On the Correspondence Between Classic Coding Theory and Machine Learning

Bachelorarbeit

zur Erlangung des akademischen Grades eines

Bachelor of Science

im Studiengang Informatik

vorgelegt von

Julia Rechberger

Matrikelnummer 3740901
s1022029@stud.uni-frankfurt.de
jm.rechberger@gmail.com

angefertigt am Lehrgebiet
Systems Engineering for Vision and Cognition
der Goethe-Universität Frankfurt am Main

Betreuer: Prof. Dr. Visvanathan Ramesh

23.07.2021

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Ebenso bestätige ich, dass diese Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

Frankfurt am Main, 23.07.2021

....
Unterschrift (Vorname, Nachname)

Summary

When we browse via WiFi on our laptop or mobile phone, we receive data over a noisy channel. The received message may differ from the one that was sent originally. Luckily it is often possible to reconstruct the original message but it may take a lot of time. That's because decoding the received message is a complex problem, NP-hard to be exact. As we continue browsing, new information is sent to us in a high frequency. So if lags are to be avoided and as memory is finite, there is not much time left for decoding.
Coding theory tackles this problem by creating models of the channels we use to communicate and tailor codes based on the channel properties. A well known family of codes are Low-Density Parity-Check codes (LDPC codes), they are widely used in standards like WiFi and DVB-T2. In practical settings the complexity of decoding a received message can be heavily reduced by using LDPC codes and approximative decoding algorithms.
This thesis lays out the basic construction of LDPC codes and a proper decoding using the sum-product algorithm. On this basis a neural network to improve decoding is introduced. Therefore the sum-product algorithm is transformed into a neural network decoder. This approach was first presented by Nachmani et al. and treated in detail by Navneet Agrawal in 2017. To find out how machine learning can improve the codes, the bit error rates of the trained neural network decoder are compared with the bit error rates of the classic sum-product algorithm approach. Experiments with static and dynamic training datasets of diverse sizes, various signal-to-noise ratios, a feed forward as well as a recurrent architecture show how to tune the neural network decoder even further. Results of the experiments are used to verify statements made in Agrawal's work. In addition, corrections and improvements in the area of metrics are presented. An implementation of the neural network to facilitate access for others will be made available to the public.

Contents

Abbreviations iv

Notations iv

1 Introduction 1

2 Coding Theory 1
- 2.1 Channels . . . 3
  - 2.1.1 Binary Symmetric Channel (BSC) . . . 3
  - 2.1.2 Additive White Gaussian Noise (AWGN) . . . 4
- 2.2 Performance metrics: Bit-Error Rate (BER), Block Error Rate (BLER) and Signal-to-Noise Ratio (SNR) . . . 4
  - 2.2.1 Signal-to-noise ratio (SNR) . . . 4
  - 2.2.2 Bit-Error Rate (BER) . . . 5
  - 2.2.3 Block Error Rate (BLER) . . . 6

3 Low-Density Parity-Check Codes (LDPC-Codes) 7
- 3.1 Linear Codes Basics . . . 7
- 3.2 Generate LDPC Codes . . . 9
- 3.3 Decoding LDPC Codes - Message Passing Decoding . . . 10
  - 3.3.1 Bit-Flipping Algorithm . . . 11
  - 3.3.2 Sum-Product Algorithm (SPA) . . . 12
- 3.4 Runtime . . . 15

4 Neural Networks to Advance LDPC-Codes 17
- 4.1 Transforming the Tanner Graph . . . 17
- 4.2 Layers . . . 18
- 4.3 From messages to activation functions . . . 23

5 Implementation 27
- 5.1 Activation Functions . . . 27
- 5.2 Loss Function . . . 28
- 5.3 Hard Decision . . . 28
- 5.4 Dimensions . . . 29

6 Experiments 29
- 6.1 Metrics . . . 30
- 6.2 Datasets . . . 31
  - 6.2.1 Training Dataset . . . 32
  - 6.2.2 Validation Dataset . . . 32
  - 6.2.3 Test Dataset . . . 32
- 6.3 Experiment: Training Dataset Size . . . 33
- 6.4 Experiment: Train on All-Zero Codeword . . . 37
- 6.5 Experiment: Batch Size . . . 39

6.6 Experiment: SNR_{dB} Combinations for Training 41
6.7 Experiment: On the Fly Training Set - Training Dataset Size 45
6.8 Experiment: On the Fly Training Set - Batch Size 47
6.9 Experiment: Feed Forward (FNN) versus Recurrent Neural Network (RNN) 49

7 Conclusion **51**

8 Repositories **53**
8.1 Implementation 53
8.2 References 53

9 References **54**

Abbreviations

AWGN	Additive White Gaussian Noise
BEC	Binary Erasure Channel
BER	Bit Error Rate
BLER	Block Error Rate
BPSK	Binary Phase-Shift Keying
BS	Batch Size
BSC	Binary Symmetric Channel
CEL	Cross Entropy Loss Function
CW/E	Codewords per Epoch
E	Epochs
F	Fixed
FNN	Feed Forward Neural Network
LDPC	Low Density Parity Check
LLR	Log-Likelihood Ratio
NND	Neural Network Decoder
NVS	Normalized Validation Score
OTF	On The Fly
PDF	Probability Density Function
RNN	Recurrent Neural Network
SNR	Signal-To-Noise Ratio
SPA	Sum-Product Algorithm, a variant of belief propagation
SPA NN	Sum-Product Algorithm Neural Network

Notations

$E_{i,j}$	short for E_{c_i,v_j}, a message sent from parity check node c_i to variable node v_j
$M_{j,i}$	short for E_{v_j,c_j}, a message sent from variable node v_j to parity check node c_i
$\oplus$	short for $\oplus_2$ denotes the addition using modulus 2
log	if not stated otherwise log refers to the base 2, log_2
$\otimes$	a vector-matrix multiplication
$\odot$	an element-wise matrix multiplication

1 Introduction

Errors occur in human communication as well as in communication between computers. Imagine talking to a friend next to a noisy street. You will not understand each other perfectly because of noise produced by cars. If just one word was overlain by noise, you can often guess this word or ask your friend to repeat what she just said. On networks, the most common way to check for errors due to noise is to build a checksum over all sent bits. This checksum is used for error detection, and if errors have occurred, a retransmission is initiated.[cf. GRS19, p. 20] This is like asking your friend to repeat her words. But there are communication applications where re-transmission is not an option (e.g. TV signals, live streams, data on a Blu-ray). How to cope with this problem? The first thing that might come to ones mind is to just repeat instead of waiting to be asked to repeat. Adding redundancy to fight errors caused by noise is called *encoding*. *Decoding* on the other hand denotes the process of removing errors using redundancy. Redundancy has a major disadvantage: it leads to longer messages to be sent. Therefore the goal is to correct as many errors as possible using as little redundancy as possible.

The focus of this thesis is on *LDPC-Codes*, forward error correction codes that are used to control errors in data transmission over unreliable or noisy communication channels. Codes like the Huffman code (used for lossless data compression) or line codes like the Manchester code are not discussed. Hereinafter, the combination of LDPC codes and machine learning to further improve the performance of already existing codes and their decoding algorithm is explored. The thesis starts with a brief overview of coding theory. Basic examples for codes, different channel models to simulate error patterns, and performance metrics are introduced.

Next, LDPC codes and two of their message passing decoding algorithms are presented. The Bit-Flipping algorithm is described to get a general idea of how message passing based decoding works. Afterwards, the Sum-Product algorithm is explained, building on the knowledge of the Bit-Flipping algorithm. In the next step it is shown how the Sum-Product algorithm can be converted into a neural network. Corrections and improvements are provided in the layers, metrics and experiment sections.[1] The implementation of the neural network is tested in several experiments in the last chapter. In the conclusion results of the experiments are used to verify statements made about the neural network decoder.

2 Coding Theory

Claude Shannon laid the foundation for methods of designing communication systems such that errors which occur in transmission can be reduced to arbitrarily small probability - the methods of designing them are known as coding theory.[cf. Yat09, p.3] Coding theory is the study of the properties of codes and their respective fitness for specific applications. Therefore coding theory is concerned with finding effective solutions to the question: *"How much redundancy is needed to correct a given amount of errors?"*[cf. GRS19, p. 21] Yehuda Lindell rephrases this as one of two coding theory goals:

1. Construct codes that correct a maximal number of errors while using minimal redundancy
2. Construct codes (as above) with efficient encoding and decoding procedures. [Lin10, p. 1]

Intuitively using a code with higher redundancy should tolerate more errors, so maximizing error correction while minimizing redundancy equals following two contradictory goals. The checksum approach, that is used when retransmission is an option, can be formalised as *parity code*. The last bit of such a codeword is equal to the sum of all bits in the message.

[1] See subsection 4.3, subsection 6.1, and subsection 6.7.

Definition 2.1: Parity Code $C_{\oplus}$

Given a binary message m of length k where $m = (m_1, m_2, ..., m_k) \in \{0,1\}^k$, its corresponding codeword w is calculated as follows:

$$w = C_{\oplus}(m) = (m_1, m_2, ..., m_k, m_1 \oplus m_2 \oplus ... \oplus m_k)$$

where $\oplus$ denotes the addition using modulus 2.

The parity code $C_{\oplus}$ uses the minimum amount of non-zero redundancy.[cf. GRS19, p. 22] When a simple *repetition code* is used every bit of the message will be repeated a fixed number of times.

Definition 2.2: Repetition Code $C_{t,rep}$

Every bit of the message m of length k where $m = (m_1, m_2, ..., m_k) \in \{0,1\}^k$ is repeated t times to create the codeword w:

$$w = C_{t,rep}(m) = (m_{1,1}, m_{1,2}, ..., m_{1,t}, m_{2,1}, m_{2,2}, ..., m_{2,t}, ..., m_{k,1}, m_{k,2}, ..., m_{k,t}).$$

It is necessary to measure the trade-off between redundancy and the number of errors a code can correct. *Redundancy* is described by the rate R, the ratio of the length of the message k and the length of the codeword n.

Definition 2.3: Rate R of a Code C [2]

The rate R of a code C with dimension k (length of the message) and block length n (length of codeword with redundancy) is calculated using k and n where $k \leq n$ and $R \leq 1$ as follows:

$$R = \frac{k}{n}.$$

A high rate R indicates a low redundancy as message and codeword have nearly the same length. Parity codes $C_{\oplus}$ have a high rate as only one parity bit is added, $n = k + 1$. Repetition codes $C_{t,rep}$ have a more balanced rate as $n = k \cdot t$. Hence repetition codes should be able to correct more errors than parity codes. A closer look at error correction shows that this is the case.

Definition 2.4: Error Correction[3]

Let C be a code and let $g \in \mathbb{N} \cup \{0\}$. C is said to be g-error-correcting if there exists a decoding function D such that for every codeword $w \in C$ and error pattern e with at most g errors the codeword can be restored: $D(w + e) = w$.

A short example should give an intuition for error correction for a simple repetition code and a parity code.

[2] cf. [GRS19, p. 22]
[3] cf. [GRS19, p. 23]

Example 2.1: Error Correction with Repetition and Parity Codes

Let $C_{3,rep}$ be a binary code with a message length $k = 4$, a codeword length $n = 12$ and a therefore a rate $R = \frac{1}{3}$. Using $C_{3,rep}$ every bit of the message m is repeated three times to generate the codeword w. To decode w, it is divided into blocks of 3 bit size. For every block the majority bit is the message bit. If we look at a block like 101, the majority bit would be 1. As we claimed that the error correction has to work for any error pattern only restrained by the number of errors, $C_{3,rep}$ can only correct one error. That's because if two or more errors would occur in the same block the majority bit would have the erroneous value.
Let $C_{\oplus}$ be a binary code with message length $k = 4$, codeword length $n = 5$ and rate $R = \frac{4}{5}$. It's impossible for $C_{\oplus}$ to correct any errors because the position of the erroneous bit in the codeword is not determinable. Also only an odd number of incorrect bits can be detected because summing up $c_1 \oplus c_2 \oplus c_3 \oplus c_4 \oplus c_5$ results in 1 instead of 0.

If a code can correct g errors, it can of course also detect these errors. This does not work the other way round, making error detection weaker than error correction. If there is the possibility to ask for a retransmission, detecting errors is sufficient. The further focus of this work is on error correction.
The reason to limit the ability to correct errors in the example is based on the argument, that the error positions are located arbitrarily, bound only on the total number of errors. It is possible to model the occurrence of errors, called noise, differently. Those models are known as *channels*.

2.1 Channels

Noise, periods of silence and other forms of signal corruption often degrade the quality of the signal. Richard Hamming studied a model assuming that any error pattern can occur during transmission as long as the total number of errors is bounded.[cf. GRS19, p. 25] By allowing every error pattern, the location of the error is arbitrary and the nature of error in binary codes will be a bit flip making the atomic unit of error a symbol from the used alphabet.[cf. GRS19, p. 25] One could also think about a channel where no more than one error can happen in any contiguous block of three bits. That kind of noise would be untroublesome to correct by the $C_{3,rep}$ code from Example 2.1. With this example in mind, it is particularly clear how the error-correcting capabilities of the code and the decoding function crucially depend on the noise model. Since the channel is of such importance, it is hardly surprising that codes are tailored to specific channels. Modelling the noise correctly is essential for real life applications.[cf. GRS19, p. 25] The following channel models are commonly used.

2.1.1 Binary Symmetric Channel (BSC)

The binary symmetric channel models the noise as a stochastic process. Claude Shannon first studied the binary symmetric channel with *crossover probability* $0 \leq p \leq 1$.[cf. GRS19, p. 25]

Definition 2.5: Binary Symmetric Channel (BSC)

The BSC has binary input $(0,1)$ and binary output $(0,1)$. With a crossover probability p, where $0 \leq p \leq 1$, the input bit is flipped.

This model proposed by Shannon is quite the opposite from the model proposed by Hamming: Hamming's model assumes **no** knowledge about the channel (except that a bound on the total number of

errors is known) while Shannon's model assumes **complete** knowledge about how noise is produced.[cf. GRS19, p. 25] In real life, the situation is often somewhere in between those two extremes.

2.1.2 Additive White Gaussian Noise (AWGN)

Additive white Gaussian noise, also known as the Gaussian channel, is used to model noise continuous instead of the discrete way of the BSC and BEC. It is the most popular model of real-input and real-output.

Definition 2.6: Additive White Gaussian Noise (AWGN)[4]

The AWGN channel has a real input X and a real output Y. The output Y is calculated by adding Gaussian distributed, independent noise $Z \sim \mathcal{N}(0, \sigma^2)$:

$$Y = X + Z.$$

Before using the AWGN channel, bits are usually transformed. A simple method for the transformation is *binary phase-shift keying (BPSK)*.

Definition 2.7: Binary Phase-Shift Keying (BPSK)

Binary phase-shift keying maps a bit to another value as follows.

$$BPSK(\text{bit}) = \begin{cases} 1, & \text{if bit } = 0 \\ -1, & \text{if bit } = 1. \end{cases}$$

To get the value of a bit mapped with BPSK back, a *hard decision* is made. In chapter 3.3.2 within the decoding algorithm, the hard decision is also used to determine a bit value.

Definition 2.8: Hard Decision

A hard decision HD is made to recover a bit from a non-binary value as follows:

$$HD(\text{value}) = \begin{cases} 0, & \text{if value } > 0 \\ 1, & \text{if value } < 0. \end{cases}$$

2.2 Performance metrics: Bit-Error Rate (BER), Block Error Rate (BLER) and Signal-to-Noise Ratio (SNR)

To measure the performance of a code on a specific channel, the bit-error rate (BER) as well as the signal-to-noise ratio (SNR) are widely used as metrics. While the SNR judges the quality of a channel the BER is used to measure the decoding quality.

2.2.1 Signal-to-noise ratio (SNR)

The signal-to-noise ratio expresses the ratio of the desired signal to the level of background noise. SNR therefore describes the degree by which signals are distorted over the channel.

[4] [cf. Mac03, p. 177]

Definition 2.9: Signal-to-noise ratio (SNR)

Let $P_{\text{signal}}, P_{\text{noise}} \in \mathbb{R}$ be average powers of the signal and the noise measured at same or equivalent points in a system and within the same bandwidth, then

$$\text{SNR} = \frac{P_{\text{signal}}}{P_{\text{noise}}}.$$

As P_{signal} is often way bigger than P_{noise} SNR is often shown on a logarithmic scale using dB:

$$\text{SNR}_{\text{dB}} = 10 \cdot log_{10}\left(\frac{P_{\text{signal}}}{P_{\text{noise}}}\right) dB.$$

The higher the SNR, the stronger the signal compared to the noise is, e.g. a 3 dB gain in SNR_{dB} is equal to $SNR = 2$ meaning the signal being two times stronger than the noise, while a 3dB loss would be equal to $SNR = \frac{1}{2}$. Dealing with a low SNR is equal to the noise having a high power which raises the probability of errors in the received message. In this thesis and its experiments SNR_{dB} is used.

Example 2.2: Calculate SNR for BPSK over AWGN

In this example the SNR for a particular modulation scheme, the binary phase-shifting keying (BPSK) is calculated. The signal power is the mean of square of symbols, assuming that each symbol is equally likely. Using BPSK the bits are mapped to 1 (for a 0 bit) and to -1 (for a 1 bit)

$$P_{\text{signal}} = \frac{(-1)^2 + (+1)^2}{2} = 1.$$

The noise power for BPSK is defined as the variance of the noise σ^2_{noise}. Using the AWGN channel the variance of the noise can be chosen freely to generate Gaussian noise of any variance. Whatever variance is used in the AWGN channel will also be used to calculate the SNR. Putting those two values into the formula leads to

$$\text{SNR} = \frac{P_{\text{signal}}}{P_{\text{noise}}} = \frac{1}{\sigma^2_{\text{noise}}}.$$

2.2.2 Bit-Error Rate (BER)

The bit-error rate describes the probability of a bit being decoded incorrectly. It therefore measures the strength of the used error correction code.

Definition 2.10: Bit-Error Rate (BER)

Let w_i be a bit of the codeword w sent and let $\hat{w}_i$ be the decoded bit. BER is the probability of $\hat{w}_i$ not matching the sent w_i.

$$\text{BER} = \mathbb{P}(w_i \neq \hat{w}_i)$$

The BER is approximated using a Monte-Carlos simulation where B is a large number of transmit-

ted bits and b_e is the fraction of incorrectly decoded bits.

$$\text{BER} \approx \frac{b_e}{B}$$

SNR and BER are linked: as SNR increases, BER is going to fall. When measuring the performance of a code, questions like "How is BER related to SNR" and "How does the BER vary with respect to the SNR?" are often raised. Using error correcting codes enables the same BER at a lower SNR.

2.2.3 Block Error Rate (BLER)

The block error rate (BLER) counts the fraction of blocks that are wrongly decoded while the block size can be chosen arbitrarily depending on the area of application. In the scenario of decoding codewords it seems reasonable to set the size of one block to be equal to the length of one codeword. In this case the BLER measures the ratio of wrongly decoded messages.

Definition 2.11: Block Error Rate (BLER)

Let one block correspond to one decoded codeword $\hat{w}$ of length n. Let there be $B \in \mathbb{N}$ blocks. BLER is the fraction of blocks that are wrongly decoded:

$$\text{BLER}(B) = \frac{\sum_{b=1}^{B} \mathbb{I}_{\hat{w}_b \neq w_b}}{B}$$

where $\mathbb{I}_{\hat{w}_b \neq w_b}$ is an indicator variable evaluating to one if a decoded codeword $\hat{w}$ is not equal to the corresponding sent codeword w.

3 Low-Density Parity-Check Codes (LDPC-Codes)

Before discussing LDPC codes in more detail, it is necessary to clarify the central concepts of linear codes and their creation such as generator matrix and parity check matrix.

3.1 Linear Codes Basics

In chapter 2 the idea of adding bits to the message to enable error correction was introduced. Instead of repeating one bit several times like the repetition code $C_{t,rep}$ (see Definition 2.2) LDPC-codes use *parity bits* p_i. They are calculated by summing certain message bits.

Definition 3.1: Parity Bit

A parity bit p_i is the sum message bits $m = (m_1, m_2, ..., m_k) \in \{0,1\}^k$ selected by the used code C. Let $I_i \subseteq \{1, ..., k\}$ be a set of indices of message bits that are used to calculate p_i:

$$p_i = \sum_{j \in I_i} m_j.$$

Adding parity bits does not encrypt the message, because the message can be read directly from the codeword, if the indices of the message bits are known. To control the received code word, the parity bit is recalculated by adding the same message bits again. If this parity bit does not correspond to the received parity bit, at least one of the message bits involved in the sum or the received parity bit has been changed in the channel. There is always the probability that a codeword has been alienated by the channel in such a way that the recalculated parity bit matches the received one even though an error occurred. If the recalculated parity bit does not match the received one, an error is sure, whereas a match of received and recalculated bit cannot completely exclude an error.

Example 3.1: Parity bit and parity check

Let $m = (m_1, m_2, m_3) = (1,1,1)$ be a message and let $w = (m_1, m_2, m_3, p_4, p_5) = (1,1,1,1,0)$ be the corresponding codeword. Let the two parity bits be defined as:

$$p_4 = m_1 \oplus_2 m_2 \oplus_2 m_3 \qquad\qquad p_5 = m_1 \oplus_2 m_3$$

Let $r = (0,1,1,1,0)$ be the received codeword. To check for errors one needs to recalculate the parity bits p_4' and p_5'.

$$\begin{aligned} p_4' &= m_1 \oplus_2 m_2 \oplus_2 m_3 = 0 \oplus_2 1 \oplus_2 1 = 0 \neq p_4 = 1 \\ p_5' &= m_1 \oplus_2 m_3 = 0 \oplus_2 1 = 1 \neq p_5 = 0 \end{aligned}$$

The received parity bits are not equal to the recalculated ones indicating a transmission error.

A codeword normally does have more than one parity bit. To calculate all of the parity bits and combine them with the message bits to create the codeword, a generator matrix G is used. By multiplying the message with G, parity bits are calculated and added for the whole message, encoding it.

Definition 3.2: Generator matrix G [5]

Let G_C be the $k \times n$ generator matrix of a linear $[n,k]$ code C, where n is the length of a codeword and k is the number of message bits. G is formed out of k codewords $g_1,...,g_k$:

$$G_C = \begin{pmatrix} g_1 \\ g_2 \\ \cdots \\ g_k \end{pmatrix}.$$

The rows of the generator matrix form a basis for C, therefore $g_1,...,g_k \in C$ are k linear independent basis pairs. The canonical form for a generator matrix is

$$G = [I_k, A],$$

where I_k is a $k \times k$ identity matrix, copying the message bits, and A is a $k \times (n-k)$ matrix generating the parity bits. A generator matrix generates a unique codeword for every message $m = (m_1, m_2, ..., m_k) \in \{0,1\}^k$ by multiplying the message with the generator matrix, where $\otimes$ indicates a vector-matrix multiplication:

$$m \otimes G_C = w \in C.$$

Conveniently, there is also a matrix-based solution for checking the received codeword. A parity check matrix H applies the control not only for one parity bit but for the whole received codeword. To be more precise H checks if $r \in C$.

Definition 3.3: Parity Check Matrix H [6]

The canonical form of a parity check matrix H is,

$$H = [A, I_{n-k}]$$

where A recalculates the parity bits and I_{n-k} is a $(n-k) \times (n-k)$ identity matrix, copying the received parity bits. One row in H therefore adds the received parity bit to the recalculated parity bit resulting in 0 if both bits have the same value. Let r be the received codeword and let H be a parity check matrix for a linear code C. A received codeword has **no identifiable errors** if multiplying it with H results in the all zero vector using $\otimes$ to denote a vector-matrix multiplication:

$$r \in C \Leftrightarrow H \otimes r = \overrightarrow{0}.$$

If $H \cdot r$ results in at least one 1 in the result vector at least one bit was corrupted during transmission.[7] As G and H perform on the same code, it is no surprise that they have a connection to each other and G can be used to create H and vice versa. Theorem 3.1 only works, if the source matrix is in canonical form, which can be achieved by using *Gaußian Elimination*.

[5] [cf. Het18, definition 5.3.10, p. 89-90]
[6] [cf. Het18, definition 5.3.20, p.93]
[7] The all zero vector $\overrightarrow{0}$ as result does not necessarily mean that no error has occurred.

Theorem 3.1: From generator matrix G to parity check matrix H and back[8]

Let H be a standard form parity check matrix and let G be a standard form generator matrix, both of codes C. One matrix can be transformed into the other using the following relationship:

$$H = [A, I_{n-k}] \Leftrightarrow G = [I_k, A^T].$$

The parity check matrix H can not only detect errors, it can also be used for decoding. As Berlekamp, McEliece, and van Tilborg have shown in 1978, decoding a received codeword is a NP-hard problem, making decoding way more complex than encoding.[BMv78] Therefore the most sensible approach is to first create the control matrix H in such a way that it makes decoding efficient, as easy as possible and then use the connection between control matrix and generator matrix to form G out of H. LDPC-codes are designed with this idea in mind.

3.2 Generate LDPC Codes

LDPC-Codes and the first efficient (approximative) algorithms to decode them have been introduced in 1962 by *Robert Gray Gallager* in his PhD thesis at MIT.[vgl. Joh, S.3][9] The defining idea for LDPC codes is to interpret the parity check matrix as the adjacency matrix of a bipartite graph, also known as *Tanner graph* and exchange messages along the vertices to decode the received codeword. Therefore the nodes are split in a set of vertices representing the codeword bits (variable nodes) and a set of vertices representing the parity check bits (check nodes).

Definition 3.4: Tanner graph[10]

A **Tanner graph** is a **bipartite graph** using the parity check matrix H of a linear code as adjacency matrix. The graph consists of two sets of vertices:

n vertices for the bits of the codeword, *variable nodes* $v_1, ...v_n$
g vertices for the parity check equations, *check nodes* $c_1, ...c_g$.

An edge joins a variable node v to a check node c if the bit represented by v is included in the corresponding parity-check equation.

The check matrices introduces by Gallager were *regular*, which means that the number of ones in each row a_r and the number of ones in each column a_c had to be the same.[cf. Joh, p. 20] However it is not necessary that a_r is equal to a_c to form a LDPC code.
Low-Density Parity-Check codes are named after their sparse parity check matrix H. In comparison to zeros in H the number of ones is stunningly low. The matrix H is called sparse if the number of ones in a row divided by the length of the codeword $\frac{a_r}{n}$ approaches zero.[cf. Mac03, p. 557] MacKay gives an example of a 10 000 $\times$ 20 000 parity check matrix containing only 6 ones per row and 3 ones per column.[cf. Mac03, p. 563] One of the 20 000 message bits would therefore be part of only 3 parity check equations. And one of the 10 000 parity check equations determines the value of the parity check

[8] [cf. Joh, p. 15]
[9] LDPC-Codes are sometimes called *Gallager Codes*, after their inventor Gallager.[cf. Mac03, p. 557]
[10] [cf. Joh, p. 12]

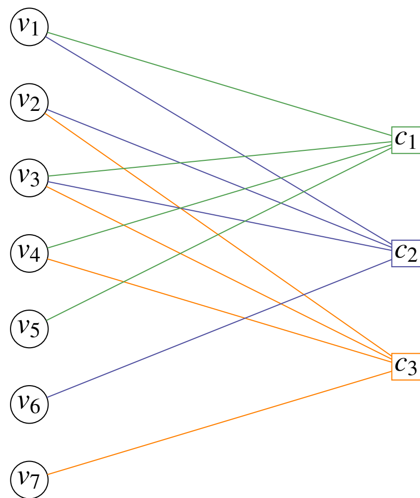
bit by summing up exactly 6 message bits. Using a sparse parity check matrix reduces the complexity of the corresponding Tanner graph by setting the number of edges to a constant level. The relationship between the control matrix and the corresponding tanner graph can easily be seen from an example.

Example 3.2: Tanner graph from parity-check matrix H

Let H be the parity-check matrix of a linear block code with length of a codeword $n = 7$ and $g = 3$ parity check equations.

$$H = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

This will lead to 7 variable nodes and 3 parity check nodes in the Tanner graph. The number of ones in H is equal to the number of edges in the corresponding graph.



Using a Tanner graph does not only help to visualize the dependencies of message bits and parity check bits. It is also the basis for applying more complex decoding algorithms, so called message passing decoding algorithms.

3.3 Decoding LDPC Codes - Message Passing Decoding

LDPC codes are used in combination with message passing decoding algorithms that operate on the Tanner graph by exchanging information between the variable nodes and the parity check nodes. This exchange enables the use of knowledge about the codeword contained in the whole graph for encoding. How exchanging messages works and how the messages to be sent are calculated depends on the used algorithm. What the algorithms have in common is a low number of maximum iterations $T_{\text{max}} < 20$ to decode fast.[cf. Mac03, p. 563] The bit-flipping algorithm is introduced to get a feeling for message passing on Tanner graphs. Next up is the sum-product algorithm (SPA), the, as MacKay states, best algorithm for decoding LDPC codes.[cf. Mac03, p. 557] SPA will also be the starting point for combining decoding and neural networks. Unless the Tanner graph is a tree, the result of the bit-flipping algorithm as well as from SPA offer only an approximative solution.[cf. Sch18, p. 285 ff.] Circles in the graphs of

LDPC codes can happen and especially short circles are a problem as they create unwanted dependencies. Only if the algorithm ends before the shortest circle closes, the graph still behaves like a tree.

3.3.1 Bit-Flipping Algorithm

The bit-flipping algorithm operates with input from the binary symmetric channel (BSC). The position of flipped bits within the received word r is therefore not known. To recover the codeword $\hat{w}$ from r, messages E_{c_i,v_j} about the extrinsic information are sent from check node c_i to variable node v_j. Extrinsic information denotes the information the graph holds about a bit. A message E_{c_i,v_j} contains the binary value the j'th bit $bit(v_j)$ should have to satisfy the i'th parity check equation represented by check node c_i. A parity check equation corresponds to the calculation of one line within parity check matrix H.

Definition 3.5: Parity Check Equation $check(c_i)$

Let H be a parity check matrix and let c_i be the i'th parity check equation represented by the i'th row in H. A parity check $check(c_i)$ can be calculated as follows:

$$check(c_i) = \sum_{j=1}^{n} H_{i,j} \otimes r_j = \sum_{j \in adj(c_i)} bit(v_j).$$

Where $adj(c_i)$ returns the indices j of all variable nodes v_j adjacent to check node c_i.

The parity check equation is considered satisfied, if it's result is equal to zero. To calculate E_{c_i,v_j} the values of all variable nodes adjacent to c_i are summed up except for node v_j. So the addressee of the message is excluded from the calculation. The result of the calculation is exactly the value the parity equation is lacking to be equal to 0 and hence satisfied. The idea behind the bit-flipping algorithm is that v_j gathers all messages from adjacent check nodes E_{c_i,v_j} and executes a majority vote with the received messages, e.g. if the majority of the received messages are equal to 1, $bit(v_j)$ will be set to 1.

Algorithm 3.1: Bit-Flipping Algorithm [11]

Input

- $T_{\max}$, maximum number of iterations
- $r = (r_1, ..., r_n)$, a codeword received over the BSC

Algorithm

1. Initialize iteration counter $c_T = 0$
   Initialize Tanner graph: $\forall j \in \{1, ..., n\} : bit(v_j) = r_j$
2. Evaluate parity check equations for all parity check nodes where $bit(c_i) = check(c_i)$. If ...
   (a) $\forall i \in \{1, ..., g\} : bit(c_i) = 0$: decoding successful,
   return: $\hat{w} = (bit(v_1), ..., bit(v_n))$
   (b) maximum number of iterations is reached $c_T = T_{\max}$: stop.
   return: error.
   (c) $\exists i \in \{1, ..., g\} : bit(c_i) \neq 0$: continue.

3. Calculate messages E_{c_i,v_j}, send them from i'th check node to j'th variable node:

$$E_{c_i,v_j} = \sum_{n \in adj(c_i) \setminus j} bit(v_n).$$

4. Perform majority vote for all variable nodes v_j and set corresponding $bit(v_j)$:

$$bit(v_j) = \begin{cases} 1, & \text{if (number of } E_{adj(v_j),j} = 1) > \text{ (number of } E_{adj(v_j),j} = 0) \\ 0, & \text{if (number of } E_{adj(v_j),j} = 1) < \text{ (number of } E_{adj(v_j),j} = 0) \\ bit(v_j), & \text{if (number of } E_{adj(v_j),j} = 1) = \text{ (number of } E_{adj(v_j),j} = 0). \end{cases}$$

5. Increment counter $c_T = c_T + 1$ and jump to 2.

3.3.2 Sum-Product Algorithm (SPA)

While the bit-flipping algorithm uses bits, the sum-product algorithm uses a probabilistic approach and real values. It is about the probability of $bit(v_j)$ having the value 0 or 1. [cf. Joh, p. 31] Depending on the channel, the initial probabilities can either be calculated or are already given. It is common to specify probabilities in a so-called Log-Likelihood ratio (LLR).[cf. Joh, p. 31]

Definition 3.6: Log-Likelihood Ratio (LLR) [12]

Let $LLR(bit(v_j)|r_j)$ be the log-likelihood ratio describing a relative probability.

$$LLR(bit(v_j)|r_j) = \ln\left(\frac{\mathbb{P}(bit(v_j) = 0|r_j)}{\mathbb{P}(bit(v_j) = 1|r_j)}\right)$$

For AWGN channel the LLR can be defined in relation to the variance σ of the channel:

$$\text{LLR}_{\text{AWGN}}(bit(v_j)|r_j) = \frac{2r_j}{\sigma^2}.$$

For the BSC model the LLR can be defined in relation to the crossover probability p:

$$\text{LLR}_{\text{BSC}}(bit(v_j)|r_j) = \begin{cases} \ln\left(\frac{p}{1-p}\right), & r_j = 1 \\ \ln\left(\frac{1-p}{p}\right), & r_j = 0. \end{cases}$$

By applying logarithmic laws, the probability of a certain bit value can be derived from the LLR:

$$\mathbb{P}(bit(v_j) = 1) = \frac{e^{-LLR(bit(v_j))}}{1 + e^{-LLR(bit(v_j))}}$$

$$\mathbb{P}(bit(v_j) = 0) = \frac{e^{LLR(bit(v_j))}}{1 + e^{LLR(bit(v_j))}}.$$

[11] [vgl. Joh, S. 27]

To convert the probability back to a binary value, a hard decision (see Definition 2.8) is made, reducing a real value to a binary value by selecting the more probable binary value. **The goal of SPA is to determine the maximum a posteriori probability for every bit under the condition that all parity check equations are satisfied.**
The message passing works similar as in the bit-flipping algorithm, but the calculation of the message about the extrinsic information E_{c_i,v_j} is changed and a message to be sent from variable node v_j to check node c_i about the intrinsic information called O_{v_j,c_i} is added. How does E_{c_i,v_j} change? In the bit-flipping algorithm, e.g. $E_{c_i,v_j} = 1$ states, that the parity check equation $check(w_i)$ would be equal to zero and therefore satisfied if $bit(v_j) = E_{c_i,v_j} = 1$. The message from the example E_{c_i,v_j} is equal to one if an odd number of ones have been summed. When calculating E_{c_i,v_j} for SPA it is about determining the LLR of the probability that $bit(w_j)$ causes parity-check $check(c_i)$ to be satisfied. This is done by looking at the number of ones in the parity check as binomial distribution.

Definition 3.7: Probability to satisfy parity-check equation[13]

The probability to satisfy parity-check equation of check node c_i if the j'th bit is a one $\mathbb{P}^{(\text{ex})}_{c_i,v_j}(w_j = 1)$ is equal to the probability of an odd number of ones in a binomial distribution:

$$\mathbb{P}^{(\text{ex})}_{c_i,v_j}(w_j = 1) = \frac{1}{2} - \frac{1}{2} \cdot \prod_{t \in adj(c_i) \setminus j} \left(1 - 2 \cdot \mathbb{P}^{(\text{in})}(w_t = 1)\right)$$

where $adj(c_i)$ returns the indices j of all variable nodes v_j adjacent to check node c_i and $\mathbb{P}^{(\text{in})}$ is the probability v_j holds.
The probability to satisfy this parity-check equation if bit j is a zero $\mathbb{P}^{(\text{ex})}_{c_i,v_j}(w_j = 0)$ can be expressed in relation to $\mathbb{P}^{(\text{ex})}_{c_i,v_j}(w_j = 1)$ using complementary probability.

$$\begin{aligned} \mathbb{P}^{(\text{ex})}_{c_i,v_j}(w_j = 0) &= 1 - \mathbb{P}^{(\text{ex})}_{c_i,v_j}(w_j = 1) \\ &= \tfrac{1}{2} + \tfrac{1}{2} \cdot \prod_{t \in adj(c_i) \setminus j} \left(1 - 2 \cdot \mathbb{P}^{(\text{in})}(w_t = 1)\right) \end{aligned}$$

Looking at the probability for even or odd number of ones results in the probability for a parity-check equation to be satisfied given the sent bit w_j was a one and the complementary probability to do so for $w_j = 0$. Based on this two values the LLR can be applied to express E_{c_i,v_j}.

Definition 3.8: Message E_{c_i,v_j} ($E_{i,j}$)[14]

The message E_{c_i,v_j} sends extrinsic information from check node c_i to variable node v_j:

$$E_{c_i,v_j} = \ln\left(\frac{\mathbb{P}^{(\text{ex})}_{c_i,v_j}(w_j = 0)}{\mathbb{P}^{(\text{ex})}_{c_i,v_j}(w_j = 1)}\right).$$

[12][cf. Joh, p. 31], [cf. Agr17, p. 6]
[13][cf. Joh, p. 32]

Substitute the extrinsic probabilities with Definition 3.7 as follows:

$$E_{c_i,v_j} = \ln\left(\frac{\frac{1}{2}+\frac{1}{2}\cdot\prod_{t\in adj(c_i)\setminus j}(1-2\cdot\mathbb{P}^{(\text{in})}(w_t=1))}{\frac{1}{2}-\frac{1}{2}\cdot\prod_{t\in adj(c_i)\setminus j}(1-2\cdot\mathbb{P}^{(\text{in})}(w_t=1))}\right).$$

Using the relationship $2\cdot tanh^{-1}(p) = ln\left(\frac{1+p}{1-p}\right)$ one can express E_{c_i,v_j} in an even shorter way:

$$E_{c_i,v_j} = 2\cdot tanh^{-1}\left(\prod_{t\in adj(c_i)\setminus j}(1-2\cdot\mathbb{P}^{(\text{in})}(w_t=1))\right).$$

Messages sent from the variable nodes to the check nodes are called O_{v_j,c_i}. To avoid sending redundant information, the message from the j-th variable node to the i-th check node excludes the message E_{c_i,v_j} received from c_i.

Definition 3.9: Message O_{v_j,c_i} ($O_{j,i}$)[15]

The message O_{v_j,c_i} sends the intrinsic probability of the j-th bit having value one as LLR from variable node v_j to check node c_i. Excluded is the summand E_{c_i,v_j} from the check node c_i the message is sent to. This missing summand does not change the fact that it is still a LLR.

$$O_{v_j,c_i} = LLR(\mathbb{P}^{\text{in}}_{v_j,c_i}) = \ln\left(\frac{\mathbb{P}^{\text{in}}_{v_j,c_i}(w_j=0)}{\mathbb{P}^{\text{in}}_{v_j,c_i}(w_j=1)}\right) = r_j + \sum_{t\in adj(v_j)\setminus i} E_{c_t,v_j}$$

where $adj(v_j)$ returns the indices i of all check nodes c_i adjacent to variable node v_j and r_j denotes the information received from the channel for the j-th bit of the codeword.
O_{v_j,c_i} can be expressed as:

$$O_{v_j,c_i} = tanh\left(\frac{1}{2}\cdot\left(r_j + \sum_{t\in adj(v_j)\setminus i} E_{c_t,v_j}\right)\right).$$

Since messages are an essential part of SPA, it was necessary to look at them closely to understand the algorithm. Now all parts can be put together using input from the BSC. When the BSC is used as channel model it's crossover probability p is known and needed for the initialization of the a priori probabilities. If p is not known an estimate for p can be sufficient.

Algorithm 3.2: Sum-Product Algorithm (SPA) using BSC [16]

Input

- $T_{\max}$, maximum number of iterations
- $r = (r_1, ..., r_n)$, a codeword received over the BSC

[14] [cf. Joh, p. 32]
[15] [cf. Joh, p. 32 f.] and [cf. Agr17, p. 27]

- p, the (estimated) error probability of the BSC

Algorithm

1. Determine the a priori probability using LLR for every bit $r' = (r'_1, ..., r'_n)$.
2. Initialize iteration counter $c_T = 0$
   Initialize Tanner graph using a priori probabilities: $\forall j \in \{1, ..., n\}$:
   $bit(v_j) = HD(r'_j)$ where HD is the hard decision (see Definition 2.8)
   $priori(v_j) = r'_j$
   $value(v_j) = 0$ where $value(v_j)$ holds the LLR of the current iteration.
3. Use $bit(v_j)$ to apply the parity check equation where $bit(c_i) = check(c_i)$. If ...
   (a) $\forall i \in \{1, ..., g\} : check(c_i) = 0$: decoding successful,
   return: $\hat{w} = (bit(v_1), ..., bit(v_n))$
   (b) $\exists i \in \{1, ..., g\} : check(c_i) \neq 0$
      i. if maximum number of iterations is reached $cT = T_{\max}$: stop,
      return: error and approximation so far $\hat{w} = (bit(v_1), ..., bit(v_n))$
      ii. else: contine
4. Calculate E_{c_i,v_j} for all nodes and send them (see Definition 3.8).

$$E_{c_i,v_j} = \ln \left(\frac{\frac{1}{2} + \frac{1}{2} \cdot \prod_{t \in Adj(i) \setminus j} \left(1 - 2 \cdot \frac{e^{-value(v_t)}}{1+e^{-value(v_t)}} \right)}{\frac{1}{2} - \frac{1}{2} \cdot \prod_{t \in Adj(i) \setminus j} \left(1 - 2 \cdot \frac{e^{-value(v_t)}}{1+e^{-value(v_t)}} \right)} \right)$$

5. Update $value(v_j)$ by adding received messages E_{c_i,v_j} for every variable node v_j.

$$value(v_j) = priori(v_j) + \sum_{i \in Adj(j)} E_{c_i,v_j}$$

6. Update $bit(v_j) = HD(value(v_j))$
7. Calculate O_{v_j,c_i} for all nodes and send them (see Definition 3.9)

$$O_{v_j,c_i} = r_j + \sum_{t \in adj(v_j) \setminus i} E_{c_t,v_j}$$

8. Increment counter $c_T = c_T + 1$ and jump to **3**.

3.4 Runtime

Message passing decoding algorithms are set to an runtime upper limit by their number of iterations. In combination with the sparse parity check matrix this results into a runtime that is linear $\mathcal{O}(n)$ in the number of variable nodes $v_1, ..., v_n$.[cf. Mac03, p. 9] This is achieved because the number of parity check bits $p_1, ..., p_g$ is smaller than the number of bits in the codeword $g < n$. If g is upwards estimated with n, the runtime is still linear because in every iteration a variable node sends messages to a constant number of parity check nodes and a parity check node sends messages to a constant number of variable nodes

[16] [cf. Joh, p. 34]

leading to a runtime in $\mathscr{O}(a_r \cdot g + a_c \cdot n) = \mathscr{O}(n)$. This is a significant runtime improvement compared to an exponential runtime of Maximum Likelihood Decoding.[17] The only disadvantage is that the solution is an approximation.

[17]Maximum-Likelihood-Decoding is a brute-force decoding approach where the received codeword r is compared with all possible codewords $c \in C$ to find the codeword with shortest Hamming distance.[cf. Joh, p. 9]

4 Neural Networks to Advance LDPC-Codes

When thinking about how neural networks could improve decoding, a first approach might be to hand over the whole process of decoding to a neural network. To train this decoder pick a channel model, feed codewords into the channel to create training data, and start. But with this naive approach all knowledge about decoding would be useless. It would not be necessary to tailor a specific parity check matrix and transform it to get a generator matrix. A more advanced approach is to look at what already exists and find parts that benefit from optimization through neural networks. Transforming a Tanner graph into a neural network is just one thought away. Wouldn't it be great if problematic short circles wouldn't be an issue any more because the decoder is trained to mitigate them? In *Machine Intelligence in Decoding of Forward Error Correction Codes* Naveet Agrawal takes a close look at the approach of training the decoder to do a better job. In their paper *AI Coding: Learning to Construct Error Correction Codes* Huang, Zhang, Li, Ge and Wang choose a completely different approach. Their goal is to train a neural network to create codes that are optimal within their code family. The two approaches start at opposite sides of the process, Huang et al. optimize the beginning with the code generation while Agrawal improves the end with the optimization of the decoder. Although both approaches are appealing and also the combination of the approaches are interesting taking a look at all ideas would have gone beyond the scope of this thesis. Therefore the focus is on optimizing the decoder by transforming the SPA into a neural network decoder (NND).
Agrawal follows Nachmanis [NML+17] idea of optimizing the decoding of LPDC codes. Therefore the bipartite Tanner graph, basis for message-passing decoding algorithms like the Sum-Product algorithm (SPA), is transformed into a neural network in a few steps. The Tanner graph is based on the parity check matrix of the used LDPC code. Messages are passed along the edges of this graph. The first step is to unfold the Tanner graph by transforming it into a directed graph and adding a copy of the graph for each iteration of the message passing algorithm. Afterwards the edges of the unfolded Tanner graph are transformed into nodes to form a neural network.[cf. Agr17, p. 24]

4.1 Transforming the Tanner Graph

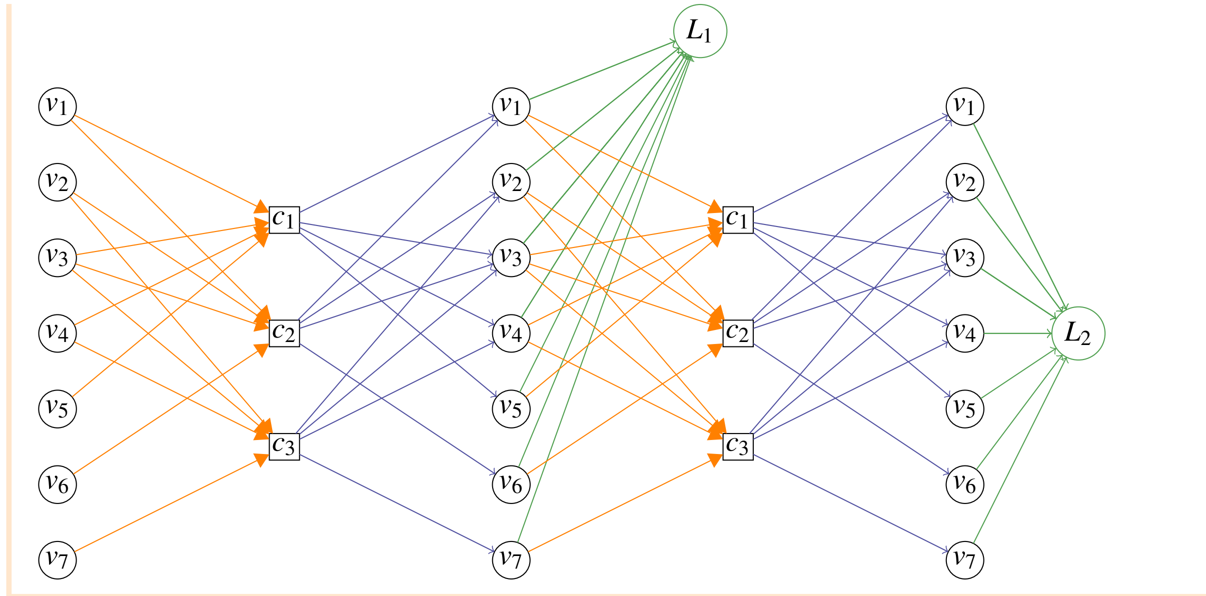
SPA has a maximum number of iterations $T_{\max}$, the associated neural network has $2 \cdot T_{\max}$ hidden layers, because each iteration involves two parts: sending messages E_{c_i,v_j} from the parity check nodes to the variable nodes and vice versa O_{v_j,c_i}. This process is unfolded in the network.

Example 4.1: Unfold Tanner graph

For the example the same parity-check matrix H as in Example 3.2 is used:

$$H = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

Edges are now directed as messages are passed on from variable node to check node only to the right. Messages passed from variable nodes to check nodes are marked in orange, messages passed from check nodes to variable nodes are marked blue. The end of an iteration L, the hard decision HD on the bit value of a variable node and the following check if all parity-checks are satisfied is marked green. The example unfolds the graph for two iterations of the message passing algorithm.



Transforming the unfolded Tanner graph to a neural network takes a few steps. The resulting network is called SPA-NN, short for *sum-product algorithm neural network*. In the next step the layers of the neural network are described.[18]

4.2 Layers

The input layer of the neural network has n nodes, one node for each bit in the received codeword r. This is equal to the first layer of nodes in the unfolded Tanner graph. The output layer of the SPA-NN also has n nodes, one node for each decoded bit $\hat{w}_j$ of the received codeword r. All hidden layers have the same number of nodes $|E|$, corresponding to the number of edges in the Tanner graph and equal to the number of ones in the parity check matrix H of the code. This is because the goal of SPA-NN is to train the weights of the edges in the unfolded Tanner graph. Therefore a node in a hidden layer is representing an **edge**.[Agr17, p. 23]
A node in the first hidden layer of the neural network represents an edge from a variable node v_j to a ceck node c_i in the unfolded Tanner graph is denoted like the message in the SPA as O_{v_j,c_i}. This is not only the case for the first hidden layer but also for each following odd hidden layer. Nodes in each even hidden layer represent an edge from a check node c_i to a variable node v_j. These nodes are labelled E_{c_i,v_j}, like the message sent in SPA. Connections between layers of the neural network are defined by four configuration matrices.[cf. Agr17, p. 26]

1. W_{in2odd} for input Layer to odd hidden layer
2. $W_{odd2even}$ for odd hidden layer to even hidden layer
3. $W_{even2odd}$ for even hidden layer to odd hidden layer
4. $W_{even2out}$ for even hidden layer to output layer

For the definitions of the configuration matrices note that E denotes the edges of the Tanner graph.

[18]The colors used in the following description match the colors used in the example.

Definition 4.1: Configuration matrix W_{in2odd}

An input-layer node r_j is connected to a node of the first hidden layer if its bit is used as variable node in an edge $(v_{j'}, c_i)$ from variable node $v_{j'}$ to check node c_i. This connection is described in the $n \times |E|$ matrix

$$W_{in2odd}(r_j, O_{(v_{j'}, c_i)}) = \begin{cases} 1, & j = j' \\ 0, & \text{otherwise} \end{cases},$$

with rows $r_j \in \{r_1, ..., r_n\}$ and columns $O_{(v_{j'}, c_i)}$ where $(v_{j'}, c_i) \in E$.

Definition 4.2: Configuration matrix $W_{odd2even}$

An odd hidden layer node $O_{(v_j, c_i)}$ is connected to an even hidden layer node $E_{(c'_i, v'_j)}$ if both edges (v_j, c_i) and (c'_i, v'_j) use the same check node $c_i = c'_i$ but a different variable node $v_j \neq v'_j$. This connection is described in the $|E| \times |E|$ matrix

$$W_{odd2even}(O_{(v_j, c_i)}, E_{(c'_i, v'_j)}) = \begin{cases} 1, & c_i = c'_i \text{ and } v_j \neq v'_j \\ 0, & \text{otherwise} \end{cases},$$

with rows $O_{(v_j, c_i)}$ where $(v_j, c_i) \in E$ and columns $E_{(c'_i, v'_j)}$ where $(c'_i, v'_j) \in E$.

Definition 4.3: Configuration matrix $W_{even2odd}$

An even hidden layer node $E_{(c_i, v_j)}$ is connected to an odd hidden layer node $O_{(v'_j, c'_i)}$ if both edges (c_i, v_j) and (v'_j, c'_i) use the same variable node $v_i = v'_i$ but a different check node $c_j \neq c'_j$. This connection is described in the $|E| \times |E|$ matrix

$$W_{even2odd}(E_{(c_i, v_j)}, O_{(v'_j, c'_i)}) = \begin{cases} 1, & v_j = v'_j \text{ and } c_i \neq c'_i \\ 0, & \text{otherwise} \end{cases},$$

with rows $E_{(c_i, v_j)}$ where $(c_i, v_j) \in E$ and columns $O_{(v'_j, c'_i)}$ where $(v'_j, c'_i) \in E$.

Definition 4.4: Configuration matrix $W_{even2out}$

An even hidden layer node $E_{(c_i, v_j)}$ is connected to an output layer node $\overline{w}_{j'}$ if the hidden layer node uses the variable node v_j corresponding to the bit $\overline{w}_{j'}$. This connection is described in the $|E| \times n$ matrix

$$W_{even2out}(E_{(c_i, v_j)}, \overline{w}_{j'}) = \begin{cases} 1, & j = j' \\ 0, & \text{otherwise} \end{cases},$$

with rows $E_{(c_i, v_j)}$ where $(c_i, v_j) \in E$ and columns $\overline{w_{j'}} \in \{\overline{w}_1, ..., \overline{w}_n\}$.

To get a feeling for the configuration matrices used, look at the following example.

Example 4.2: Neural Network based on Unfolded Tanner Graph

The same parity-check matrix H and the same (unfolded) Tanner graph as in Example 4.1 is used.

1. **Number of Layers in the SPA-NN**
   Based on two iterations of SPA, there will be one input layer and one output layer. As the number of iterations T is 2, the neural network will have $2 \cdot T = 2 \cdot 2 = 4$ hidden layers.

2. **Number of Nodes in the layers of SPA-NN**
   H indicates the number of nodes in the layers of the SPA-NN. Input and output layers have n nodes in a $k \times n$ matrix. In this example a 3×7 matrix is used, so these two layers have 7 nodes each. The nodes in each hidden-layer can be read from the number of ones in the matrix, because each one represents an edge in the tanner graph. This leads to 12 nodes for each hidden layer, also referred to as 12 hidden units.

3. **Generate configuration matrices**
   Use Definition 4.1 to create W_{in2odd} with nodes r_j from the input layer and nodes $O_{(v'_j,c'_i)}$ from the first hidden layer. For better readability $O_{(v_j,c_i)}$ will be shortened to $O_{j,i}$ and $E_{(c'_i,v'_j)}$ to $E_{(i,j)}$.

	$O_{1,1}$	$O_{3,1}$	$O_{4,1}$	$O_{5,1}$	$O_{1,2}$	$O_{2,2}$	$O_{3,2}$	$O_{6,2}$	$O_{2,3}$	$O_{3,3}$	$O_{4,3}$	$O_{7,3}$
r_1	1	0	0	0	1	0	0	0	0	0	0	0
r_2	0	0	0	0	0	1	0	0	1	0	0	0
r_3	0	1	0	0	0	0	1	0	0	1	0	0
r_4	0	0	1	0	0	0	0	0	0	0	1	0
r_5	0	0	0	1	0	0	0	0	0	0	0	0
r_6	0	0	0	0	0	0	0	1	0	0	0	0
r_7	0	0	0	0	0	0	0	0	0	0	0	1

   Use Definition 4.2 to create $W_{odd2even}$ with nodes $O_{(v_j,c_i)}$ from an odd hidden layer and nodes $O_{(v'_j,c'_i)}$ from an even hidden layer. If c_i is equal to c'_i it will be higlighted blue but only if v_j is **not** equal to v'_j a 1 is entered.

	$E_{1,1}$	$E_{1,3}$	$E_{1,4}$	$E_{1,5}$	$E_{2,1}$	$E_{2,2}$	$E_{2,3}$	$E_{2,6}$	$E_{3,2}$	$E_{3,3}$	$E_{3,4}$	$E_{3,7}$
$O_{1,1}$	0	1	1	1	0	0	0	0	0	0	0	0
$O_{3,1}$	1	0	1	1	0	0	0	0	0	0	0	0
$O_{4,1}$	1	1	0	1	0	0	0	0	0	0	0	0
$O_{5,1}$	1	1	1	0	0	0	0	0	0	0	0	0
$O_{1,2}$	0	0	0	0	0	1	1	1	0	0	0	0
$O_{2,2}$	0	0	0	0	1	0	1	1	0	0	0	0
$O_{3,2}$	0	0	0	0	1	1	0	1	0	0	0	0
$O_{6,2}$	0	0	0	0	1	1	1	0	0	0	0	0
$O_{2,3}$	0	0	0	0	0	0	0	0	0	1	1	1
$O_{3,3}$	0	0	0	0	0	0	0	0	1	0	1	1
$O_{4,3}$	0	0	0	0	0	0	0	0	1	1	0	1
$O_{7,3}$	0	0	0	0	0	0	0	0	1	1	1	0

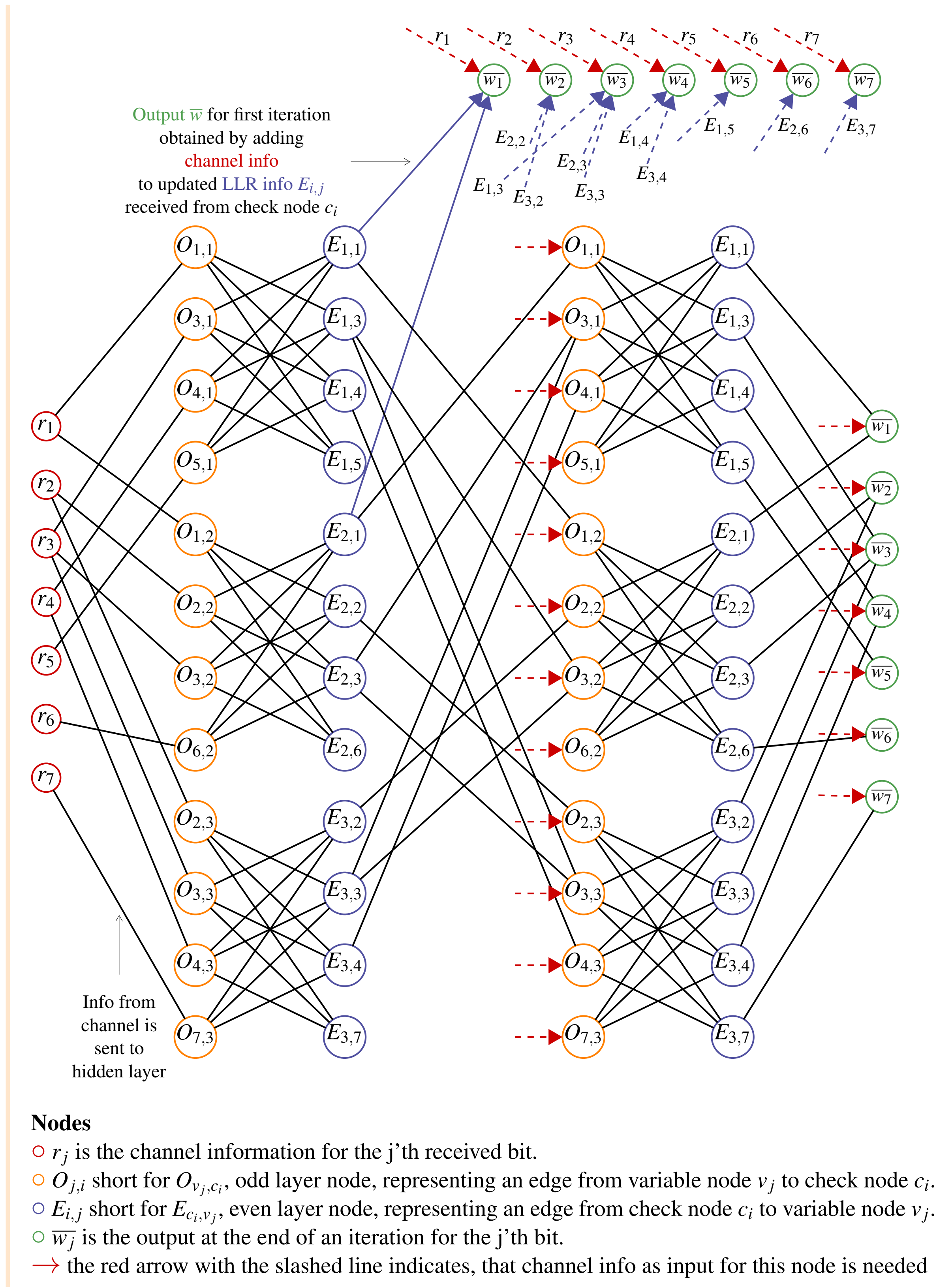
Use Definition 4.3 to create $W_{even2odd}$ with nodes $E_{(c_i,v_j)}$ from an even hidden layer and nodes $O_{(v'_j,c'_i)}$ from an odd hidden layer. First check if both edges two nodes represent use the same variable node $v_j = v'_j$. If this holds true it will be marked orange. Only of those two nodes also **do not** share the same check node in the edges they represent, a 1 will be entered.

	$O_{1,1}$	$O_{3,1}$	$O_{4,1}$	$O_{5,1}$	$O_{1,2}$	$O_{2,2}$	$O_{3,2}$	$O_{6,2}$	$O_{2,3}$	$O_{3,3}$	$O_{4,3}$	$O_{7,3}$
$E_{1,1}$	0	0	0	0	1	0	0	0	0	0	0	0
$E_{1,3}$	0	0	0	0	0	0	1	0	0	1	0	0
$E_{1,4}$	0	0	0	0	0	0	0	0	0	0	1	0
$E_{1,5}$	0	0	0	0	0	0	0	0	0	0	0	0
$E_{2,1}$	1	0	0	0	0	0	0	0	0	0	0	0
$E_{2,2}$	0	0	0	0	0	0	0	0	1	0	0	0
$E_{2,3}$	0	1	0	0	0	0	0	0	0	1	0	0
$E_{2,6}$	0	0	0	0	0	0	0	0	0	0	0	0
$E_{3,2}$	0	0	0	0	0	1	0	0	0	0	0	0
$E_{3,3}$	0	1	0	0	0	0	1	0	0	0	0	0
$E_{3,4}$	0	0	1	0	0	0	0	0	0	0	0	0
$E_{3,7}$	0	0	0	0	0	0	0	0	0	0	0	0

In the last column one can see how important it is that the odd hidden layer after the first hidden layer gets the channel info in addition to the information from the previous layer. In this case the node $O_{(v_7,c_3)}$ has no edge to its previous layer and would therefore not receive any information at all.

Use Definition 4.4 to create $W_{even2out}$ with nodes $E_{(c_i,v_j)}$ from the last even hidden layer and nodes $\overline{w_j}$ from the output layer.

	$\overline{w_1}$	$\overline{w_2}$	$\overline{w_3}$	$\overline{w_4}$	$\overline{w_5}$	$\overline{w_6}$	$\overline{w_7}$
$E_{1,1}$	1	0	0	0	0	0	0
$E_{1,3}$	0	0	1	0	0	0	0
$E_{1,4}$	0	0	0	1	0	0	0
$E_{1,5}$	0	0	0	0	1	0	0
$E_{2,1}$	1	0	0	0	0	0	0
$E_{2,2}$	0	1	0	0	0	0	0
$E_{2,3}$	0	0	1	0	0	0	0
$E_{2,6}$	0	0	0	0	0	1	0
$E_{3,2}$	0	1	0	0	0	0	0
$E_{3,3}$	0	0	1	0	0	0	0
$E_{3,4}$	0	0	0	1	0	0	0
$E_{3,7}$	0	0	0	0	0	0	1


Output $\overline{w}$ for first iteration
obtained by adding
channel info
to updated LLR info $E_{i,j}$
received from check node c_i
Info from
channel is
sent to
hidden layer
Nodes
○ r_j is the channel information for the j'th received bit.
○ $O_{j,i}$ short for O_{v_j,c_i}, odd layer node, representing an edge from variable node v_j to check node c_i.
○ $E_{i,j}$ short for E_{c_i,v_j}, even layer node, representing an edge from check node c_i to variable node v_j.
○ $\overline{w_j}$ is the output at the end of an iteration for the j'th bit.
→ the red arrow with the slashed line indicates, that channel info as input for this node is needed

4.3 From messages to activation functions

To complete the transformation from the Sum-Product algorithm to the NND an activation function is needed. The messages in the Tanner graph can be calculated using $tanh$ and $tanh^{-1}$ making them good candidates for activation functions. Therefore the equation calculated in the check node, must be split in two parts. Check node c_i sends messages E_{c_i,v_j} containing the extrinsic information about the bit value of the variable node v_j to this variable node. While v_j adds received E_{c_i,v_j} (see Definition 3.8) and sends the result known as intrinsic information O_{v_j,c_i} (see Definition 3.9). SPA does not use matrices to calculate the messages. A method is introduced to calculate the messages based on matrix and vector operations. The result is then combined with the activation function.

Algorithm 4.1: From Message O_{v_j,c_i} to activation function of odd layers[19]

The odd layer activation function is based on the calculation of O_{v_j,c_i}, see Definition 3.9.

$$O_{v_j,c_i} = tanh\left(\frac{1}{2}\cdot\left(r_j + \sum_{t\in adj(v_j)\setminus i} E_{c_t,v_j}\right)\right)$$

Calculation for odd layer output vector X_i of iteration T_i:

1. If the first hidden odd layer is calculated ($T_i = T_1$) then $T_{i-1} = T_0$ is set as zero vector because there is no initial information from the check nodes (no messages have been sent from check nodes c_i to variable nodes v_j yet)
2. Calculate X_i, the output vector of size $|E|$ of the i-th odd layer

$$X_i = tanh\left(\frac{1}{2}\cdot\left(W^T_{in2odd}\otimes R + W^T_{even2odd}\otimes X_{i-1}\right)\right)$$

Where $\otimes$ denotes vector-matrix multiplication, both configuration matrices are transformed, X_{i-1} is the output vector of the previous hidden even layer with size $|E|$ and
R a vector of size n with the input LLR channel information for each bit.

Algorithm 4.2: From Message E_{c_i,v_j} to activation function of even layers[20]

The calculation of an even layer activation function is based on the calculation of E_{c_i,v_j}, cf. Definition 3.8.

$$E_{c_i,v_j} = 2\cdot tanh^{-1}\left(\prod_{t\in adj(c_i)\setminus j}\left(1-2\cdot\mathbb{P}^{(\mathrm{in})}(w_t = 1)\right)\right)$$

Calculation for even layer T_i:

1. Form a $|E|\times|E|$ matrix M_{i-1} by repeating the odd layer output from layer T_{i-1} as a column $|E|$ times.
2. Calculate an auxiliary matrix $M^*_{i-1} = W_{odd2even}\odot M_{i-1}$, where $\odot$ is the element-wise multi-

[19] In Agrawals thesis there is a typing error in formula (3.5). [cf. Agr17, p. 27] The formula is missing the inner brackets that enclose this part: $\left(r_j + \sum_{t\in adj(v_j)\setminus i} E_{c_t,v_j}\right)$. The error has been corrected in for the formula used in this thesis.

plication

3. Replace zeros in auxiliary matrix M^*_{i-1} with ones
4. Calculate auxiliary vector X^*_{i-1} of size $|E|$ by multiplying along the column elements of M^*_{i-1}. The products in X^*_{i-1} correspond to $\prod_{t\in adj(c_i)\setminus j}(1-2\cdot\mathbb{P}^{(\text{in})}(w_t=1))$.
5. Calculate the output vector X_i of the even layer T_i using $tanh^{-1}$ as activation function.

$$X_i = 2\cdot tanh^{-1}\left(X^*_{i-1}\right)$$

The output previous to a hard decision is only adjusted in so far that the calculation is now also done by matrix and vector operations. The result is an output vector holding LLR for each bit. This LLR output can be obtained after each full iteration of SPA. In the NND it can be calculated at the same time as the output for an even layer.

Algorithm 4.3: From output to output layer[21]

The calculation of the output layer is based on the calculation of the LLR of one single bit.

$$\overline{w}_j = r_j + \sum_{t\in adj(v_j)} E_{c_t,v_j}$$

If an even layer output is calculated, also calculate vector $\overline{W_i}$ of size n, the estimated LLR of this iteration:

$$\overline{W_i} = R + W^T_{even2out}\otimes T_{i-1}$$

where
R is a vector of size n with input LLR channel information for each bit,
$W^T_{even2out}$ is the transposed configuration matrix describing edges from an even hidden layer to the output layer,
$\otimes$ denotes vector-matrix multiplication and
T_{i-1} is the output vector of the previous even hidden layer with size $|E|$.

In Algorithm 4.3 $W^T_{even2out}$ is used to select the bits from the previous layer that are adjacent to the variable node, sum their values and add them to the initial LLR from the channel row per row.

Example 4.3: From Message E_{c_i,v_j} to activation function of even layers

Following the steps of Algorithm 4.2 the activation function for the example based on the same parity check matrix H used in the last examples is calculated as follows.

1. The output row vector $(O_{1,1},O_{3,1},O_{4,1},O_{5,1},O_{1,2},O_{2,2},O_{3,2},O_{6,2},O_{2,3},O_{3,3},O_{4,3},O_{7,3})$ from the odd layer T_{i-1} is used to form matrix M_{i-1} by repeating it 12 times (because there are

[20] [cf. Agr17, p. 27]
[21] [cf. Agr17, p. 28]

$|E| = 12$ edges in the Tanner graph) as columns.

$$M_{i-1} = \begin{pmatrix}
O_{1,1} & O_{1,1} & O_{1,1} & O_{1,1} & O_{1,1} & O_{1,1} & O_{1,1} & O_{1,1} & O_{1,1} & O_{1,1} & O_{1,1} & O_{1,1} \\
O_{3,1} & O_{3,1} & O_{3,1} & O_{3,1} & O_{3,1} & O_{3,1} & O_{3,1} & O_{3,1} & O_{3,1} & O_{3,1} & O_{3,1} & O_{3,1} \\
O_{4,1} & O_{4,1} & O_{4,1} & O_{4,1} & O_{4,1} & O_{4,1} & O_{4,1} & O_{4,1} & O_{4,1} & O_{4,1} & O_{4,1} & O_{4,1} \\
O_{5,1} & O_{5,1} & O_{5,1} & O_{5,1} & O_{5,1} & O_{5,1} & O_{5,1} & O_{5,1} & O_{5,1} & O_{5,1} & O_{5,1} & O_{5,1} \\
O_{1,2} & O_{1,2} & O_{1,2} & O_{1,2} & O_{1,2} & O_{1,2} & O_{1,2} & O_{1,2} & O_{1,2} & O_{1,2} & O_{1,2} & O_{1,2} \\
O_{2,2} & O_{2,2} & O_{2,2} & O_{2,2} & O_{2,2} & O_{2,2} & O_{2,2} & O_{2,2} & O_{2,2} & O_{2,2} & O_{2,2} & O_{2,2} \\
O_{3,2} & O_{3,2} & O_{3,2} & O_{3,2} & O_{3,2} & O_{3,2} & O_{3,2} & O_{3,2} & O_{3,2} & O_{3,2} & O_{3,2} & O_{3,2} \\
O_{6,2} & O_{6,2} & O_{6,2} & O_{6,2} & O_{6,2} & O_{6,2} & O_{6,2} & O_{6,2} & O_{6,2} & O_{6,2} & O_{6,2} & O_{6,2} \\
O_{2,3} & O_{2,3} & O_{2,3} & O_{2,3} & O_{2,3} & O_{2,3} & O_{2,3} & O_{2,3} & O_{2,3} & O_{2,3} & O_{2,3} & O_{2,3} \\
O_{3,3} & O_{3,3} & O_{3,3} & O_{3,3} & O_{3,3} & O_{3,3} & O_{3,3} & O_{3,3} & O_{3,3} & O_{3,3} & O_{3,3} & O_{3,3} \\
O_{4,3} & O_{4,3} & O_{4,3} & O_{4,3} & O_{4,3} & O_{4,3} & O_{4,3} & O_{4,3} & O_{4,3} & O_{4,3} & O_{4,3} & O_{4,3} \\
O_{7,3} & O_{7,3} & O_{7,3} & O_{7,3} & O_{7,3} & O_{7,3} & O_{7,3} & O_{7,3} & O_{7,3} & O_{7,3} & O_{7,3} & O_{7,3}
\end{pmatrix}$$

2. Calculate an auxiliary matrix $M^*_{i-1} = W_{odd2even} \odot M_{i-1}$, where $\odot$ is the element-wise multiplication. M^*_{i-1} is equal to $W_{odd2even}$ with the output values of the previous layer instead of ones in it.

	$E_{1,1}$	$E_{1,3}$	$E_{1,4}$	$E_{1,5}$	$E_{2,1}$	$E_{2,2}$	$E_{2,3}$	$E_{2,6}$	$E_{3,2}$	$E_{3,3}$	$E_{3,4}$	$E_{3,7}$
$O_{1,1}$	0	$O_{1,1}$	$O_{1,1}$	$O_{1,1}$	0	0	0	0	0	0	0	0
$O_{3,1}$	$O_{3,1}$	0	$O_{3,1}$	$O_{3,1}$	0	0	0	0	0	0	0	0
$O_{4,1}$	$O_{4,1}$	$O_{4,1}$	0	$O_{4,1}$	0	0	0	0	0	0	0	0
$O_{5,1}$	$O_{5,1}$	$O_{5,1}$	$O_{5,1}$	0	0	0	0	0	0	0	0	0
$O_{1,2}$	0	0	0	0	0	$O_{1,2}$	$O_{1,2}$	$O_{1,2}$	0	0	0	0
$O_{2,2}$	0	0	0	0	$O_{2,2}$	0	$O_{2,2}$	$O_{2,2}$	0	0	0	0
$O_{3,2}$	0	0	0	0	$O_{3,2}$	$O_{3,2}$	0	$O_{3,2}$	0	0	0	0
$O_{6,2}$	0	0	0	0	$O_{6,2}$	$O_{6,2}$	$O_{6,2}$	0	0	0	0	0
$O_{2,3}$	0	0	0	0	0	0	0	0	0	$O_{2,3}$	$O_{2,3}$	$O_{2,3}$
$O_{3,3}$	0	0	0	0	0	0	0	0	$O_{3,3}$	0	$O_{3,3}$	$O_{3,3}$
$O_{4,3}$	0	0	0	0	0	0	0	0	$O_{4,3}$	$O_{4,3}$	0	$O_{4,3}$
$O_{7,3}$	0	0	0	0	0	0	0	0	$O_{7,3}$	$O_{7,3}$	$O_{7,3}$	0

3. Replace zeros in auxiliary matrix M^*_{i-1} with ones, leading to the following matrix.

$$M^*_{i-1} = \begin{pmatrix}
1 & O_{1,1} & O_{1,1} & O_{1,1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
O_{3,1} & 1 & O_{3,1} & O_{3,1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
O_{4,1} & O_{4,1} & 1 & O_{4,1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
O_{5,1} & O_{5,1} & O_{5,1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & O_{1,2} & O_{1,2} & O_{1,2} & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & O_{2,2} & 1 & O_{2,2} & O_{2,2} & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & O_{3,2} & O_{3,2} & 1 & O_{3,2} & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & O_{6,2} & O_{6,2} & O_{6,2} & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & O_{2,3} & O_{2,3} & O_{2,3} \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & O_{3,3} & 1 & O_{3,3} & O_{3,3} \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & O_{4,3} & O_{4,3} & 1 & O_{4,3} \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & O_{7,3} & O_{7,3} & O_{7,3} & 1
\end{pmatrix}$$

4. Calculate auxiliary vector X^*_{i-1} of size $|E|$ by multiplying along the column elements of M^*_{i-1}. The products in X^*_{i-1} correspond to the inner part of messages E_{c_i,v_j} sent from check nodes to variable nodes: $\prod_{t\in adj(c_i)\setminus j}(1-2\cdot\mathbb{P}^{(\text{in})}(w_t=1))$.

$$E_{c_i,v_j} = 2\cdot tanh^{-1}\left(\prod_{t\in adj(c_i)\setminus j}(1-2\cdot\mathbb{P}^{(\text{in})}(w_t=1))\right).$$

As the ones are the neutral element in a multiplication, they do not change the result of the multiplication. The correct calculation of the first entry of the vector X^*_{i-1} in this example is:

$$1\cdot O_{3,1}\cdot O_{4,1}\cdot O_{5,1}\cdot 1\cdot 1\cdot 1\cdot 1\cdot 1\cdot 1\cdot 1\cdot 1$$

For better readability the ones are omitted in the following calculation of X^*_{i-1}.

$$X^*_{i-1} = \begin{pmatrix} O_{3,1}\cdot O_{4,1}\cdot O_{5,1}, \\ O_{1,1}\cdot O_{4,1}\cdot O_{5,1}, \\ O_{1,1}\cdot O_{3,1}\cdot O_{5,1}, \\ O_{1,1}\cdot O_{3,1}\cdot O_{4,1}, \\ O_{2,2}\cdot O_{3,2}\cdot O_{6,2}, \\ O_{1,2}\cdot O_{3,2}\cdot O_{6,2}, \\ O_{1,2}\cdot O_{2,2}\cdot O_{6,2}, \\ O_{1,2}\cdot O_{2,2}\cdot O_{3,2}, \\ O_{3,3}\cdot O_{4,3}\cdot O_{7,3}, \\ O_{2,3}\cdot O_{4,3}\cdot O_{7,3}, \\ O_{2,3}\cdot O_{3,3}\cdot O_{7,3}, \\ O_{2,3}\cdot O_{3,3}\cdot O_{4,3}, \end{pmatrix}^T$$

5. Calculate the output vector X_i of the even layer T_i using $tanh^{-1}$ as activation function. By applying $2\cdot tanh^{-1}$ to the vector, the calculation resembles the well known calculation of E_{c_i,v_j} from Definition 3.8.

$$X_i = 2\cdot tanh^{-1}\left(X^*_{i-1}\right) = (E_{1,1},E_{1,3},E_{1,4},E_{1,5},E_{2,1},E_{2,2},E_{2,3},E_{2,6},E_{3,2},E_{3,3},E_{3,4},E_{3,7})$$

For the implementation of the SPA-NN this vector is transposed such that every output of every layer is always a vector (not a lying vector).

5 Implementation

This section describes findings and decisions made during the implementation. For the realization Pytorch 1.8.1+cu102, CUDA 10.2 and Python 3.7 was used. Pytorch was picked over Tensorflow as this approach demanded custom weights, weight masks as well as custom activation and loss functions which where easier to maintain and debug using Pytorch.

5.1 Activation Functions

The implementation of the activation function of the even layer is a special case because the weights for the connections between an odd and an even layer are not to be trained. This is not explained explicitly by [Agr17], he states that all weights **except** $W_{odd2even}$ are open for training[cf. Agr17, p. 28 f.] At first it seemed that the explanation for not training weights $W_{odd2even}$ was the even layers $tanh^{-1}(x)$ activation function.

Definition 5.1: $\tanh^{-1}$

The inverse hyperbolic tangent function $\tanh^{-1}$ is defined for $-1 < x < 1$ as

$$\tanh^{-1} = \frac{1}{2} \cdot \ln\left(\frac{1+x}{1-x}\right)$$

and its output range is $-\infty < tanh^{-1}(x) < +\infty$.

So $tanh^{-1}(x)$ is only defined for an input between minus one and one. As the output of the odd layer is produced by $\tanh(x)$ with a output value range between minus one and one this part is not a problem.

Definition 5.2: tanh

The hyperbolic tangent function $\tanh(x)$ is defined for $-\infty < x < +\infty$ as

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} = 1 - \frac{2}{e^{2x} + 1}$$

and its output range is $-1 < \tanh(x) < 1$.

As soon as the weights have values bigger than one or smaller than minus one the input for $tanh^{-1}$ is not valid any more. But it turned out that the reason for not training $W_{odd2even}$ lies in step 4 of Algorithm 4.2 where the output of the previous layer is multiplied by the weight an then multiplied with each other instead of added. By multiplying the derivative is made dependent on multiple weights.

The $tanh^{-1}$ activation function of the even layer also turned out to be a problem when one message E_{c_i,v_j} is equal to ± 1. As one can see in Definition 3.8 the input of $p = 1$ into $tanh^{-1}(p)$ will lead to infinity, propagating as `nan` values through the further Pytorch operations. To avoid this behaviour it would be possible to subtract the smallest possible ε from p or replace infinity values by a fixed value which is called *clipping*. Agrawal is aware that using $tanh^{-1}$ as activation function leads to exploding output values when the input values reach ± 1 and he pleads for clipping the function's output. [cf. Agr17, p. 16] He suggests clipping the output $(-E, E)$ from even layers to $10 \leq E \leq 20$. [cf. Agr17, p. 35] This

clipping approach was not suitable for Pytorch. Clipping $\pm\infty$ to 20 worked in the forward pass but an output of infinity broke Pytorchs autograd resulting in `nan` values during back propagation. Subtracting an ε from the input of the $\tanh^{-1}$ function did not break back propagation and clips the output of an even layer to $\approx \pm 16.6$, a value that fits perfectly into the suggested clipping interval.

5.2 Loss Function

The target variables of the SPA-NN are Bernoulli distributed, therefore for experiments and training a Cross Entropy based loss function (CEL) as used. [cf. Agr17, p. 41 f.]

Definition 5.3: Cross Entropy Loss Function (CEL)

The CEL calculates the error based on the neural networks prediction $\mathbb{P}(x=0)$ for input codeword x, and the target codeword y.

$$CEL(\mathbb{P}(x=0), y) = -\frac{1}{N}\sum_{n=1}^{N} y_n \cdot \ln(1-\mathbb{P}(x_n=0)) + (1-y_n)\cdot \ln(\mathbb{P}(x_n=0)$$

where
N is the length of both codewords
y is the binary vector of the target codeword
y_n is the nth bit of the target codeword and
$\mathbb{P}(x_n=0)$ denotes the NNDs prediction for the n-th bit of the input vector being a zero.

In the CEL the target bit value is used as an selector whether the logarithm is used on the probability for the bit to be one or the bit to be zero. In the implementation a perfect prediction was a problem because this selection was applied after evaluating the logarithm. This lead to the calculation of $\ln(0)$ in the not selected part. Pytorch evaluates $\ln(0)$ to $-\infty$ and the multiplication $0\cdot-\infty$ to `nan`. This `nan` value of one bit propagates through the whole loss calculation. To prevent this behaviour a check was added to add 0 to the sum whenever a prediction was perfect. In the implementation the loss is calculated for every codeword and at the end of one epoch all losses are summed up. To compare the losses of training-, test- and validation dataset all losses in plots have been normalized by the number of codewords in the dataset to make it easier to compare them.

5.3 Hard Decision

There is no hard decision made on the output as described in Definition 2.8 because the output of the neural network decoder is not using a LLR. This is because a sigmoid function is used to transform the probability of a bit being zero from the its LLR value.[cf. Agr17, p. 29]

Definition 5.4: Sigmoid Function

The sigmoid function $\sigma(x)$ is defined as follows with e being Euler's number:

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

This step is crucial because it squeezes the obtained values into probabilities that can be used for the Cross Entropy based Loss function. [cf. Agr17, p. 41 f.] In my implementation a hard decision for the sigmoid of the LLR was added.

Definition 5.5: Hard Decision Sigmoid

A hard decision on the output of the sigmoid function $HD_\sigma(x)$ is made as follows:

$$HD_\sigma(x) = \begin{cases} 0, & \text{if } x \geq 0.5 \\ 1, & \text{if } x < 0.5 \end{cases}$$

To check intermediate results a hard decision based on the LLR could also be made after every two layers of the neural network which is equivalent to making a hard decision after every iteration of the SPA.

5.4 Dimensions

For the implementation I decided that every output of every layer is always a vector (not a lying vector).

6 Experiments

This section gives an overview about the results and learnings during the experiments. Each experiment is preceded by a hypothesis that is evaluated in the process of interpreting the results of the experiment. It is described where the results matched the expectation and where they differ. The experiments focus on different SNR_{dB}s for training and a comparison between a fixed training set and a on the fly generated training set as well as on trying out different batch sizes. When selecting hyper-parameters it is a trade-off between performance of the NND and complexity. Experiments with hyper parameters like learning rate and optimizer are not performed because Agrawal states that the choice of the optimizer does not make a difference therefore RMS-Prop as supposed by Agrawal is chosen for the implementation. [cf. Agr17, p. 37]. According to Agrawal the NND decoder performs only slightly better than SPA because LDPC Codes do not have many small girth cycles or trapping sets.[Agr17, p. 63] The following experiments where done on the example that was used in this thesis which is not as sparse as bigger LDPC codes would be. However it was possible in the experiments to find best practices for training the NND based on this example. Within the experimental protocol the following new abbreviations are used:

BS	Batch Size
E	Epochs
CW	Codewords
CW/E	Codewords per Epoch
F	Fixed dataset (as opposite from a on the fly generated dataset)
OTF	On The Fly generated dataset (as opposite from a fixed dataset)
val.	validation

A Jupyter Notebook is provided to explore the measured performances of the network during training as well as an evaluation of the trained network. Plots for all experiments can be found and reproduced in the Jupyter Notebook. In order to stay within the scope of the thesis, only a selection of plots is printed here. Particular attention was paid to selecting plots that were especially remarkable, or that supported a statement well. Keep in mind that losses in the plots as well as in the experiment tables are normalized

by the number of codewords used during training and for validation to bring all losses to the same scale. For better readability, values (e.g. losses, NVS) are rounded to three decimal places.

6.1 Metrics

To measure the performance of the trained NND the **Bit Error Rate** BER (see Definition 2.10) is used. As a baseline of how hard a noisy codeword is to decode all codewords are also decoded by the classic SPA. Therefore the LDPC decoder implemented by Veeresh Taranalli was used.[22] In this implementation the LLR is swapped when compared to the LLR defined in this thesis (see Definition 3.6): $\mathbb{P}(bit(v_j) = 0)$ is now below the fraction line while $\mathbb{P}(bit(v_j) = 1)$ is above the fraction line. This issue was resolved by multiplying the LLR of the noisy codeword with -1 before using Taranallis decoder. Also the implementation was not stable on LLR values exceeding ± 10. Therefore when data is generated the LLR will be clipped to ± 10, which still is within the suggested LLR range (see subsection 5.1).
A **Normalized Validation Score** is used to combine the BERs of NND and SPA decoding SNR_{dB}-wise.

Definition 6.1: Normalized Validation Score (NVS)[23]

Let S be a set of different SNR_{dB} and $s \in S$. Let $D(s)$ be a dataset created by using s. The NVS normalizes the BER of the NND with the BER of the SPA. The results are summed up and averaged over the number of used SNR_{dB}s. Note that the same dataset per SNR_{dB} $D(s)$ is used to calculate the neural networks bit-error-rate $\text{BER}_{\text{NND}}(D(s))$ and the bit-error rate of the sum-product algorithm $\text{BER}_{\text{SPA}}(D(s))$.

$$\text{NVS}(S) = \frac{1}{|S|} \cdot \sum_{s \in S} \frac{\text{BER}_{\text{NND}}(D(s))}{\text{BER}_{\text{SPA}}(D(s))}$$

Both the NND and the SPA are given five iterations for decoding. **The lower the NVS, the better the performance of the NND is compared to classic SPA decoding.** The NVS is not covering the case where SPA perfectly decodes all codewords in a SNR_{dB} bucket. This case would lead to a BER_{SPA} of zero and therefore to a division by zero. In my implementation I replace BER_{SPA} with an epsilon $\varepsilon = \frac{0.5}{\text{number-of-bits-in-SNR-bucket}}$ if BER_{SPA} is equal to zero. The value of 0.5 is quite arbitrarily chosen, the main point is that the resulting BER is smaller than the smallest possible BER of $\frac{1}{\text{number-of-bits-in-SNR-bucket}}$.
One would expect that SPA and NND have the same BERs and therefore a NVS of one at the start of the training, which is not the case. The reason for this behaviour is that the bit error rates per SNR_{dB} for the neural network are calculated at the end of an epoch. Therefore the first value in the plot shows already the improvement of the neural network in relation to the SPA.
As BERs of zero can happen, I created a Simple BER Comparison(SBC) metric that is less prone to zeroes. Instead of looking at the ratio between BER_{NND} and BER_{SPA} it focuses at the difference between the BERs.

[22] Veeresh Taranalli, CommPy: Digital Communication with Python, version 0.3.0. Available at `https://github.com/veeresht/CommPy`, 2015.
[23] [cf. Agr17, p. 33 ff.]

Definition 6.2: Simple BER Comparison (SBC)

Let S be a set of different SNR_{dB} and $s \in S$. Let $D(s)$ be a dataset created by using the SNR_{dB} s. The differences between the two BERs are summed up and averaged over the number of used SNR_{dB}s. Note that the same dataset per SNR_{dB} $D(s)$ is used to calculate the neural networks bit-error-rate $\text{BER}_{\text{NND}}(D(s))$ and the bit-error rate of the sum-product algorithm $\text{BER}_{\text{SPA}}(D(s))$.

$$\text{SBC}(S) = \frac{1}{|S|} \cdot \sum_{s \in S} \text{BER}_{\text{NND}}(D(s)) - \text{BER}_{\text{SPA}}(D(s))$$

With the SBC a value of minus one would be the best (but very unlikely) result, where the BERs over all SNRs achieved by the neural network are zero while the BERs achieved by SPA are one. The lower the SBC the better the neural network performs in comparison to the SPA. In comparison with the NVS the SBC leads to similar results. The SBC was therefore removed from the plots in favor of the NVS for the sake of clarity. To take a look at the overall BER during training without having to distinguish between the used SNRs an Average BER can be calculated.

Definition 6.3: Average BER (ABER)

Let S be a set of different SNR_{dB} and $s \in S$. Let $D(s)$ be a dataset created by using the SNR_{dB} s. The average BER is calculated as follows:

$$\text{ABER}(S) = \frac{1}{|S|} \cdot \sum_{s \in S} \text{BER}_{\text{NND}}(D(s))$$

where BER_{NND} can be replaced by BER_{SPA} if ABER for SPA needs to be calculated.

6.2 Datasets

The NND is trained with supervised learning on structured data. Training data is generated by transmitting codewords through a noisy channel. AWGN is used for modelling magnitude and error patterns as functions of the variance of the noise and the randomness of the noise process. As the performance of SPA is independent of the transmitted codeword, training data can be created by using only one codeword and modify it through the channel model. [cf. Agr17, p. 47] Apart from independence, the Tanner graph is the decisive factor that one code word is sufficient. The neural network created by unfolding the Tanner graph does not change the structure and the information flow by assigning weights to the edges.[24] Therefore a big plus when using NND compared to a fully-connected feed-forward neural network is that it does not require training over a large proportion of the entire code.

A short overview over the used datasets is provided in the following table. A detailed description of the datasets and an explanation for decisions made regarding datasets follow in the next chapters.

[24] If a fully-connected feed-forward neural network is used, training over a large proportion of the whole code would be necessary as the structure of the Tanner Graph is missing. This would lead to an increase in the size and complexity of the training data, which makes it impracticable to train for larger dimension codes.[cf. Agr17, p. 47]

Dataset	Number of CWs	SNR	CW Seed	Noise Seed
Training	Experiment dependent	Experiment dependent	0	11
Validation	500	same as experiment	4	5
Test S5	54000	[-5.0, -4.5, ..., 8]	5	5
Test S4	54000	[-5.0, -4.5, ..., 8]	4	4

6.2.1 Training Dataset

The all-zero codeword is part of every linear code and therefore predestined to build the training data set on it. This leads to a significant reduction in size as well as in complexity of the training data. In the following experiments using the all zero codeword during training is compared with using random generated codewords. Another experiment tries to improve the NND performance by moving from a fixed training dataset to a not fixed training dataset. For the not fixed dataset within every batch during training the noise is created anew and added to the all zero codeword. Therefore no all zero codeword - noise combination is reused on purpose. The parameters for training used by Agrawal are 2^{18} epochs, 1024 batches per epoch and a batch size depending on the size of the code. Agrawal suggests a twice as big batch size for LDPC codes than for Polar codes as the training size required to learn the error pattern is usually larger than in high density codes. For example a (n, k) LDPC code, where n is the length of the codeword and k is the length of message to be encoded, with $n = 32$ and $k = 16$ requires a batch size of at least 256. The code example used for the experiments in this thesis is a (n= 7, k=4) LDCP code which makes is sufficient to work with smaller batch sizes, less batches and less epochs.
Normally an epoch describes how often the neural network sees the whole dataset during training. With the approach of creating noise on the fly for each batch the network sees a different dataset in each epoch.

6.2.2 Validation Dataset

A validation of the NND during training is done after every epoch using a validation data set of 500 codewords.[25] If not stated otherwise in the experiment description the validation set shares properties with the training dataset when it comes to SNR and the use of all-zero-codeword. This approach differs from the procedure described by Agrawal in several ways. In his validation data set he used random codewords and a wide range of SNR values to represent a more realistic set of data but also draws attention to the problem that unrealistic SNR values or a small data set could lead to unreliable scores [cf. Agr17, p. 47 f.] He also describes the choice of these parameters as subjective and states a lack of a standard method, which enables a new point of view on the training process.

6.2.3 Test Dataset

To test the performance of the trained NND a fixed test dataset of randomly chosen codewords is used. Noise from 27 different signal-to-noise ratios in the range of $S = [-5.0, -4.5, -4.0, ..., 8.0]$ is added to the codewords. The codewords are evenly distributed over all SNRs with 2000 codewords per SNR. This leads to a test dataset size of 54000 codewords. SNR range is chosen to match the setup described by Agrawal. [cf. Agr17, p. 53] To calculate NVS and compare results the NND achieves with the results from SPA Agrawals approach ”For each SNR value, the decoding is performed till either at least 500 codewords are found in error, or a total of 50,000 codewords are tested.” is not followed. [Agr17, p.

[25] There are some special cases where not exactly 500 codewords could be used. In this case the number of codewords in the validation set is stated in the experiment setup to guarantee reproducible results.

53] The NVS calculated on the test data set to evaluate the trained network uses a 5 as seed for the codewords and the noise. Therefore this test dataset is called *Test S5*. To reproduce the results exactly the seed does matter as the NVS varies from the third decimal place depending on the noise. For results that contradicted expectation or had other abnormalities, an evaluation was performed with an alternative test data set generated with 4 as a different seed, called *Test S4*. This procedure was chosen to find out if the behaviour is conspicuous in principle or whether special outliers in the test set caused it.

6.3 Experiment: Training Dataset Size

Hypothesis: The bigger the fixed training dataset size the better the performance of the trained neural network.".

A bigger training dataset size will expose the network to more different noise patterns during training, making it easier to generalize noise patterns from a SNR_{dB} different to the training SNR_{dB}. The smaller the training data set, the faster the training, therefore the goal is to find the smallest training dataset that performs well.

Fixed training parameters:

- a fixed set of random codewords, not the all-zero-codeword only
- SNR 2
- batch size 26
- fnn structure
- 2000 training epochs

Fixed validation set parameters:

- 500 random codewords
- SNR 2

The following table compares metrics of the neural network for different training dataset sizes during training (train, validation) and after training (test). Train and validation values are picked from epoch 2000, plots are shortened to 800 epochs because loss as well as NVS are converging afterwards not showing new information but making the existing information harder to see.

CW/E	loss train	loss validation	NVS train	NVS validation	NVS test S5
260	0.074	0.121	0.299	0.573	0.566
520	0.066	0.126	0.314	0.603	0.54
1040	0.074	0.102	0.384	0.506	0.495
2080	0.079	0.099	0.399	0.487	0.493
5200	0.084	0.094	0.405	0.487	0.487

The following plots show that a training dataset with only 260 codewords has not enough codewords to generalize well to another dataset from the same SNR distribution.

Loss indicates overfitting already around epoch 100, while the normalized validation score (NVS) shows that the neural network is performing quite the same on both datasets up to epoch 500. A bigger training dataset should help here. Doubling the training dataset size is not enough to prevent this effect: loss as well as NVS are even diverging further. The NVS achieved by both networks on the test dataset show on the one hand already an immense improvement in comparison to SPA, on the other hand when

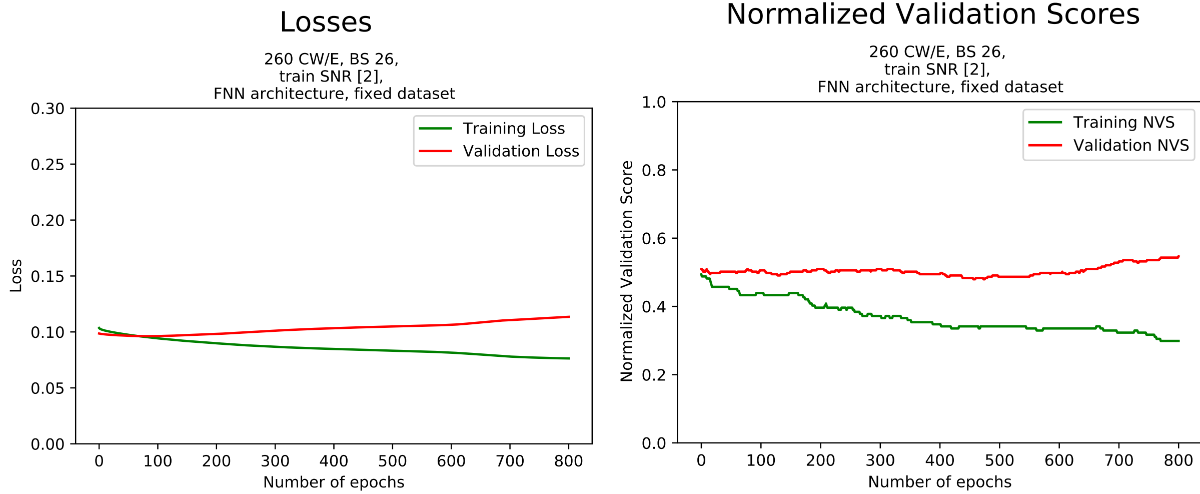


Figure 1: Training dataset size of 260 CW/E.

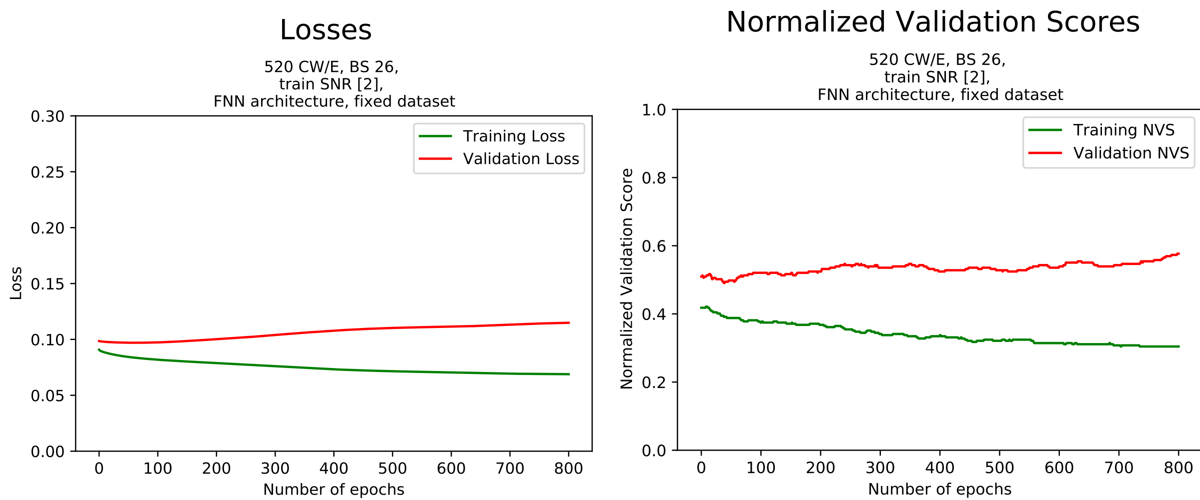


Figure 2: Training dataset size of 520 CW/E.

compared to the NVSs that are achieved by the networks trained on the bigger datasets there is still room for improvement.

Starting with a training dataset of size 1040 not only the NVS is significantly better, also training and validation loss as well as training and validation dataset NVS are not diverging as strongly any more.

Using more codewords seems to enable overcoming the drawback of overfitting to the training dataset. Taking a first look at the plots where a dataset of 1040, 2080 or even 5200 codewords was used during training might raise the thought that the network is not learning after a view epochs. The reason for this behaviour can be found in the relationship between dataset size and learnable parameters. The trained network is a minimal working example. It has eleven layers and therefore eleven weight matrices but

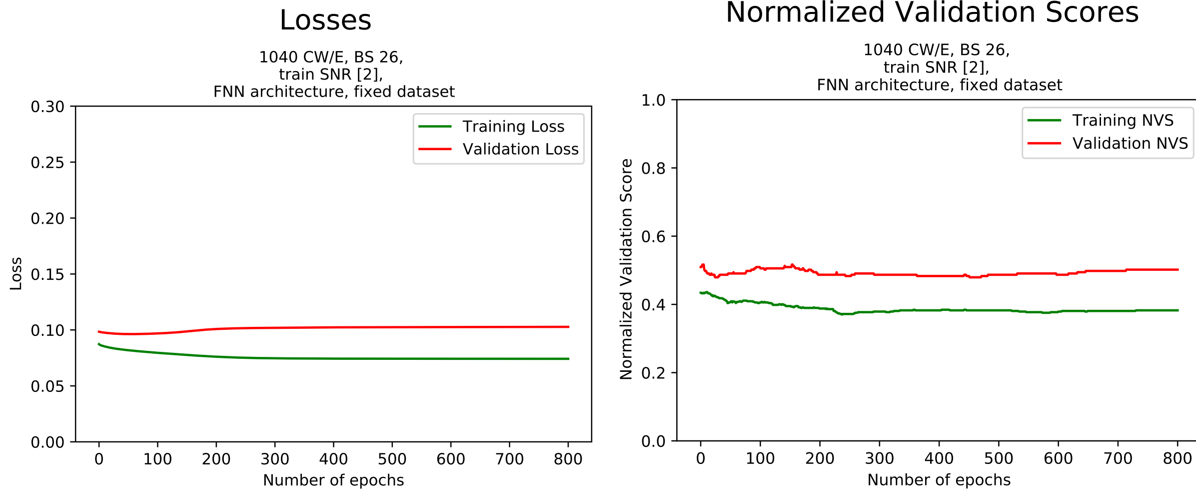


Figure 3: Training dataset size of 1040 CW/E.

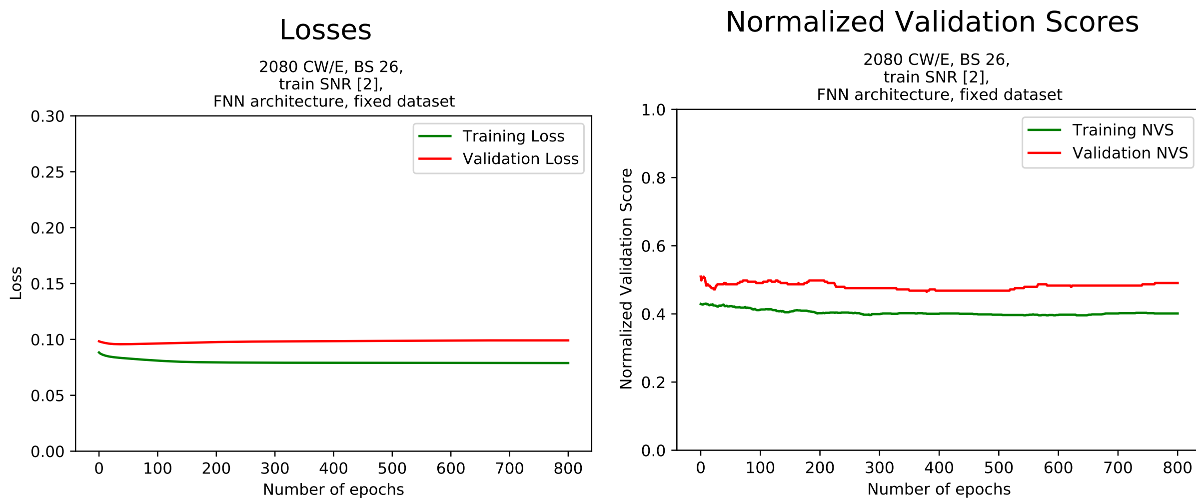


Figure 4: Training dataset size of 2080 CW/E.

only the four $W_{even2odd}$ matrices are trained. In these matrices, there are only twelve non-zero entries in total allowed. This is necessary to ensure the same information flow as provided by SPA. These sparse weight matrices lead to only 48 learnable parameters in total.

When comparing NVSs of trained networks one can clearly see that a bigger fixed training dataset size enhances the performance of the network. The experiment also showed that when using more than 2000 codewords the major training steps happen within the first 200 epochs. Based on this discovery, fewer epochs are also used in the following experiments when possible to speed up the process.

The last plot gives an overview of the BER achieved by the different dataset sizes on the test dataset. A logarithmic scale is used for BER to make differences clearer visible on higher signal-to-noise ratios. If

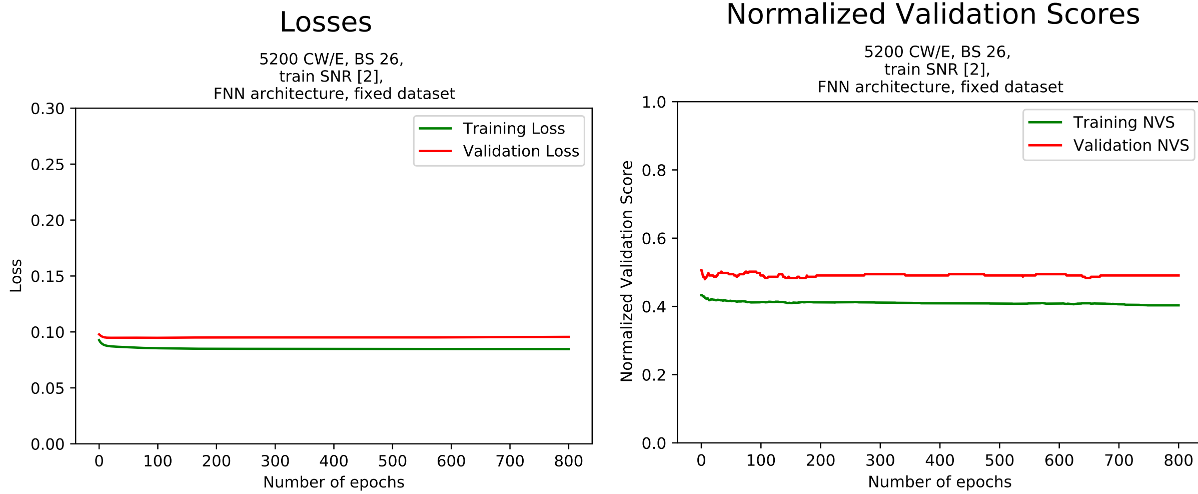


Figure 5: Training dataset size of 5200 CW/E.

a BER is equal to zero the logarithmic scale will result in negative infinity. Therefore the plot is cropped at 10^{-5} in the y axis. A plot for the BERs achieved by every training dataset size compared to the BERs achieved by SPA on a non logarithmic scale can be found in the evaluation jupyter notebook.

The logarithmic scale focuses on the higher SNRs. One can see that on the used testset 1040 CW/E and 5200 CW/E decoded perfectly with a BER of 0 for all SNRs bigger than 7, but on lower SNRs the bigger training dataset was able to prepare the neural network better. Overall one can see that decoding with the trained neural network leads to a better result than only using SPA.

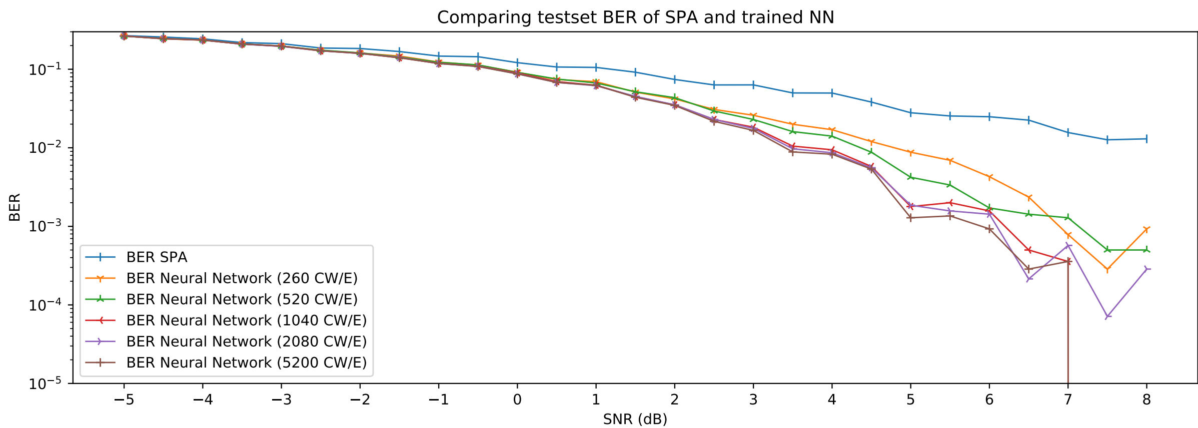


Figure 6: Compare BERs of trained network on networks trained on different training dataset sizes.

6.4 Experiment: Train on All-Zero Codeword

Hypothesis: "It does not matter if the all-zero codeword only or randomly picked codewords are used in the training data set."
The performance of SPA is independent of the transmitted codeword and the transformation of SPA to the NND via unfolding the Tanner graph does not change the flow of information. Therefore the NND should also be independent of the codewords it is trained on. The decoding performance should not depend on the values of the transmitted bits but on the error patterns induced by the channel. This is because output produced by variable nodes and check nodes is a function of the error-patterns in the channel input represented by the LLR.
Fixed training parameters:

- 5200 CW/E
- SNR 2
- batch size 26
- fnn structure
- 2000 training epochs

Fixed validation set parameters:

- 500 random codewords
- SNR 2

To compare the performance the following four different combinations of training and validation datasets are used. For the first case (train: random, validation: random) it was possible to reuse the results from the training dataset size experiment.

train: zero, random validation: zero, random	loss train	loss validation	NVS train	NVS validation	NVS test S5
train: random validation: random	0.084	0.094	0.405	0.487	0.487
train: zero validation: zero	0.081	0.087	0.404	0.458	0.486
train: random validation: zero	0.084	0.086	0.405	0.442	0.487
train: zero validation: random	0.081	0.096	0.404	0.494	0.486

The noise added to the codewords has the same seed whether the all zero or random codewords are used. One would expect the losses be exactly the same because the decoding performance depends on the channels error patterns and not on the transmitted bits. This experiment shows that for training a neural network this is not the case. It does make a slight difference if training is done on random codewords or the all-zero codeword. This deviation from the expected behaviour can be attributed to the loss function used. The cross entropy loss function punishes medium probabilities while the hard decision in SPA is insensitive here. For the hard decision is does not matter if a probability is 0.51 or 1, both lead to the same result. When random codewords are used, bit values of 1 occur. Using BPSK a bit value of 1 is mapped to -1 while a bit value of 0 is mapped to 1. When for example noise of 0.5 is added to this bit,

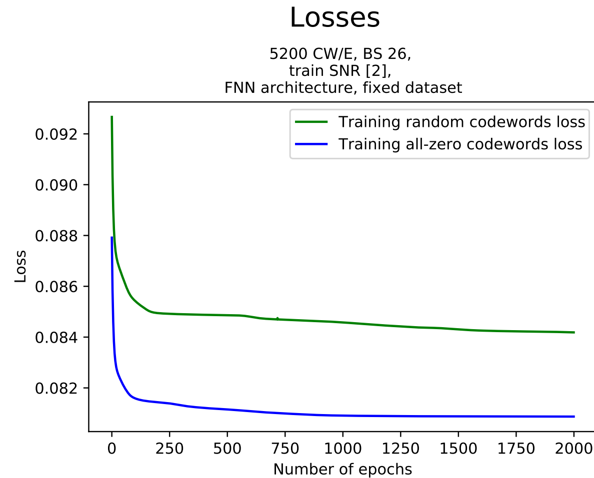


Figure 7: Comparing losses during training with all-zero codeword and a dataset of random codewords.

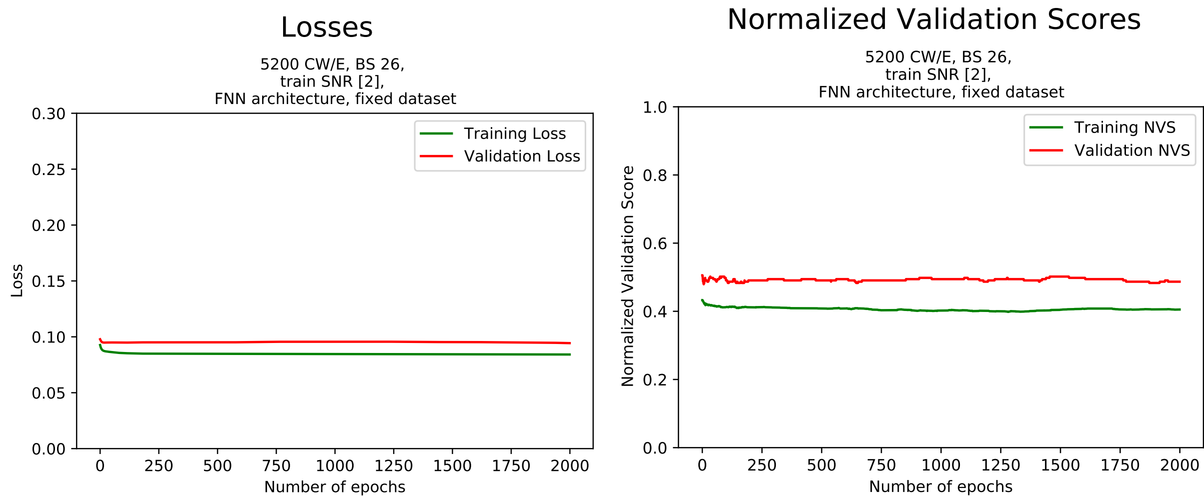


Figure 8: Random codewords in both training dataset and validation dataset.

this noise would lead to -0.5 in the one case and 1.5 in the other, introducing error in the first case while enforcing the sent value in the second case.

The experiment showed that training the neural network is not as independent from the transmitted codeword as simple decoding using SPA is. As exciting as the finding that training with an unrealistic set of all-zero codewords leads to a better result than training with a realistic dataset of random codewords is, further experiments would be needed to proof if this always is the case. The parameters of the training data set play such a significant role when it comes to the loss, with another noise or SNR the result may be different.

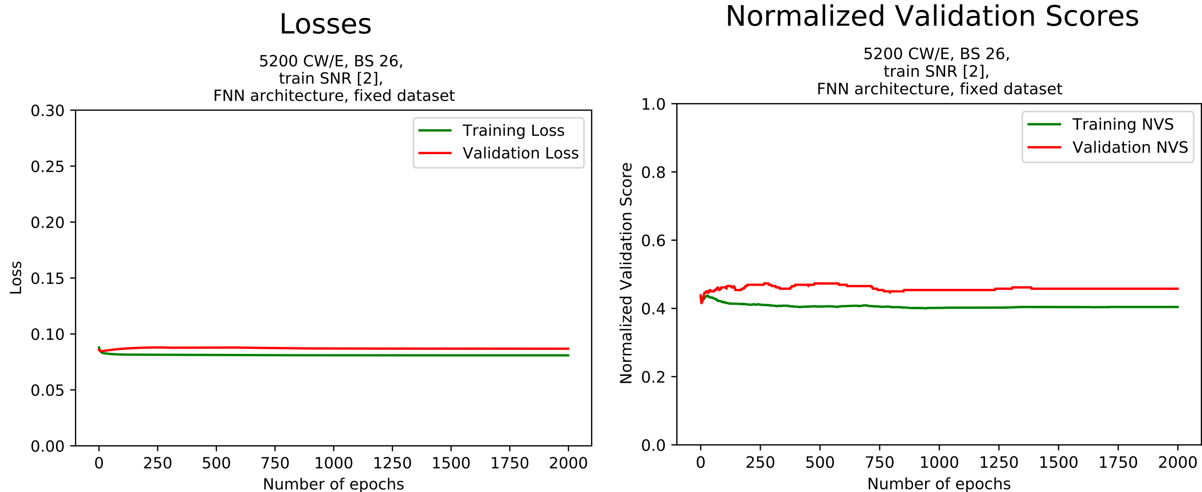


Figure 9: All-zero codewords in both training dataset and validation dataset.

6.5 Experiment: Batch Size

Hypothesis: "Code size and batch size share a linear dependency."
As stated in subsubsection 6.2.1 the best batch size depends on the size of the code. The expectation is a sweet spot around batch size 64 because the code based on H in this example is a fourth of the size of the code described in the subsection were a batch size of 256 was recommended. If code and batch size share a linear dependency it would make sense to reduce also the batch size to a fourth. To compare different batch sizes the number of codewords per epoch is set to a fixed value because the loss and NVS is calculated for a whole epoch. For easier calculation multiples of two are taken into account for possible batch sizes, starting with 16 going up to 256. The previous experiment showed that training on the all-zero codeword does not lead to a low performance, therefore the all-zero codeword is used during training in this experiment.

Fixed training parameters:

- 4096 CW/E (to use powers of two in BS)
- all-zero-codeword only
- SNR 2
- fnn structure
- 2000 training epochs

Fixed validation set parameters:

- 500 all-zero-codeword only
- SNR 2

With 4096 CW/E one can observe two different behaviours during training. The loss of all batchsizes except BS 64 look alike with the training and validation losses diverging after a view epochs. BS 64 shows unique behaviour during training with both losses and NVSs nearly melding which can be observed in fig. 11.

SNR_{dB}	loss train	loss val.	NVS train	NVS val.	NVS test S5	NVS test S4
16	0.08	0.088	0.434	0.462	0.489	0.492
32	0.077	0.086	0.399	0.454	0.488	0.49
64	0.083	0.083	0.435	0.431	0.492	0.489
128	0.082	0.087	0.449	0.473	0.49	0.495
256	0.081	0.086	0.407	0.454	0.489	0.487

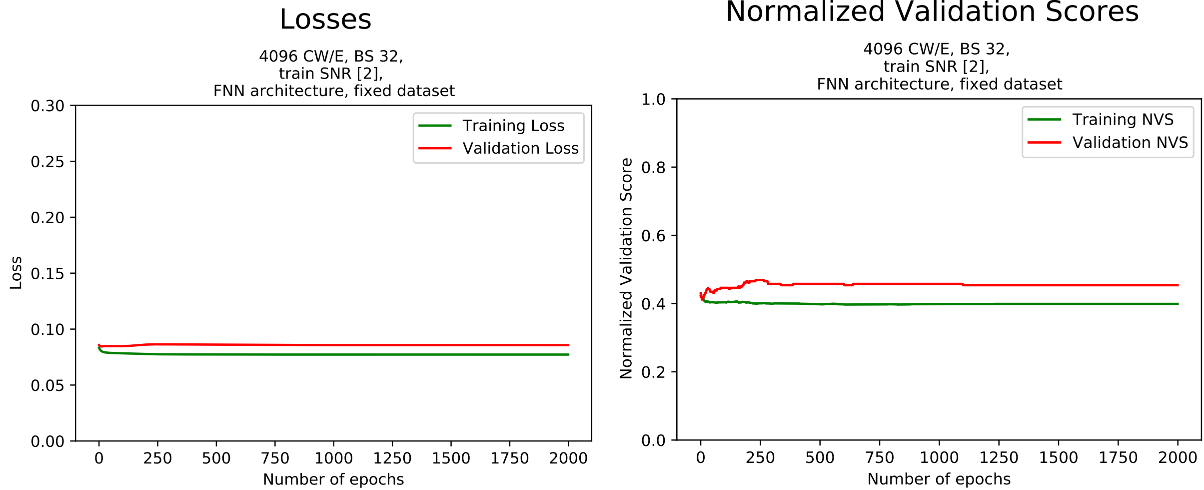


Figure 10: Losses and NVS during training for BS 32.

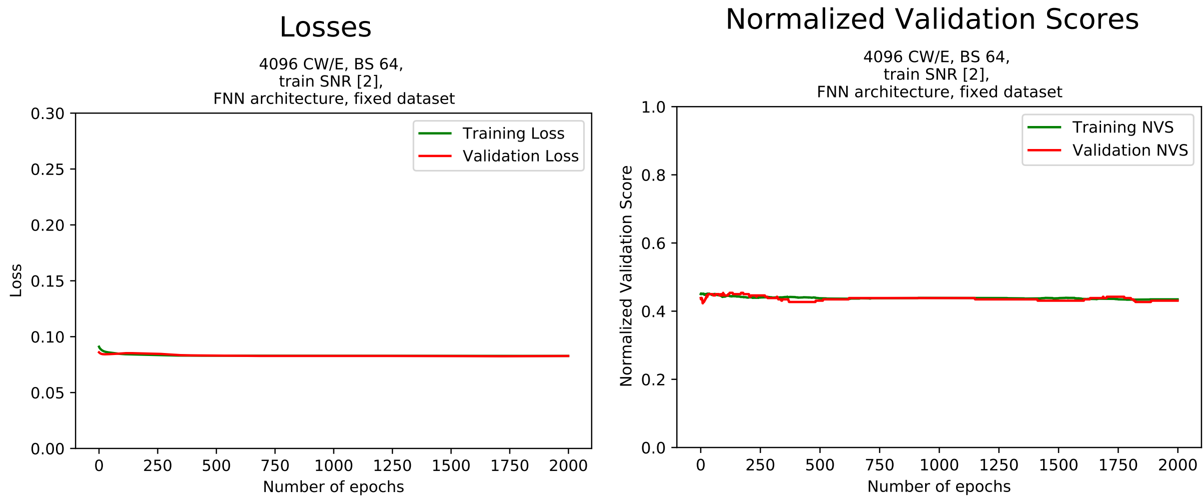


Figure 11: Losses and NVS during training for BS 64.

When evaluating the trained networks on the test dataset, BS 64 had the worst NVS, all tough all batch-sizes were very close to each other. It is not possible to deduce a linear dependency from this result, it

rather is to be excluded.

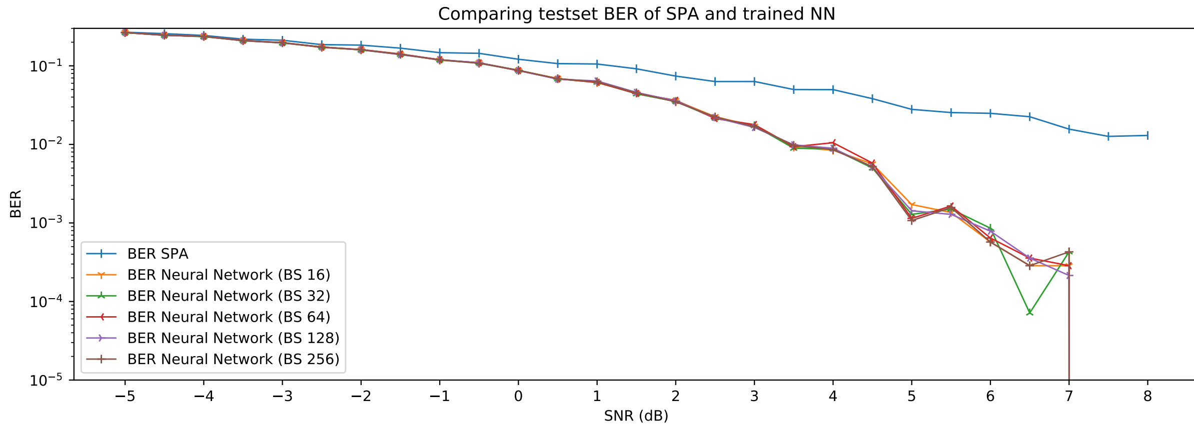


Figure 12: Comparing BERs of trained networks on test dataset S5, using a logarithmic scale for BER (where a BER of $0 \to -\infty$).

When comparing the raw BERs instead of the achieved NVS in fig. 12, one immediately sees why BS 32 seems to be the best choice. The network trained with BS 32 shows a significantly better BER on SNR_{dB} 6.5 but otherwise shows nearly the same behaviour as the other networks. This outlier of BS 32 indicates that the evaluation of which batch size is the best at this point depends predominantly on the test set used. To test this assumption, the evaluation is performed once with seed 4 for code words and their noise in test set. The result confirms the assumption. The worst NVS of 0.495 is achieved with BS 128 while the best NVS of 0.487 is accomplished by BS 256, followed by BS 64 with a NVS of 0.489. The batchsize used does not make a noticeable difference for this example when evaluating the trained network.

6.6 Experiment: SNR_{dB} Combinations for Training

Hypothesis: "Training on only one SNR_{dB} leads to results equivalent to using a range of SNR_{dB} in the training data set."

Picking the right combination of SNR_{dB} in training is not well explored yet. In one experiment on a (32, 16) polar code training on a fixed SNR_{dB} of 1 dB lead to an equal result as training with a varying SNR_{dB} from -2 to 4 dB, but the training SNR_{dB} value might be different for different families or size of the code. [cf. Agr17, p. 49] The goal of the first experiment is to find the best single SNR_{dB} for training. SNR_{dB} in the range of [-4, -3, -2, -1, 0, 1, 2, 3, 4] have been tested.

Fixed training parameters:

- 4096 CW/E
- a fixed set of random codewords
- BS 64
- fnn structure
- 2000 training epochs

Fixed validation set parameters:

- 500 random codewords
- use same SNR as in train dataset

From the NVS the trained networks achieve on the test set one can see that training on a single SNR_{dB} works especially well in the high SNR_{dB} area. At a SNR_{dB} of zero the signal power is just as strong as the noise power, a SNR_{dB} of 1 is about equal to the signal power being by ≈ 0.25 stronger than the noise. If the signal is by ≈ 0.6 stronger than the noise a SNR_{dB} of 2 is reached. On the example used in this thesis training on a SNR_{dB} of 2 is the smallest SNR_{dB} that leads to the best result during training. One should keep in mind that a too high SNR_{dB} will not expose the network to enough noise. This could not observed for the chosen test set. Therefore a run on an alternative test set using seed 4 for both codewords and noise is conducted. The trained networks showed the same tendency for all SNR_{dB}s, only at SNR_{dB} 4 the network started to perform slightly worse. It seems that SNR_{dB} 4 reaches the point where noise gets sparse.

SNR_{dB}	loss train	loss val.	NVS train	NVS val.	NVS test S5	NVS test S4
-4	0.471	0.486	0.967	0.944	0.59	0.598
-3	0.416	0.433	0.939	0.944	0.52	0.52
-2	0.354	0.368	0.896	0.896	0.507	0.511
-1	0.284	0.298	0.809	0.792	0.496	0.497
0	0.214	0.224	0.73	0.708	0.494	0.495
1	0.146	0.156	0.598	0.614	0.491	0.492
2	0.087	0.094	0.433	0.487	0.49	0.489
3	0.044	0.052	0.276	0.333	0.49	0.489
4	0.017	0.025	0.143	0.244	0.49	0.491

The second part of this experiment searches for different SNR_{dB} combinations within the training dataset that outperform the neural networks trained on one single SNR_{dB}. Therefore multiple combinations of SNR_{dB} are compared to each other. The number of codewords in the training dataset will be slightly adjusted so codewords are identically distributed over all used SNR_{dB}. In these experiments the network is trained for 750 epochs. This value is picked according to the experiments done so far which show that the most relevant learning happens within the first hundred epochs. The experiment on training on the all-zero codeword showed, that training with the all-zero codeword does not lower the performance, the all-zero codeword is used in train and validation set. Batch size, CW/E and codewords in validation data set are picked to ensure a uniform distribution: the same number of codewords per SNR_{dB} within the train dataset. This leads to slightly different batch sizes as noted below.

Fixed training parameters:

- all-zero-codeword training data
- BS 64 for 4 SNR_{dB}, BS 63 for 3 and 7 different SNR_{dB}, BS 60 for 5 and 6 SNR_{dB}
- fnn structure
- 750 training epochs
- about 4200 CW/E (4221 CW/E for 7 different SNR_{dB} and 4224 CW/E for 4 different SNR_{dB})

Fixed validation set parameters:

- 420 all-zero codewords
- same SNR distribution as in training

SNR	loss train	loss validation	NVS train	NVS validation	NVS test S5
[-4, -3, ..., 2]	0.271	0.295	0.75	0.769	0.493
[-3, -2, ..., 3]	0.21	0.224	0.63	0.656	0.49
[-2, -1, ..., 4]	0.152	0.163	0.523	0.529	0.488
[-4, -3, ..., 1]	0.311	0.326	0.809	0.813	0.497
[-3, -2, ..., 2]	0.247	0.253	0.721	0.7	0.492
[-2, -1, ..., 3]	0.186	0.192	0.611	0.618	0.495
[-1, 0, ..., 4]	0.131	0.129	0.498	0.506	0.489
[-4, -3, -2, -1, 0]	0.346	0.357	0.872	0.894	0.497
[-3, -2, -1, 0, 1]	0.279	0.285	0.784	0.769	0.495
[-2, -1, 0, 1, 2]	0.213	0.213	0.668	0.681	0.49
[-1, 0, 1, 2, 3]	0.151	0.146	0.543	0.528	0.491
[0, 1, 2, 3, 4]	0.099	0.098	0.424	0.385	0.49
[-4, -3, -2, -1]	0.379	0.389	0.897	0.915	0.503
[-3, -2, -1, 0]	0.312	0.321	0.834	0.814	0.491
[-2, -1, 0, 1]	0.242	0.249	0.738	0.723	0.488
[-1, 0, 1, 2]	0.173	0.184	0.63	0.712	0.488
[0, 1, 2, 3]	0.113	0.125	0.515	0.578	0.49
[1, 2, 3, 4]	0.066	0.072	0.365	0.437	0.489
[-3, -1, 1, 3]	0.215	0.218	0.646	0.724	0.488
[-4, -2, 2, 4]	0.226	0.223	0.605	0.634	0.49
[-4, -3, 3, 4]	0.232	0.225	0.595	0.64	0.493
[-3, 0, 3, 4]	0.165	0.164	0.512	0.521	0.487
[-4, -3, -2]	0.409	0.429	0.915	0.958	0.546
[-3, -2, -1]	0.346	0.367	0.86	0.894	0.496
[-2, -1, 0]	0.275	0.299	0.795	0.868	0.494
[-1, 0, 1]	0.205	0.225	0.691	0.781	0.491
[0, 1, 2]	0.14	0.154	0.546	0.643	0.487
[1, 2, 3]	0.086	0.095	0.424	0.521	0.492
[2, 3, 4]	0.047	0.05	0.276	0.297	0.492
[-4, 0, 4]	0.226	0.239	0.601	0.743	0.492
[-3, 0, 3]	0.215	0.234	0.631	0.775	0.49
[-2, 0, 2]	0.208	0.228	0.669	0.839	0.489

Combinations that do not work well are combinations with only negative SNR_{dB}s like [-4, -3, -2, -1] or [-4, -3, -2]. With those combinations too much noise hinders learning during training. What cannot be observed is a deterioration in training when (only) high positive SNR_{dB}s are used. This behaviour is expected because the same tendency occurred when a single SNR_{dB} was used during training. Taking a closer look and comparing the NVS achieved by the validation dataset and the NVS achieved by test dataset S5 one can observe very significant differences. Take for example the last entry in the table of SNR_{dB} combinations: combination [-2, 0, 2]. The NVS on the validation set is 0.839 while the NVS on the test set is 0.489. The gap can be explained by the differing data sets. The validation set has 140 codewords for each SNR_{dB}, so 140 CW with SNR_{dB} -2, 140 CW with SNR_{dB} 0 and 140 CW with

SNR_{dB} 2, so 420 codewords in total. The test dataset has 2000 codewords for each SNR_{dB} and it covers an SNR_{dB} range of [-5, -4.5, ... 8] with a total of 54000 codewords. Since the NVS of the test dataset shows that the trained network is capable of generalization, the validation dataset is not able to describe this property sufficiently. However, the validation dataset used during training does not sufficiently describe this property. This is most likely caused by the fact that, compared to the training with only a single SNR_{dB}, only one third of the code words per SNR_{dB} are now contained in the validation set. The codewords per SNR_{dB} in the validation sets are too small to properly represent their noise distribution. To confirm this assumption the experiment for SNR_{dB} [-2, 0, 2] was carried out again but with 500 codewords per SNR_{dB} in the validation dataset.

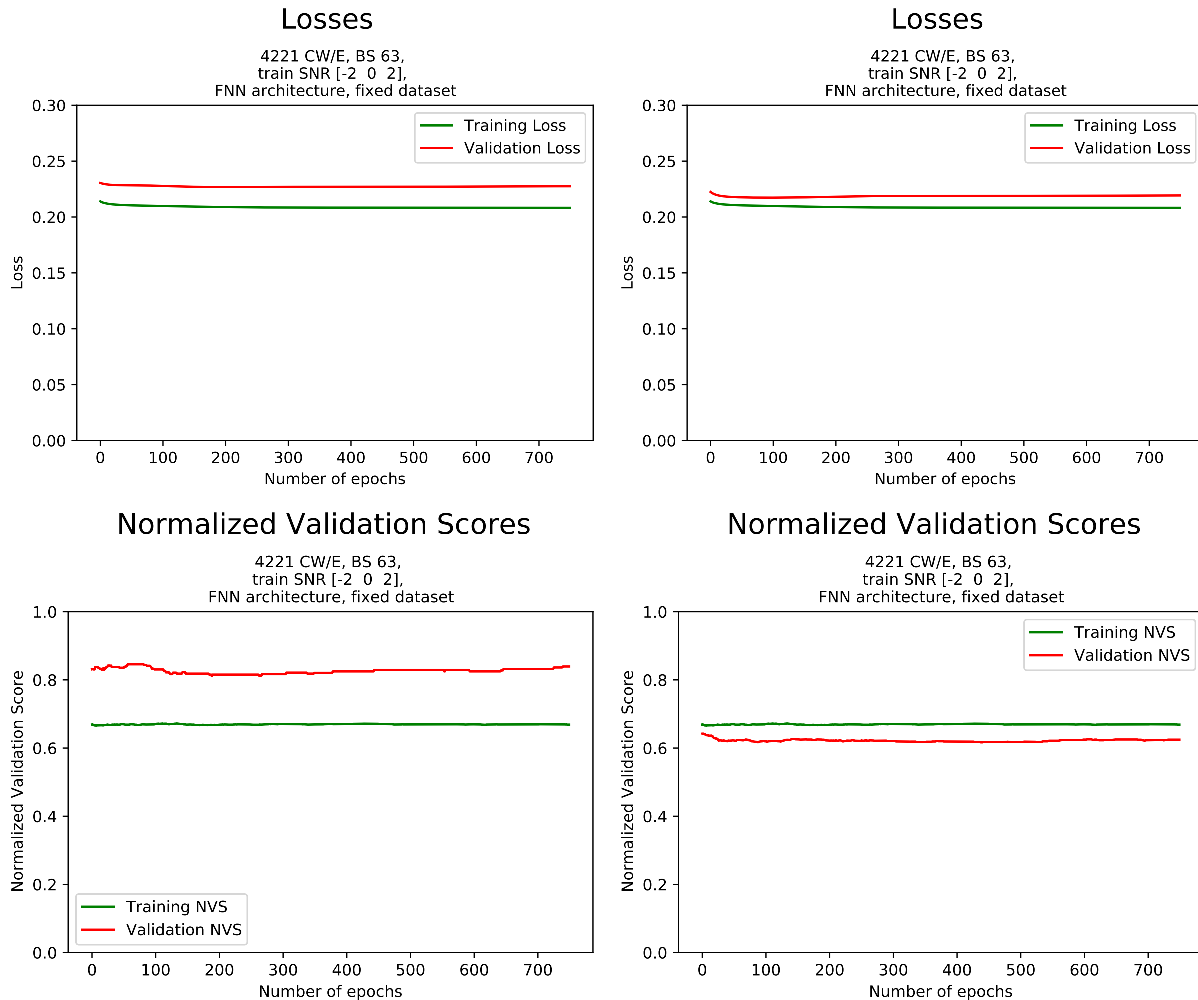


Figure 13: Left side: Loss and NVS for the validation dataset originally used with 140 codewords for each SNR_{dB}. Right side: Loss and NVS for the adapted validation dataset with 500 codewords for each SNR_{dB} mimicking the setup when only one SNR_{dB} is used.

The behaviour changed as expected so for a re-run of the experiment, there should also be 500 codewords for each SNR_{dB} in the validation dataset to provide more meaningful NVS for the validation dataset. This

procedure in turn slows down the training, since after each epoch the NVS has to be calculated for a three- to sevenfold amount of code words (depending on the SNR_{dB} range used).
The best NVS on the test dataset achieved with training on one SNR_{dB} is 0.49 while the best NVS achieved through SNR_{dB} combinations is 0.487. The hypothesis is therefore not true. However if the single SNR_{dB} NVS is compared with the worst SNR_{dB} combination NVS of 0.546 and the best SNR_{dB} combination NVS of 0.487 one can see that training with only one SNR_{dB} leads to an only slightly worse outcome than the best SNR_{dB} combination. From 32 tested combinations 17 performed worse, six equal and nine better than a single SNR_{dB}. Training on a combination leads to better results but finding the right combination takes some time and experiments. If one does not have the time to go over all combinations training with only one SNR_{dB} can be sufficient.

6.7 Experiment: On the Fly Training Set - Training Dataset Size

Hypothesis: "An on the fly created training dataset using one SNR_{dB} is able to train the network better than a fixed dataset."
One could assume that creating noise anew for every batch would lead to a better result because the neural network is able to train on more and different noisy codewords. Also during training the network is exposed to more and different forms of noise. This approach is also challenging because the networks weights will be adjusted to a different minimum every time what can prevent learning. Creating new training data during training leads to more overhead (computing the BER of the SPA for each epochs training data, saving and storing those BERs for NVS calculation and plotting). For 260 codewords per epoch the difference was already 1.5 seconds per epoch. Therefore it takes longer to train with this approach. To generate a different dataset every epoch but keep the results reproducible the number of the actual calculated epoch is given as noise seed for the training dataset creation.

Fixed training parameters:

- all-zero-codeword
- SNR 2
- batch size 65
- fnn structure
- 2000 training epochs

Fixed validation set parameters:

- 500 random codewords
- SNR 2

CW/E	loss train	loss validation	NVS train	NVS validation	NVS test S5
260	0.084	0.085	0.449	0.446	0.49
520	0.107	0.086	0.522	0.45	0.488
1040	0.088	0.085	0.515	0.446	0.487
2080	0.077	0.085	0.454	0.427	0.487
5200	0.084	0.086	0.443	0.458	0.486

As anticipated the OTF training datasets loss and NVS fluctuate a lot more than the loss and NVS of the fixed dataset. This behaviour can be explained by the fact that the NND on an OTF dataset is striving to

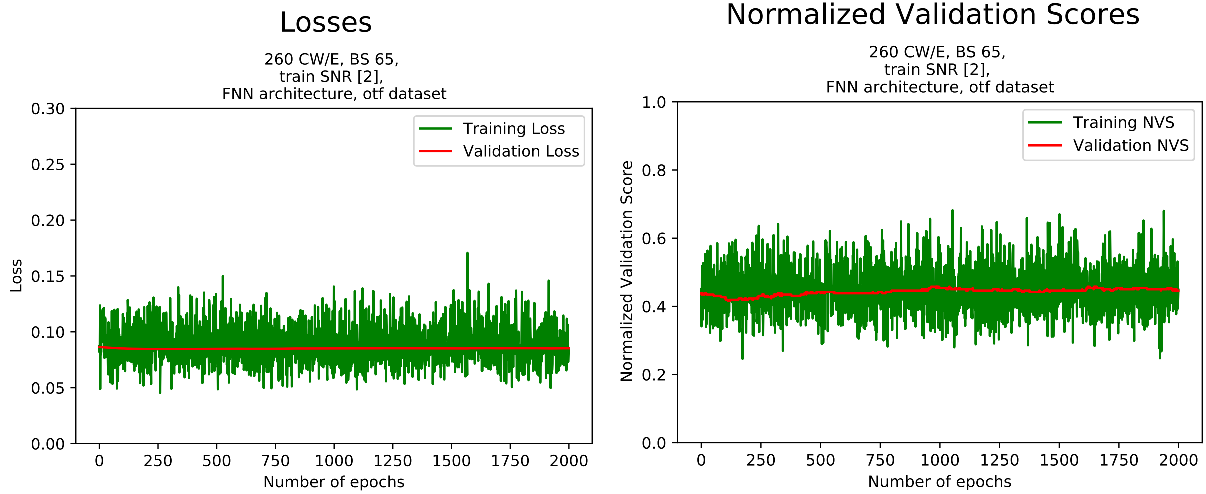


Figure 14: Loss and NVS for OTF training data set with 260 CW/E.

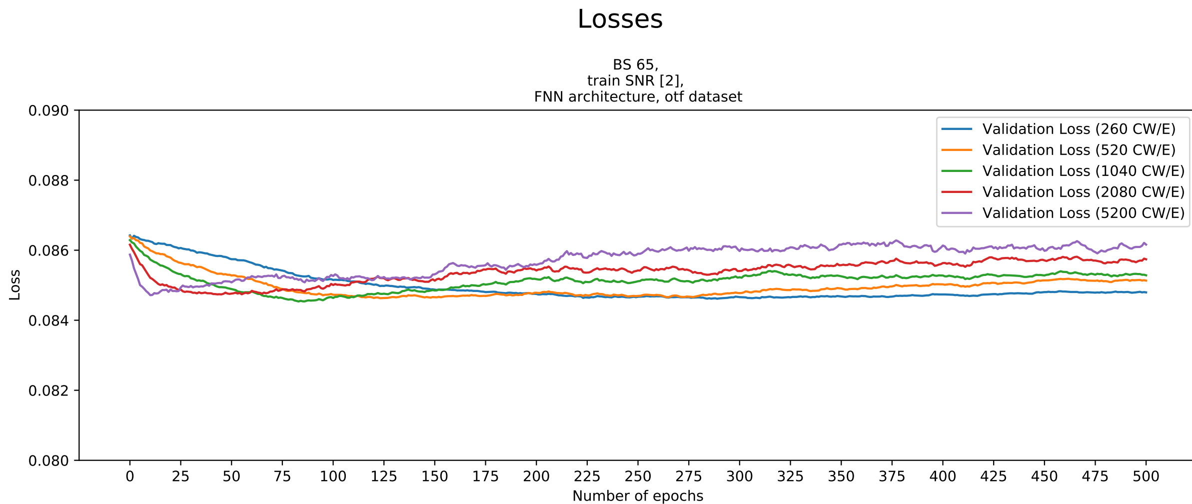


Figure 15: Comparison of the validation losses during training on OTF dataset.

a different minimum in every batch in every epoch. While the training dataset is created anew for each epoch, the validation dataset stays the same during training. So it is possible to compare the performance of the networks during training based on the validation datasets loss.

When comparing this experiments loss with the training dataset size experiment, it is remarkable that the loss in the training dataset size experiment becomes smaller the higher the number of CW/E is (from 0.121 to 0.094) while all losses in this experiment are either 0.085 or 0.086. Since the loss is related to the number of errors made during decoding, a low loss is an indicator of a good performance. The same is true for a low NVS. With a NVS of 0.49 on the test set especially the network trained with 260 OTF generated CW/E clearly outperforms the network trained on 260 fixed CW/E which only achieves a NVS

of 0.566 on the test dataset (see subsection 6.3). Both experiments do not share the same batch size but as the experiment on batch size showed, this hyper-parameter is not a crucial one which makes it valid to compare the results.
One could assume now that the more epochs the network is trained with an OTF dataset the better the metrics are. If looking more closely at the plots of the training process focusing on the loss of the validation set in fig. 15, it is noticeable that the loss does only mildly change after about 200 epochs. This indicates that the network has converged. One can also observe an overfitting to the training dataset which is especially noticeable for training datasets with more than thousand codewords in them. When one compares the development of training and validation loss in other experiments it is evident that a converging validation loss goes hand in hand with a converging training loss. This strongly indicates that it would be possible to train the network for more epochs on OTF generated data because the training loss keeps fluctuating, but it is not necessary to do so because the network already learned its parameters. The OTF trained network does not need more epochs as it already outperforms networks trained on a fixed dataset. The experiment could thus confirm the hypothesis.

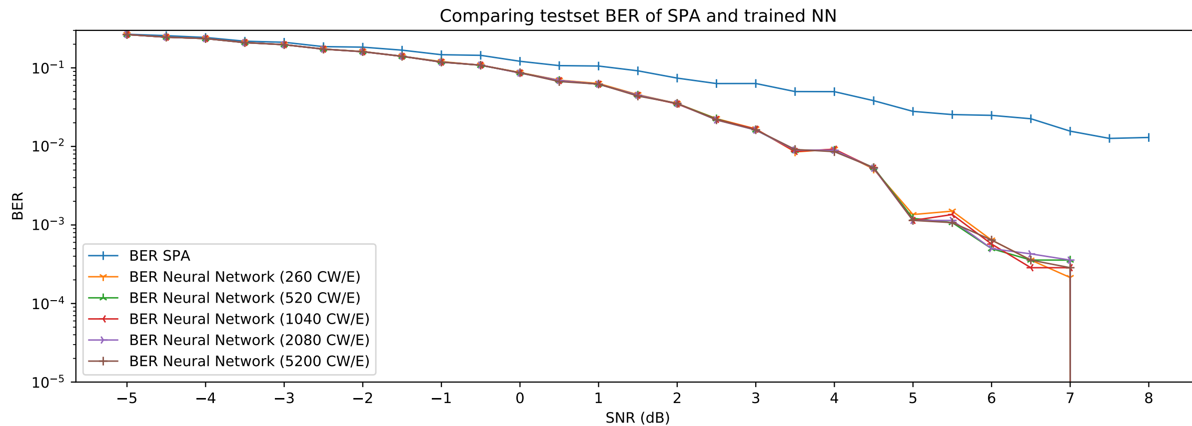


Figure 16: Comparing BERs of OTF dataset trained networks on test dataset S5 for different training dataset sizes.

When the training data is generated on the fly the concept of epoch becomes obsolete. Epoch describes no longer that the whole training data set has been seen once. It seems that it is more important to count the number of batches seen as well as the batch size. All tough the batch size was not essential for training on a fixed dataset it might be possible to be of importance for OTF generated training data.

6.8 Experiment: On the Fly Training Set - Batch Size

Hypothesis: "A bigger batch size in combination with an OTF training dataset leads to better trained network results."
One would assume that the weight update step after a big batch is more likely to show a tendency towards the minimum of the loss function than after a small batch.

Fixed training parameters:

- all-zero-codeword

- SNR 2
- fnn structure
- 2000 training epochs
- 4096 CW/E

Fixed validation set parameters:

- 500 random codewords
- SNR 2

SNR_{dB}	loss train	loss val.	NVS train	NVS val.	NVS test S5	NVS test S4
16	0.085	0.083	0.414	0.427	0.487	0.486
32	0.083	0.085	0.435	0.442	0.487	0.486
64	0.083	0.084	0.449	0.427	0.486	0.485
128	0.087	0.086	0.413	0.45	0.486	0.487
256	0.083	0.085	0.429	0.446	0.487	0.486
512	0.082	0.085	0.442	0.442	0.487	0.487
1024	0.087	0.085	0.444	0.446	0.487	0.487

Since it was up to the seed of the test set which network performed better in the last batch size experiment, the results of this experiment will be evaluated also on the second test dataset S4. The NVS on both test sets was not able to proof the hypothesis of a bigger batch size leading to a better result. It again seems like the batch size does not matter much. However, it is interesting to observe that on both test sets, BS 64 delivers the best NVS.

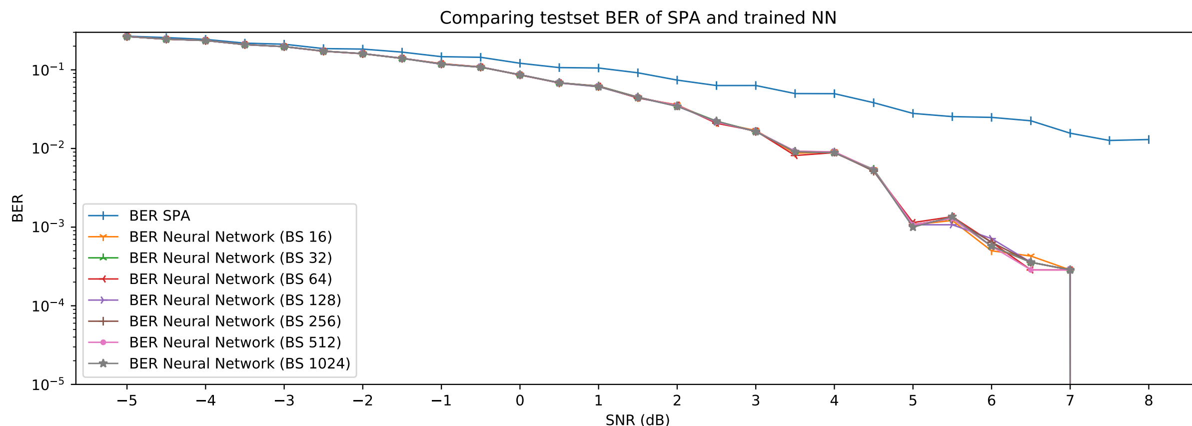


Figure 17: Comparing BERs of OTF dataset trained networks on test dataset S5 for different batch sizes.

When comparing fig. 16 and fig. 17 and Plot B, it is noticeable that both different dataset sizes and different batch sizes lead to an exceedingly similar result. Only in the SNR_{dB} range [5, ..., 6.5] are there minimal deviations.

6.9 Experiment: Feed Forward (FNN) versus Recurrent Neural Network (RNN)

Hypothesis: "Training with a feed forward architecture leads to a better result than a recurrent neural network structure."
Argawal points out that the neural network has the same connections for each iteration, so it is advantageous to share the learnable parameters to reduce the number of learn-able paramters and speed up training convergance.[cf. Agr17, p. 39] To achieve a RNN architecture the same $W_{even2odd}$ is used every second layer. On a bigger example than the one used in this thesis, sharing the weight matrix is an advantage of RNN because it requires less memory to store the model, since it always requires only four weight matrices, regardless of the number of iterations. While a feed forward architecture for this thesis' example has 48 learnable parameters, the recurrent architecture only has the 12 weights from a single $W_{even2odd}$.[26] For this minimal example using a FNN architecture should be the better choice. Not only does it allow the parameters to be learned independently for each iteration, also its higher degree of freedom will be able to adjust the network precisely.

Fixed training parameters:

- all-zero-codeword only
- SNR 2
- batch size 64
- 4096 CW/E
- 750 training epochs

Fixed validation set parameters:

- 500 all-zero-codewords
- SNR 2

Structure	CW/E	loss train	loss validation	NVS train	NVS validation	NVS test S5
FNN	256	0.073	0.091	0.353	0.454	0.508
RNN	256	0.088	0.088	0.405	0.485	0.51
FNN	512	0.079	0.09	0.424	0.462	0.498
RNN	512	0.086	0.087	0.469	0.458	0.497
FNN	1024	0.093	0.086	0.46	0.45	0.496
RNN	1024	0.1	0.087	0.491	0.45	0.498
FNN	2048	0.087	0.085	0.455	0.465	0.494
RNN	2048	0.092	0.086	0.476	0.454	0.498
FNN	4096	0.083	0.083	0.436	0.438	0.492
RNN	4096	0.089	0.086	0.461	0.446	0.498

One can see that the RNN architecture converges faster during training than the FNN architecture. In the case of 256 CW/E, RNN architecture holds a loss of ≈ 0.0876 for the penultimate 152 epochs, and a loss of ≈ 0.0875 for the last 144 epochs while the loss for FNN architecture only remains the same for less than 10 epochs. Looking at the NVS and the BER achieved on the test dataset S5, it is noticeble that

[26] To find out if more trainable parameters could improve the performance of the RNN architecture, a run was started for all CW/E where weights of W_{in2odd} and $W_{even2out}$ were trainable. This led to a degradation of the NVS on the test data set which indicates that more trainable parameters are not helpful at this point.

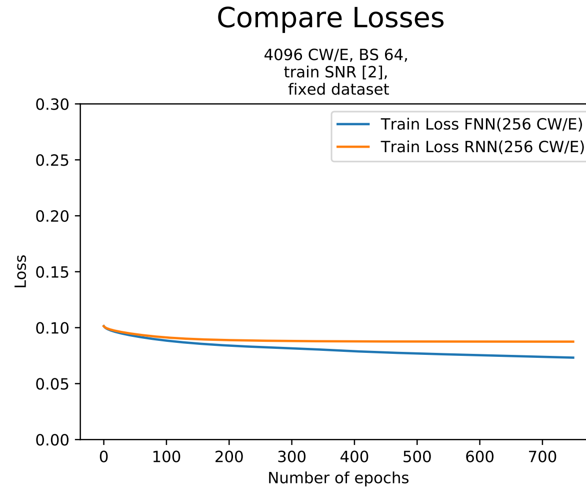


Figure 18: Comparing losses during training on RNN and FNN architecture.

the FNN slightly outperforms the RNN. The difference between FNN and RNN becomes more apparent with the two largest training datasets of 4096 CW/E. One can see from the loss that the RNN converges after only a few epochs, while the curve describing the loss under the FNN architecture elongates further. The NVS makes it clear how finely the FNN architecture can be adjusted. The NVS of the validation data set of the FNN network keeps changing slightly up to epoch 500. The NVS of the RNN network, on the other hand, fluctuates only up to epoch 100, after which there are only two changes that are noticeable. Although larger differences in performance was expected, the hpyothesis can still be confirmed.

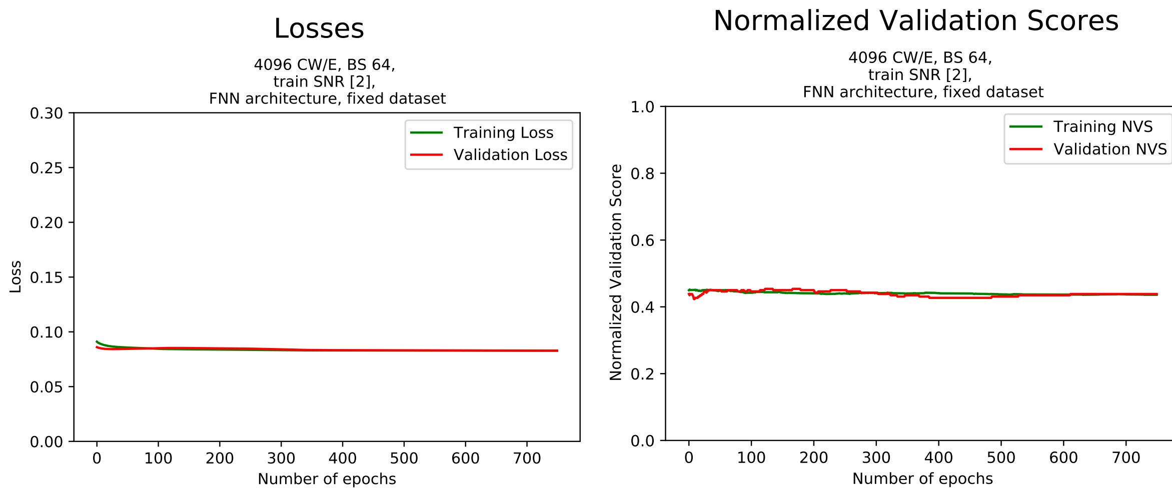


Figure 19: Loss and NVS for a FNN where the biggest dataset of 4096 CW/E was used during training.

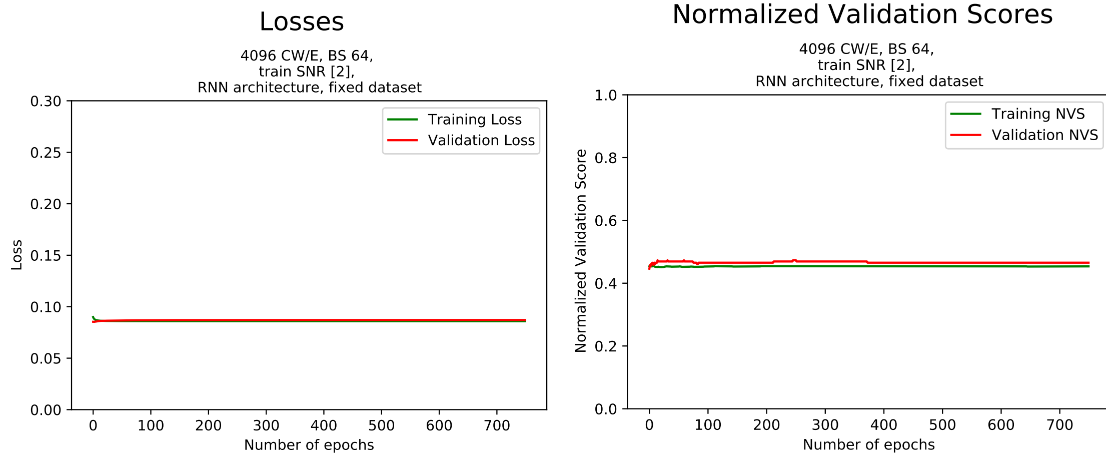


Figure 20: Loss and NVS for a RNN where the biggest dataset of 4096 CW/E was used during training.

7 Conclusion

The problem of decoding noisy messages is based on communication models like the one introduced by Shannon in "A Mathematical Theory of Communication" in 1948. [Sha48] By using a neural network for decoding, the model based approach is replaced by another technology. How can decoding benefit from this new technology? The major drawback to SPA is that circles in the Tanner graph negatively affect the flow of information (see subsection 3.3).

The experiments conducted have shown that the use of a neural network leads to better decoding by lowering the BER especially in higher signal-to-noise ratios, indicating the reduction of adverse effects of cycles. As described in subsection 2.2 the performance can be measured by comparing the achieved BER on different SNR_{dB}s. The trained network reaching a similar BER as SPA but at a lower SNR_{dB} shows that it can handle more noise. As a proxy, the trained network from experiment subsection 6.7 with 260 CW/E is used. Comparing the BERs per SNR_{dB} on the test dataset S5, SPA achieves a BER of 0.022 at SNR_{dB} 6.5 while the NND achieves a BER of 0. The NND achieves a BER of 0.023 comparable to the SPA at an SNR_{dB} of 2.5. This corresponds to an SNR_{dB} increase of 4 dB. As expected, a bigger training dataset resulted in a well trained network.

One goal of this thesis is to review parts of Agrawals thesis by putting the experiments outcome in relation to statements made there. Training with the all-zero codeword leads to results different from training with random codewords even if the same noise is used in both datasets. Even though both approaches produce good results one can not state that "[t]he performance of the decoder does not depend on the exact values of the transmitted bits, but what matters is how the channel induces patterns of errors to the transmitted signal." [Agr17, p. 47] For the example used in this thesis, batch size does not seem to play a significant role on either a fixed or OTF training dataset. Furthermore, no linear dependency between code size and batch size could be found. It is assumed that this behaviour changes for a large codes with more trainable parameters. When it comes to the SNR_{dB} it was not possible to find a single SNR_{dB} that outperformed a combination of multiple SNR_{dB}. Only Agrawals statement that "the optimal choice of the training SNR value lies somewhere in the range of -2 dB and 4 dB" could be confirmed.[Agr17, p. 49] Both SNR_{dB} 2 and SNR_{dB} 3 lead to the best NVS on the test dataset independent form the used seed. The OTF generated training dataset was especially beneficial when the number of CW/E was smaller than 5200.

Using an OTF data set, as Agrawal suggests, can thus only be encouraged. An interesting finding from this experiment is that the network trained on the OTF training data set converges after only 200 epochs. This makes it unnecessary to run the training for additional epochs even though the training data set loss still varies widely. In contrast to Agrawal's tests, which use an RNN architecture, an FNN architecture was predominantly chosen in the experiments of this thesis. Argrawal's assumption that smaller codes benefit from an FNN architecture is illustrated in the last experiment.[cf. Agr17, p. 58] For the petite example used in this thesis, the FNN architecture is clearly the better choice.
Another goal of this thesis is to improve what already exists and to introduce innovations. For this purpose, an implementation of the algorithm was developed using the Pytorch library. The code is made available to the public, which was not possible for Agrawal's implementation. The NVS has been further developed in such a way that it now covers cases where the BER achieved by SPA is zero (see subsection 6.1). A typing error in the activation function formula for odd layers has been found in Agrawals thesis and has been corrected for the use in this thesis (see subsection 4.3). Since the use of combinations of multiple SNR_{dB}s has not been well explored yet, particularly detailed experiments were conducted in this area (see subsection 6.6). It was possible to dismiss certain SNR_{dB} combinations. Finally, a new method was presented to adequately monitor the training process on an OTF data set (see subsection 6.7). The metric needs to be independent from the training dataset loss to determine convergence, as the loss varies greatly on an OTF dataset. The experience gained from all the experiments performed, shows that for a sufficiently large dataset, test set loss and validation dataset loss behave similarly. This knowledge was used to provide an easy to adapt solution by determining the convergence of the neural network based on the validation dataset loss.
During all experiments, the performance of the trained network was consistently better than that of the SPA. What are possible disadvantages of using a network instead of SPA? The NND is designed and trained for a fixed number of iterations in SPA. Decoding with the Sum-Product algorithm alone is more flexible as it works with any given number of iterations as input. On tree-structured Tanner graphs the NND can perform as good as SPA, but it's not possible to perform better. [cf. Agr17, p. 55] A further interesting point for experiments would be the systematic comparison of SPA and the neural network, where SPA finds optimal conditions such that the Tanner graph is a tree. In this experimental set-up it would be possible to measure how stable the training methodology is by checking how close the trained network comes to the optimum. Codes in real life do come with circles I am therefore convinced that it is beneficial to use a trained neural network instead of SPA because this approach is a potential improvement.

8 Repositories

8.1 Implementation

The implementation of the neural network, checkpoints produced during experiments and evaluation Jupyter Notebook can be found on github.

`https://github.com/ccc-frankfurt/spa-nnd`

8.2 References

`https://github.com/facebookresearch/HyperNetworkDecoder`

Veeresh Taranalli, CommPy: Digital Communication with Python, version 0.3.0. Available at `https://github.com/veeresht/CommPy`, 2015.

9 References

[Agr17] AGRAWAL, Navneet: *Machine Intelligence in Decoding of Forward Error Correction Codes*. Stockholm, Sweden, KTH Royal Institute of Technology School of Electrical Engineering, Diss., 2017

[Ave19] AVERY PARKINSON: *The Epsilon-Greedy Algorithm for Reinforcement Learning*. `https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d`. Version: 2019. – last visited 11.06.2020

[BMv78] BERLEKAMP, E. ; MCELIECE, R. ; VAN TILBORG, H.: On the inherent intractability of certain coding problems (Corresp.). In: *IEEE Transactions on Information Theory* 24 (1978), May, Nr. 3, S. 384–386. `http://dx.doi.org/10.1109/TIT.1978.1055873`. – DOI 10.1109/TIT.1978.1055873. – ISSN 1557–9654

[CTWE18] CHOI, Kristy ; TATWAWADI, Kedar ; WEISSMAN, Tsachy ; ERMON, Stefano: NECST: Neural Joint Source-Channel Coding. In: *CoRR* abs/1811.07557 (2018). `http://arxiv.org/abs/1811.07557`

[DFB14] DECLERCQ, David ; FOSSORIER, Marc ; BIGLIERI, Ezio: *Channel Coding: Theory, Algorithms, and Applications*. Academic Press, 2014

[GRS19] GURUSWAMI, Venkatesan ; RUDRA, Atri ; SUDAN, Madhu: *Essential Coding Theory*. 2019. – Based on lecture notes from coding theory courses taught by Venkatesan Guruswami at University at Washington and CMU; by Atri Rudra at University at Buffalo, SUNY and by Madhu Sudan at Harvard and MIT.

[Het18] HETTERICH, Dr. S.: *Mathematik für die Informatik II - Numerik und Diskrete Mathematik*. Frankfurt am Main, Juli 2018. – Skript SoSe18

[Hil86] HILL, Raymond: *A First Course in Coding Theory*. Oxford University Press, 1986

[HSG+19] HELMLING, Michael ; SCHOLL, Stefan ; GENSHEIMER, Florian ; DIETZ, Tobias ; KRAFT, Kira ; RUZIKA, Stefan ; WEHN, Norbert: *Database of Channel Codes and ML Simulation Results*. `www.uni-kl.de/channel-codes`, 2019

[HZL+19] HUANG, Lingchen ; ZHANG, Huazi ; LI, Rong ; GE, Yiqun ; WANG, Jun: AI Coding: Learning to Construct Error Correction Codes. In: *CoRR* abs/1901.05719 (2019). `http://arxiv.org/abs/1901.05719`

[Joh] JOHNSON, Sarah J.: Introducing Low-Density Parity-CheckCodes. In: *School of Electrical Engineering and Computer Science*

[Lin10] LINDELL, Yehuda: *Introduction to Coding Theory Lecture Notes*. Israel, 2010

[Mac03] MACKAY, David J.: *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003

[NML+17] NACHMANI, Eliya ; MARCIANO, Elad ; LUGOSCH, Loren ; GROSS, Warren J. ; BURSHTEIN, David ; BE'ERY, Yair: Deep Learning Methods for Improved Decoding of Linear Codes. In: *CoRR* abs/1706.07043 (2017). `http://arxiv.org/abs/1706.07043`

[o.V16] O.V. CREONIC GMBH: *LDPC Decoder Applications*. `https://www.ldpc-decoder.com`. Version: 2016. – last visited 02.12.2019

[o.V20] O.V. ACADEMIA.EDU: *LDPC codes*. https://www.academia.edu/Documents/in/LDPC_codes. Version: 2020. – last visited 15.01.2020

[RU08] RICHARDSON, Tom ; URBANKE, Rüdiger: *Modern Coding Theory*. Cambridge University Press, 2008

[Sö19] SÖDER, Günter: *Grundlegendes zu den Low–density Parity–check Codes*. https://www.lntwww.de/Kanalcodierung/Grundlegendes_zu_den_Low%E2%80%93density_Parity%E2%80%93check_Codes. Version: 2019. – last visited 25.10.2019

[sam] SAMISHAWL: *Epsilon-Greedy Algorithm in Reinforcement Learning*. https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/. – last visited 11.06.2020

[San18] SANYAM KAPOOR: *Policy Gradients in a Nutshell*. https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d. Version: 2018. – last visited 09.06.2020

[Sch12] SCHNITGER, Prof. Dr. G.: *Theoretische Informatik 1*. Frankfurt am Main, 2012. – Skript WiSe2012

[Sch17] SCHNITGER, Prof. Dr. G.: *Diskrete Modellierung - Eine Einführung in grundlegende Begriffe und Methoden der Theoretischen Informatik*. Frankfurt am Main, September 2017. – Skript WiSe2017

[Sch18] SCHNITGER, Prof. Dr. G.: *Computational Learning Theory*. Frankfurt am Main, Juli 2018. – Skript SoSe2018

[Sha48] SHANNON, Claude E.: A Mathematical Theory of Communication. In: *Bell System Technical Journal* (1948)

[Sho03] SHOKROLLAHI, Amin: LDPC codes: An introduction. In: *Digital Fountain, Inc., Tech. Rep* 2 (2003), S. 17

[SMM13] SAHA, Arijit ; MANNA, Nilotpal ; MANDAL, Surajit: *Information Theory, Coding and Cryptography*. Pearson India, 2013

[TKLJ07] TRAORE, Nana ; KANT, Shashi ; LINDSTROM-JENSEN, Tobias: Message Passing Algorithm and Linear Programming Decoding for LDPC and Linear Block Codes. In: *Aalborg University - Institute of Electronic Systems* (2007)

[TL10] TAN, P. ; LI, J.: Efficient Quantum Stabilizer Codes: LDPC and LDPC-Convolutional Constructions. In: *IEEE Transactions on Information Theory* 56 (2010), Jan, Nr. 1, S. 476–491. http://dx.doi.org/10.1109/TIT.2009.2034794. – DOI 10.1109/TIT.2009.2034794. – ISSN 1557–9654

[TTA+17] TOMLINSON, M. ; TJHAI, Cen ; AMBROZE, Marcel ; AHMED, M. ; JIBRIL, Mubarak: *Error-Correction Coding and Decoding*. 2017. http://dx.doi.org/10.1007/978-3-319-51103-0. http://dx.doi.org/10.1007/978-3-319-51103-0

[Yat09] YATES, Randy: *A Coding Theory Tutorial*. August 2009