$\mathbf{v}(D) = \mathbf{v}^{(1)}(D^n) + D\mathbf{v}^{(2)}(D^n) + \cdots + D^{n-1}\mathbf{v}^{(n)}(D^n) \tag{1.23}$$

The encoding equations can also be written as :

$$\mathbf{v}(D) = \mathbf{u}\left(D^n\right)\mathbf{g}(D) \tag{1.24}$$

where

$$\mathbf{g}(D) \triangleq \mathbf{g}^{(1)}\left(D^n\right) + D\mathbf{g}^{(2)}\left(D^n\right) + \cdots + D^{n-1}\mathbf{g}^{(n)}\left(D^n\right) \tag{1.25}$$

Consider the sample example of the convolutional code presented in 1.1.4.2.1 :
Time Domain

$$\mathbf{v} = \mathbf{uG} = (11, 01, 00, 10, 01, 10, 11, 00, 00, \cdots)$$

Transform Domain

$$\begin{aligned} \mathbf{v}(D) &= \mathbf{u}\left(D^2\right)\mathbf{g}(D) = \left(1 + D^4 + D^6 + D^8\right)\left(1 + D + D^3 + D^4 + D^5\right) \\ &= 1 + D + D^3 + D^6 + D^9 + D^{10} + D^{12} + D^{13} \end{aligned}$$

1.1.4.2.3 State diagram

As a sequential circuit, a convolutional encoder can be represented by a state diagram.

Where the encoder's state corresponds to the contents of its shift registers. Consider an (n,1,m) convolutional code at time instant l, then the state is defined by the m-tuple [12] [11]

$$S_l = (x_{l-1}, x_{l-2}, \ldots, x_{l-m})$$

A (n,1,m) convolutional code comprises 2^{m} possible states. At each time instant l, the output of the convolutional code depends on the input and the current state.

$$v_l = f(u_l, S_l)$$

The convolutional encoder experiences a state transition whenever a new information bit is fed into the encoder [12] [11].

Time unit	Message bit	State
l	u_l	$S_l = (x_{l-1}, x_{l-2}, \ldots, x_{l-m})$
$l+1$	u_{l+1}	$S_{l+1} = (x_l, x_{l-1}, \ldots, x_{l-m+1})$

A state transition is depicted by a directed edge linking two states, S_l and S_{l+1}. These transitions are annotated with the information and coded bits associated with that specific transition [12] [11].

Figure 1.4 below presents the state diagram of the convolutional encoder (2,1,2) shown in Figure 1.3.

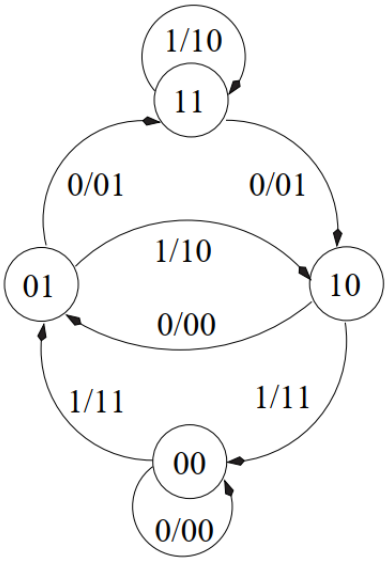


Figure 1.4: State diagram of a rate R = 1/2 binary non-systematic feedforward convolutional encoder with memory order m = 2 [12].

1.1.4.2.4 Trellis diagram

The *trellis diagram* is constructed using the state diagram by tracing all conceivable input/output sequences and the accompanying state transitions[8], thus it includes a time dimension.

Within each section of the trellis, states are depicted twice, once at time l and again at time $l+1$. A branch connects the state S_l to the state S_{l+1} if an input u_l at time l leads the encoder from state S_l to state S_{l+1}. By combining trellis sections across various time units, a trellis diagram for a convolutional code is constructed. Each code word consists of the labels on the trellis transitions, representing a particular path through the trellis.

Figure 1.5 below represents a trellis diagram of the convolutional encoder (2,1,2).

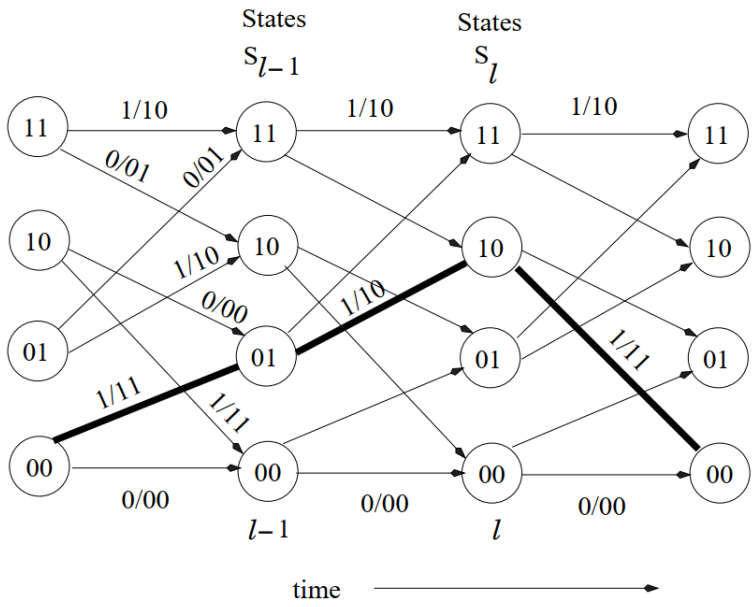


Figure 1.5: Trellis diagram of a rate R = 1/2 binary non-systematic feedforward convolutional encoder with memory order m = 2 [12].

1.1.4.3 Classification of convolutional codes

Convolutional encoders, distinguished by their structural and encoding properties, can be classified into several categories based on key differentiating aspects.

1.1.4.3.1 Systematic encoder

A convolutional code of rate $R = k/n$, in which the k information sequences appear explicitly unchanged within the n code sequences is referred to as a systematic encoder, and the corresponding generator matrix of this encoder is called a systematic generator matrix [12]. An example of a systematic convolutional encoder is shown in Figure 1.7

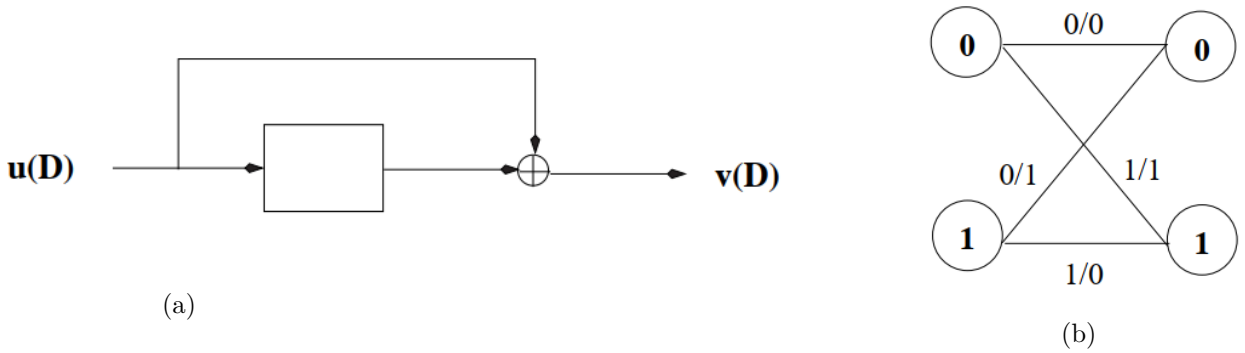
1.1.4.3.2 Nonsystematic encoder

In a nonsystematic convolutional encoder, the k information sequence does not appear unchanged within the n coded sequences [12]. An example of a nonsystematic convolutional encoder is shown in Figure 1.6

1.1.4.3.3 Feedforward encoder

A feedforward encoder corresponds to a polynomial generator matrix that lacks any feedback path. The output of such an encoder can be expressed as a linear combination of the current input and a finite number of past inputs. It can also be termed as *non-recursive encoder* [12].

Figure 1.6 shows the diagram of the encoder of a rate R = 1, 2-state feedforward encoder with generator matrix $\mathbf{G}(D) = [1 + D]$ is shown using a shift register implementation.



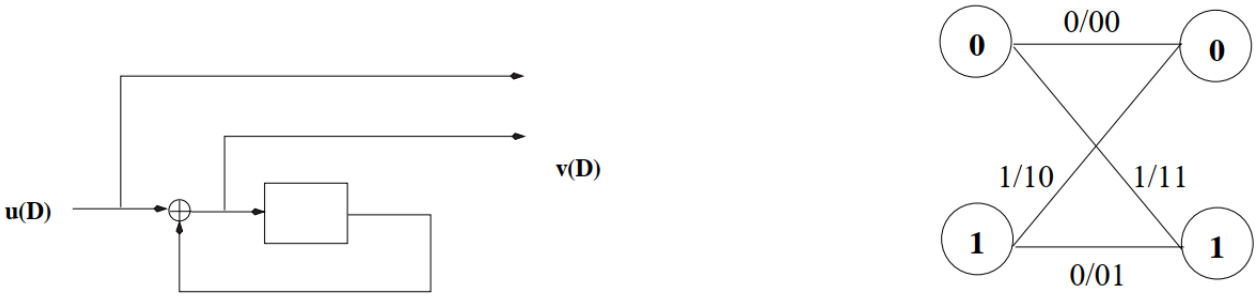
(a) (b)

Figure 1.6: Rate R = 1, 2-state feedforward encoder, $\mathbf{G}(D) = [1 + D]$ [12].

1.1.4.3.4 Feedback encoder

A feedback encoder corresponds to a rational generator matrix featuring at least one non-polynomial transfer function that includes a feedback path. The output of such an encoder can be expressed as a linear combination of past inputs and past outputs, implying its dependence on an infinite number of past inputs. This encoder can also be termed as *recursive encoder* [12].

In Figure 1.7 the diagram of the encoder of a rate R = 1/2, 2-state feedback encoder with generator matrix $\mathbf{G}(D) = [1 \ \frac{1}{1+D}]$ is shown.



(a) (b)

Figure 1.7: Rate R = 0.5, 2-state feedback encoder, $\mathbf{G}(D) = [1 \ \frac{1}{1+D}]$ [12].

1.1.4.4 Equivalent encoder

Two convolutional encoders are considered equivalent if their generator matrices $\mathbf{G}(D)$ and $\mathbf{G}'(D)$, are equivalent, meaning they encode the same code. If and only if there exists a rational invertible matrix $T(D)$ such that $\mathbf{G}'(D) = T(D)\mathbf{G}(D)$, thus $\mathbf{G}(D)$ and $\mathbf{G}'(D)$ are called equivalent [12].

Example: The generator matrices $\mathbf{G}(D) = [1 \ \frac{1}{1+D}]$ and $\mathbf{G}(D) = [1 + D \ 1]$ are equivalent, while $T(D) = [\frac{1}{1+D}]$

1.1.4.5 Catastrophic encoder

A convolutional encoder is considered catastrophic if it encodes an information sequence containing infinitely many non-zero symbols into a code sequence with only finitely many non-zero symbols. This implies that a finite number of errors in the channel can cause infinitely many errors in the received sequences[12].

Example:

Consider the convolutional encoder with the following generator matrix $\mathbf{G}(D) = [1+D \quad 1+D^2]$. For an input sequence, $\mathbf{u}(D) = [\frac{1}{1+D}] = 1 + D + D^2 + \cdots$, the output sequence is $[1 \quad 1 + D]$ has only weight 3, despite the input sequence having an infinite weight.

1.1.5 Distance properties of convolutional codes

The distance is a parameter that defines the performance of the convolutional code. The minimum free distance of a convolutional code is denoted as

$$d_{free} \triangleq \min_{\mathbf{u},\mathbf{u}'} d(\mathbf{v}, \mathbf{v}') : \mathbf{u} \neq \mathbf{u}' \tag{1.26}$$

Such that $\mathbf{v}$ and $\mathbf{v}'$ represent the code sequences corresponding to the information sequences $\mathbf{u}$ and $\mathbf{u}'$, respectively.

d_{free} is the minimum Hamming distance between any two distinct code sequences in the code. Additionally, d_{free} can be understood as the minimum weight of a non-zero sequence in the code.

$$d_{free} = min\{w(\mathbf{v} : \mathbf{u} \neq 0)\} \tag{1.27}$$

1.2 Convolutional turbo codes

1.2.1 Structural overview

A turbo coding system consists of three primary components: the constituent encoders, the interleaver, and the iterative decoder. While none of these components is novel on its own, the innovation introduced in [1] lies in combining them with a near-optimal decoding scheme, which is crucial to the success of turbo codes.

The constituent encoder is responsible for generating redundant data from the input information, enhancing the robustness of the transmission.

The interleaver, on the other hand, rearranges the order of the input bits to ensure that errors occurring in bursts during transmission are spread out when decoded.

The iterative decoder is the final component of the turbo coding system. It applies a process of repeated decoding and information exchange between multiple decoders to progressively improve the estimate of the transmitted message.

The synergy of these three components, when combined with an efficient decoding strategy as proposed in [1], leads to the remarkable performance of turbo codes, making them a powerful tool in modern communication systems. In this section 1.2, we will discuss the turbo encoder components in detail to provide a comprehensive understanding of their individual roles and contributions to the overall system, the turbo decoder components will be discussed in section 1.3.

1.2.2 Constituent encoders

The constituent encoders of turbo codes are recursive systematic convolutional (RSC) encoders, that are generally identical and come as two or more. The encoder structure is a parallel concatenation because each constituent encoder works on the same input sequence, rather than passing coded information from one encoder to another. The block diagram of a turbo encoder

with two constituent convolutional encoders is shown, in Figure 1.8.

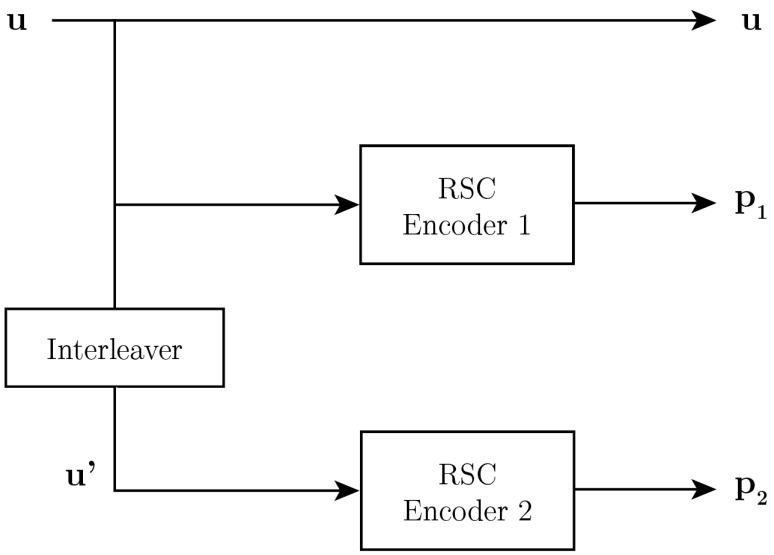


Figure 1.8: Block diagram of a turbo encoder with two constituent convolutional encoders.

The encoding process for two constituent encoders can be described as follows. The input information bit sequences **u** feed the first encoder and, after having permuted by the interleaver, the resulting bit sequences labeled **u'** enter the second encoder. A codeword of a turbo code is formed by the output bit sequences from the encoders, together with the information bit sequence. The role of the interleavers is one of the key factors determining the outstanding performance of turbo codes. Traditionally, interleavers are used to combat burst channel noise, which degrades the performance of the coding system. Here, interleavers are deployed but for a different reason, as will be further explained in the subsection 1.2.3. Next, we introduce the notions of trellis termination for convolutional encoders and punctured turbo codes, which are key properties that influence the performance and efficiency of the encoding scheme.

1.2.2.1 Trellis termination

Trellis termination is a technique used to drive the encoder to the all-zero state. This is required when a new block of N information bits is passed to be encoded, in order to have the initial state for the new block as the all-zero state.

Given that the constituent encoders are recursive, it is not possible to terminate a trellis by transmitting zero bits. We need to take into consideration the state of the constituent encoder after N information bits to correctly terminate it. A solution is proposed in Figure 1.9. A switch in each constituent encoder is in position "A" for the first N cycles, and then in position "B" for an amount ν of additional cycles corresponding to the memory order of the constituent encoder. This flush the register with zeros and drives the encoder to the all-zero state. This

technique creates ν additional bits that are called tail bits.

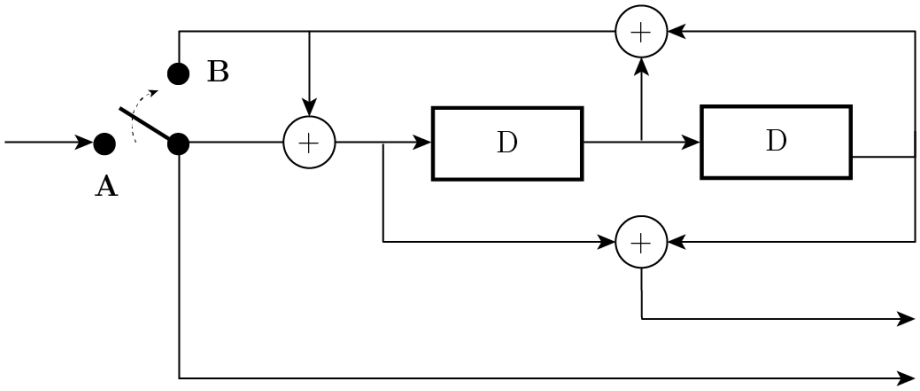


Figure 1.9: Trellis termination illustrated on a RSC constituent encoder.

1.2.2.2 Punctured turbo codes

The overall turbo code rate R does not necessarily inherit from the code rate of its constituent encoders. Various overall code rates can be obtained from the rate 1/3 turbo encoder shown in Figure 1.8 by puncturing the coded bits. When puncturing is applied, some output bits of the codeword are deleted according to a chosen pattern defined by a puncturing matrix P. For instance, a rate 1/2 turbo code can be obtained by puncturing a rate 1/3 turbo code. The commonly used puncturing matrix is given in Equation 1.28 and its results are shown in Figure 1.10. A value of 0 in a puncturing matrix implies that the corresponding bit is punctured, and a value of 1 that the bit is kept.

$$P = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{1.28}$$

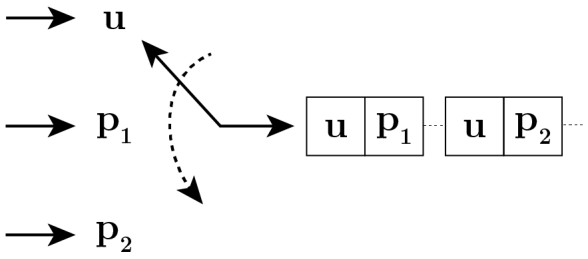


Figure 1.10: Puncturing of a rate 1/3 turbo code using P matrix in 1.28.

The selection of the puncturing matrix often follows established practices that have proven effective. We avoid puncturing systematic bits because they are more important than parity

bits. Additionally, we prevent the puncturing of tail bits, as non-termination of the turbo code can result in significant performance loss. And to maximize the balance in strength between the multiple constituent encoders, we alternate the puncturing of the parity bits.

1.2.3 Interleaving

The interleaver in turbo coding is a device that scrambles the input information based on a permutation of N elements with no repetitions. Interleaving involves dispersing the information over time, in a way that error events associated with a small distance in one code lead to an error event in the other code with a very large distance. Its use is profitable to reduce the effect of long attenuations in transmissions affected by fading and also where perturbation can alter consecutive symbols [13]. It has thus been pointed out that the design of the interleaver plays a central role for turbo code design and different approaches to the design of the interleaver have been proposed.

Regular permutation

The regular permutation or block permutation is the first approach developed to fulfill the interleaving task. The method states that N information bits can be organized in a table lookup of rows and columns. The process is done by writing the data in the memory table row by row, and reading it column by column. Figure 1.11 shows a block permutation interleaver.

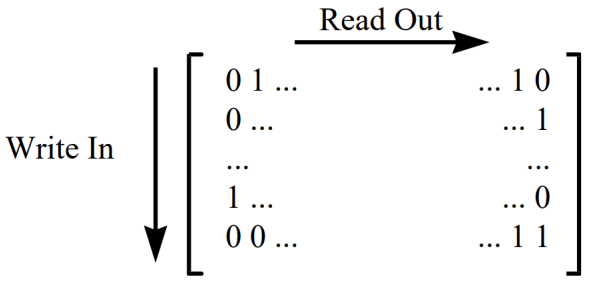


Figure 1.11: Block permutation interleaver [14].

Quadratic permutation

The use of quadratic permutation polynomials (QPP) as a class of deterministic interleavers, fulfills the task using a simple arithmetic computation instead of a table lookup. Given an information block of $N > 2$ elements, a polynomial

$$\pi(x) = (a_0 + a_1 x + a_2 x^2) \text{ modulo} N \tag{1.29}$$

where the coefficients a_0, a_1 and a_2 represent a shift of the permutation elements are non-negative integers, is said to be a quadratic permutation polynomial over $\mathbb{Z}_N$ when $\pi(x)$ permutes $\{0, 1, 2, \ldots, N-1\}$ [15].

As for the QPP interleavers in the Long-Term Evolution (LTE) standard, we only consider polynomials with free term $a_0 = 0$ [16]. In the search of best performing quadratic permutation polynomial for various block lengths, conditions have been established in [16] over the choice of quadratic coefficients. The proposed polynomials will be used further on in our interleaving process to guarantee the maximum efficiency of turbo codes.

1.3 Iterative turbo decoding

The decoding of turbo codes takes its roots from the original Viterbi decoding algorithm, proposed in 1967 [17]. The algorithm aims to decode convolutional codes based on the knowledge of the structure of convolutional encoders and the synergy between the state evolution and the produced parity bits. Many modified versions of the Viterbi algorithm have been introduced to overcome its limitations in terms of performance. Turbo codes benefit from these algorithms to decode the output of each of its constituent encoders in an iterative manner. We will first introduce the Viterbi decoding algorithm for convolutional codes. Then, its integration in the iterative decoding of turbo codes. Finally, the modified decoding algorithm, Soft-Output Viterbi Algorithm (SOVA) and its implementation for performance analysis of turbo codes.

1.3.1 Viterbi decoding algorithm for convolutional code

A $(n, 1, m)$ convolutional code can be represented using its trellis diagram. Typically, the initial state of a convolutional code is the all-zero state. For time $l \leq m$, as information bits are shifted into the encoder, the number of states doubles with each shift. By time $l = m$, the number of states reaches 2^m. Since one information bit enters the encoder at each time step, two branches leave each state in the trellis diagram. For $l > m$, there are also two branches merging into each state. Encoding an information sequence is equivalent to tracing a path through the trellis [12]. An illustration of the path tracing in a trellis is shown in Figure 1.12.

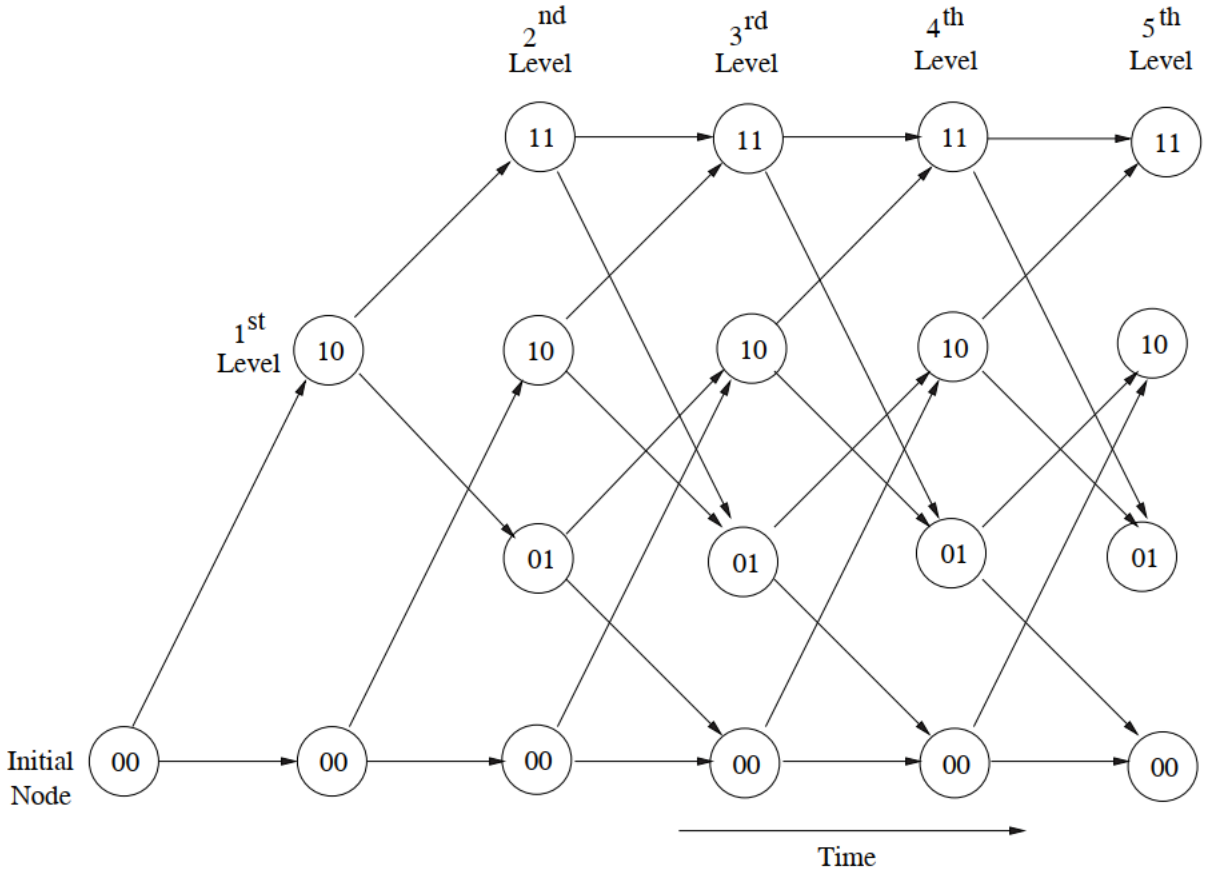


Figure 1.12: Trellis diagram of $(2, 1, 2)$ convolutional encoder with L=5 [12].

The encoder resets to an all-zero state after processing an L-bit information sequence, $\mathbf{u} = (\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_{L-1})$. During the termination process, the number of states is reduced to half repeatedly until all paths in the trellis diagram converge back to the all-zero state. For $l \leq m$, there is exactly one path of length l entering each node at time l. For $l > m$, there are exactly 2^{l-m} paths of length l entering each node at time l. In total, there are 2^l paths of length l.

On a Binary Symmetric Channel (BSC):

Let the information sequence of length L be

$$\mathbf{u} = (u_0, u_1, \ldots, u_l, \ldots, u_{L-1})$$

This sequence is encoded into a code sequence of length $N \triangleq (L + m)n$

$$\mathbf{v} = (v_0, v_1, \ldots, v_l, \ldots, v_{L+m-1})$$

If the code sequence $\mathbf{v}$ is transmitted over the channel, the received sequence is

$$\mathbf{r} = (r_0, r_1, \ldots, r_l, \ldots, r_{L+m-1})$$

where the l-th received block is

$$\mathbf{r}_l = (r_l^{(1)}, r_l^{(2)}, \ldots, r_l^{(n)})$$

A maximum likelihood decoder identifies the path through the trellis that maximizes the conditional probability of the received sequence given the code sequence:

$$P(\mathbf{r}|\mathbf{v}) = \prod_{l=0}^{L+m-1} P(\mathbf{r}_l|\mathbf{v}_l) \tag{1.30}$$

The branch conditional probability is given by:

$$P(\mathbf{r}|\mathbf{v}) = \prod_{i=1}^{n} P(\mathbf{r^{(i)}}_l|\mathbf{v^{(i)}}_l) \tag{1.31}$$

where the bit conditional probabilities $P(\mathbf{r}_l^{(i)}|\mathbf{v}_l^{(i)})$ are the channel transition probabilities.

Maximizing $P(\mathbf{r}|\mathbf{v})$ is equivalent to maximizing:

$$M(\mathbf{r}|\mathbf{v}) \triangleq \log P(\mathbf{r}|\mathbf{v}) \tag{1.32}$$

This $M(\mathbf{r}|\mathbf{v})$ is called the path metric:

$$M(\mathbf{r}|\mathbf{v}) = \sum_{l=0}^{L+m-1} \log P(\mathbf{r}_l|\mathbf{v}_l) = \sum_{l=0}^{L+m-1} M(\mathbf{r}_l|\mathbf{v}_l) \tag{1.33}$$

where the branch metrics are:

$$M(\mathbf{r}_l|\mathbf{v}_l) = \sum_{i=1}^{n} \log P(r_l^{(i)}|v_l^{(i)}) = \sum_{i=1}^{n} M(r_l^{(i)}|v_l^{(i)}) \tag{1.34}$$

The partial path metric for the first j branches of a path $\mathbf{v}$ is given by:

$$M([\mathbf{r}|\mathbf{v}]_j) = \sum_{l=0}^{j-1} M(\mathbf{r}_l|\mathbf{v}_l) \tag{1.35}$$

For a BSC, the maximum likelihood decoder decodes the received sequence $\mathbf{r}$ into the code sequence $\mathbf{v}$ that minimizes the Hamming distance $\mathbf{d}(\mathbf{r}, \mathbf{v})$.
The Viterbi algorithm is a computationally efficient method for finding the path through the trellis with the best metric.
The Viterbi decoder proceeds through the trellis level by level, searching for the path with the optimal metric.
At each level, the decoder compares the metrics of all partial paths entering each state. It

stores the partial path with the best metric (the survivor path) and eliminates all other partial paths.
For $m \leq l \leq L$, there are a total of 2^m survivor paths. The number of survivors decreases during the termination process until, at time $l = L + m$, there is only one survivor path left. This surviving path is the maximum likelihood path.

The Viterbi algorithm can be implemented as follows[14]:

$S_{k,t}$ is the state in the trellis diagram that corresponds to state $\mathbf{S_k}$ at time t. Every state in the trellis is assigned a value denoted $V(S_{k,t})$.

1. a. Initialize time $t = 0$.

   b. Initialize $V(S_{0,0}) = 0$ and all other $V(S_{k,t}) = +\infty$.

2. a. Set time $t = t + 1$.

   b. Compute the partial path metrics for all paths going to state S_k at time t. First, find the t-th branch metric

   $$M(\mathbf{r}_t|\mathbf{v}_t) = \sum_{j=1}^{n} M(r_t^{(j)}|v_t^{(j)}).$$

   This is calculated from the Hamming distance

   $$\sum_{j=1}^{n} |r_t^{(j)} - v_t^{(j)}|.$$

   Second, compute the t-th partial path metric

   $$M^t(\mathbf{r}|\mathbf{v}) = \sum_{i=0}^{t} M(\mathbf{r}_i|\mathbf{v}_i).$$

   This is calculated from

   $$V(S_{k,t-1}) + M(\mathbf{r}_t|\mathbf{v}_t).$$

3. a. Set $V(S_{k,t})$ to the "best" partial path metric going to state S_k at time t. Conventionally, the "best" partial path metric is the partial path metric with the smallest value.

   b. If there is a tie for the "best" partial path metric, then any one of the tied partial path metrics may be chosen.

4. Store the "best" partial path metric and its associated survivor bit and state paths.

5. If $t < L + m - 1$, return to Step 2.

1.3.2 Soft-Ouput Viterbi Algorithm (SOVA) for turbo codes

After introducing the Viterbi algorithm, we will delve into its modified version, the Soft-Output Viterbi Algorithm (SOVA). We will also present the general scheme for a turbo decoder, emphasizing the benefits of using SOVA and its soft output values in the decoding process of turbo codes. Finally, we will present a simulation environment for a communication system based on turbo codes and turbo decoding using SOVA, analyze its performance, and discuss the key considerations when configuring SOVA algorithm.

1.3.2.1 Principal of the decoder

The Viterbi algorithm is known for generating the maximum likelihood (ML) output sequence for convolutional codes, providing optimal sequence estimation for single-stage convolutional codes [14]. However, when applied to turbo codes, where a concatenated decoding approach is needed, the algorithm fails due to multiple reasons. The first Viterbi decoder suffers from a channel burst phenomenon and tends to commit a burst of errors. This output, when fed to the second decoder, harms the efficiency of the concatenated decoding. Also, the hard decision on the output data neglects the advantage of having soft decision data.

These issues can be mitigated, and the overall performance of the concatenated decoder significantly enhanced, by enabling the Viterbi decoders to generate reliability (soft-output) values[1]. These reliability values serve as a-priori information for subsequent Viterbi decoders, improving decoding accuracy. This enhanced version of the Viterbi decoder is known as the soft-output Viterbi algorithm (SOVA) decoder. We note that SOVA is a simplified version of the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm. While both algorithms are used for decoding convolutional codes, SOVA simplifies the process by generating soft outputs from the Viterbi Algorithm. In contrast, the BCJR algorithm provides exact a posteriori probabilities for each bit, which requires a more complex forward-backward procedure. SOVA, therefore, offers a balance between performance and computational complexity, making it a practical choice for many applications. Figure 1.13 illustrates a concatenated SOVA decoder, where y represents the received channel values, u represents the hard decision output values, and L represents the associated reliability values.

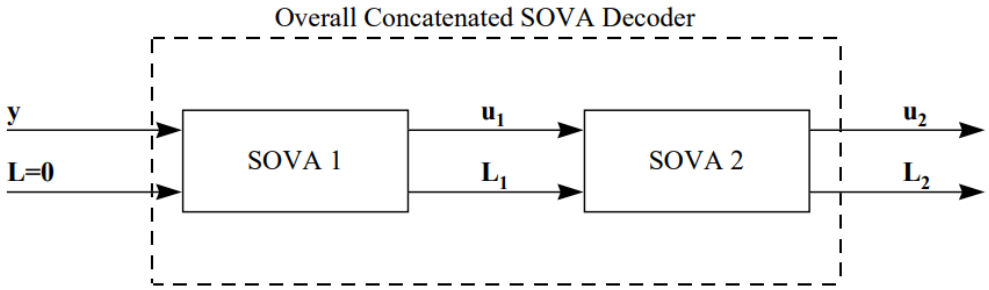


Figure 1.13: A concatenated SOVA decoder [14].

The soft value or reliability produced by a SOVA component decoder is calculated from the trellis diagram, as show in the example figure 1.14. The solid line indicates the survivor path and the dashed line indicates the competing (concurrent) path at time t for state 1. The paths shown in the figure are relative to the time t and the same logic is applied to other nodes for other time steps. The label $S_{1,t}$ represents state 1 and time t. The labels 0,1 shown on each path indicate the estimated binary decision for the paths. The survivor path for this node is assigned an accumulated metric $V_s(S_{1,t})$ and the competing path for this node is assigned an accumulated metric $V_c(S_{1,t})$. The fundamental information for assigning a reliability value L(t) to node $S_{1,t}$'s survivor path is the absolute difference between the two accumulated metrics, L(t)= | $V_s(S_{1,t})$ - $V_c(S_{1,t})$| [1]. The greater this difference, the more reliable is the survivor path. For this reliability calculation, it is assumed that the survivor accumulated metric is always "better" than the competing accumulated metric. Furthermore, the reliability values only need to be calculated for the ML survivor path, which corresponds to the survivor path at the last time step at the node of state 0 of a terminated trellis. [14]

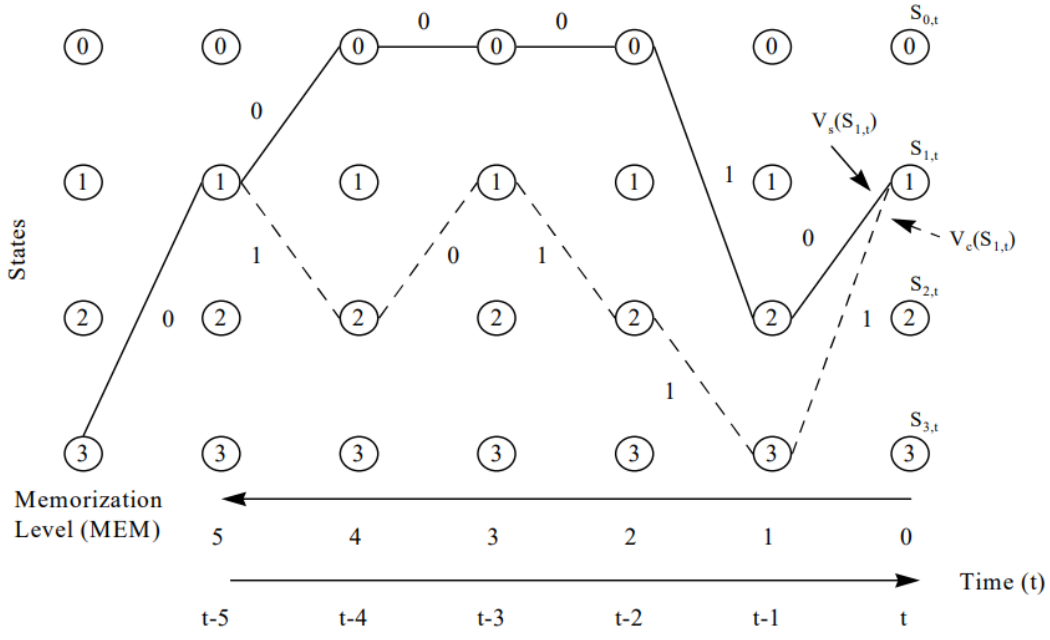


Figure 1.14: Example of survivor and competing paths for reliability estimation at time t [14].

To improve the reliability values of the survivor path, a traceback operation to update the reliability values has been suggested [1]. This updating procedure is integrated into the Viterbi algorithm as follows :

For node $S_{k,t}$ in the trellis diagram (corresponding to state k at time t),

1. Store L(t) = | $V_s(S_{k,t})$ - $V_c(S_{k,t})$|. This is also denoted as Δ in other papers. If there is more than one competing path, then multiple reliability values must be calculated and the smallest reliability value is then set to L(t).
2. Initialize the reliability value of $S_{k,t}$ to $+\infty$ (most reliable).
3. Compare the survivor and competing paths at $S_{k,t}$ and store the memorization levels (MEMs) where the estimated binary decisions of the two paths differ.
4. Update the reliability values at these MEMs with the following procedure:
    a. Find the lowest MEM>0, denoted as MEM_{low}, whose reliability value has not been updated.
    b. Update MEM_{low}'s reliability value L(t-MEM_{low}) by assigning the lowest reliability value between MEM = 0 and MEM = MEM_{low}.

1.3.2.2 Overview of the component decoder

The SOVA component decoder estimates the information sequence using one of the two encoded streams produced by the turbo code encoder, information and parity bits. Figure 1.15 shows the inputs and outputs of the SOVA component decoder [14].

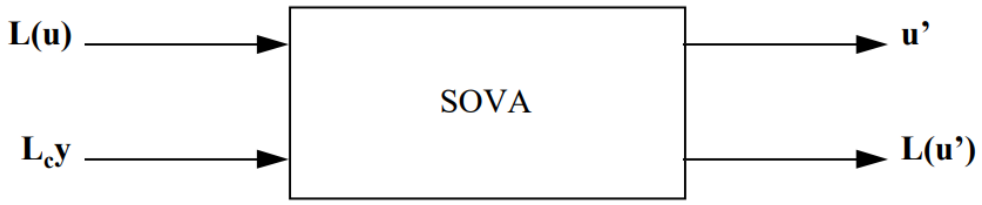


Figure 1.15: SOVA component decoder [14].

The SOVA component decoder operates similarly to the Viterbi decoder, except the ML sequence is found by using a modified metric. This modified metric, which incorporates the a-priori value, is given in Equation 1.36.

$$M_t^{(m)} = M_{t-1}^{(m)} + \sum_{j=1}^{N} x_{t,j}^{(m)} L_c y_{t,j} + u_t^{(m)} L(u_t) \tag{1.36}$$

Where m represents the index of the node state, t the time step, $x_{t,j}^{(m)}$ and $u_t^{(m)}$ the trellis transition bits and systematic bit, $y_{t,j}$ received bits at time t, L_c the channel reliability, and $L(u_t)$ the soft information value at time t from the previous component decoder. [14]

For systematic codes, this metric can be modified as shown in Equation

$$M_t^{(m)} = M_{t-1}^{(m)} + u_t^{(m)} L_c y_{t,1} + \sum_{j=2}^{N} x_{t,j}^{(m)} L_c y_{t,j} + u_t^{(m)} L(u_t) \tag{1.37}$$

As seen from 1.36 and 1.37, the SOVA metric incorporates values from the past metric, the channel reliability, and the source reliability (a-priori value). [14]

The soft output Viterbi algorithm (along with its reliability updating procedure) can be implemented as follows:

1. a. Initialize time $t = 0$.

   b. Initialize M_0^m only for the zero state in the trellis diagram and all other states to $-\infty$.

2. a. Set time $t = t + 1$.

   b. Compute the metric $M_t^{(m)} = M_{t-1}^{(m)} + u_t^{(m)} L_c y_{t,1} + \sum_{j=2}^{N} x_{t,j}^{(m)} L_c y_{t,j} + u_t^{(m)} L(u_t)$ for each state in the trellis diagram, where m denotes the allowable binary trellis branch/transition to a state (m= 1, 2).

   M_t^m is the accumulated metric for time t on branch m.

   u_t^m is the systematic bit for time t on branch m.

   $x_{t,j}^m$ is the j-th bit of N bits for time t on branch m ($2 \leq j \leq N$).

   $y_{t,j}^m$ is the received value from the channel corresponding to $x_{t,j}^m$.

   $L_c = 4\frac{E_b}{N_0}$ is the channel reliability value.

   $L(u_t)$ is the a-priori reliability value for time t. This value is from the preceding decoder. If there is no preceding decoder, then this value is set to zero.

3. Find $\max_m M_t^{(m)}$ for each state. For simplicity, let $M_t^{(1)}$ denote the survivor path metric and $M_t^{(2)}$ denote the competing path metric.

4. Store $M_t^{(1)}$ and its associated survivor bit and state paths.

5. Compute $\Delta_t^0 = \frac{1}{2}|M_t^{(1)} - M_t^{(2)}|$.

6. Compare the survivor and competing paths at each state for time t and store the MEMs where the estimated binary decisions of the two paths differ.

7. Update $\Delta_t^{MEM} \approx \min_{k=0,\ldots,MEM} \left\{\Delta_t^k\right\}$ for all MEMs from smallest to largest MEM.

8. Go back to Step (2) until the end of the received sequence.

9. Output the estimated bit sequence u' and its associated "soft" or L-value sequence $L(u') = u' \bullet \Delta$, where $\bullet$ operator defines element by element multiplication operation and Δ is the final updated reliability sequence. $L(u')$ is then processed (to be discussed later) and passed on as the a-priori sequence $L(u)$ for the succeeding decoder.

1.3.2.3 SOVA iterative turbo decoder

The iterative turbo code decoder is composed of two concatenated SOVA component decoders. Figure 1.16 shows the turbo code decoder structure.

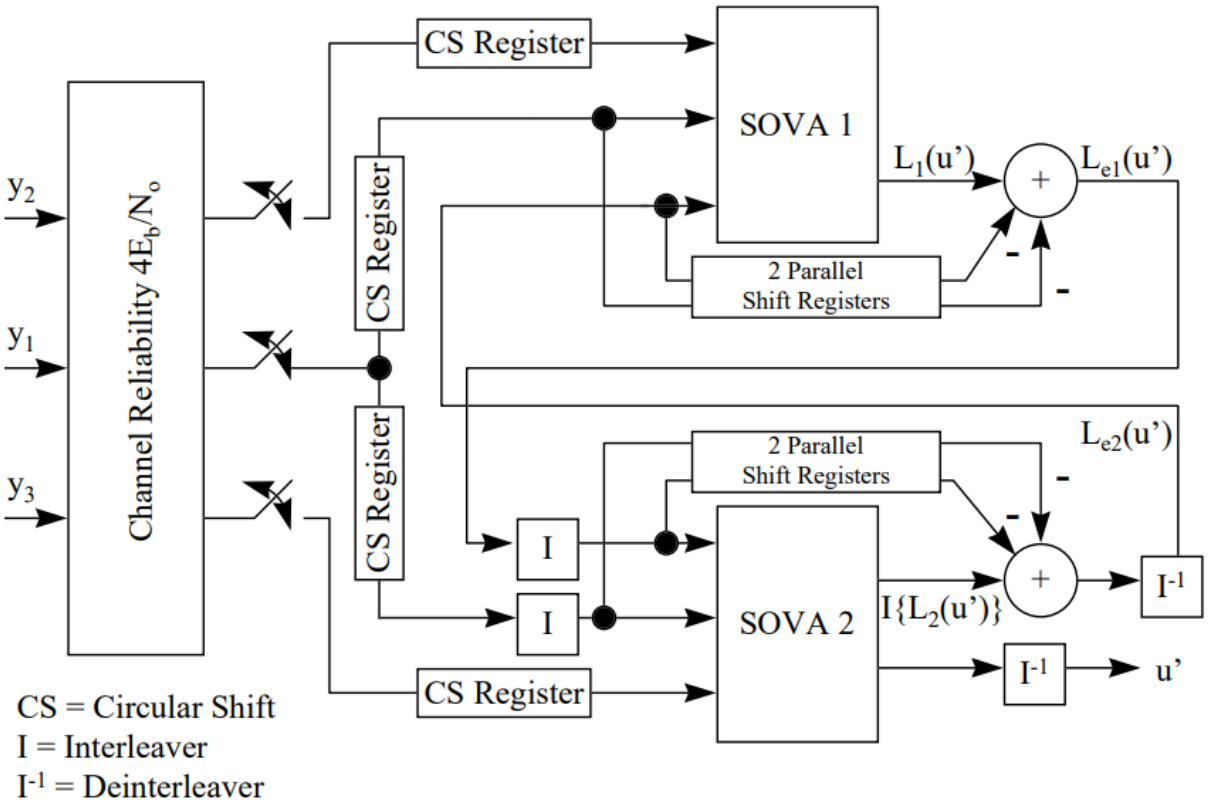


Figure 1.16: SOVA iterative turbo code decoder [14].

The turbo code decoder processes the received channel bits on a frame basis. As shown in Figure 1.16, the received channel bits are demultiplexed into the systematic stream y_1 and two parity check streams y_2 and y_3 from component encoders 1 and 2 respectively. These bits are weighted by the channel reliability value and loaded onto the circular shift registers. [14] The registers shown in the figure are used as buffers to store sequences until they are needed. The switches are placed in the open position to prevent the bits from the next frame from being processed until the present frame has been processed. The SOVA component decoder produces the "soft" or L-value $L(u-t')$ for the estimated bit u'_t (for time t). The "soft" or L-value$L(u-t')$

can be decomposed into three distinct terms,

$$L(u'_t) = L(u_t) + L_c y_{t,1} + L_e(u'_t) \tag{1.38}$$

$L(u_t)$ is the a-priori value and is produced by the preceding SOVA component decoder. $L_c y_{t,1}$ is the weighted received systematic channel value. $L_e(u'_t)$ is the extrinsic value produced by the present SOVA component decoder. The information that is passed between SOVA component decoders is the extrinsic value

$$L_e(u'_t) = L(u'_t) - L(u_t) - L_c y_{t,1} \tag{1.39}$$

The a-priori value $L(u_t)$ is subtracted out from the "soft" or L-value $L(u'_t)$ to prevent passing information back to the decoder from which it was produced. Also, the weighted received systematic channel value $L_c y_{t,1}$ is subtracted out to remove "common" information in the SOVA component decoders. [14]

Figure 1.16 shows that the turbo code decoder is a closed-loop serial concatenation of SOVA component decoders. In this closed-loop decoding scheme, each of the SOVA component decoders estimates the information sequence using a different weighted parity check stream. The turbo code decoder further implements iterative decoding to provide more dependable reliability/a-priori estimations from the two different weighted parity check streams, hoping to achieve better decoding performance. [14]

The iterative turbo code decoding algorithm for the n-th iteration is as follows:

1. The SOVA1 decoder inputs sequences $4\frac{E_b}{N_0}y_1$ (systematic), $4\frac{E_b}{N_0}y_2$ (parity check), and $L_{e2}(u')$ and outputs sequence $L_1(u')$. For the first iteration, sequence $L_{e2}(u') = 0$ because there is no initial a-priori value (no extrinsic values from SOVA2).

2. The extrinsic information from SOVA1 is obtained by $L_{e1}(u') = L_1(u') - L_{e2}(u') - L_c y_1$ where $L_c = 4\frac{E_b}{N_0}$.

3. The sequences $4\frac{E_b}{N_0}y_1$ and $L_{e1}(u')$ are interleaved and denoted as $I\{4\frac{E_b}{N_0}y_1\}$ and $I\{L_{e1}(u')\}$.

4. The SOVA2 decoder inputs sequences $I\{4\frac{E_b}{N_0}y_1\}$ (systematic), $I\{4\frac{E_b}{N_0}y_3\}$ (parity check that was already interleaved by the turbo code encoder), and $I\{L_{e1}(u')\}$ (a-priori information) and outputs sequences $I\{L_2(u')\}$ and $I\{u'\}$.

5. The extrinsic information from SOVA2 is obtained by $I\{L_{e2}(u')\} = I\{L_2(u')\} - I\{L_{e1}(u')\} - I\{L_c y_1\}$.

6. The sequences $I\{L_{e2}(u')\}$ and $I\{u'\}$ are deinterleaved and denoted as $L_{e2}(u')$ and u'. $L_{e2}(u')$ is fed back to SOVA1 as a-priori information for the next iteration, and u' is the estimated bits output for the n-th iteration.

1.3.2.4 Algorithm implementation

In the scope of analyzing the performance of the decoding using the Soft-Output Viterbi algorithm, we established a MatLab environment to simulate a turbo code based communication system.

1.3.2.4.1 Configuration of the turbo code

The turbo code selected for this study follows a commonly used configuration, called the 757 turbo code. The "757" notation indicates the configuration of the constituent recursive systematic encoders of memory order $\nu = 2$. It comes from the octal representation $(5)_8 = (101)_2$ of the feedforward polynomial $[1 + D^2]$, and the octal representation $(7)_8 = (111)_2$ of the feedback polynomial $[1 + D^1 + D^2]$. The 757 turbo code uses two of these constituent encoders, and each encoder is terminated using the technique described previously in 1.2.2.1. The overall code rate is 1/3, as no puncturing of the output was applied. The interleaver uses quadratic permutation polynomials, of which the coefficients were extracted from the research done in [15]. The Figure 1.17 shows a detailed diagram of the 757 turbo encoder used.

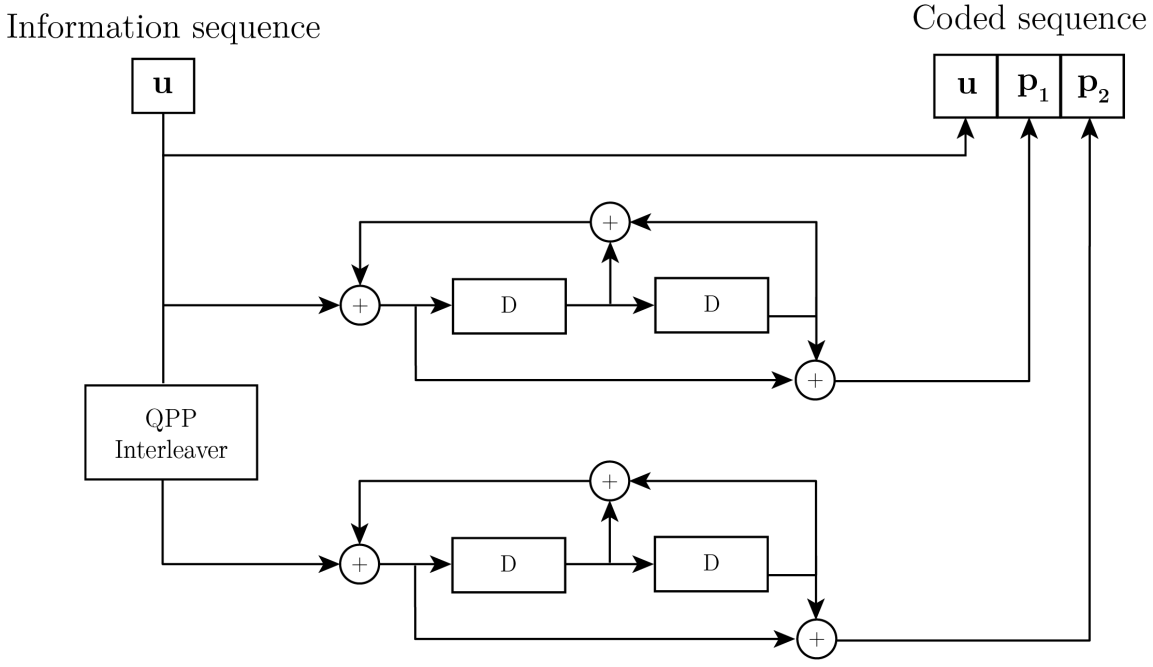


Figure 1.17: Diagram of the 757 turbo encoder used in simulation.

1.3.2.4.2 Configuration of the simulation environment

The flow of the communication system simulation is as follows. Random information bits are generated. The sequence of generated bits is divided and encoded using the previous encoder into blocks of various lengths.

Once the information is encoded into the coded sequence v, we modulate the data using a modulation of order $M = 2$. The modulation technique used is the Binary Phase Shift Keying (BPSK). We directly applied the constellation transformation, mapping bits 0 to -1 and bits 1 to 1. This approach was chosen instead of incorporating carrier waves and other details, as they are unnecessary for analyzing error performance.

The transmission of the modulated data is simulated using the integrated MatLab *awgn* function to mimic an Additive White Gaussian Noise (AWGN) channel. This type of channel is not best for modeling terrestrial path transmission, since it lacks notions of multipath, terrain blocking and interference but its complexity is sufficient enough for our analysis of decoding performance over noisy channels.

The received sequence, or the noisy sequence, is demodulated using a hard BPSK demodulation and passed to the turbo decoder. The architecture of the turbo decoder was previously described in 1.3.2.3.

After demultiplexing of the received sequences, the iterative decoding between SOVA blocks begins. The code implementation of the Soft-Output Viterbi Algorithm is provided as the MATLAB function given below.

The MATLAB code of SOVA is provided in the appendix.

1.3.2.5 Simulation results

The goal of the simulation is to determine the performance of the soft-output Viterbi algorithm under different configurations and channel conditions. The studied configurations cover the parameters of block length value K, number of decoding iterations N, and normalization of the extrinsic information with scaling factor k and threshold th. To thoroughly assess the impact of different SOVA configurations on decoding performance, we performed simulations with all possible combinations over an extended range of values. This exhaustive approach allowed us to systematically analyze and understand the significance and impact of each parameter on the overall decoding performance. To quantify the performance, we rely on two metrics that are essential in channel coding. Thus, the Bit Error Rate (BER) and Block Error Rate (BLER) metrics are introduced.

1.3.2.5.1 Bit Error Rate (BER) is a measure of the number of bit errors that occur in a transmission system over a given period of time. It is defined as the ratio of the number of erroneous bits received to the total number of bits sent. BER is an important parameter in digital communications, as it indicates the quality of the transmission and the performance of the error correcting code. Mathematically, it is expressed as:

$$\text{BER} = \frac{\text{Number of bit errors}}{\text{Total number of transmitted bits}} \tag{1.40}$$

1.3.2.5.2 Block Error Rate (BLER) is a measure of the number of erroneous blocks in a transmission system over a given period of time. A block is considered erroneous if any bit within the block is incorrect. BLER is defined as the ratio of the number of erroneous blocks received to the total number of blocks sent. This metric is particularly useful for understanding the performance of error correction codes that operate on blocks of data. Mathematically, it is expressed as:

$$\text{BLER} = \frac{\text{Number of erroneous blocks}}{\text{Total number of transmitted blocks}} \tag{1.41}$$

1.3.2.5.3 Effect of the block length

The block length parameter, though not directly related to SOVA, is a fundamental variable to consider. In a turbo code, the block length determines the size of the interleaver. It is well established in the literature that larger interleavers generally provide a better performance, a point further confirmed by our study. We utilized permutation of type QPP with coefficients from [15] wherever possible.

For the simulation, a bit stream of 2^{16} bits is generated and divided into blocks of different lengths. It is crucial to perform this comparison by evaluating the same total number of generated bits, rather than an equal number of blocks.

The BER performance comparison based on the information block length, with a number of decoding iterations fixed at $N = 6$ and the best performing normalization scheme, is presented in Figure 1.18.

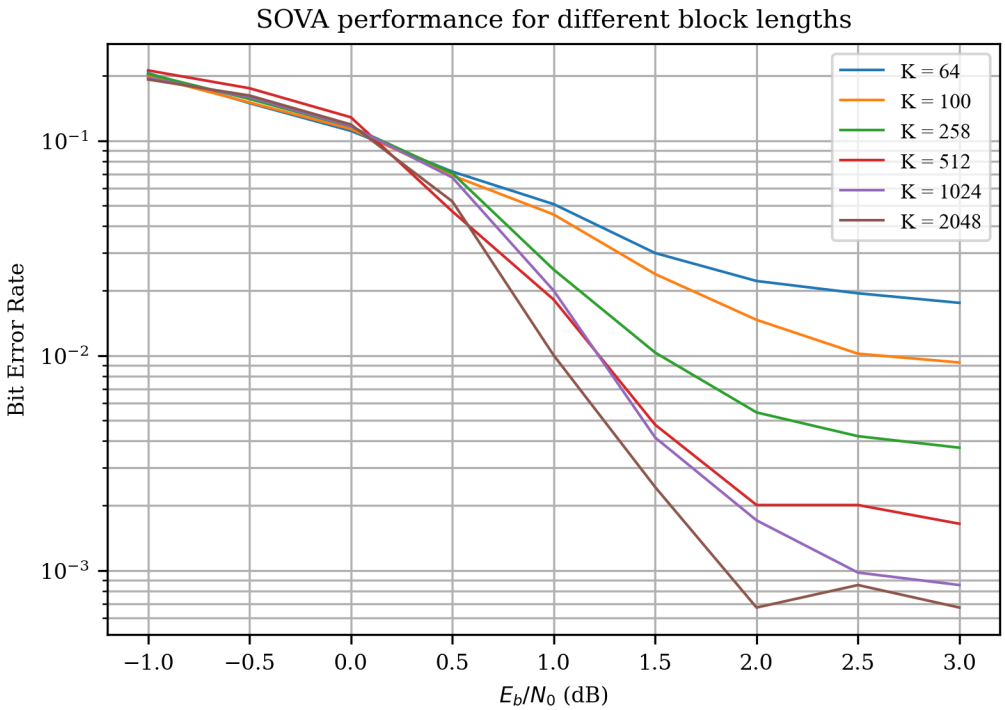


Figure 1.18: Comparison of SOVA BER performance for different block lengths.

As shown in 1.18, larger block lengths (K = 512, K = 1024, K = 2048) offer significantly better performance (lower BER) at higher E_b/N_0 values compared to smaller block lengths (K = 64, K = 100, K = 258). This suggests that increasing the block length improves the performance of SOVA, especially in higher SNR. This finding reinforces the earlier point regarding the positive relationship between interleaver size and turbo code performance.

1.3.2.5.4 Effect of the number of decoding iterations

The number of decoding iterations is a crucial parameter in determining SOVA's complexity and execution time. The BER performance comparison based on the number of decoding iterations, with block length fixed at $K = 100$ and the best performing normalization scheme is presented in Figure 1.19.

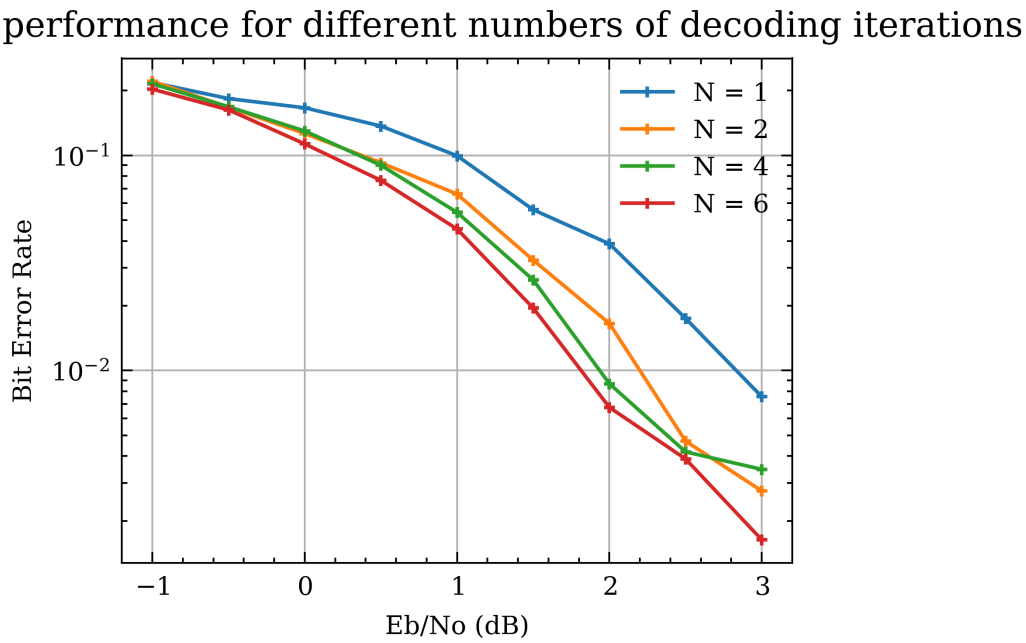


Figure 1.19: Comparison of SOVA BER performance for different numbers of decoding iterations.

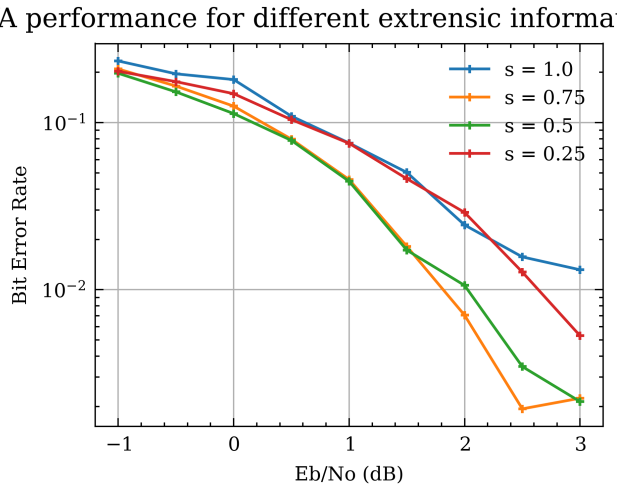
As shown in Figure 1.19, increasing the number of iterations (N = 6, N = 4) leads to a noticeable improvement in performance (lower BER) compared to fewer iterations (N = 2, N = 1). This improvement comes at the cost of increased computational complexity and execution time. Therefore, while more iterations can enhance the decoding accuracy of SOVA, it is essential to balance this with the available computational resources and the desired latency requirements of the system.

We must also point out that while conducting this comparison, each number of decoding iterations was matched with its best-performing extrinsic information normalization. The scaling and thresholding of the soft information, which is exchanged between decoders 2×N times, naturally requires different configurations for each iteration count. This ensures that the comparison accurately reflects the optimal performance achievable with varying numbers of iterations.

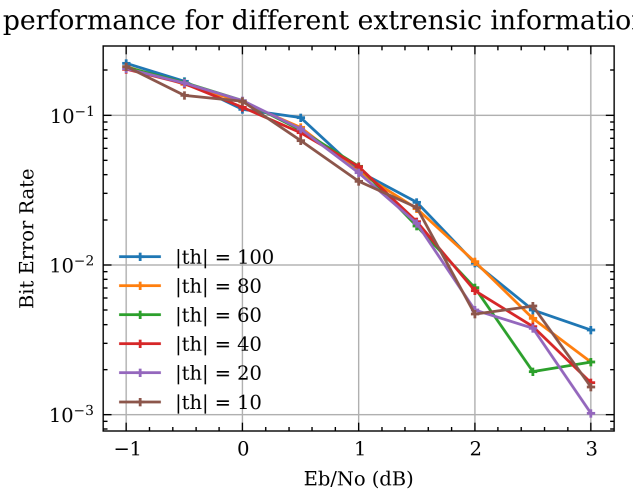
1.3.2.5.5 Effect of the extrinsic information normalization

The process of normalization adjusts the extrinsic information values to an optimal range, ensuring that they accurately reflect the reliability of the decoding. This adjustment is crucial, as improper normalization can lead to either an underestimated or overestimated reliability. Two primary parameters in soft information normalization are scaling and thresholding. Scaling modifies the amplitude of the soft information, while thresholding defines the limits beyond which values are clipped to prevent extreme fluctuations.

The BER performance comparison based on the extrinsic information scaled thresholding, with block length fixed at $K = 100$ and a number of iterations of $N = 6$ is presented in Figure 1.20.



(a) scaling



(b) thresholding

Figure 1.20: Comparison of SOVA BER performance for different extrinsic information normalization schemes.

As shown in 1.20, The scale (s = 0.75) offers the best performance (lowest BER) across all E_b/N_0 values, while the smallest scale (s = 0.25) shows the worst performance. Performance differences across thresholds are less pronounced compared to the other parameters, but generally, lower thresholds ($|th| = 10, |th| = 20, |th| = 40$) slightly outperform higher thresholds ($|th| = 80, |th| = 100$). We also address that threshold values are highly correlated to the scaling value. A larger scaling factor requires a lower clipping point to prevent the explosion of soft information values, whereas a smaller scaling factor benefits from a higher threshold to ensure values converge gradually within an optimal range, particularly over a larger number of decoding iterations.

1.3.2.5.6 Considerations for upcoming work

In the upcoming work, we fix the block length to the value of $K = 100$. This value was chosen because of its common use in LTE standard and in research on channel coding. The

corresponding quadratic permutation polynomials is $\pi(x) = (34x + 63x^2)\text{mod}(100 - 2)$. We note that the interleaving is not applied on the tail bits, meaning that the last $\nu = 2$ tail bits are not interleaved.

The BER of the selected configuration is reviewed in Figure 1.21. The shown performance represents the key point of comparison in Chapters X and Y, as it represents the SOVA algorithm with the best-averaged performance over a selected signal-to-noise ratio range. The stated SOVA configuration parameters are listed below.

- Number of decoding iterations of $N = 6$.
- Soft information scaling with a factor $s = 0.75$.
- Soft information threshold at $|th| = 40$.

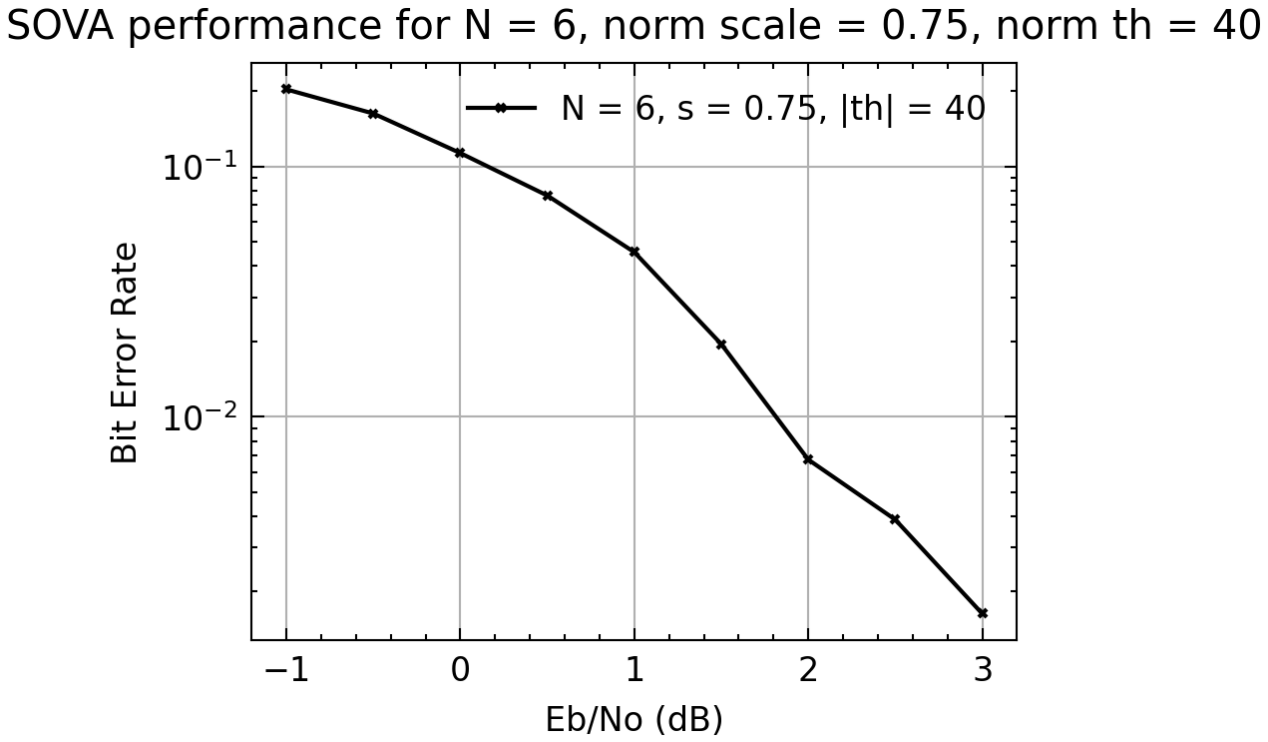


Figure 1.21: SOVA BER performance for the selected configuration.

Conclusion

A general overview of the theory related to channel coding has been explained in detail, with an explanation of convolutional codes, turbo codes, and iterative turbo decoder SOVA, plus MATLAB simulation results to examine the different parameters that influence the performance of the turbo decoder.

In the next chapter, we get more into the details related to the subject, when we introduce neural networks and state-of-the-art works on turbo decoding using machine learning techniques

Chapter 2

Neural Networks for Turbo Decoding

Introduction

Communication standards generally specify a fixed encoder as a capacity-approaching code, while permitting stakeholders to create their own decoders [4], making the development of highly reliable, robust, and adaptive decoding algorithms crucial for both industry and academia. Among various standard codes, turbo codes are extensively used in modern communication systems. The standard Turbo decoder utilizes the iterative BCJR and SOVA algorithms along with an interleaving decoding process [18]. Historically, turbo codes were the first channel coding scheme to achieve capacity-approaching performance under AWGN channels [1].

Since the 1990s, communication engineering specialists have shown significant interest in using deep learning methods to decode convolutional and turbo codes, exploring various neural network architectures (e.g., FFNN, RNN, LSTM) to evaluate their performance in terms of error rate, time complexity, and hardware complexity compared to iterative turbo decoders.

In this chapter, in the first part, we explain the different types of deep neural networks and their different architectures and applications, which gives an overview and helps to introduce the second part. In the second part, we examine various methods and architectures detailed in research papers and articles focusing on deep learning techniques for decoding convolutional and turbo codes. Each section of this chapter discusses a different type of deep learning architecture, analyzing its specific contributions and performance.

2.1 Neural networks

In this section, we introduce fundamental concepts of neural networks, with a specific focus on architectures that exhibit strong potential for application in channel decoding. Various neural network architectures will be discussed in sufficient detail. This foundational understanding will pave the way for the subsequent chapter, where neural network models will be proposed for the decoding of turbo codes.

2.1.1 Feed Forward Neural Networks (FFNN)

A feedforward network is a *multilayer network* where units are connected without cycles; each layer's outputs are passed to the next higher layer without feedback [19].

Feedforward networks consist of three types of nodes: *input units*, *hidden units*, and *output units*. Figure 2.1 shows a typical structure. The input layer $\mathbf{x}$ is a vector of scalar values. The core of the network is the hidden layer $\mathbf{h}$, comprising *hidden units* h_i. Each hidden unit h_i takes a weighted sum of its inputs and applies a non-linearity [19].

In a *fully-connected* architecture, each unit in one layer connects to all units in the previous layer. Thus, each hidden unit sums over all input units. The parameters for the entire hidden layer are represented by a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$. The weight matrix $\mathbf{W}$ has elements W_{ji} representing the weight from the i-th input unit x_i to the j-th hidden unit h_j. This allows efficient hidden layer computations using simple matrix operations: multiplying the weight matrix by the input vector $\mathbf{x}$, adding the bias vector $\mathbf{b}$, and applying the activation function g. For instance, using the sigmoid function σ, the hidden layer output is given by:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.1}$$

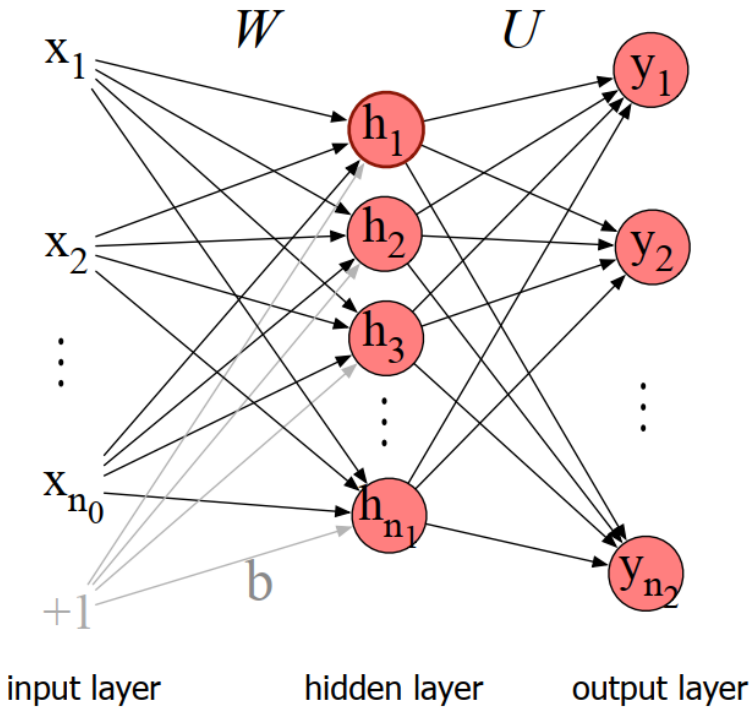


Figure 2.1: A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer [19].

Here, σ is applied element-wise. Let n_0 be the number of inputs, n_1 the number of hidden units, and n_2 the number of output nodes. The input vector $\mathbf{x}$ is a column vector of dimension $[n_0, 1]$. The hidden layer vector $\mathbf{h}$ and bias vector $\mathbf{b}$ are columns vectors of dimension $[n_1, 1]$. The weight matrix $\mathbf{W}$ with dimension $[n1, n0]$ [19].

The hidden layer output $\mathbf{h}$ represents the input in a new form. The output layer computes the final output, often for classification tasks. For binary tasks, a single output node represents the probability of positive versus negative sentiment. For multinomial classification, each output node represents the probability of a specific class, with all output values summing to one [19].

The output layer has a weight matrix $\mathbf{U}$ (without a bias vector for simplicity) of a dimension

$[n_2, n_1]$. The intermediate output $\mathbf{z}$ is:

$$\mathbf{z} = \mathbf{U}\mathbf{h} \tag{2.2}$$

The output vector $\mathbf{y}$ is a probability distribution obtained by normalizing $\mathbf{z}$ using the *softmax* function:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{d} \exp(z_j)} \quad \text{for} \quad 1 \leq i \leq d \tag{2.3}$$

A neural network classifier with one hidden layer computes $\mathbf{h}$ as a representation of the input and then uses multinomial logistic regression on $\mathbf{h}$. Unlike traditional feature engineering, the network itself induces the feature representations [19].

The final equations for a feedforward network with a single hidden layer are:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.4}$$

$$\mathbf{z} = \mathbf{U}\mathbf{h} \tag{2.5}$$

$$\mathbf{y} = \text{softmax}(\mathbf{z}) \tag{2.6}$$

The activation functions *sigmoid* and *tanh* are give as follows respectively :

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

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.8}$$

2.1.2 Recurrent Neural Network (RNN)

Recurrent Neural Networks (RNNs) revolutionized the use of neural networks for sequential data. Early RNN models, like the Elman and Jordan networks introduced in 1990 [20], extended feedforward neural networks by processing sequences one step at a time using internal feedback loops. Unlike feedforward networks, RNNs can produce outputs at each step and rely on inputs and outputs from previous steps, utilizing an internal memory state, or hidden state, connected by a delay function. This allows RNNs to exploit temporal dependencies, making them effective for discrete data [21].

RNNs consist of an input vector x_t, a hidden state h_t, and an output vector y_t. The hidden state at step t, h_t, is a function of the current input x_t and the previous hidden state h_{t-1} :

$$h_t = f_h(x_t, h_{t-1}) \tag{2.9}$$

The output y_t at step t is a function of the hidden state:

$$y_t = f_y(h_t) \tag{2.10}$$

. This process, illustrated in Figures 2.2 and 2.3, unrolls the network over time.

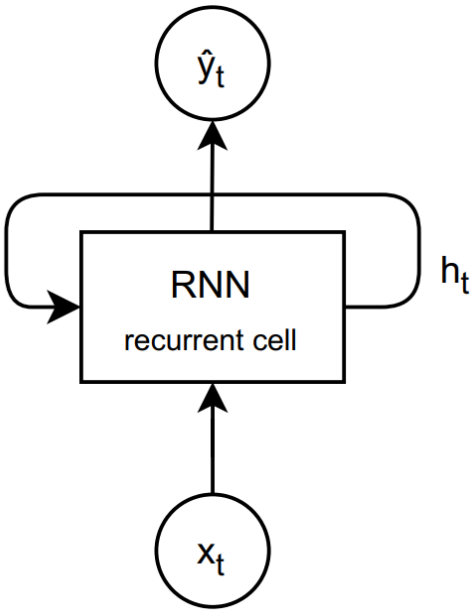


Figure 2.2: Schematic of a recurrent network, with an input, output, and a delay function h_t [21].

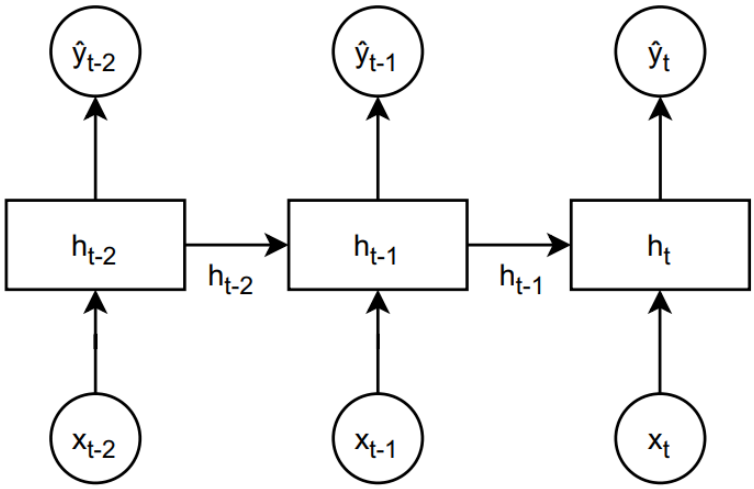


Figure 2.3: Schematic overview of the recurrent network, unfolded [21].

A standard RNN computes a sequence of classifications $(y_1, \cdots, y_T)$ from a sequence of inputs $(x_1, \cdots, x_T)$ using:

$$h_t = \phi_h(W_h x_t + U_h h_t - 1 + b_h) \tag{2.11}$$

$$y_t = \phi_y(W_y h_t + b_y) \tag{2.12}$$

where ϕ_h and ϕ_y are activation functions, and W,U, and b are model parameters. Common

activation functions include the hyperbolic tangent ($tanh$) for ϕ_h and the identity function for ϕ_y. To produce a discrete probability distribution over classes, a softmax function is applied to the output, and cross-entropy loss is used to compute the error between the predicted and target classes [21].

$$y_t = softmax(y_t) \tag{2.13}$$

RNNs are trained using stochastic gradient descent (SGD) and backpropagation through time (BPTT), which unrolls the network and updates weights iteratively. This method, however, makes RNNs challenging to train for long sequences due to computational complexity and issues like vanishing and exploding gradients, which are further discussed in the context of Long Short-Term Memory (LSTM) networks [21].

2.1.2.1 Stacked RNNs

Usually, the inputs of RNNs are sequences of embedded tokens[1] (vectors), and the outputs are vectors used for predicting words, sequence labels, or turbo decoding. However, we can also use the entire sequence of outputs from one RNN as the input sequence to another RNN [22].

Stacked RNNs, consist of multiple layers, where the output of one layer becomes the input for the next. These stacked RNNs generally outperform single-layer networks. Their success likely stems from their ability to create representations at different levels of abstraction across layers. The initial layers of stacked RNNs form useful abstractions for subsequent layers. These abstractions would be difficult to achieve with a single RNN [22].

The optimal number of stacked layers varies by application and training set, but increasing the number of layers also increases training costs significantly [22].

2.1.2.2 Bidirectional RNNs

RNNs traditionally use left-to-right context to make predictions at time t. However, the entire input sequence is available in many applications, allowing us to utilize context from both sides. One method is to use two separate RNNs: one processing the sequence left-to-right and another right-to-left, then concatenating their outputs [22].

In a left-to-right RNN, the hidden state at time t (h_t^f) captures all information from the start

[1] In Sequence models like RNNs, embedded tokens are dense vector representations of words or characters. They enable the RNN to understand and process sequences more effectively. By converting tokens into low-dimensional continuous vectors, the RNN can leverage rich semantic information, improving performance in tasks like language modeling and decoding sequences.

up to t:

$$h_t^f = \text{RNNforward}(x_1, ..., x_t) \tag{2.14}$$

To leverage right-side context, a right-to-left RNN processes the sequence in reverse, with the hidden state at t (h_t^b) capturing information from t to the end:

$$h_t^b = \text{RNNbackward}(x_t, ..., x_n) \tag{2.15}$$

A bidirectional RNN combines these two RNNs, processing inputs forward and backward. The hidden states from both directions are concatenated to form a comprehensive representation:

$$h_t = [h_t^f; h_t^b] = h_t^f \oplus h_t^b \tag{2.16}$$

This concatenation captures information from both the left and right contexts of an input at each point in time, as illustrated in Figure 2.4. Other methods like element-wise addition or multiplication can also combine these contexts [22].

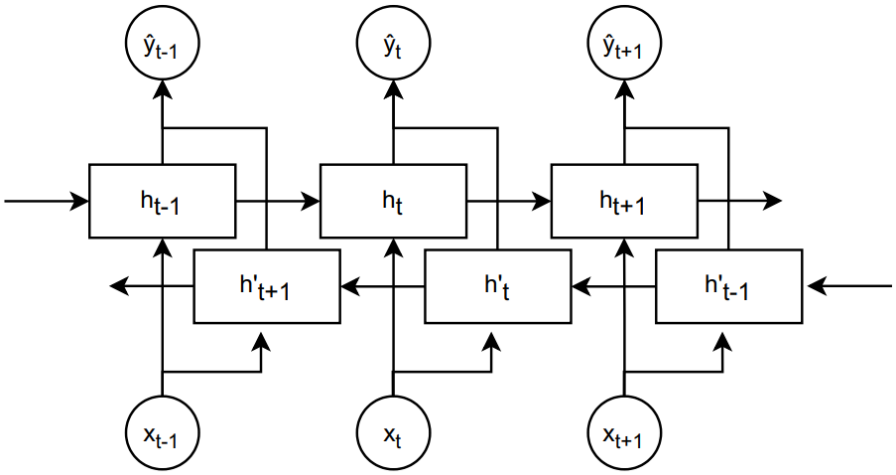


Figure 2.4: Schematic of a bidirectional RNN layer [21].

2.1.3 Long Short Term Memory (LSTM)

While plain RNNs can theoretically model any temporal relationships within sequences given sufficient hidden units, in practice, they struggle to train effectively when long-term dependencies are present. This difficulty is mainly due to the vanishing gradient problem, where gradients either vanish or explode as they flow through the network. This issue arises because the dimensionality of the neural network is proportional to both the number of neurons and the number of steps. As a result, RNNs tend to have a limited memory range, effectively making them short-term [21].

To address these stability issues and improve the modeling of long-term dependencies, gating

mechanisms have been proposed. These mechanisms guide gradient flow and protect the internal state by controlling when and how values are updated and produced. The Long Short-Term Memory (LSTM) network is a well-known example of a gated RNN. It replaces a standard RNN cell with one that includes specific connections to better manage gradient flow [21].

LSTM introduces a cell state c_t with three gates: a forget gate, an input gate, and an output gate. These gates control the information added to and removed from the cell state:

1. **Forget gate**: Determines which information is kept or discarded by applying point-wise multiplication of the cell state values with a zero or one. The forget gate f_t is calculated as:

$$f_t = \sigma(W_f h_{t-1} + W_f x_t + b_f) \tag{2.17}$$

2. **Input gate**: Decides what information is added to the cell state based on a candidate value $\tilde{c}_t$ and a calculated input value i_t, both derived from the last hidden state h_{t-1} and the current input x_t:

$$\tilde{c}_t = \tanh(W_{\tilde{c}} h_{t-1} + W_{\tilde{c}} x_t + b_{\tilde{c}}) \tag{2.18}$$

$$i_t = \sigma(W_i h_{t-1} + W_i x_t + b_i) \tag{2.19}$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \tag{2.20}$$

3. **Output gate**: Determines the output at each step based on the cell state, input, and hidden state at time t:

$$o_t = \sigma(W_h h_{t-1} + W_h x_t + b_o) \tag{2.21}$$

$$h_t = o_t \cdot \tanh(c_t) \tag{2.22}$$

Figure 2.5 depicts the schematic of an LSTM cell.

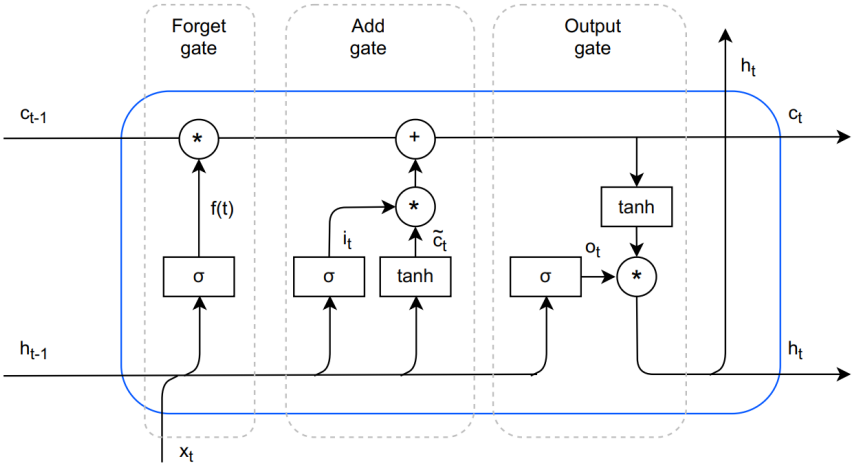


Figure 2.5: Schematic of an LSTM cell with a forget gate, add gate and output gate illustrated from left to right [21].

LSTMs significantly enhance the temporal modeling capabilities of RNNs by efficiently gating information. This improvement has made LSTMs the preferred choice for many applications in language processing and other domains.

Another type of gating mechanism is the Gated Recurrent Unit (GRU), which is a simplified version of the LSTM. GRUs have only two gates and are less computationally complex but also less powerful. However, this work focuses on RNNs with LSTM cells due to their superior performance and robustness.

2.1.4 Training neural networks

Neural networks, with their deep architectures and large weight matrices, have millions of parameters initialized randomly before training. Training involves using batches of examples from the training set, where the network predicts outputs $\hat{y}$ for each example. In supervised learning, known correct labels y help evaluate prediction accuracy through a cost or loss function $\mathcal{L}$. The gradient of this cost function guides the adjustment of parameters: decrease for positive gradients and increase for negative ones. This process is repeated for each batch until a predefined number of iterations is reached or parameter values converge. Performance is also monitored using a development set or validation set, distinct from the training set, to prevent overfitting and ensure the model generalizes well. Overfitting occurs if the model memorizes the training set labels instead of learning to generalize on unseen data [23].

2.1.4.1 Loss function

To train a Recurrent Neural Network (RNN) as a language model, we use a self-supervision algorithm. This involves using a set of sequences as training data and asking the model to predict the next element in the sequence at each time step t. The model is considered self-supervised because it doesn't require additional labeled data; the sequence itself provides the necessary supervision. The training objective is to minimize the error in predicting the true next token using cross-entropy as the loss function, which measures the difference between the predicted probability distribution and the correct distribution [22].

The cross-entropy loss is defined as:

$$\mathcal{L}_{CE} = -\sum_{w \in V} y_t[w] \log \hat{y}_t[w] \tag{2.23}$$

For turbo decoding, where the goal is to predict the next bit in a sequence, the correct distribution y_t comes from the known value of the next bit. The loss function used is binary cross-entropy (BCE), treating the problem as a sequence of binary classification tasks where

each bit is classified as 0 or 1. The BCE loss, suited for this type of application, is determined by the probability the model assigns to the correct next-bit value. The binary cross entropy is defined as :

$$\mathcal{L}_{BCE} = -\frac{1}{N}\sum_{t=1}^{N}\left[y_t \log(p_t) + (1-y_t)\log(1-p_t)\right] \tag{2.24}$$

where:

- N is the number of observations.

- y_i is the actual binary label (0 or 1) of the i-th observation.

- p_i is the predicted probability of the i-th observation being in class 1.

The network weights are adjusted to minimize the average BCE loss over the training sequence via gradient descent.

2.1.4.2 Optimization

To optimize neural networks efficiently, several key factors are crucial: initializing parameters correctly, ensuring all functions are differentiable, and implementing effective weight updates.

Initialization At the start of training, model parameters must be initialized. Common methods include:

- **Random Isotropic Gaussian Initialization:** Uses a normal distribution with mean 0 and low standard deviation (e.g., 0.01).
- **Xavier Initialization:** it uses a uniform distribution with boundaries $\pm\sqrt{\frac{6}{n_j+n_{j+1}}}$, where n_j and n_{j+1} are the number of neurons in the current and previous layers. This keeps gradient variance stable, aiding training.

It's important not to initialize all weights to the same value to ensure asymmetry, allowing neurons to learn different functions [23].

Backpropagation Backpropagation computes gradients needed for parameter updates. For multiple neuron layers, gradients are calculated using the chain rule. In recurrent neural networks (RNNs), gradients are computed through backpropagation through time (BPTT), which processes gradients backward from the last time step to the first. Frameworks like Theano can automate this computation [23].

Weight Update Parameter updates are based on the computed gradients. The update size impacts the training speed and the quality of the local optimum found. Too small updates lead to slow convergence, while too large updates risk overshooting the optimum [23].

Several optimization techniques manage update sizes:

- **Stochastic Gradient Descent (SGD):** Updates parameters using the formula

$$\Delta w_t = -\alpha \cdot \frac{\partial E}{\partial w_t}, \tag{2.25}$$

  where α is the learning rate. The pseudo-code for SGD is shown in Algorithm 1.

- **Momentum:** Enhances SGD by considering past gradients to adjust the current update, using

$$\Delta w_t = \beta \cdot \Delta w_{t-1} - \alpha \cdot \frac{\partial E}{\partial w_t}, \tag{2.26}$$

  where β is the decay rate. The pseudo-code for momentum is shown in Algorithm 2.

- **Adaptive Moment Estimation (Adam):** Combines momentum with an exponentially decaying average of past squared gradients. This technique adjusts the learning rate dynamically but has potential convergence issues. The pseudo-co de for Adam is shown in Algorithm 3. Adam optimizer is currently often used, we used it to optimize the proposed model in Chapter 3.

By effectively initializing parameters, using backpropagation to calculate gradients, and employing robust weight update methods, neural networks can be optimized efficiently, achieving fast and reliable convergence.

Algorithm 1 Pseudo-code of SGD.

Require: α: learning rate
Require: θ_0: parameter set
Require: $f(\theta)$: network function
1: $t \leftarrow 0$
2: **while** θ_t not converged **do**
3: $t \leftarrow t + 1$
4: $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
5: $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot g_t$
6: **end while**
7: **return** θ_t

Algorithm 2 Pseudo-code of momentum.

Require: α: learning rate
Require: $\beta \in [0, 1]$: decay rate
Require: θ_0: parameter set
Require: $f(\theta)$: network function
$w_0 \leftarrow 0$
$t \leftarrow 0$
while θ_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $\Delta w_t \leftarrow \beta \cdot \Delta w_{t-1} - \alpha \cdot g_t$
  $\theta_t \leftarrow \theta_{t-1} + \Delta w_t$
end while
return θ_t

Algorithm 3 Pseudo-code of Adam.

Require: α: learning rate
Require: $\beta_1, \beta_2 \in [0, 1)$: decay rate
Require: θ_0: parameter set
Require: $f(\theta)$: network function
$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
$t \leftarrow 0$
while θ_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$
end while
return θ_t

2.1.5 Sequence to Sequence models

We present this type of model and then introduce the notion of attention and how it can be adapted for use in sequence-to-sequence models.

2.1.5.1 Sequence to Sequence Model

One limitation of single recurrent neural networks (RNNs) such as LSTMs is their difficulty in mapping input sequences to output sequences of different lengths without post-processing. Previously, there was no general end-to-end method for sequence-to-sequence classification tasks,

such as segment classification with $k \neq T$ and temporal classification, without making strong assumptions about the data, like the monotonicity of alignment [21].

The development of a general end-to-end approach for sequence-to-sequence mapping began with the introduction of the sequence-to-sequence (seq2seq) model [24] in 2014. Although any model capable of learning to map one sequence to another can be termed a sequence-to-sequence model, the term specifically refers to a model that adopts an encoder-decoder architecture [21].

In the seq2seq model, the encoder and decoder are composed of separate RNNs, typically LSTMs. The encoder processes the input sequence to the end, producing a final hidden state h_T, which the decoder then uses as its initial state to predict the target sequence [21].

During inference, the decoder's prediction at time step t, $\hat{y}_t$, is fed as input to the next time step, such that:

$$x_t = \hat{y}_{t-1}. \tag{2.27}$$

Thus, seq2seq models can be autoregressive, meaning the output at each step y_t depends on previous outputs $y_{t-1}, y_{t-2}, y_{t-3}$, and so on. This dependence models the relationships in sequential data, assuming that consecutive data points are not independently and identically distributed (i.i.d.) [21].

To predict the target sequence, a 'start of sequence' (SOS) symbol is provided to the first step of the decoder or zero-initialized input, and outputs are generated conditionally until the end of the sequence. This allows the generation of output sequences of arbitrary length. The hidden state of the decoder at time step t is denoted as h'_t [21].

Using this model, the probability of a source/target pair (x, y), $P(y|x)$, can be approximated as follows:

$$P(y|x) = P(y_k, \ldots, y_1|x_T, \ldots, x_1) = \prod_{t=1}^{k} P(y_t|y_{t-1}, \ldots, y_1, x). \tag{2.28}$$

Therefore, at each time step, we can choose the token that maximizes the joint likelihood:

$$\arg\max_{y_t \in y} P(y_t|y_{t-1}, \ldots, y_1, x). \tag{2.29}$$

During training, rather than using the decoder's prediction as input for the next time step, the ground truth target sequence is provided step by step, starting with the SOS symbol. For each step $t > 1$, the input to the decoder, x_t, represents the true target of the preceding step, y_{t-1}. This method, known as teacher forcing [25], trains the model conditionally on the correct previous value.

Figure 2.6 illustrates an autoregressive sequence-to-sequence model that employs an encoder-decoder architecture. In this model, the decoder receives the encoder's hidden states and its own output $\hat{y}_t$ from the previous time step as input for each subsequent step, except at the initial step where it is provided with a start-of-sequence (SOS) symbol, or it can be a zero-initialized input.

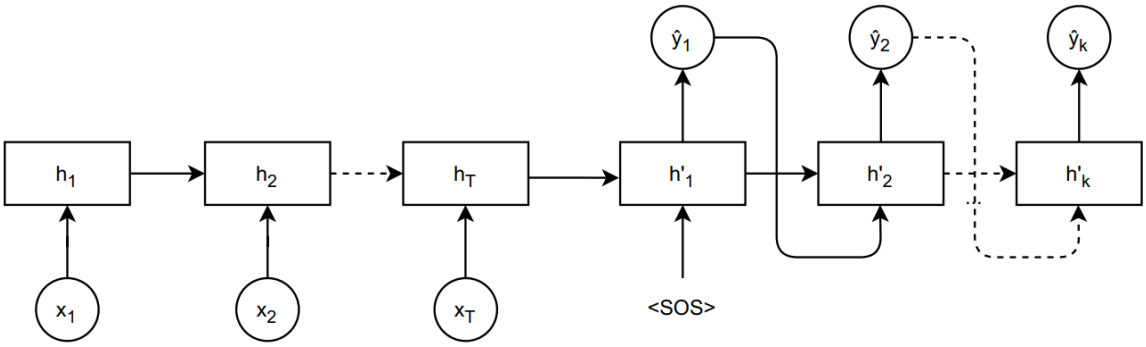


Figure 2.6: Schematic of an autoregressive sequence to sequence model [21].

The flexible structure of the seq2seq model makes it effective for a variety of sequence-to-sequence tasks, notably machine translation, video description, and others. Seq2seq models also apply to time series predictions, such as stock market and weather forecasting.

Scheduled sampling While using the decoder's own predictions ($\hat{y}_{t-1}$) instead of the ground truth (y_{t-1}) during training can lead to unstable training, teacher forcing mitigates this by always providing the correct previous value from the target sequence. However, this can result in the decoder being poorly trained to handle its own mistakes during inference, leading to exposure bias where errors are amplified over time.

Scheduled sampling [26] addresses this by gradually shifting from using the ground truth to using the model's own predictions during training. At each step t, the probability of choosing the true previous value y_{t-1}, denoted as ϵ_t, changes according to a predefined schedule. This approach exposes the model to its own predictions progressively, aiming to minimize the performance gap between training and inference without adding significant overhead to training times.

Beam search Greedy decoding, which selects the most likely output at each time step, can lead to error accumulation in autoregressive sequence-to-sequence models. Beam search mitigates this by maintaining a set of candidate sequences (the beam) and updating them iteratively based on the predicted probabilities, thereby maximizing the joint likelihood of the sequence.

Beam search increases the likelihood of finding a reasonable result by exploring a wider search

space than greedy decoding. A typical beam size of 5-10 is most effective for well-trained decoders. However, it should be viewed as an optimization technique that minimizes decoding errors in a well-trained model, rather than a method to correct a poorly trained one.

2.1.5.2 Sequence to Sequence attention model

One of the limitations of plain sequence-to-sequence models is that the entire input sequence is compressed into a single vector—the last hidden state of the encoder—which is then passed to the decoder. This creates a challenge for the encoder, as it must compress the entire input sequence into one vector using gradients flowing through the network. The decoder's LSTM relies entirely on this single hidden state to access the memory of the encoder's LSTM, despite different information being relevant at different time-steps[27]. While, theoretically, with enough neurons and time, relationships of arbitrary complexity could be trained using a standard seq2seq model, in practice, this compression restricts the flexibility and strength of the decoder's memory, making it difficult to train on longer sequences. Repeating the encoder's last hidden state at each decoder time step might offer slight improvement, but the overall memory and temporal dependency transmission remain weak [21].

To address this, an attention mechanism [27, 28] was introduced, allowing the decoder's LSTM flexible access to the encoder's memory. This concept is analogous to human attention: when focusing on a specific part of an image or a set of words, we extract detailed information from that focal point while the surrounding area remains slightly out of focus. Similarly, an attention mechanism directs the model to different parts of the input at different times during decoding. At each decoding step, the mechanism estimates which hidden representations of the encoder are most relevant [22].

This is achieved by providing the decoder with a sequence of vectors (the hidden states at each time step of the encoder) instead of a single vector. At each decoder time-step y_t, the attention mechanism combines the encoder's hidden states $(h_1, \ldots, h_T)$ into a context vector c_t by weighting each hidden state h_i with an attention weight α_{ti} for that step:

$$c_t = \sum_{i=1}^{T} \alpha_{ti} h_i \tag{2.30}$$

These attention weights α are computed using the attention layer bottleneck between the encoder and the decoder, this layer combines the activations from all hidden vectors with the hidden vector from the last time-step in the decoder h'_{t-1} to compute the attention scores. Then a softmax function normalizes these scores to give the weights that sum to one.

First, an attention score e_{ti} for each hidden state h_i is calculated based on the current decoder

time-step t:

$$e_{ti} = \phi(h'_{t-1}, h_i) \tag{2.31}$$

While ϕ is a predefined function called *score function*.

Then, the attention weights α_{ti} are determined as:

$$\alpha_{ti} = \text{softmax}(e_{ti}) \tag{2.32}$$

The hidden vectors from the encoder are then multiplied with the attention weights α_{ti}, and the result is a context vector containing useful activations for every step of the LSTM decoder[22].

At a given time step t, the decoder receives as input the concatenation of the context vector c_t and the decoder's current input. The decoder's input can be either the actual target y_t (in the case of teacher forcing) or the highest probability output from the previous state y_{t-1} (in Greedy Search). For the initial time step, the input to the decoder can be the Start of Sequence (SOS) symbol or a zero-initialized input, which signifies the absence of information.

Attention score function

The score function ϕ in Equation 2.31 can take various forms depending on the application and the required model complexity. Below, we highlight some of the most commonly used score functions:

- **Dot product score**

  The simplest score, called dot-product attention, implements relevance as attention similarity, measuring how similar the decoder's hidden state is to an encoder's hidden state, by computing the dot product between them:

  $$\phi(h'_{t-1}, h_i) = h'_{t-1} \cdot h_i \tag{2.33}$$

  The score that results from this dot product is a scalar that reflects the degree of similarity between the two vectors. The vector of these scores across all the encoder hidden states gives us the relevance of each encoder state to the current step of the decoder.

  Another variant of this function is called the *scaled dot score*, when the product is divided by $\sqrt{d_{h'}}$, while $d_{h'}$ is the dimension of the decoder's hidden state h'_{t-1}. Thus, the score function becomes as follows :

  $$\phi(h'_{t-1}, h_i) = \frac{h'_{t-1} \cdot h_i}{\sqrt{d_{h'}}} \tag{2.34}$$

  The scaling factor of $\sqrt{d_{h'}}$ is introduced to reduce the magnitude of the scores and decrease the chances of encountering vanishing gradients. [22]

- **Bilinear score**

  A more powerful function that computes the relevance of each encoder hidden state to the decoder hidden state by parameterizing the score with its own set of weights, $\mathbf{W}$. The weights $\mathbf{W}$, which are then trained during normal end-to-end training, allow the network to learn which aspects of similarity between the decoder and encoder states are important to the current application. This bilinear model also allows the encoder and decoder to use different dimensional vectors, whereas the simple dot-product attention requires that the encoder and decoder's hidden states have the same dimensionality. [22]

  The bilinear score function is given as follows:

$$\phi(h'_{t-1}, h_i) = h'_{t-1}\mathbf{W}h_i \tag{2.35}$$

Figure 2.7 represents a schematic of the attention mechanism in an encoder-decoder model. The inputs $\hat{y}_t$ to the decoder are provided using the Greedy Search technique. While this method is primarily used during inference, it can also be beneficial during training, particularly to mitigate overfitting caused by teacher forcing.

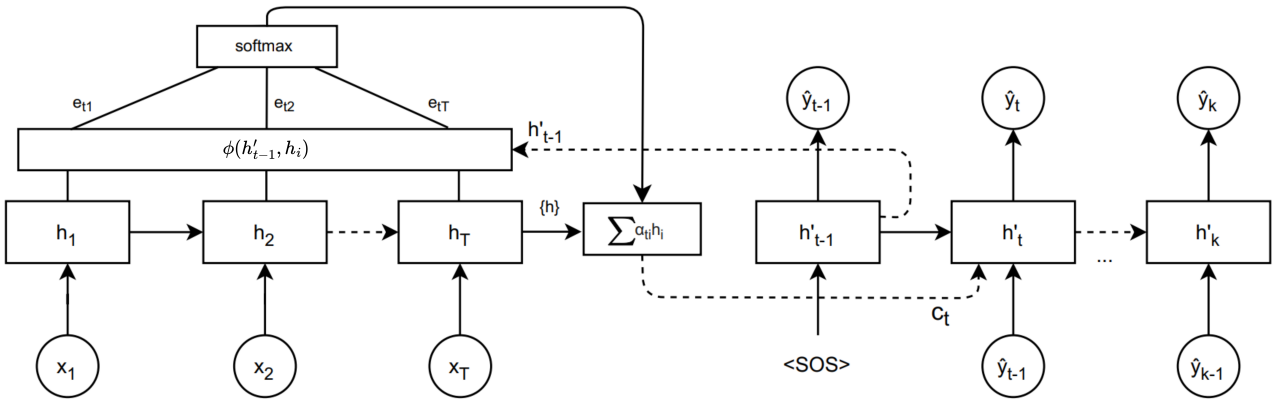


Figure 2.7: Attention mechanism illustrated for an encoder-decoder model with two LSTMs: h and h' [21].

2.2 State of the art

In this investigate in this part the most significant works that has been done in using machine learning models for tubo decoding, partitioned to the types of models.

2.2.1 Feed Forward Neural Networks

Feed-forward artificial neural networks (FFNNs or ANNs) are one of the basic architectures of deep learning models, and it's used in multiple ways and applications, notably wireless

communications.

Caid and Means [29] proposed in their work the use of neural networks for error correction in Electronic Counter Measures (ECM) environments, where traditional assumptions like additive white Gaussian noise (AWGN) and binary symmetric channels (BSC) fail. By training feed-forward neural networks with backpropagation to correct corrupted code vectors, the method shows superior performance to conventional decoders. For example, neural decoders outperformed hard decision decoders for both Hamming $(7, 4)$ block codes and $k = 3$, rate $1/2$ convolutional codes, suggesting significant improvements in coding gain, throughput, and cost for digital transmissions in ECM scenarios.

Marcone et al.[30] present a feedforward artificial neural network (FFNN) trained to decode $1/N$ rate convolutional codes. The method partitions the received signal into overlapping sliding windows, each representing a codeword, and trains the ANN to correct errors and decode the transmitted bits from noisy signals. With optimal training, the neural network decoder performs comparably to the Viterbi decoder, but its extensive training length limits its practicality for real-time applications.

The neurlaized viterbi decoder introduced by Xiao-an Wang and Stephen B. Wicker[31] is an artificial neural network Viterbi decoder, surpassing digital-only designs with its fully parallel architecture. The decoder utilizes analog neurons to implement most of the Viterbi algorithm, eliminating the need for digital circuits and simplifying the design. This results in a significantly faster decoder, requiring only one-sixth the number of transistors of a digital decoder. The authors demonstrate that the neural network Viterbi decoder can be designed with fixed weights and without training, making it suitable for VLSI implementation due to its modularity and local connectivity. Simulation results confirm its performance equivalence to an ideal Viterbi decoder.

After introducing Turbo Codes and their attractive error performance, which approached Shannon's limit, Annauth and Rughooputh [32] proposed a neural network for predicting decoder error in turbo decoders. They utilized the decoder's log-likelihood ratio (LLR) values as inputs to train the neural network, optimizing training time by using a global dataset with subsets of received sequences of specific Hamming distances. Randomly selecting 20% of this dataset yielded a neural network with 99.8% accuracy. Although slightly less performant than a MAP decoder, the neural network proved faster and less complex. This approach enhances turbo decoder performance by adapting the decoding algorithm to specific error conditions.

The adaptive soft decoder, by Rajbhandari et al.[33], proposes a novel approach to decoding convolutional codes using an FFNN. The neural network decoder uses a sliding block decoding algorithm, where the received sequence is divided into fixed-size blocks and each block is classified in a complex vector plane. The overall mechanism is illustrated in Figure 2.8. The neural network is trained using a log sigmoid transfer function to decode a 1/2 rate convolutional

code. The results show that the decoder outperforms Viterbi's 'hard' decoder but falls short of Viterbi's 'soft' decoder in terms of BER performance across the SNR range.

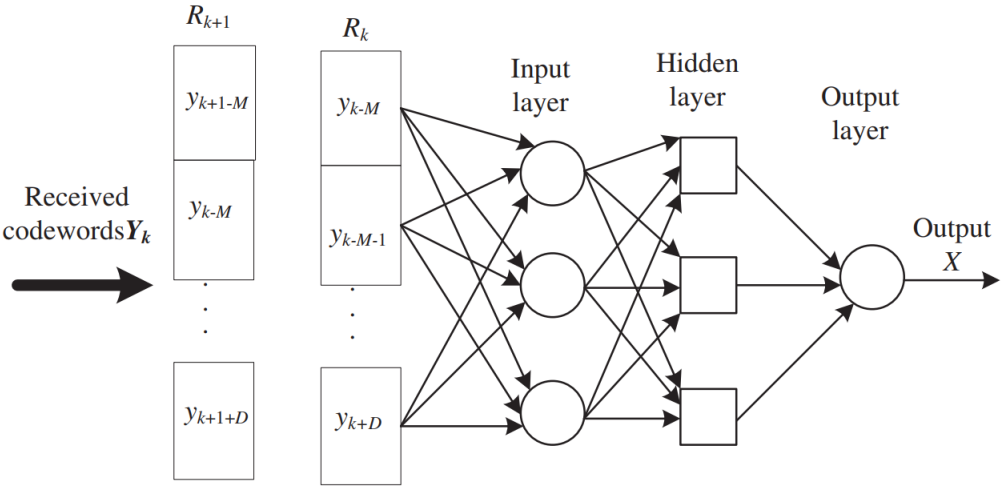


Figure 2.8: Diagram of ANN soft sliding decoder with input codewords and output bits [33].

2.2.2 Recurrent Neural Networks

The two papers, by Berber et al.[34] and Salcic et al.[35], both focus on the development and implementation of Recurrent Neural Network (RNN) decoders for convolutional codes. The first paper in 2005, introduces a novel RNN decoder that addresses a minimization problem concerning a Noise Energy Function by minimizing the Euclidean distance between received noisy bits and decoded bits using the Gradient Descent algorithm. This decoder operates without supervision, leveraging the RNN structure to exploit temporal dependencies in sequences and enhancing message estimation accuracy by optimizing the energy function. The decoder resets its parameters randomly when transitioning to decode the next message, which allows it to adapt to changing noise conditions. The authors demonstrate that this decoder exhibits superior decoding performance by enhancing the Gradient Descent algorithm's ability to find the global minimum of the Noise Energy Function through the addition of AWGN noise to estimated message bit values. The decoder was implemented using the Soft Decision Decoding method, which outperforms Hard Decision Decoding, and the RNN decoder demonstrates performance closely comparable to the Viterbi decoder.

The 2006 [35] paper discusses the Field Programmable Gate Arrays(FPGA) prototyping of this RNN decoder. The authors use a system-level design for easy mapping to the FPGA. The decoder, coupled with a simple neuron activation function, has low complexity and fits into a standard Altera FPGA. This allows for a complete test bed for prototyping RNN decoders, enabling real-time evaluation of the decoder's BER performance under various channel conditions.

Figure 2.9 shows the structure of the RNN decoder, where r is the received sequence, w repre-

sents the weights of the neurons, f is the activation function, and b is the decoded sequence.

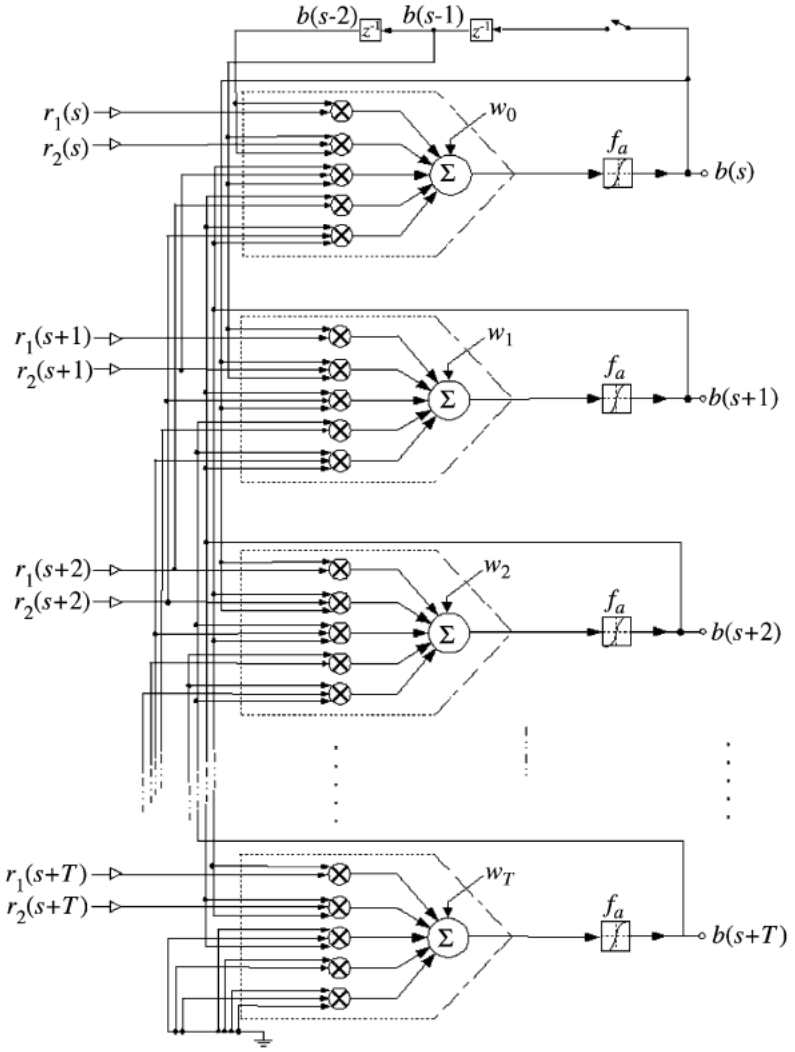


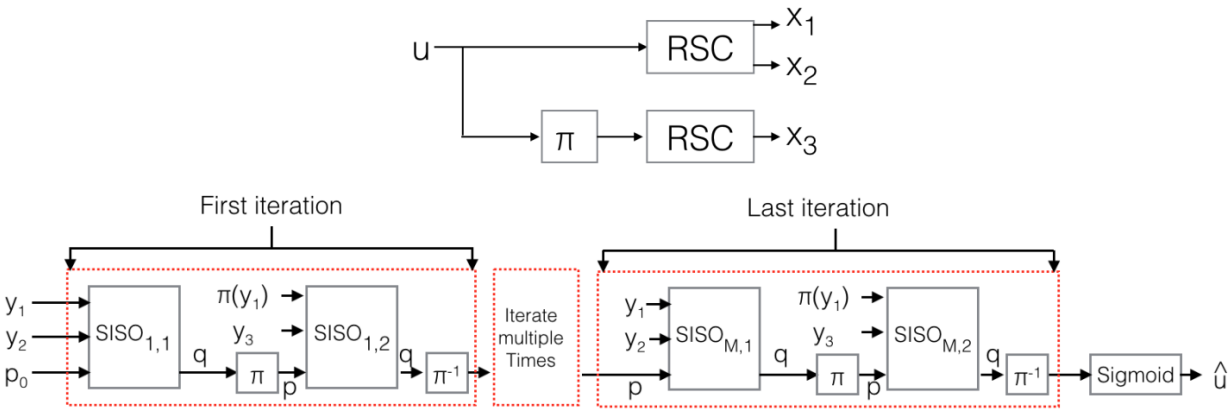
Figure 2.9: Structure of RNN Decoder [34].

2.2.3 Turbo inspired models

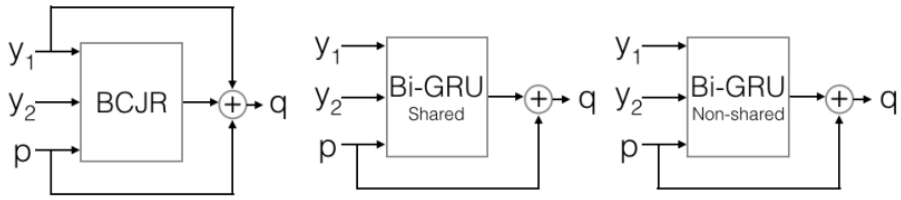
Hyeji Kim et al.[7] introduce N-Turbo, a novel neural decoder for turbo codes. N-Turbo uses multiple layers of neural decoders designed for Recursive Systematic Convolutional (RSC) codes, utilizing Recurrent Neural Networks (RNNs). Inspired by traditional turbo decoders, N-Turbo's architecture features neural RSC decoders called N-BCJR layers. Each layer is trained individually to address the difficulties of training this deep recurrent model end-to-end. These pre-trained models are then used to initialize the N-Turbo decoder, which is further trained. The N-BCJR architecture is flexible, handling various bit-wise prior distributions as input. The study shows that N-Turbo matches or exceeds the performance of conventional turbo decoders.

DEEPTURBO by Jiang et al.[36] is a novel architecture, deep learning-based approach for Turbo decoding that does not require knowledge of the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm. It uses a 2-layer Bidirectional Gated Recurrent Unit (Bi-GRU) with non-shared weights

and passes more information to the next stage. The authors, compare the (Soft Input Soft Output) SISO design of the standard turbo decoder BCJR called TURBO, NEURALBCJR, and *DEEPTURBO*, showing that *DEEPTURBO* outperforms both TURBO and NEURALBCJR in terms of bit error rate (BER) and block error rate (BLER) for both Turbo-757 and Turbo-LTE codes under both AWGN and non-AWGN channels. *DEEPTURBO* achieves superior performance with reduced decoding iterations and does not require BCJR knowledge, making it a promising solution for modern communication systems. The authors evaluate the performance of *DEEPTURBO* under both AWGN and non-AWGN channels. Figure 2.10b illustrates the Bi-GRU-based SISO decoder utilized within the turbo decoder architecture depicted in Figure 2.10a. This configuration explains the structural framework of the *DEEPTURBO* decoder.



(a) Turbo Encoder (up) and Decoder (down) [36].



(b) Different SISOs: TURBO (left), NEURALBCJR (middle) and *DEEPTURBO* (right) [36].

Figure 2.10

With a similar architecture of *DEEPTURBO*, Zhang et al.[37] worked on a parallel turbo decoder architecture long short-term memory (LSTM) networks, to improve the bit error rate (BER) performance and computational complexity of traditional turbo decoders. The LSTM decoder uses a bidirectional LSTM structure, fully connected dense layers, and custom neural network Lambda layers to simulate the iterative decoding process. The authors evaluate the performance of the LSTM decoder in both Gaussian white noise and t-distributed noise channels, comparing it with the traditional BCJR decoding algorithm. The results show that the LSTM decoder achieves better BER performance and computational efficiency compared to the BCJR algorithm in both environments, with a 0.5 dB improvement at BER = 10÷4 after 15 iterations in the Gaussian white noise channel and a 1 dB improvement at BER = 10÷4 after 15 iterations in the t-distributed noise channel. Figure 2.11 illustrates the pipeline of parallel

decoding based on bidirectional LSTM layers.

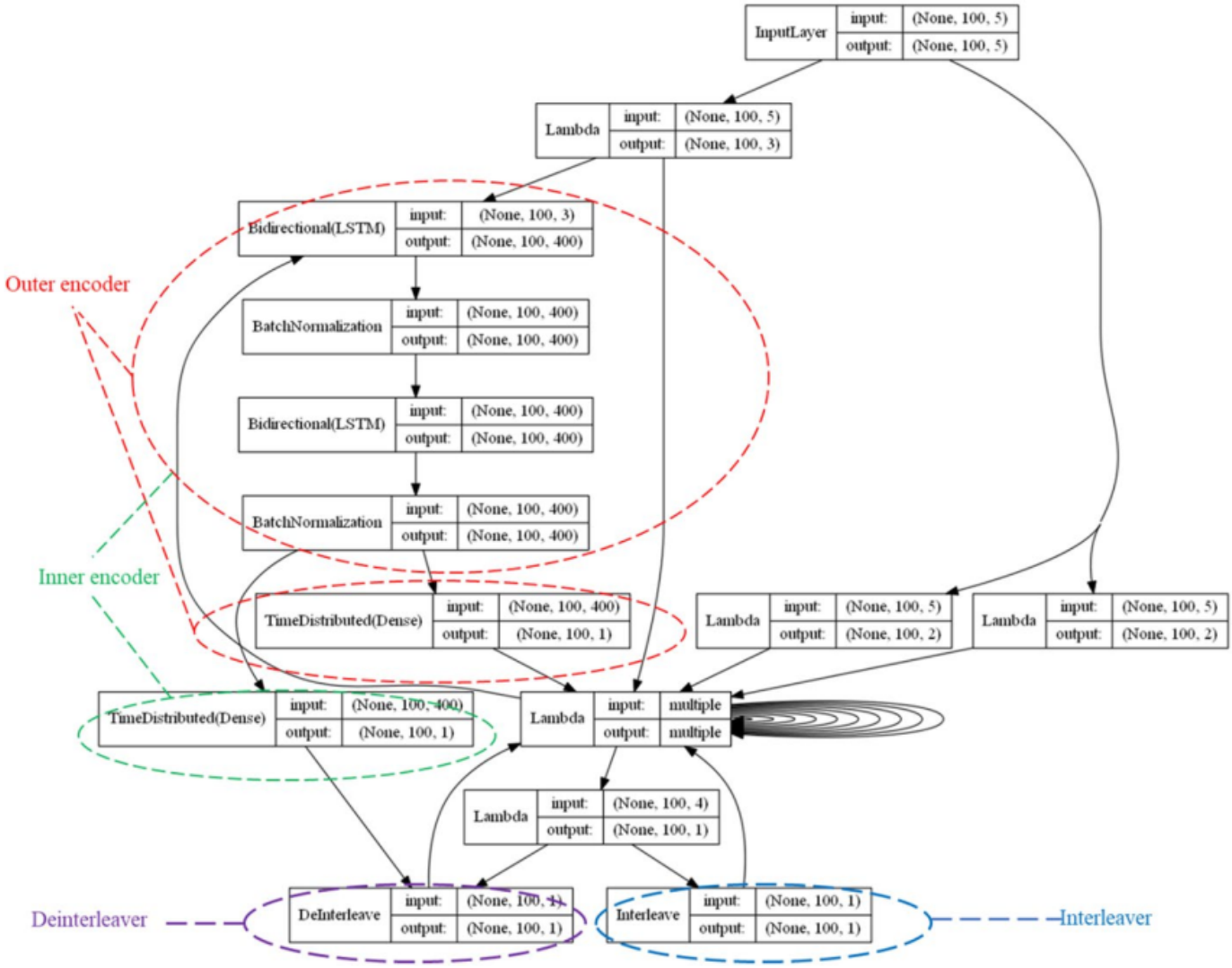


Figure 2.11: Turbo code decoding network structure [37].

2.2.4 Autoencoders

Sattiraju el al.[38] explores the use of Recurrent Neural Networks (RNNs), for turbo encoding and decoding in LTE systems. Traditional turbo codes, while effective, are computationally intensive and limit applicability in devices with constrained resources. The authors propose using Gated Recurrent Units (GRUs), a type of RNN, to learn the encoding and decoding processes. Their method frames these operations as supervised learning problems, using GRUs to process data sequences. Simulations demonstrate that the RNN-based model outperforms conventional turbo decoders in low Signal to Noise Ratio (SNR) conditions. This approach leverages the RNNs' ability to optimize end-to-end performance and efficiently handle temporal dependencies.

The novel approach introduced by Balevi and G.Andrews [39] of a channel autoencoder that

can be used to obtain optimum channel codes for any block length. The proposed method involves training an autoencoder with a one-bit quantized AWGN channel. The results show that the proposed coding scheme can achieve very close performance to turbo codes and turbo decoders that work with unquantized samples. The authors also demonstrate that the proposed method can be extended to higher-order modulations, although the performance may degrade. Figure 2.12 is the representation of the channel autoencoder.

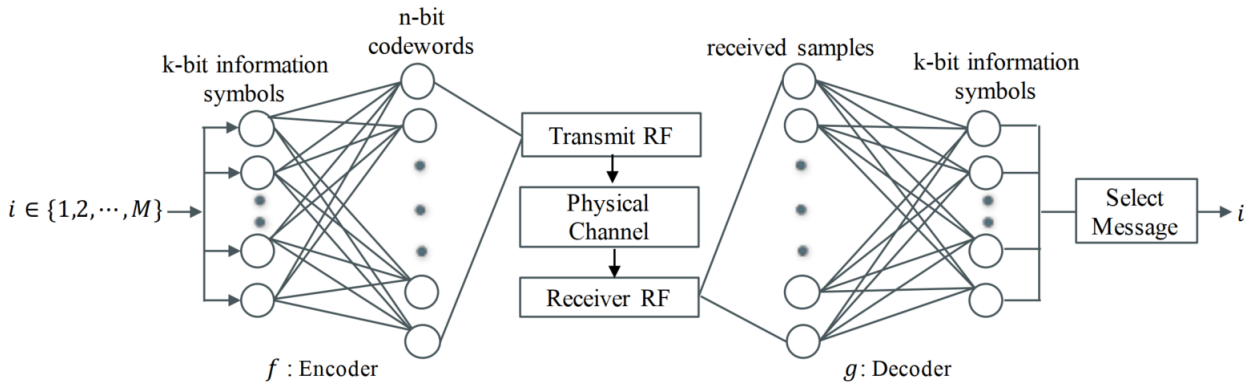


Figure 2.12: A channel autoencoder encodes a message into a k-bit sequence, maps it to a length-n codeword for transmission, and decodes the received signal to retrieve the original message [39].

2.2.5 Turbo autoencoders

In their novel work, Jiang et al.[40] introduce the Turbo AutoEncoder (TurboAE), a new neural network-based architecture for channel coding that aims to achieve performance comparable to traditional Turbo codes for Additive White Gaussian Noise (AWGN) channels. The method involves using a deep learning framework to jointly train the encoder and decoder as an autoencoder. The key innovation is the use of convolutional neural networks (CNNs) to generate parity bits and an interleaver to enhance memory and performance. The results demonstrate that TurboAE can achieve bit error rates (BER) similar to those of traditional Turbo codes under AWGN conditions, showing that deep learning can be an effective tool for designing robust channel codes. However, the study primarily focuses on AWGN channels and does not explore performance in more complex channel conditions. Figure 2.13 represents a general overview of the design of TurboAE.

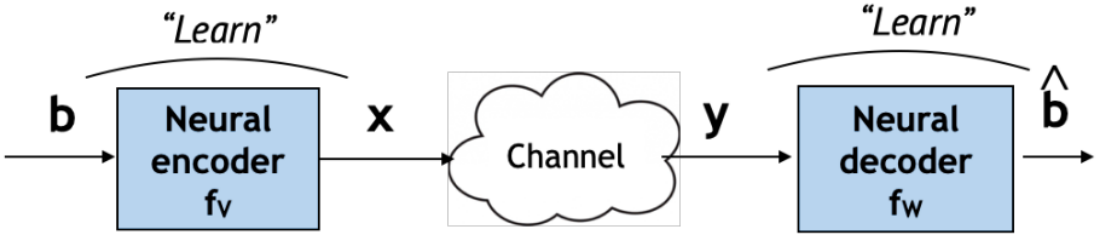


Figure 2.13: Autoencoder framework for channel coding: channel encoder and decoder are modeled as neural networks and are trained jointly [41].

Subsequently, Chahine et al.[41] , building upon the original TurboAE, introduce Turbo Autoencoder with a Trainable Interleaver (TurboAE-TI). The core contribution is the integration of a trainable interleaver within the TurboAE framework, allowing the interleaver to be optimized jointly with the encoder and decoder. This is achieved through a carefully designed training procedure and an interleaver penalty function to guide the optimization. The results show that TurboAE-TI outperforms both the original TurboAE and LTE Turbo codes in various practical channels, including fading channels and channels with bursty noise. The performance improvements are significant, with gains in reliability up to 1dB over AWGN channels. This paper highlights the adaptability and robustness of TurboAE-TI, demonstrating its superiority in more realistic communication scenarios.

Following that, a study by Hatami et al.[42] evaluates the performance of Turbo Autoencoder (TurboAE) under various interleaver configurations, including random, trainable, and optimized interleavers. The method involves systematic testing of these interleavers across different channel conditions to understand their impact on the overall coding performance. The results indicate that trainable interleavers generally offer better performance compared to random interleavers, particularly in challenging channel conditions like fading and bursty noise. However, the optimized interleavers, which are designed through an additional layer of complexity, provide the best results, showing significant improvements in BER. This paper emphasizes the critical role of interleaver design in the effectiveness of TurboAE, suggesting that future work should focus on further refining these components to enhance communication reliability.

Table 2.1 summarizes the techniques introduced previously, highlighting the neural network architecture used in each work, along with their advantages, limitations, and key performance metrics.

Method	**Architecture**	**Advantages**	**Limitations**	**Performance**
Caid and Means [29]	FFNN	Superior to conventional decoders in ECM environments	Requires extensive training	Outperforms hard decision decoders for convolutional codes
Marcone et al.[30]	FFNN	Comparable to Viterbi	Limited practicality for real-time applications	Comparable to Viterbi decoder
Wang and Wicker [31]	Neuralized Viterbi Decoder	Fast, fewer transistors, suitable for VLSI	Fixed weights, no training required	Equivalent to ideal Viterbi decoder
Annauth and Rughooputh [32]	FFNN	Fast, less complex	Slightly less performant than MAP decoder	Faster and less complex than MAP
Rajbhandari et al.[33]	FFNN for Convolutional Codes	Outperforms hard decision decoder	Falls short of soft decision decoder in BER	Improved BER across SNR range
Berber et al.[34]	RNN	Exploits temporal dependencies, unsupervised	Requires parameter reset for each message	Comparable to Viterbi decoder
Salcic et al.[35]	RNN Decoder on FPGA	Low complexity, real-time evaluation	Limited to FPGA resources	Effective BER performance under various conditions
Hyeji Kim et al.[7]	RNN-based Neural Decoder	Matches/exceeds conventional turbo decoders	Training challenges	High performance, flexible architecture
Jiang et al. (*DEEPTURBO*) [36]	Bi-GRU-based Turbo Decoder	Superior BER and BLER, no BCJR knowledge required	Requires reduced decoding iterations	Outperforms TURBO and NEURALBCJR
Sattiraju et al.[38]	GRU Turbo Decoder	Effective in low SNR conditions	Computationally intensive	Outperforms conventional turbo decoders
Balevi and Andrews [39]	Channel Autoencoder	Close performance to turbo codes	Performance may degrade with higher-order modulations	Near-turbo code performance
Jiang et al. (TurboAE)[40]	CNN Turbo Autoencoder	Comparable to traditional turbo codes	Focused on AWGN channels	Similar BER to traditional turbo codes
Chahine et al. [41]	Turbo Autoencoder with Trainable Interleaver	Superior in various practical channels	Complexity in training and optimization	Significant BER improvements in practical scenarios

Table 2.1: Summary of neural network-based decoding methods

Conclusion

This chapter focuses on introducing deep neural networks (DNNs), including architectures from Feed Forward Neural Networks (FFNN) to sequence-to-sequence LSTM attention models, these architectures mark a departure towards more adaptive and reliable decoding solutions.

Moreover, we explore deep learning methods for decoding convolutional and turbo codes, revealing a significant shift from traditional iterative turbo decoders like BCJR and SOVA.

Each neural network type is explored in detail, emphasizing its potential to improve error rates, computational efficiency, and adaptability compared to conventional methods.

The exploration covers various neural network architectures and their applications in decoding techniques, laying the groundwork for more effective error correction strategies in communications engineering.

This investigation leads to the introduction of new approaches of using other machine learning architectures on turbo decoding, specifically attention models, which will be explored in two ways in the next Chapters 3 and 4 .

Chapter 3

TurboAttention: Sequence to Sequence Bi-LSTM Attention Model for Turbo Decoding

Introduction

In this chapter, we introduce a new approach to turbo decoding using RNNs, specifically a sequence-to-sequence attention model based on Bi-LSTMs, which we call ***TurboAttention***. We explore the inspiration and motivation behind employing this technique, provide a detailed description of the model's architecture, and outline the methodology for training and testing. Finally, we present the results produced by this decoder.

TurboAttention represents a novel approach by leveraging turbo-inspired decoding models and Natural Language Processing models, particularly the sequence-to-sequence models based on attention mechanism 2.1.5.2. This approach draws an analogy between text translation and turbo decoding sequences. We investigate whether the model can identify patterns in turbo codes to correct errors in these sequences and outperform both standard decoders (SOVA) and state-of-the-art deep learning decoding models.

3.1 Background and motivation

Iterative turbo decoders (BCJR, SOVA) lack robustness and adaptivity in non-AWGN settings, with performance severely degraded by burst noise. Additionally, turbo codes suffer from an error floor at high SNRs, making them unsuitable for high-reliability applications such as secure communications. Techniques like better interleavers, and concatenating outer codes like BCH have been proposed to mitigate these issues but have had limited success. Consequently, traditional turbo designs struggle to consistently achieve high reliability, robustness, adaptivity, and low error floors.

Sequence-to-sequence models 2.1.5 are a class of neural networks designed to transform an input sequence into an output sequence. Initially introduced for natural language processing, these models consist of an encoder, which generates a context vector from the input sequence, and a decoder, which produces the output sequence based on this context vector. To enhance the model's ability to focus on important parts of the sequence, the attention mechanism was developed and has proven to be highly efficient. Applications of sequence-to-sequence models extend beyond natural language processing to other domains, such as wireless communications. Cao el al.[43] proposes a Long Short-Term Memory (LSTM) Attention Neural Network-based signal detection method for hybrid modulated Faster-Than-Nyquist (FTN) Optical Wireless Communications (OWC), significantly improving the accuracy of signal recovery in atmospheric turbulence channels. The proposed model outperforms traditional methods.

The inspiration of *TurboAttention* model came from *DEEPTURBO* [36] [36] and its technique to use the same structure of a iterative turbo decoder but based on learnable RNN model as Soft

Input SOft Output (SISO) block, plus the sequence to sequence attention models by Bahadanau et al. [27] and Luong et al. [28], where they introduced new effective mechanisms added to language translation models to solve some of the problems that the previous sequence models suffered from like the limitations in processing long sequences and improving the performance of translation.

In the next part, we explain in detail the architecture of *TurboAttention* model and how it functions by explaining its composing parts.

3.2 Architecture of *TurboAttention*

To investigate the impact of the attention mechanism on turbo decoding, we propose *TurboAttention* inspired by *DEEPTURBO* [36] model and sequence to sequence LSTM attention model. Each (Soft Input Soft Output) SISO block uses an attention model based on multilayer Bi-LSTMs as the building block. Keeping the extrinsic connection between each block as a shortcut for gradient. For this model, the number of iterations defines the number of *TruboAttention* SISO blocks, so the model doesn't share weights across each iteration to make each block deal with the prior information differently. Moreover, non-shared weights improve the training stability [36]. *TurboAttention* model passes more information from one stage to the next one, unlike iterative turbo decoders (SOVA) represent the posterior of each code bit by a single value log-likelihood (LLR). A single value for each code bit may not convey adequate information [36]. Drawing inspiration from ensemble methods used to address calibration issues [44], K bits are employed instead of just 1 bit for each code bit position. For instance, with a block length L, the SOVA posterior LLR has a shape of $(L, 1)$, whereas *TurboAttention* transmits a posterior with a shape of (L, K) to the next stage.

The expected advantage of using the attention mechanism for decoding lies in its ability to dynamically focus on different parts of the received sequence, assigning higher weights to more reliable parts and down-weighting erroneous parts using the iterative update of posterior information between each *TurboAttention* SISO block, similar to iterative turbo decoding, allows the model to improve its estimates over multiple iterations, thereby improving error localization and correction.

By maintaining a context vector that captures relevant information from the entire received sequence, the attention mechanism helps the decoder make better-informed decisions, contributing to robustness against noise and higher decoding accuracy, especially in the presence of burst errors. Despite introducing additional parameters for computing attention scores and more layers of Bi-LSTMs for the encoder and decoder, and increasing memory usage, the attention mechanism requires more computations per training iteration, resulting in longer training times per epoch compared to models without attention. However, these models tend to con-

verge faster to better solutions by effectively handling long-range dependencies and complex alignments, which is not the case for simple sequence models, leading to fewer epochs required to reach desired performance levels. Consequently, the overall impact is that attention models might take more total training time due to higher per-iteration costs, but they typically provide significantly better performance, making the trade-off worthwhile.

The configuration of the proposed *TurboAttention* decoder is depicted in Figure 3.1, which showcases a 1/3 rate decoder. Each decoding iteration employs two *TurboAttention* SISO decoders. The initial block utilizes the deinterleaved posterior p from the preceding block as prior information, along with the received systematic bits y_1 and parity check bits y_2 as inputs. Conversely, the second block uses the interleaved $\pi(q)$ as prior and the received interleaved systematic bits $\pi(y_1)$ and parity check bits y_3 as inputs. The *TurboAttention* block generates the posterior q, which is then passed to the subsequent block. In the final iteration, decoding is executed based on the estimated posterior, which is the input of the sigmoid function that estimates the decoded bits $\hat{u}$.

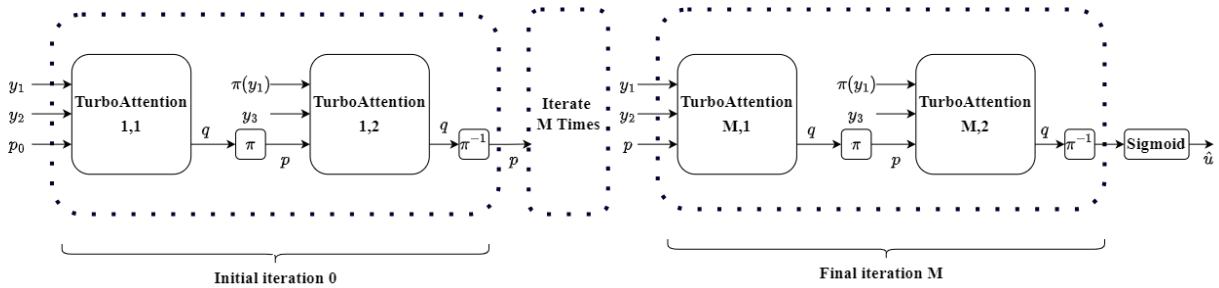


Figure 3.1: Diagram of *TurboAttention* Trubo Decoder

In the following sections, we will explain the main constituents of *TurboAttention*: the Encoder, Attention Layer, and Decoder.

3.2.1 The encoder

The encoder is the first stage of *TurboAttention*. It comprises stacked Bi-LSTM layers, with the number of layers adjustable to match the desired model complexity. The LSTM is chosen for its ability to handle long-term dependencies, provide sequence flexibility, learn context, resist noise, and integrate well with attention mechanisms. Additionally, the bidirectional LSTM captures context from both past and future states in the received sequence.

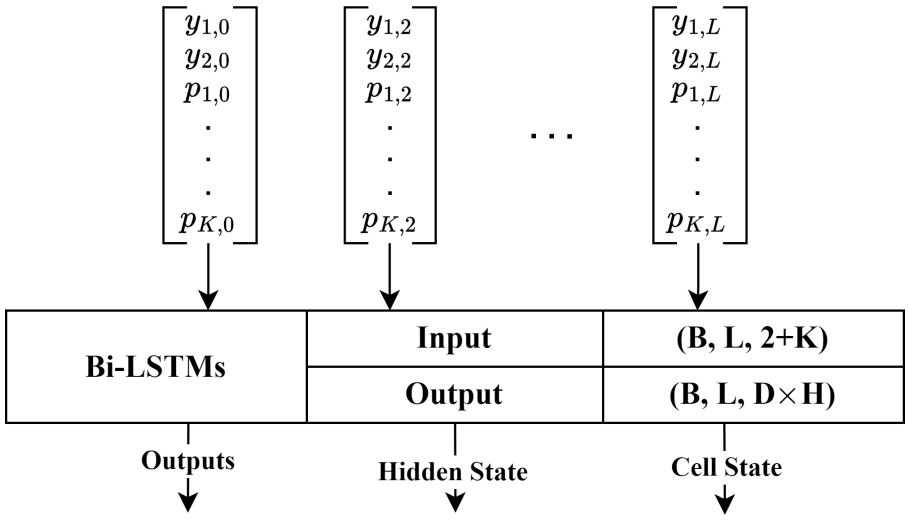
At each iteration, the received sequence $\mathbf{y}$ is demultiplexed, reshaped, and fed into the encoder of *TurboAttention* block. For decoding iteration m, the $\mathit{TurboAttention}_{1,\mathrm{m}}$ block receives at each time step l the systematic bit $\mathbf{y}_{(1,l)}$, the parity check bit $\mathbf{y}_{(2,l)}$, and the deinterleaved posterior $\pi^{-1}(\mathbf{q})$ as prior information from the previous block. The $\mathit{TurboAttention}_{2,\mathrm{m}}$ block receives the interleaved systematic bits $\pi(\mathbf{y}_{(1,l)})$, the parity check bits $\mathbf{y}_{(3,l)}$, and the interleaved posterior

$\pi(\mathbf{p})$ from the previous block as prior information.

The received sequence is structured as follows:

$$\mathbf{y} = (y_{1,1}, y_{2,1}, y_{3,1}, \ldots, y_{1,i}, y_{2,i}, y_{3,i}, \ldots) \tag{3.1}$$

Figure 3.2 illustrates the diagram of the encoder, the Bi-LSTM layers project an input sequence of size $[B, L, 2+K]$ to an output of size $[B, L, D \times H]$. Additionally, they output the final hidden states and cell states of the latest element in each batch of sequences. The shape of hidden states and cell states is $[B, 1, D \times H]$. These parameters are utilized in the subsequent attention mechanism stage of the model.



B : Batch Size
L : Sequence Length
K : Number of prioir features
D : 1 if Unidirectional LSTM, 2 if Bidirectional LSTM
H : Number of hidden units
y_1 : Systematic Bits (Or Interleaved Systematic Bits)
y_2 : Parity Check Bit
p : Prior Features

Figure 3.2: Encoder of *TurboAttention* Block

3.2.2 The attention mechanism

The attention mechanism introduces an additional computational stage after the encoder, allowing the model to focus on relevant parts of the input sequences, which may be erroneous and noisy, at each decoding step. This improves the accuracy and contextual relevance of the generated outputs and enhances the model's adaptability to varying block lengths.

The computation of the attention mechanism is illustrated in Figure 3.3, and can be described

in the following steps:

1. The attention mechanism iterates over each position in the input sequence of length L, extracting the query vector from the hidden state of the decoder at the previous decoding step $l-1$. For a batch of sequences, at a decoding step l, the shape of the query is $[B, 1, D \times H]$. At the initial position $l = 0$, it uses the latest encoder's hidden state as the query vector instead of the decoder's current hidden state, as the latter is not yet available.

2. The attention scores are calculated by batch matrix multiplication between the query and the transpose of the encoder's outputs, scaling those scores by the dimension of the query vector using the scaled dot product function:

$$\text{Score}(l) = \text{Query}(l) \times \text{Encoder Outputs}^T \tag{3.2}$$

3. The scores are then masked to exclude future positions by setting their values to a very low number. Masking the attention score vector enables efficient handling of variable-length sequences by focusing on actual and previous sequence content. This enhances generalization, improves training efficiency, synergizes with attention mechanisms for better relevance capture, and enhances model interpretability [45].

   The mask used is an upper triangular mask where the upper elements take a very low value and the lower elements, including the diagonal, take a zero value. Every row of the mask matrix corresponds to its index l to mask the attention scores given at the time step l:

$$\text{Masked Score}(l) = \text{Score}(l) + \text{Mask}(l, 0:L) \tag{3.3}$$

$$\text{Mask}_{(L,L)} = \begin{pmatrix} 0 & -\infty & -\infty & \cdots & -\infty \\ 0 & 0 & -\infty & \cdots & -\infty \\ 0 & 0 & 0 & \cdots & -\infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix} \tag{3.4}$$

4. The Softmax function is applied to these masked scores to convert them into a probability distribution, allowing the model to assign relative importance to different parts of the input sequence by producing attention weights α that sum to a value of one.

$$\text{Attention Weights}(l) = \textbf{Softmax}(\text{Masked Score}(l)) \tag{3.5}$$

5. The context vector is the result of the batch matrix multiplication between the attention

weights and the encoder outputs.

$$\text{Context}(l) = \text{Attention Weights} \times \text{Encoder Outputs}^T \tag{3.6}$$

6. Finally, the context vector is concatenated with the current decoder target to form the attention parameters, which serve as the input to the decoder. The context vector makes the decoder focus on the most relevant parts of the input sequence at each decoding step.

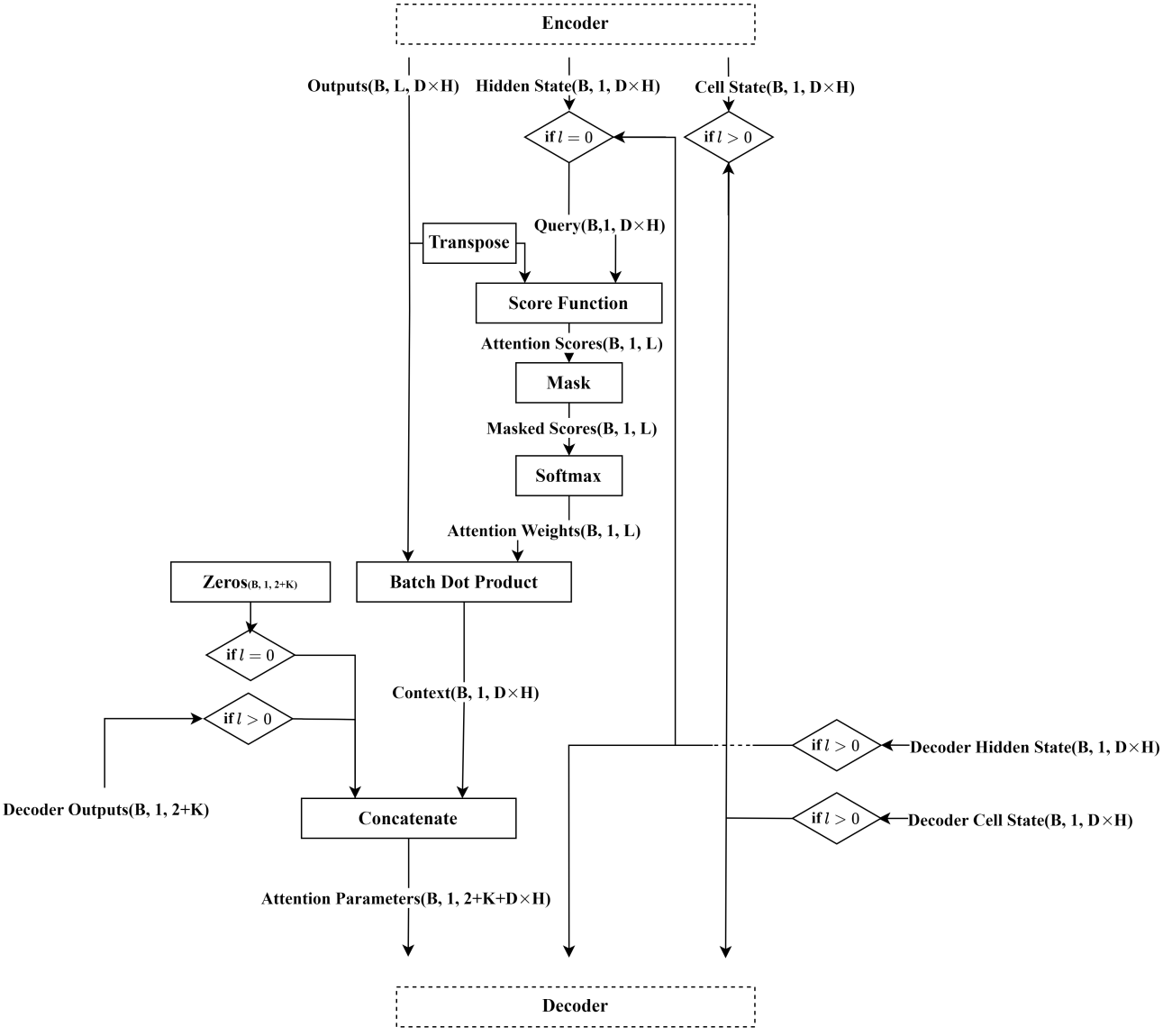


Figure 3.3: The Attention Layer of *TurboAttention* block

3.2.3 The decoder

The decoder is the final stage of *TurboAttention* block and shares an identical architecture with the encoder.

At time step l, the input to the decoder is the concatenation of the attention parameters with the target. Additionally, it inputs the hidden state and cell state from the previous decoding step. At step $l = 0$, it uses the latest hidden state and cell state from the encoder.

The output of the last Bi-LSTM layer in the decoder is passed to a Feed-Forward Neural Network (FFNN), which produces the decoder output for step l that has the shape $[B, 1, 2+K]$.

Using a greedy search approach, the targets of the decoder at step $l = 0$ are zero-initialized to indicate the absence of information about this initial state. For subsequent steps, the decoder uses the previous output from the FFNN layer.

Figure 3.4 shows the diagram of the decoder. The input size is $[B, 1, 2 + K + D \times H]$, which represents the size of the concatenated decoder target and attention parameters.

The output of the FFNN corresponds to one bit in the sequence, so at each decoding step, the outputs of the FFNN are concatenated to get a shape $[B, L, K + 2]$ for the whole sequences in a batch, this output is passed through a *log-softmax* function, it improves numerical stability and computational efficiency when calculating softmax probabilities, especially when dealing with large and low numbers.

Finaly, if the decoder corresponds to the latest iteration block $\textit{TurboAttention}_{\mathrm{M},2}$, the output of the *log-softmax* function passes through a FFWD linear layer with an output shape $[B, 1, L]$ which represents the estimated information sequence $\hat{u}$. If the decoder corresponds to the previous iterations, the prior features of the previous iteration are subtracted from the output of the *log-softmax* function to give the posterior features that pass to the next stages.

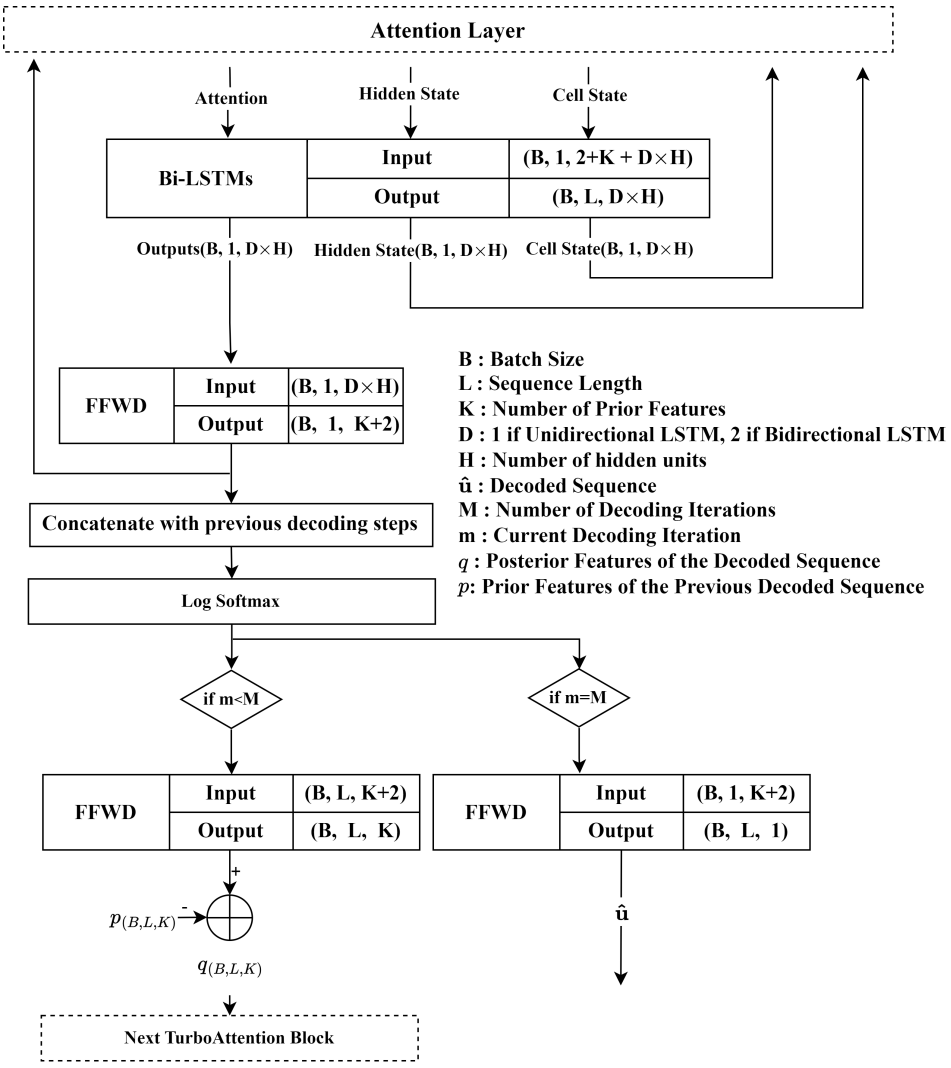


Figure 3.4: The Decoder of *TurboAttention* Block

3.3 Methodology

The methodology sums up the process of training and evaluating the model, including details about the data we used, configurations of models, and some metrics

3.3.1 Data preparation

The data is randomly generated binary sequences using MATLAB, the block length can be defined according to the needed application. The block length chosen to train and test our model is 100, 98 bits are information bits and the last 2 bits are tail bits.

The turbo encoder $(7, 5, 7)$ is used for in this work, with code rate $R = \frac{1}{3}$. The quadratic interleaving 1.2.3 is also chosen for this encoder. The interleaver configuration for sequence length 100 is given as follows:

$$\pi(x) = (34 \cdot x + 63 \cdot x^2) \text{ modulo} N \tag{3.7}$$

The resulting encoded sequence is 302 bits, thus 294 bits are the encoding of the original information sequence and 8 bits represent two tail bits of systematic bits, parity check bits 1, parity check bits 2, and interleaved systematic bits.

To create a training set that mimics transmitted sequences in a wireless communication system, we simulate the communication channel, with its consecutive stages.

First is the modulation, where the sequences are modulated using Binary Phase Shift Keying (BPSK) to give polarized values. Next, the modulated sequences pass through an Additive White Gaussian Noise (AWGN) channel with a noise level controlled by the Signal-to-Noise Ratio (SNR).

The training set consists of 10^5 noisy messages. Of this set, 90% have an SNR range between -1.5 dB and 4 dB, with noise levels added randomly to these sequences. The remaining 10% have an SNR fixed at -1.5 dB to provide additional data for very low SNR levels, helping the model adapt better to worst-case noise scenarios.

The validation set contains 2×10^4 encoded sequences, with an SNR range between -1.5 dB and 4 dB randomly distributed among the sequences in this set.

3.3.2 Training

3.3.2.1 Loss function and optimizer

In the final decoding iteration, the output is passed through a sigmoid function, which provides the probability of each decoded bit. If this probability is greater than 0.5, the bit is decoded as 1; otherwise, it is decoded as 0. This approach transforms the sequence decoding problem into a series of binary classification problems, making the Binary Cross Entropy (BCE) loss function the most suitable choice.

The BCE loss function compares the estimated information sequence $\hat{\mathbf{u}}$ with the ground truth information sequence $\mathbf{u}$, and is calculated as follows:

$$\text{BCE}(\mathbf{u}, \hat{\mathbf{u}}) = -\frac{1}{L} \sum_{i=1}^{L} [u_l \log(\hat{u}_i) + (1 - u_l) \log(1 - \hat{u}_l)] \tag{3.8}$$

where L is the number of bits in the sequence, u_i is the i-th bit of the ground truth sequence, and $\hat{u}_i$ is the i-th bit of the estimated sequence. The optimizer chosen to train this model is ADAM optimizer 3 is advantageous due to its adaptive learning rates, efficient computation,

and fast convergence, making it effective easy to use, and more reliable in training deep neural networks.

3.3.2.2 Hyperparameters of models

The trained *TurboAttention* model comprises identical encoder and decoder components, each with two Bi-LSTM layers containing 100 hidden units per LSTM. The number of features K is set to 5, which was determined to be optimal for training stability and performance, and the model undergoes 6 iterations.

This configuration was selected after model tuning to balance complexity, in terms of parameter count and time complexity, with performance. The chosen model will be compared to the baseline model.

The learning rate is initially set to 0.001 for the first epoch and is reduced when the validation loss saturates. The batch size is set to 100. Although a larger batch size could be used. However, a small batch size like this helps to update the loss more quickly, leading to faster convergence. Additionally, the noise introduced by a small batch size acts as a regularization method, helping to prevent overfitting.

The model trained on 16 epochs only when the loss saturates, knowing that this small number of epochs is due to limitations on computational resources, which makes trying multiple models with different configurations and more regularization are not considered that much.

Table 3.1 summarizes the hyperparameters and the configuration of the model:

Model Configurations	
***TurboAttention* Block**	2-layer Bi-LSTM with 100 units
Learning rate	0.001, decay when saturated loss
Num epoch	16
Block length	100
Batch per epoch	1000
Optimizer	Adam
Loss	BCE
Train SNR	range -1.5 dB to 4 dB
Batch size	100
Posterior feature size K	5
Decode iterations	6
Number of Trainable Parameters	9.828532 M

Table 3.1: Model Configurations of *TurboAttention* Block

We train the baseline model *DEEPTURBO* [36] using the same dataset and under comparable conditions to facilitate a performance comparison with our proposed approach. The architecture of this model comprises two Bidirectional Gated Recurrent Unit (Bi-GRU) layers, each with 100 hidden units. The number of prior features K is set to 5, and the model includes 6 decoding iterations. This configuration mirrors the model proposed in the referenced paper. The training was conducted over 25 epochs, at which point the validation loss ceased to improve.

Table 3.2 summarizes the configuration of the baseline model and the number of trainable parameters.

***DEEPTURBO* [36] Configurations**	
***DEEPTURBO* Block**	2-layer Bi-GRU with 100 units
Learning rate	0.001, decay when flattened loss
Num epoch	25
Block length	100
Batch per epoch	1000
Optimizer	Adam
Loss	BCE
Train SNR	Range :-1.5 dB to 4 dB, step 0.2 dB
Batch size	100
Posterior feature size K	5
Decode iterations	6
Number of Trainable Parameters	2.970456 M

Table 3.2: Model Configurations of *DEEPTURBO* [36] Block

3.3.3 Evaluation metrics

Two main metrics are used to evaluate the decoding performance, the *bit error rate (BER)* introduced in 1.3.2.5.1 and the *block error rate (BLER)* introduced in 1.3.2.5.2, those metrics can give a better view of the decoding performance on different levels of noise that affected the received sequences.

3.4 Experimental setup

Machine learning models demand a specific environment, whether it is hardware or software, plus previous implementations and source codes to inspire from. This all is explained on this part.

3.4.1 Hardware

For our experiments, we used a Kaggle platform that provides a cloud computational environment. It environment contains **73.1 GB** disk space, with **Intel Xeon 2.20 GHz CPU**. The GPU used is **Nvidia Tesla P100** with **VRAM of 16 GB**, paired with **32 GB RAM**.

3.4.2 Software

For Data preparation, we use MATLAB to generate noisy sequences, decode those sequences using SOVA decoder, and retrieve the BER values that will be compared to the performance of the model. The deep learning framework **PyTorch 2.4** is used, with **Python 3.10** and **CUDA 12.1** for parallel computing using the GPU.

3.4.3 Implementation

The implementation of the selected model was based on the *DEEPTURBO* [36] code provided by the authors online.[1] This code served as the foundation upon which we added an attention layer, thereby structuring each block to include an encoder, decoder, and attention mechanism.

The implementation of the attention mechanism was inspired by Bahdanau Attention [27] used in an online source code[2] of neural machine translation (NMT) model. Several modifications were made to adapt the mechanism for turbo decoding. Specifically, the word embeddings used in the original NMT implementation were removed, allowing the inputs to be directly fed into the model.

3.5 Results and analysis

This part analyses the results related to our method, like the training process, comparison of tests with the baseline model, and some specific experiments.

3.5.1 Training results

The training was done for our proposed model *TurboAttention* and the baseline model *DEEPTUEBO* in the same conditions using the same train, validation, and test sets to make the comparison more fair.

[1] DeepTurbo code available at: https://github.com/yihanjiang/turboae
[2] NMT code available at: https://github.com/mhauskn/pytorch_attention/tree/main

3.5.1.1 *TurboAttention* model :

The training of *TurboAttention* was followed by monitoring the training loss for each epoch on the whole train set, plus monitoring the loss value and the average BER value on the validation set after each epoch, thus we can observe better the progress of the model and investigate if any improvement or actions are needed for the model, like adjusting the learning rate, regularization, or even the data.

Training time for each epoch is 52 minutes, and validation time is 2 minutes.

Figure 3.5 , shows the results obtained after training 16 epochs, where the loss values for both train and validation sets decreased to low values after a few epochs only to keep decreasing slowly for the rest of the training steps to reach **0.067** for the validation loss, this latter started to increase at the latest few epochs which led to early stop the model to avoid overfitting. More regularization could be added like augmenting data or introducing dropout, but that will demand retraining the model since keeping that model and fine tuning is not very stable.

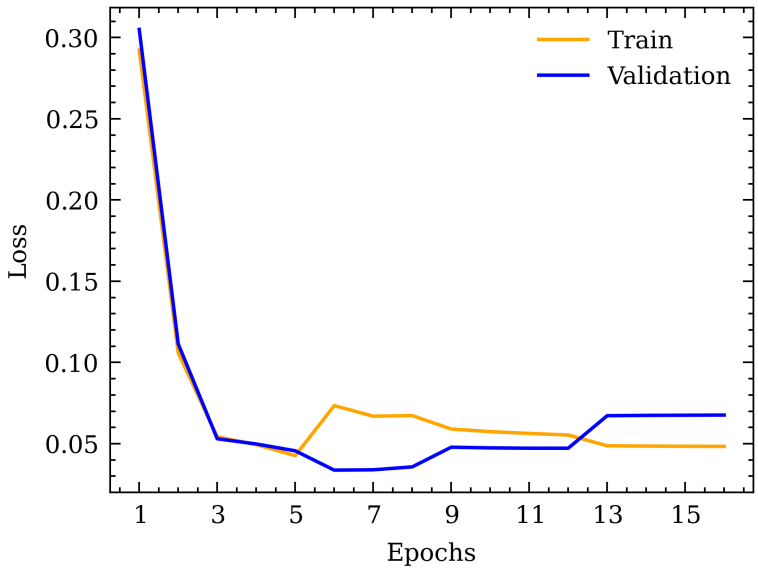


Figure 3.5: *TurboAttention* : Train and Validation Loss

Figure 3.6 depicts the progress of the average of the BER on the validation set, those values are not significant to evaluate the performance of the model since the average is on samples that have different SNR levels, but it is still important to monitor the training.

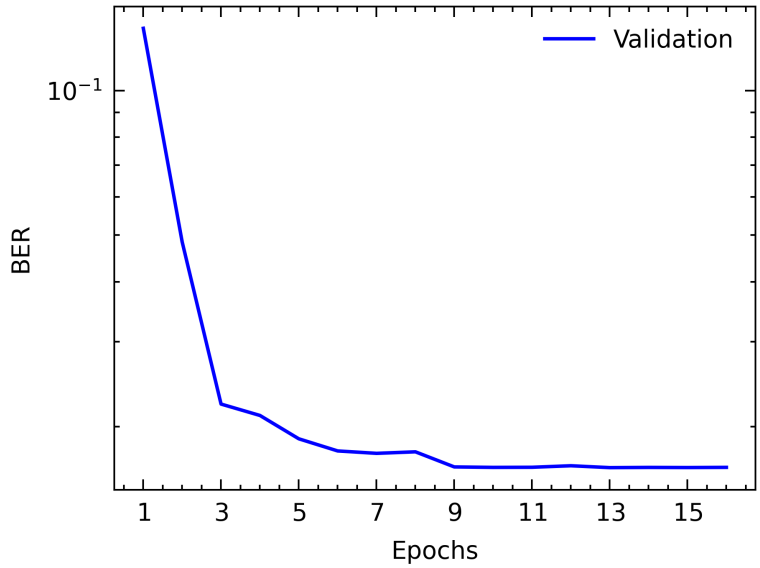


Figure 3.6: *TurboAttention* : Validation Average BER

We can see that the loss or the BER of the train set is most of the time higher than the validation set because of the high percentage of samples of SNR value -1.5, which their sequences have a higher number of errors.

3.5.1.2 Baseline model: *DEEPTURBO* [36]

We train the model on 25 epochs, every epoch takes time of 1 minute and 35 seconds and validation time of 3 seconds.

Figure 3.7 shows the training results, we can see that the model's validation loss started to overfit after the 20th epoch, also reached a validation loss **0.0319**.

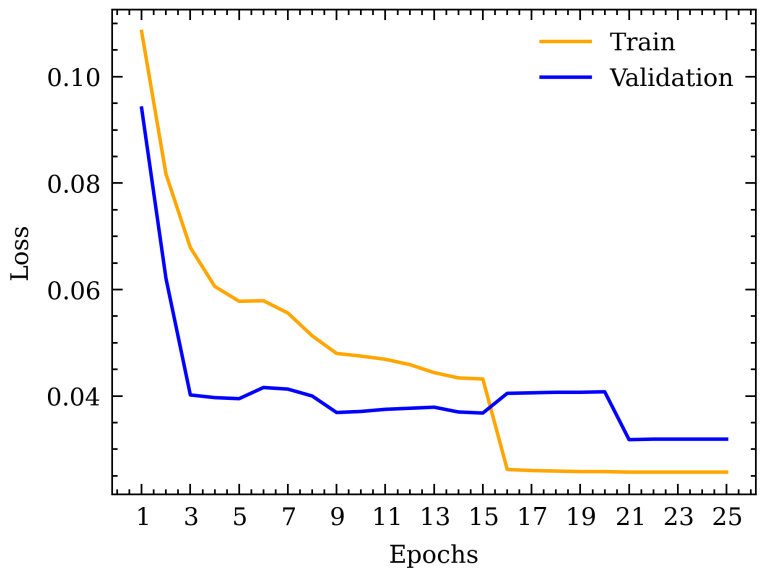


Figure 3.7: *DEEPTURBO* [36] : Train and Validation Loss

Figure 3.8 shows the average BER for the baseline model training, it has the same behavior as the loss, where the BER of the train set keeps decreasing after epoch 20, but the validation BER starts to increase.

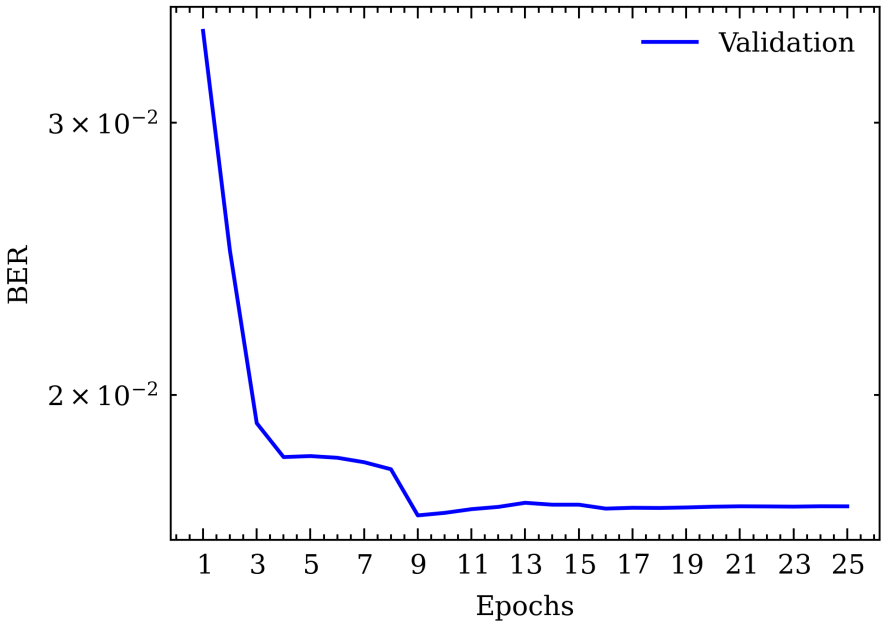


Figure 3.8: *DEEPTURBO* [36] : Validation Average BER

Figure 3.9 illustrates a detailed comparison of the average BER progression across validation epochs for both the attention model and the baseline model. Initially, at epoch one, the attention model exhibits a relatively high BER. However, it demonstrates a significant and rapid decrease in BER over the subsequent epochs. This rapid decline highlights the efficiency and effectiveness of the attention model in achieving superior performance within a relatively small number of iterations.

Notably, the attention model consistently outperforms the baseline model, even from the early stages of training. The plot shows that, despite the initial high BER, the attention model converges faster towards a lower BER. By requiring fewer epochs to reach a similar or better performance level compared to the baseline, the attention model proves its robustness and capability in optimizing the error rate efficiently. This observation underscores the advantage of incorporating attention mechanisms in the model, leading to improved performance with reduced training time.

This expanded version provides a clearer and more detailed explanation of the figure, emphasizing the efficiency and superiority of the attention model in reducing BER over fewer epochs compared to the baseline model.

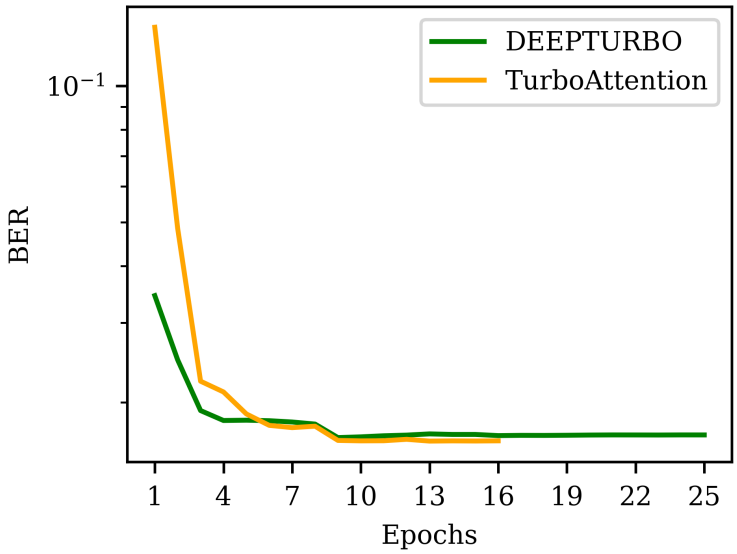


Figure 3.9: *TurboAttention*, *DEEPTURBO* [36] : Average Validation BER

3.5.2 Test results

The test set comprises 1.5×10^5 encoded sequences, each with an information sequence block length of 100 bits. The SNR for this set ranges from -2 dB to 5 dB, with increments of 0.5 dB between each SNR value. Consequently, each SNR level is represented by 10^4 samples.

Figures 3.10 and 3.11 show the performance metrics of *TurboAttention*, *DEEPTURBO* [36] , and Soft Output Viterbi Algorithm (SOVA) turbo decoder on the test set, evaluated in terms of Bit Error Rate (BER) and Block Error Rate (BLER).

In terms of BER, *TurboAttention* consistently outperforms the iterative SOVA decoder across the entire SNR range. It also closely matches the performance of the *DEEPTURBO* [36] baseline model or slightly better numerically. Notably, both *TurboAttention* and *DEEPTURBO* [36] achieve zero errors for all test samples at SNR levels above 3 dB, indicating their robustness in high SNR conditions.

The BLER results, as shown in Figure 3.11, mirror the observations in the BER performance. *TurboAttention* demonstrates a similar advantage over SOVA and comparable performance to *DEEPTURBO* [36] . This consistent performance across both BER and BLER metrics underscores the efficacy of the *TurboAttention* model in decoding efficiency and error correction, even under varying SNR conditions.

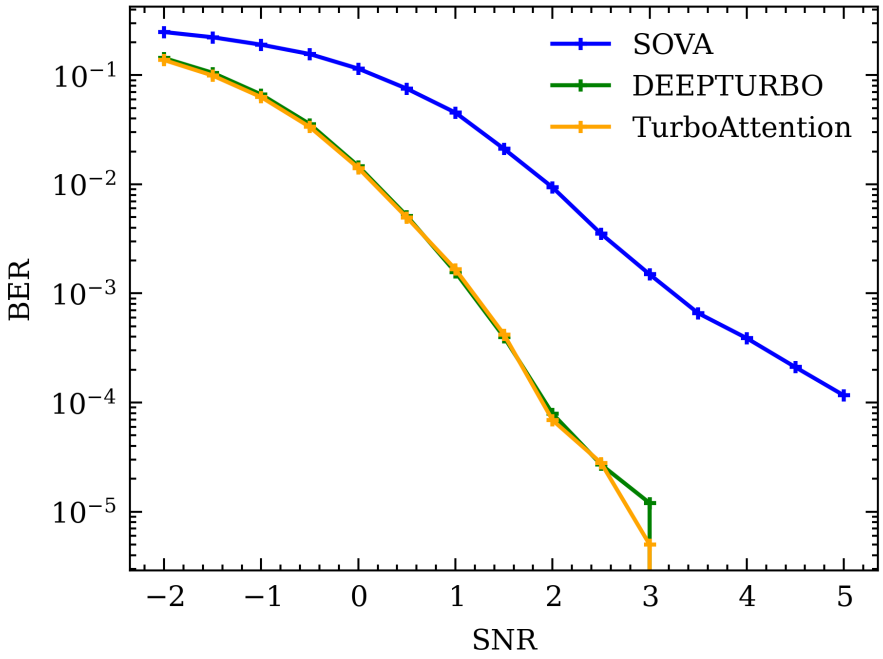


Figure 3.10: *TurboAttention*, *DEEPTURBO* [36] , SOVA : BER performance on different SNRs

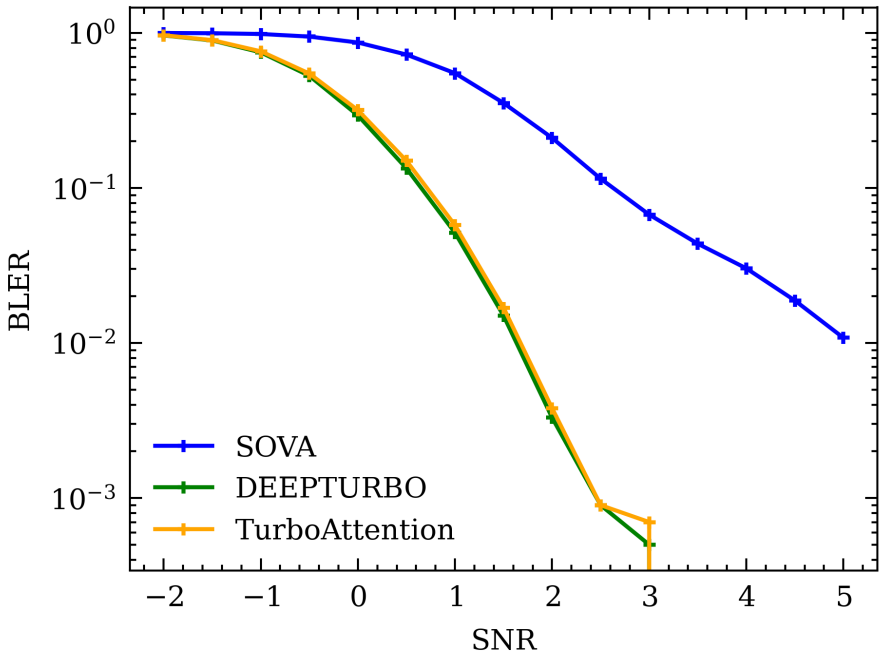


Figure 3.11: *TurboAttention*, *DEEPTURBO* [36] , SOVA : BLER performance on different SNRs

In the next part, we explore more details about specific cases on the proposed model, mentioning tests on a different block length, and hardware inference.

3.6 Case studies

We will study the capacity of *TurboAttention* to generalize the predictions to different block lengths and then analyze the hardware implementation in terms of execution time.

3.6.1 Genralization on different block lengths

Standard turbo decoders, such as the Soft Output Viterbi Algorithm (SOVA), exhibit asymptotic performance improvements with increasing block length, meaning that the longer the received encoded message, the lower the error rate in decoding. This behavior, however, is not observed in the baseline model, where performance deteriorates with increasing sequence length when the model is trained on a fixed sequence length.

To investigate the effect of block length on performance, we compare *TurboAttention* and *DEEPTURBO* [36] using a different test set comprised of encoded sequences with an information sequence length of 258 bits. The SNR range for this test set spans from -1.5 dB to 4 dB, with steps of 0.5 dB between each value. For each SNR value, there are 10^4 samples, resulting in a total of 1.2×10^5 samples for the entire test set.

Figures 3.13 and 3.12 present the results of these tests. The performance of *TurboAttention* and *DEEPTURBO* [36] is analyzed in terms of both Bit Error Rate (BER) and Block Error Rate (BLER). These figures highlight the incapability of *TurboAttention* model in handling longer sequence lengths, which means that training it on constant block length (100 bits) is not helpful for the model to generalize to the variety of block lengths. *DEEPTURBO* is better in generalization, while SOVA can even outperform *TurboAttention* on higher SNRs.

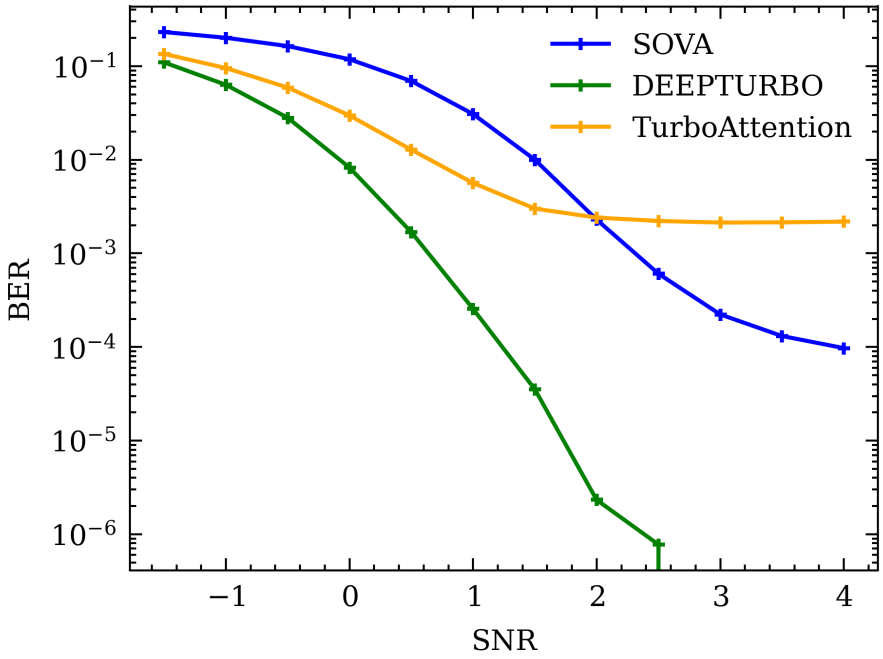


Figure 3.12: (Block Length = 258) *TurboAttention*, *DEEPTURBO* [36] , SOVA : BER performance on different SNRs

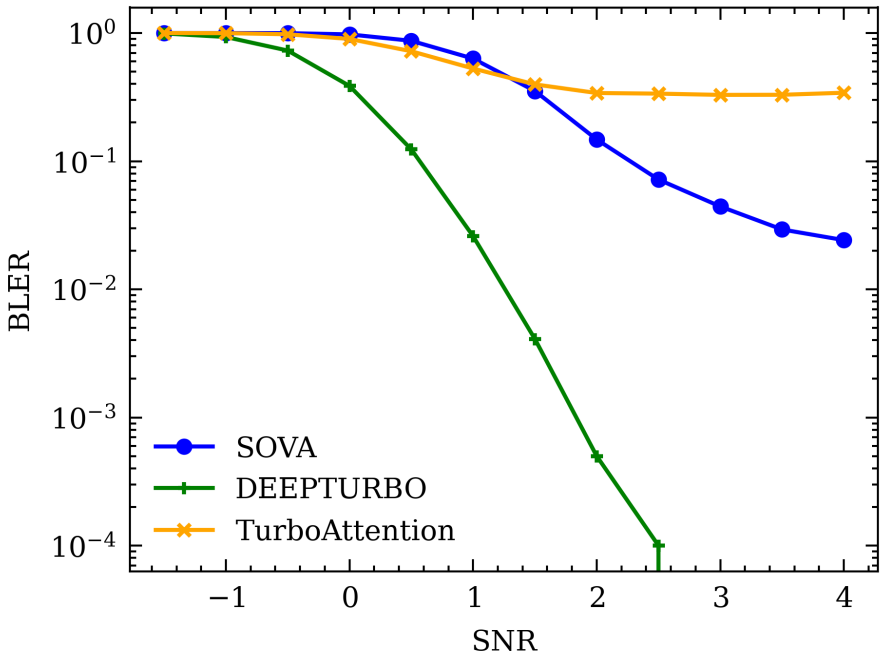


Figure 3.13: (Block Length = 258) *TurboAttention*, *DEEPTURBO* [36] , SOVA : BLER performance on different SNRs

3.6.2 Inference execution time on edge hardware

In this study, we performed inference tests of our deep learning model on the NVIDIA *Jetson Nano*, a compact AI computer designed for edge computing applications. The Jetson Nano is equipped with a quad-core ARM Cortex-A57 CPU, a 128-core NVIDIA Maxwell GPU, 4 GB of

LPDDR4 RAM, and supports various AI frameworks.*Jetson Nano* is compatible with **Pytorch 1.10** and **CUDA 10.2**.

The hardware and software configuration used for the tests included the following settings:

- CPU: Quad-core ARM Cortex-A57
- GPU: 128-core NVIDIA Maxwell
- Integrated GPU : 1 GB
- RAM: 4 GB LPDDR4
- Operating System: Ubuntu 18.04
- Python 3.6
- Pytroch 1.10
- CUDA 10.2

We evaluate the performance of the trained models by running inference with a batch size of 200 and measuring the batch execution time, sequence execution time, GPU usage, and RAM usage. The results are summarized in Table 3.3. Using CUDA environment, we can parallelize the process during inference, but we should mention that the models use only the integrated GPU of 1 GB of memory.

Model	**Batch Size**	**Batch Execution Time (s)**	**Sequence Execution Time (ms)**	**GPU Usage (%)**	**RAM Usage (MB)**
TurboAttention	200	10.5	52	85.3	314
DEEPTURBO [36]	200	1.45	7	48.3	483

Table 3.3: Inference Performance on *Jetson Nano*.

Parallel computing in inference enables models to simultaneously process multiple sequences, significantly enhancing the throughput of the decoder. However, methods like *Turboattention* suffer from higher time complexity, primarily due to the attention bottleneck. In contrast, *DEEPTURBO* [36] demonstrates approximately seven times faster performance. Figure 3.14 illustrates *Jetson Stats*, an interface displaying resource utilization on *Jetson Nano*. It tracks real-time metrics such as GPU and CPU usage, RAM consumption, power consumption, and component temperatures during the execution of the model.

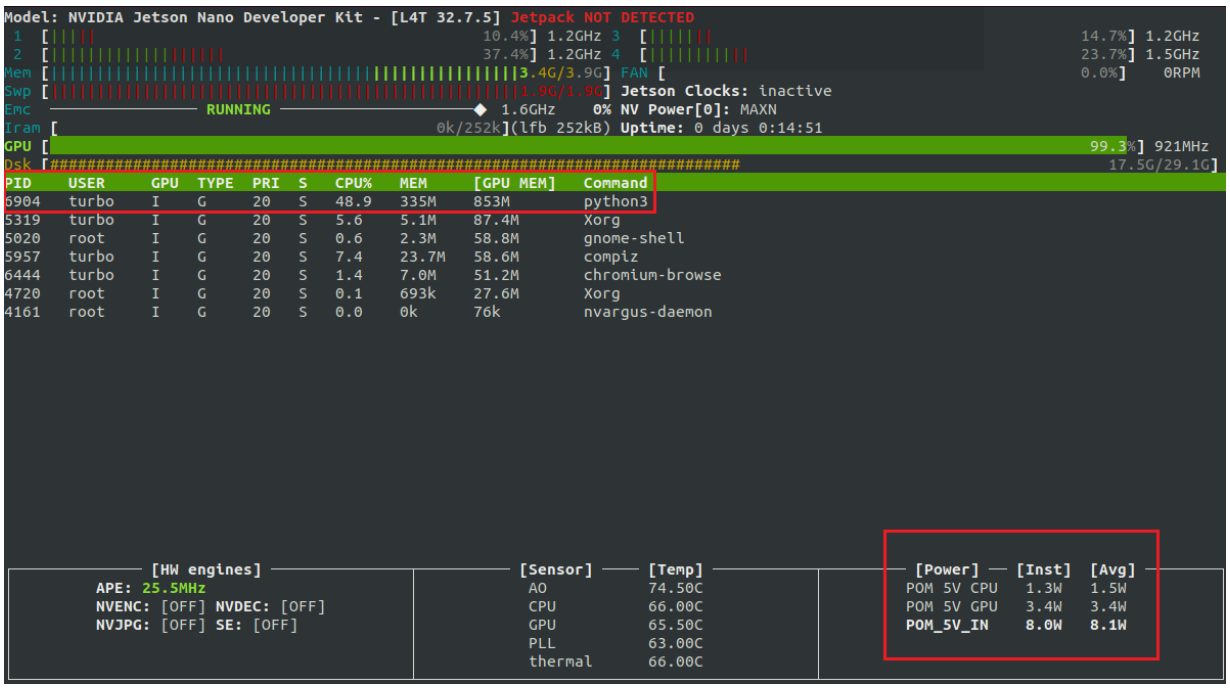


Figure 3.14: *Jetson Stats*: Usage of resources usage on Jetson Nano

3.7 Discussion

The performance of *TurboAttention* closely matches or slightly exceeds that of the baseline model when trained on constant block lengths. However, extending testing to longer block lengths did not improve the model's ability to generalize, suggesting potential overfitting to specific block lengths. In contrast, typical sequence-to-sequence attention models used in language processing tasks are trained across various sequence lengths to enhance generalization, a capability lacking in our model.

The attention mechanism in our model is trained to establish relationships between input and output sequences, which may explain its difficulty in generalizing to longer sequences and the occurrence of gradient vanishing in LSTM layers. Furthermore, attention models heavily depend on large datasets to maximize their performance and outperform traditional sequence models. Interestingly, the Bi-GRU baseline model did not experience a significant performance drop compared to *TurboAttention*, which is back to its lower complexity that didn't make the model overfit on a single block length. Instead, *TurboAttention* is more complex and needs more regularization and data variety to explore the full potential of seq2seq models based on attention.

Moreover, the attention model introduces additional complexity to the decoder, particularly in terms of model parameters due to both encoder and decoder components, and in terms of time complexity due to the attention bottleneck involving matrix multiplication and step-by-step decoding.

Regarding inference time on hardware, experiments demonstrate that models like *TurboAttention* and *DEEPTURBO* [36] can be effectively implemented on edge devices and used in real-time applications across various domains.

Conclusion

The attention model *TurboAttention*, originally designed for turbo decoding, showcased the potential of applying attention mechanisms—typically used in natural language processing—to the decoding of erroneous binary sequences. By leveraging the encoding scheme and updating prior information at each epoch, the model effectively identified temporal dependencies within sequences, enhancing its ability to decode messages reliably.

In comparison to the iterative turbo decoder, our proposed approach achieved performance on par with the baseline model, with notable degradation observed only for longer block lengths in *TurboAttention*. To further explore the potential of attention models in this context, additional avenues such as increasing dataset size, applying regularization techniques, and configuring more complex model architectures warrant detailed investigation.

Demonstrating the feasibility of deploying sequence models for turbo decoding on edge hardware devices is a significant achievement. Future research could expand beyond these experiments to explore implementations on FPGA, Raspberry Pi, and conduct real-world tests using radio interfaces to evaluate the real-time applicability of these models in communication channels.

While both models could potentially benefit from further parameter tuning and extended training, computational resource limitations currently constrain our ability to achieve optimal results, particularly notable for *TurboAttention*, which requires longer epochs for training.

In the next chapter, another deep learning approach for tutro decoding is introduced, it also relies on attention mechanism but in another way and another view.

Chapter 4

TurboTransformer: Transformer Model for Turbo Decoding

Introduction

In this chapter, we introduce a new approach to turbo decoding using Transformer models, which we call ***TurboTransformer***.

TurboTransformer represents a novel approach by leveraging the gathered knowledge on turbo codes and advancements in Natural Language Processing. This approach draws an analogy between text generation and turbo decoding sequences. We investigate whether the model can identify patterns in turbo codes to correct errors in these sequences and outperform the standard iterative decoder based on SOVA.

The Transformer model, renowned for its success in NLP tasks, relies entirely on self-attention and cross-attention mechanisms that allow it to process sequences in parallel, rather than sequentially as in RNNs or convolution. This parallel processing capability, combined with its ability to capture long-range dependencies in sequences, makes it a promising candidate for turbo decoding. In *TurboTransformer*, we adapt this architecture to decode turbo codes by training the model to recognize and correct errors in the received sequences.

In the following sections, we will provide a comprehensive overview of the *TurboTransformer* architecture, detailing each component and its role in the decoding process. We will also describe the training and testing procedures used to evaluate the model's performance. Finally, we will compare the results of *TurboTransformer* with traditional decoder and the decoder proposed in the previous chapter 3 to highlight its effectiveness and potential for future applications in error correction coding.

4.1 Background and motivation

Channel decoding succeeds due to its ability to detect and correct errors in received sequences. While knowing the encoding scheme is essential, ensuring reliable communication through highly corrupted channels is insufficient. Deep learning approaches enable models to learn how to transform corrupted encoded sequences into the desired information sequences. In this context, a transformer architecture with advanced attention mechanisms can learn to focus on the corrupted positions within a sequence.

The decoding scheme used by transformers, where tokens are generated incrementally based on previously generated outputs, mirrors the dependency introduced in convolutional codes. This concept of focusing attention on errors and maintaining a gradual dependency inspired the use of transformer architecture for turbo decoding. Notably, the transformer architecture has not yet been explored in the context of turbo decoding. This presents a significant challenge, as there is no baseline or starting point in the existing literature to guide this research.

The only notable use of transformers in channel decoding, introduced in [46], focused on block codes and other encoding techniques. However, this approach was not adapted to turbo codes, as *TurboAttention* does. This pioneering approach aims to leverage the strengths of transformer models in handling sequence dependencies and attention mechanisms to enhance error correction in turbo decoding.

4.2 TurboTransformer architecture

In this section, we present the detailed architecture of the proposed model. The description is provided block by block, outlining both the foundational components derived from the transformer architecture introduced in the paper "Attention is All You Need" [47], and the unique adaptations specific to *TurboTransformer*. These adaptations represent our efforts to tailor the architecture for our specific application. Figure 4.1 illustrates the detailed architecture of the model. We will proceed with a block-by-block explanation.

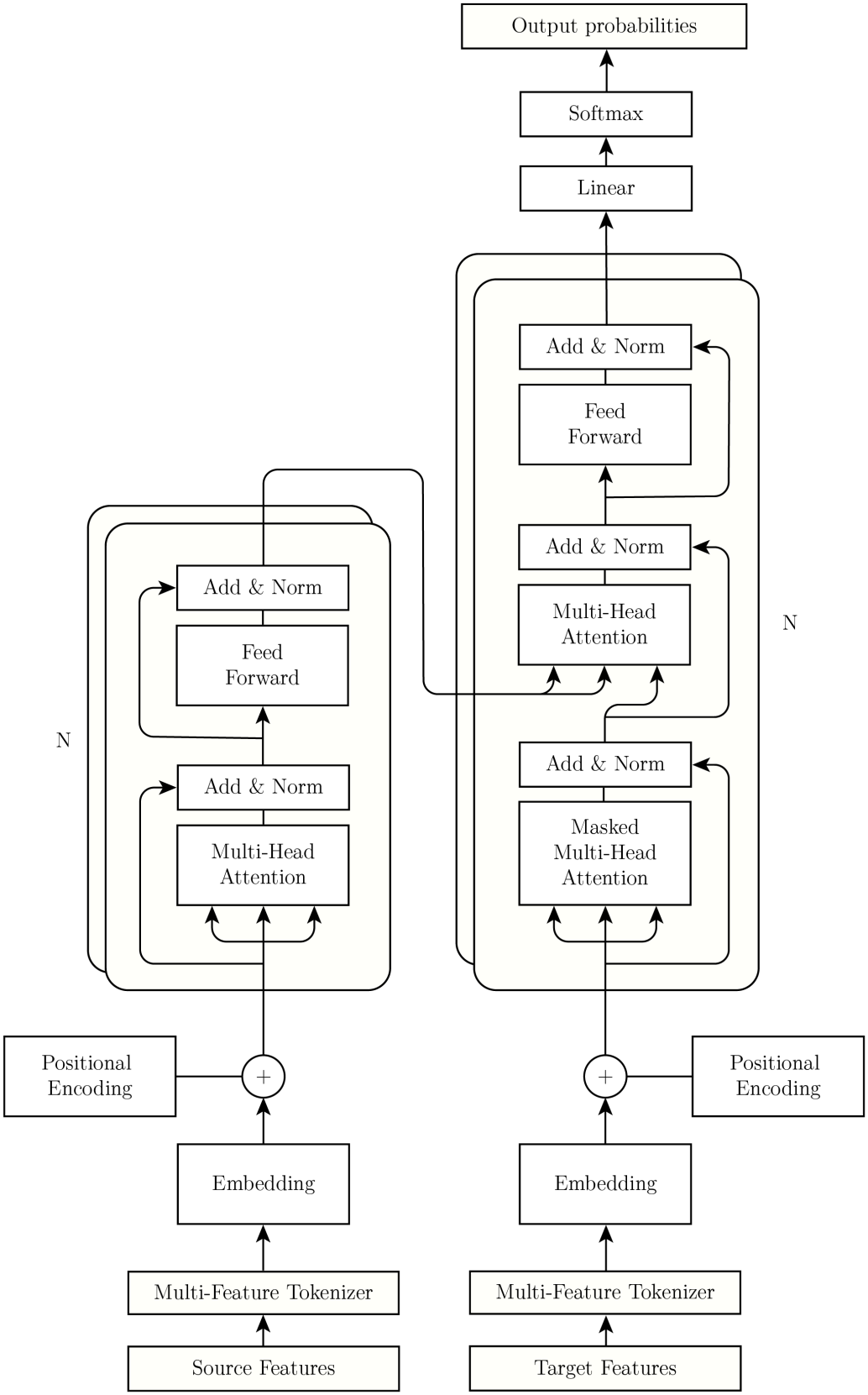


Figure 4.1: TurboTransformer detailed architecture.

4.2.1 Input features

The input features constitute the initial block of the model, gathering the necessary data to be fed into the system. In the context of turbo decoding, a set of key features is employed to capture the complexities of the code sequences. For the turbo code configuration described previously in 1.3.2.4.1, the extracted features are as follows:

- **Systematic bits:** This is the most important feature, representing the original information sequence.
- **Interleaved systematic bits:** The interleaved version of the information sequence, included to help the model understand the interleaving scheme and the relationships and dependencies within the sequence.
- **Parity 1 bits:** The parity bits sequence produced by the inner RSC encoder, which provides information on the encoding of the systematic sequence.
- **Parity 2 bits:** The parity bits sequence produced by the outer RSC encoder, which provides information on the encoding of the interleaved information sequence.
- **Convolutional states 1:** These states are related to the encoding/decoding of the coded sequence from the inner RSC encoder. Including this feature reinforces the relationship between the systematic bits and parity 1 bits, enhancing the model's understanding of the convolutional encoding scheme.
- **Convolutional states 2:** These states are related to the encoding/decoding of the coded sequence from the outer RSC encoder. Including this feature reinforces the relationship between the interleaved systematic bits and parity 2 bits, further enhancing the model's understanding of the convolutional encoding scheme.

After further preparation, these features are designated for either the encoder block or the decoder block. This distinction allows us to categorize the features into two sets: source features, which are intended for the encoder block, and target features, which are intended for the decoder block.

4.2.1.1 Target features

The target features for the decoder block are the features that we want our model to learn and replicate. As the name implies, the decoder aims to match these target features during its learning process. This concept of target features is crucial in the training process, which will be discussed later. Essentially, the target features are the correct turbo code features that

have not been affected by channel impairments. These include the original information bits, the correctly encoded parity bits, and the convolutional states from the encoding process of the two RSC encoders.

4.2.1.2 Source features

The source features for the encoder block are the features that we want our model to correct. The encoder aims to analyze these features and learn the tendencies of error occurrences. Essentially, the source features are the corrupted turbo code features received from a noisy channel transmission, including the corrupted received information bits and parity bits. Regarding the convolutional states, in a reception situation, no knowledge is communicated about the encoding states of the sequence, nor are decoding states provided. Therefore, we face a lack of state features. To address this, several methods can be employed, as listed below:

1. Set all source convolutional states to a fixed state, preferably the all-zeros state. This approach conveys no information to the encoder block about these features. The decoder will have to learn to infer the corresponding states from scratch.

2. Set source convolutional states to random sequences of states. This approach introduces randomness to the input data. Neural networks are known for their capacity to model well on random data following a fixed distribution, such as the Gaussian distribution.

3. Use a pretrained neural network model that estimates the convolutional states of a sequence. This approach adds a layer of complexity to the model but enables the model to begin with somewhat less erroneous feature values.

All three approaches were thoroughly tested to determine the best solution for the lack of source convolutional state values.

Approach (2) was the first to be eliminated because the model could not comprehend the relationships between the received bits and the random states. This approach produced an error rate of 88.3% on the generated states.

Approach (3) involved training a model to input the received encoded sequence and output an estimation of the encoding states. The model, termed the "States Estimator," used an LSTM sequential network to analyze the received bits, passing the values of its hidden states to a feedforward network (FFN) with two hidden layers, ultimately outputting the estimated states after a softmax activation. The training process used the correct encoding states as labels. This method reduced the error rate on the convolutional states to 31.6%. Despite this improvement, the estimations were not consistent enough for the TurboTransformer to extract meaningful dependencies from the sequence.

The selected approach was Approach (1). This technique showed the most consistency. Its impact was particularly noticeable in the initial stages of training, where the model converged significantly faster compared to the other techniques. The all-zero states initialization allowed the model to build an estimation of the features from values that had not been corrupted or altered. This straightforward approach performed best overall and was adopted in subsequent work.

4.2.2 Multi-feature tokenizer

Tokenization, in the realm of Natural Language Processing (NLP) and machine learning, refers to the process of converting a sequence of text into smaller parts, known as tokens. These tokens can be as small as characters or as long as words.

In our application, tokenization aims to construct a dictionary of unique values for a given set of features. Each combination of feature values should have a unique representation to be distinguished by the model. Essentially, we need to combine our set of features in a way that guarantees distinctiveness for correct reconstruction. Figure 4.2 shows the simplified operation of the multi-feature tokenizer.

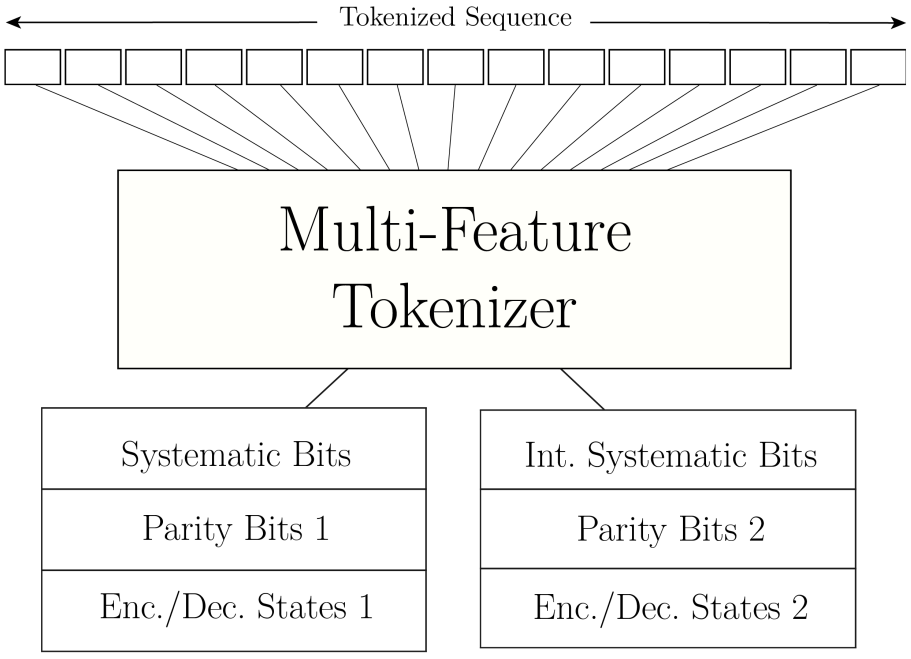


Figure 4.2: Simplified operation of the multi-feature tokenizer.

Given that the token values must be of an unsigned integer type, the proposed approach to tokenization for a set of N features of values $\{f_1, f_2, \ldots, f_N\}$, with these values in ranges

$\{R_1, R_2, \ldots, R_N\}$, is given by the following formula:

$$\text{TOKEN} = f_1 + f_2 \times (R_1) + f_3 \times (R_1 R_2) + \ldots + f_N \times (R_1 R_2 \ldots R_{N-1}) \tag{4.1}$$

Which can be expressed as:

$$\text{TOKEN} = \sum_{i=1}^{N} f_i \prod_{j=1}^{i-1} R_j \tag{4.2}$$

This mathematical solution ensures the correct reconstruction of the unique generated features using a simple iterative method, called tokenizer decoding, which proceeds as follows:

1. Initialize the values of feature ranges R, ordered according to feature placements.
2. Initialize index $i = N$, where N is the number of features.
3. Repeat until $i = 0$:
   a. Compute the scale, $s_i = \prod_{j=1}^{i-1} R_j$
   b. Compute feature value f_i using floor division of the tokenized sequence by the scale s_i.
   c. Update the tokenized sequence to be the remainder after division by the scale.
   d. Decrement the index i.

Having presented the proposed tokenization scheme and its usage for our application, we conducted a grid search on tokenizers with all possible combinations of feature placements. Our comparison focused on the reconstruction of the most important feature, the systematic bits. We altered a number of tokenized sequences within a fixed range of errors and passed them to different tokenizers for systematic bit extraction. We then compared the resulting Bit Error Rate (BER) after reconstruction.

Tokenizers that assign the largest scale to the information bits systematically outperform other configurations. A larger scale for this feature allows for a greater error margin before producing an erroneous result. The best-performing configuration is described in Table 4.1. We also note that for this configuration, the extraction of the information bit is even simpler. Specifically, a token value greater than or equal to 128 indicates an information bit of 1, while a token value less than 128 indicates an information bit of 0.

Features	Sys. Bit	Parity 1	States 1	Int. Sys. Bit	Parity 2	States 2
Values	[0, 1]	[0, 1]	[0, 1, 2, 3]	[0, 1]	[0, 1]	[0, 1, 2, 3]
Range	2	2	4	2	2	4
Notation	u	p_1	s_1	ui	p_2	s_2
$\text{TOKEN} = 128u + 64p_1 + 16s_1 + 8ui + 4p_2 + s_2$						

Table 4.1: Summary of the designed multi-feature tokenizer.

The proposed tokenizer creates 256 tokens to represent different turbo code features. The number of tokens in this collection is referred to as the vocabulary size in the literature. Additionally, we introduce special tokens that do not hold any meaningful turbo code information but are used to signal events to the transformer and organize the sequences.

First, the Start Of Sequence (SOS) token indicates the beginning of the sequence and is added to the start of a sequence to signal the model that a new sequence has begun. Next, the End Of Sequence (EOS) token signals the model that the sequence has ended. Lastly, the Padding (PAD) token is added as a filler to ensure that all sequences are of the same length. These three special tokens are assigned the next available token values: 256, 257, and 258, respectively. Including these special tokens, the total vocabulary size increases from 256 to 259.

This results in an enhanced tokenization scheme that accommodates special tokens while maintaining a distinct representation for each combination of turbo code features.

4.2.3 Embedding

Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model}. We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation. In the embedding layers, we multiply those weights by $\sqrt{d_{model}}$ [47].

4.2.4 Multi-head attention

Instead of performing a single attention function with d_{model}-dimensional keys, values, and queries, we found it beneficial to linearly project the queries, keys, and values h times with different, learned linear projections to d_k, d_k, and d_v dimensions, respectively. On each of these projected versions of queries, keys, and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 4.3.

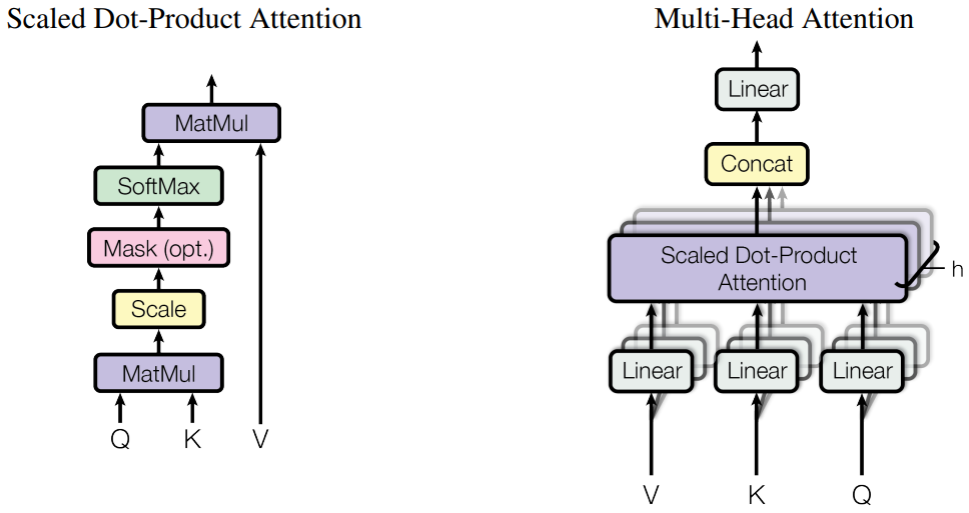


Figure 4.3: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel [47].

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{4.3}$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$. [47]

4.2.5 Feed forward layer

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between: [47]

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{4.4}$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer. [47]

4.2.6 Normalization and residual connections

Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i. [47]

4.2.7 Positional encoding

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed. In this work, we use sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{4.5}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{4.6}$$

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}. [47]

4.2.8 Encoder block

The encoder is composed of a stack of N identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is:

$$\text{LayerNorm}(x + \text{Sublayer}(x)) \tag{4.7}$$

where $\text{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce

outputs of dimension d_{model}. [47]

4.2.9 Decoder block

The decoder is also composed of a stack of N identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i. [47]

4.3 Methodology

In this section, we describe the methodology applied to the *TurboTransformer*. We specify the properties of the data used, detail the training and inference processes, and introduce the evaluation metrics employed for analyzing the results discussed in Section4.5.

4.3.1 Data preparation

The data used for this model can be represented as a source to construct a dictionary of various turbo code features. This set of features, that were earlier discussed in the model's architecture, were generated in MatLab. The environment used simulated a communication system with turbo code channel coding. We established a random generation of binary information sequences, divided into blocks of length $K = 100$. The information is encoded using the turbo code configuration as presented in the consideration on further work in 1.3.2.5.6. The encoded information is then modulated using BPSK modulation and passed through an AWGN channel, and the recieved sequences are collected. This procedure enables us to generate the following:

- Encoded sequences, from which we extract the following features: systematic bits, interleaved systematic bits and parity bits for the model's target.
- Encoding states, that represent features for the model's target.
- Received encoded sequences, from which we extract the following features: systematic bits, interleaved systematic bits and parity bits for the model's source.

The considerations regarding the SNR values used in the channel are specific to each dataset generated. Since, we generated a total of three datasets: the train set, validation set and test set.

The train set contains 800000 samples and the validation set contains 50000 samples. The SNR distributions across the received sequences in train and validation set are shown in Figure4.4. The test set mainly has low values of SNR, with a considerable amount of data at SNR$=-1dB$. This choice was motivated by multiple suggestions in literature, that state that the model learns best when faced to highly corrupted data. The validation set was chosen to have an averagely distributed SNR to specifically verify the generalization capacity of the model. The test set on the other hand contains a number of 2000 samples per SNR value in the range of -1.5 to 7 dB.

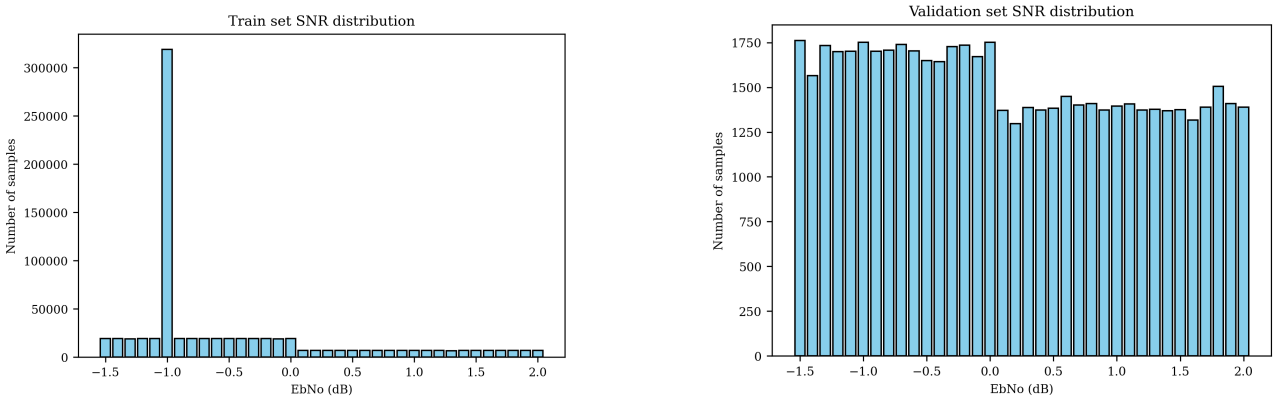


Figure 4.4: SNR distributions in train and validation sets.

We must note that transformer models are highly dependent on the quantity of data they are fed. While generating larger datasets is not particularly challenging, as the datasets described above only took approximately 16 hours to generate, the substantial size of these datasets impacts the training time of the model. Given our constrained resources, this configuration of data preparation represents the maximum feasible amount for our needs.

4.3.2 Training

The *TurboTransformer* model was trained from scratch, as the only publicly available pre-trained transformers are specific to NLP tasks. Adapting a pre-trained NLP model to our application was not considered viable due to the significant differences between our task and typical NLP applications. We will specify the training method used, the loss function and optimizer applied and present the different model configurations trained.

4.3.2.1 Training method

Various methods exist for training transformer models, including teacher forcing, scheduled sampling, and normal mode. We will focus on the chosen method: teacher forcing.

In teacher forcing mode, the decoder predicts the next token based on the correct input, referred to as the target input. The decoder leverages the cross-attention mechanism, which incorporates the attention given to the source sequence by the encoder. This approach forces the decoder to replicate the correct input patterns during training, enhancing its ability to generate accurate predictions during inference.

Unlike teacher forcing in RNNs, the transformer model executes a training step in a single pass. Instead of predicting the output token-by-token as in recurrent networks, the causal masking applied in the decoder simulates this behavior. This allows the model to encode, decode, and project once for any given sequence of any length, as each token only attends to previous tokens.

The training process described in Figure 4.5 goes as follow:

- The tokenized source sequence is concatenated with an EOS token and PAD tokens if needed. And the tokenized target sequence is concatenated with an SOS token, EOS token, and PAD tokens if needed. Adding the SOS token is crucial for the inference scheme that will be discussed later.

- The source sequence, after embedding and positional encoding, is fed to the encoder. The encoder applies self-attention to the source sequence and passes the output to the decoder.

- The target sequence, after embedding and positional encoding, is fed to the decoder. The decoder applies self-attention to the target sequence, followed by cross-attention with the encoder's output.

- The decoder output is projected, and a softmax activation is applied to provide a sequence where each position in the sequence contains the probability distribution over the entire vocabulary for the token predicition in that position.

- The loss function, which we will discuss later, takes the model output and the label sequence (a sequence with target features, tokenized and concatenated with an EOS token). The loss value is backpropagated and an optimizer updates and the model's weights.

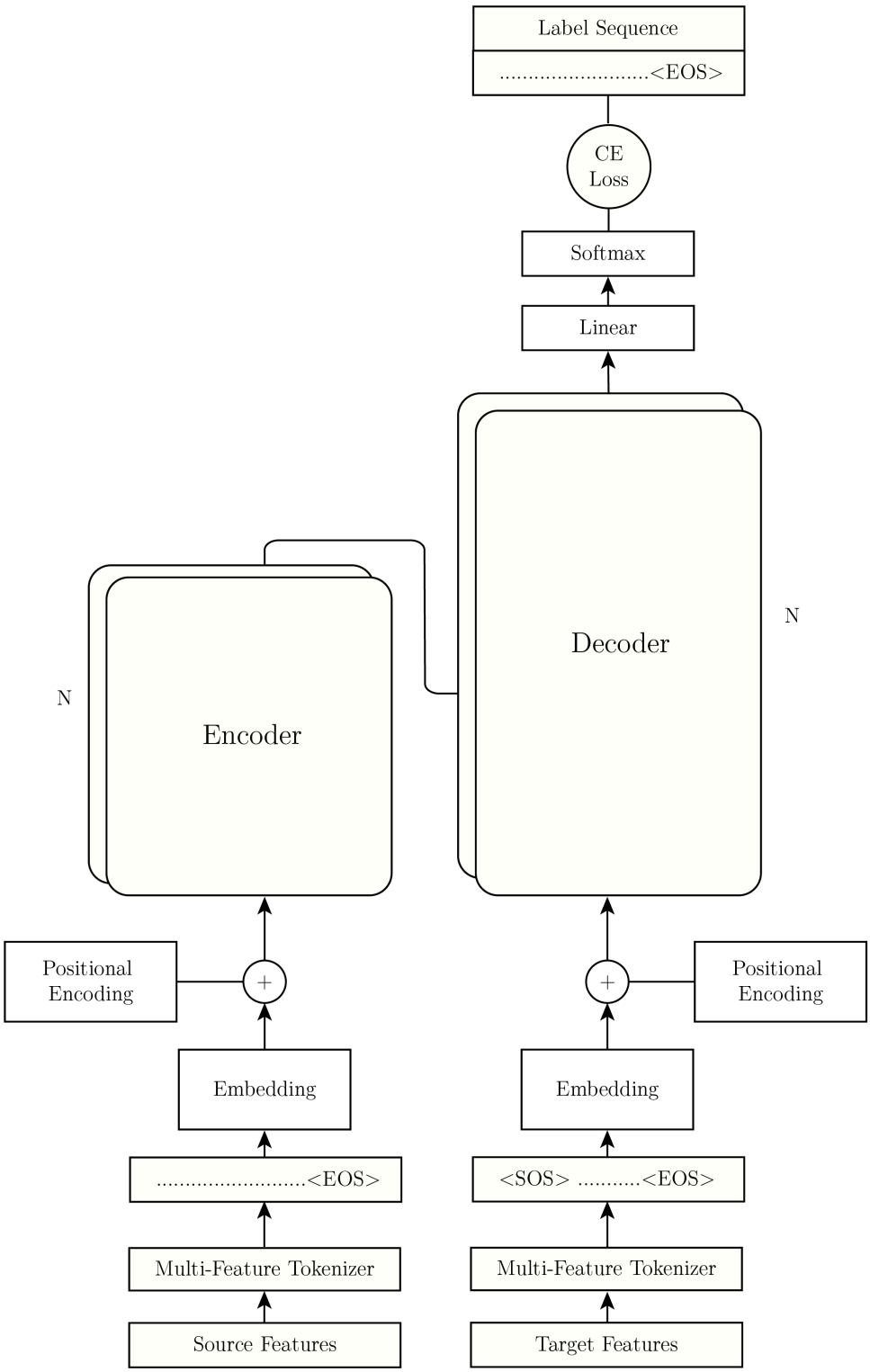


Figure 4.5: TurboTransformer during training.

4.3.2.2 Loss function

The model output provides a probability distribution over the vocabulary for each token in the sequence. The labels represent the correct tokens that the model should predict, with each label value indicating the index of the correct token in the probability vectors. Given the nature of

our task, we employ a Cross-Entropy (CE) loss function, which is well-suited for measuring the performance of tokens predictions.

For a single sequence of length N, assuming that the labels are represented by the correct tokens $\{t_1, t_2, \ldots, t_N\}$, the formula for the CE loss in this context is:

$$\text{CE} = -\frac{1}{N}\sum_{i=1}^{N}\log(p_{i,t_i}) \tag{4.8}$$

where:
t_i is the correct token index for the i-th position.
p_{i,t_i} is the predicted probability of the i-th position being the correct token t_i.

The CE loss used was often joint to the label smooting technique. It is a regularization technique used during training to improve the generalization of neural networks. In the context of CE loss, label smoothing adjusts the target probabilities y by distributing some probability mass from the correct target 1 to other incorrect targets. This is achieved by interpolating between the hard targets and a uniform distribution.

For a single sequence of length N, assuming label smoothing parameter ϵ, the modified target distribution $\tilde{y}$ is:

$$\tilde{y}_{i,t_i} = (1-\epsilon)\cdot y_{i,t_i} + \frac{\epsilon}{V} \tag{4.9}$$

where:
y_{i,t_i} is the original target probability for correct token t_i at position i.
V is the total number of possible tokens, or vocabulary size.
ϵ is a small positive smoothing parameter.

The modified CE loss with label smoothing is then computed as:

$$\text{CE}_{\text{smooth}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{V}\tilde{y}_{i,k}\log(p_{i,k}) \tag{4.10}$$

where:
$p_{i,k}$ is the predicted probability of token k at position i.

Adjusting ϵ allows for controlling the degree of smoothing applied during training.

4.3.2.3 Optimizer

For our application, we employ the Adaptive Moment Estimation with Weight Decay (AdamW) optimizer. It is an extension of the Adaptive Moment Estimation (Adam) optimizer. The AdamW optimizer improves upon Adam by decoupling the weight decay from the gradient update, which helps to ensure that L2 regularization does not interfere with the learning rate.

The configuration of the optimizer used covers values of the exponential decay rates for the moment estimates, β_1 and β_2, fixed at 0.9 and 0.98 respectively and the value of the weight decay coefficient, λ. The optimization process is defined the Algorithm 4

Algorithm 4 Optimization using AdamW.

Require: α: learning rate
Require: $\beta_1, \beta_2 \in [0, 1]$: exponential decay rates
Require: λ: weight decay coefficient
Require: θ_0: parameter set
Require: $f(\theta)$: network function
  $m_0 \leftarrow 0$
  $v_0 \leftarrow 0$
  $t \leftarrow 0$
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot (\hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) + \lambda \cdot \theta_{t-1})$
  end while
  return θ_t

where:

g_t is the gradient of the loss function with respect to the parameters at time step t.

m_t and v_t are the first and second moment estimates, respectively.

$\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected moment estimates.

α is the learning rate.

ϵ is a small constant to prevent division by zero.

When using AdmaW instead of Adam, we reaffirmed that the decoupling of weight decay from the gradient update leads to more stable convergence, which is crucial for training transformer models. Also the weight decay term acts as a regularizer, preventing overfitting and helping the model generalize better.

During training, we observed that setting the weight decay value relatively high, such as $\lambda = 0.05$ or higher, in initial stages promotes a smoother convergence. In advanced stages, this value was reduced to decrease regularization, allowing the model to focus more on learning.

4.3.2.4 Model configurations and hyperparameters

Given the absence of a predefined starting point for the *TurboTransformer*, the process of selecting the optimal model configuration involved extensive experimentation. A model configuration is characterized by the following components:

- N: This parameter represents the number of stacked encoder and decoder blocks.

- d_{model}: This denotes the dimensionality of the model's hidden states or embeddings. It determines the size of the vector space in which the model operates.

Before presenting the experimented configuration, we address the learning rate hyperparameter. We observed that the use of a learning rate scheduler is crucial for the convergence during training. We used the following types of schedulers:

- Scheduler with learning rate warmup posposed in [47]. This scheduler was used at training start, where the warm-up stage was observed to be crucial for a faster and stable convergence. The scheduling formula, based on the current step is given by:

$$\alpha = d_{\text{model}}^{-0.5} \cdot \min(\text{step\_num}^{-0.5}, \text{step\_num} \cdot \text{warmup\_steps}^{-1.5}) \tag{4.11}$$

  This corresponds to increasing the learning rate linearly for the first warmup_steps training steps, and decreasing it thereafter proportionally to the inverse square root of the step number [47].

- The learning rate reducing on plateau. This scheduler reads a metrics quantity and if no improvement is seen for an epoch, the learning rate is reduced. It is called when the learning stagnates and the tracked metric stopped improving. We used this technique at late training stages with the validation loss as the tracked metric and a reducing factor $k = 0.5$.

Building on this foundation, we introduce the experimented configurations in Table 4.2, focusing primarily on the *tiny*, *small*, *medium*, and *large* architecture setups.

Configurations	**Tiny**	**Small**	**Medium**	**Large**
d_model	128	256	256	512
Number of layers	6	4	6	12
Block Length	100	100	100	100
Vocabulary Size	259	259	259	300,000
Train epochs	30	30	30	15
Training time (hours)	62	76	94	210
Batch size	32-1024	64-2048	64-4096	256-4096
Dropout	0.1	0.1	0.1	0.3
Loss Function	Cross-Entropy	Cross-Entropy	Cross-Entropy	Cross-Entropy
Label Smoothing	0.05	0.01	0.01	0.005
Optimizer	AdamW	AdamW	AdamW	AdamW
Weight Decay	0.05	0.05	0.05	0.01
Warmup Steps	2500	3000	3000	4000
Trainable Parameters (Millions)	2.86	7.56	11.24	88.61

Table 4.2: Experimental configurations for *TurboTransformer*.

Our initial experimentation began with the *large* configuration, which featured a highly complex model with 88.61M parameters. This choice was inspired by findings from [48], demonstrating that increasing both model width and depth can expedite convergence in terms of gradient steps and training duration.

Our observations indicate that the *large* model experienced significant underfitting, likely due to its complexity and challenges in pattern detection. Switching to smaller configuration not only accelerated the training but offered better convergence and overall better performance.

The chosen configuration for *TurboTransformer* is the *medium* setup. Models with fewer than 6 layers in both encoder and decoder blocks proved lacking during inference. Similarly, models with a dimensionality less than 256 struggled to capture a detailed enough representation of sequences.

4.3.3 Inference

Various variations exist for the inference of transformer model. These variations get in common the same baseline process of decoding, the greedy decoding. The inference process for *TurboTransformer* is described in Figure 4.5.

4.3.3.1 Greedy decoding

The greedy decoding process goes as follow:

- The tokenized source sequence is concatenated with an EOS token and PAD tokens if needed.
- The source sequence, after embedding and positional encoding, is fed to the encoder. The encoder applies self-attention to the source sequence and passes the output to the decoder. The same encoder output is used across all decoding iterations for a given sequence.
- The decoder input is initially a SOS token. The process aims to fill this sequence over the decoding iterations. The decoder applies self-attention to the target sequence, followed by cross-attention with the encoder's output.
- The decoder output is projected, and a softmax activation is applied to provide a sequence where each position in the sequence contains the probability distribution over the entire vocabulary for the token predicition in that position.
- The output probabilities go through an argmax function. Argmax returns the index of the maximum value in the input array. In this case, we obtain the next predicted token.
- The predicted token is fed back and concatenated to the decoder output for the next decoding iteration.
- The steps mentioned above are repeated until the model predicts the total number of tokens in a sequence.
- At last, we obtain a decoder input buffer with the complete predicted sequence, right-shifted by the SOS token.

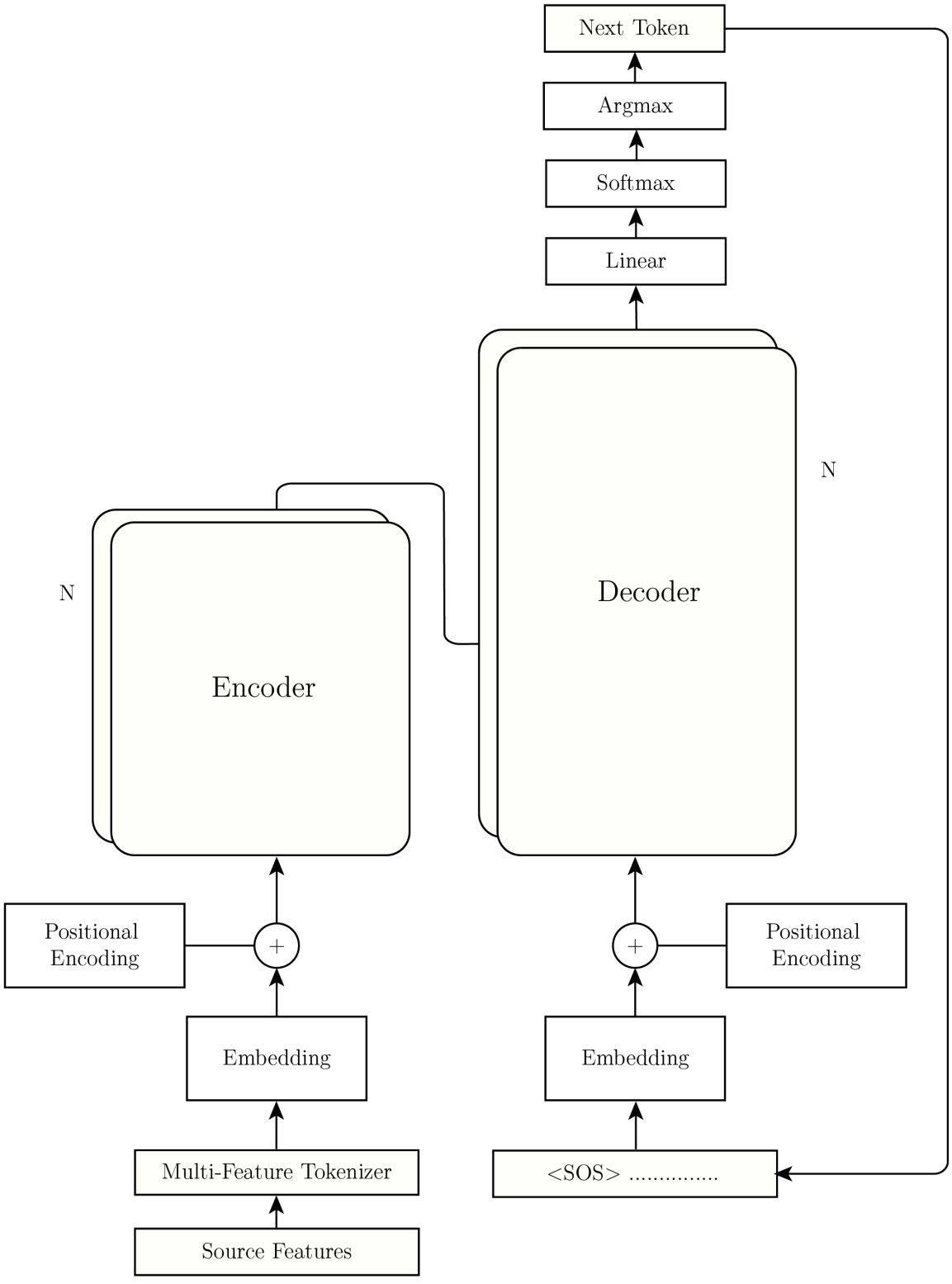


Figure 4.6: TurboTransformer during inference.

Greedy decoding is a simple and computationally effective method to predict corrected tokens. The predicted tokens are then used to reconstruct the turbo code features. The reconstruction of the information bits feature finalizes the process of decoding turbo codes using **TurboTransformer**.

Given the tokenization scheme described earlier, an alternative approach to the extracting of information bits was considered. This technique takes advantage of the tokenization scheme used and the redundancy of information present in a tokenized sequence. We call the classic approach "Hard reconstruction" and the proposed technique "Soft reconstruction". Both techniques are explained in the following:

4.3.3.1.1 Hard reconstruction of information bits simply consists of using the tokenizer decoding presented in the process in 4.2.2 and selecting the systematic bits feature.

4.3.3.1.2 Soft reconstruction of information bits takes advantages of the existence of two features that convey knowledge on the information bits. Indeed, both the systematic bits and interleaved systematic bits features can be used to obtain a better reconstruction of information.

We know that a token at a position i will contain insight on the predicted bit for position i through the systematic bit feature, but also on the predicted bit for the interleaved position $int(i)$ through the interleaved systematic bit feature. Inversely, the same applies for a token at the interleaved position $int(i)$ that will contain insight on the predicted bit for position $int(i)$ through the systematic bit feature, but also on the predicted bit for the interleaved position i through the interleaved systematic bit feature. Essentially, this offers a doubled prediction for each information bit. The predictions hold probabilities that represent the model's confidence in the selection of token. These probabilities combined with the decisions on a specific bit can be used to offer a more reliable reconstruction of information bits.

Given token predictions at position i and $int(i)$, with a probabilities P_i and $P_{int(i)}$ to which we associate weight factors w_1 and w_2, and bit decisions u_i systematic and $u_{int(i)}$ interleaved systematic respectively which we transform to a signed representation (bit 0 takes -1 and bit 1 takes 1), we define the soft reconstruction of information bit $\hat{u}_i$ for position i as follow:

$$\hat{u}_i = w_1 \cdot (u_i P_i) + w_2 \cdot (u_{int(i)} P_{int(i)}) \tag{4.12}$$

The Figure 4.7 illustrates the explained workflow of the proposed approach to bit reconstruction.

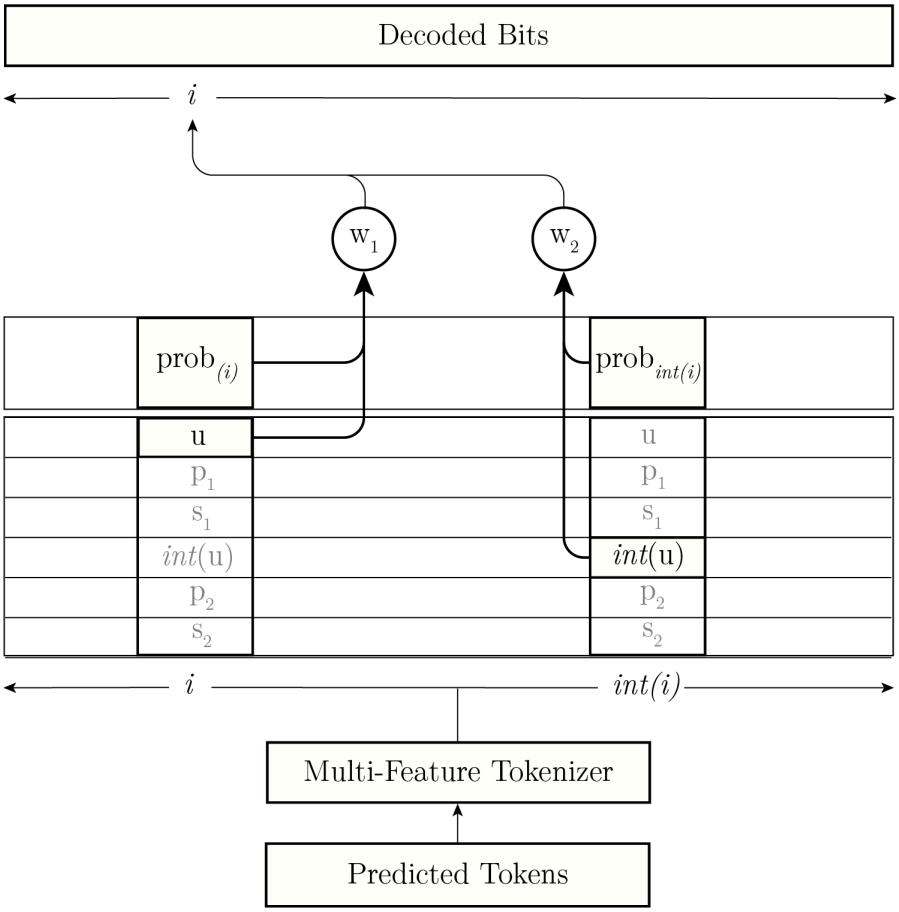


Figure 4.7: Illustrated process of the soft reconstruction of information bits.

This reconstruction approach has proven efficient for greedy decoding, and the results of its application are presented in the upcoming sections. Despite some improvements to greedy decoding, it remains constrained by the limitations of the standard iterative technique. Greedy decoding for token prediction suffers from multiple disadvantages. First, it is short-sighted: Greedy search only considers the best immediate option at each step without regard for the long-term implications of its choices. Second, it is prone to the propagation of errors: Once a erroneous decision is made, all future decisions are affected, with no mechanism to correct the course. Therefore, we introduce beam search decoding next to mitigate some of these issues.

4.3.3.2 Beam search decoding

Beam search is an advanced decoding technique used in sequence generation tasks, such as in Turbo decoding, where it maintains multiple candidate sequences, known as the beam width k, at each decoding step. This approach contrasts with Greedy Search, which simply selects the most likely word at each step without considering alternatives.

To illustrate, in the turbo decoding context. Greedy Search settles on the most probable token

at each step, potentially missing a more coherent overall sequence. In contrast, Beam Search expands upon this by predicting several possible continuations for each sequence it's considering, evaluating and retaining the top k sequences based on their combined probabilities. This flexibility increases the likelihood of finding a more contextually appropriate sequence, even if it doesn't always guarantee the most probable one. Figure 4.8 shows a simplified explanation of the search process, where T_i are the candidate tokens for a sequence position i.

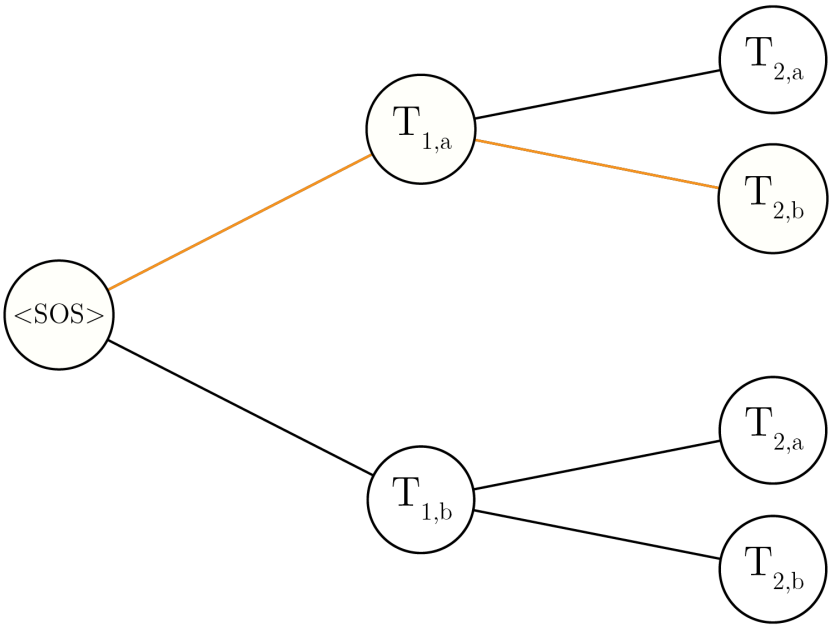


Figure 4.8: Simplified beam search for a beam size of $k = 2$.

Beam search's advantage lies in its ability to explore a wider range of potential sequences, often offering more reliable outputs compared to Greedy Search. By maintaining multiple hypotheses of predictions, it mitigates the risk of prematurely committing to less likely sequences due to locally optimal decisions.

However, beam search comes with computational costs. It requires resources to manage and compute probabilities for multiple sequences at each decoding step. Additionally, the effectiveness of beam search is influenced by the choice of beam width k; a smaller k may restrict its ability to capture the most probable sequence effectively.

4.3.4 Evaluation metrics

The train, validation, and test performances are quantified using the following metrics:

Bit Error Rate (BER) and Block Error Rate (BLER): These metrics are already defined.

Token Error Rate: TER is a metric used to evaluate the performance of a token-based model.

It measures the number of token errors to the total number of tokens. The TER is given by:

$$\text{TER} = \frac{\text{Number of erroneous predicted tokens}}{\text{Total number of tokens}} \tag{4.13}$$

Mean Absolute Error: MAE is a common regression metric that measures the average magnitude of the errors in a set of predictions, without considering their direction. It is the average over the test sample of the absolute differences between prediction and actual observation. The MAE is given by:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{4.14}$$

where y_i is the actual token, $\hat{y}_i$ is the predicted token, and n is the number of observations.

Perplexity: is a measurement of how well a probability model predicts a sample. It is commonly used in language modeling to evaluate the performance of models in inference. Lower perplexity indicates better predictive performance. The perplexity is given by:

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i)\right) \tag{4.15}$$

where N is the number of tokens, and $P(w_i)$ is the probability of the i-th token in the sequence.

Additionally, during training, gradient norms-2 are monitored. Gradient norms-2 provide information on the magnitude of the gradients during training, which helps in diagnosing issues like vanishing or exploding gradients. Monitoring these norms ensures that the gradients are within a reasonable range and helps in stabilizing the training process. The gradient norm-2 is given by:

$$\|\nabla\|_2 = \sqrt{\sum_i (\nabla_i)^2} \tag{4.16}$$

where ∇_i represents the gradient of the i-th parameter.

4.4 Experimental setup

The setup for ***TurboTransformer*** is similar to *TurboAttention* in 3.

4.4.1 Hardware

The cloud computational environment of Kaggle is used, with 73.1 GB disk space available , Intel Xeon 2.20 GHz CPU. The GPU used is **Nvidia Tesla P100** with **16 GB VRAM**, and **32 GB RAM**. In addition, the platform provides a double GPU **Nvidia Tesla T4**, this latter was used to divide *TurboTransformer* model into two main parts and run each one on a GPU,

this will help to improve training conditions of larger models.

4.4.2 Software

To develop the model, we use **PyTorch 2.4** with **Python 3.10** and **CUDA 12.1** to run the GPU.

4.4.3 Implementation

In order to develop the code for *TurboTransformer* we use the basic architecture of transformer which its source code provided in online github repository *"pytorch-transformer"*[1].

The additional features of *TurboTransformer* were implemented from scratch.

4.5 Results and analysis

We present the results of *TurboTransformer* and its performance over the training and validation steps for all of the presented configuration. Then the results of tests, using the selected configuration "Medium".

4.5.1 Training results

For the training results, we only present the most important metrics for the configurations that were not selected as the final *TurboTransformer* model.

4.5.1.1 Results of the "Large" configuration

Figure 4.9 represents the train and validation loss of the given configuration.

[1]PyTorch Transformer code available at: https://github.com/hkproj/pytorch-transformer

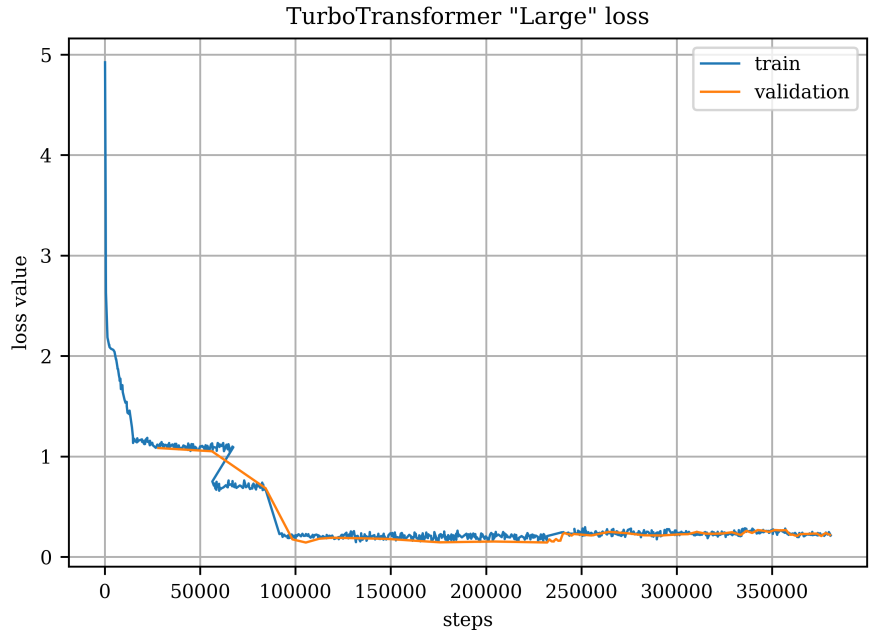


Figure 4.9: TurboTransformer "Large": train and validation loss.

4.5.1.2 Results of the "Medium" configuration

Figures 4.10, 4.11, 4.12, 4.13, 4.14 and 4.15 represent the train and validation loss, the TER, MAE, Perplexity, BER and gradient norms evolution of the given configuration.

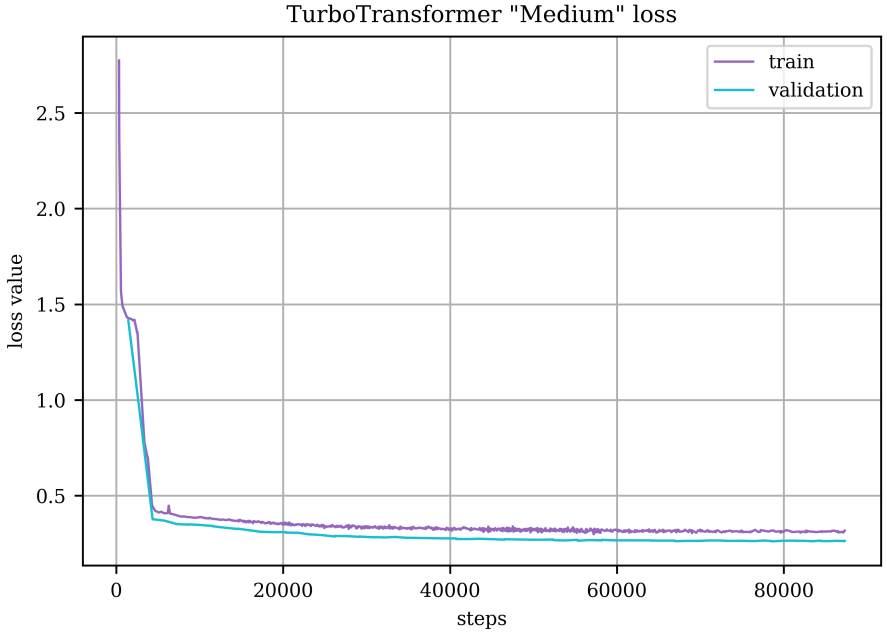


Figure 4.10: TurboTransformer "Medium": train and validation loss.

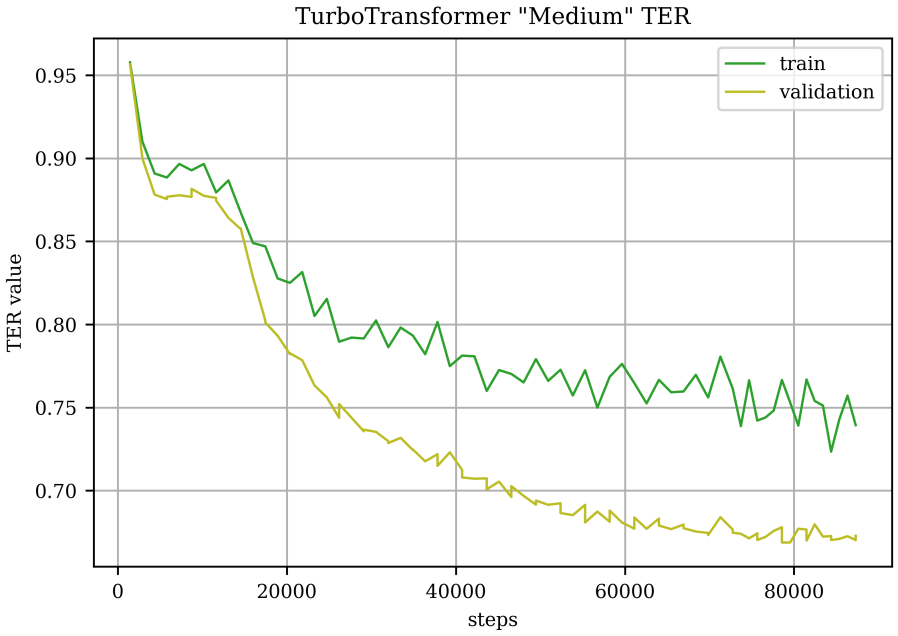


Figure 4.11: TurboTransformer "Medium": train and validation TER.

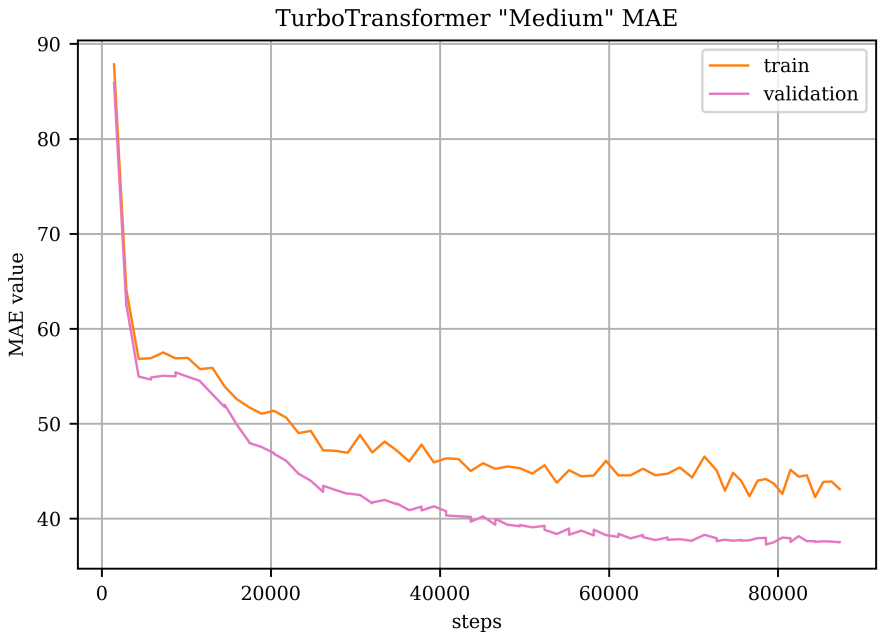


Figure 4.12: TurboTransformer "Medium": train and validation MAE.

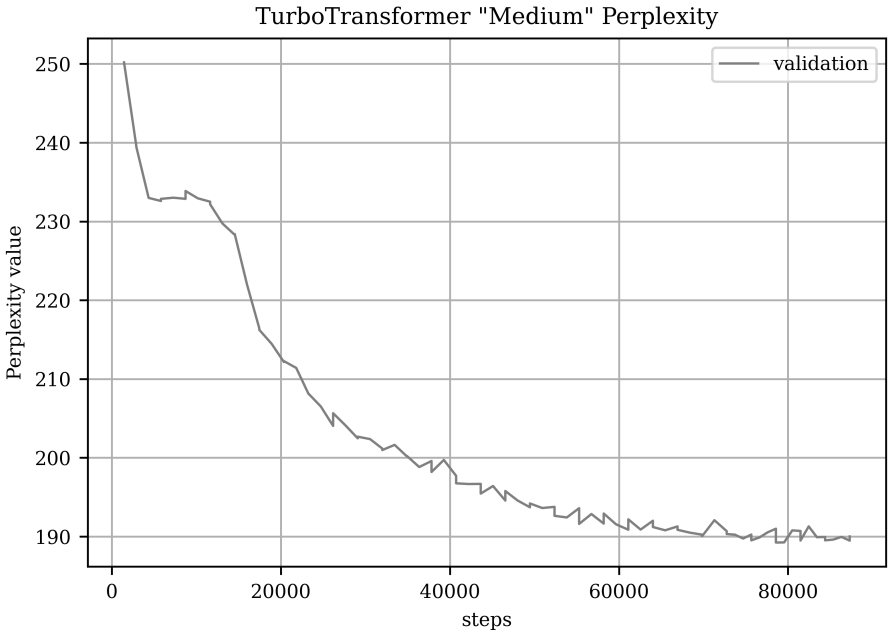


Figure 4.13: TurboTransformer "Medium": validation Perplexity.

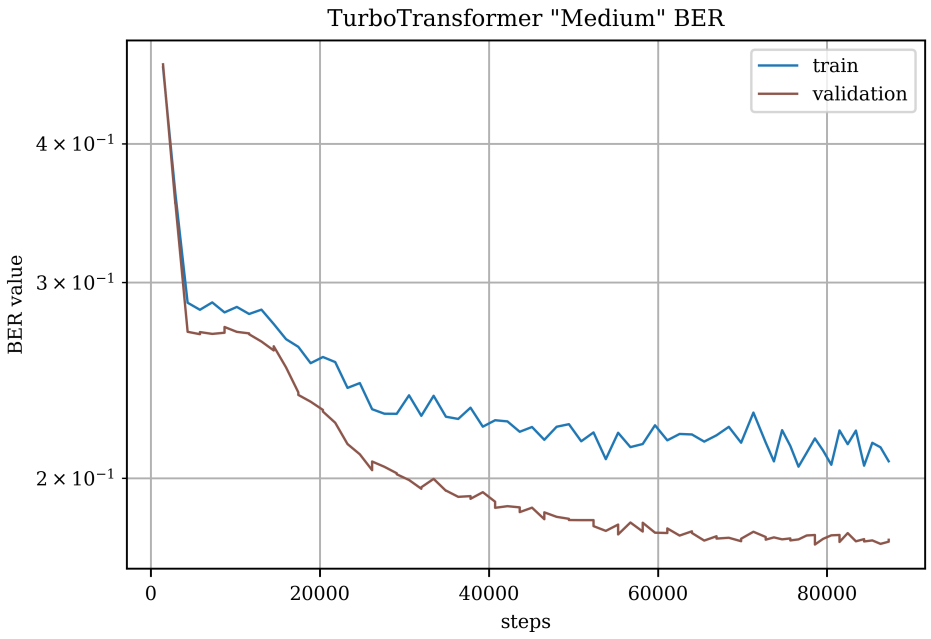


Figure 4.14: TurboTransformer "Medium": train and validation BER.

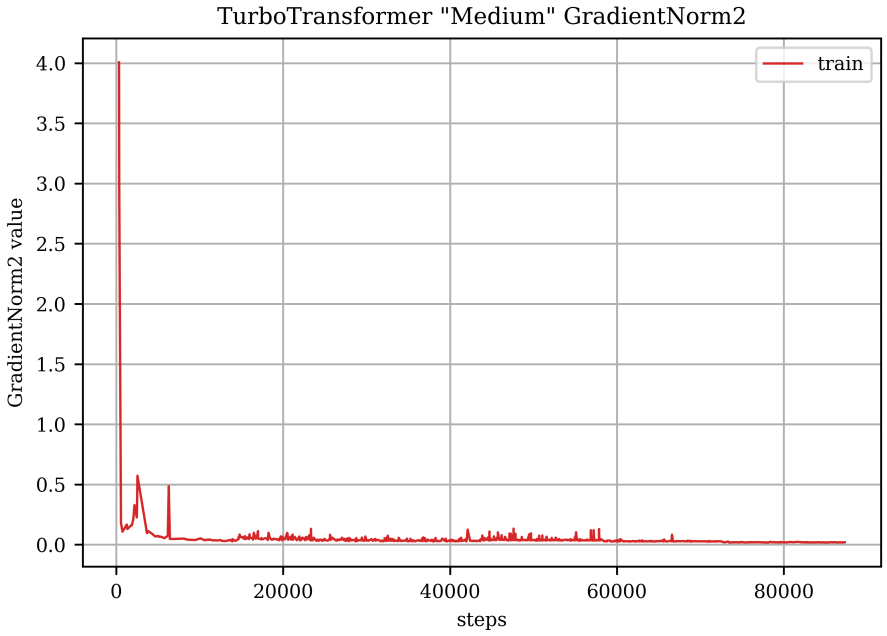


Figure 4.15: TurboTransformer "Medium": train gradient norms.

4.5.1.3 Results of the "Small" configuration

Figures 4.16, 4.17 and 4.18 represent the train and validation loss, TER and BER for the given configuration.

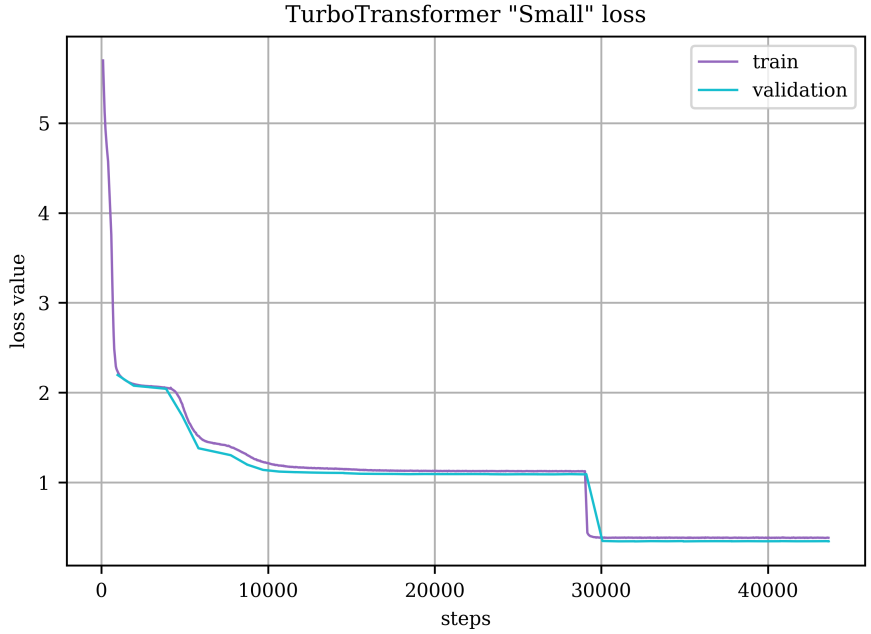


Figure 4.16: TurboTransformer "small": train and validation loss.

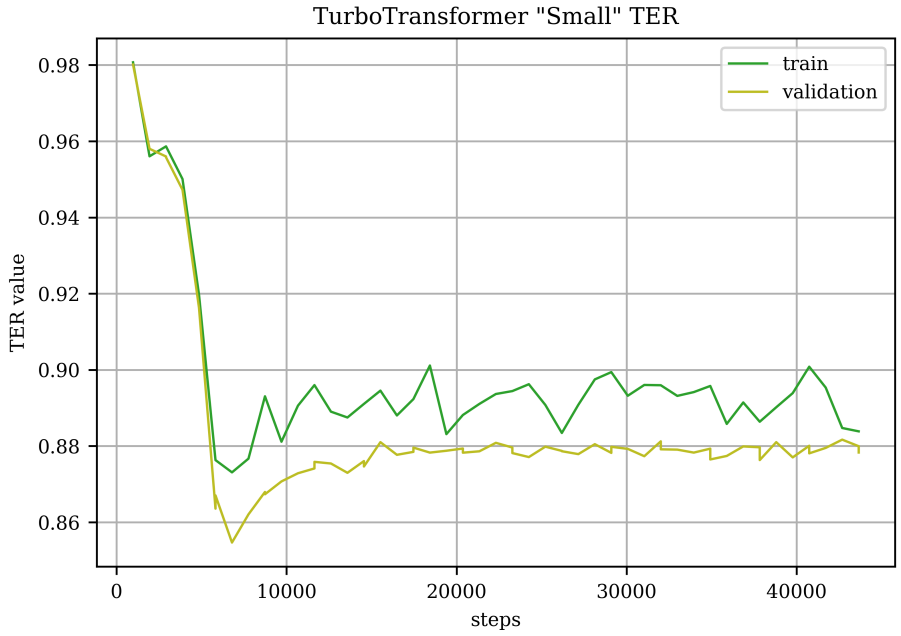


Figure 4.17: TurboTransformer "small": train and validation TER.

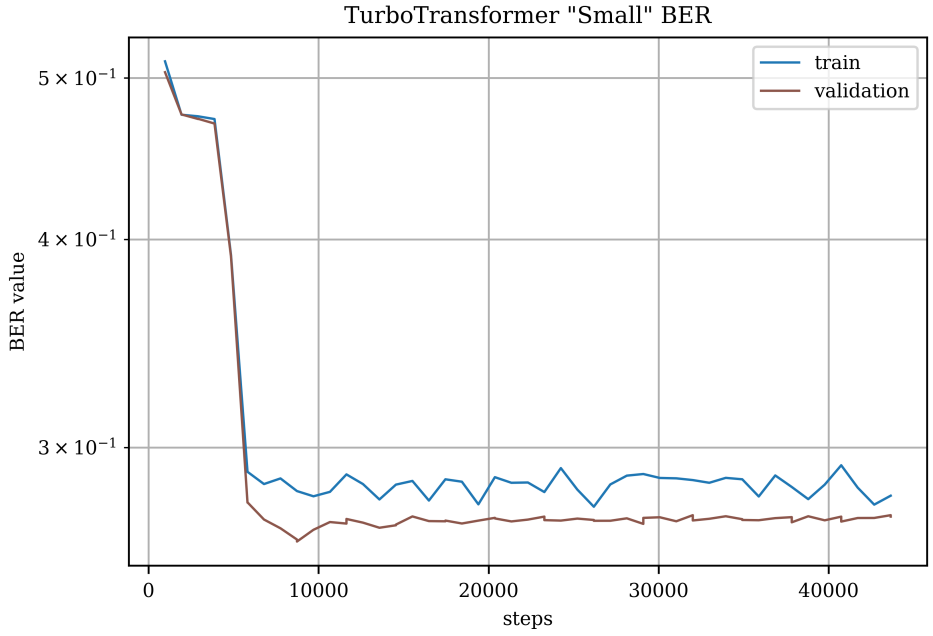


Figure 4.18: TurboTransformer "small": train and validation BER.

4.5.1.4 Results of the "Tiny" configuration

Figures 4.19, 4.20 and 4.21 represent the train and validation loss, TER and BER for the given configuration.

Figure 4.19: TurboTransformer "tiny": train and validation loss.

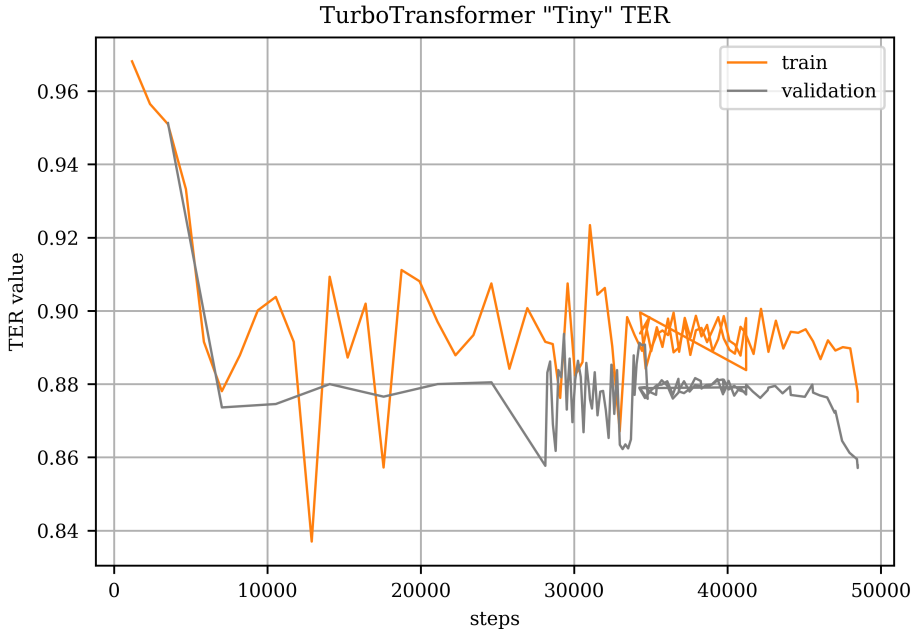


Figure 4.20: TurboTransformer "tiny": train and validation TER.

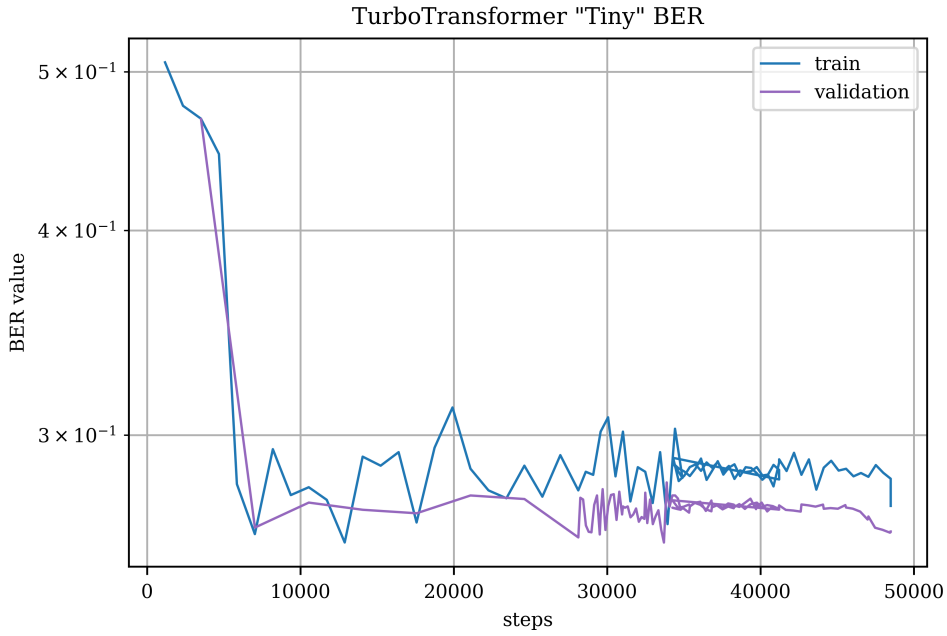


Figure 4.21: TurboTransformer "tiny": train and validation BER.

4.5.1.5 Discussion

From the training results obtained for each configuration of the model, several observations can be made:

- All configurations exhibit a stable learning process as shown by the evolution of loss values. Sudden plateau changes in loss at certain steps can be attributed to adjustments in the loss function parameter, specifically label smoothing, which was tuned during training to improve model performance.

- Comparing the evolution of train and validation losses across all configurations reveals close alignment between their values. This consistency suggests that the models are not underfitting, especially given the large size of the dataset, which typically mitigates risks of underfitting. Regarding overfitting, while it's unclear whether the models have reached their global minima, the "Medium" configuration consistently showed the best convergence.

- The TER and BER show a strong correlation in their evolution. The model's performance in information bit reconstruction is notably efficient, owing to the effectiveness of the proposed tokenizer, which significantly reduces error rates from token prediction to bit reconstruction.

- Perplexity values for the "Medium" configuration closely track TER evolution, reaffirming perplexity's role in assessing the model's predictive efficiency.

- Gradient norms for the "Medium" configuration were monitored throughout training. Initial stages showed steep gradient explosions, which were effectively managed using gradient clipping. This technique involves limiting gradient norms-2 to a threshold value to stabilize training. In this case, a threshold of $th = 2.5$ was applied to prevent excessive gradient magnitudes.

4.5.2 Test results

For the tests, we conduct an evaluation of the selected model configuration "Medium" over an extended range of SNRs. Then we proceed by comparing the model to the iterative decoding SOVA approach.

4.5.2.1 Evaluation of the model

Figures , , and represent the test TER, MAE, BER and BLER respectively for the final selected configuration "Medium".

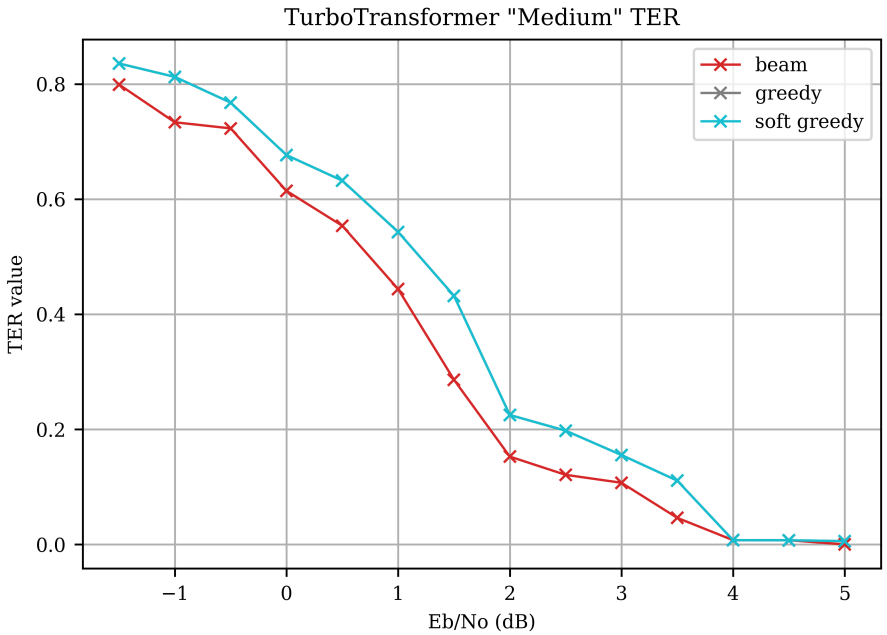


Figure 4.22: TurboTransformer "Medium": test TER.

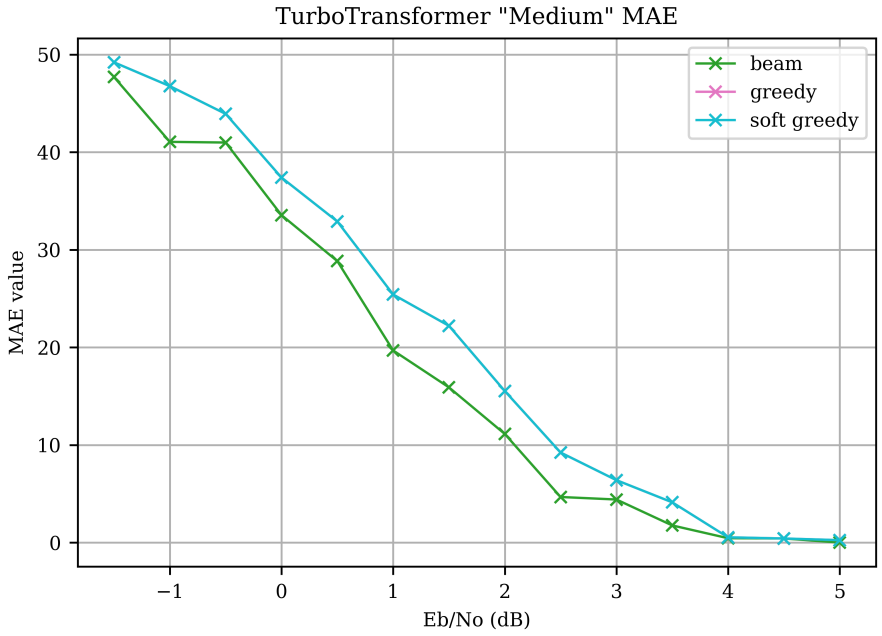


Figure 4.23: TurboTransformer "Medium": test MAE.

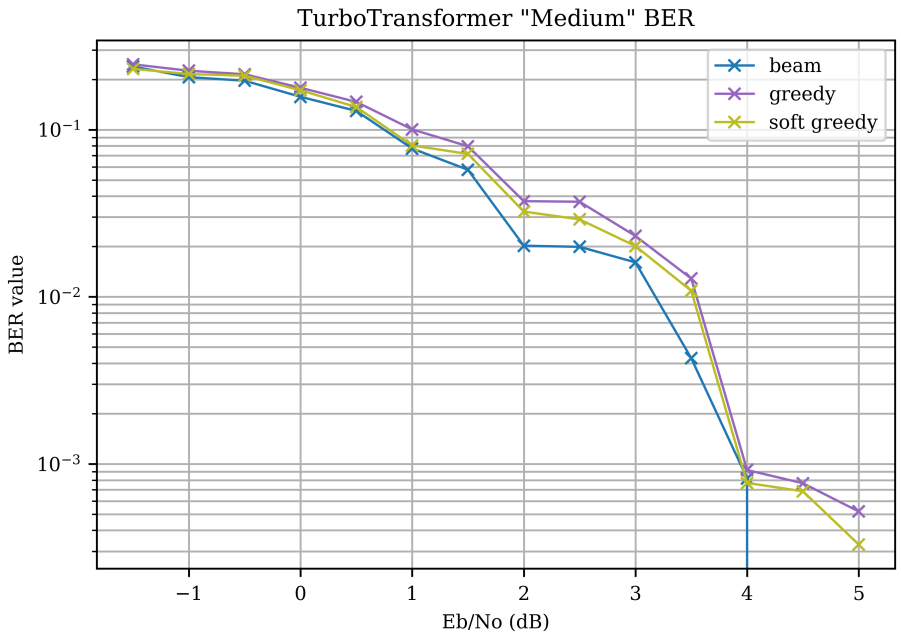


Figure 4.24: TurboTransformer "Medium": test BER.

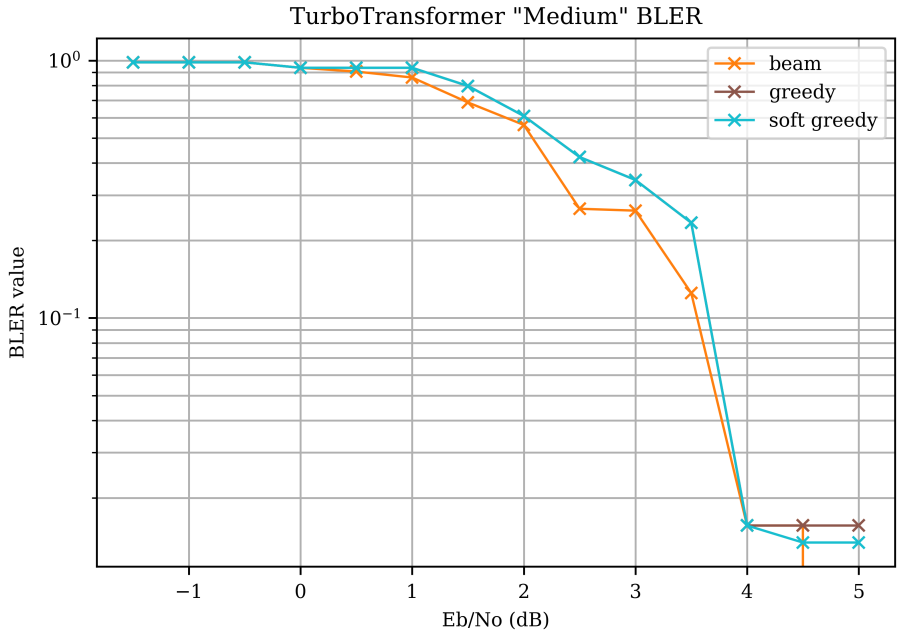


Figure 4.25: TurboTransformer "Medium": test BLER.

4.5.2.2 Comparison to SOVA

Figure 4.26 shows the results of our model *TurboTranformer* in terms of BER performance compared to the studied iterative decoder SOVA.

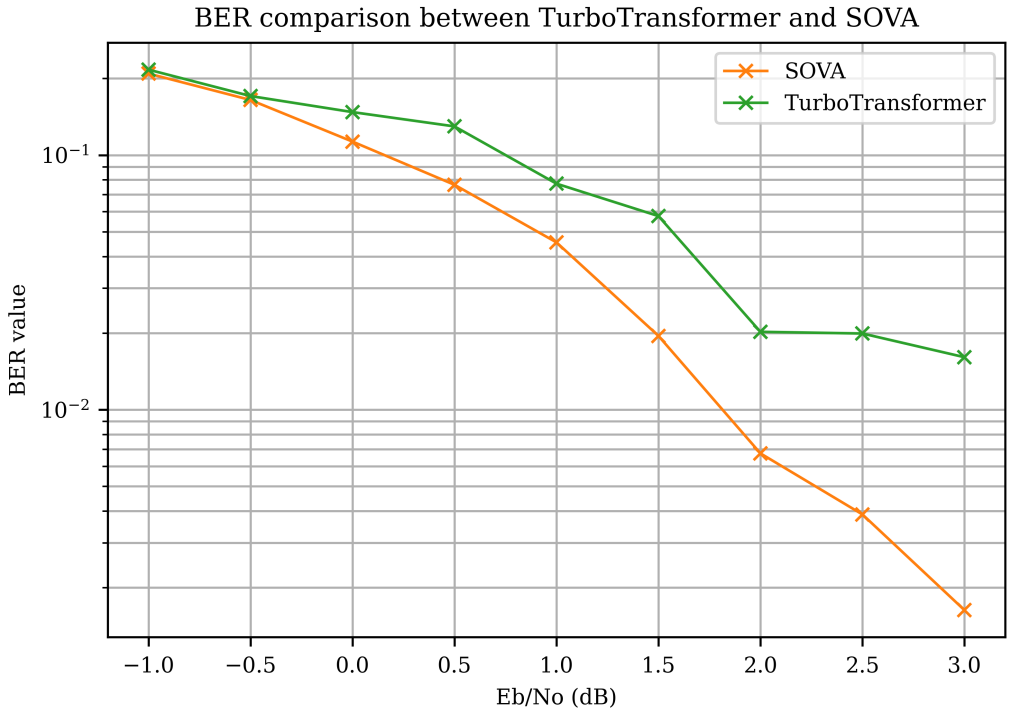


Figure 4.26: BER comparison between TurboTransformer and SOVA.

4.5.2.3 Discussion

From the test results given for the selected configuration "Medium" for *TurboTransformer*, we can extract the following interpretations:

- The consistent decrease in evaluation metrics across varying SNR ranges indicates the model's ability to generalize predictions across data of different natures. This reaffirms the importance of training on samples with lower SNR values to enhance overall performance.
- When comparing the different decoding techniques, we observe:
    - Naturally, both the greedy and soft greedy techniques perform similarly on token prediction-based metrics, due to their shared estimation scheme.
    - Beam search decoding consistently outperforms greedy decoding across all SNRs, utilizing a beam size of $k = 5$.
    - While beam search improves token prediction, its impact on BLER performance is less pronounced, suggesting limitations in predicting entirely correct bit sequences at lower SNRs.
    - The proposed soft reconstruction of information bits demonstrates superior BER performance compared to traditional methods across the entire SNR range. However, it falls short compared to beam search decoding due to its inherent disadvantage relative to the greedy approach.
- When comparing the *TurboTransformer* performance to the SOVA algorithm, we observe:
    - At lower SNRs, specifically at $E_b/N_0 = -1 : -0.5dB$, *TurboTransformer* matches the SOVA performance. This shows that the train set, containing samples of low SNRs helped the model predict in this range.
    - At higher SNRs, starting from $E_b/N_0 = 2dB$, the difference in performance becomes highly noticeable. *TurboTransformer* could was able to generalize to a limited extent. This suggests that the inference performance is highly reliant on the nature of the data from which the model learned.

4.5.3 Inference execution time on edge hardware

We performed inference tests of *TurboTransformer* on the NVIDIA *Jetson Nano*, in the same way for *TurboAttention*.

We evaluate the performance of the trained model by running inference with a batch size of 20 and measuring the batch execution time, sequence execution time, GPU usage, and RAM usage. The results are summarized in Table 4.3.

Model	Batch Size	Batch Execution Time (s)	Sequence Execution Time (ms)	GPU Usage (%)	RAM Usage (MB)
TurboTransf	20	16	800	78.4	513

Table 4.3: Inference performance on *Jetson Nano*

TurboTransformer has the highest time complexity compared to other models, *DEEPTURBO* and I, due to its large number of parameters which represents a computing complexity.

Figure 4.27 illustrates the interface *Jetson Stats*, tracking real-time metrics such as GPU and CPU usage, RAM consumption, power consumption, and component temperatures during the execution of the *Python3*-based **PyTorch** model.

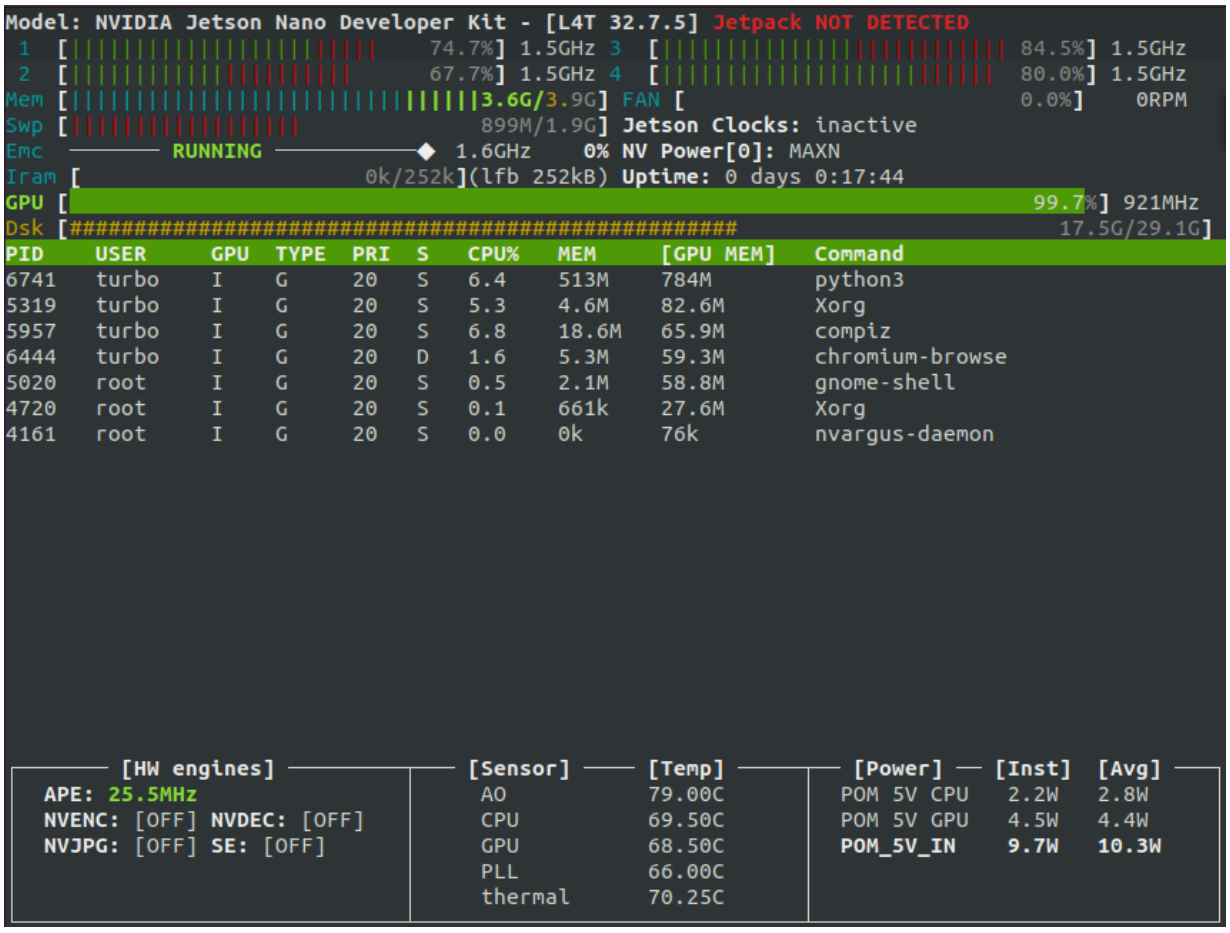


Figure 4.27: ***Jetson Stats*** : Usage of resources usage on Jetson Nano of *TurboTransformer*

4.6 Discussion

Based on the architecture explanation of *TurboTransformer* and the provided results, the training process yielded satisfactory outcomes. However, the large dataset and extensive training time constrained further improvements. Extending the training period would likely enhance

performance significantly. Comparing our results with the SOVA iterative decoding demonstrates competitive performance, though some gaps remain, especially at higher SNRs. This could be a topic for further fine-tuning with higher SNR training sets. Additionally, the inference time of the selected configuration is less efficient, indicating the need for developing more performant inference approaches in the future.

4.7 Conclusion

The application of transformers in turbo decoding, exemplified by *TurboTransformer*, holds considerable promise for advancing error correction techniques in communication systems. While our current results indicate a need for further training and optimization, *TurboTransformer* represents a novel approach for this application. The absence of prior studies focusing on transformers in turbo decoding underscores the novelty and potential of this work. Future efforts should prioritize enhancing training methodologies to achieve even better performance, particularly in challenging conditions such as SNRs. Additionally, optimizing the inference efficiency of *TurboTransformer* remains a critical area for improvement, offering opportunities for developing more efficient decoding approaches. Overall, *TurboTransformer* establishes a foundation for future research endeavors in utilizing transformer architectures for advanced error correction applications.

General Conclusion

This work encompasses extensive results and details related to the subject and objectives of this study.

First, it includes a detailed analysis of turbo decoding, covering related aspects such as convolutional codes, turbo codes, their various structures, and key characteristics.

Simulations were conducted for the SOVA turbo decoder, with experiments demonstrating its performance under different conditions.

A theoretical overview of neural networks is also provided, focusing on architectures useful for turbo decoding, such as sequence models like LSTMs and RNNs. Basic methods for training and optimizing these models, including optimization algorithms and hyperparameter tuning, are introduced.

To investigate related works and gain a better understanding of existing methods, the state-of-the-art section reviews various machine learning techniques used for turbo and convolutional code decoding over the past three decades.

Inspired by prior works on neural network-based turbo decoders and natural language processing models, we present *TurboAttention* and *TurboTransformer*, attention-based models. These models, originally developed for NLP tasks, were adapted to explore their utility in turbo decoding and to investigate their performance in this context.

Finally, inference tests on the Jetson Nano were successfully conducted using the available environment for parallel computing AI models. The resource usage varied depending on the complexity of the models.

Perspectives

The promising results achieved by ***TurboAttention*** and ***TurboTransformer*** highlight the potential for further enhancements through refined training and tuning. Scaling these models to achieve higher accuracies and surpass current benchmarks could involve enriching the training datasets with diverse data variations such as varying block lengths, different SNR levels, and complex channel conditions, leveraging their foundation in NLP models trained on extensive datasets.

Balancing model performance with computational efficiency is crucial for integrating these advancements into high-throughput communication systems. Enhancements in neural network turbo decoders, such as shared weighting schemes across iterations, could offer a viable approach to achieve this balance.

Exploring the applicability of these algorithms to other types of channel codes, particularly those used in advanced communication systems like 6G and 5G, presents an exciting avenue for future research.

Given the resource-intensive nature of attention models, optimizing computational resources and experimenting with different configurations and hyperparameters will be essential to realizing their full potential.

Finally, considering the implementation of these decoders on programmable hardware like FPGA holds significance, as it can validate their utility in real-world applications and clears the way for practical deployment.

Bibliography

[1] C. Berrou, A. Glavieux, and P. Thitimajshima. "Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1". In: ICC '93 - IEEE International Conference on Communications. IEEE. DOI: 10.1109/icc.1993.397441. URL: http://dx.doi.org/10.1109/ICC.1993.397441.

[2] John G. Proakis. *Digital Communications*. en. McGraw-Hill Science, Engineering Mathematics, 2001. ISBN: 9780072321111. URL: https://books.google.com/books/about/Digital_Communications.html?hl=&id=sbr8QwAACAAJ.

[3] Junyi Li, Xinzhou Wu, and Rajiv Laroia. *OFDMA Mobile Broadband Communications*. Jan. 17, 2013. DOI: 10.1017/cbo9780511736186. URL: http://dx.doi.org/10.1017/CBO9780511736186.

[4] Tom Richardson and Rüdiger Urbanke. *Modern Coding Theory*. Mar. 17, 2008. DOI: 10.1017/cbo9780511791338. URL: http://dx.doi.org/10.1017/CBO9780511791338.

[5] L.C. Perez, J. Seghers, and D.J. Costello. "A distance spectrum interpretation of turbo codes". In: *IEEE Transactions on Information Theory* 42.6 (1996), pp. 1698–1709. ISSN: 0018-9448. DOI: 10.1109/18.556666. URL: http://dx.doi.org/10.1109/18.556666.

[6] Tom Richardson. "Error Floors of LDPC Codes". In: *ResearchGate* (Jan. 2003). Available at: https://www.researchgate.net/publication/228641078_Error_floors_of_LDPC_codes.

[7] Hyeji Kim et al. "Communication Algorithms via Deep Learning". In: (2018). DOI: 10.48550/ARXIV.1805.09317. URL: https://arxiv.org/abs/1805.09317.

[8] Branka Vucetic and Jinhong Yuan. *Turbo Codes*. en. Springer, June 30, 2000. ISBN: 9780792378686. URL: https://books.google.com/books/about/Turbo_Codes.html?hl=&id=AsaoGJcIp4QC.

[9] P. Elias. "Error-free Coding". In: *Transactions of the IRE Professional Group on Information Theory* 4.4 (Sept. 1954), pp. 29–37. ISSN: 2168-2690. DOI: 10.1109/tit.1954.1057464. URL: http://dx.doi.org/10.1109/TIT.1954.1057464.

[10] G. Forney. "Convolutional codes I: Algebraic structure". en. In: *IEEE Transactions on Information Theory* 16.6 (Nov. 1970), pp. 720–738. ISSN: 0018-9448. DOI: 10.1109/tit.1970.1054541. URL: http://dx.doi.org/10.1109/TIT.1970.1054541.

[11] Shu Lin and Daniel J. Costello. *Error Control Coding*. en. Pearson, 2004. ISBN: 9780130426727. URL: https://books.google.com/books/about/Error_Control_Coding.html?hl=&id=ENwdtAEACAAJ.

[12] Banerjee. *NOC: Error Control Coding: An Introduction to Convolutional Codes*. This introductory course will discuss theory of convolutional codes, their encoding and decoding techniques as well as their applications in digital communications. Mar. 2016. URL: https://archive.nptel.ac.in/courses/117/104/117104120/.

[13] Claude Berrou. *Codes et turbocodes*. fr. Springer Science Business Media, Mar. 13, 2007. ISBN: 9782287327391. URL: https://books.google.com/books/about/Codes_et_turbocodes.html?hl=&id=W5v779TZp1UC.

[14] Fu-hua Huang. *Evaluation of Soft Output Decoding for Turbo Codes*. Virginia, USA, May 1997.

[15] Jing Sun and O.Y. Takeshita. "Interleavers for turbo codes using permutation polynomials over integer rings". In: *IEEE Transactions on Information Theory* 51.1 (Jan. 2005), pp. 101–119. ISSN: 0018-9448. DOI: 10.1109/tit.2004.839478. URL: http://dx.doi.org/10.1109/tit.2004.839478.

[16] Lucian Trifina, Daniela Tarniceriu, and Valeriu Munteanu. "Improved QPP interleavers for LTE standard". In: 2011 10th International Symposium on Signals, Circuits and Systems (ISSCS). IEEE, June 2011. DOI: 10.1109/isscs.2011.5978745. URL: http://dx.doi.org/10.1109/ISSCS.2011.5978745.

[17] A. Viterbi. "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm". In: *IEEE Transactions on Information Theory* 13.2 (Apr. 1967), pp. 260–269. ISSN: 0018-9448. DOI: 10.1109/tit.1967.1054010. URL: http://dx.doi.org/10.1109/TIT.1967.1054010.

[18] L. Bahl et al. "Optimal decoding of linear codes for minimizing symbol error rate (Corresp.)" en. In: *IEEE Transactions on Information Theory* 20.2 (Mar. 1974), pp. 284–287. ISSN: 0018-9448. DOI: 10.1109/tit.1974.1055186. URL: http://dx.doi.org/10.1109/TIT.1974.1055186.

[19] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Draft of February 3, 2024. n/a, 2023.

[20] Jeffrey L. Elman. "Finding Structure in Time". en. In: *Cognitive Science* 14.2 (Mar. 1990), pp. 179–211. ISSN: 0364-0213. DOI: 10.1207/s15516709cog1402_1. URL: http://dx.doi.org/10.1207/s15516709cog1402_1.

[21] Berrie Trippe. “Evaluation of Neural Networks for Sequence Classification”. Supervised by dr. Vlado Menkovski and Simon Koop MSc. Master’s thesis. Eindhoven, Netherlands: Eindhoven University of Technology, May 2021. URL: https://research.tue.nl/en/studentTheses/c49cb687-d974-4338-b681-c18024c5399e.

[22] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. 3rd. Draft of February 3, 2024. New Jersey: Prentice Hall, 2023. URL: https://web.stanford.edu/~jurafsky/slp3/.

[23] Christopher Jan-Steffen Brix. “Extension of the Attention Mechanism in Neural Machine Translation”. Updated version. The submitted thesis is available at the RWTH Aachen. Bachelor’s Thesis. Aachen, Germany: Rheinisch-Westfälische Technische Hochschule Aachen, Mar. 2018.

[24] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: (2014). DOI: 10.48550/ARXIV.1409.3215. URL: https://arxiv.org/abs/1409.3215.

[25] Ronald J. Williams and David Zipser. “A Learning Algorithm for Continually Running Fully Recurrent Neural Networks”. en. In: *Neural Computation* 1.2 (June 1989), pp. 270–280. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.2.270. URL: http://dx.doi.org/10.1162/neco.1989.1.2.270.

[26] Samy Bengio et al. “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks”. In: (2015). DOI: 10.48550/ARXIV.1506.03099. URL: https://arxiv.org/abs/1506.03099.

[27] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: (2014). DOI: 10.48550/ARXIV.1409.0473. URL: https://arxiv.org/abs/1409.0473.

[28] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation”. In: (2015). DOI: 10.48550/ARXIV.1508.04025. URL: https://arxiv.org/abs/1508.04025.

[29] W.R. Caid and R.W. Means. “Neural network error correcting decoders for block and convolutional codes”. In: GLOBECOM ’90: IEEE Global Telecommunications Conference and Exhibition. IEEE. DOI: 10.1109/glocom.1990.116658. URL: http://dx.doi.org/10.1109/GLOCOM.1990.116658.

[30] G. Marcone, E. Zincolini, and G. Orlandi. “An Efficient Neural Decoder for Convolutional Codes”. English. In: *European Transactions on Telecommunications and Related Technologies* 6.4 (1995), pp. 439–445. ISSN: 1120-3862.

[31] Xiao-An Wang and S.B. Wicker. “An artificial neural net Viterbi decoder”. In: *IEEE Transactions on Communications* 44.2 (1996), pp. 165–171. ISSN: 0090-6778. DOI: 10.1109/26.486609. URL: http://dx.doi.org/10.1109/26.486609.

[32] R. Annauth and H.C.S. Rughooputh. “Neural network decoding of turbo codes”. In: International Conference on Neural Networks. IEEE. DOI: 10.1109/ijcnn.1999.836196. URL: http://dx.doi.org/10.1109/IJCNN.1999.836196.

[33] Sujan Rajbhandari, Zabih Ghassemlooy, and Maia Angelova. “Adaptive ‘soft’ sliding block decoding of convolutional code using the artificial neural network”. en. In: *Transactions on Emerging Telecommunications Technologies* 23.7 (Mar. 21, 2012), pp. 672–677. ISSN: 2161-3915. DOI: 10.1002/ett.2523. URL: http://dx.doi.org/10.1002/ett.2523.

[34] Stevan M. Berber, Paul J. Secker, and Zoran A. Salcic. “Theory and application of neural networks for 1/n rate convolutional decoders”. en. In: *Engineering Applications of Artificial Intelligence* 18.8 (Dec. 2005), pp. 931–949. ISSN: 0952-1976. DOI: 10.1016/j.engappai.2005.05.001. URL: http://dx.doi.org/10.1016/j.engappai.2005.05.001.

[35] Zoran Salcic, Stevan Berber, and Paul Secker. “FPGA Prototyping of RNN Decoder for Convolutional Codes”. en. In: *EURASIP Journal on Advances in Signal Processing* 2006.1 (Apr. 18, 2006). ISSN: 1687-6180. DOI: 10.1155/asp/2006/15640. URL: http://dx.doi.org/10.1155/ASP/2006/15640.

[36] Yihan Jiang et al. “DeepTurbo: Deep Turbo Decoder”. In: (2019). DOI: 10.48550/ARXIV.1903.02295. URL: https://arxiv.org/abs/1903.02295.

[37] Li Zhang et al. “A Parallel Turbo Decoder Based on Recurrent Neural Network”. en. In: *Wireless Personal Communications* 126.2 (June 6, 2022), pp. 975–993. ISSN: 0929-6212. DOI: 10.1007/s11277-022-09779-8. URL: http://dx.doi.org/10.1007/s11277-022-09779-8.

[38] Raja Sattiraju, Andreas Weinand, and Hans D. Schotten. “Performance Analysis of Deep Learning based on Recurrent Neural Networks for Channel Coding”. In: 2018 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS). IEEE, Dec. 2018. DOI: 10.1109/ants.2018.8710159. URL: http://dx.doi.org/10.1109/ANTS.2018.8710159.

[39] Eren Balevi and Jeffrey G. Andrews. “Autoencoder-Based Error Correction Coding for One-Bit Quantization”. In: (2019). DOI: 10.48550/ARXIV.1909.12120. URL: https://arxiv.org/abs/1909.12120.

[40] Yihan Jiang et al. “Turbo Autoencoder: Deep learning based channel codes for point-to-point communication channels”. In: (2019). DOI: 10.48550/ARXIV.1911.03038. URL: https://arxiv.org/abs/1911.03038.

[41] Karl Chahine et al. “Turbo Autoencoder with a Trainable Interleaver”. In: (2021). DOI: 10.48550/ARXIV.2111.11410. URL: https://arxiv.org/abs/2111.11410.

[42] Homayoon Hatami, Hamid Saber, and Jung Hyun Bae. “Performance Evaluation of Turbo Autoencoder with Different Interleavers”. In: 2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring). IEEE, June 2023. DOI: 10.1109/vtc2023-spring57618.2023.10200461. URL: http://dx.doi.org/10.1109/VTC2023-Spring57618.2023.10200461.

[43] Minghua Cao et al. “LSTM Attention Neural-Network-Based Signal Detection for Hybrid Modulated Faster-Than-Nyquist Optical Wireless Communications”. en. In: *Sensors* 22.22 (Nov. 20, 2022), p. 8992. ISSN: 1424-8220. DOI: 10.3390/s22228992. URL: http://dx.doi.org/10.3390/s22228992.

[44] C. E. Shannon. “A Mathematical Theory of Communication”. en. In: *Bell System Technical Journal* 27.3 (July 1948), pp. 379–423. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x. URL: http://dx.doi.org/10.1002/j.1538-7305.1948.tb01338.x.

[45] “ENHANCING SEQUENCE LEARNING WITH MASKING IN RECURRENT NEURAL NETWORKS”. In: *International Research Journal of Modernization in Engineering Technology and Science* (Jan. 20, 2024). ISSN: 2582-5208. DOI: 10.56726/irjmets48489. URL: http://dx.doi.org/10.56726/IRJMETS48489.

[46] Yoni Choukroun and Lior Wolf. *Error Correction Code Transformer*. 2022. arXiv: 2203.14966 [cs.LG]. URL: https://arxiv.org/abs/2203.14966.

[47] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[48] Zhuohan Li et al. “Train Large, Then Compress: Rethinking Model Size for Efficient Training and Inference of Transformers”. In: *CoRR* abs/2002.11794 (2020). arXiv: 2002.11794. URL: https://arxiv.org/abs/2002.11794.

APPENDIX : SOVA code implementation

SOVA code implementation

```
% Function name: SOVA
%
% Goal:
% This SOVA function is designed to decode 4-state convolutional codes with
% the ability to provide soft output information.
% The primary purpose of this function is to deliver both the hard decision
% output (decoded bit sequence) and the associated soft metric values
%  for each bit in the sequence.
%
% Inputs:
% - y1: systematic information bits sequence.
% - y2: parity bits sequence.
% - Lc: channel reliability value.
% - Le_in: extrinsic information from previous decoder.
% - norm: soft information normalization settings,
%         vector 1x2 of scale and threshold value.
% - BITS_SIZE: block length value.
%
% Outputs:
% - decoded_bits: estimated decoded sequence.
% - Le_out: extrinsic information produced.
% - ML_path: maximum Likelihood path/ decoding states.
%
% Example usage:
% - For SOVA_1 block:
% [decoded_u, Le_1, decoding_states_1] = ...
% SOVA(y1, y2, Lc, deintrlv(Le_2), normalization, 100)
% - For SOVA_2 block:
% [decoded_u_interleaved, Le_2, decoding_states_2] = ...
% SOVA(intrlv(y1), y3, Lc, intrlv(Le_1), normalization, 100)
%
% External functions used:
% - Function name: get_trellis

function [decoded_bits,Le_out, ML_path] = sova(y1, y2, Lc, Le_in, norm,
    BITS_SIZE)

% Initialization of path and branch metrics
M = zeros(4,2,BITS_SIZE+1);          % All path metrics
M1 = zeros(4,BITS_SIZE+1);           % All Survivor paths metrics
M2 = zeros(4,BITS_SIZE+1);           % All Competing paths metrics
Mt = zeros(4,2);                     % Branch metrics for time t

% Initialization of survivor and competing branches
SURV_branches = ones(4,BITS_SIZE);
```

```
45 COMP_branches = ones(4,BITS_SIZE);
46
47 % Initialization of ML/competing paths and correspoding hard decision
48 ML_path = ones(1,BITS_SIZE+1);
49 COMP_path = ones(1,BITS_SIZE+1);
50 ML_bits = zeros(1,BITS_SIZE);
51 COMP_bits = zeros(1,BITS_SIZE);
52
53 % Initialization of the reliability values
54 Rel = zeros(1,BITS_SIZE);
55
56
57 % Forward iterations,
58 % Compute all paths metrics, survivor and competing paths
59 for ii=2:BITS_SIZE+1
60
61     % Get trellis properties for the first 2 timesteps then keep it for the
    rest of the sequence
62     if ii<=4
63         [Prev_State,Prev_u,Prev_p,~] = get_trellis(ii);
64     end
65
66     % Compute next path metrics
67     Mt(:,1) = M1(Prev_State(:,1),ii-1) + Lc*Prev_u(:,1)*y1(ii-1) + Lc*Prev_p
    (:,1)*y2(ii-1) + Le_in(ii-1)*Prev_u(:,1);
68     Mt(:,2) = M1(Prev_State(:,2),ii-1) + Lc*Prev_u(:,2)*y1(ii-1) + Lc*Prev_p
    (:,2)*y2(ii-1) + Le_in(ii-1)*Prev_u(:,2);
69
70     % Store paths metrics
71     M(:,:,ii) = Mt;
72
73     % Extract survivor and competing metrics and indices
74     [M1(:,ii), SURV_branches_index]= max(Mt,[],2);
75     [M2(:,ii), ~]= min(Mt,[],2);
76
77     % Deduce surv. and comp. branches from metric index
78     SURV_branches(:,ii-1) = transpose(Prev_State(sub2ind(size(Prev_State),
    1:length(SURV_branches_index), SURV_branches_index')));
79     COMP_branches(:,ii-1) = transpose(Prev_State(sub2ind(size(Prev_State),
    1:length(SURV_branches_index), (3-SURV_branches_index)')));
80 end
81
82 % Back tracing,
83 % Compute the ML path and decoded sequence
84
85 [~,~,~,pn_u] = get_trellis(1);
86
87 for jj=BITS_SIZE:-1:1
```

```
88      % Tracing from last all-zeros state through the survivor branches
89      ML_path(jj) = SURV_branches(ML_path(jj+1),jj);
90      ML_bits(jj) = pn_u(ML_path(jj),ML_path(jj+1));
91  end
92
93  % Compute and update the realibility values
94  for ii=1:BITS_SIZE
95
96      Rel(ii) = 0.5*abs(M1(ML_path(ii+1),ii+1) - M2(ML_path(ii+1),ii+1));
97
98      if(ii<=3)
99          % Attribute reliability value to the first 2 timesteps
100         Rel(1:ii) = Rel(ii);
101     else
102         MEM = ii;
103
104         % Extraction of the competing path
105         COMP_path(ii+1) = ML_path(ii+1);
106         COMP_path(ii) = COMP_branches(COMP_path(ii+1),ii);
107
108         for jj=ii-1:-1:1
109             COMP_path(jj) = SURV_branches(COMP_path(jj+1),jj);
110
111             % Store levels where surv. and comp. bit estimation differs
112             if( pn_u(COMP_path(jj),COMP_path(jj+1)) ~= ML_bits(jj))
113                 MEM = [jj MEM];
114             end
115
116             % No need to continue if ML and comp. paths merge
117             if COMP_path(jj)==ML_path(jj)
118                 break
119             end
120         end
121
122         % Update reliability values from smallest to largest MEM level
123         for kk=length(MEM):-1:2
124             Rel(MEM(kk-1)) = min(Rel(MEM(kk-1:length(MEM))));
125         end
126     end
127 end
128
129 % Hard ecoded sequence
130 decoded_bits = ML_bits;
131
132 % Normalization parameters of extrinsic information
133 scale = norm(1);
134 threshold = norm(2);
135
```

```
% Compute extrinsic information
Le_out = scale*(ML_bits.*Rel - Lc*y1' - Le_in);
Le_out(Le_out>=threshold) = threshold;
Le_out(Le_out<=-threshold) = -threshold;

end
```

We also provide the code for the dependencies of the given SOVA function. The code for the external functions used by the SOVA function is provided below.

```
% Function name: get_trellis
%
% Goal:
% Extract the trellis properties of the 757 convolutional encoder
% based of the current time step in trellis.
% Information is provided using predefined lookup tables.
%
% Inputs:
% - t: time step in the trellis.
%
% Outputs:
% - Prev_State: row indexing with current state,
%               columns give possible previous states.
% - Prev_u: row indexing with current state,
%           columns give systematic bit decisions.
% - Prev_p: row indexing with current state,
%           columns give parity bit decisions.
% - pn_u: row indexing with previous state,
%         column indexing with current state,
%         gives systematic bit decision.

function [Prev_State, Prev_u, Prev_p, pn_u] = getTrellis(t)

if(t<=3)
    % Tables for trellis debut
    Prev_State = [1,1; 3,3; 1,1; 3,3];
    Prev_u = [-1,-1; 1,1; 1,1; -1,-1];
    Prev_p = [-1,-1; -1,-1; 1,1; 1,1];
else
    % Tables if trellis construction completed
    Prev_State = [1,2; 3,4; 1,2; 3,4];
    Prev_u = [-1,1; 1,-1; 1,-1; -1,1];
    Prev_p = [-1,1; -1,1; 1,-1; 1,-1];
end

pn_u = [-1,0,1,0; 1,0,-1,0; 0,1,0,-1; 0,-1,0,1];

end
```