UNIVERSITÀ DI PARMA

Department of Engineering and Architecture

Master of Science in Communication Engineering

Exploring Machine Learning Algorithms for Decoding Linear Block Codes

Advisor:

Prof. Riccardo Raheli

Co-Advisors:

Prof. Henry D. Pfister

Dr. Christian Häger

Dr. Marco Martalò

Thesis presented by:

Fabrizio Carpi

Academic Year 2017-2018

Thesis presented at University of Parma (Italy) - October 12, 2018.

For correspondance: fabri.carpi93@gmail.com

To my beloved family

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ADAM	Adaptive Moment estimator
AI	Artificial Intelligence
APP	A Posteriori Probability
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BF	Bit-Flipping
BI-AWGN	Binary AWGN
BP	Belief Propagation
BPSK	Binary Phase-Shift Keying
BSC	Binary Symmetric Channel
CER	Codeword Error Rate
CN	Check Node
DL	Deep Learning

DQN	Deep Q-Network
FC	Fully Connected
FG	Factor Graph
GD	Gradient Descent
LDPC	Low-Density Parity-Check
LLR	Log-Likelihood Ratio
MAP	Maximum A Posteriori
MDP	Markov Decision Process
ML	Maximum-Likelihood
MWPC	Minimum Weight Parity Check
NLP	Natural Language Processing
NN	Neural Network
no-op	no operation
PC	Parity Check
PDF	Probability Density Function
PDF	Probability Density Function
ReLU	Rectified Linear Unit
RL	Reinforcement Learning

RM	Reed-Muller
RV	Random Variable
SGD	Stochastic Gradient Descent
SNR	Signal to Noise Ratio
SPA	Sum Product Algorithm
VN	Variable Node
WBF	Weighted Bit-Flipping

Chapter 1

Introduction

"We are all now connected by the Internet,

like neurons in a giant brain."

Stephen Hawking

Machine learning is a subfield of Artificial Intelligence (AI), which has become one of the fastest growing areas in many technological fields during these years. In general, it provides to computers the capability to learn from data, without being explicitly programmed for a specific task. Nowadays, many applications are powered by AI technologies. Some of the most famous examples are speech recognition [3] and computer vision [4]. Moreover, most of AI techniques, are inspired by psychology and neuroscience. Among them, Reinforcement Learning (RL) deals with situations where a sequence of decisions has to be taken. For example, it is successfully applied to games and allows to beat human performances [5], [6]. Some interesting talks about machine learning and RL can be found in [7], [8].

In every AI technique, data and interactions are fundamental aspects for the learning

process. Inspired by behaviourists, learning from interaction is the main idea underlying almost all theories of learning and intelligence. For example, animals and humans, since the first moments of their life, learn to act by the interaction with the surrounding environment. Furthermore, the more experience (data) they have, the better they behave.

From a high-level point of view, machine learning algorithms can be divided into the following three cases [7].

1. *Supervised learning* consists of learning with a teacher, who knows the right answer that we have to learn. The model has to produce the output answer, given some input data. The performance is evaluated in term of errors, with respect to the teacher answer.

2. *Unsupervised learning* consists of learning without a teacher. The model has to learn by itself what are the good answers, given some input data.

3. *Reinforcement learning* consists of learning with a critic, who gives an evaluation (reward/penalty) after each action is taken. In general, RL is searching the best action by trial-and-errors, and remembering it for each situation. Then, RL is about caching the best search results for a specific situation, so that it does not have to keep searching.

Alongside the recent machine learning growth, telecommunications is another field where scientists spend efforts to design algorithm for achieving better performance. Since the foundation of information theory in 1948, by Claude Shannon, many coding and decoding algorithms have been developed. Nowadays, as we are living in the age of Big Data, faster and more reliable communications are needed. Then, we can merge the transmission rate-hunger with the novel machine learning approaches, in order to optimize well-known communication architectures.

In this thesis, we explore machine learning algorithms for communication problems. As discussed in [9], [10], there are some aspects that have to be clarified when we apply machine learning to communications.

- *Training data*: in general the channel model is assumed to be known, so that we can generate an infinite supply of data for the learning process.

- *Exploit domain knowledge*: having the channel model allows us to implement optimal or close-to-optimal solutions in many cases. Then, we can optimize a parameterized version of the computation graph, exploiting our domain knowledge about the problem, instead of optimizing a black-box architecture. This may, for example, yield approximate solutions with lower complexity.

- *No domain knowledge*: sometimes the channel model does not exist, or it is too complex to be used in practical scenarios. In this case, if it is easy to collect real-world data and the analyzed phenomenon does not vary rapidly, we can adopt a machine learning algorithm (supervised, unsupervised or reinforcement learning) to produce a parameterized model that solves the communication problem.

In particular, in our work we investigate the use of machine-learning-inspired decoding algorithms for linear block codes, such as Reed-Muller codes.

The first goal of the thesis is to optimize Belief-Propagation (BP) iterative decoding through supervised learning. As proposed in [11], we can parameterize the BP computation graph and learn these new parameters optimizing the system performance. So, we exploit domain knowledge about this well-known decoding algorithm to improve performance. The decoder proposed in [11] has a number of parameters in the order of the number of edges of the Tanner graph, while our decoder has a number of parameters in

the order of the number of BP iterations. So, we propose an optimized solution with much lower complexity. Part of our work is presented in [12].

Moreover, we propose a novel RL-inspired method for bit-flipping decoding. We approach the decoding problem as a game, exploiting the interesting results achieved in the past years [5], [6]. One can view the RL decoder as a player of a game where the goal is to correct the error pattern. At first, the RL architecture has no prior knowledge about the code, but it learns to act by trial and errors, with surprisingly good performance. In particular, our performance results match the standard decoding algorithms, and beat them in some scenarios.

The rest of the thesis is structured as follows.

- Chapter 2 provides the basic background and notation for linear block codes.
- Chapter 3 introduces machine learning techniques.
- Chapter 4 treats the first experimental part, about BP decoding optimization of Reed-Muller codes through machine learning techniques introduced in Chapter 3.
- Chapter 5 introduces the basic concepts of RL.
- Chapter 6 treats the second experimental part, about the application of RL to bit-flipping decoding or Reed-Muller codes.

Chapter 2

Linear Block Codes

"The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point."

Claude Shannon

The main objective of digital communication systems is to reproduce the source message at the receiver side. In this scenario, errors, which are due to the distortions introduced by the propagation medium and equipment, can affect the received message. Error-correcting codes are used by communication systems to improve reliability. Among these, linear block codes are those for which any linear combination of codewords is also a codeword. In our work, we consider only binary codes, i.e., the symbol alphabet is $\mathcal{X} = \{0, 1\}$.

This chapter reviews the most important concepts about linear block codes and introduces our notation. In Section 2.1, we briefly recap the fundamentals of linear block codes. In Section 2.2, we define the matrix and the graphical representation of codes, in-

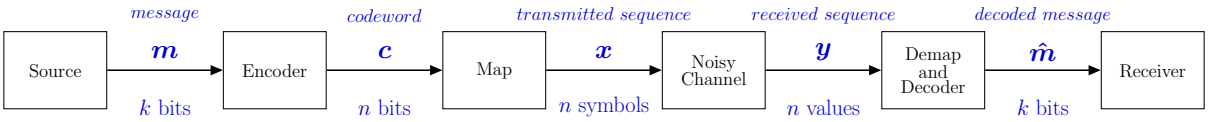


Figure 2.1: Coding scheme.

troducing factor graphs. Then, we define the communication channel in Section 2.3. The Belief-Propagation (BP) algorithm is treated in Section 2.4. In Section 2.5, we give an overview of Reed-Muller codes, which will be used in our experiments. This introductive chapter about coding is based on [2], [13], [14].

2.1 Fundamentals

The coding scenario is illustrated in Figure 2.1. We assume that the binary information sequence, called *message*, is composed of k bits, so there are 2^k possible distinct messages. At the channel encoder, each message is encoded into a longer binary vector, called *codeword*, of n code bits ($n > k$). There are 2^k distinct codewords, one for each distinct message. This set of codewords form an (n, k) block code. The encoder adds $n - k$ *redundant* bits to each message. These bits do not add new information, but provide the capability of *detecting* and *correcting* errors. The aim of the encoder is to map each message $\boldsymbol{m}$ to a codeword $\boldsymbol{c}$ in a sufficiently robust way to recover distortions due to noise. Then, the codeword is mapped to the transmitted sequence $\boldsymbol{x}$, that is sent through the channel. At the receiver side, the decoder processes the received sequence $\boldsymbol{y}$, which is a noisy version of $\boldsymbol{x}$ and exploits the code structure to decide the decoded message $\hat{\boldsymbol{m}}$.

A binary block code of length n with 2^k codewords is called *linear* if its 2^k codewords form a k-dimensional subspace of the vector space V of all n-tuples over $\mathbb{F}_2$, where $\mathbb{F}_2$ is the Galois Field of two elements. In other words, every codeword can be expressed as

linear combination of k codewords that form a basis for V.

The *code rate* $R = k/n$ represents the ratio of information bits over code bits.

2.2 Code Representation

Let $\mathcal{C}$ be a linear block code (n, k). It can be represented by a generator matrix $\boldsymbol{G}$, of size $k \times n$, or, equivalently, by a Parity Check (PC) matrix $\boldsymbol{H}$, of size $m \times n$, where m is the number of parity check equations to be satisfied. Let $\boldsymbol{m} = (m_0, m_1, ..., m_{k-1})^T$ be the message to be encoded. The codeword $\boldsymbol{c} = (c_0, c_1, ..., c_{n-1})^T$ is given by

$$\boldsymbol{c} = \boldsymbol{G}\,\boldsymbol{m}.$$

The PC matrix $\boldsymbol{H}$ is defined as a generator matrix for the dual code $\mathcal{C}^{\perp}$, which is the orthogonal complement of $\mathcal{C}$ in $\mathbb{F}_2$, i.e.,

$$\mathcal{C}^{\perp} = \{\boldsymbol{h} \in \mathbb{F}_2^n | \boldsymbol{h}^T \boldsymbol{c} = 0 \ \ \forall \boldsymbol{c} \in \mathcal{C}\}.$$

This means that $\boldsymbol{G}\,\boldsymbol{H}^T = \boldsymbol{O}$, where $\boldsymbol{O}$ is a $k \times m$ all-zero matrix. In general, neither the generator or the PC matrices are unique. We can add rows that are linear combinations of the other ones. Furthermore, left multiplication by any $(n-k) \times (n-k)$ matrix gives another PC matrix for the same code. Given a received word $\boldsymbol{y}$, the vector $\boldsymbol{s} = \boldsymbol{H}\,\boldsymbol{y}$ is called the *syndrome*. Hence, any codeword $\boldsymbol{c} \in \mathcal{C}$ satisfies $\boldsymbol{s} = \boldsymbol{H}\,\boldsymbol{c} = \boldsymbol{0}$, where $\boldsymbol{0}$ is the all-zero vector of size m.

Another important definition in coding theory is the Hamming distance. Let $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}_2^n$,

then the *Hamming distance* is the number of bits at equal position that are different

$$d(\boldsymbol{a}, \boldsymbol{b}) = \sum_{i=1}^{n} [1 - \delta_{a_i b_i}]$$

where $\delta_{a_i b_i}$ is the Kronecker delta for the bits a_i and b_i, which are the i-th bits of the binary vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. In general, the *minimum distance* $d^{\mathcal{C}}_{\min}$ of $\mathcal{C}$ is the minimum distance between any two codewords. For a linear code, it simplifies to

$$d^{\mathcal{C}}_{\min} = \min_{\boldsymbol{a}, \boldsymbol{b} \in \mathcal{C}, \boldsymbol{a} \neq \boldsymbol{b}} d(\boldsymbol{a}, \boldsymbol{b}) = \min_{\boldsymbol{b} \in \mathcal{C}, \boldsymbol{b} \neq \boldsymbol{0}} d(\boldsymbol{0}, \boldsymbol{b}).$$

Another method to represent linear block codes is through Factor Graphs (FGs). FGs are bipartite graphs representing the factorization of a function [15]. Nodes may be separated into two types, with edges connecting only nodes of different types. FGs are widely used in applications where there is a joint distribution and the marginals have to be computed.

The FG of a linear block code is called a *Tanner graph.* The two types of nodes in a Tanner graph are the *Variable Nodes* (VNs), or *bit nodes*, and the *Check Nodes* (CNs), or *constraint nodes.* The Tanner graph of a code is defined as follows. CN i is connected to VN j whenever $h_{ij} = 1$, where h_{ij} is the element of the PC matrix in the i-th row and j-th column. There are m CNs in a Tanner graph, one for each PC equation, and n VNs, one for each code bit.

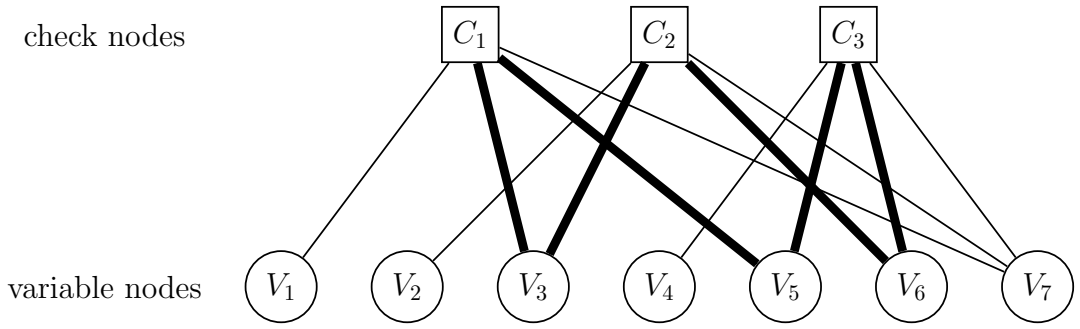


Figure 2.2: Tanner graph for the Hamming (7,4) code. Highlighted edges show a length-6 cycle in the graph. Moreover, the bits adjacent to highlighted edges form a weight-3 codeword.

□ **Example.** The PC matrix of the (7,4) Hamming code [2] is

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

The corresponding Tanner graph is illustrated in Figure 2.2.

■

2.3 Communication Channel

A communication channel is the medium between the transmitter and the receiver, as shown in Figure 2.1. Given an input Random Variable (RV) $X \in \mathcal{X}$ and an output RV $Y \in \mathcal{Y}$, the sets $\mathcal{X}$ and $\mathcal{Y}$ are called *transmitter* and *receiver alphabet*, respectively. The transfer function is described by the probability function $p(y|x) = \Pr(Y = y|X = x)$.

We can assume equiprobable source, if $\Pr(X_i = 0) = \Pr(X_i = 1) = 0.5$. Given a

transmitted sequence $\boldsymbol{x} \in \mathcal{X}^n$ and a received sequence $\boldsymbol{y} \in \mathcal{Y}^n$, starting from the log-ratio of *A Posteriori* Probability (APP), i.e., $p(x_i|y_i)$, the *Log-Likelihood Ratio* (LLR) L_i of the i-th bit is defined as

$$\begin{aligned} \log[\text{APP}] &= \log \frac{\Pr(X_i = 0|Y = y_i)}{\Pr(X_i = 1|Y = y_i)} \qquad (2.1) \\ &= \log \frac{\Pr(Y = y_i|X_i = 0)\ \Pr(X_i = 0)\ \Pr(Y = y_i)}{\Pr(Y = y_i|X_i = 1)\ \Pr(X_i = 1)\ \Pr(Y = y_i)} \\ &= \log \frac{\Pr(Y = y_i|X_i = 0)}{\Pr(Y = y_i|X_i = 1)} = L_i \qquad (2.2) \end{aligned}$$

The LLR is a powerful metric for the decoding procedure. At the receiver side, $\text{sign}(L_i)$ yields the decision of the transmitted bit, while $|L_i|$ expresses the *reliability* of this decision. For *hard-decision* decoding only the sign of the LLRs is taken into account. On the other hand, *soft-decision* decoding algorithms exploit also the magnitude of the LLRs.

We now introduce two commonly used channel models, which will be used in our experiments.

2.3.1 BSC

The Binary Symmetric Channel (BSC) is a binary-input binary-output channel and it is represented in Figure 2.3. It is characterized by $\mathcal{X} = \{0, 1\}$, $\mathcal{Y} = \{0, 1\}$, and

$$p(y = 1|x = 0) = p(y = 0|x = 1) = p$$
$$p(y = 1|x = 1) = p(y = 0|x = 0) = 1 - p$$

where p is known as the *crossover* probability. In other words, a bit is received correctly with probability $1 - p$. This is the simplest channel that we can use to model errors.

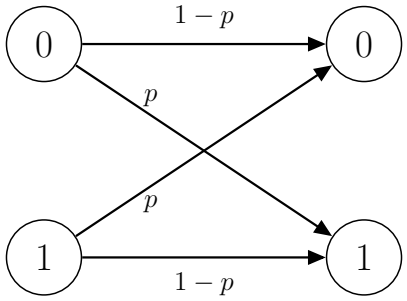


Figure 2.3: Block diagram for the BSC.

From equation (2.2), for $0 \leq p < 0.5$, the LLR for this channel can be computed as

$$L_i = \begin{cases} -A & \text{if } y_i = 1 \\ +A & \text{if } y_i = 0 \end{cases}$$

where $A = \log(1-p) - \log(p)$. Equivalently, the LLR for BSC can be expressed as

$$L_i = (-1)^{y_i} \log \frac{1-p}{p}.$$

2.3.2 BI-AWGN Channel

In this case, we assume that the bits are modulated with Binary Phase-Shift Keying (BPSK), so that the information bits are mapped as follows: $0 \longmapsto 1$ and $1 \mapsto -1$. The block diagram of the Binary-Input Additive White Gaussian Noise (BI-AWGN) Channel is shown in Figure 2.4. Given a transmitted symbol $x \in \mathcal{X} = \{-1, 1\}$, the observation $y \in \mathcal{Y} = \mathbb{R}$ is defined as $y = x + n$, where the noise n has Gaussian distribution with zero-mean and variance σ^2. The conditional channel distribution are shown in Figure 2.5.

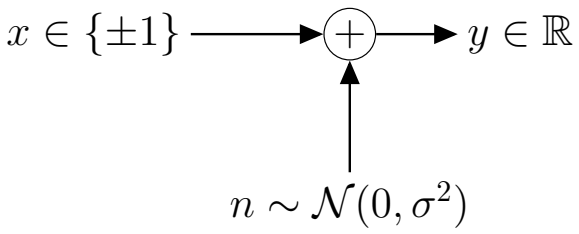


Figure 2.4: System model for the BI-AWGN channel.

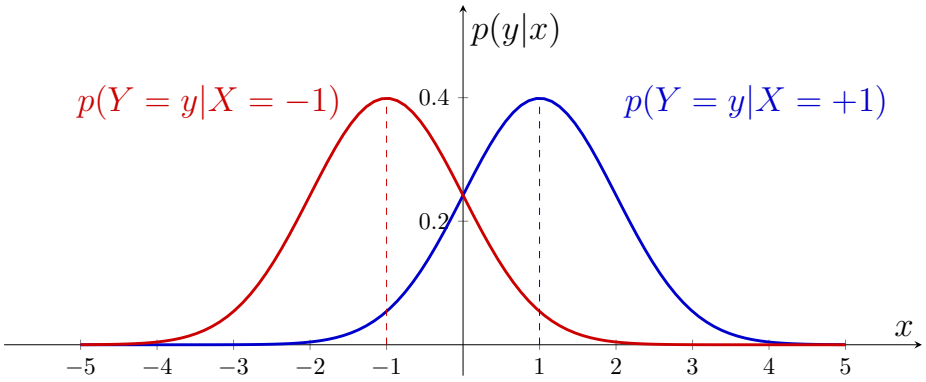


Figure 2.5: Conditional distributions for the BI-AWGN channel.

Thus, the channel is described by

$$p(y|x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-x)^2}{2\sigma^2}}.$$

Starting from the general case of equation (2.2), the LLR of the i-th bit is

$$\begin{aligned} L_i &= \log \frac{\Pr(y_i|x_i = 1)}{\Pr(y_i|x_i = -1)} \\ &= \log \frac{\exp[-(y_i - 1)^2/2\sigma^2]}{\exp[-(y_i + 1)^2/2\sigma^2]} = \frac{2\, y_i}{\sigma^2} \end{aligned} \tag{2.3}$$

where x_i is the transmitted BPSK symbol and y_i is the corresponding observation. Therefore, to compute (2.3), the receiver needs to know the value of σ^2.

In this channel, we can exploit soft-information from the LLRs, i.e., $|L_i|$. Otherwise, we can consider only $\text{sign}(L_i)$ to decide on the received bits (hard-decision), restoring to the BSC.

In our experiments, we refer to the Signal-to-Noise Ratio (SNR) as

$$\frac{E_s}{N_0} = \frac{\mathbb{E}\left[x^2\right]}{\mathbb{E}\left[n^2\right]} = \frac{1}{\sigma^2}.$$

2.4 Belief-Propagation Decoding Algorithm

At the receiver side, the decoder wishes to minimize the probability of a codeword error. This is equivalent to maximizing the *a posteriori* probability $p(\boldsymbol{x}|\boldsymbol{y})$. The Maximum A Posteriori (MAP) decoder chooses the codeword $\hat{\boldsymbol{c}} \in \mathcal{C}$ that maximizes the posterior probability, i.e.,

$$\hat{\boldsymbol{c}}_{\text{MAP}} = \arg\max_{\boldsymbol{c}\in\mathcal{C}} p(\boldsymbol{c}|\boldsymbol{y}) \tag{2.4}$$

where $\boldsymbol{c}$ is the transmitted codeword and $\boldsymbol{y}$ is the observation.

Assuming that the source produces equiprobable information and since the code produces a 1:1 mapping of the messages, $p(\boldsymbol{c}) = 1/|\mathcal{C}|$, where $|\mathcal{C}|$ is the cardinality of the code. Hence (2.4) simplifies to

$$\begin{aligned}\hat{\boldsymbol{c}}_{\text{MAP}} &= \arg\max_{\boldsymbol{c}\in\mathcal{C}} p(\boldsymbol{c}|\boldsymbol{y}) \\ &= \arg\max_{\boldsymbol{c}\in\mathcal{C}} p(\boldsymbol{y}|\boldsymbol{c})\frac{p(\boldsymbol{c})}{p(\boldsymbol{y})} \\ &= \arg\max_{\boldsymbol{c}\in\mathcal{C}} p(\boldsymbol{y}|\boldsymbol{c}) = \hat{\boldsymbol{c}}_{\text{ML}}\end{aligned} \tag{2.5}$$

where we used the fact that $p(\boldsymbol{c})$ is uniform over all codewords. Equation (2.5) is the

expression for the codeword returned by the *Maximum-Likelihood* (ML) decoder, i.e., that chooses the most likely codeword given $\boldsymbol{y}$.

The MAP (and ML) decoding strategy can be computationally challenging, since the equipment complexity and time required to decode the noisy received sequence may be not suitable for practical implementations. Thus, iterative decoding algorithms were proposed to achieve high data rates and good error rates with reasonable equipment.

Belief-Propagation (BP) is an iterative message-passing algorithm that provides a method to approximate the marginal distribution of the bit nodes, based on the code's Tanner graph. It is also known as the Sum-Product Algorithm (SPA) and it was introduced by Gallager [14] in 1960 in his seminal work on Low-Density Parity-Check (LDPC) codes.

The Tanner graph of a code acts as map for the iterative decoder in the following way. Each of the nodes acts as a local processor and each edge acts as a bus that conveys information from a given node to each of its neighbors. The information conveyed is typically probabilistic, e.g., the LLR. The decoder is initialized by n LLRs from the channel, which are received by the n VNs. At the beginning of each half-iteration, each VN processor takes inputs from the channel and each of its neighboring CNs. From these, it computes outputs for each one of its neighboring CN processors. In the next half-iteration, each CN processor takes inputs from each of its neighboring VNs, and from these, it computes outputs for each one of its neighboring VN processors. The VN$\leftrightarrow$CN iterations continue until a codeword is found or a pre-determined maximum number of iterations has been reached.

The effectiveness of the iterative decoder depends on the Tanner graph of the code. Observe the six bold edges in Figure 2.2, which form a closed path is called a *cycle*. The *length of a cycle* is equal to the number of edges which form the cycle, so the length of the

cycle in Figure 2.2 is 6. Also, the bits adjacent to the highlighted edges, i.e. $[V_3, V_5, V_6]$, form a weight-3 codeword. The minimum cycle length in a given bipartite graph is called the graph's *girth*. The girth of the Tanner graph for the example code is 6. The shortest possible cycle in a bipartite graph has length 4, and such cycles manifest themselves in the $\boldsymbol{H}$ matrix as four 1s that lie on the four corners of a rectangular sub-matrix of $\boldsymbol{H}$.

The core of SPA is formed by the messages that are passed between nodes. The main principle is that each node forwards to its neighbors all the incoming messages, except the one received from each of them. This is known as the *extrinsic information*. The objective is to maximize the posterior probabilities symbol-wise. In the next paragraphs we recap the main idea of BP, referring to [2], Chapter 5.

Let $\boldsymbol{x} = (x_1, ..., x_n)$ be the transmitted codeword and $\boldsymbol{y} = (y_1, ..., y_n)$ be the received sequence. We are interested in computing the APP that a transmitted bit x_j equals "1", i.e., $\Pr(x_j = 1|\boldsymbol{y})$. Then, the LLR is

$$L_j = L(x_j|\boldsymbol{y}) = \log \frac{\Pr(\boldsymbol{y}|x_j = 0)}{\Pr(\boldsymbol{y}|x_j = 1)}.$$

Each VN j receives messages L_j from the channel and from all its neighbor CNs; when computing message $L_{j\to i}$ to CN i, the message $\hat{L}_{i\to j}$ from CN i is not used, according to the extrinsic information principle. The same holds for each CN i, but they only receive messages from the neighbor VNs. Then, VNs and CNs work iteratively to estimate $L(x_j|\boldsymbol{y})$ for every bit node, $j = 1, 2, ...n$. When the cycles are large, the estimates will be very accurate and the decoder will have near-optimal (MAP) performance.

The core messages of the Gallager SPA decoder are:

$$L_{j\to i} = L_j + \sum_{i' \in N(j)\setminus\{i\}} \hat{L}_{i'\to j} \tag{2.6}$$

$$\hat{L}_{i\to j} = 2\ \tanh^{-1}\left(\prod_{j'\in N(i)\setminus\{j\}} \tanh\left(\frac{1}{2}L_{j'\to i}\right)\right) \tag{2.7}$$

where $N(k)$ denotes the set of neighbors of node k and L_j are the LLRs initialized according to the channel model (section 2.3). The iteration-stopping criterion is when the estimated codeword $\hat{\boldsymbol{c}}$ satisfies all the PC equations, i.e. $\hat{\boldsymbol{c}}\boldsymbol{H}^{\boldsymbol{T}} = \boldsymbol{0}$, or when the number of iterations reaches a maximum threshold. The estimated codeword $\hat{\boldsymbol{c}}$ is decided based on the sign of the output LLR information from the last iteration.

Equation (2.7) can be numerically challenging, but it can be simplified as follows. First, factor $L_{j\to i} = \alpha_{ji}\beta_{ji}$ into its sign $\alpha_{ji} = \text{sign}(L_{j\to i})$ and magnitude $\beta_{ji} = |L_{j\to i}|$ (or *bit value* and *bit reliability*). Exploiting the odd symmetry of $\tanh(\cdot)$, equation (2.7) we can be rewritten as

$$\tanh\left(\frac{1}{2}L_{i\to j}\right) = \left(\prod_{j'\in N(i)\setminus\{j\}} \alpha_{j'i}\right)\cdot\left(\prod_{j'\in N(i)\setminus\{j\}} \tanh\left(\frac{1}{2}\beta_{j'i}\right)\right)$$

from which

$$L_{i\to j} = \left(\prod_{j'} \alpha_{j'i}\right)\cdot 2\ \tanh^{-1}\left(\log^{-1}\sum_{j'}\log\left(\tanh\left(\frac{1}{2}\beta_{j'i}\right)\right)\right)$$

which further simplifies to

$$L_{i\to j} = \left(\prod_{j'} \alpha_{j'i}\right)\cdot\phi\left(\sum_{j'}\phi\left(\beta_{j'i}\right)\right)$$

where

$$\phi(x) = -\log\left[\tanh(x/2)\right] = \log\frac{e^x+1}{e^x-1}.$$

The function $\phi(x)$ is plotted in Figure (2.6). Note that $\phi^{-1}(x) = \phi(x)$ when $x > 0$.

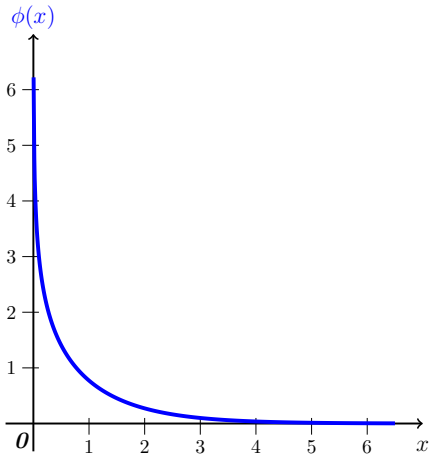


Figure 2.6: $\phi(x)$ graph.

The BP algorithm is listed in Algorithm 2.1 and this implementation will be used for the experiments in Chapter 4.

2.5 Reed-Muller Codes

For our experiments described in Chapters 4 and 6, we focus on Reed-Muller (RM) codes, which are an important class of linear block codes, because of their richness in algebraic and geometric structure. They were first introduced by Muller [16] and Reed provided a decoding algorithm in 1954 [17]. We consider only the binary version, but modern generalizations to q-ary codes exist. In particular, we use the RM(2,5) code and the interesting results of [18] to build the parity check matrix $\boldsymbol{H}$. We define the main notation in the next paragraphs. For a deeper background the reader can consult [19].

For each positive integer m and each integer r with $0 \leq r \leq m$, there is an r-th order Reed-Muller Code RM(r,m) of length $n = 2^m$. Each codeword in RM(r,m) is defined by evaluating a multivariate polynomial $f \in \mathbb{F}_2[x_1, ..., x_m]$ of degree at most r at all points in $\mathbb{F}_2^m$. The code RM(r,m) has minimum distance $d_{\min} = 2^{m-r}$ and size $k = \binom{m}{0} + ... + \binom{m}{r}$.

Algorithm 2.1 BP algorithm.

1. **Initialization:** for all VNs $j = 1, ..., n$, initialize the LLRs L_j from the received vector. Then, for all $i = 1, ...m$ for which $h_{ij} = 1$, set $L_{j \to i} = L_j$.

2. **CN update:** compute outgoing CN messages $\hat{L}_{j \to i}$ for each CN:

$$\hat{L}_{i \to j} = \prod_{j' \in N(i) - \{j\}} \alpha_{ji} \quad \phi \left(\sum_{j' \in N(i) \setminus \{j\}} \phi(\beta_{ji}) \right) \tag{2.8}$$

   where $\alpha_{ji} = sign(L_{j \to i})$, $\beta_{ji} = |L_{j \to i}|$ and $\phi(x) = log \frac{e^x + 1}{e^x - 1}$

3. **VN update:** compute outgoing VN messages $L_{j \to i}$ for each VN:

$$L_{j \to i} = L_j + \sum_{i' \in N(j) \setminus \{i\}} \hat{L}_{i' \to j} \tag{2.9}$$

4. **Total LLR:** for $j = 0, 1, ..., n - 1$, compute

$$L_j^{total} = L_j + \sum_{i \in N(j)} \hat{L}_{i \to j} \tag{2.10}$$

5. **Stopping criteria:** according to BPSK mapping, for $j = 0, 1, ..., n - 1$, set

$$\hat{c_j} = \begin{cases} 1 & if\ L_j^{total} \leq 0 \\ 0 & else \end{cases} \tag{2.11}$$

   to obtain $\hat{\boldsymbol{c}}$. If $\hat{\boldsymbol{c}} \boldsymbol{H}^{\boldsymbol{T}} = \boldsymbol{0}$ or the number of iterations equals the maximum limit, stop; else, go to Step 2.

This Algorithm is adapted from [2].
L_j is the LLR for VN j according to the channel model
$L_{j \to i}$ is the message computed by VN j and destined to CN i
$\hat{L}_{i \to j}$ is the message computed by CN i and destined to VN j
L_j^{total} is the marginal output for VN j
$\hat{c_j}$ is the estimated bit for VN j

Table 2.1: RM codes used in our experiments: RM(2,5) and RM(3,7).

Characteristic for RM(r,m) code	RM(2,5)	RM(3,7)
message length $k = \binom{m}{0} + ... + \binom{m}{r}$	16	64
codeword length $n = 2^m$	32	128
code rate $R = k/n$	0.5	0.5
minimum distance $d_{min} = 2^{m-r}$	8	16
number of MWPCs	620	94,488

In our experiments, we use the RM(2,5) and RM(3,7) codes described in Table 2.1. The main idea, from [20], is to apply iterative decoding to a highly-redundant PC matrix that contains only the Minimum-Weight Parity Checks (MWPCs). Note that highly-symmetric codes such as RM codes can have a very large number of MWPCs. For the RM codes, we denote as $\boldsymbol{H}_{\text{oc}}$ the *overcomplete* matrix, formed by all the MWPCs [18]. For example, the size of $\boldsymbol{H}_{\text{oc}}$ for the RM(2,5) is 620×32, while for the RM(3,7) it is 94488×128. Then, we may use the entire $\boldsymbol{H}_{\text{oc}}$ or a fraction of its rows for the decoding algorithms described in Chapters 4 and 6.

Chapter 3

Machine Learning

"Artificial Intelligence is the new electricity."

Andrew Ng

We are in the age of Big Data: an enormous quantity of data is generated every second by social networks, navigation systems, searches, video streaming, trading and so on. Almost every action that we take can be associated with digital information and stored in some database. Machine learning is a tool that allows us to analyze data, detect patterns, and then make prediction of future observations. It is a branch of Artificial Intelligence, which is the science of getting computers to act in desirable ways without being explicitly programmed. Usually, this class of algorithms is used when it is too complex or expensive to directly program a machine to perform a specific task. The key idea for most machine learning techniques is to have a large amount of data in order to train the model with enough sampled experiences. The objective is that the resulting system generalizes well to future observations.

We already deal with many machine learning applications in our everyday life, such as

computer vision, natural language processing, speech recognition, online recommendations and data mining. Some examples are modern smartphones with virtual assistants, social networks that suggest photos to tag and targeted advertising while surfing the Internet. This thesis focuses on machine learning as tool for function approximation, which is a form of regression problem.

In this chapter, we recap the main concepts of machine learning. In Section 3.1 we define the possible training sets and objectives for machine learning algorithms. The Neural Network (NN) is explained in Section 3.2. Then, we define the loss function and optimization method in Section 3.3. For further background, the reader can refer to [21]–[23].

3.1 Training Set and Objective

The training set $\mathcal{D}$ is the set of data-points from which the model has to learn. It is composed by *labeled* or *unlabeled* samples, or observations, of a given process. Given a sample x, the label y, or *class label*, is the categorical class that include the sample. For example, samples can be pictures of animals and the class labels are "cat", "dog" and "horse". If the source process has distribution $p(\tilde{x}, \tilde{y})$, the training set $\mathcal{D}$ has to have a data distribution $p(x, y)$ as close as possible to the true distribution of the process, in order to describe correctly the problem and produce reliable results. This holds also for unlabeled training sets, where $\mathcal{D}$ is only formed by samples x, without class labels. In this case, $p(x)$ is the distribution that describes the training set.

The training set, or dataset, and the objective of the algorithm determines different types of learning. We can distinguish among different frameworks as follows.

- *Supervised learning*, where the training set $\mathcal{D} = \{(x^i, y^i)\}_{i=1}^N$, with data-points drawn

from the distribution $p(x, y)$, i.e. $(x^i, y^i) \sim p(x, y)$, is formed by observations x and true class labels y. As an example, x^i is an image and y^i is the class label, such as "house" or "plane".

- *Unsupervised learning*, where the training set $\mathcal{D} = \{x^i\}_{i=1}^N$ is composed by unlabeled observations, with $(x^i) \sim p(x)$. As an example, x^i is a generic image without an associated class label.

- *Reinforcement learning* is related to sequential decision processes, in which each action receives a feedback. More details will be discussed in Chapter 5.

Taking into account the objective of the framework, the possible type of problems are the following.

- *Classification*, where the final goal is to predict the label (or class) of the input, e.g., image recognition, spam detection.

- *Regression*, where the final goal is to predict a continuous value, given the input, e.g., stock market predictions, function estimation.

- *Clustering*, where the final goal is to divide the inputs in groups, according to the extracted features, without having previous knowledge of the groups, e.g., customer segmentation, content grouping.

3.2 Neural Network

A Neural Network (NN) is the basic element of most of the current machine learning architectures. It is inspired by the biological neural networks that constitute animal brains. This network is composed by interconnected elements, called neurons, that process

the input data and return an output function. These neurons are specified by their *weights*, *biases* and *activation functions*. NNs are composed of layers: the *input* layer which manages the input data, the *hidden* layers that are formed of sets of neurons, and the *output* layer that produces the desired target. Below, we discuss an example NN with one hidden layer of fully-connected (FC) neurons. Other interactive examples can be found in reference [24].

Let $\boldsymbol{x} = (x_1, ..., x_i, ..., x_M)$ be the input vector and $\boldsymbol{y} = (y_1, ..., y_j, ..., y_N)$ be the output vector. The output of the hidden layer is denoted by $\boldsymbol{n} = (n_1, ..., n_k, ..., n_K)$. The input weights are denoted by the $M \times K$ matrix $\boldsymbol{W}^{\text{in}} = \left\{w_{ik}^{\text{in}}\right\}$ and the output weights are denoted by the $K \times N$ matrix $\boldsymbol{W}^{\text{out}} = \left\{w_{kj}^{\text{out}}\right\}$. The biases are denoted by $\boldsymbol{b}^{\text{in}} = (b_1^{\text{in}}, ..., b_k^{\text{in}}, ..., b_K^{\text{in}})$ and $\boldsymbol{b}^{\text{out}} = (b_1^{\text{out}}, ..., b_k^{\text{out}}, ..., b_N^{\text{out}})$, respectively. A nonlinear function $\sigma(\cdot)$ is applied to $\boldsymbol{n}$, and it is also called activation function. The block diagram is illustrated in Figure 3.1.

Each neuron sums all the incoming quantities, scaled by a weight, adds a bias term and then applies a nonlinear activation function. This concept is depicted in Figure 3.2. Accordingly, the output of k-th neuron is

$$n_k = \sigma(\boldsymbol{x}\, \boldsymbol{w}_k^{\text{in},T} + b_k^{\text{in}})$$

where $\boldsymbol{w}_k^{\text{in}} = (w_{1k}, ..., w_{Mk})$ is the k-th column of

$$\boldsymbol{W}^{\text{in}} = \begin{bmatrix} \boldsymbol{w}_0^{\text{in},T} & \cdots & \boldsymbol{w}_k^{\text{in},T} & \cdots & \boldsymbol{w}_K^{\text{in}} \end{bmatrix}$$

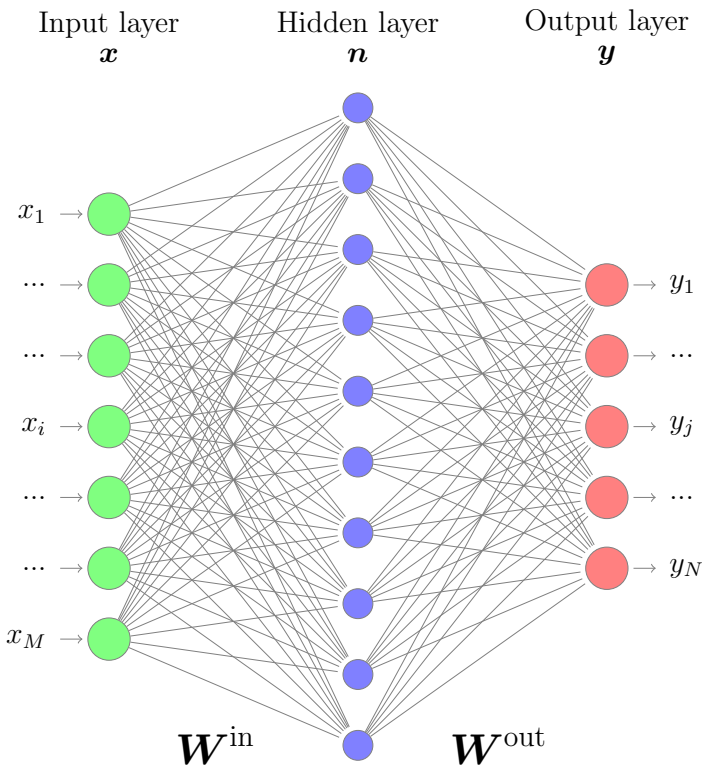


Figure 3.1: Fully Connected (FC) NN.

i.e. the weights that link the input $\boldsymbol{x}$ to neuron k. The j-th output component is

$$y_j = \boldsymbol{w}_j^{\text{out}}\, \boldsymbol{n}^T + b_j^{\text{out}}$$

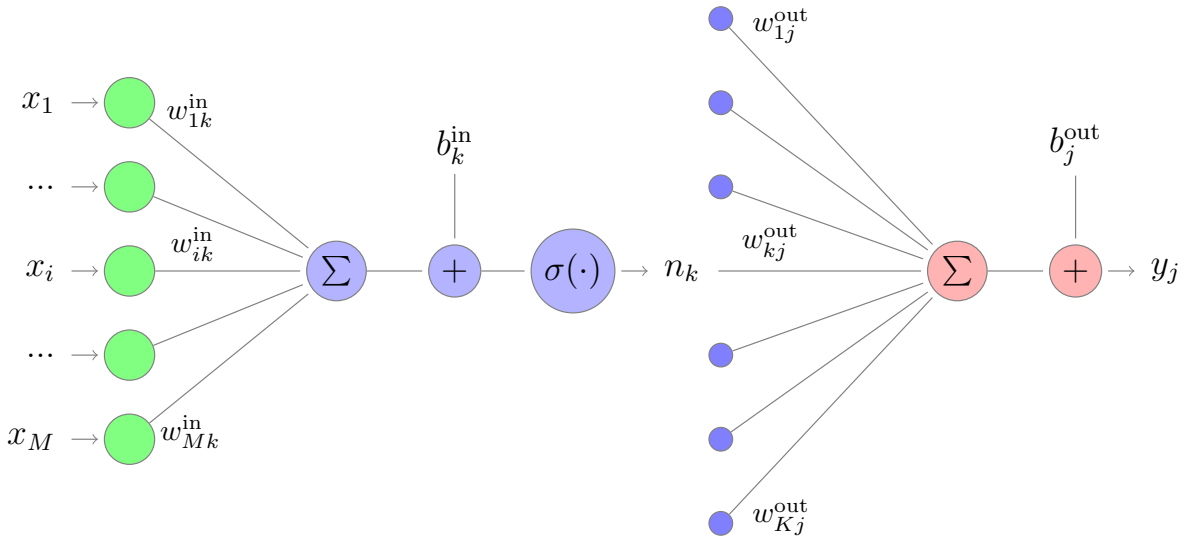


Figure 3.2: Focusing on one branch of the the FC NN: w's are the input/output weights, b's are the input/output biases, $\sigma(\cdot)$ is the activation function and n_k is the output from the k-th neuron.

where $\boldsymbol{w}_j^{\text{out}} = (w_{j1}, ..., w_{jK})$ is the j-th row of

$$\boldsymbol{W}^{\text{out}} = \begin{bmatrix} \boldsymbol{w}_0^{\text{out}} \\ \vdots \\ \boldsymbol{w}_j^{\text{out}} \\ \vdots \\ \boldsymbol{w}_N^{\text{out}} \end{bmatrix}.$$

The *parameters* of a NN are denoted by $\boldsymbol{\theta}$ and include the set of all weights and biases. Then, we can denote the output $\boldsymbol{y} = g(\boldsymbol{x}; \boldsymbol{\theta})$, where $g(\cdot; \boldsymbol{\theta})$ represents the neural network input-output function.

The most common non-linearities, or activation functions, are:

- sigmoid, which maps the input to a squeezed range $[0, 1]$

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

- tanh, which maps the input to a squeezed range $[-1, 1]$

$$\begin{aligned} \sigma_t(x) &= \tanh(x) \\ &= 2 \cdot \sigma_s(2x) - 1 \end{aligned}$$

- ReLU (Rectified Linear Unit), which replaces negative values with zero and keeps the positive ones without changing

$$\sigma_r(x) = \max(0, x).$$

The plots of the above functions are shown in Figure 3.3.

Weights, biases and activation functions are the core of a NN. Weights allow to change the slope of $\sigma(\cdot)$, while biases permit to shift horizontally its function domain. Another important point is that the activation has to be non-linear in order to have the capability to model relevant output. For example, if the activation is linear, we could reduce a multi layer NN to a single neuron network, so that the output is just a simple linear combination of the input. Instead, with non-linearities, the network has the capability to learn complicated functional mappings between input and output.

Among NN architectures, the ones in which the number of hidden layers is "large" are called Deep Learning (DL) networks. DL attempts to imitate how the human brain

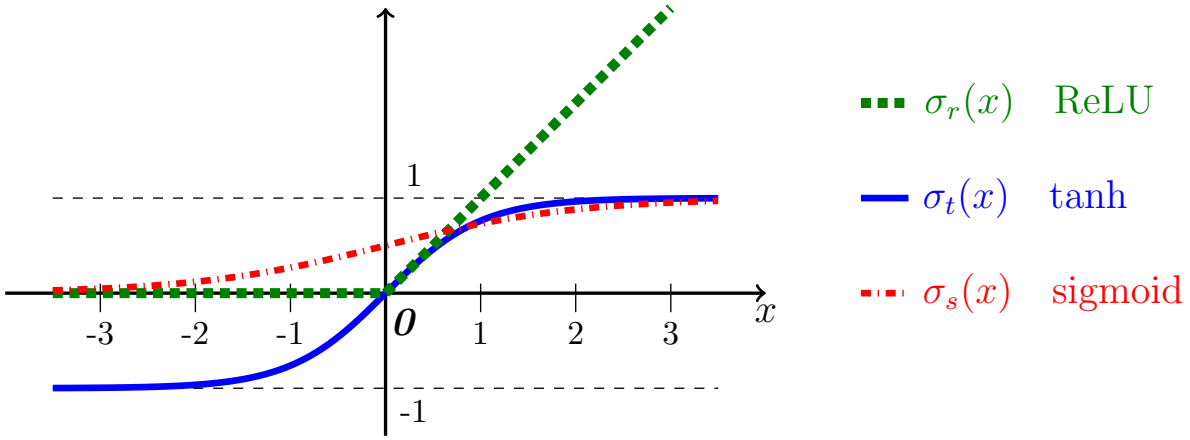


Figure 3.3: Activation functions: sigmoid (red), tanh (blue) and ReLU (green).

works. A DL architecture allows computational models to learn representations of data with multiple levels of abstraction, discovering intricate structure in large data sets by using the backpropagation algorithm [25], which is described in the next section. These models use a cascade of nonlinear layers, with many neurons, in order to extract or transform features of the input data. The output of one layer serves as the input of the successive layer. DL methods achieved amazing performance in Natural Language Processing (NLP) [26], games such as Go [5], and computer vision. For example, the current best performance on ImageNet[1] classification are achieved by an 11 layers deep NN [27].

3.3 Loss Function and Optimization Method

The loss function measures the quality of a classifier or regressor, giving an estimate of the distance between the desired values and the output of the algorithm. In general, we

[1] http://www.image-net.org/

look for an algorithm that minimizes the empirical loss

$$L_\theta = \frac{1}{N}\sum_{i=1}^{N} \ell(\hat{y}^i, y^i),$$

where $\hat{y}^i = g(x^i; \boldsymbol{\theta})$ is the estimated output and $g(\cdot; \boldsymbol{\theta})$ represents the parametric input-output function, with parameters $\boldsymbol{\theta}$; y^i can be the true class label (supervised learning) or some other measure that represent the data structure, such as the cluster center μ_k (unsupervised learning) or the action-value Q_i (reinforcement learning); the function $\ell(\cdot)$ represents the mathematical model of the loss. Some of the most used such functions are the squared loss, or $\ell 2$, where $\ell(a, b) = \ell 2(a, b) = (a - b)^2$, and the cross entropy loss $\ell(p, q) = H(p, q) = H(p) + D_{KL}(p||q)$, where p and q are distributions over classes, $H(p)$ is the entropy of p and $D_{KL}(p||q)$ is the Kullback-Leibler divergence of q from p.[2] In other words, cross entropy measures the "distance" between the true probability distribution p and the estimated output distribution q. Cross entropy is the standard loss function for supervised classification algorithms.

Hence, given a NN, the goal is to minimize the loss, with respect to the parameters of the NN. If the loss if a convex function, the minimum value is unique. Gradient Descent (GD) provides an optimization algorithm to find the minimum of the loss function. Let $L(\boldsymbol{\theta})$ be a scalar loss function, where $\boldsymbol{\theta} = \left(\theta^1, ..., \theta^J\right)$ is the vector of J NN parameters. Then, the GD update, for time instant t, for the parameters is

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \gamma \, \nabla L(\boldsymbol{\theta}_{t-1}) \tag{3.1}$$

[2] The cross-entropy notation $H(p, q)$ must not be confused with the joint entropy.

where γ is the learning rate and $\nabla L(\boldsymbol{\theta})$ is the gradient vector

$$\nabla L(\boldsymbol{\theta}) = \left[\frac{\partial L(\boldsymbol{\theta})}{\partial \theta^1}, ..., \frac{\partial L(\boldsymbol{\theta})}{\partial \theta^j}, ..., \frac{\partial L(\boldsymbol{\theta})}{\partial \theta^J}\right]$$

i.e., it is the vector of partial derivatives with respect to each parameter θ^j and it can be interpreted as the direction of steepest ascent in the parameter space. At every update step, according to (3.1), each parameter θ^j is corrected by $-\gamma \, {}^{\partial L(\boldsymbol{\theta})}/_{\partial \theta^j}$, in the direction against the gradient, because we want to move toward the minimum. If γ is adjusted properly, then GD converges to a local minimum of $L(\boldsymbol{\theta})$. An example of GD idea is depicted in Figure 3.4.

For NN, this update method is also referred to *back-propagation*. Initially the parameters $\boldsymbol{\theta} = \boldsymbol{\theta}_0$ are assigned randomly according to some initialization strategy, e.g. Gaussian. Given the input from the training set, the output of the NN is observed and the new value of the loss is computed $L(\boldsymbol{\theta}_0)$. Then, the loss is propagated back to the previous layers, adjusting the parameters $\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_0 - \gamma \cdot \nabla L(\boldsymbol{\theta}_0)$ as in equation (3.1).

Consider supervised learning with training set $\mathcal{D} = \{(x^i, y^i)\}_{i=1}^N$. Depending on the amount of data used for the gradient computation, we can recognize three variants of GD algorithms [28].

1. Batch GD computes $\nabla L(\boldsymbol{\theta})$ using the entire training set $\mathcal{D}$ in just one update.

2. Stochastic GD (SGD) performs the parameter update for each single sample of the training set (x^i, y^i). It is usually faster than Batch GD and can also be used for online learning.

3. Mini-batch GD mixes Batch GD and SGD, since it performs the update for every mini-batch of n training samples $\{(x^i, y^i)\}_{i=1}^n \subset \mathcal{D}$, where n is called mini-batch size

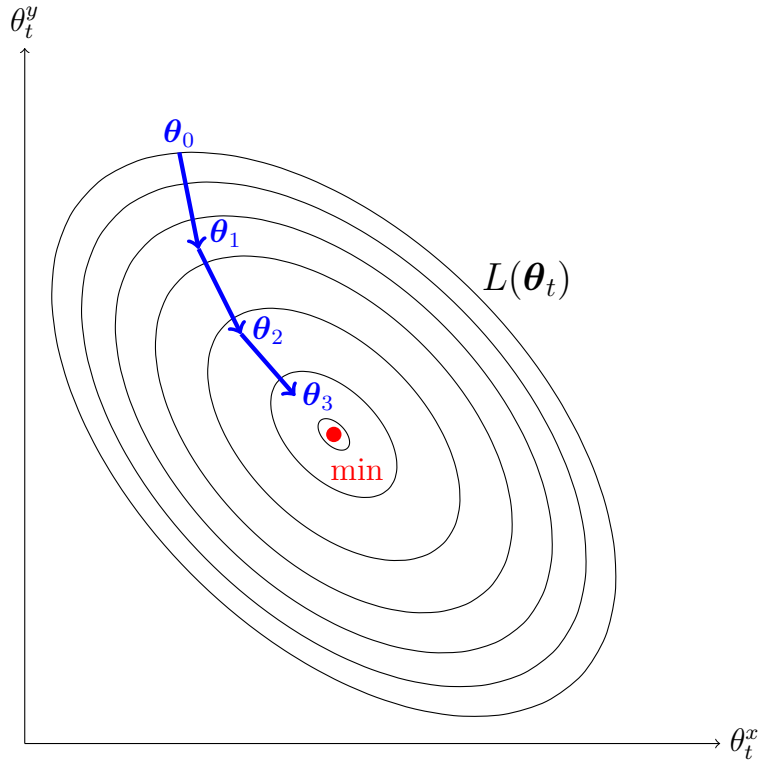


Figure 3.4: GD illustration for the 2-dimensional parameter $\boldsymbol{\theta}_t = (\theta_t^x, \theta_t^y)$. The black lines are the contour levels for the loss function $L(\boldsymbol{\theta}_t)$. Each blue arrow represent the gradient update from $\boldsymbol{\theta}_{t'}$ to $\boldsymbol{\theta}_{t'+1}$, which corresponds to the loss transition from $L(\boldsymbol{\theta}_{t'})$ to $L(\boldsymbol{\theta}_{t'+1})$, for $t' = 0, 1, 2$. The goal is to approach $L(\boldsymbol{\theta}_T) \approx \min_{\boldsymbol{\theta}_t} L(\boldsymbol{\theta}_t)$, after some time instant $t' = T$.

and $n < N$. This is the most used method in current machine learning applications and also referred as SGD.

Adaptive Moment estimation (ADAM) is another optimization algorithm, where running averages of both the gradients and the second moments of the gradients are used [29]. It is widely used in the literature and it can also be trained with the mini-batch method.

The learning rate γ is usually a small value, tuned according to the mini-batch size [30]. Ideally, if the mini-batch size is multiplied by k, then the learning rate should be multiplied by k. In general, the learning rate needs to be small enough to let the training

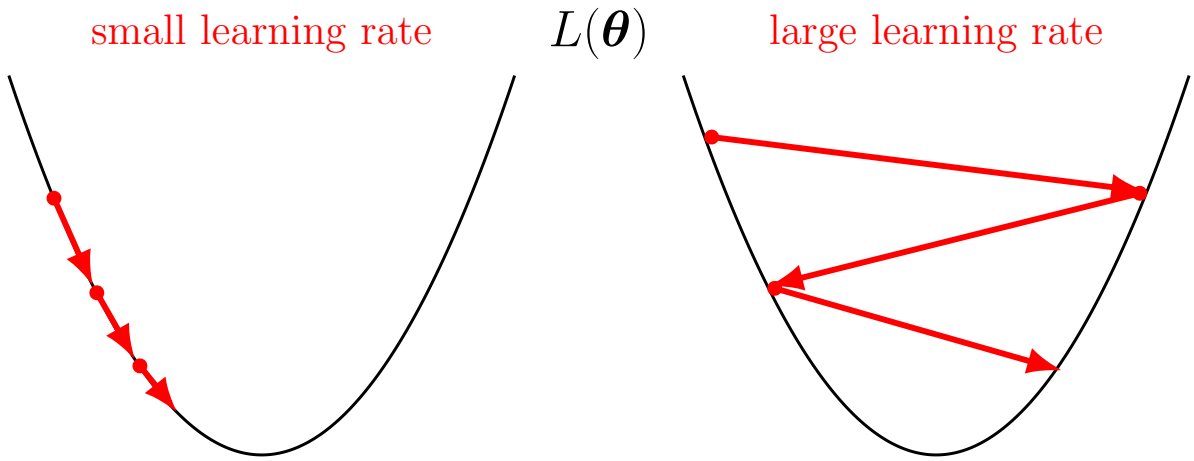


Figure 3.5: Example of GD for different learning rate values.

proceed toward the minimum, and avoid jumps over it. This concept is sketched in Figure 3.5. On the other hand, if it is too small, the training process will be much slower.

The *hyperparameters* of the NN are those parameters which are not learned from the learning process, such as learning rate, mini-batch size, training epochs, number of hidden layers and neurons. Usually, they are fixed before the training process. They require to be tuned properly, according to the input data, to allow the network to produce desired results.

Chapter 4

Belief-Propagation Decoding Optimization

"More data beats clever algorithms,
but better data beats more data."
Peter Norving

In this chapter, we study one machine learning application, based on the description given in Chapter 3, to Belief-Propagation (BP) decoding, which has been described in Section 2.4. Previous work [11] has shown that BP performance can be improved by assigning proper weights to the edges of the Tanner graph for scaling the exchanged messages. These weights are then trained using machine learning techniques. The key idea is that the feed-forward unrolled BP iterations can be interpreted as a deep Neural Network (NN), where the neurons are the Variable Nodes (VNs) and Check Nodes (CNs), and the connections are drawn by the Parity Check (PC) matrix $\boldsymbol{H}$ of the code. For our experiments, we use the RM(2,5) code, which is introduced in Section 2.5. Part of our

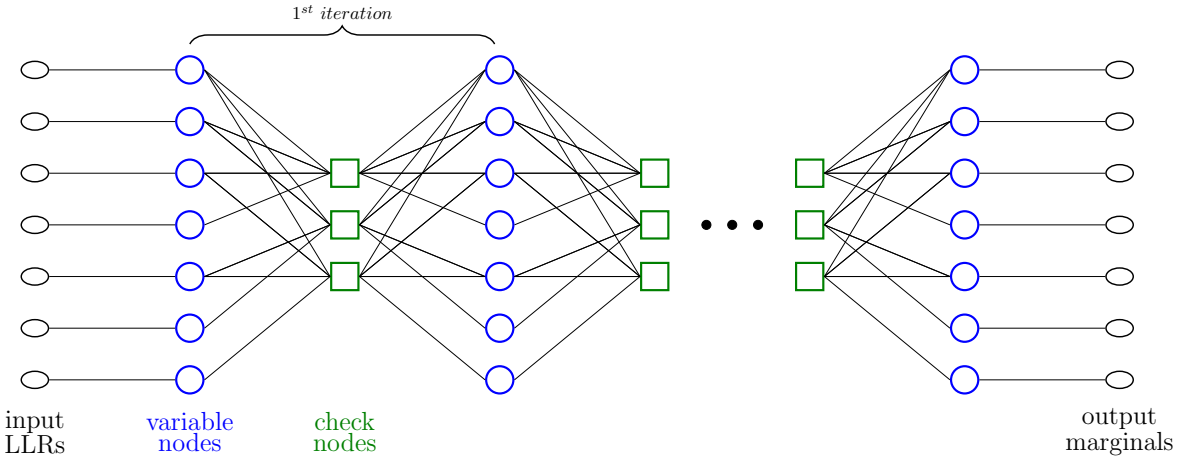


Figure 4.1: BP unrolled iterations for the (7,4) Hamming code. VNs are the blue circles, while CNs are the green squares. The black ellipses represent the input LLRs and the output marginals after BP decoding.

work is presented in [12].

In Section 4.1, we describe the NN decoder architecture. The training procedure is described in Section 4.2. Then, the results are shown in Section 4.3 and the conclusions are discussed in Section 4.4.

4.1 NN Decoder Architecture

Given a linear block code with PC matrix $\boldsymbol{H}$, one can build the Tanner graph, as explained in Section 2.2. For a fixed number of iterations, the BP decoder can be interpreted as a fixed feed-forward chain of operations on the repeated Tanner graph connections. A simple example of unrolled BP iterations for the Hamming(7,4) code is shown in Figure 4.1 and its original Tanner graph is in Figure 2.2.

Nachmani et al. proposed a parameterized BP decoder, where scaling factors are applied to every edge of the Tanner graph [11]. Then, these parameters are learned using

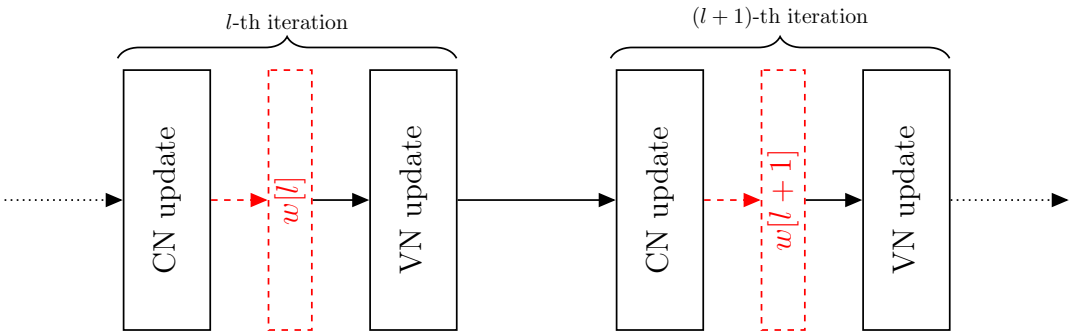


Figure 4.2: Block diagram of the message weighting algorithm, where $w[l]$ is the weight for the l-th iteration. The blocks CN/VN update represent the processors that compute the messages in the BP decoding algorithm.

machine learning techniques in order to improve decoding performance. Since the number of edges in the Tanner graph may be large, in our experiments we also investigate simplified models with fewer parameters. In this way, our BP parameterization has a reduced complexity. We consider the two following scaling schemes, whose block diagrams are shown in Figures 4.2 and 4.3. First, we apply one weight per iteration, which scales all messages from CNs to VNs, so that every edge for a given iteration share the same optimized weight. Second, we introduce a *damping* among successive iterations with a proper coefficient, which mitigates the message updates oscillations and improve convergence [10]. Note that this method was not considered in [11].

Recalling Algorithm 2.1, we can rewrite the VN update equation (2.9) as

$$L_{j\to i} = L_j + w[l] \cdot \sum_{i' \in N(j)\setminus\{i\}} \hat{L}_{i'\to j}, \tag{4.1}$$

where $w[l]$ is the weight that scales CN messages for the l-th iteration. This is illustrated in Figure 4.2.

Every message, related to one edge of the Tanner graph, is scaled by the damping

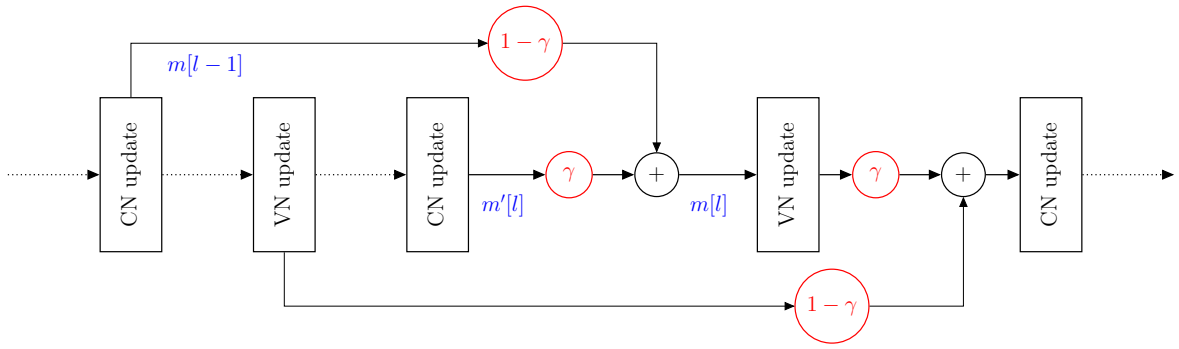


Figure 4.3: Block diagram for message damping algorithm, where γ is the damping coefficient.

coefficient γ and summed to the message at the previous iteration, which is scaled by $(1-\gamma)$. Let $m\,[l]$ be a generic message at iteration l and $m'\,[l]$ the message which comes from the previous update block, as illustrated in Figure 4.3. Then the damping equation is

$$m\,[l] = \gamma \cdot m'\,[l] + (1-\gamma) \cdot m\,[l-1]\,. \tag{4.2}$$

The complete scheme, including damping coefficient and weights for 4 BP iterations is shown in Figure 4.4.

4.2 Training

RM(2,5) is a linear code, so the the all-zero codeword is included in the codebook. Without loss of generality, we can always transmit the all-zero codeword, because the BP performance is independent of the transmitted codeword for a binary memoryless symmetric channel [31, Lemma 4.92]. We use a fixed number of BP iterations equal to 4 and no other stopping criterion is implemented. Then, there are 5 coefficients to be optimized, 4 of them are the iteration weights and one is the damping coefficient. We also clip the values of the CN update, to avoid divergence of $\phi(\cdot)$. Then, $\phi(x) < A$, with $A = 8$. The

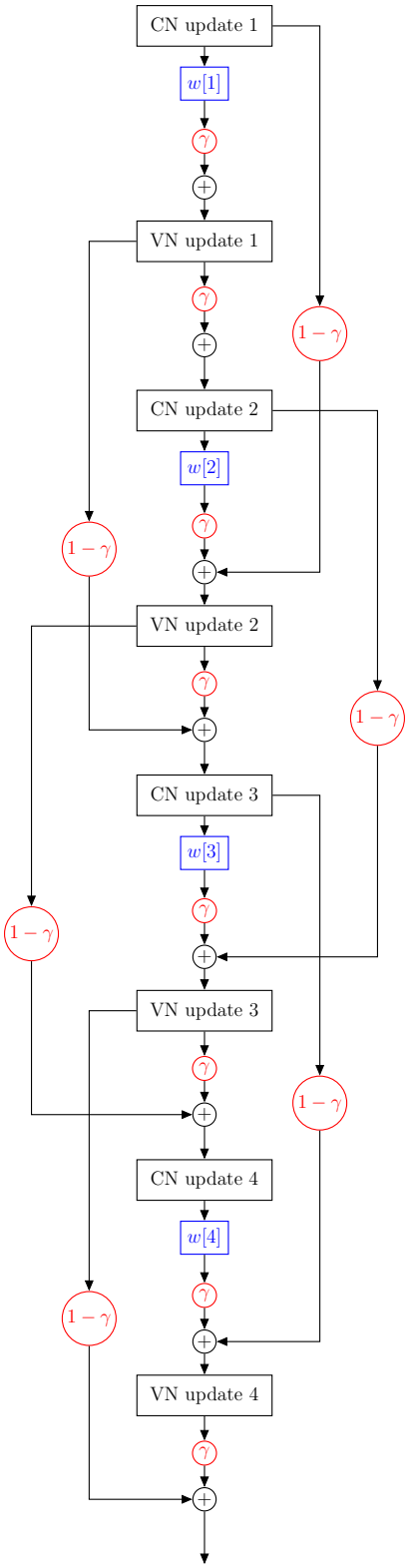


Figure 4.4: Block diagram of the parameterized BP decoder with 4 iterations. The iteration weights are drawn in blue, the damping coefficient is drawn in red. The black boxes are VN and CN update processors.

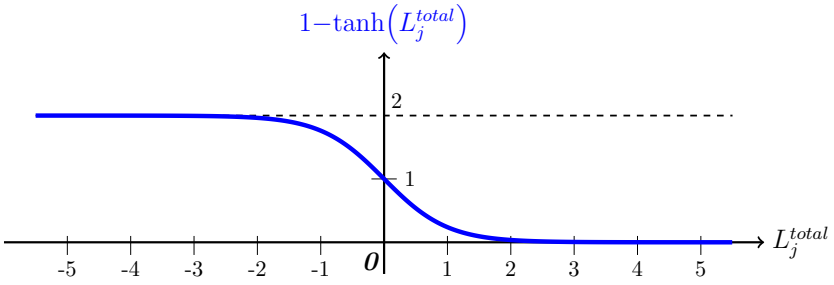


Figure 4.5: Loss function used for training, where y_i is the i-th entry of the vector of marginal LLRs.

experiments are performed in Python [32] and Tensorflow [33].

For the training process, we need to choose a loss function to be minimized by SGD, as explained in Chapter 3. Our loss function is proportional to a soft version of the Bit Error Rate (BER) defined as

$$\ell\left(\boldsymbol{L}^{total}\right) = \sum_{j=1}^{N} \left[1 - \tanh\left(L_j^{total}\right)\right] \tag{4.3}$$

where $\boldsymbol{L}^{total}$ is the vector of marginal LLRs at the end of the BP iterations. It corresponds to equation (2.10) of Algorithm 2.1. The function $1-\tanh\left(L_j^{total}\right)$ is drawn in Figure 4.5. When the all-zero codeword is transmitted, with BPSK mapping, the received values are $y_j = +1 + n\left(0, \sigma^2\right)$, hence, for BI-AWGN channel with LLRs $L_j = 2\, y_j/\sigma^2$, the output marginal LLRs L_j^{total} are expected to be greater than zero. So, $\ell\left(\boldsymbol{L}^{total}\right)$ is proportional to an error counter, since for negative L_j^{total} values, which correspond to incorrectly decoded bits, the function returns a value greater than 1 and it saturates to 2 for $L_j^{total} \lesssim -2$. On the other hand, for positive L^{total} it returns values lower than 1 and it saturates to 0 for $L^{total} \gtrsim 2$.

We adopt a multi-loss model [11], i.e., the marginals are computed after every iteration. At the end of the decoding procedure, we calculate the loss according to (4.3), using B

vectors of marginal LLRs $\left(\boldsymbol{L}^{total,(1)},...,\boldsymbol{L}^{total,(B)}\right)$, where B is the total number of BP iterations. For our experiment we consider $B = 4$. Then, we average all of them to obtain the multi-loss function

$$\frac{1}{B}\sum_{l=1}^{B}\ell\left(\boldsymbol{L}^{total,(l)}\right).$$

SGD is used to minimize the multi-loss function, with respect to the parameter vector

$$\boldsymbol{\theta} = (\gamma, w[1],...,w[B]).$$

The cross entropy has also been investigated as loss function, as proposed in [11], but the training did not produce any improvement in terms of Codeword Error Rate (CER). As a motivation, we provide the result of a grid search of the CER, cross-entropy and $\ell(\cdot)$ versus the scaling parameters. The 3D plot can be found in [34]. The local minimum of CER and cross entropy are not aligned, so the learning process with respect to cross entropy loss does not produce the desired performance improvement. Further investigation is provided in [12].

The mini-batch size, which is the number of codewords analyzed for each training step, is another important choice for the learning process. In order to guarantee a sufficient number of errors for each SGD step, we use a large mini-batch of 2000 codewords. In this way, the variance of the loss function is reduced. According to this, the learning rate starts from 0.5 and it is decreased by a factor 3 every $T_{\text{steps}}/3$ gradient steps, where $T_{\text{steps}} = 4000$ is the total number of training steps.

4.3 Results

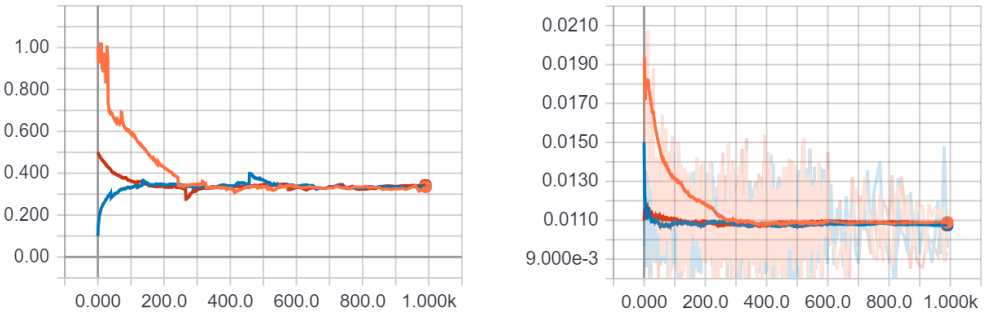
Experiments performed in the BI-AWGN channel, at fixed SNR $E_s/N_0 = 3\,\text{dB}$, are presented in this section. We use the RM(2,5) code, which has length $n = 32$. We use m to denote as the number of PC equations used in the decoding algorithm. Hence, the PC matrix $\boldsymbol{H}$ has dimension $m \times n$. Using the results on redundant PC matrices from [18], we apply BP decoding with the overcomplete matrix $\boldsymbol{H}_{\text{oc}}$, which has dimension 620×32. It is composed of Minimum-Weight Parity Checks (MWPCs) with Hamming weight equal to 8 and the total number of ones in $\boldsymbol{H}_{\text{oc}}$, or edges in the Tanner graph, is 4960. Later, we also provide results for random subsets of $\boldsymbol{H}_{\text{oc}}$, i.e., we randomly choose a subset of its rows and run BP on this subgraph.

The reference curves are the plain BP, which is the standard version with $\gamma = 1$ and $w[l] = 1\ \forall l$, and the Ordered Statistic Decoder (OSD), which performs close to Maximum Likelihood (ML) decoding for this code [35]. Our decoder is tested to produce at least 500 errors for each SNR point of the performance curve.

We run simulations with different training targets, defined as follows.

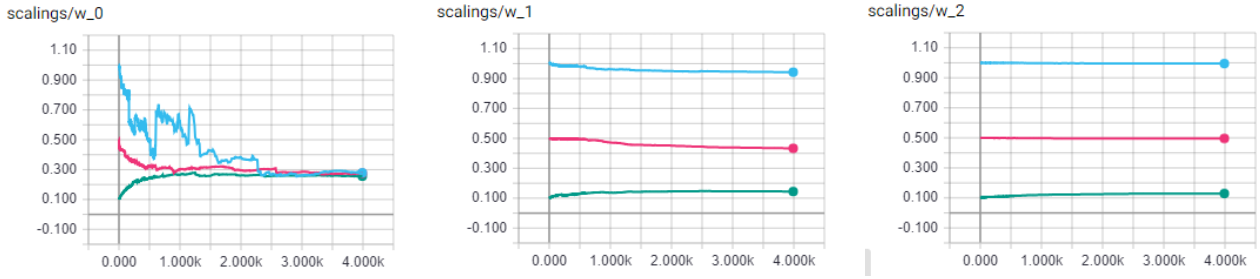
- *Scenario 1*: only the damping coefficient is trained, i.e. the gradient is applied only to γ. Weights are constant $w[l] = 1\ \forall l$. The damping coefficient is initialized with different values $\gamma = [1.0, 0.5, 0.1]$, to check if the initialization choice affects the learning process. The damping updates are shown in Figure 4.6a. The loss evolution is plotted in Figure 4.6b.

- *Scenario 2*: only weights $w[l]$ are trained, while the damping coefficient is constant $\gamma = 1$. The weights are initialized with the same different values $w[l] = [1.0, 0.5, 0.1]\ \forall l$, to verify if the learning results are independent from the parameter initialization. The weights updates are shown in Figure 4.7.



(a) Damping updates. (b) Loss evolution.

Figure 4.6: Learning outcomes for scenario 1. The damping coefficient is initialized to 1 (orange), 0.5 (red) or 0.1 (blue). For subfigures (a) and (b) the x-axis is the number of training steps.



(a) (b) (c)

Figure 4.7: Weight updates for scenario 2. Weights are initialized to 1 (cyan), 0.5 (magenta) or 0.1 (green). The x-axis is the number of training steps. Note that the weight of the first iteration (a) converges, while the others keep almost constant (b),(c).

- *Scenario 3*: both damping coefficient γ and weights $w[l]$ are trained together. We initialize the decoder as plain BP. The parameters updates are shown in Figure 4.8.

The loss evolution for scenarios 2-3 is omitted, since our learning process exhibits the same trend for each analyzed training target.

For scenario 1, we found that the optimal damping, which minimizes the loss function, is $\gamma \approx 0.33$. On the other hand, for scenario 2, the only weight that is consistently updated

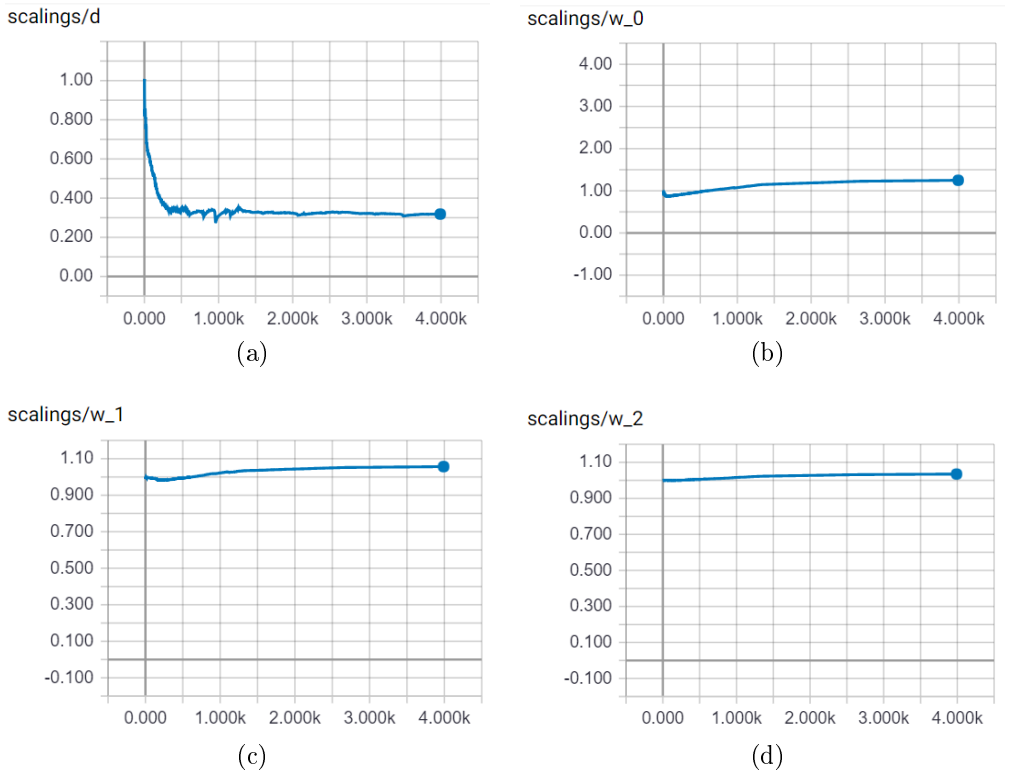


Figure 4.8: Parameters updates for scenario 3. The (a) plot is referred to damping, the others to the weights (b),(c),(d). All the parameters are initialized to 1, i.e., we initialize the model as plain BP. Note that the damping coefficient is the only parameter that is updated significantly, while the others remain almost constant.

is $w[1] \approx 0.28$, that is the weight for the first iteration. The others remain almost constant during training, i.e., the gradient with respect to them is negligible. Finally, in scenario 3, the damping coefficient converges to the same value $\gamma \approx 0.33$, while all the weights exhibit negligible gradient update.

For scenarios 1, 2 and 3, the resulting performance in terms of CER is almost the same. A plot of CER vs E_s/N_0 can be found in Figure 4.9. Note that with the optimized BP decoder, at CER $= 10^{-2}$, we obtain a 0.33 dB gain from the plain BP decoder, and we are only 0.22 dB far from the OSD decoder.

We also run other experiments, using randomly chosen subsets of a rows from $\boldsymbol{H}_{\text{oc}}$.

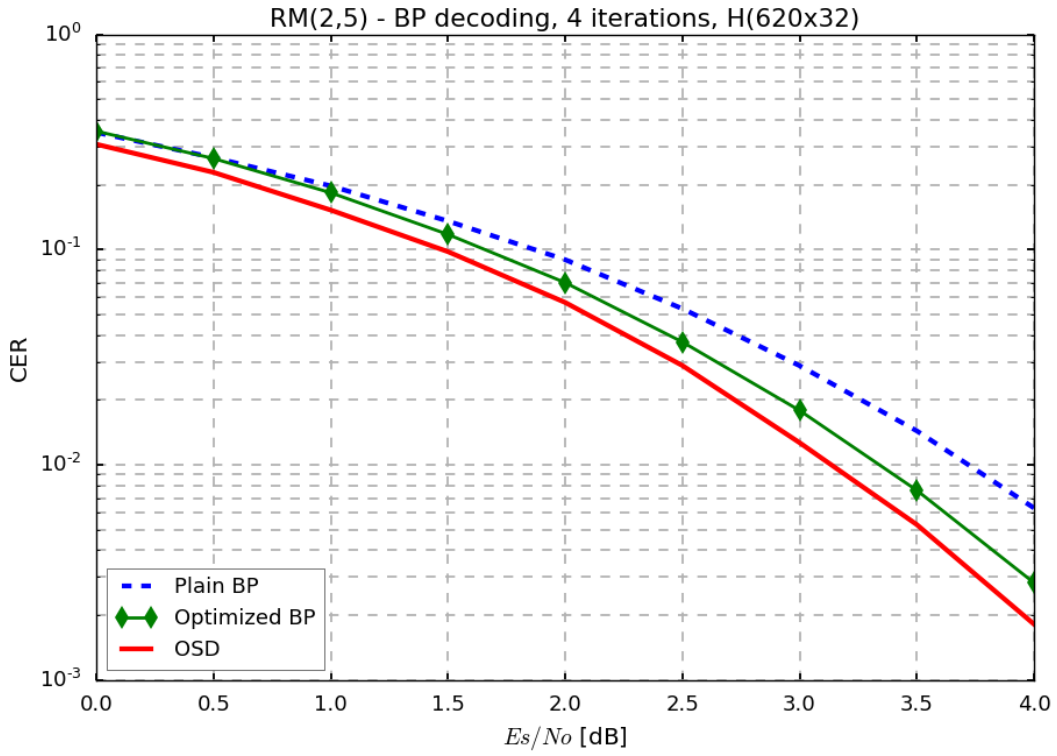


Figure 4.9: CER vs E_s/N_0 for the learned decoder. We use the overcomplete PC matrix $\boldsymbol{H}_{\text{oc}}$ for decoding. For all scenarios 1-2-3 the CER performance are almost equivalent.

Namely, we apply BP decoding with $\boldsymbol{H}'$, whose dimension is $a \times 32$. We use values of a that correspond to $[5, 10, 15, ..., 90, 95]$ % of the rows of $\boldsymbol{H}_{\text{oc}}$. For each experiment, we average over 20 unique choices of $\boldsymbol{H}'$, and we notice that the performance are always the same. Then, the choice of the MWPCs does not affect the performance, only the dimension of the PC matrix $\boldsymbol{H}'$ matters.

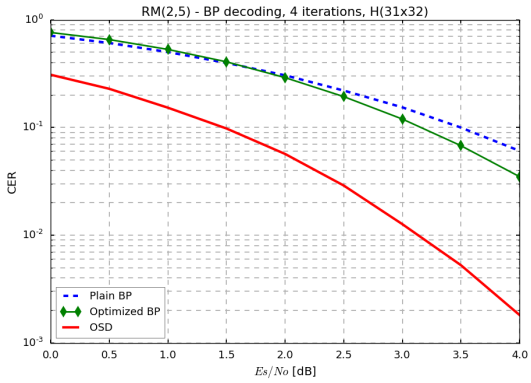
We report the average performance curve, for both the learned BP decoder and the plain BP decoder, which is the decoder with all parameters equal to 1. The optimized decoder has always a lower CER than the plain BP reference curve, as we expected. The resulting performance curves for some random subset are in Figure 4.10. A summary curve

that shows performance with respect to OSD is shown in Figure 4.11. We can notice that as we increase the dimension of $\boldsymbol{H'}$, the CER is closer to OSD case. Furthermore, after some point (e.g., using $\boldsymbol{H'}$ 496×32, that is 80% of the rows of $\boldsymbol{H}_{\text{oc}}$), the performance of the learned decoder is almost the same. This fact is highlighted in Figure 4.12, where we plot the performance gap in terms of E_s/N_0 for a given value of CER between our learned decoder, OSD and plain BP decoder. We can observe the following facts.

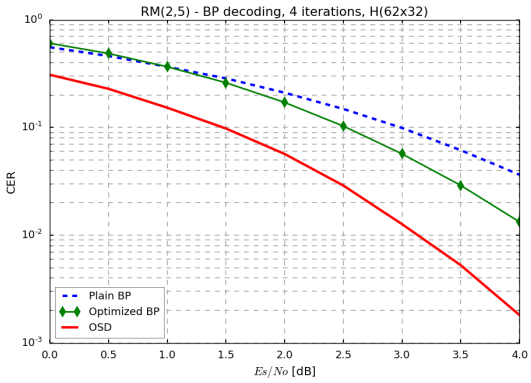
- Gap OSD vs plain BP (green line): it is decreasing as we use a larger PC matrix $\boldsymbol{H'}$. It is higher than the learned decoder case, as we expected.
- Gap OSD vs learned BP (blue line): same shape as the green one, but lower. The learned decoder is able to reach performance closer to OSD, with an almost flat 0.25 dB penalty after using at least 60% of the rows.
- Gap learned BP vs plain BP (magenta line): the shape of this curve is slightly different from the others. We can notice that we have a local maximum gain by using 15% of the rows of $\boldsymbol{H}_{\text{oc}}$, after that the performance gap decreases. In addition, the variance of this performance gap is also smaller than the other cases.

4.4 Conclusions

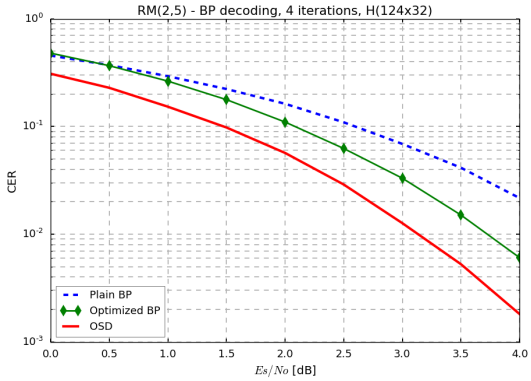
In this chapter, we have shown how to improve the decoding performance, by applying scaling factors to the Tanner graph and learning these parameters. From our experiments, we observed that the damping coefficient provides most of the gain in performance and it dominates the training process. On the other hand, the learned values of the weights are different, depending on the damping coefficient. If this parameter is trained, weights will be almost unchanged during the training. On the other hand, if no-damping is set, the



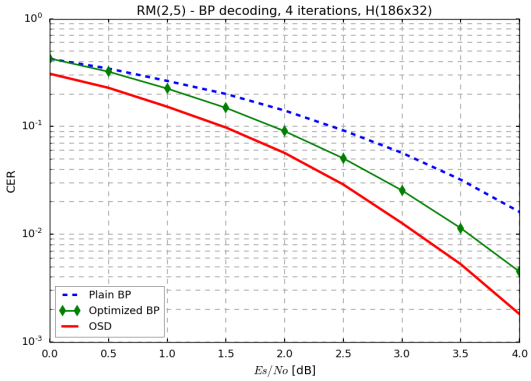
(a) $a = 31$



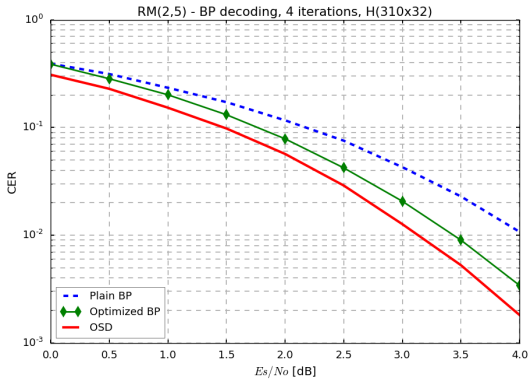
(b) $a = 62$



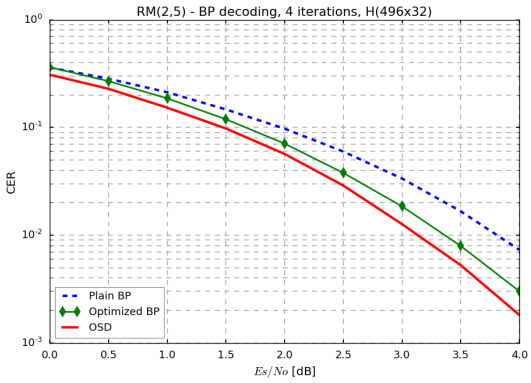
(c) $a = 124$



(d) $a = 186$



(e) $a = 310$



(f) $a = 496$

Figure 4.10: CER vs E_s/N_0 for random subsets of $\boldsymbol{H}_{\text{oc}}$. The PC matrix $\boldsymbol{H'}_{\text{oc}}$ that we use for decoding is composed by a rows, with $a = [31, 62, 124, 186, 310, 496]$. These values correspond to $[5, 10, 20, 30, 50, 80]$ % of the rows of $\boldsymbol{H}_{\text{oc}}$.

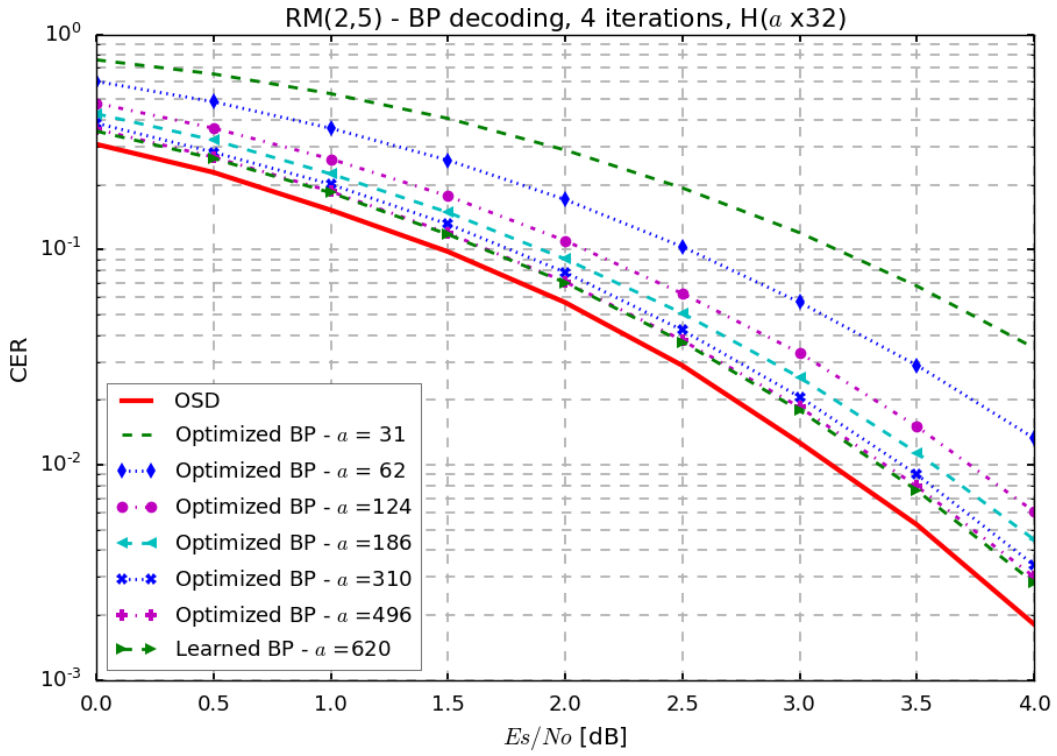


Figure 4.11: Summary CER vs E_s/N_0 for random subsets of $\boldsymbol{H}_{\text{oc}}$. Different colors show performance for different $\boldsymbol{H}'$, of dimension $a \times 32$.

first weight will converge to a small value. Hence, the messages need to be scaled down in both cases, in order to improve BP performance.

We can suggest the following interpretation: BP decoding on loopy graphs converges very rapidly to certainty once it is close to a codeword. For short codes such as the RM(2,5), it can converge to incorrect codewords, and the resulting penalty could be large, e.g., on the order of the max LLR value per incorrect bit. So, scaling down the messages of the first iteration, i.e., when more "correctable" errors are present, helps the BP decoder to not converge hastily to a codeword that may be wrong.

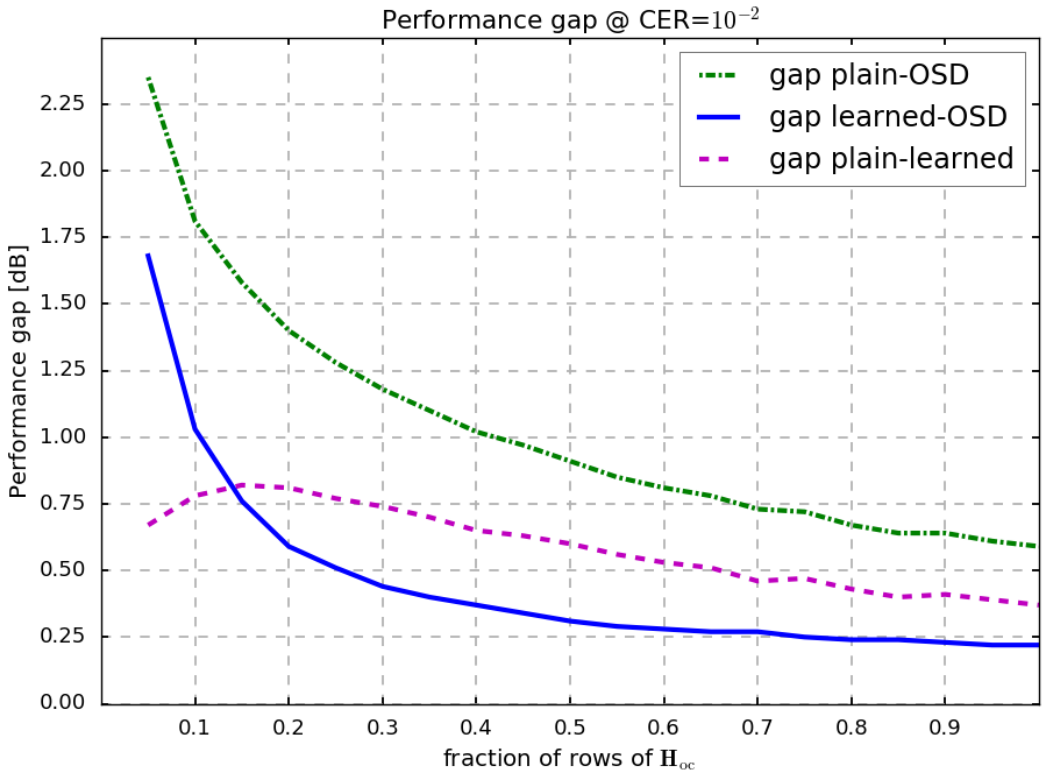


Figure 4.12: The fraction of rows of $\boldsymbol{H}_{\text{oc}}$, from 0.05 to 1, is on the x-axis. On the y-axis, the performance gap between decoders is plotted. The CER is fixed to 10^{-2}.

Chapter 5

Reinforcement Learning

"You don't learn to walk by following rules.
You learn by doing, and by falling over."
Richard Branson

Learning by interacting with the environment is probably the most common method employed by intelligent beings. Infants learn to control their muscles and walk thanks to repeated experiments in their surrounding environment. One can also train pets by giving a reward after some good action. Intelligent beings adjust their behavior after experiencing events, trying to obtain the "best" possible result.

Reinforcement learning (RL) is the branch of machine learning that is concerned with making sequences of decisions. It consists of mapping the *state* of an *environment*, into an *action* taken by an agent that moves within it. The sequence of chosen actions has to maximize the future *reward*. The environment is modeled as a Markov Decision Process (MDP), which is a mathematical framework that models the environment as a controlled stochastic processes, using basic elements, such as states, transition probabilities and

rewards. In RL, the agent does not have prior knowledge about the quality of the possible actions and the goal is to maximize the total reward. As some rewards may arrive in the future, it has to learn by trial and error, trying to maximize an uncertain delayed reward.

In general, RL requires many interactions with the environment for the training process. RL algorithms are often applied in the context of playing games or robotics [1], [6], [36], since the realizations of the process can be easily simulated. In supervised learning, for example, training a neural network to play "Pong" game would need some hours of recorded game-play from an expert human player, with all the game frames labeled and stored in a training set. Furthermore, the learned agent may not beat the human performance, because it learns to mimic the human choices. On the other hand, with RL the agent has more degrees of freedom since it learns from rewards, and it can discover other sequences of actions to maximize them. In this thesis, we approach the decoding problem of a linear code as a "game", where the agent gets a high reward if the correct decision is made. This approach will be pursued in Chapter 6 for decoding RM codes on the BSC.

In this chapter, we review the main background regarding RL. Section 5.1 defines the MDP framework. Section 5.2 introduces the concept of RL, pointing out the main features. Finally, in Section 5.3 we focus on Q-learning, which is the RL technique used in our experiments. For further material, the reader can consult [37, Lectures 8-11] and [38, Chapter 3].

5.1 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework used to understand stochastic sequential decision processes. It is defined by the following quantities.

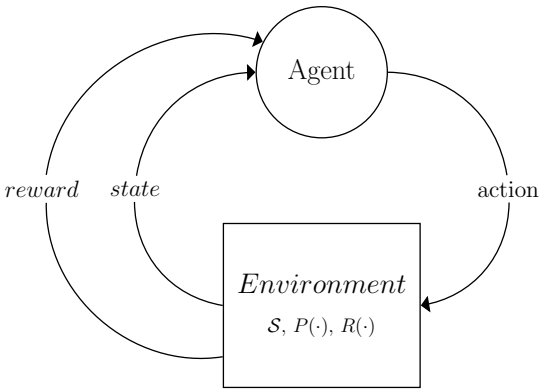


Figure 5.1: Block diagram of MDP.

- A **state** $s \in \mathcal{S}$, where $\mathcal{S}$ is the state space.
- An **action** $a \in \mathcal{A}$, where $\mathcal{A}$ is the action space.
- A **transition** probability function defined as the probability that being in state s_t and taking action a_t, the agent lands in state s_{t+1}, namely $P(s_{t+1}|s_t, a_t)$. It is usually referred to the model.
- A **reward** function $R(s_t, a_t, s_{t+1})$. Usually, the reward depends only on the current state s_t, i.e., $R(s_t, a_t, s_{t+1}) \simeq R(s_t)$. When negative, it can be interpreted as a penalty.
- An initial state s_0, or distribution $P(s_0)$.
- An **agent** (or decision maker) that takes actions attempting to maximize the expected future rewards.

The block diagram is illustrated in Figure 5.1.

By definition, MDPs satisfy the **Markov property**, namely:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ..., s_0) = P(s_{t+1}|s_t, a_t)$$

i.e., given the present state, the future is conditionally independent from the past. In an MDP, a **policy** $\pi : \mathcal{S} \to \mathcal{A}$ defines the action to be taken by the agent for each state. The sequence of states and actions $\{(s_t, a_t)\}_{t=0}^{T}$ is denoted as **episode**, where T is the stopping time of the process. The corresponding cumulative reward, adopting policy π, is

$$R^{\pi}_{\text{cum}} = \sum_{t=0}^{T} R^{\pi}_{t}(s_t, a_t, s_{t+1}) = r^{\pi}_{0} + r^{\pi}_{1} + ... + r^{\pi}_{T}$$

where r^{π}_{t} is the reward obtained at time t, taking actions accordingly to policy π. So, the goal is to find the **optimal policy** $\pi*$ that maximizes the expected cumulative reward

$$\pi^{*} = \arg\max_{\pi} \mathbb{E}[R^{\pi}_{\text{cum}}].$$

Typically, the agent has to find a compromise between immediate rewards and future rewards, which arise from the choice of transition probabilities that eventually lead to "high reward" states. To simplify the notation, the superscript $^{\pi}$ is often dropped from the reward function. The context should eliminate any ambiguity.

A problem could occur at this point: MDP that may not terminate in finite time, i.e., $\Pr(T = \infty) > 0$, may have infinite total reward. In this case, one uses instead the *discounted cumulative reward*, which is defined as

$$\tilde{R}_{\text{cum}} = \sum_{t=0}^{\infty} \gamma^{t} r_t \leq \frac{r^{\max}}{1-\gamma},$$

where $0 < \gamma < 1$ is the discounting factor and $r^{\max} = \max_t r_t$. It is worth noting that, when γ is small, the agent is focused on maximizing the short term rewards. On the other hand, for large γ the focus will be on the long term rewards.

Another solution to avoid reward divergence is to assume *finite horizon*, i.e., the

episodes end after a fixed number of steps, denoted as $T^{\max}$. In this case, state $s_{T^{\max}}$ is always assumed to be an exit state.

In order to find the optimal policy π^*, two methods can be applied, i.e., value iteration and policy evaluation. We will focus on value iteration in order to understand the application discussed in Chapter 6. Policy evaluation is not treated, since it assumes to fix the policy π, and it is not of practical use for the our discussed application.

The value function is defined as

$$V_t^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{i=0}^{T} R^{\pi}(s_{t+i}, a_{t+i}, s_{t+i+1}) \middle| s_t = s\right]$$

and gives the *value* of being in state s at time t under policy $\pi(s)$. I.e., it is the expected sum of rewards accumulated when starting from state s and acting according to policy $\pi(s)$ for a horizon of T steps. In the following derivations, we interchangeably use the notation $(s_t, a_t) \rightarrow s_{t+1}$ or $(s, a) \rightarrow s'$ to denote the transition from state s to the next state s'. Once defined the value function, one can iteratively compute V to find the optimal policy that maximizes the rewards. Then, the **value iteration** approach is:

1. Initialize: $\forall s \in \mathcal{S} \; : \; V_0^*(s) = 0$

2. For $t = 1, .., T$, given $V_t^*(s)$, calculate

$$V_{t+1}^*(s) = \max_{a \in \mathcal{A}} \sum_{s'} P(s'|s, a) \left[R(s, a, s') + \gamma \, V_t^*(s')\right] \qquad \forall s \in S \tag{5.1}$$

   where $V_t^*(s) = \max_{\pi} V_t(s)$, i.e., the expected sum of discounted rewards accumulated when starting from state s and acting optimally for a horizon of t time steps. Equation (5.1) is called *value update* or *Bellman update*.

We can think of value iteration as the computation of the sequence $V_0^*, V_1^* ..., V_T^*$ until

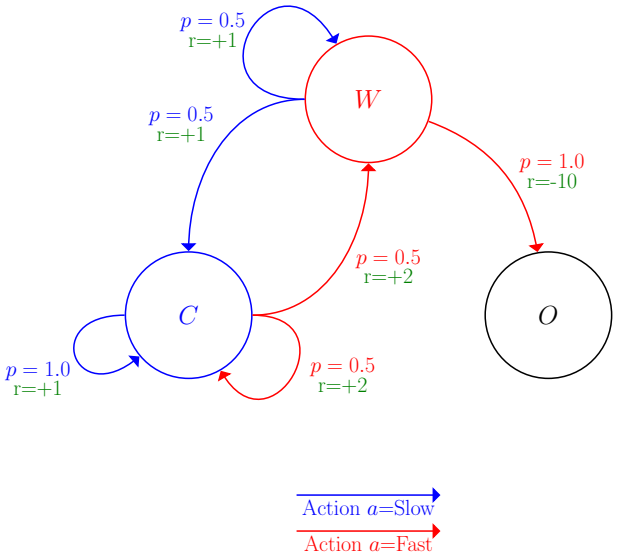


Figure 5.2: MDP of car example. States $\{C, W, O\}$ correspond to *cool*, *warm* and *overheated*, respectively. Blue arrows correspond to *slow* action, while red arrows correspond to *fast* action. The probabilities of these transition are written with the same color. The rewards are written in green.

convergence is reached. Then, observing V^*, we can derive the optimal policy as:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s'} P(s'|s,a) \; [R(s,a,s') + \gamma \, V^*(s')] \, .$$

□ **Example.** This example is taken from [37, Lecture 8].

A car can be in 3 possible states: cool, warm and overheated, which are denoted by $s \in \mathcal{S} = \{C, W, O\}$ respectively. The possible actions are to go slow or fast, i.e. $a \in \mathcal{A} = \{\text{Slow}, \text{Fast}\}$. The transition probabilities and rewards are known. This MDP is represented in Figure 5.2. Note that $s = O$ is an exit state, since there are no actions to take after reaching this state.

Assuming discount factor $\gamma = 0$, we compute the first $V_0^*(s)$, $V_1^*(s)$, $V_2^*(s)$ using value

Table 5.1: Value iteration for car example.

	Cool	Warm	Overheated
V_2^*	3.5	2.5	0
V_1^*	2	1	0
V_0^*	0	0	0

iteration. The results are summarized in Table 5.1. For each state, we have to evaluate the policy values $V_t^*(s)$ as in (5.1) for the possible actions, and choose the maximum value. Clearly $V_t^*(s = O) = 0 \quad \forall t$, since the transition probability is $P(s'|s = O, a) = 0$.

For $s = C$:

- $V_1(s = C) = \begin{cases} 1 & \text{for } \pi(C) : \ a = \text{Slow} \\ 2 & \text{for } \pi(C) : \ a = \text{Fast} \end{cases}$
- $V_2(C) = \begin{cases} 3 & \text{for } \pi(C) : \ a = \text{Slow} \\ 3.5 & \text{for } \pi(C) : \ a = \text{Fast} \end{cases}$

The corresponding optimal values are $V_1^*(C) = 2$ and $V_2^*(C) = 3.5$.

For $s = W$:

- $V_1(s = W) = \begin{cases} 1 & \text{for } \pi(W) : \ a = \text{Slow} \\ -10 & \text{for } \pi(W) : \ a = \text{Fast} \end{cases}$
- $V_2(W) = \begin{cases} 2.5 & \text{for } \pi(W) : \ a = \text{Slow} \\ -10 & \text{for } \pi(W) : \ a = \text{Fast} \end{cases}$

The corresponding optimal values are $V_1^*(W) = 1$ and $V_2^*(W) = 2.5$.

■

5.2 Reinforcement Learning

RL is a technique [39] based on MDP environment, where the goal is to train an agent to take actions maximizing the cumulative reward. The novelty, with respect to value iteration, is that the agent has no prior knowledge about the MDP, i.e., it does not know the transition probability function $P(s_{t+1}|s_t, a_t)$ and the reward function $R(s_t, a_t, s_{t+1})$. Still, the agent tries to learn the best policy from the experienced state transitions and obtained rewards. Hence, RL is a form of online learning, whereas value iteration is an offline planning method.

The agent receives feedback in the form of rewards from the environment and has to learn how to maximize them. This approach is different from other machine learning techniques, such as supervised or unsupervised learning. In fact, in RL we do not have labeled data or features to extract, but we only learn by trial and error.

In general, the learning procedure is longer, because there is a trade-off between *exploration* and *exploitation*. During exploration, the agent tries random actions to get information about the environment (states, rewards). If exploration is repeated for a long amount of time, the agent has a good overview of the environment. On the other hand, exploitation refers to the use of the acquired knowledge to choose the best action, which leads to the highest rewards.

Typically, in the first part of the learning, the agent needs to explore the environment, to obtain information about the possible transitions and rewards. One strategy is the so called ϵ-greedy exploration: the agent chooses the best action (from its current knowledge) with probability $1 - \epsilon$, or it chooses a uniformly random action with probability ϵ, with $0 < \epsilon < 1$. The parameter ϵ has to be tuned properly, usually by decreasing it during the training as the agent becomes more experienced. Ideally, with $\epsilon = 1$, the agent is

only exploring, since it always takes random actions. On the other hand, with $\epsilon = 0$, the agent is always exploiting, since it chooses the action that it believes to lead to the highest future reward.

5.3 Q-Learning

Q-learning [40],[41] is a form of model-free RL. The environment is an MDP, but reward and transition probability functions are not known. Q-learning is a modified version of value iteration. The values involved in the iteration are called *Q-values* or *action-values.* They express the value of taking action a from state s, i.e., the cumulative expected reward. Thus, for every state s, one must track $|\mathcal{A}|$ different Q-values $Q(s, a)$, one for each possible action $a \in \mathcal{A}$.

The agent interacts with the environment, according to RL principles and experiences a sequence of state-action transitions. For each of them, at a generic time instant t, the agent updates the Q-values in the following way:

- observe the current state s_t;
- select and perform an action a_t;
- observe the subsequent state s'_t;
- receive an immediate reward r_t;

- update the Q-values according to learning rate α and discount factor γ with

$$Q_t(s,a) = \begin{cases} (1-\alpha)\, Q_{t-1}(s,a) + \alpha\left[r_t + \gamma \max_{a'} Q_{t-1}(s',a')\right] & \text{if } \begin{cases} s = s_t \\ a = a_t \end{cases} \\ Q_{t-1}(s,a) & \text{otherwise} \end{cases} \tag{5.2}$$

  assuming that the initial values $Q_0(s,a)$ are known for all states and actions. This is the *Bellman equation* related to the Q-learning scenario. For each experienced transition, starting from state s_t, the agent updates only the Q-value corresponding to the action a_t that it performed.

Given $t \to \infty$, bounded rewards $|r_t| \leq \mathcal{R}$, learning rate $0 \leq \alpha < 1$, it is proven that $Q_t(s,a) \to Q^*(s,a)$, i.e., the Q-value for the optimal policy $\pi^*(s) = \arg\max_a Q^*(s,a)$, with probability 1 [41]. This convergence is guaranteed even if the agent acts suboptimally. The drawback is that then one has to observe a larger number of transitions to find the optimal policy. This is called *off-policy* learning. Some useful examples of Q-learning can be found in [42].

Hence, Q-learning provides a method to map the state to a vector of action-values, as illustrated in Figure 5.3. In Chapter 6, we will propose a neural network architecture to estimate the Q-values.

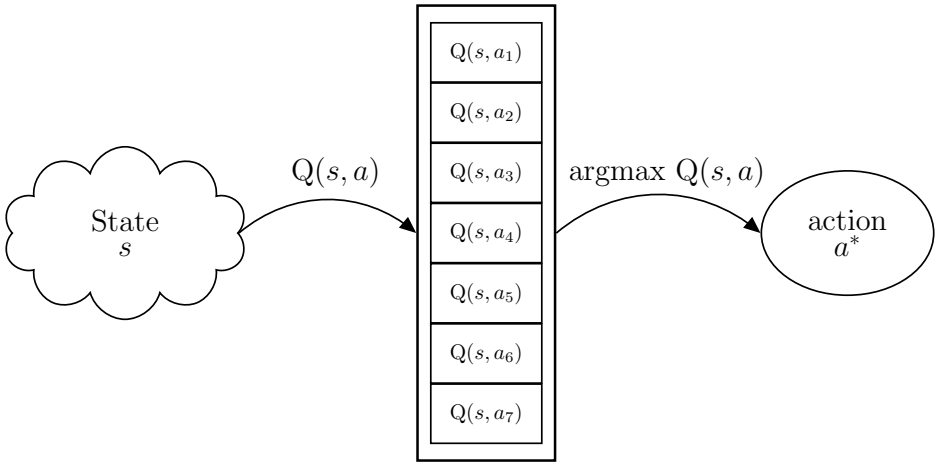


Figure 5.3: Q-function mapping, from state s to action a', through Q-values $Q(s, a_i)$.

Chapter 6

Learning Bit-Flipping Decoding

"Exploration is really the essence of the human spirit."

Frank Borman

In Chapter 5, we have provided an overview of RL. In this chapter, we consider the application of RL to a specific communications problem, i.e., the decoding of linear block codes. Inspired by the results achieved on videogames [1], [6], [36], we approach the decoding problem as a game. Our "player" is a *virtual agent*, moving in the *decoding space*, which is modeled as a Markov Decision Process (MDP). In particular, we consider a Bit-Flipping (BF) decoding scenario working as follows. Given a syndrome, which is a function of the word received from the noisy channel, the possible *actions* are to flip one of the bits, if assumed to be wrong, or to accept the codeword, if assumed to be correct. We use the Reed-Muller codes RM(2,5) and RM(3,7) introduced in Section 2.5. We use a PC matrix $\boldsymbol{H}$ composed of Minimum-Weight Parity Checks (MWPCs). As in every game, the agent receives rewards according to the achieved goals. In our decoding game, the highest reward is achieved when the agent recovers the transmitted codeword. The agent

is trained with a Q-learning Neural Network (NN) model, with no previous knowledge about the environment and/or the code structure, because the state transitions and the rewards are intrinsically specified in the MDP environment, by means of the PC matrix $\boldsymbol{H}$. This approach allows us to train agents that potentially could learn to decode any linear block code, given a sufficient number of training episodes and well-shaped rewards, and potentially achieve better performance than other low-complexity algorithms. All the experiments assume transmission over the Binary Symmetric Channel (BSC).

In Section 6.1, we explore the details of our decoding environment class, and the framework that supports our experiments. Section 6.2 defines the deep RL approach and possible NN architectures. In Section 6.3, we present our application to RM codes and discuss the results.

6.1 Decoding Environment

The game environment is a crucial part of the experiments, because it provides the test-bed to run simulations. Inspired by the algorithmic environment of OpenAI Gym [36], we developed our Python class named *DecodingEnv*. The conceptual diagram is illustrated in Figure 6.1.

This class models the MDP for the BF decoding scenario and its objectives are the following.

- Generate a length-k message and encode it with the code generator matrix $\boldsymbol{G}$ to obtain a length-n codeword.
- Add noise to the codeword using the selected channel model.
- Compute the length-m syndrome $\boldsymbol{s} = \boldsymbol{H}\,\boldsymbol{z}$ for the noisy received word $\boldsymbol{z} \in \mathbb{F}_2^n$,

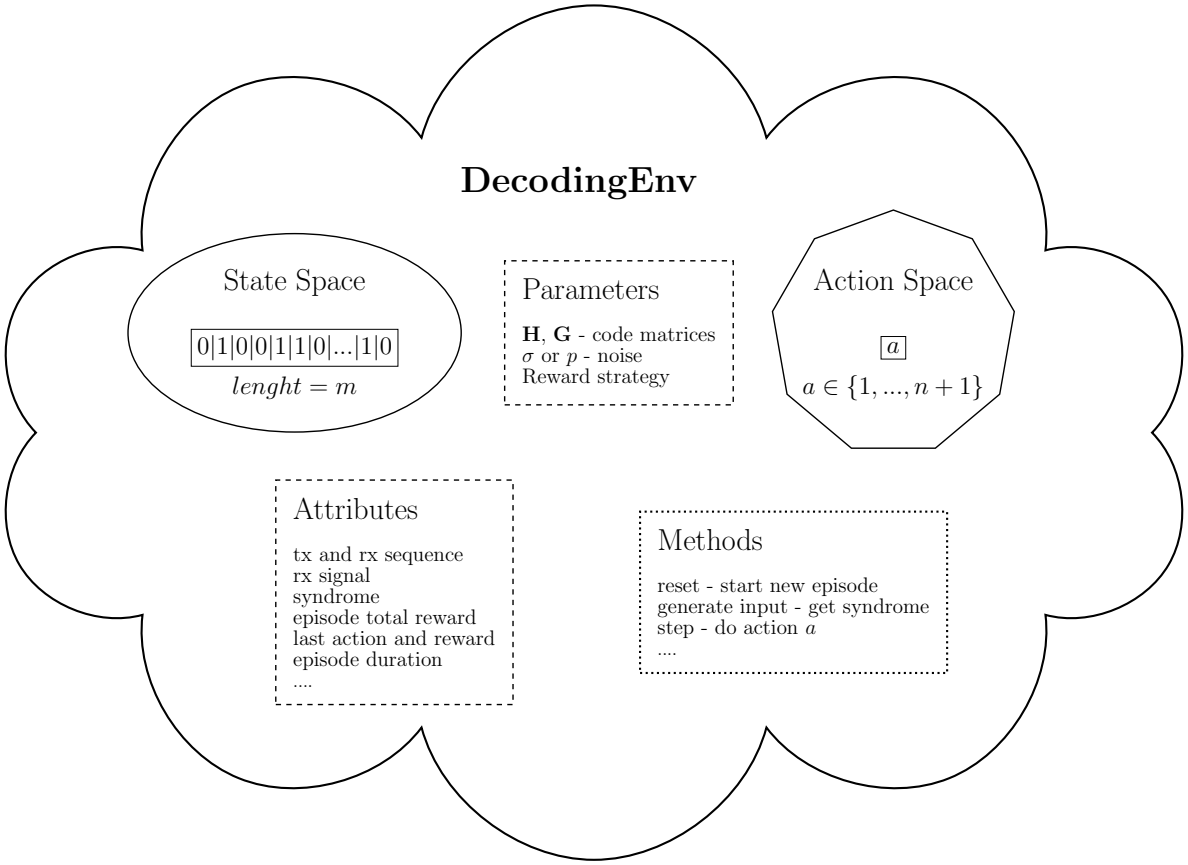


Figure 6.1: DecodingEnv diagram. Each container represents an element of the class.

where $\boldsymbol{H}$ is the code PC matrix. This is the *state* of the environment.

- Perform action a, according to BF decoding scenario:
  - if $1 \leq a \leq n$, flip code bit at position a;
  - if $a = n+1$, no flipping is performed. This action is called no-operation (no-op).
- After each action, define the next state and return the appropriate reward.
- Display the environment realizations, such as state, action and reward.

The structure is very flexible and allows one to shape the experiments as one desires. It can also provide states in form of the codeword, the syndrome, the received signal or a

combination of all of these. The action can be any possible move, the only requirement is that it has to be numerically represented. The reward strategy can be customized and redefined at any time. The communication channel can be BSC, with transition probability p, or BI-AWGN, with noise variance σ^2. It can also be changed at any time. For the experiments of this thesis, we only use the syndrome as state and the BSC as communication channel.

The class is initialized providing the generator matrix $\boldsymbol{G}$, the PC matrix $\boldsymbol{H}$, the reward strategy and the noise parameter. A flowchart of the tasks is illustrated in Figure 6.2. Every episode, which correspond to a new received word (and syndrome), is started by calling the *reset*() method. Then, the agent chooses the next action to perform through the *action*($\cdot$) method. If the next state is not terminal, the agent chooses another action, otherwise it calls *reset*() to start a new episode.

6.1.1 State Transitions

We assume that the state of the MDP is the syndrome of the received word. Then, the possible transitions between states are determined by the code structure. More precisely, they are defined by the code PC matrix $\boldsymbol{H}$. The syndrome is computed as $\boldsymbol{s} = \boldsymbol{H}\,\boldsymbol{z}$, where $\boldsymbol{z} \in \mathbb{F}_2^n$ is the received word. Given an action a, i.e., flip a bit z_i of the received word or no-op, a new syndrome is computed, hence the next state is determined. Then, the code PC matrix $\boldsymbol{H}$ underlies the MDP environment. An example of state transitions is depicted in Figure 6.3. In general, transitions are bidirectional, since the agent returns to the departing state if it flips twice the same bit. Circles represent states (syndromes) and arrows represent actions (flip a bit). States and actions are denoted with an integer, e.g., State k corresponds to a syndrome $[0, 1, 0, 0, 1, ..., 1]$, which clearly is a binary vector.

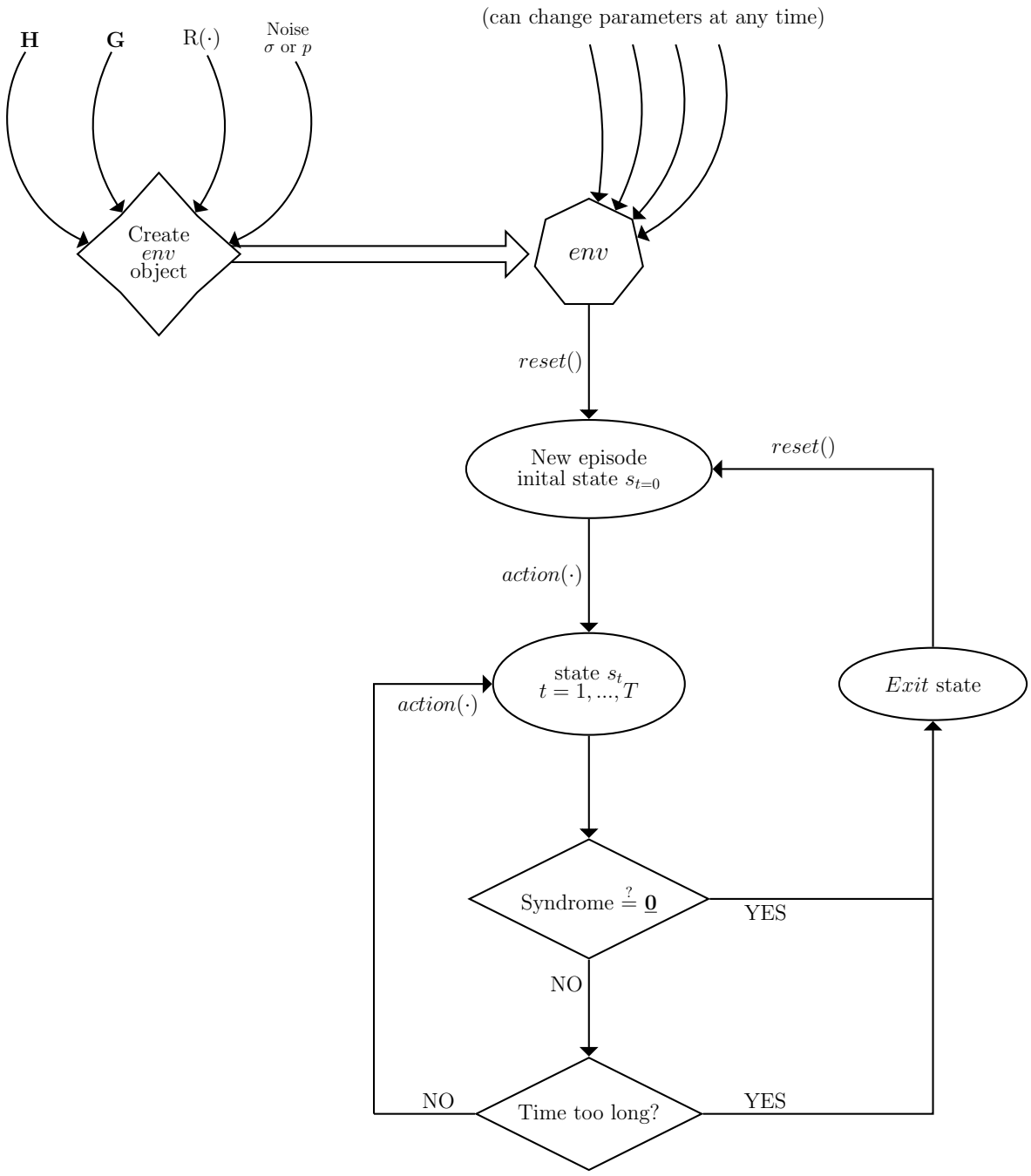


Figure 6.2: DecodingEnv flowchart.

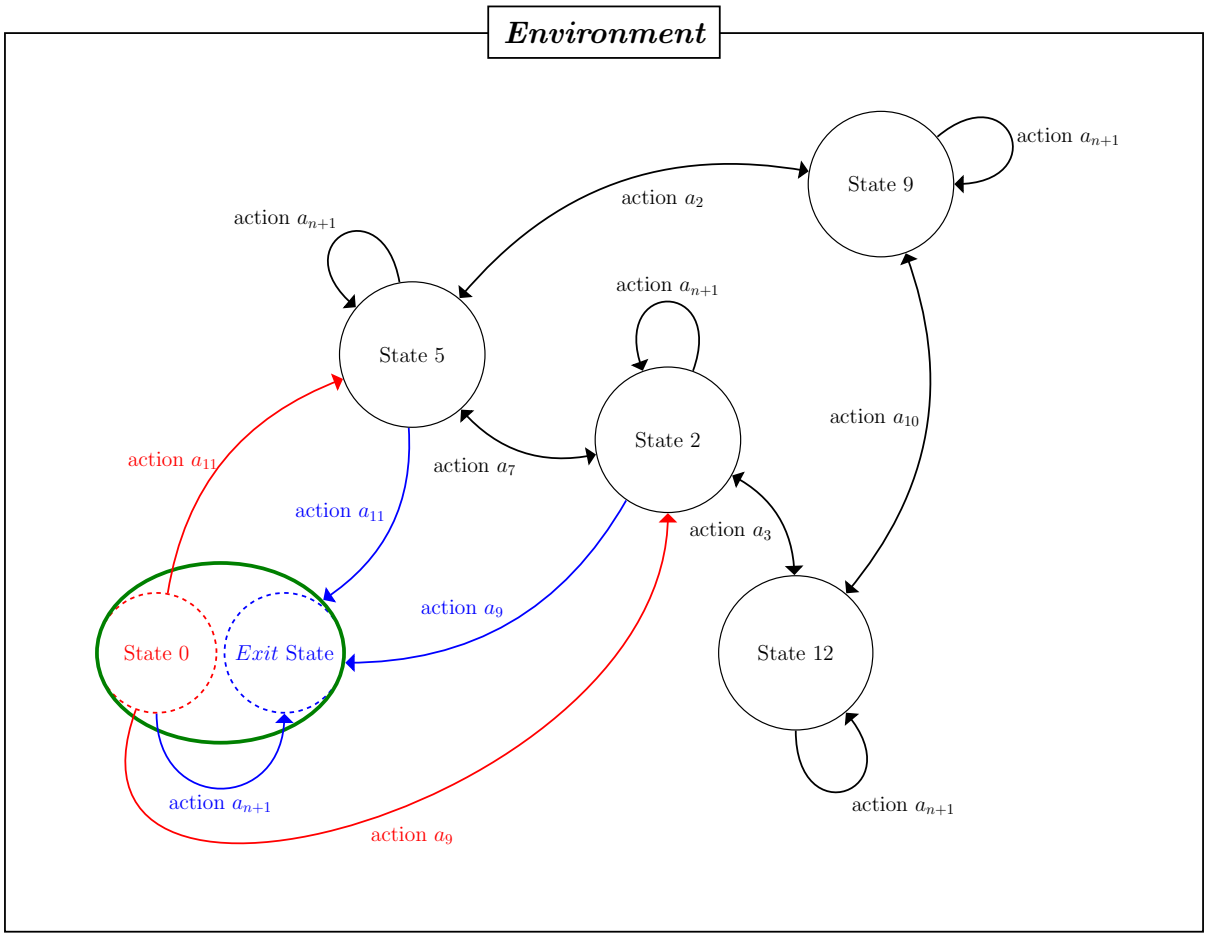


Figure 6.3: Example of state transitions.

State 0 corresponds to the all-zero syndrome. States $\{2, 5, 9, 12\}$ correspond to other syndromes, where at least one PC equation is violated. Blue arrows are directed to the exit state, in which a codeword is found. The other exit state, corresponding to the agent exceeded the episode duration, is omitted.

State 0 (green ellipse) is the all-zero syndrome, hence it is also an exit state (blue arrows) with high reward. If it is the initial state, actions that lead to other states (red arrows) can occur. Clearly these transitions are unwanted, therefore the agent has to

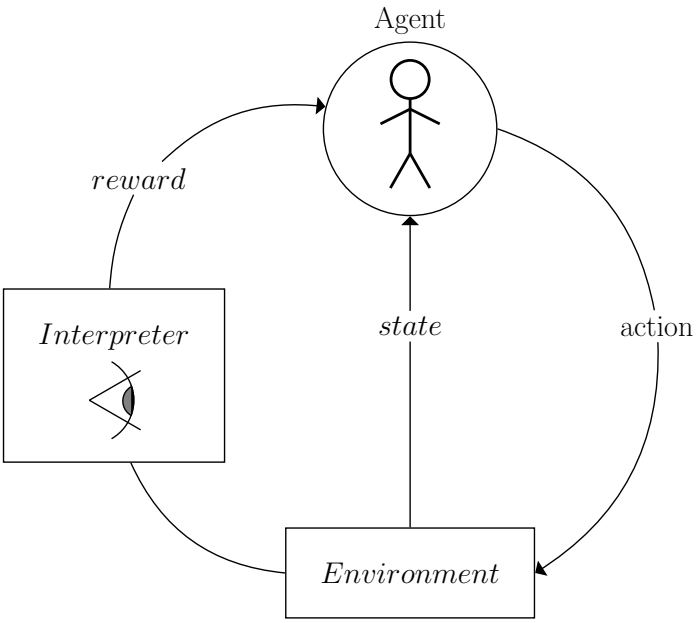


Figure 6.4: RL scheme. The interpreter box represents the reward function, implemented by the developer, for the given environment.

learn that when it is in state 0, it has to take action no-op to get the maximum reward.

There is also another hidden exit state, related to the duration of the episode: if the agent takes more than T actions, it is stopped with a proper reward.

6.1.2 Reward Strategy

Alongside the exploration/exploitation trade-off, discussed in Chapter 5, the reward strategy is another crucial point of RL algorithms. The system designer assumes the role of interpreter of the environment, as sketched in Figure 6.4. The interpreter needs to design a well-suited reward function to guide learning in the desired direction.

In sparse reward strategies a few transitions receive rewards different from zero. For example, considering Atari games, the MDP returns a high reward only when the agent wins a game and a high negative reward when the agent looses. Generally speaking, sparse

rewards are preferable, because the agent is free to find the best sequence of actions to reach the desired goal [43]. However, sometimes the rewards are too sparse and the agent cannot start to learn with random actions. This means that it will not learn a useful policy. In this case, the interpreter can change the reward function, making it more "dense", i.e., introducing small rewards when the agent reaches intermediate goals. This leads the agent to get closer to desired final result, but some sort of *human assistance* is introduced. We can consider chess game as an example. Providing a sparse reward only when the agent wins will require longer training time, but it may find game strategies that beat human performance. On the other hand, dense rewards, i.e., rewards whose objective is to mimic some known strategy, will very likely guide the agent to learn the policy proposed by human knowledge.

We consider the reward events listed in the next paragraph to design our reward strategy. All the reward entries are cumulative, that is the agent can experience more than one of these events and get the sum of correspondent rewards. The possible causes of reward are described as follows.

1. True CW: a codeword is detected (all-zero syndrome) and it corresponds to the transmitted one.

2. Other CW: a codeword is detected (all-zero syndrome), but it is not the transmitted one.

3. Repeat act: the same action is repeated, i.e., $a_t = a_{t-1}$.

4. Too long: the agent took a number of actions $k > T$, there T is the maximum number of actions for each episode.

5. Useless no-op: action no-op was chosen, but the syndrome is not equal to all-zero.

6. PCs net change: consider the net balance between the number of violated PCs at time $t-1$ versus time t. The reward is determined according to

$$r_t = c \cdot (d_{t-1} - d_t)$$

where c is a scaling coefficient, d_t is the number of violated PCs and r_t is the reward returned by the environment, at time t. We assume $c > 0$. Clearly, $r_t < 0$ for $d_t > d_{t-1}$, i.e., the agent gets a negative reward if the number of violated PC increases with respect to the previous time instant.

Points 1,2 and 4 are related to exit states. The main goal for the agent is to converge to a codeword, hopefully the correct one. This is why 1 and 2 are the highest rewards, so that the agent will be attracted to this long term rewards. After the experiments, we noticed that a penalty for taking more than T actions is not required.

Points 3,5 and 6 are related to actions' feedback. We want to avoid the repetition of actions, since the agent would return to the same state if it flips twice the same bit. In particular, 5 and 6 are a sort of continuous action-feedback, about the utility of the action in terms of intermediate goals, e.g., decrease the number of violated PCs. Moreover, 6 determines a positive (negative) reward if the action decreases (increases) the number of violated PCs.

6.2 Deep RL Architectures

RL is a powerful algorithm, but for many real-world problems the environment can be too complex to successfully train the agent. NNs, which are introduced in Chapter 3, can provide an excellent tool to compute an approximation of the Q-function, introduced in

Chapter (5). In the next paragraphs, we explain some possible NN architectures.

A Deep Q-Network (DQN) architecture is proposed in [1], which can be used to approximate the Q-function. It has been applied to Atari games[1] as benchmark. This architecture includes several convolutional layers to extract features from the game frames, followed by two Fully Connected (FC) layers that map the features to the Q-values. Each hidden layer is followed by Rectifier Linear Unit (ReLU) activation function.

We use the syndrome as state, hence the input data for the NN is a binary vector. Then, the convolutional layers are not required and we can design a simpler architecture with only one FC layer. The NN maps the state to the estimated Q-function, through a sufficiently large hidden layer, which is FC with ReLU or tanh activation function. An example of DQN scheme for our experiment is depicted in Figure 6.5.

In [44], the Dueling network architecture is proposed as the current state-of-the-art for Deep Q-learning in Atari domain. It can be seen as an evolution of DQN. After some convolutional layers, as in DQN, this architecture explicitly separates the representation of the state values and action advantages, building two different streams of FC layers that are then combined in a final aggregating layer to produce the estimated Q-function, as shown in Figure 6.6. Again, since our experiment processes binary vectors, the convolutional layers are not required for our setup. The final configuration of the two different streams is the main difference with respect to DQN [1]. The intuition behind this scheme is that this NN can learn which states are valuable, without having to learn the effect of each action for each state.

In the next paragraphs, we report the main concepts from [44]. Consider an MDP as introduced in Chapter 5, characterized by states $s_t \in \mathcal{S}$, actions $a_t \in \mathcal{A} = \{1, ..., |\mathcal{A}|\}$ and rewards r_t at time step t. The agent's goal is to maximize the discounted reward

[1] Wikipedia: Atari games. https://it.wikipedia.org/wiki/Atari_Games

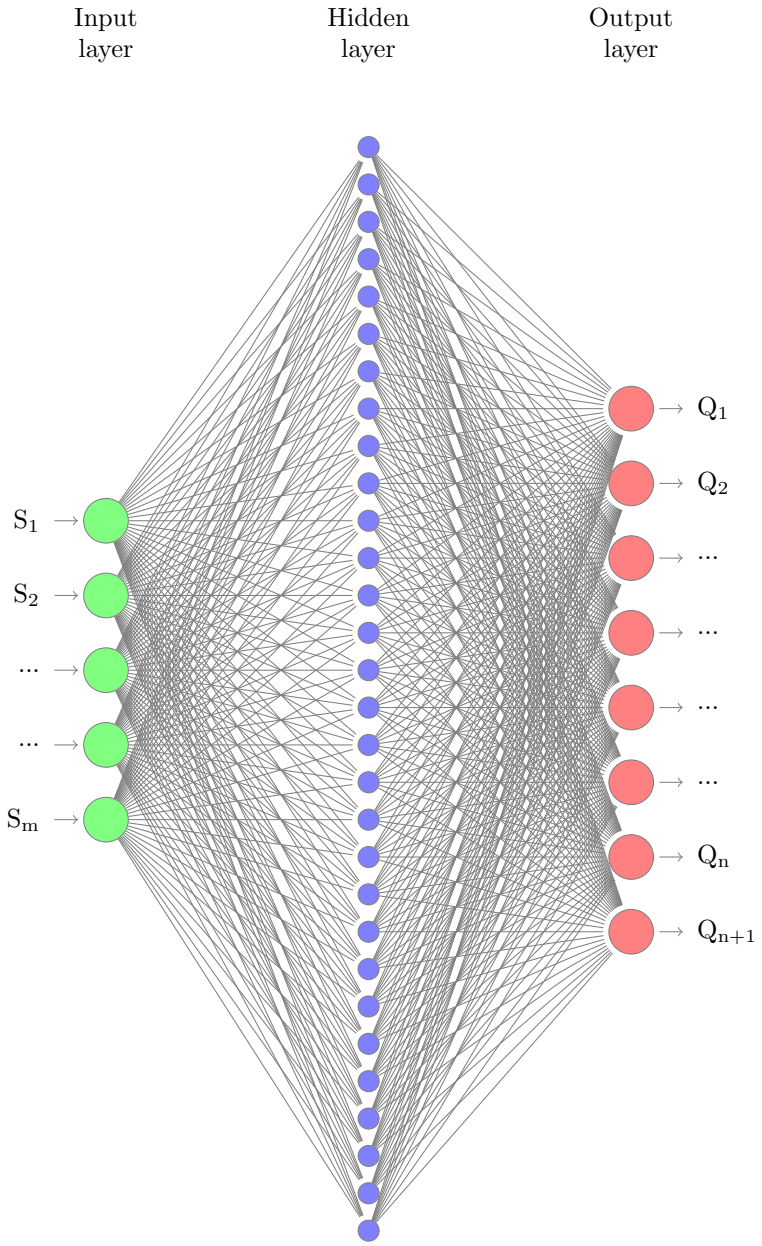


Figure 6.5: Example of DQN model for our experiment. The input layer (green) corresponds to the syndrome and the output layer (red) corresponds to the vector of estimated Q-values. The hidden layer (blue) is composed by a certain number of neurons, at least greater than the input/output size.

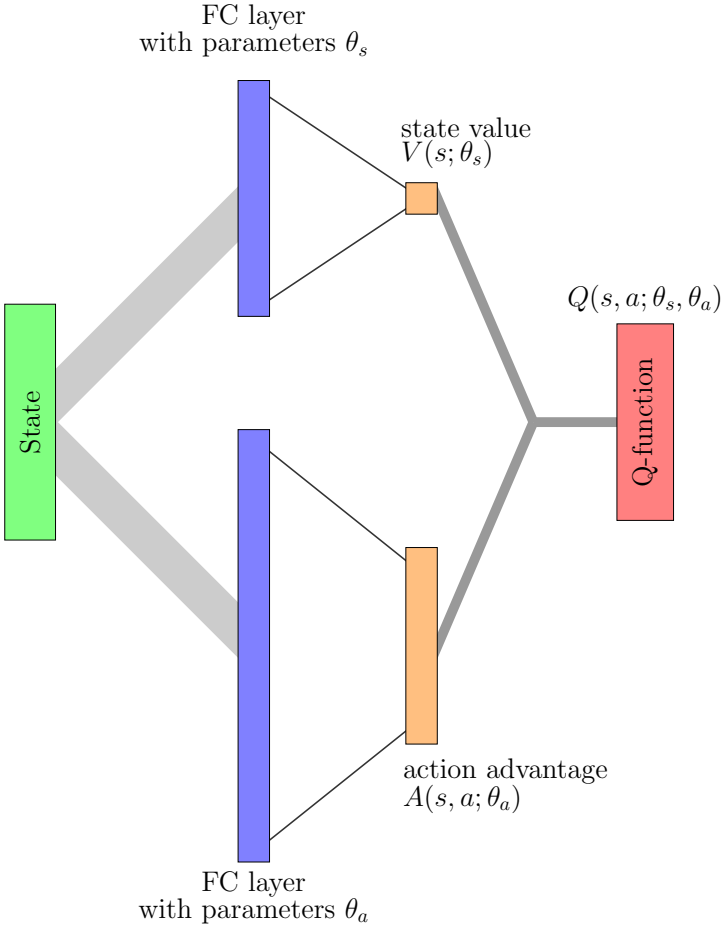


Figure 6.6: Dueling network architecture.

$R_t = \sum_{\tau-t}^{T} \gamma^{\tau-t} r_\tau$, where γ is the discount factor and T is the maximum number of steps per episode. If the agent behaves according to a stochastic policy $a \sim \pi(s)$, the state-action value, or Q-value, of (s, a) is defined as

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}\left[R_t \middle|\, s_t = s, a_t = a, \pi\right] \\ &= \mathbb{E}_{s'}\left[r + \gamma\, \mathbb{E}_{a' \sim \pi(s')}\left[Q^\pi(s', a')\right] \middle|\, s, a, \pi\right] \end{aligned}$$

where r is the reward returned by the MDP, γ is the discount factor, $\mathbb{E}_{a' \sim \pi(s')}\left[Q^\pi(s', a')\right]$ is the average Q-value for the next state s'.

The state value V is defined as

$$V^\pi(s) = E_{a \sim \pi(s)}\left[Q^\pi(s, a)\right]$$

and the $|\mathcal{A}|$-dimensional action-advantage function A is

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s). \tag{6.1}$$

Note that $E_{a \sim \pi(s)}\left[A^\pi(s, a)\right] = 0$. Intuitively, the value function V measures how good a particular state is, while the action-advantage function A expresses the relative importance of each action. Then, the optimal Q-function, which satisfies the Bellman equation, is defined as

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q^\pi(s,a) \\ &= E_{s'}\left[r + \gamma \max_{a'} Q^*(s', a') \middle|\, s, a\right]. \end{aligned}$$

In deep Q-learning, the NN is used as function approximation tool to obtain a pa-

rameterized version of $Q(s,a)$. From the architecture [44], depicted in Figure 6.6, we denote the parameterized state value as $V(s;\theta_s)$ and the parameterized action-advantage as $A(s,a;\theta_a)$, where, θ_s and θ_a denote the parameters (the set of weights and biases, as explained in Chapter (3)) of the two streams of FC layers. The final aggregated output, i.e., the parameterized Q-function, is defined as

$$Q(s,a;\theta_s,\theta_a) = V(s;\theta_s) + \left[A(s,a;\theta_a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a';\theta_a) \right]. \tag{6.2}$$

Hence, the advantages A only need to change as fast as the mean. Equation (6.2) is viewed as part of the network and not as a separate algorithmic step, so that the estimates of $V(s;\theta_s)$ and $A(s,a;\theta_a)$ are computed automatically without any extra supervision. The input-output mapping is the same of DQN, so we can also reuse all learning algorithms designed for Q network training.

6.3 Learning to Decode the RM Codes

We consider the RM codes described in Section 2.5. In particular, we use RM(2,5) and RM(3,7), which are described in Table 2.1. The experiments for this part of the thesis are performed on the BSC.

6.3.1 Proposed Model

The goal is to estimate the Q-function through a NN, as explained in Section 6.2. We use the Dueling architecture [44] for all the experiments in this thesis.

The input of the NN is the state s of the MDP

$$s \in \mathcal{S} = \{\boldsymbol{s'} \in \mathbb{F}_2^{m'}, \boldsymbol{z} \in \mathbb{F}_2^n | \boldsymbol{s'} = \boldsymbol{z}\,\boldsymbol{H'^T}\}$$

which corresponds to the syndrome $\boldsymbol{s'} = (s_1, ..., s_{m'})$. The vector $\boldsymbol{z}$ represents the received binary sequence and the state space $\mathcal{S}$ is composed of all the possible combination of the product of $\boldsymbol{z}$ and the PC matrix $\boldsymbol{H'}$, i.e., all possible syndromes.

The action

$$a \in \mathcal{A} = \{1, ..., n, n+1\}$$

corresponds to flipping the a-th bit, if $a < n+1$, or no-op, if $a = n+1$.

The output of the NN is the estimate of the Q-function $Q(s, a; \theta)$, where θ represents the parameters of the NN. This output is an $|\mathcal{A}|$-dimensional vector, with $|\mathcal{A}| = n+1$.

We use $\tanh(\cdot)$ activation for the hidden layers and batch normalization to speed up the training [45].

In [43], multiple strategies are analyzed for some simple tasks. They claim that "0/1" and "discretized" rewards (i.e. a few reward values) work better. From our experience, training with a "0/1" reward does not work well for our task, while "discretized" rewards achieve good performance. On the other hand, dense rewards seem to confuse the agent. They induce to learn a policy that does not converge to decode codewords, but rather obtain short term rewards for intermediate goals (e.g., points 5 and 6 of Section 6.1.2). In the end, it will not learn an effective decoding strategy; thus, giving too much human assistance is not the right approach for this application of artificial intelligence. We tried different reward values for points 1-6 of Section 6.1.2. At the end, from the above considerations, we found that the reward strategy of Table 6.1 produces the desired results for this MDP modeling BF decoding.

Table 6.1: Reward strategy for out experiments. The events are described in Section 6.1.2.

Event	Reward r_t
1. True CW	50
2. Other CW	5
3. Repeat act	-40
4. Too long	0
5. Useless no-op	0
6. PCs net change	0

6.3.2 Agent Training and Testing

The NN is used by the agent to choose the action to take at every time step, selecting $a = \arg\max_{a'} Q(s, a'; \theta)$. The agent starts with no prior knowledge about the code and the MDP. According to the RL principles, described in Chapter 5, Q-learning is a model-free and off-policy algorithm that allows the agent to converge to the optimal Q-values even if it acts suboptimally. In the next paragraphs, we explain our choices for the proposed RL decoder.

For the first portion of training episodes, whose ending time is denote as $T_{\mathrm{G}}^{\mathrm{end}}$, we adopt the following ϵ-greedy exploration policy to choose action a

$$a = \begin{cases} U(1, ..., n+1) & \text{with probability } \epsilon_G \\ \arg\max_{a'} Q(s, a'; \theta) & \text{with probability } (1 - \epsilon_{\mathrm{G}}) \end{cases}$$

where $U(1, ...k)$ is a function that picks an integer from 1 to k with uniform probability $1/k$ and ϵ_{G} is the probability of ϵ-greedy exploration. The ϵ-greedy policy allows the agent to explore the MDP for the first part of the training, obtaining knowledge about

the state transitions and related rewards. The coefficient ϵ_{G} is discounted by $\epsilon_{\mathrm{G}}^{\mathrm{discount}}$ every T_{G} episodes, namely ϵ_{G} is update with $\epsilon_{\mathrm{G}} \leftarrow \epsilon_{\mathrm{G}} \cdot \epsilon_{\mathrm{G}}^{\mathrm{discount}}$ at time $t = j \cdot T_{\mathrm{G}} < T_{\mathrm{G}}^{\mathrm{end}}$ for integer j.

In addition to ϵ-greedy exploration, we also adopt another exploration strategy which we call ϵ-right exploration. According to [46], goal-oriented trajectories can improve the agent's performance, by choosing actions that lead to short term goals during the exploration phase. In our case, we induce the agent to choose one of the "good" actions that flips an incorrect bit, with probability ϵ_{R}, during the ϵ-right exploration phase. Then, the ϵ-right exploration policy is the following

$$a = \begin{cases} C(\boldsymbol{z}) & \text{with probability } \epsilon_R \\ \arg\max_{a'} Q(s, a'; \theta) & \text{with probability } (1 - \epsilon_{\mathrm{R}}) \end{cases}$$

where $C(\boldsymbol{z})$ is a function that select randomly one of the actions a which corrects a bit z_i. The ϵ_R coefficient is maintained constant and this policy is adopted for the first $T_{\mathrm{R}}^{\mathrm{end}}$ training episodes.

For the training process, we choose a noise parameter that is consistent with the decoding capability, i.e., the noise is chosen in order to produce a number of errors that may be corrected by the related code.

The goal is to estimate the Q-function, so a target value is needed to apply the gradient descent. The target function y_t at time instant t is computed according to Q-learning in the following way

$$y_t = \begin{cases} r_t & \text{if } s_{t+1} \text{ is terminal} \\ r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{otherwise} \end{cases}$$

where r_t is the reward returned by the MDP, γ is the discount factor and $Q(s_{t+1}, a'; \theta)$ is the output of the NN with parameters θ. Then, $\max_{a'} Q(s_{t+1}, a'; \theta)$ represents the maximum expected future reward for the future state s_{t+1}.

The training procedure is described in Algorithm 6.1. We also use experience replay [1], [44], i.e., we store the experience (state, action, target Q-value) into the replay memory $\mathcal{D}$. Then, every C episodes, we apply the gradient descent step to update the NN parameters, sampling the experienced transitions of a certain number of episodes, defined as F, from the replay memory $\mathcal{D}$. This allows us to increase the data efficiency and reduces the variance of the updates, since uniform sampling from the replay memory reduces the correlation among the samples [1].

We use the $\ell 2$ loss function, which measures squared distance penalty between the NN output and the Q-learning target y. Finally, the training hyperparameters that we used are listed in Table 6.2.

6.3.3 Reference Performance Algorithms

We compare the performance of the agent to the standard BF decoding algorithm, which is performed in the BSC. The idea of standard BF is to flip the bit that corrects most parity checks. A deeper introduction to this algorithm can be found in [2, Chapter 10]. This algorithm is listed in Algorithm 6.2.

We also compare the agent's performance to Maximum-Likelihood (ML) decoding.

6.3.4 Results for RM(2,5) and RM(3,7) Codes

We consider RM codes RM(2,5), whose length is $n = 32$, and RM(3,7), whose length is $n = 128$. Both of these codes have rate equal to 0.5. We use redundant PC matrices,

Algorithm 6.1 Q-function NN learning algorithm. Adapted from [1].

Initialize NN with parameters θ.
Initialize replay memory $\mathcal{D}$ to capacity D^{mem}.
For episode $i = 1$, K **do**
  Initialize state s_1.
  For t = 1,T^{agent} **do**
    With probability ϵ_R select a "right" action a_t,
    With probability ϵ_G select a random action a_t,
    otherwise select $a_t = \arg\max_a \hat{Q}(s_t, a)$.
    Execute action a_t and observe reward r_t and the next state s_{t+1}.
    Set the target y_t to

$$y_t = \begin{cases} r_t & \text{if } s_{t+1}\text{is terminal} \\ r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{otherwise} \end{cases}$$

    Store transition (s_t, a_t, y_t) in memory $\mathcal{D}$.
  End For
  Every C episodes:
  sample F episodes' transitions from $\mathcal{D}$, perform gradient descent step on

$$\mathbb{E}_{(s_t,a_t,y_t)^F \sim \mathcal{U}(\mathcal{D})} \left[(y_{t'} - Q(s_{t'}, a_{t'}; \theta))^2 \right]$$

  and update the network parameters θ through backpropagation.
End For

Table 6.2: Learning hyperparameters for our RL model.

Hyperparameter	value
discount factor γ	0.9
number of training episodes K	$8 \cdot 10^5$
ϵ-greedy starting probability ϵ_{G}	0.8
ϵ-greedy discount $\epsilon_{\mathrm{G}}^{\mathrm{discount}}$	$1/3$
ϵ-greedy discount interval T_{G}	$5 \cdot 10^4$
ϵ-greedy ending episode $T_{\mathrm{G}}^{\mathrm{end}}$	$0.7 \cdot K$
ϵ-right probability ϵ_{R}	0.05
ϵ-right ending episode $T_{\mathrm{R}}^{\mathrm{end}}$	$0.7 \cdot K$
memory $\mathcal{D}$ size in episodes	100
update gradient interval C	5
number of episodes for each gradient update F	10

whose rows contain MWPCs [18], and we denote as $\boldsymbol{H}_{\mathrm{oc}}$ the overcomplete matrix that contains all the MWPCs for a given RM code. We randomly pick m' rows from the overcomplete matrix, which is composed of 620 and 94488 MWPC equations for the RM(2,5) and RM(3,7), respectively. Then, we build the PC matrix $\boldsymbol{H}'$, of dimension $m' \times 32$, to perform decoding over the BSC.

We use the Dueling architecture, shown in Figure 6.6, where the hidden layers have variable dimensions. The number of neurons of the hidden layers related to the state value $V(s;\theta_s)$ and action-advantage $A(s,a;\theta_a)$ are denoted with U_a and U_v, respectively.

Algorithm 6.2 Bit-Flipping (BF) decoding algorithm for BSC. Adapted from [2].

1. Compute the syndrome component $s_m = \sum_{n=1}^{N} z_n H_{mn}$ from $\boldsymbol{z}$
2. Compute $k^e = \sum_{m=1}^{M} s_m$
3. For $n = 1, 2, ..., N$, build a new set of syndromes $\boldsymbol{s^{(n)}}$, computed by flipping bit z_n; then compute $k_n = \sum_{m=1}^{M} s_m^{(n)}$.
4. For $n = 1, 2, ..., N$, compute: $e_n = k^e - k_n$
5. If $\max_{1 \leq n \leq N} e_n > 0$, flip bit z_n for

$$n = \arg \max_{1 \leq n \leq N} e_n.$$

Steps 1-5 are repeated until all the parity check equations are satisfied or the maximum number of iterations is reached.

$\boldsymbol{c} = (c_1, ..., c_N)$ is the binary codeword
$\boldsymbol{x} = (x_1, ..., x_N)$ is the transmitted BPSK signal
$\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{w} = (y_1, ..., y_N)$ is the received signal, where $\boldsymbol{w} \sim \mathcal{N}(0, \sigma^2)$
$\boldsymbol{z} = (z_1, ..., z_N)$ is the hard-decided binary word, based on $\boldsymbol{y}$

Then, we define

$$U_a = m' \cdot u_a$$

$$U_v = m' \cdot u_v$$

where the coefficients u_a and u_v represent the expansion factors with respect to the input size m'. If $u = 1$, then the hidden layer has a number of neurons equal to the input dimension. If $u > 1$, the number of hidden neurons is greater than the input size.

For RM(2,5) we choose $m' = [16, 32, 64, 128]$. The related performance curves in terms of Codeword Error Rate (CER) vs E_s/N_0 are shown in Figures 6.7 and 6.8. Note that for all the considered PC matrices our agent performed as the standard BF decoding.

Furthermore, when the PC matrix $\boldsymbol{H'}$ is small (e.g., $m' = [16, 32]$), increasing the hidden layers size may outperform the standard BF performance. Our intuition is that the agent is able to see a higher-dimensional representation of the syndrome, exploiting the hidden layers of the NN. In this way, the model is able to find better approximations of the Q-function to choose the action.

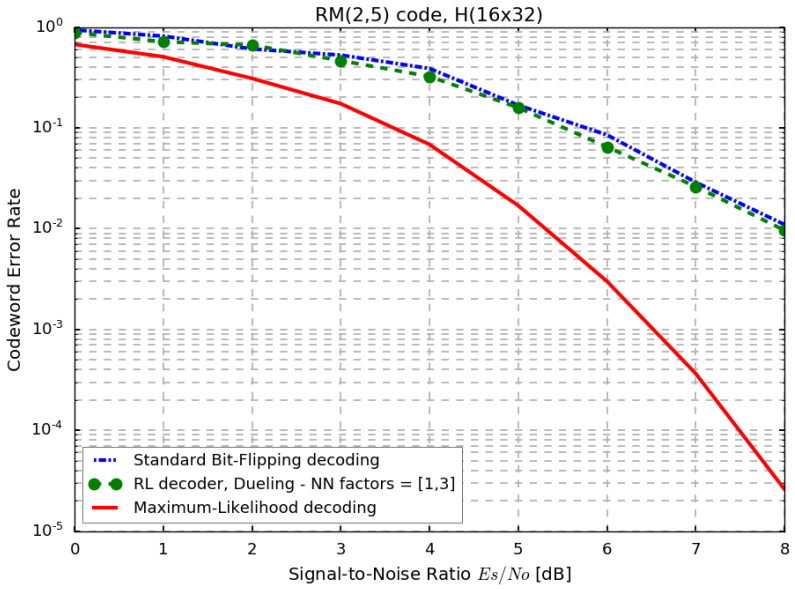
We also analyzed the decoder efficiency, defined as the ratio between the number of errors in the received word and the number of actions taken by the agent, i.e.,

$$\text{efficiency} = \frac{\#\text{ errors in the received word}}{\#\text{ actions taken by the agent}}.$$

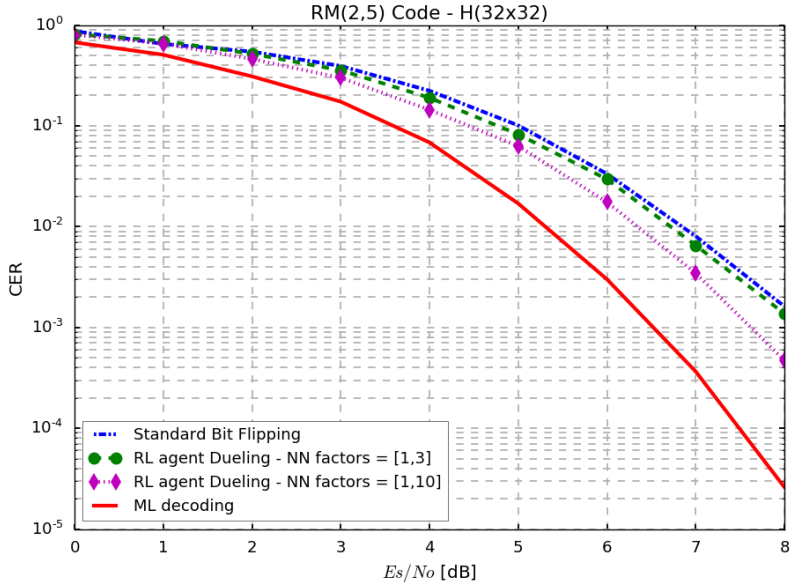
Ideally, we expect the efficiency to be close to 1, since it means that the agent has not wasted time by flipping twice a bit which was not wrong. The efficiency for the RM(2,5) and $\boldsymbol{H}'$ of size 32×32 is shown in Figure 6.9. For this plot we consider only the correctly decoded codewords at a fixed $E_s/N_0 = 4$ dB. The efficiency in this case is equal to 1 for 97% of these episodes. In the other 3% of the episodes, the agent takes a larger number of actions with respect to the errors, i.e., it has flipped twice some bit that was not wrong. So, first it has taken a wrong decision, but it was able to recover the error pattern.

For the same code, in Figure 6.10 we also show the average reward obtained from the agent during the training episodes. Note that 50 is the maximum reward achievable by the agent, according to the reward strategy of Table 6.1.

We perform the same simulation for RM(3,7), choosing $m' = [64, 128]$. The related performance curves in terms of CER vs E_s/N_0 are shown in Figure 6.11. The learning time was longer with respect to the RM(2,5) code, since the agent needs to approximate a Q-function which is an $(n+1)$-dimensional vector, with $n+1 = 129$. We noticed that as we increase the input dimension, the RL decoder obtains slightly worse performance with

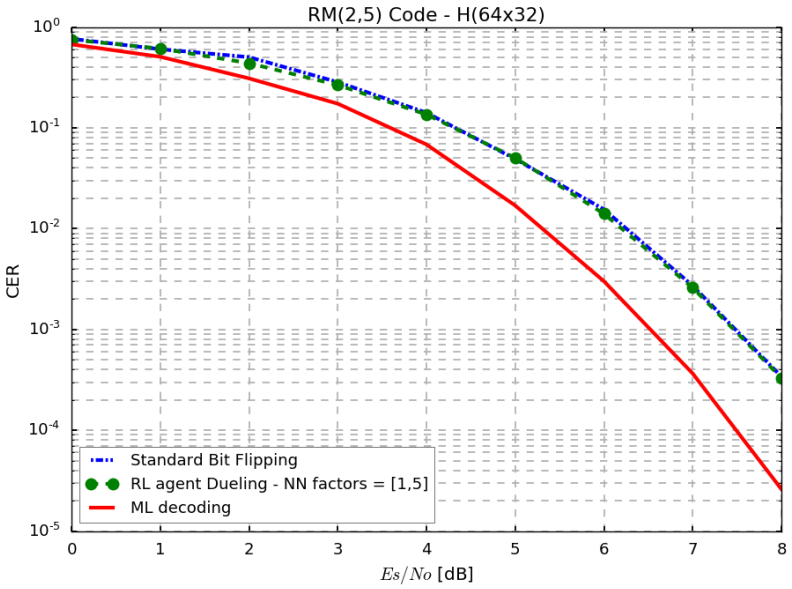


(a) $m' = 16$, $u_a = 1$, $u_v = 3$.

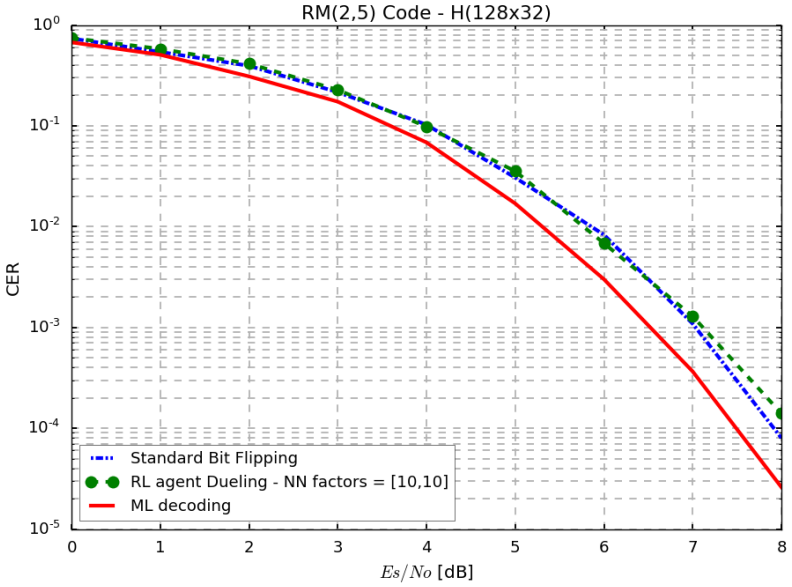


(b) $m' = 32$, $u_a = 1$, $u_v = 3$ (green) and $u_v = 10$ (magenta).

Figure 6.7: CER vs E_s/N_0 for RM(2,5) code, using $m' = [16, 32]$ random MWPCs randomly chosen from the overcomplete matrix $\boldsymbol{H}_{\text{oc}}$, with size 620×32. The blue curve is related to standard BF performance and the red one is ML performance over the BSC.



(a) $m' = 64$, $u_a = 1$, $u_v = 5$.



(b) $m' = 128$, $u_a = 10$, $u_v = 10$.

Figure 6.8: CER vs E_s/N_0 for RM(2,5) code, using $m' = [64, 128]$ random MWPCs randomly chosen from the overcomplete matrix $\boldsymbol{H}_{\text{oc}}$, with size 620×32. The blue curve is related to standard BF performance and the red one is ML performance over the BSC.

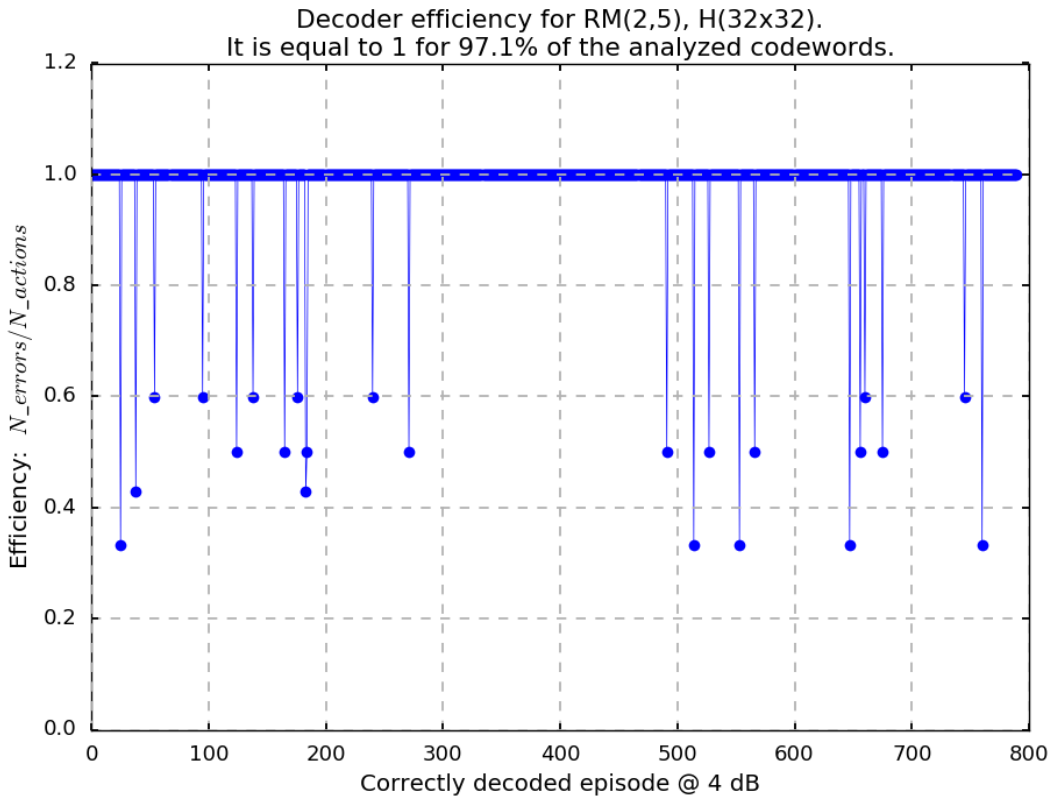


Figure 6.9: Decoder efficiency for RM(2,5), using $\boldsymbol{H}'$ of size 32×32. This plot is related to the correctly decoded codewords at $E_s/N_0 = 4$ dB. The efficiency is expressed as the ratio between the number of errors in the codeword and actions needed to correct them. Note that the efficiency is equal to 1 for 97.1% of the codewords that are considered.

respect to standard BF. This may need further investigation to improve the RL model, in order to allow the NN to learn from longer codes with larger syndrome and Q-value vectors.

6.3.5 Conclusions

In this chapter, we proposed a novel application of RL to decoding of linear block codes. We observed that our RL agent is able to achieve the same performance of the standard BF decoder for the RM codes RM(2,5) and RM(3,7), using random subsets of the

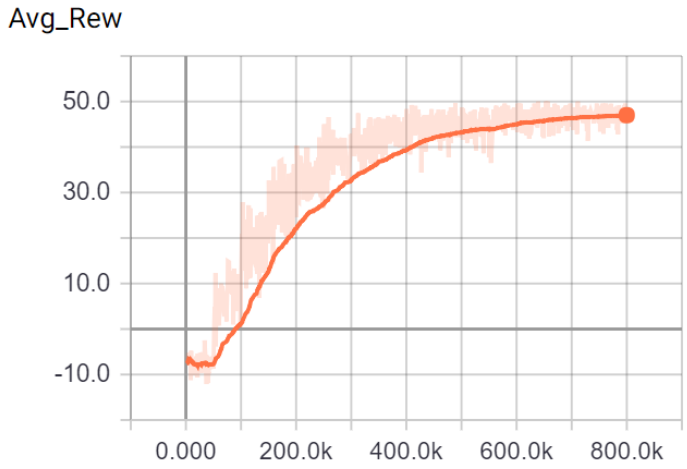
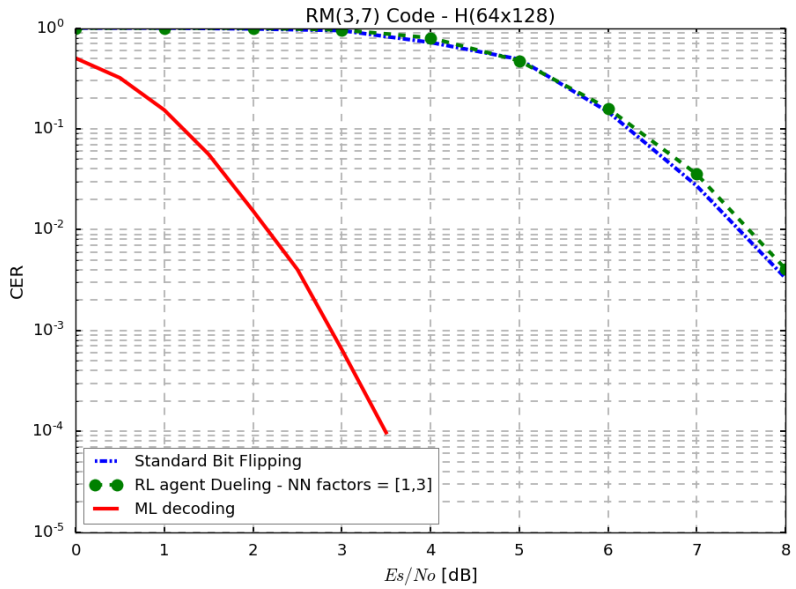


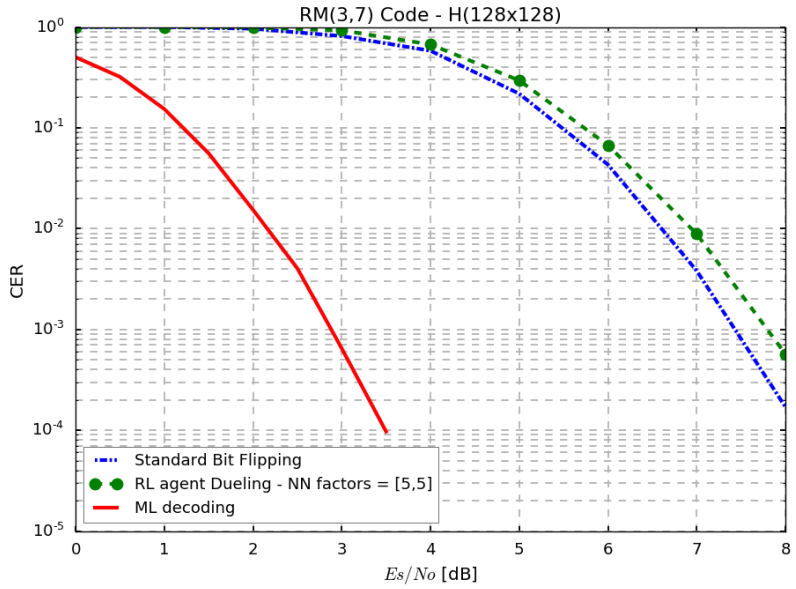
Figure 6.10: Average reward during training for RM(2,5), using $\boldsymbol{H}'$ of size 32×32. The number of training epochs is on the x-axis. Note that 50 is the maximum reward achievable by the agent, according to the reward strategy of Table 6.1.

overcomplete matrices in the BSC. The interesting fact is that our agent starts without previous knowledge about the code and the MDP. The code structure (i.e., the PC matrix $\boldsymbol{H}$) is intrinsically drawn in the MDP, since the state transitions and rewards are determined according to the syndromes. At first, the agent does not know the relation between the input (syndrome) and output (Q-value) of the NN, i.e., it does not know which PC equations involves a certain bit. The agent learns a policy by trial-and-error, following the Q-learning algorithm. In particular, the agent uses the Dueling NN architecture to approximate the Q-function. The Q-learning algorithm is guided by a discretized reward strategy, i.e., the MDP returns a high-reward only when a long term goal is achieved (e.g., a codeword is decoded).

Further work is needed to implement this approach over the BI-AWGN channel, which allows the agent to also exploit the soft information (bit reliability). The objective in the BI-AWGN channel is to match the standard Weighted BF (WBF) decoding algorithm, which combines the syndrome and the reliability of the received sequence to choose the



(a) $m' = 64$,, $u_a = 1$, $u_v = 3$.



(b) $m' = 128$, $u_a = 5$, $u_v = 5$.

Figure 6.11: CER vs E_s/N_0 for RM(2,5) code, using $m' = [64, 128]$ random MWPCs randomly chosen from the overcomplete matrix $\boldsymbol{H}_{\text{oc}}$, which has dimension 94488×128. The blue curve is related to standard BF performance and the red one is ML performance over the BSC.

bit to flip.

More investigation is also needed to improve the RL model in order to easily learn longer codes. The dimension of the input (syndrome), the output (Q-values) and hidden layers play a key role in the learning capabilities and may yield to unpractical models in terms of storage requirements and computation complexity.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529 EP –, Feb 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[2] W. E. Ryan and S. Lin, *Channel Codes Classical and Modern*. Cambridge University Press, 2009.

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

[4] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, August 2013.

[5] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354 EP –, October 2017, article. [Online]. Available: http://dx.doi.org/10.1038/nature24270

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. [Online]. Available: https://arxiv.org/abs/1312.5602

[7] A. G. Barto, "A history of reinforcement learning," July 19th, 2018, Talk at IJCAI2018. [Online]. Available: https://www.youtube.com/watch?v=ul6B2oFPNDM

[8] A. Ng, "The state of artificial intelligence," November 7th, 2017, Talk. [Online]. Available: https://www.youtube.com/watch?v=NKpuX_yzdYs&t=185s

[9] O. Simeone, "A very brief introduction to machine learning with applications to communication systems," 2018. [Online]. Available: http://arxiv.org/abs/1808.02342

[10] M. Lian, C. Häger, and H. D. Pfister, "What can machine learning teach us about communications?" in *Proc. Information Theory Workshop (ITW)*, Guangzhou, China, November 2018.

[11] E. Nachmani, Y. Be'ery, and D. Burshtein, "Learning to decode linear codes using deep learning," in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sept 2016, pp. 341–346. [Online]. Available: https://doi.org/10.1109/ALLERTON.2016.7852251

[12] M. Lian, F. Carpi, C. Häger, and H. D. Pfister, "Learned belief-propagation decoding with damping and redundant parity-check matrices," 2018, to be submitted for publication.

[13] A. B. Carlson and P. B. Crilly, *Communication Systems*. McGraw-Hill, 2001.

[14] R. G. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, January 1962.

[15] F. R. Kschischang, B. J. Frey, and H. A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[16] D. E. Muller, "Application of boolean algebra to switching circuit design and to error detection," *Trans. I.R.E. Prof. Group on Electronic Computers*, vol. 3, pp. 6–12, 1954.

[17] I. S. Reed, "A class of multiple-error-correcting codes and the decoding scheme," *IRE Professional Group on Information Theory*, vol. 4, no. 4, pp. 38–49, 1954.

[18] E. Santi, C. Häger, and H. D. Pfister, "Decoding reed-muller codes using minimum-weight parity checks," in *Proc. IEEE ISIT 2018*, Vail, Colorado, USA, Jun 2018. [Online]. Available: http://arxiv.org/abs/1804.10319

[19] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. North-Holland, 1977.

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

[21] D. Barber, *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

[22] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[23] A. Ng, "Machine learning." Coursera. [Online]. Available: https://www.coursera.org/learn/machine-learning

[24] "A neural network playground." [Online]. Available: https://playground.tensorflow.org/

[25] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436 EP –, May 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14539

[26] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, vol. 12, pp. 2493–2537, 2011.

[27] F. Wang, M. Jiang, C. Qian, S. Yang, C. Li, H. Zhang, X. Wang, and X. Tang, "Residual attention network for image classification," pp. 6450–6458, July 2017. [Online]. Available: http://doi.org/10.1109/CVPR.2017.683

[28] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[30] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," 2017. [Online]. Available: http://arxiv.org/abs/1706.02677

[31] T. Richardson and R. Urbanke, *Modern Coding Theory*. Cambridge University Press, 2008.

[32] Python Software Foundation, "Python language." [Online]. Available: https://www.python.org/

[33] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[34] "Grid search: tanh loss, cross entropy loss and cer versus damping and weight coefficients." [Online]. Available: https://plot.ly/~fabrizio.carpi/35.embed

[35] M. P. C. Fossorier and S. Lin, "Soft-input soft-output decoding of linear block codes based on ordered statistics," in *IEEE GLOBECOM 1998 (Cat. NO. 98CH36250)*, vol. 5, Nov 1998, pp. 2828–2833 vol.5.

[36] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[37] P. Abbeel and D. Klein, "Cs 188." [Online]. Available: http://ai.berkeley.edu/

[38] R. G. Gallager, "6.262 discrete stochastic processes," Spring 2011, MIT OpenCourseWare. [Online]. Available: https://ocw.mit.edu

[39] A. G. B. Richard S. Sutton, *Reinforcement Learning: An Introduction*. MIT press, 1998.

[40] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, UK, May 1989.

[41] C. J. C. H. Watkins and P. Dayan, "Technical note: Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992. [Online]. Available: https://doi.org/10.1023/A:1022676722315

[42] "Lecture 11: Reinforcement learning ii," U.C. Berkeley - CS188. [Online]. Available: https://www.youtube.com/watch?v=yNeSFbE1jdY

[43] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus, "Learning simple algorithms from examples," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 421–429. [Online]. Available: http://proceedings.mlr.press/v48/zaremba16.html

[44] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1995–2003. [Online]. Available: http://proceedings.mlr.press/v48/wangf16.html

[45] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[46] F. Pardo, V. Levdik, and P. Kormushev, "Goal-oriented trajectories for efficient exploration," July 2018. [Online]. Available: https://arxiv.org/abs/1807.02078

Acknowledgments

"We must find time to stop and thank
the people who make a difference in our lives."
John F. Kennedy

I would like to thank my advisor Professor Riccardo Raheli, who gave me the possibility to do this thesis abroad. It was a great experience for my personal and academic development. Thanks also to University of Parma, which made this possible granting me a scholarship through the "Overworld" program. Moreover, thanks to Professor Raheli and Dr. Marco Martalò for their advice and support during my period abroad and once back in Italy.

I am profoundly grateful to Professor Henry D. Pfister, who hosted me for 6 months at Duke University, North Carolina. Thanks for welcoming me in the group and guiding my research path during this thesis. It was not straightforward, but his stimuli and enthusiasm inspired my work and my future perspectives.

My sincere thanks to Dr. Christian Häger, who co-advised my thesis at Duke. Thanks for his precious support during my first work in the machine learning field.

I would also like to thank my colleagues at Duke, Mengke Lian and Narayanan Rengaswamy. Thanks for the useful academic conversation and for helping me during my visit to the US.

I want to thank my family for the unfailing support, encouragement and love received throughout my years of study. Thanks for teaching me that the sacrifice is paid. Thanks for always being by my side. Thanks for taking me this far.

Last but not least, I would like to thank my friends. Thanks for the funny moments, the ups and the downs we shared. Thanks for making this a great journey.

Fabrizio

Parma, October 12, 2018