Stochastic Models for Sparse Codes

Maria Urze Osório

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Andreas Miroslaus Wichert

Examination Committee

Chairperson: Prof. Pedro Tiago Gonçalves Monteiro
Supervisor: Prof. Andreas Miroslaus Wichert
Member of the Committee: Prof. Rui Miguel Carrasqueiro Henriques

February 2022

Acknowledgments

My deep gratitude goes to Professor Andreas Wichert and Luis Sá Couto for the insight, support and sharing of knowledge that has made this thesis possible. I would also like to thank my parents Paula and Luís for their unconditional support over all these years, for always being there for me through thick and thin and without whom this thesis would not be possible. My grandmother, aunts, brothers and sisters-in-law for their understanding and support throughout all these years. My boyfriend Miguel, for the unconditional support through doubt and uncertainty. Last but not least, all my friends, specially Carlota and Maria, that helped me grow as a person and were always there for me during the good and bad times in my life. To each and every one of you – Thank you.

Abstract

There is a consensus about information in the brain being represented by using Sparse Distributed Representations. These representations, however, are high-dimensional input vectors and consequently they affect classification performance due to the notoriously complex problem known as “the curse of dimensionality”. In tasks for which there is a vast amount of labeled data, Deep Learning seems to solve this issue with many layers and a non-biologically plausible Backpropagation algorithm. The purpose of this research is to find a way to learn from high-dimensional sparse data side stepping these limitations and adopting a more biologically plausible approach. Actually, hidden units in Stochastic Models, represent hidden correlations between present dimensions of sparse vectors. These models can map a high-dimensional sparse vector into a hidden layer with few hidden units, while capturing the relevant features. Motivated by these reasons, we implement several classifiers inspired by Stochastic Models. In order to test them on a high-dimensional sparse data, we start by using the sparse codes generation mechanism structured in [1]. The implemented Stochastic Models are tested on these codes and their performance is compared with a simple Logistic Regression. Both the Stochastic Models and the Logistic Regression achieve good results. However, these good results archived by the Logistic Regression classifier led us to believe that the generated codes lie on a low-dimensional manifold embedded in a higher-dimensional space, which suggests that the real dimensionality of the data is highly inferior to the number of features. Afterwards, we propose a different way to generate sparse data, where each class follows a multivariate normal distribution and the sparseness is controlled by randomly deleting values in each sample. The experiments using this data confirm our initial intuition as the Restricted Boltzmann Machine shows a good generalization performance, while the Logistic Regression overfits the training

data.

Keywords

Stochastic Models; Restricted Boltzmann Machines; Sparse Distributed Representations; Learning.

Resumo

Há um consenso em torno da ideia de que a informação no cérebro pode ser reproduzida através de representações distribuídas e esparsas. No entanto, estas representações constituem vetores de alta dimensionalidade e, consequentemente, afetam o desempenho da classificação. Em tarefas para as quais há uma grande quantidade de dados, os modelos de aprendizagem profunda resolvem esse problema com diversas camadas e um algoritmo não biologicamente plausível. O objetivo da presente investigação é, justamente, procurar encontrar uma forma de classificar conjuntos de dados esparsos e de alta dimensionalidade, superando as limitações anteriormente referidas e seguindo uma abordagem biologicamente aceitável. Na verdade, os neurónios ocultos nos modelos estocásticos representam correlações entre dimensões presentes de vetores esparsos. Esses modelos conseguem mapear vetores esparsos e de alta dimensionalidade, numa camada oculta com poucos neurónios. Tendo por base esta motivação, implementamos vários classificadores inspirados nos modelos estocásticos, no propósito de os testar em dados esparsos e de alta dimensionalidade, começando por usar o mecanismo de geração de dados esparsos estruturado no artigo [1]. Os modelos estocásticos implementados são testados nestes códigos e seu desempenho é comparado com uma simples regressão logística. Tanto os modelos estocásticos como a regressão logística apresentam boa performance. No entanto, os bons resultados alcançados pela regressão logística leva-nos a acreditar que a dimensionalidade real dos dados gerados é bastante inferior ao número de características. Em seguida, propomos uma nova abordagem na geração de dados esparsos, em que cada classe segue uma distribuição normal multivariada e a esparsidade é controlada pela exclusão aleatória de valores em cada amostra. As experiências com esses dados confirmam a intuição inicial, na medida em que a Máquina de Boltzmann Restrita demonstra um bom desempenho de generalização, enquanto a regressão logística se adapta

demasiado aos dados de treino.

Palavras Chave

Modelos Estocásticos; Máquinas de Boltzmann Restritas; Representações Distribuídas e Esparsas; Aprendizagem.

Contents

1 Introduction **1**

1.1 Problem . . . 3

1.2 Motivation . . . 4

1.3 Thesis outline . . . 6

2 Background **9**

2.1 Sparse Distributed Representations . . . 11

2.2 Associative Memories . . . 12

2.3 Markov Chain . . . 12

2.4 Markov Chain Monte Carlo . . . 13

2.4.1 Metropolis Algorithm . . . 13

2.4.2 Gibbs sampling . . . 15

2.5 Simulated Annealing . . . 15

2.6 Evaluation Measures . . . 16

2.6.1 Pseudo-likelihood . . . 16

2.6.2 Mean Squared Error . . . 17

2.6.3 Accuracy . . . 17

3 Stochastic Models **19**

3.1 Hopfield Network . . . 21

3.1.1 Energy function . . . 23

3.2 Ising Model . . . 23

3.2.1 Spin glass . . . 24

3.2.2 Finite temperature dynamics . . . 24

3.2.3 Boltzmann-Gibbs distribution . . . 25

3.2.4 Stochastic dynamics . . . 26

3.2.5 How an Ising Model Generates Data . . . 27

3.3 Boltzmann Machine . . . 27

3.3.1 How a Boltzmann Machine Generates Data . . . 29

3.3.2 Learning 29
3.4 Restricted Boltzmann Machine 30
3.4.1 Contrastive divergence 33
3.4.2 Persistent Contrastive divergence 34
3.4.3 Weight-decay 34
3.4.4 Momentum 35
3.4.5 Different types of units 35
3.4.5.A Softmax visible units 35
3.4.5.B Gaussian visible units 36
3.4.6 Restricted Boltzmann Machine for classification 36
3.4.7 Deal with missing data with RBMs 37
3.4.7.A The model 38
3.4.7.B Learning 39
3.5 Deep Belief Networks 39
4 Experiments **43**
4.1 Datasets description 45
4.1.1 MNIST 45
4.1.2 Sparse MNIST generation 46
4.2 Stochastic Models for image reconstruction 47
4.3 Classification using Stochastic Models 50
4.3.1 Restricted Boltzmann Machine 50
4.3.2 Deep Belief Network 55
4.3.3 Restricted Boltzmann Machine followed by Logistic Regression 57
4.3.4 Deep Belief Network followed by Logistic Regression 59
4.3.5 Restricted Boltzmann Machine with 3 state neurons 60
4.4 Models comparison 63
4.5 Learn from a Sparse Normal Distributed Dataset 64
4.5.1 Dataset generation 65
4.5.2 Pipeline 65
4.5.3 Experimental Analysis 66
5 Conclusion **71**
Bibliography **75**

List of Figures

1.1 Ten image sample of MNIST test set. The ten top images represents a binarized version of an original MNIST sample, while the bottom images represent the same sample flipping the bits. 5

3.1 MNIST handwritten digits storage and retrieval using Hopfield Model. 22
3.2 The energy surface where the valleys correspond to the attractors of the system. 23
3.3 Comparison between the architectures of the Hopfield Model, the Boltzmann Machine (BM) and the Restricted Boltzmann Machine (RBM) 31
3.4 Network graph of an RBM with n hidden units and m visible units 31
3.5 Contrastive Divergence (CD) with single-step reconstruction 34
3.6 RBM that models the joint probability distribution of input images and the corresponding labels. 37
3.7 Adapted from [2], a RBM with binary hidden units and softmax visible units is represented. For each dataset sample (user), the RBM only includes softmax units for the movies that user has rated. In addition, to the symmetric weights between each hidden unit and each of the K = 5 values of a softmax unit, there are 5 biases for each softmax unit and one for each hidden unit. 38
3.8 Deep Belief Network 40

4.1 Ten image sample of MNIST training set. 45
4.2 Adapted from [1], we have the overview of the strategy that transforms visual patterns of digits into sparse and distributed codes. The first step is local feature extraction (Retinotopic Step). Each feature is depicted as a window with an oriented line and each image containing a number is parsed with a sliding window, finally each occurrence of each feature is signaled at the middle layer. The Object-Dependent Step maps these positions to an object-dependent, radius one polar coordinate system. 46
4.3 Training patterns 47

4.4 Corrupted training patterns 47
4.5 Energy evolution during pattern reconstruction with stochastic update implementation of the Hopfield Network. 48
4.6 Mean pseudo-likelihood during training of RBM models with different learning rates 49
4.7 Pseudo-likelihood during training of RBM models with different number of hidden units . . 49
4.8 Ten image sample of MNIST test set. Corrupted images (top) are given to the network in order to perform Gibbs sampling and then get the reconstructed images (bottom). 50
4.9 Train and test accuracies of RBM on the original MNIST given an increasing number of hidden units. 52
4.10 Train and test accuracy varying numbers of neurons per class, namely N=1, N=2 and N=5 53
4.11 Train and test accuracy of Logistic Regression (LR) and RBM given sparse MNIST datasets with increasing sparseness 54
4.12 Deep Belief Network (DBN) with 2 layers that models the joint probability distribution of hidden activations given the input images and the corresponding labels. 55
4.13 Train and test accuracies of sparse MNIST datasets given the increasing number of hidden units in the last layer of the DBN. 56
4.14 Scheme of baseline models. 57
4.15 Scheme of the classifier composed by a RBM followed by a LR 58
4.16 Train and test accuracies of original MNIST and sparse MNIST datasets given the increasing number of hidden units. 59
4.17 Scheme of the classifier composed by a DBN followed by a LR. 60
4.18 Performance of the LR in a dense versus a high-dimensional sparse dataset. 67
4.19 Comparison between accuracies of LR and RBM classifiers with increasing percentage of sparseness. 68
4.20 The left heatmap shows the difference between train accuracies of RBM and LR considering increasing dimensionality and sparseness. The right heatmap shows the difference between test accuracies of RBM and LR considering increasing dimensionality and sparseness percentage. 69

List of Tables

1.1 Train and test mean accuracies of RBM when classifying a sample of the original binarized MNIST and a flipped version of the same sample. 5

1.2 Train and test mean accuracies of LR when classifying a sample of the original binarized MNIST and a flipped version of the same sample. 6

4.1 Train and test accuracy of RBM and LR given two different generated sparse datasets . . 54

4.2 Train and test accuracy of DBN with 2 layers, DBN with 3 layers and LR given the Sparse datasets 2. 57

4.3 Train and test accuracy of LR and RBM followed by LR models considering the same sparse dataset (Sparse dataset 2) 59

4.4 Train and test accuracy of LR and DBN followed by LR models considering the same sparse dataset (Sparse dataset 2) 60

4.5 Train and test accuracy comparison between the original RBM implementation (2 state neurons) and the 3 state neurons implementation given the same sparse dataset 63

4.6 Train and test accuracy of all the models implemented considering the same sparse dataset (Sparse dataset 2) 64

4.7 Train and test mean accuracies of Multi-Layer Perceptron (MLP) and RBM given the generated data. 70

Acronyms

BM	Boltzmann Machine
CD	Contrastive Divergence
DBN	Deep Belief Network
DBM	Deep Boltzmann Machine
LR	Logistic Regression
MCMC	Markov Chain Monte Carlo
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
PCD	Persistent Contrastive divergence
RBM	Restricted Boltzmann Machine
SDR	Sparse Distributed Representation

1

Introduction

Contents

1.1 Problem 3
1.2 Motivation 4
1.3 Thesis outline 6

Traditional computer data structures cannot represent efficiently all concepts, the relationships between them, and the exceptions that each concept definition may hold.

The human brain does not have this problem: in order to represent information, it shares neurons between concepts, which means that a single neuron can be part of the representation of many different concepts. Furthermore, empirical evidence demonstrates that every region of the neocortex represents information by using sparse activity patterns [3]. When looking at any population of neurons in the neocortex their activity will be sparse, whenever a low percentage of neurons are highly active and the remaining neurons are inactive.

Sparse Distributed Representation (SDR) is the method used to implement computationally the way information is represented in the brain [4]. An SDR is a binary vector composed of a large number of bits where each bit represents a neuron in the neocortex.

Consider that one wants to recognize a particular activity pattern in a neuron. Then, one says that a neuron forms synapses to the active cells in that pattern of activity. This way, a neuron only needs to form a small number of synapses, to accurately recognize a sparse pattern in many cells. The formation of new synapses is the pillar of all memory in the brain [5].

Memory in the brain is called Associative Memory, in which different SDRs (input patterns) become associated with one another depending on the similarity between them [6]. This means, SDRs are recalled through “association” with other SDRs. In Associative Memories, there is no centralized memory and no random access [7]. Every neuron present in the brain is an integral part of each SDR, which means each neuron participates in forming both SDRs and in learning the associations between them [5].

1.1 Problem

As discussed previously, SDRs are binary vectors composed of many bits [8], which means we are dealing with a high dimensional input. These sparse representations are known to work well with associative memories but when we try to classify them, we must deal with some problems.

The main problem that brought us here is known as “the curse of dimensionality” caused by the high-dimensionality of SDRs. Classic Machine Learning models, for example, Feed-Forward Networks, are not good at dealing with high-dimensional sparse inputs given the vast number of parameters. [9], [10].

Many machine learning problems become exceedingly difficult when the number of dimensions in the data is high. As discussed in [11] the number of possible distinct configurations of a set of variables increases exponentially with the number of variables.

There is no universal answer for how data sparsity would affect learning convergence behaviour in Machine Learning models [12]. Though, [10] discusses that most features in high-dimensional vectors

are usually non-informative or noisy and may decrease the model's generalization performance.

In a Neural Network, the input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes. Therefore, considering an SDR as an input (high dimensionality vector), the number of weights from the input layer to the first hidden layer will be the number of entries of the binary input vector multiplied by the number of hidden units in the first hidden layer [11]. This way, the model will have a really large number of parameters making it prone to overfitting.

The phenomenon of overfitting occurs when a network fits the training data more than it should. When overfit happens, it captures noise from the training data. This leads to a model with weak generalization capability, increasing the error when classifying new instances from the test set [13]. With the intent of reducing overfitting, the necessity of generating large amounts of labelled data arises, which is an expensive task.

1.2 Motivation

As argued in [9] and discussed in the former section, a well-known problem in Machine Learning is sparse data, which alters the performance of Machine Learning algorithms and their ability to calculate accurate predictions. A high-dimensional sparse input leads to the well-known problem denominated "the curse of dimensionality".

The purpose of this thesis is to show that it is possible to learn good and general classifiers from high-dimensional sparse representations generated by biologically plausible models while, unlike the deep learning approach, staying under the biological constraints.

With this idea in mind, we chose to investigate Stochastic Models, i.e., Restricted Boltzmann Machine (RBM) and Deep Belief Network (DBN), to classify high-dimensional sparse data motivated on the following arguments:

- As proved in [1], Willshaw's model of associative memory has been showed to work well with high-dimensional sparse codes. Therefore, given the inspiration of Stochastic Models on Associative Memories, they seem to be a great candidate to perform well with those codes.
- Boltzmann Machines have local learning rules (Hebbian rule), which are biologically plausible [14].
- Hidden units in Stochastic Models represent hidden correlations between present dimensions of sparse vectors. As these models only learn the correlation between active units, then stochastic models can map a high-dimensional sparse vector into a hidden layer with few hidden units, while capturing the relevant features.

The later argument represents the core motivation of using Stochastic Models to classify high-dimensional sparse data. These models learn the correlations between active neurons, which means the hidden units will exclusively change their state based on the input units that are different from zero. This allows Stochastic Models to have a compact hidden layer that captures the information present on the high-dimensional sparse data without falling into overfitting.

To deeply ground this motivation a trivial experiment was performed in which, a RBM was used to classify a sample of the original binarized MNIST[1] dataset and a flipped version of that same sample. In Figure 1.1, the ten top images represent the binarized version of the original MNIST, in which the bits representing each digit are set to 1 and the background information to 0. The ten bottom images represent a flipped sample of the original binarized MNIST, where the bits representing each digit are set to 0 and the background information to 1.

Figure 1.1: Ten image sample of MNIST test set. The ten top images represents a binarized version of an original MNIST sample, while the bottom images represent the same sample flipping the bits.

We considered a RBM with the same architecture to classify both versions of the binarized MNIST, the original and the flipped one. If our intuition points in the right direction, the model should be able to accurately classify the original version in which the digits information are represented by 1s and fail on the flipped version in which the digits are represented by 0s.

With a sample of 5000 training examples and 1000 test examples of both datasets, Table 1.1 shows the results achieved by the RBM classifier.

	Train accuracy	Test accuracy
Original version	91.6%	86.2%
Flipped version	15.5%	13.8%

Table 1.1: Train and test mean accuracies of RBM when classifying a sample of the original binarized MNIST and a flipped version of the same sample.

By analysing the results achieved by a RBM with 500 hidden units, one can conclude that in the original version of the binarized MNIST, the model learns the correlations between active neurons, which

[1] http://yann.lecun.com/exdb/mnist/

represent the digits. As the active bits represent a relatively small percentage of each sample, the model is able to capture the correlations between these active features and have a good generalization performance.

In the flipped version of the binarized MNIST, the RBM fails completely. This is justified by the fact that this model exclusively learns correlations between present dimensions and not between 0s. With a hidden layer of 500 units, the RBM is unable to catch the correlations between all the active neurons that represent the background, and consequently fails when classifying the flipped version of the MNIST.

This small experiment validates the strong potential of Stochastic Models to deal with high-dimensional sparse inputs, as they can capture the correlation between present dimensions of the input data.

By performing the same experiment with the Logistic Regression (LR) classifier, and analysing the results on Table 1.2, one concludes that this model has similar performances when classifying the original and the flipped version of the MNIST sample.

	Train accuracy	Test accuracy
Original version	99.6%	85.2%
Flipped version	99.7%	84.5%

Table 1.2: Train and test mean accuracies of LR when classifying a sample of the original binarized MNIST and a flipped version of the same sample.

The similar accuracy results achieved by LR in both problems suggests that this model learns the information given by 1s in the same way it does with 0s. This implies that, when LR is dealing with high-dimensional sparse inputs, it learns all the dimensions of the sparse vector. Consequently, this model is prone to fall into overfitting.

On the other hand, the RBM classifier learns exclusively the correlations between active neurons. Given that high-dimensional sparse inputs have a low percentage of 1s, this model can map the high-dimensional sparse vector into a hidden layer with few hidden units, while capturing the relevant features.

But does this really indicates that the RBM avoids the overfitting problem when classifying high-dimensional sparse data? The purpose of this research work is to answer this question by investigating the potentiality of the Stochastic Models to deal with high-dimensional sparse data.

1.3 Thesis outline

The thesis is structured into the following main sections:

1. Deepening the concepts of Sparse Distributed Representations and Associative Memories (section 2.1 and 2.2).

2. Describing algorithms and techniques which will be fundamental to understand the Stochastic Methods (section 2.3, 2.4 and 2.5).

3. Describing the evaluation measures needed to ensure a trustful solution (section 2.6).

4. Studying different models from the Stochastic methods family, which will be the foundation of the proposed solution (chapter 3).

5. Describe the implemented Stochastic Models used to perform the experiments and perform a detailed analysis of those experiments (chapter 4).

6. Taking the final conclusions and remarks regarding the research performed (chapter 5).

2

Background

Contents

2.1 Sparse Distributed Representations 11
2.2 Associative Memories 12
2.3 Markov Chain 12
2.4 Markov Chain Monte Carlo 13
2.5 Simulated Annealing 15
2.6 Evaluation Measures 16

This section starts by explaining in detail the concepts of SDRs (section 2.1) and Associative Memory (section 2.2), already addressed in the introduction. These two concepts are the basis for understanding the inspiration of using stochastic models in this research process.

Then, we focused on the introduction of the following concepts: Markov Chain, Markov Chain Monte Carlo, Gibbs sampling and Simulated Annealing, which are fundamental to understand the Stochastic Models described in chapter 3.

Finally, in section 2.6, we describe the measures used to evaluate the implemented Stochastic Models, which are described in chapter 4.

2.1 Sparse Distributed Representations

Neuroscience has shown that information in the brain is represented by the sparse activation of clusters of neurons in the neocortex [15]. With this idea in mind, we can state that SDR are biologically plausible informative representations.

SDR is the method used to implement computationally the way information is represented in the brain. An SDR is a binary vector composed of a large number of bits where each bit represents a neuron in the neocortex.

As previously addressed, an SDR is a binary vector composed of a large number of bits where each bit represents a neuron in the neocortex. As the brain activates just a few neurons at a time, in the SDR, only a small percentage of bits are 1's (active neurons), typically less than 2%, and the rest are 0's (inactive neurons) [5].

Each bit of a SDR has a meaning associated. The set of active bits in the SDR encodes the set of semantic attributes of what is being represented. Therefore, if we compare two SDRs and they have active bits in the same index, we can conclude that those SDRs share the semantic attributes represented by the common active bits.

In SDRs information is carried in the representation itself and not stored externally, which makes representations of SDRs informative. Since Associative Memories (content-addressable memories) handle inputs as content/information, then they benefit from this SDR property.

In an SDR, each neuron represents a property of a certain concept and then one can use the same group of neurons, with different activation patterns to represent lots of different concepts. The active neurons change over time, which means that the same set of neurons in a certain moment can represent one thing and, in the next moment, represent another. Any two different Sparse Distributed Representations that follow each other in time can associatively be linked, and this sequence of SDRs can be learned [16].

2.2 Associative Memories

As discussed in [17] an association process starts when we try to find a specific piece of information in our memory, and we are not capable of retrieving it immediately. This way, our brain starts a sequential association process from one item to the next until it reaches the missing information. Once we retrieved that piece of information, we immediately can recognize it since it fits perfectly into the context that triggered our mental search. Therefore, we can conclude that our brain associates a new output to a given input depending on contextual information [18].

The associative memory is composed of a cluster of units which represent a simplified model of real neurons. It is based on associations with the memories it has stored. This type of memory is designated content-addressable (CAM), which means a particular part of the memory is connected with the rest. We can describe two different mechanisms that are core in the process of association: hetero-association, which is the process of associating one pattern to the next, and auto-association described as the association of a pattern to itself [7].

It is important to underline that Associative Memories are quite different from traditional artificial memories, where an address is provided and its content is obtained back. Actually, Associative Memories do not employ addresses explicitly. A question content vector is provided and an answer content vector is returned [19]. The goal of associative memories is to store a finite set of S of P associations or pairs, (x, y), where x and y are the question vector and the answer vector respectively

$$S \doteq (x^{\mu} \rightarrow y^{\mu}) : \mu = 1, \ldots, P. \tag{2.1}$$

The Associative Memory establishes a mapping $(x^{\mu} \rightarrow y^{\mu})$ which is denoted hetero-association process. In the special case where $x = y$ the memory is said to perform the auto-association process. The combination of hetero-association and auto-association capabilities of Associative Memories allows to naturally implement functions of biological memories [20].

2.3 Markov Chain

A Markov chain is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state of the previous event [21].

Considering a sequence of random variables $x_1, x_2, \ldots, x_n$, we can affirm that x_{n+1} forms a Markov chain, if the probability that the system is in state x_{n+1}, given the sequence of past states it has gone through, is exclusively determined by the state x_n. Therefore, transition probabilities between states are represented by conditional probabilities represented by

$$p_{ij} = P(X_{n+1} = j | X_n = i). \tag{2.2}$$

As part of the definition of a Markov chain, there is some probability distribution on the states at time 0. Each time step the distribution on states evolves, which means, some states may become more likely then others and this is dictated by the transition matrix P. After a sufficiently long time the Markov chain will eventually converge to its stationary distribution.

The stationary distribution of a Markov Chain with transition matrix P is some vector, π, such that $\pi P = \pi$.

Markov processes are the basis for general stochastic simulation methods known as Markov chain Monte Carlo, which are used for simulating sampling from complex probability distributions.

2.4 Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) techniques are methods for sampling from probability distributions using Markov chains.

An MCMC method for the simulation of a probability distribution π is any method that produces an ergodic Markov chain (aperiodic, positive recurrent and irreducible) whose own stationary distribution is the distribution π. The more steps that are included, the more closely the distribution of the sample matches the actual desired distribution π.

Following this brief introduction we will explain two different MCMC methods: the Metropolis algorithm and the Gibbs Sampling. These two methods are crucial to understand how data is generated in the Stochastic models described in chapter 3.

2.4.1 Metropolis Algorithm

The Metropolis algorithm allows us to simulate the evolution of a physical system to thermal equilibrium [22].

It was introduced in the early days of scientific computation for the stochastic simulation of a collection of atoms in equilibrium at a given temperature. Metropolis proposed a stochastic matrix that is composed of a set of transition probabilities denoted by T_{ij}, which satisfy the following conditions:

$$T_{ij} \geq 0, \; T_{ij} = T_{ji}, \; \sum_j T_{ij} = 1. \tag{2.3}$$

Let π_i denote the steady-state probability that the Markov chain is in state x_i, $i = 1, 2, ..., K$. Steady-state probabilities are average, constant probabilities that the system will be in a certain state after a

large number of transition periods. We may then use T_{ij} and the probability distribution ratio π_i/π_j, to formulate the set of transition probabilities [23] as described below:

$$P_{ij} = \begin{cases} T_{ij} & \frac{\pi_i}{\pi_j} \geq 1 \\ T_{ij} \cdot (\frac{\pi_i}{\pi_j}) & \frac{\pi_i}{\pi_j} < 1 \end{cases} \tag{2.4}$$

The only outstanding requirement is determining how to choose the ratio π_i/π_j. In order to fulfill this requirement, as this is a technique which will be applied in our stochastic models (described in chapter 3) and following [23], the probability distribution to which we want the Markov chain to converge to is a Boltzmann-Gibbs distribution

$$\pi_i = \frac{1}{Z} e^{(-\beta \cdot H_i)}, \tag{2.5}$$

with

$$\beta = \frac{1}{k \cdot T}. \tag{2.6}$$

The Boltzmann-Gibbs distribution gives the probability that a system will be in a certain state as a function of that state's energy H_i and a constant $k \cdot T$ of the distribution is the product of Boltzmann's constant k and thermodynamic temperature T. The distribution shows that states with lower energy will always have a higher probability of being occupied than the states with higher energy.

The probability distribution ratio π_i/π_j is represented by

$$\frac{\pi_i}{\pi_j} = e^{(-\beta \cdot \Delta H)}, \tag{2.7}$$

with

$$\Delta H = H_i - H_j. \tag{2.8}$$

Using Equation (2.4) we get

$$P_{ij} = \begin{cases} T_{ij} & e^{(-\beta \cdot \Delta H)} \geq 1 \\ T_{ij} \cdot e^{(-\beta \cdot \Delta H)} & e^{(-\beta \cdot \Delta H)} < 1 \end{cases} \tag{2.9}$$

A sufficient condition to ensure that the system is in thermal equilibrium is the principle of Detailed Balance which states that the rate of occurrence of any transition equals the corresponding rate of occurrence of the inverse transition [23], and this is shown by :

$$\pi_i \cdot P_{ij} = \pi_j \cdot P_{ji}. \tag{2.10}$$

The Metropolis algorithm accepts a transition if the new configuration has a lower energy. Otherwise, the algorithm accepts the change that increases the energy but only with probability $e^{(-\beta \cdot \Delta H)}$. To verify this condition, a random value $\xi \in [0, 1]$ is generated and if $\xi < e^{(-\beta \cdot \Delta H)}$ then the transition is accepted. The acceptance depends on the temperature. With high temperature the probability of acceptance is high, by lowering the temperature the probability of acceptance diminishes [24].

2.4.2 Gibbs sampling

Gibbs sampling that was introduced in the context of image processing by [25] is a simple MCMC algorithm and a specialization of the previously discussed Metropolis algorithm [22] for producing samples from the joint probability distribution of multiple random variables [21]. It is a special case of Metropolis algorithm in which the newly proposed state is always accepted with probability one [26].

Suppose we want to sample from a distribution $p(x) = p(x_1, x_2, \ldots, x_n)$ and that we have chosen some initial state for the Markov chain. Then, each step of the Gibbs sampling algorithm consists of replacing the value of one of the variables by a new value drawn from the distribution of that variable conditioned on the values of the remaining variables. That is, generating a value for the conditional distribution of each component of the random vector x, given the values of all other components of x [24].

Gibbs Algorithm:

1. Initialize

$$\{x_i : i = 1, \ldots, M\}.$$

2. For $t = 1, \ldots, T$:

   -Sample $x_1^{(t+1)}$ and replace $x_1^{(t)}$ by $x_1^{(t+1)} \backsim p(x_1|x_2^{(t)}, x_3^{(t)}, \ldots, x_M^{(t)})$.

   -Sample $x_2^{(t+1)}$ and replace $x_2^{(t)}$ by $x_2^{(t+1)} \backsim p(x_2|x_1^{(t+1)}, x_3^{(t)}, \ldots, x_M^{(t)})$.

   - ...

   -Sample $x_M^{(t+1)}$ and replace $x_M^{(t)}$ by $x_M^{(t+1)} \backsim p(x_M|x_1^{(t+1)}, x_2^{(t+1)}, \ldots, x_{M-1}^{(t-1)})$.

2.5 Simulated Annealing

At very low temperatures the convergence rate of the Markov chain of the Gibbs sampling or Metropolis algorithm to thermal equilibrium is extremely slow. Thus, to improve computational efficiency, it is necessary to operate the stochastic system at high temperature where convergence to equilibrium is fast and then maintain the system in an equilibrium state as the temperature is slowly lowered [23].

Simulated annealing can be enhanced by the following characteristics:

- The algorithm does not need to get stuck in local minima because the transition out of a local minimum is possible when the system operates at a non-zero temperature.
- The global features of the final state of the system are seen at higher temperatures, while the fine details of the state appear at lower temperatures.

The probability of making a transition from the current state s to a candidate new state s_{new} is specified by an acceptance probability function, that depends on the energies of the two states, and on a global time-varying parameter T called the temperature. States with a smaller energy are better than those with a greater energy. The probability function must be positive even when the energy of s_{new} is greater than the energy of s. This feature allow us to explore the space, even the areas with lower probability (high energy) which prevents the method from becoming stuck at a local minimum that is worse than the global one, as we want the stochastic system to converge to the global minimum.

We start with a high temperature T and during the simulated annealing process T is slowly decreased [27]. The initial value T_0 of the temperature must be chosen to be sufficiently high in order to ensure that all possible transitions are accepted by the simulated annealing algorithm with a certain probability.

Actually, one can define two distinct phases on decrementing the temperature in simulated annealing, i.e., a fast decrease at the very high temperature T_0 to a certain temperature in the first phase followed by a very slow decrease in the second phase. At the final temperature T_F, the system is fixed, and the annealing process stops.

When the system is operating in a temperature $T > 0$ the transition out of a local minimum is possible, therefore, the algorithm does not need to get stuck in local minima. For sufficiently small values of T, the system will then increasingly favor moves that go "downhill" (i.e., to lower energy values), and avoid those that go "uphill".

Finally, with $T = 0$ the procedure reduces to the greedy algorithm, which makes only the downhill transitions [28]. In this process, the global features of the final state of the system are seen at higher temperatures, while the fine details of the state appear when the temperature is decreased [24].

2.6 Evaluation Measures

2.6.1 Pseudo-likelihood

The Pseudo-likelihood is introduced as a measure that helps tracing the quality of the training phase of the stochastic models. To simplify the process instead of using the original measure we used an approximation of the pseudo-likelihood. In order to obtain this approximation, one starts by computing a

quantity called the free energy, which is given by the following equation:

$$F(v) = -\ln\left(\sum_h e^{-E(v,h)}\right). \tag{2.11}$$

The free energy is computed on the original data that the model receives and on a randomly corrupted version of it. Then, the difference between these free energies is calculated and the approximation of the pseudo-likelihood is computed by the log of the logistic function of the difference between both free energies [29]. This measure is a proxy that indicates how likely the data is. Thus, during the training process the training data becomes more likely and the pseudo-likelihood of the training data is expected to increase .

2.6.2 Mean Squared Error

The Mean Squared Error (MSE) is a model evaluation metric. The mean squared error of a model with respect to a train or test set is, respectively, the mean of the squared prediction errors over all instances in the train or test set [11].

The prediction error, in the context of the Stochastic Models, is calculated with the difference between the original pattern and the reconstruction pattern pixels.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2. \tag{2.12}$$

2.6.3 Accuracy

To correctly evaluate our models and fully understand its capabilities, it is crucial to define appropriate performance measures. Most of the measures used in machine learning problems can be defined as a combination of the following values:

1. True Positives (TP): Total number of instances that belong to class c and that are classified as c.
2. True Negatives (TN): Total number of instances that do not belong to class c and that are not classified as c.
3. False Positives (FP): Total number of instances that do not belong to class c and that are classified as c.
4. False Negatives (FN): Total number of instances that belong to class c and that are not classified as c.

Accuracy is the most widely used metric, and denotes the percentage of correct predictions made. As we have a Multi-class problem the accuracy of each class c can be defined as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \quad (2.13)$$

The accuracy of each class can be important to evaluate the quality of each class predictions. Though, the crucial evaluation measure is the total accuracy of all classes, which consists of all the instances that the model classified correctly divided by the total of instances that the model classified [11].

3

Stochastic Models

Contents

3.1 Hopfield Network 21
3.2 Ising Model 23
3.3 Boltzmann Machine 27
3.4 Restricted Boltzmann Machine 30
3.5 Deep Belief Networks 39

With the intent to propose a robust model that performs accurately when given a high-dimensional sparse input, we describe a set of models which belong to the family of Stochastic methods and show a great potential to solve the problem presented before (section 1.1).

The Hopfield Network and the Ising Model described in section 3.1 and 3.2 will not be used in the proposed solution, however they comprise the basic concepts to fully understand the more complex models of Boltzmann Machine (BM) (section 3.3), RBM (section 3.4) and DBN (section 3.5).

3.1 Hopfield Network

The Hopfield model consists of N binary threshold units where each neuron is binary, it is either active $(s_i = 1)$ or inactive $(s_i = -1)$ [30]. In a Hopfield model, every pair of units i and j are connected by a weight w_{ij} [31], [32]. Each neuron updates itself according to the rule:

$$s_i = sgn\left(net_i\right) = sgn\left(\sum_{j=1}^{N} w_{ij} \cdot s_j\right). \tag{3.1}$$

The dynamic update described in Equation (3.1) can be performed either synchronously, where at each clock cycle all s_i are updated in a synchronous way according to Equation (3.1) or asynchronously, where each s_i is updated independently. In what follows, we will consider the asynchronous update rule, which seems biologically more plausible than the synchronous one [6]. The basic problem setup is that we want to store a set of patterns $\boldsymbol{\xi_\mu}$ where $\mu = 1, 2, ..., p$ in such a way that when the network is initialized with a new pattern $\boldsymbol{\zeta_\mu}$, it should eventually converge, through the update dynamics, to the stored pattern that most closely resembles $\boldsymbol{\zeta_\mu}$ [33]. We start by considering the storage of a single pattern $\boldsymbol{\xi} = [\xi_1, ..., \xi_N]$ [34]. From Equation (3.1) we can conclude that the condition for the pattern $\boldsymbol{\xi}$ to be stable is:

$$\xi_i = sgn\left(\sum_{j=1}^{N} w_{ij} \cdot \xi_j\right), \quad \forall_i \in \{1, ..., N\}. \tag{3.2}$$

So, to store a pattern $\boldsymbol{\xi}$, we need to find a weight matrix $\mathbf{w}$ that satisfies Equation (3.2). This is guaranteed if

$$w_{ij} \propto \xi_i \cdot \xi_j. \tag{3.3}$$

For convenience, we consider that the constant of proportionality is $1/N$ where N represents the number of units of the network, and so we can conclude that the rule to store a single pattern $\boldsymbol{\xi}$ is:

$$w_{ij} = \frac{1}{N} \cdot \xi_i \cdot \xi_j. \tag{3.4}$$

When the network is initialized with a new pattern $\boldsymbol{\zeta_\mu}$, and fewer than half of the bits are different from the stored pattern $\boldsymbol{\xi}$, then the network will converge to $\boldsymbol{\xi}$, and therefore we can say that $\boldsymbol{\xi}$ is an attractor of the system. If the starting configuration has more than half the bits different from $\boldsymbol{\xi}$, it will end up in the reversed state $-\boldsymbol{\xi}$ which is an attractor of the system as well [34], [6]. In the case "where" we have a set of P binary patterns ξ_μ $(\mu = 1, 2, \ldots, P)$ we use the Hebbian learning rule in order to store those $\boldsymbol{\mu}$ patterns:

$$w_{ij} = \frac{1}{N} \cdot \sum_{\mu=1}^{p} \xi_i^\mu \cdot \xi_j^\mu, \tag{3.5}$$

where w_{ij} = w_{ji} for all i, j and $w_{ii} = 0$ for all i [34].

Previously, we saw that given a new pattern $\boldsymbol{\zeta_\mu}$, the network can converge either to $\boldsymbol{\xi}$ or to $-\boldsymbol{\xi}$. However, those are not the only attractors of the system, sometimes the network can converge to spurious states. These states are different from the training patterns, though if the network initial configuration is close to a spurious state, the network can converge to it. This constitutes a problem as we do not want the network to converge to spurious states [6].

Figure 3.1 illustrates the concepts of recalling a stored pattern (good minimum) or a spurious pattern previously explained.

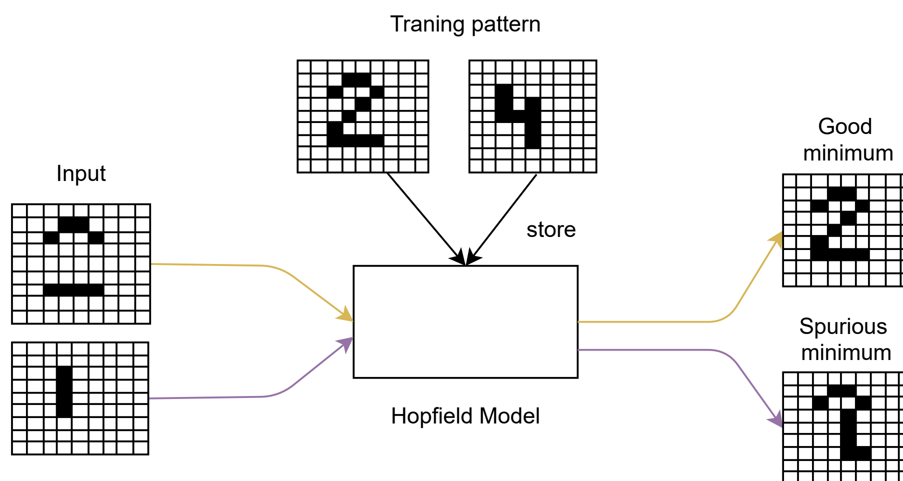


Figure 3.1: MNIST handwritten digits storage and retrieval using Hopfield Model.

The storage capacity of the Hopfield network tells us how many patterns can be stored in the model while still being able to recall them. This amount is determined by the number of neurons within a network given by n and can be shown [24] to be equal to $C = 0.138 \cdot n$.

3.1.1 Energy function

One of the most important properties of the Hopfield model was the introduction of an energy function. Each binary "configuration" of the whole network has an energy, which is given by the following equation:

$$H = -\frac{1}{2} \cdot \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} \cdot s_i \cdot s_j. \tag{3.6}$$

The global energy function of the system can be described as a landscape in which the valleys correspond to the local minima of the energy surface where the attractors (memorized patterns) are [35]. The main property of the energy function is that it always decreases (or remains constant) as the system evolves according to its dynamic rule. The dynamics can be thought of as the motion of a particle on the energy surface under the influence of gravity and friction. From any starting point, the particle slides downhill until it reaches the lowest point of a valley (local minimum) — one of the attractors of the system as shown in 3.2 [6], [36].

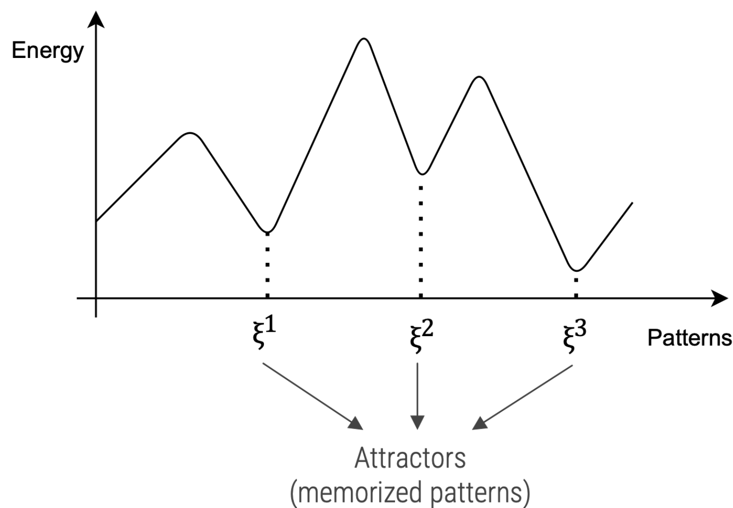


Figure 3.2: The energy surface where the valleys correspond to the attractors of the system.

3.2 Ising Model

The **Ising model** consists of discrete variables (s_i) that represent magnetic dipole moments of atomic "spins" which can be oriented in one of two different ways, either "up" if $s_i = +1$ or "down" if $s_i = -1$. These spins are arranged in a lattice, allowing each spin to interact with its neighbors. We can relate the Ising model to the Hopfield associative memory in which an active unit in the network corresponds to "spin up" in the magnet and an inactive one to "spin down" . In a magnetic material each of the spins is influenced by the magnetic field h, this magnetic field consists of an internal field produced by the other

spins plus an external field h_{ext} [37]. Thus, considering the contributions of all the neighboring spins we have the following magnetic field for spin s_i

$$h_i = \sum_{j=1}^{N} w_{ij} \cdot s_j + h_{ext}. \tag{3.7}$$

The coefficients w_{ij} measure the strength of the influence of spin s_j on the field at s_i. In a magnet these interactions are necessarily symmetric, therefore we have that w_{ij} = w_{ji} [24]. Considering the system at a low temperature, we can say that spin s_i tends to line up parallel to the local field h_i. So, we have that, at a low temperature the spin is updated by the following expression

$$s_i = sgn(h_i) = \sum_{j=1}^{N} w_{ij} \cdot s_j + h_{ext}. \tag{3.8}$$

The spin updates are taken to happen asynchronously in random order. Besides, we can specify the Hamiltonian (energy function) corresponding to Equation (3.8) as

$$H = -\frac{1}{2} \cdot \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} \cdot s_i \cdot s_j - h_{ext} \sum_{i=1}^{N} s_i. \tag{3.9}$$

3.2.1 Spin glass

The Spin glass model is an Ising model without the influence of an external field; the only influence each spin is subject to is the internal field produced by the other spins [24]. Therefore, we can represent the energy function (Hamiltonian) of the system as

$$H = -\frac{1}{2} \cdot \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} \cdot s_i \cdot s_j. \tag{3.10}$$

3.2.2 Finite temperature dynamics

So far, we have seen the behavior of the system at low temperatures, where the spins are updated deterministically following Equation (3.8). At high temperatures the thermal fluctuations tend to flip the spins randomly from up to down or from down to up. In an Ising model, the thermal fluctuations can be described by the Glauber dynamics that result in the following stochastic rule

$$s_i = \begin{cases} +1 & with\, probability\; g(h_i) \\ -1 & with\, probability\; 1 - g(h_i) \end{cases} \tag{3.11}$$

This rule is applied whenever a spin s_i is updated, where the Glauber function $g(h_i)$ depends on the temperature of the system and is a sigmoid-shaped function

$$g(h) = \frac{1}{1 + e^{(-2 \cdot \beta \cdot h)}}, \tag{3.12}$$

where β is related to the absolute temperature of the system T by

$$\beta = \frac{1}{k \cdot T}, \tag{3.13}$$

with k being the Boltzmann's constant. Note that

$$g(-h) = 1 - g(h). \tag{3.14}$$

So, we can write that the probability of each spin s_i being 1 or –1 as

$$p(\pm s_i) = g(\pm h_i) = \frac{1}{1 + e^{(\pm 2 \cdot \beta \cdot h_i)}}. \tag{3.15}$$

3.2.3 Boltzmann-Gibbs distribution

A Boltzmann-Gibbs distribution is a probability distribution that represents the probability (P_α) that a system will be in a certain state α as a function of that state's energy H_α and the temperature of the system T . A fundamental result from physics tells us that in thermal equilibrium, the temperature within the system is spatially uniform and temporally constant, each of the possible states α occurs with probability

$$p_\alpha = \frac{1}{Z} \cdot e^{(-\frac{H_\alpha}{K \cdot T})}, \tag{3.16}$$

where Z is a normalization constant called the sum over states, or the partition function, to allow the total probability (probability of being in each one of the α possible states) to be 1. It is represented by the symbol Z because the German name for this term is Zustadsumme.

So, we get

$$Z = \sum_\alpha e^{(-\frac{H_\alpha}{K \cdot T})}. \tag{3.17}$$

In neural networks, the temperature T of a stochastic network is not related to the physical temperature, it is used as a parameter to control the update rule. So, its scale is irrelevant, and we can choose to measure it in units such that $k = 1$.

This way if we know the energy function H_α we can use Equation (3.16) in order to compute the probability of finding the network in each one of its possible states α. Then we can compute the average value $\langle A \rangle$ (thermal average) of any quantity A which has a particular value A_α in each possible state α,

through

$$\langle A\rangle = \sum_{\alpha} A_\alpha \cdot p_\alpha. \tag{3.18}$$

3.2.4 Stochastic dynamics

We are interested in rewriting Equation (3.15) as a transition probability of flipping the spin from s_i to $-s_i$. This can be represented by

$$W(s_i \to -s_i) = \frac{1}{1+e^{(\beta\cdot\Delta H_i)}}, \tag{3.19}$$

where

$$\Delta H_i = H(s_1, s_2, \ldots, -s_i, \ldots, s_N) - H(s_1, s_2, \ldots, s_i, \ldots, s_N) = 2\cdot h_i \cdot s_i. \tag{3.20}$$

Equation (3.20) represents the energy change that occurs in the system when we have a spin s_i flipping. In fact, the general case of a transition probability for all pairs of states α and $\alpha\prime$ is represented by $W(\alpha \to \alpha\prime)$. In equilibrium the probability p_α of finding the system in state α is given by the Boltzmann-Gibbs distribution. At thermal equilibrium, the rate of occurrence of any transition equals the corresponding rate of occurrence of the inverse transition, as shown by

$$p_\alpha W\left(\alpha \to \alpha'\right) = p\prime_\alpha W\left(\alpha' \to \alpha\right), \tag{3.21}$$

and

$$\frac{W\left(\alpha \to \alpha'\right)}{W\left(\alpha' \to \alpha\right)} = \frac{p\prime_\alpha}{p_\alpha} = e^{(-\beta\cdot\Delta H)}, \tag{3.22}$$

where p_α represents the probability that the system is in state α and

$$\Delta H = H_\alpha - H_{\alpha'}. \tag{3.23}$$

This way we have the principle of detailed balance, and we can state that

$$W(\alpha \to \alpha') = \frac{1}{1+e^{(\beta\cdot\Delta H)}}, \qquad W(\alpha' \to \alpha) = \frac{1}{1+e^{(-\beta\cdot\Delta H)}}, \tag{3.24}$$

with Boltzmann-Gibbs distribution, we have that

$$p_\alpha = \frac{1}{Z} e^{(-\beta\cdot H_\alpha)}, \qquad p'_\alpha = \frac{1}{Z} e^{(-\beta\cdot H'_\alpha)}, \tag{3.25}$$

with that in mind, one could simulate the dynamics of such a stochastic system, using the Metropolis algorithm as follows

$$W(\alpha \to \alpha') = \begin{cases} 1 & if\, \Delta H < 0 \\ e^{(-\beta \cdot H_\alpha)} & otherwise \end{cases} \tag{3.26}$$

The stochastic dynamics are described by Monte Carlo methods. As described in Section 2.4 these are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results.

3.2.5 How an Ising Model Generates Data

The stochastic dynamics of the Ising model can be described by Gibbs sampling [38]. Suppose the system is in a state s and we have chosen an arbitrary coordinate i. We can then ignore the actual state of the spin s_i and ask for the conditional probability that this spin points upwards $(s_i = 1)$ given all other spins.

$$h_i = \sum_{j=1, j\neq i}^{N} w_{ij} \cdot s_j + h_{ext}, \tag{3.27}$$

with

$$p(s_i = 1 | \{s_j\}_{j\neq i}) = \frac{1}{1 + e^{(-2\cdot\beta\cdot h_i)}}, \quad \beta = \frac{1}{k \cdot T}. \tag{3.28}$$

A single Gibbs sampling step now proceeds as follows:

1. Randomly pick a coordinate i.
2. Calculate the conditional probability $p = p(s_i = 1|\{s_j\}_{j\neq i})$.
3. Draw a real number $\xi \in [0, 1]$ from the uniform distribution.
4. If ξ is at most equal to p, set spin i to $+1$, otherwise set it to -1.

3.3 Boltzmann Machine

We introduce the BM in order to have hidden units that can help capturing relations between visible units. This model has a more expressive power then the previously addressed ones.

The BM can be seen as a stochastic Hopfield network with hidden units. It is a network of symmetrically connected units that makes stochastic decisions about whether to be active $(s_i = 1)$ or inactive $(s_i = 0)$. These units are divided into hidden units and visible units [39].

Visible neurons provide an interface between the network and the environment in which it operates, while hidden units have no connection with the environment [24].

Usually the units are updated asynchronously, which means that one unit is updated at a time. When unit i is given the opportunity to update its binary state, it follows the update rule :

$$net_i = \sum_{j=1, j \neq i}^{N} w_{ij} \cdot s_j + b_i, \tag{3.29}$$

with b_i being the bias. As previously referred, in this model the units have a stochastic behavior, therefore we say that unit i becomes active with a probability given by:

$$p(s_i = 1 | s_1, ... s_{i-1}, s_{i+1}, s_N) = \frac{1}{1 + e^{(-\beta \cdot net_i)}} \quad , \beta = \frac{1}{T}. \tag{3.30}$$

For a particular set of parameters w_{ij} and b_i, the BM defines a probability distribution over various state configurations [40], [24]. The energy of a particular configuration s is denoted by:

$$H(s) = -\sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} \cdot s_i \cdot s_j - \sum_{i=1}^{N} b_i \cdot s_i. \tag{3.31}$$

However, these configurations are only probabilistically known in the case of the BM. The conditional distribution of Equation (3.30) follows from a more fundamental definition, previously presented in Equation (3.16), of the unconditional probability $P(s)$ of a particular configuration s:

$$P(s) = \frac{1}{Z} e^{(-H(s))}. \tag{3.32}$$

We define Z as:

$$Z = \sum_{s} e^{(-H(s))}. \tag{3.33}$$

If the weights on the connections are chosen so that the energies of state vectors represent the cost of those state vectors, then the stochastic dynamics of a BM can be viewed as a way of escaping from poor local optima while searching for good (low-cost) solutions. The total input to unit i (net_i) represents the difference in energy depending on whether that unit is active $(s_i = 1)$ or inactive $(s_i = 0)$, and the fact that unit i occasionally becomes active even if net_i is negative means that the energy can occasionally increase during the search, which allows the search to jump over energy barriers.

The search can be improved by using simulated annealing (described in section 2.5). This scales down all of the weights and energies by a factor which is analogous to the temperature of a physical system, T. If we apply simulated annealing, we can start from a large initial value of T and decrease it to a small final value [41].

3.3.1 How a Boltzmann Machine Generates Data

In a BM, the dynamics of the data generation is complicated as there are dependencies between states. Therefore, we need an iterative process to generate sample data points from the BM so that Equation (3.30) is satisfied for all states [42].

We can use Gibbs sampling to describe the stochastic dynamics of a BM. Consider that the system is in a state s and we have chosen a random coordinate i. In order to compute the probability of unit s_i being active, we can ignore the actual state of unit s_i and ask for the conditional probability

$$p(s_i = 1|\{s_j\}_{j\neq i}) = \frac{1}{1 + e^{(-\beta \cdot net_i)}}. \tag{3.34}$$

A single Gibbs sampling step now proceeds as follows:

1. Randomly pick a coordinate i.
2. Calculate the conditional probability $p = p(s_i = 1|\{s_j\}_{j\neq i})$.
3. Draw a real number $\xi \in [0, 1]$ from the uniform distribution.
4. If ξ is at most equal to p, set unit i to $+1$, otherwise set it to 0.

Provided that the stochastic simulation is performed long enough, the network will reach thermal equilibrium at temperature T. The search can be improved by using simulated annealing, as this process reduces the time the network takes to reach thermal equilibrium [24].

3.3.2 Learning

Given a training set of binary state vectors (training data), learning consists of finding weights and biases (network parameters) that define a Boltzmann distribution in which the training vectors have high probability. This means maximizing the log-likelihood of the specific training data set [43]. The probability associated with finding the network in a particular global state depends on the energy function. The log-likelihoods of individual states are computed by using the logarithm of the probabilities [44]. By differentiating Equation (3.31) and using the fact that $\frac{\partial H}{\partial w_{ij}} = -s_i s_j$, it can be shown that by taking the logarithm of Equation (3.32), we obtain the following:

$$\sum_{s \in data} \frac{\partial log[P(s)]}{\partial w_{ij}} = \langle s_i s_j \rangle_{data} - \langle s_i s_j \rangle_{model}. \tag{3.35}$$

We have that $\langle s_i s_j \rangle_{data}$ represents the averaged value of $s_i s_j$ at thermal equilibrium when the visible states are clamped to attribute values in a training point. While $\langle s_i s_j \rangle_{model}$ represents the averaged value of $s_i s_j$ at thermal equilibrium without fixing visible states to training points (with no external interference) [14], [45].

During the network training, there are two phases:

- In the positive phase, the weights are raised in proportion to the correlations between the states of nodes i and j, when the visible vectors are clamped to a vector in the training data and the hidden states are randomly chosen to be 0 or 1. Gibbs sampling is performed until we reach thermal equilibrium. In this phase the network learns the good minima (attractors of the system). Though, the network can learn as well unwanted minima called spurious minima, for that reason we need the negative phase (unlearning phase)

$$\langle s_i s_j \rangle_{data} = \sum_{k=1}^{N} \sum_{h} P(v_k, h) \cdot s_i \cdot s_j. \tag{3.36}$$

- In the negative phase, the network is allowed to run freely, and therefore with no external input where states of the units are determined randomly (0 or 1). In this phase the weights are reduced in proportion to how often those two units are active together when sampling from the model's distribution. Gibbs sampling is performed in this phase until we reach thermal equilibrium. During this phase the network unlearns the spurious minima, as these are minima that the network should not converge to.

$$\langle s_i s_j \rangle_{model} = \sum_{k=1}^{N} \sum_{s} P(s) \cdot s_i \cdot s_j. \tag{3.37}$$

After performing the positive and negative phases of the learning process we can perform the gradient ascent and adapt the weights as follows:

$$w_{ij} = \eta \cdot (\langle s_i s_j \rangle_{data} - \langle s_i s_j \rangle_{model}). \tag{3.38}$$

3.4 Restricted Boltzmann Machine

The negative phase of learning process (Equation (3.37)) in BM model is slow and expensive, particularly when the number of hidden neurons used in the machine is large. The reason for this undesirable behavior is that the BM takes a long time to reach an equilibrium distribution, thereby limiting its practical usefulness. Given these limitations we introduce the so-called RBM as discussed in [46] and [2].

RBM were initially invented under the name Harmonium [14]. They are a variant of BMs, with the restriction that there is a single layer of m visible units $\mathbf{v} = (v_1, v_2, ..., v_m)$ and a single layer of n hidden units $\mathbf{h} = (h_1, h_2, ..., h_n)$ with no visible-visible or hidden-hidden connections.

With Figure 3.3 it is easier to understand the main differences in the architectures of the Hopfield model, BM, and the RBM.

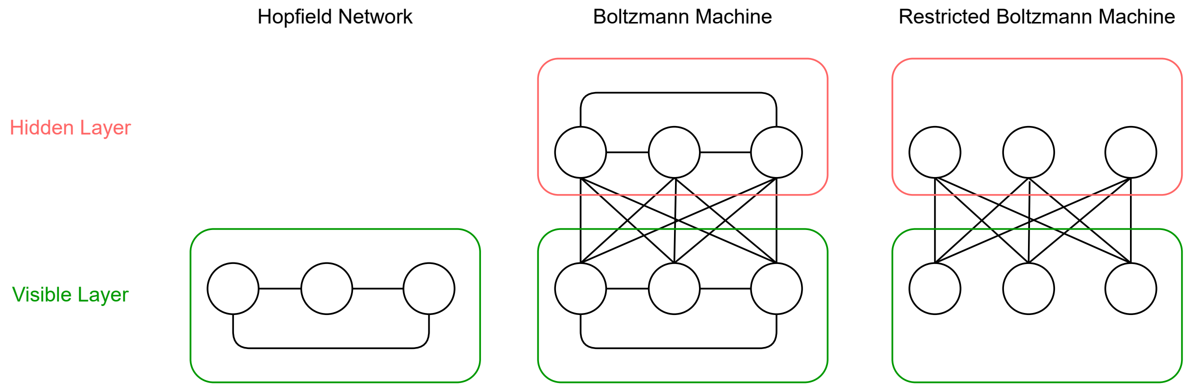


Figure 3.3: Comparison between the architectures of the Hopfield Model, the BM and the RBM

The energy function of an RBM can be written as

$$H(v,h) = -\sum_{i=1}^{n}\sum_{j=1}^{m} w_{ij} \cdot h_i \cdot v_j - \sum_{j=1}^{m} b_j \cdot v_j - \sum_{i=1}^{n} c_i \cdot h_i. \tag{3.39}$$

For all $i \in 1, ..., n$ and $j \in 1, ..., m$, w_{ij} is a real valued weight associated with the edge between the units v_j and h_i, and b_j and c_i are real valued bias terms associated with unit j of the visible layer and unit i of the hidden layer, respectively.

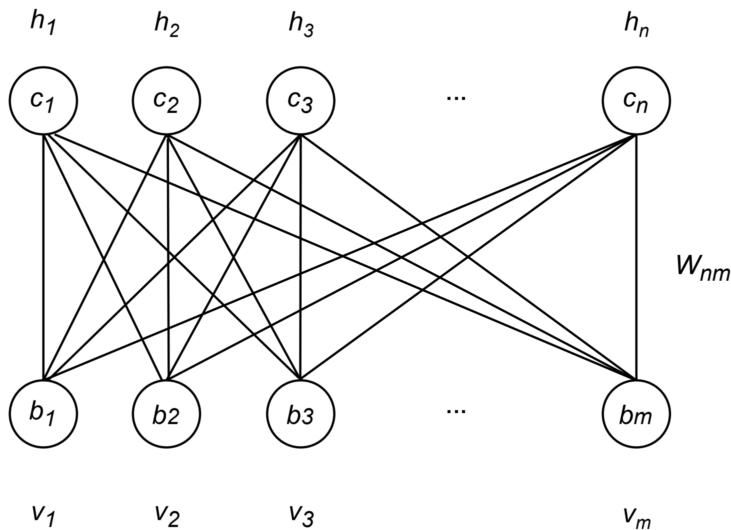


Figure 3.4: Network graph of an RBM with n hidden units and m visible units

The graph of an RBM has no connections between two variables of the same layer, as we can observe in Figure 3.4. In terms of probability, this means that the visible variables are independent given the state of the hidden variables and vice versa:

$$p(h|v) = \prod_{i=1}^{n} p(h_i|v), \tag{3.40}$$

and

$$p(v|h) = \prod_{j=1}^{m} p(v_j|h). \tag{3.41}$$

The conditional independence between the variables in the same layer makes Gibbs sampling an easy task. Instead of sampling new values for all variables subsequently, the states of all variables in each layer can be sampled jointly. Thus, Gibbs sampling can be performed sampling a new state h for the hidden neurons based on $p(h|v)$ and sampling a state v for the visible layer based on $p(v|h)$. This process is also referred to as block Gibbs sampling [21] and it represents the main advantage in using RBM instead of BM as the Negative phase of the learning process (unlearning phase) becomes considerably simplified.

The weights update of the RBM is computed using a type of learning rule similar to the one used in BM. In particular, it is possible to create an efficient algorithm based on mini-batches. The weights w_{ij} are initialized to small values, and for the current set of weights w_{ij}, they are updated as follows:

1. Positive phase: Visible units are clamped and the hidden units are randomly chosen (0 or 1). The algorithm uses a mini-batch of training instances, and computes the probability of the state of each hidden unit in exactly one step. Then a single sample of the state of each hidden unit is generated from this probability. This process is repeated for each element in a mini-batch of training instances. The correlation between these different training instances of v_i and generated instances of h_j is computed; it is denoted by $\langle v_i h_j \rangle_{data}$. This correlation is essentially the average product between each such pair of visible and hidden units [42].

2. Negative phase: Visible and hidden units are chosen randomly (0 or 1). The algorithm starts with a mini-batch of training instances and then for each training instance, it goes through a phase of Gibbs sampling after starting with randomly initialized states. This is achieved by using Equations (3.40) and (3.41) to compute the probabilities of the visible and hidden units, and using these probabilities to draw samples. The values of v_i and h_j at thermal equilibrium are used to compute $\langle v_i h_j \rangle_{model}$ in the same way as the positive phase [42].

We can write our update rule as in BM:

$$\Delta w_{ij} = \eta \cdot (\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}). \tag{3.42}$$

After training the model we clamp the visible units with some configuration s_{query}. The network will converge to an attractor (stored pattern) after performing several steps using the update rule [39].

3.4.1 Contrastive divergence

Obtaining unbiased estimates of the log-likelihood gradient using MCMC methods typically requires many sampling steps. However, it has been shown that estimates obtained after running the chain for just a few steps can be sufficient for model training [21].

Contrastive Divergence (CD) speeds up the computing time of $\langle v_i h_j \rangle_{model}$ as it does not use Gibbs sampling to reach thermal equilibrium. In this algorithm, the training phase starts by clamping the visible units with v^0 and the hidden layer units h^0 can be computed by

$$p(h_i = 1|v) = \frac{1}{1 + e^{-\left(\sum_{j=1}^{n} w_{ij} \cdot v_j + c_i\right)}} = \sigma\left(\sum_{j=1}^{n} w_{ij} \cdot v_j + c_i\right), \tag{3.43}$$

that define

$$\langle v_i h_j \rangle_{data}^{0} . \tag{3.44}$$

As we saw previously there are no visible-visible or hidden-hidden connections. For that reason each unit h_i is independent of the other hidden units. Therefore, h^0 can be computed in parallel as each hidden unit only depends on the visible units connected to it [47].

The second step consists in updating all the visible units in parallel to get a "reconstruction" v^1, which can be computed by

$$p(v_i = 1|h) = \frac{1}{1 + e^{-\left(\sum_{j=1}^{n} w_{ij} \cdot h_j + b_i\right)}} = \sigma\left(\sum_{j=1}^{n} w_{ij} \cdot h_j + b_i\right), \tag{3.45}$$

that define

$$\langle v_i h_j \rangle_{recon}^{1} . \tag{3.46}$$

The visible units are now clamped with v^1 and the hidden layer units h^1 are computed in parallel using Equation (3.43).

The reconstruction algorithm can be computed τ times or until convergence is reached. Sometimes CD may take many iterations $(1 \ll \tau)$ to converge. When $\tau = 1$ we are computing a single-step reconstruction [24].

The weights update computed for τ steps of the reconstruction algorithm is given by

$$\Delta w_{ij} = \eta \cdot (\langle v_i h_j \rangle_{data}^{0} - \langle v_i h_j \rangle_{recon}^{\tau}). \tag{3.47}$$

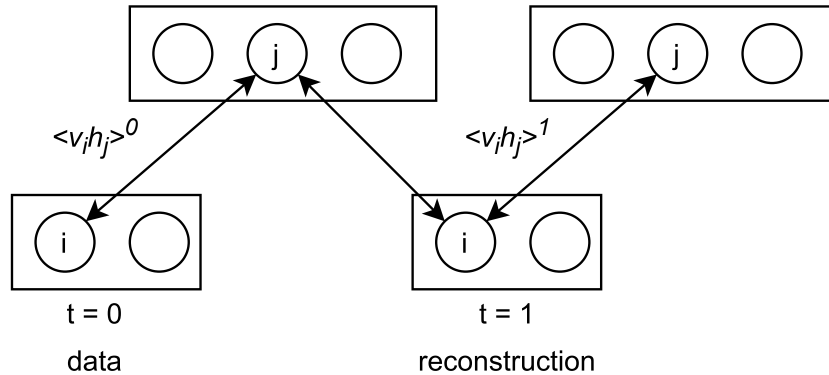


Figure 3.5: CD with single-step reconstruction

3.4.2 Persistent Contrastive divergence

A different strategy that resolves many of the problems with CD is to initialize the Markov chains at each gradient step with their states from the previous gradient step. This approach was first discovered under the name Stochastic Maximum Likelihood in the applied mathematics and statistics community and later independently rediscovered under the name Persistent Contrastive divergence (PCD) [48].

The idea behind this approach is that, as long as the steps taken by the stochastic gradient algorithm are small, the model from the previous step will be similar to the current model. It follows that the samples from the previous model's distribution will be very close to being fair samples from the current model's distribution.

As each Markov chain is continually updated throughout the learning process, rather than restarted at each gradient step, the chains are free to wander far enough to find all the model's minima. PCD is thus considerably more resistant to forming models with spurious minima than the original CD algorithm is [49].

3.4.3 Weight-decay

There are a number of regularization techniques that are widely used. One of this technique is weight-decay regularization which adds an extra term to the normal gradient. In fact, this extra term is the derivative of a function that penalizes large weights and is given by the term:

$$L(w) = \frac{\lambda}{2} \sum_{i=1}^{M} w_{ij}^2. \tag{3.48}$$

There are diverse reasons to use regularization with RBMs. The first one is to reduce overfitting, i.e. improve generalization to new data. Additionally, the use of regularization makes the hidden units receptive fields smoother and more interpretable by shrinking useless weights [50].

3.4.4 Momentum

Momentum is a simple method that helps increase the speed of learning. When the objective function contains long, narrow and straight ravines with a gentle but consistent gradient along the floor of the ravine and much steeper gradients up the sides of the ravine. The momentum method simulates a heavy ball rolling down a surface. The ball builds up velocity along the floor of the ravine, but not across it because the opposing gradients on opposite sides of the ravine cancel each other out over time [50].

Instead of using exclusively the estimated gradient times the learning rate to increment the values of the parameters, the momentum method uses α to increment the velocity, v, of the parameters. The velocity of the ball is assumed to decay with time, though the momentum parameter α is the fraction of the previous velocity that remains after computing the gradient on a new mini-batch:

$$v_i(t) = \alpha v_i(t-1) - \eta \cdot \frac{dE}{dw_i}(t). \tag{3.49}$$

The temporal smoothing in the momentum method avoids the oscillations across the ravine that would be caused by simply increasing the learning rate. After applying the momentum method we can write our weights update rule as following:

$$\Delta w_i = v_i(t). \tag{3.50}$$

3.4.5 Different types of units

In the previous sections we have been describing the Bernoulli-Bernoulli RBM, which uses binary visible and hidden units. However, many other types of unit can also be used.

3.4.5.A Softmax visible units

The softmax unit is the appropriate way to deal with a quantity that has K alternative values which are not ordered in any way. A softmax can be viewed as a set of binary units whose states are mutually constrained so that exactly one of the K states has value 1 and the remaining have value 0 [51]. With softmax visible units, the learning rule is identical to the rule for standard binary units. So, the CD (section 3.4.1) and PCD (section 3.4.2) can still be applied in the same way [50].

Actually, the only difference is in the way the probabilities of the visible states are computed. The probability of the visible units given the activation of the hidden units is described by the following equation:

$$p\left(v_i = 1 \mid \boldsymbol{h}\right) = \frac{\exp\left(b_i + \sum_{j=1}^{F} h_j w_{ij}\right)}{\sum_{l=1}^{K} \exp\left(b_i^l + \sum_{j=1}^{F} h_j w_{ij}^l\right)}. \tag{3.51}$$

3.4.5.B Gaussian visible units

The Gaussian-Bernoulli RBM has visible units with real-value v_m and binary hidden units h_n. The conditional probabilities for visible and hidden units are described by the following equations:

$$p(v_i = v \mid \boldsymbol{h}) = N\left(\boldsymbol{v} \mid b_i + \sum_j h_j w_{ij}, \sigma_i^2\right), \tag{3.52}$$

$$p(h_j = 1 \mid \boldsymbol{v}) = f\left(c_j + \sum_i w_{ij} \frac{v_i}{\sigma_i^2}\right), \tag{3.53}$$

where $N(\mu, \sigma^2)$ denotes the Gaussian probability density function with mean μ and standard deviation σ^2 [51].

In the parameter updating process, the CD learning is highly successful and is becoming the standard learning method to train the Bernoulli-Bernoulli RBM parameters [52]. However, when using CD with Gaussian visible units, it is difficult to learn the variance of the noise for each visible unit [50]. With Gaussian-Bernoulli RBM the update rule for the weights is defined by the following equation:

$$\nabla w_{ij} =< \frac{1}{\sigma_i^2} v_i h_j >_{data} - < \frac{1}{\sigma_i^2} v_i h_j >_{model} . \tag{3.54}$$

3.4.6 Restricted Boltzmann Machine for classification

Until now we have described RBM as a generative model, where given a corrupted pattern we can reconstruct the original pattern. Actually, this section describes how RBM model can also be used as a classifier.

First we have the training phase, where the RBM learns to model the joint probability distribution of input data (explanatory variables) and the corresponding labels (output variables), both represented by the visible units of the model as shown in Figure 3.6. The RBM is trained with one of the previously described algorithm: either CD (described in 3.4.1) or PCD (described in 3.4.2).

Following the training phase, we have the sampling where the label corresponding to an input example can be obtained by fixing the visible variables that correspond to the data and then sampling the remaining visible variables allocated to the labels from the joined probability distribution of data and labels modeled by the RBM. Hence, a new input example can be clamped to the corresponding visible neurons and the label can be predicted by sampling [21].

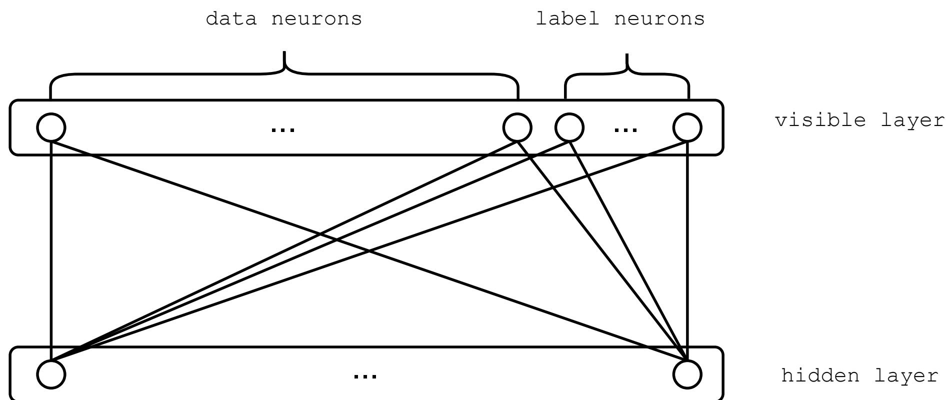


Figure 3.6: RBM that models the joint probability distribution of input images and the corresponding labels.

3.4.7 Deal with missing data with RBMs

The paper “Restricted Boltzmann Machines for Collaborative Filtering” [2] shows how a class of two-layer undirected graphical models, called RBM, can be used to model tabular data, such as user’s ratings of movies.

Considering a dataset with M movies, N users, and integer rating values from 1 to K. A highly relevant problem when applying the RBM model to movie ratings is efficiently dealing with missing ratings. Ideally all the N users would have rated the same set of M movies, and each user could be treated as a single training case for an RBM which had M softmax visible units symmetrically connected to a set of binary hidden units.

However, when dealing with missing ratings, a different RBM is used for each user. Every “single user RBM” has the same number of hidden units, but each RBM only has visible softmax units for the movies rated by that user. So, if a user rated few movies, then the corresponding RBM has few connections. Even though each “single user RBM” only has a single training case, all the corresponding weights and biases are tied together.

So, if two users have rated the same movie, both RBMs must use the same weights between the softmax visible unit for that movie and the hidden units. The binary states of the hidden units, however, can be quite different for different users, given that the hidden units’ activation is calculated with respect to each single user RBM visible units. Finally, the full gradients with respect to the shared weight parameters can then be obtained by averaging over all N users.

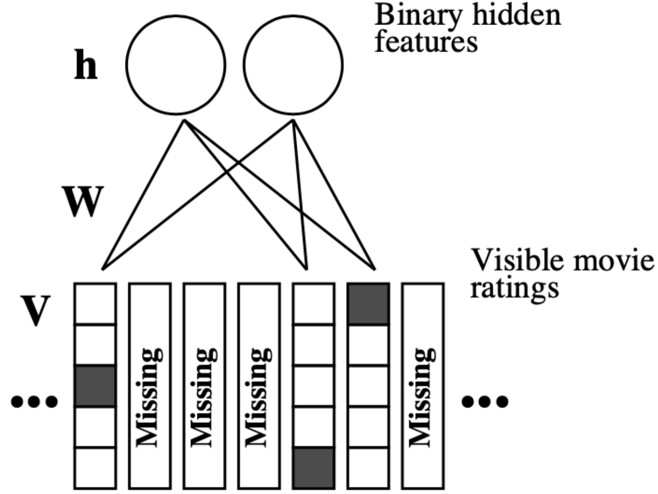


Figure 3.7: Adapted from [2], a RBM with binary hidden units and softmax visible units is represented. For each dataset sample (user), the RBM only includes softmax units for the movies that user has rated. In addition, to the symmetric weights between each hidden unit and each of the K = 5 values of a softmax unit, there are 5 biases for each softmax unit and one for each hidden unit.

Suppose a user rated M movies. Let V be a $K \times M$ observed binary indicator matrix with $v_i^k = 1$ if the user rated movie i as k and 0 otherwise. We also let $hj, j = 1, ..., F$, be the binary values of hidden variables, that can be thought of as representing stochastic binary features that have different values for different users.

3.4.7.A The model

To model each column of the observed visible binary rating matrix V, a conditional multinomial distribution (softmax) is used. So, the probability of the k softmax visible units given the activation of the hidden units is given by the following equation:

$$p\left(v_i^k = 1 \mid \mathbf{h}\right) = \frac{\exp\left(b_i^k + \sum_{j=1}^{F} h_j W_{ij}^k\right)}{\sum_{l=1}^{K} \exp\left(b_i^l + \sum_{j=1}^{F} h_j W_{ij}^l\right)}. \tag{3.55}$$

The hidden units follow a conditional Bernoulli distribution, and the probability of each hidden unit given the visible layer is given by the following equation:

$$p\left(h_j = 1 \mid \mathbf{V}\right) = \sigma\left(b_j + \sum_{i=1}^{m} \sum_{k=1}^{K} v_i^k W_{ij}^k\right), \tag{3.56}$$

where σ is the logistic function, W_{ij}^k is a symmetric interaction parameter between feature j and rating k of movie i, b_i^k is the bias of rating k for movie i, and b_j is the bias of feature j.

Moreover, the marginal distribution over all the visible ratings V is given by the following equation:

$$p(\mathbf{V}) = \sum_{\mathbf{h}} \frac{\exp(-E(\mathbf{V}, \mathbf{h}))}{\sum_{\mathbf{V}', \mathbf{h}'} \exp\left(-E\left(\mathbf{V}', \mathbf{h}'\right)\right)}. \tag{3.57}$$

Each binary "configuration" of the whole network, visible and hidden units, has an energy, which is given by the following equation:

$$E(\mathbf{V}, \mathbf{h}) = -\sum_{i=1}^{m}\sum_{j=1}^{F}\sum_{k=1}^{K} W_{ij}^{k} h_j v_i^k + \sum_{i=1}^{m} \log Z_i - \sum_{i=1}^{m}\sum_{k=1}^{K} v_i^k b_i^k - \sum_{j=1}^{F} h_j b_j, \tag{3.58}$$

where $Z_i = \sum_{l=1}^{K} \exp\left(b_i^l + \sum_j h_j W_{ij}^l\right)$. The movies with missing ratings do not make any contribution to the energy function.

3.4.7.B Learning

In order to update the required parameters during learning process and reviewing what we studied in section 3.3.2, to perform gradient ascent, i.e. maximizing the log-likelihood of the specific training data set, we have the following equation:

$$\Delta W_{ij}^{k} = \eta \cdot \frac{\partial \log p(\mathbf{V})}{\partial W_{ij}^{k}} = \eta \cdot \left(< v_i^k h_j >_{\text{data}} - < v_i^k h_j >_{\text{model}} \right), \tag{3.59}$$

where η is the learning rate. The expectation $\langle v_i^k h_j \rangle_{data}$ defines the frequency with which movie i with rating k and feature j are on together when the features are being driven by the observed user-rating data from the training set using Equation 3.56, while $\langle v_i^k h_j \rangle_{model}$ is an expectation with respect to the distribution defined by the model. This expectation cannot be computed analytically in less than exponential time. So, as we previously discussed, an approximation to the gradient is followed, which is called Constrastive Divergence.

Recalling the Equation 3.47 previously defined, the expectation $\langle v_i h_j \rangle_{recon}^{\tau}$ represents a distribution of samples from running the Gibbs sampler, initialized at the data, for τ full steps. τ is typically set to one at the beginning of learning and increased as the learning converges.

However, by increasing τ to a sufficiently large value, it is possible to approximate maximum likelihood learning arbitrarily well [53], but large values of τ are seldom needed in practice. When running the Gibbs sampler, the distribution over the non-missing ratings are the only reconstructed. Considering that each user N has a respective RBM, the approximate gradients of CD with respect to the shared weight parameters can be then averaged over all N users.

3.5 Deep Belief Networks

A DBN is a powerful generative model that uses a deep architecture of multiple stacks of RBM .

DBNs were one of the first non-convolutional models to successfully admit training of deep architectures [54], [55]. Deep belief networks are generative models composed of several layers of latent variables (hidden units) with connections between adjacent layers but not between units within each layer [56], [11]. Deep belief networks capture the higher-level representations of input features [24].

There is an efficient, layer-by-layer procedure for learning the top-down, generative weights that determine how the variables in one layer depend on the variables in the layer above, and is composed by the following steps:

1. An RBM, as discussed in the section above, is trained directly on the input data, therefore the stochastic units in the hidden layer of the RBM are able to capture the important features that characterize the input data.

2. The activations of the trained features are then treated as input data which are used to train a second RBM. In effect, we can learn the features of features in a second hidden layer.

3. The process of learning the features of features is continued until the number n of hidden layers is reached, which is the same of saying until n RBMs have been trained.

The main idea is to apply DBNs in order to transform a Sparse Distributed Representation into a compact representation given the dimensionality reduction from layer to layer [57]. The compact representation generated by this model will then be used for classification [58].

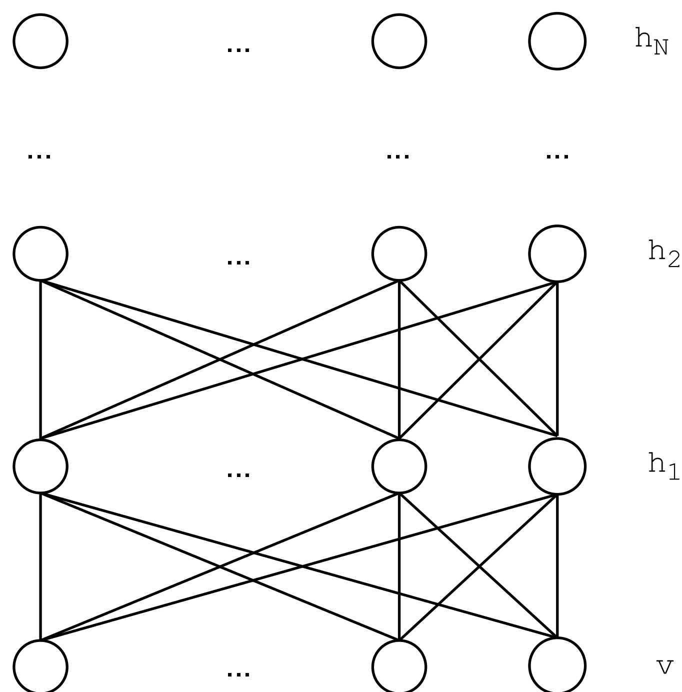


Figure 3.8: Deep Belief Network

Deep Boltzmann Machine (DBM) is another kind of deep generative model. Most of the power of conventional neural architectures arises from having multiple layers of units [59]. Even though DBMs are an interesting model to write about, they are out of the scope of the thesis.

4

Experiments

Contents

4.1 Datasets description 45
4.2 Stochastic Models for image reconstruction 47
4.3 Classification using Stochastic Models 50
4.4 Models comparison 63
4.5 Learn from a Sparse Normal Distributed Dataset 64

This section starts describing the first used datasets and briefly introduce the generation process of the sparse data.

Next, the first set of performed experiments are described in order to understand the potential of the previously described Stochastic Models. By baring in mind that a reliable model is crucial, some baseline experiments were carried out and their complexity was incrementally increased until reaching a set of suitable models, inspired in Stochastic Models and capable of performing classification.

To understand whether or not stochastic models have a better generalization performance than classic Machine Learning models when dealing with high-dimensional sparse data, a comparison was made between the implemented models and a LR. The good test accuracy achieved by the LR when classifying the generated high-dimensional sparse data, led to the suspicion that these generated datasets were "living" in a lower dimensional space.

Subsequently, a different way to generate high-dimensional sparse data was proposed, where each class follows a multivariate normal distribution and the sparseness of the data is controlled by deleting the values of random features in each sample. By investigating the RBM and the LR performance with this sparse data, we were able to conclude that the RBM shows a good generalization performance, while LR falls into overfitting.

4.1 Datasets description

Before diving into the experiments it is important to describe the datasets used in the research process. The first dataset we briefly describe is the MNIST, which was the starting point for our experimental analysis. Then, the encoding process that provided the MNIST sparse representations was described and used for the core experiments.

4.1.1 MNIST

The MNIST dataset[1], created by Yann Le Cun, contains 60,000 digits in the range 0 to 9 for training image recognition models, and another 10,000 digits as test data. Each digit is normalized and centered in a gray-level image with size 28×28, or with 784 pixel in total. Image 4.1 shows a set of ten training images of the MNIST dataset.

Figure 4.1: Ten image sample of MNIST training set.

[1] http://yann.lecun.com/exdb/mnist/

4.1.2 Sparse MNIST generation

The strategy used to generate the sparse codes is structured in [1]. In this paper, Sa-Couto & Wichert propose an encoding function that maps visual patterns into informative binary sparse vectors. This encoding requires the following two steps:

1. The Retinotopic Step: This first layer performs a local feature extraction which is organized in K planes of I x J feature extraction units. Each image is parsed with K sliding windows, with size f x f, to extract the K most relevant visual features of the images. The occurrence of the extracted features is then signalled at the middle layer. These features are determined beforehand using the unsupervised *K-means* algorithm. This layer performs information compression by establishing a many-to-one relationship between groups of pixels and receptive units. Thereby, this step transforms the dense representation presented in the input layer into a sparse representation.

2. The Object-Dependent Step: The fixed coordinate system where the features occurrences are signalled is turned into an object-dependent, radius one polar coordinate system. In this step, the mapping of the previously extracted features for each plane K to the new coordinate system is performed by a Q x Q units plane. This operation provides invariance to size and position, and makes the resulting representations well-distributed since the features detected in the previous step will be mapped quasi-uniformly into all the dimensions of the object space.

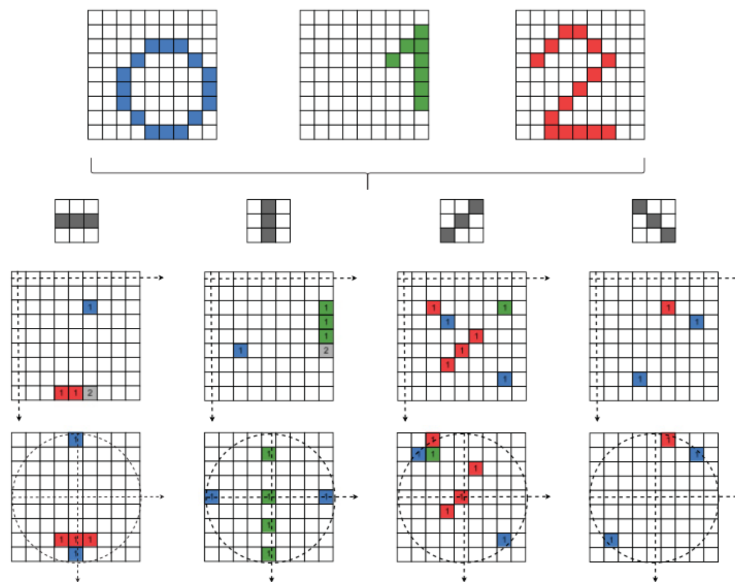


Figure 4.2: Adapted from [1], we have the overview of the strategy that transforms visual patterns of digits into sparse and distributed codes. The first step is local feature extraction (Retinotopic Step). Each feature is depicted as a window with an oriented line and each image containing a number is parsed with a sliding window, finally each occurrence of each feature is signaled at the middle layer. The Object-Dependent Step maps these positions to an object-dependent, radius one polar coordinate system.

The encoding function proposed in [1] and shown in Figure 4.2 requires as input the following parameters: *K* specifies the number of features we want to be extracted, *f x f* specifies the window size that will be parsed trough the image to extract the *K* most relevant features, *Q* represents the size of the new coordinate system plane and, finally, T_what that controls the percentage of similarity needed to recognize a feature in an image.

Thus, in order to generate the sparse representations of the MNIST dataset we provide the encoder with the train and test of the MNIST handwritten digits as well as the defined parameters, and the encoder function returns a train and a test sets with 3 dimensions. These returned sets represent a sparse binary MNIST encoding with the following shape: *(N, QxQ, K)*, where *N* is the number of samples we encoded, *Q* and *K* are the parameters described above.

4.2 Stochastic Models for image reconstruction

Before progressing to the implementation of classifiers using stochastic models, Hopfield Model was explored by performing various experiments. We started by this model as it comprises the basic concepts to fully understand the Boltzmann Machine, RBM and DBN.

Due to the pattern storage capacity of the Hopfield Model a simple dataset was used. The dataset contains only four training patterns with size 5×5, or with 25 pixels in total, as we can observe in Figure 4.3.

Figure 4.3: Training patterns

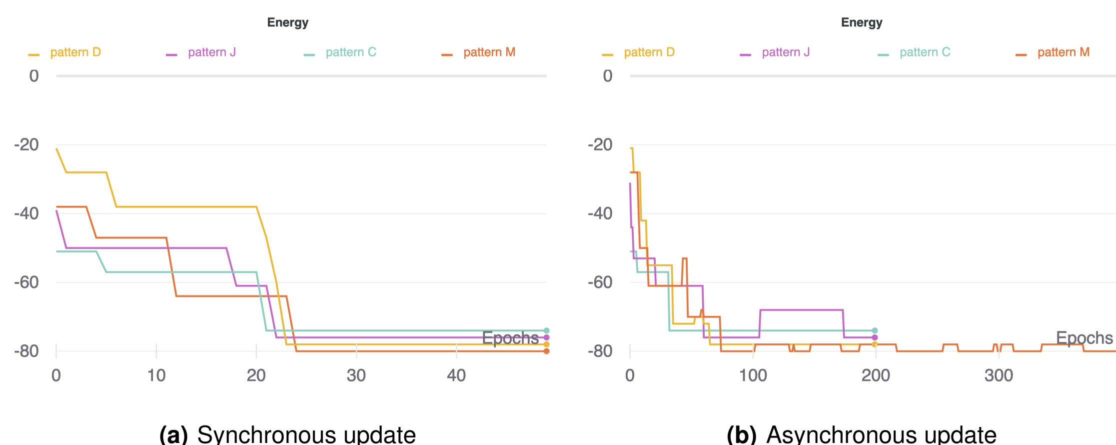
To test the Hopfield Network the four training patterns presented in Figure 4.3 were corrupted. In this way it was possible to check if, during the test phase, the original patterns were recovered.

Figure 4.4: Corrupted training patterns

In this experiment, during the training phase the network weights for the training patterns were calculated following Equation 3.5. After the training patterns were stored, we tried to visually understand if the

model was able to retrieve the original patterns. This experiment was carried out for both a deterministic and a stochastic implementation of the Hopfield Network. In the stochastic version of the model, the synchronous (Figure 4.5(a)) and asynchronous (Figure 4.5(b)) update versions were implemented. In both implementations the original patterns were fully recovered.

For each experiment, the energy of the network configuration after each bit update was updated following equation 3.6. When the energy of the neurons configuration remains the same after a full epoch, we achieve the network convergence. Each graphic in Figure 4.5 shows, for each corrupted pattern, the evolution of the energy during successive updates of the network neurons.



(a) Synchronous update

(b) Asynchronous update

Figure 4.5: Energy evolution during pattern reconstruction with stochastic update implementation of the Hopfield Network.

Although the Hopfield Network fully recovered the original patterns, we were working with a trivial dataset and due to the storage capacity, this model cannot accurately deal with complex datasets. Thus, to have a model with more expressiveness power and hidden units that can capture relations between visible units, we progressed to the study of the well-known BM model. Given the drawbacks of the negative phase of learning process in Boltzmann Machines, we decided to move forward to the implementation of the RBM model [14]. To train the RBM two different algorithms were implemented: the CD (explained in section 3.4.1) and PCD (explained in section 3.4.2).

As a starting point, the dataset previously described was used with the main objective of checking the quality of the reconstruction given the corrupted patterns in Figure 4.4.

As the RBM model perfectly reconstructed the original patterns in Figure 4.3 given the simplicity of the data, we progressed to experiments with the MNIST dataset.

First, several experiments were performed aimed at finding which architecture would better reconstruct the patterns. In this primary stage, the measures used to evaluate the model's learning during training were the reconstruction error allied with the pseudo-likelihood.

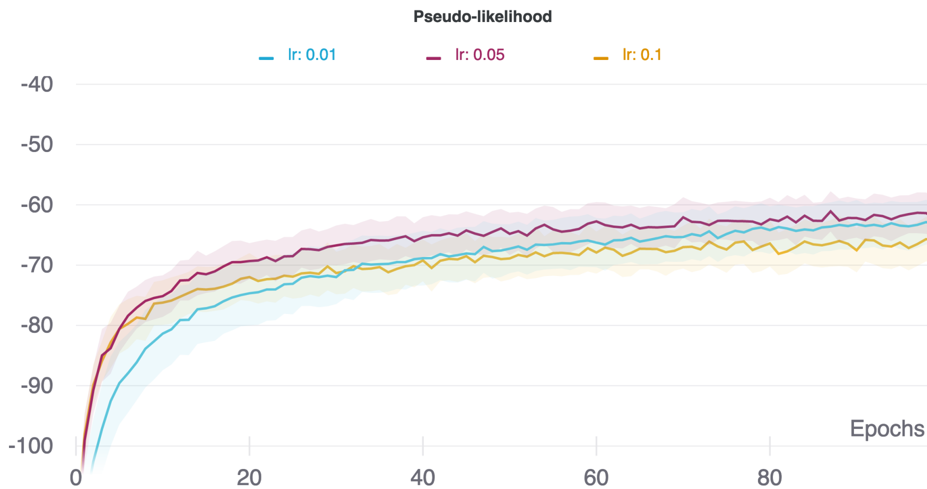


Figure 4.6: Mean pseudo-likelihood during training of RBM models with different learning rates

Figure 4.6 shows the mean value of pseudo-likelihood for all the experiments that were performed with each value for the learning rate that we decided to analyse. The results plotted in this graphic led to the conclusion that the model's architecture, which achieved a higher pseudo-likelihood was trained with learning rate of 0.05. To find the best number of hidden units we have fixed the remaining parameters guided by the best model in Figure 4.6 and plotted the pseudo-likelihood during the training stage, with different number of hidden units.

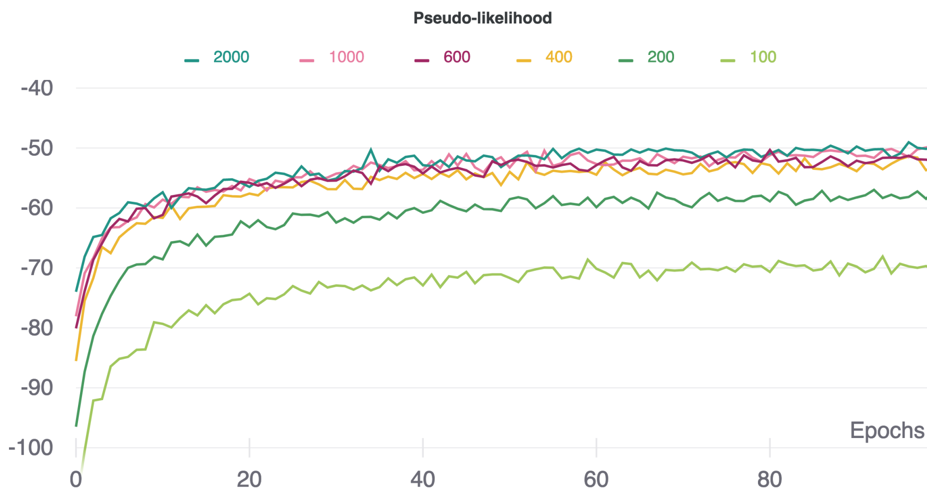


Figure 4.7: Pseudo-likelihood during training of RBM models with different number of hidden units

Figure 4.7 shows that models with fewer hidden units have a lower pseudo-likelihood, which let us conclude that more hidden units increase the expressiveness power of the model. As the pseudo-likelihood of the models with 2000 and 1000 hidden units converges to a similar value and as the computational cost scales with the number of hidden units as well as the risk of overfitting, we decided to show the reconstruction for the RBM with 1000 hidden units.

The sampling phase took place after the model's training. Here, we provide corrupted patterns to the model and perform Gibbs sampling in order to obtain a reconstruction of these patterns. With the purpose of evaluating the reconstruction quality of the chosen model's architecture, we corrupted ten test set MNIST images, flipping the value of twenty random bits in each image, as can be observed in the top images of Figure 4.8. The bottom images correspond to the reconstruction of the corrupted images, which are returned by the model after performing Gibbs sampling.

Figure 4.8: Ten image sample of MNIST test set. Corrupted images (top) are given to the network in order to perform Gibbs sampling and then get the reconstructed images (bottom).

To assess the quality of the reconstructed images, the MSE, briefly explained in section 2.6.2, was used. First, the MSE between the original test patterns and the randomly corrupted ones was calculated, which reached approximately 2.35%. Then, the MSE between the original test patterns and the reconstruction was calculated, which gave approximately 1.59%. Despite the absence of a perfect reconstruction of the original patterns, the MSE between the original patterns and the reconstruction is smaller than between the randomly corrupted patterns and the original test patterns.

With the results of the experiments we performed until now we can trust our implementation of the RBM and start to add the remaining pieces. The final goal is to have a set of models capable of solving the presented problem. In the next sections, the already implemented and tested model was adapted to perform classification.

4.3 Classification using Stochastic Models

Until now we have performed some basic experiments to test the reliability of the model we built from scratch. In this second phase, we started by adapting the RBM model to be able to perform classification, given that the main goal of this thesis is to explore the potentiality of stochastic models to deal with high dimensional sparse inputs.

4.3.1 Restricted Boltzmann Machine

To use the RBM model to perform classification, we followed the architecture described in section 3.4.6.

In this set of experiments, we started by training the model in labelled data, MNIST images combined with ten binary indicator variables, one of which is set to 1, indicating that the image shows a particular digit while the others are set to 0, this process being also known as one hot encoding representation. In order to fully profit from the model's capacities a parameter tuning during training was performed, i.e. adjusting the learning rate, momentum and weight-decay. Choosing the best parameters combination is not an easy task, but by following a practical guide to train Restricted Boltzmann Machines [50], and with some experimental analysis we managed to significantly reduce the reconstruction error.

In the prediction phase two different approaches were implemented and tested. In the first approach, an image was given to the model and the label corresponding to that input image could be obtained by fixing the image neurons and performing N steps of Gibbs Sampling until a reconstruction of the ten visible units corresponding to the class is obtained. The alternative approach consists of calculating the probability of activation for the ten visible units corresponding to the class. To calculate this probability, one needs to multiply the hidden units' probabilities by the weights entries that correspond to the label units and sum the bias of the visible units corresponding to the label. We apply the softmax function to this vector and the label corresponding to that input image is the index of the maximum probability value.

Starting by the experiments with the original MNIST dataset and performing the appropriate train parameter tuning, we concluded that performing more than one step in the negative phase of the training algorithm significantly increases the computation time needed to train the model and does not produce a significant accuracy improvement. Besides, the PCD algorithm, explained in section 3.4.2, showed to perform slightly better than the CD algorithm. Consequently, in the remaining experiments, the algorithm used to train the implemented Stochastic Models is the PCD.

Afterwards, we proceeded to the prediction phase in which experiments with both approaches described above were performed. By using the same model to directly compare both approaches, one can conclude that the second approach showed an accuracy increase of approximately 2%. After multiple insightful experiments, the Figure 4.9 illustrates the best train and test accuracy of the RBM models given an increasing number of hidden units for the original MNIST dataset.

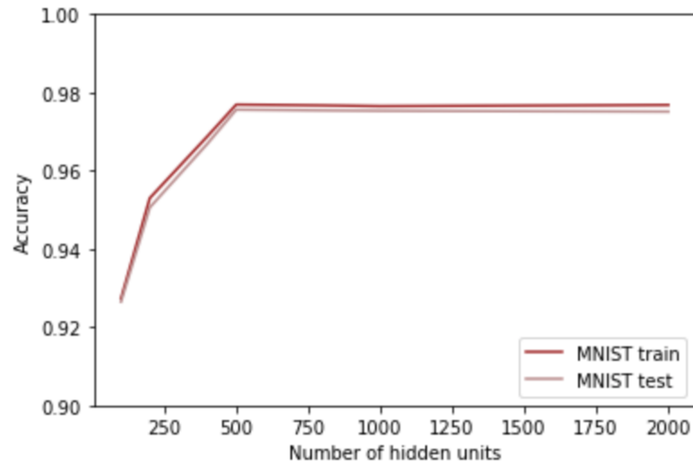


Figure 4.9: Train and test accuracies of RBM on the original MNIST given an increasing number of hidden units.

By analysing the accuracy of the RBM (Figure 4.9), one can conclude that more than 500 hidden units do not bring any advantage in terms of accuracy, and it bears the need for additional time in the learning process. A RBM with 500 hidden units reached a train accuracy of 97.69% and a test accuracy of 97.56%.

With the intent of trying to increase the accuracy, a small change in the previous model was implemented. Before, there were ten binary indicator variables, one of which set to 1 indicating that the image showed a particular digit while the others were set to 0. Now, instead of having ten fixed neurons, the idea was to have a constant N which determines the number of neurons representing each class. With this strategy outlined, first one trains the RBM to model the joint probability distribution of input images and the corresponding labels, though instead of 1 neuron per class one has N as described above.

When it comes to the prediction phase, for each digit image, the class that has more active neurons (neurons set to 1) in the reconstruction is the class predicted to label that input image.

Figure 4.10 shows the best accuracy of the performed experiments with each different values for constant N. Although this idea seemed to have a great potential, the results show that, the higher the number of neurons per class (constant N), the worse the accuracy gets.

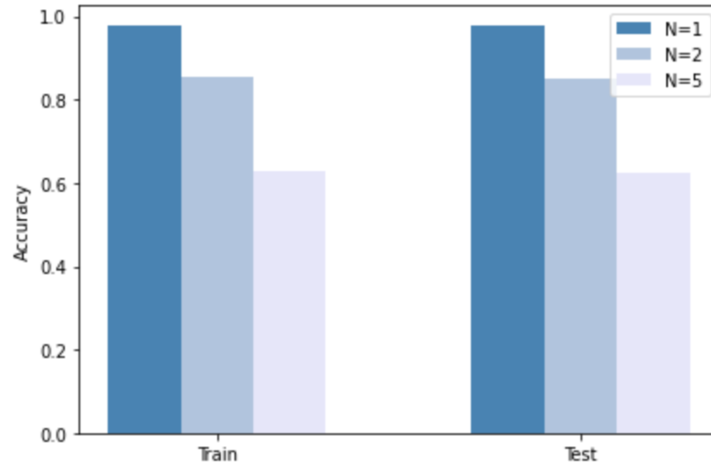


Figure 4.10: Train and test accuracy varying numbers of neurons per class, namely *N*=1, *N*=2 and *N*=5

Guided by the results shown in Figure 4.10, in the following experiments with the RBM model, we will use exclusively 1 neuron representing each class.

If the aim is to show that it is possible to produce a good and general classifier from Stochastic Models, particularly RBM and DBN, first one needs to understand how these models deal with high-dimensional sparse data. Thus, before exploring deeper models we performed an experiment where we compared the performance of a LR and a RBM. The main purpose of this set of experiments was to analyse the behavior of these two models given sparse MNIST datasets generated with increasing sparseness.

To generate the sparse codes for these experiments, we used the strategy described in section 4.1.2. The encoder parameters were defined as *K = 7*, which means we consider the seven most significant features, each defined as a *5x5* window. Parameter *Q* was defined as *Q = 12*, which means that for each *K* there is a new coordinate system plane with size *Q x Q*. Additionally, given the high computational time of these experiments, we started by using a sample of the sparse datasets, 5000 training samples and 1000 test samples. By considering the above information, one has a training set with shape *(5000, 144, 7)* and a test set with shape *(1000, 144, 7)*.

The generated data was reshaped to a two-dimensional array before serving as input to the RBM model, so the final dimensionality of the data was given by *Q x Q x K*, which in this case was 1018. Thereby, the dimensionality of the data indicates the number of visible units of the model.

As discussed above, in section 4.1.2, parameter T_what defines the level of similarity that is needed for a feature in the image to be considered, which means that a higher T_what requires a higher similarity for the feature to be recognized and, consequently, the sparseness of the generated codes increases. Therefore, for these set of experiments we have fixed the remaining parameters and increased T_what, which indirectly means that the sparseness of the data was increased.

In Figure 4.11 the T_what parameter was set to each x-axis value, so that the behaviour of the RBM and LR can be compared when increasing the sparsity of the data, i.e. increasing the level of similarity needed to consider a feature in the image.

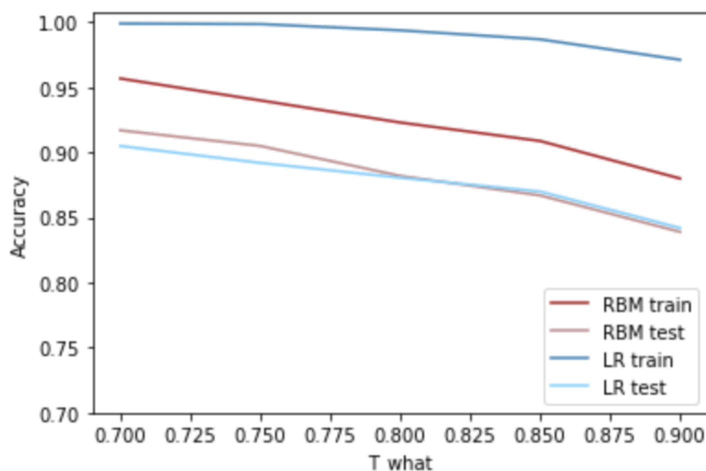


Figure 4.11: Train and test accuracy of LR and RBM given sparse MNIST datasets with increasing sparseness

By analysing the plot in Figure 4.11 one can observe that the RBM shows a smaller gap between train and test accuracies, so it seems to be generalizing the training set better than the LR.

Moreover, as mentioned above, the experiments plotted in Figure 4.11 consider just a sample of the generated sparse datasets. Thus, to understand the behavior of the RBM model and its potentiality to learn a good and general classifier, the dataset with $T_what = 0.85$ was chosen and the same experiment with all the dataset performed. In Table 4.1, in the columns referring to Sparse dataset 1, one has the best accuracy of the RBM and LR, both with this dataset.

In fact, the sparse codes we generated for the experiments plotted in Figure 4.11 had all the same dimensionality. To conclude the experiments that compare these two classifiers, a much higher dimensional and sparser dataset was generated and the same experiments repeated. The encoder parameters were defined as *K = 30*, which means that the thirty most significant features were considered and *Q = 18*, which means that for each *K* there is a new coordinate system plane with size *Q x Q*. By considering these parameters, the dimensionality of the generated sparse dataset is 9720 and the best accuracy results are in Table 4.1, in the columns referring to Sparse dataset 2.

	Sparse dataset 1		Sparse dataset 2	
	Train accuracy	Test accuracy	Train accuracy	Test accuracy
Restricted Boltzmann Machine	92.35%	92.02%	97.35%	96.98%
Logistic Regression	93.62%	92.77%	100%	97.64%

Table 4.1: Train and test accuracy of RBM and LR given two different generated sparse datasets

Once again the best results of the RBM model were with 500 hidden units for both datasets. Concerning the remaining parameters, the learning rate was initialized to 0.01 and decreased during training until reaching 0.001, the momentum used was 0.5 to start the learning process and then increased to 0.9. In these experiments we did not need to use any regularization (Weight-Decay).

By analysing the results in Table 4.1, one can conclude that LR has a slightly higher accuracy than the RBM. By observing the difference between train and test accuracies in Sparse dataset 2, the RBM seems to be better generalizing the rules learned during training than LR. Guided by this conclusion, the Sparse dataset 2 was selected to continue our research described in the following sections.

In what follows, we will introduce the DBN architecture used for classification with the intent to explore whether this model is able to overtake the accuracy reached in Sparse dataset 2 by the models described above.

4.3.2 Deep Belief Network

To implement the DBN, the architecture presented in Figure 4.12 inspired by the paper "Learning multiple layers of representation" [54] was used. Instead of having one RBM, this model consists of two stacked RBMs, which is called a DBN. The first RBM will be trained just on the image neurons which are the high dimensional part of our dataset. Then, the activation of the trained features in the first RBM combined with ten binary indicator variables which represent the class, are treated as input data to train the second RBM. In effect, the features of features can be learned in the second hidden layer.

Afterwards, like in the RBM model, one has the prediction phase where the label corresponding to an input image is obtained by the index of the maximum probability value.

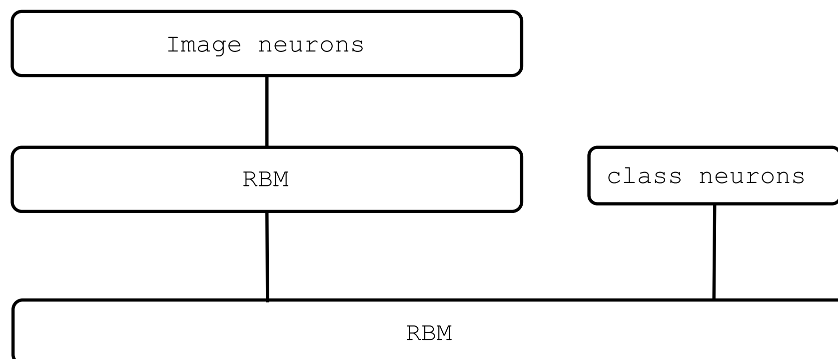


Figure 4.12: DBN with 2 layers that models the joint probability distribution of hidden activations given the input images and the corresponding labels.

As in the experiments with the architecture outlined above, we started by exploring the model with the original MNIST dataset before stepping into the sparse MNIST representations.

The number of hidden units in each layer corresponds to the features of the input images stored in

the model. Thus, it is crucial to find the adequate number of hidden units as well as hidden layers, since models with too few or too many hidden units can result in slow learning and poor performance.

The architecture that resulted in a better accuracy with the MNIST dataset was a first layer RBM with 500 hidden units, a second RBM with 500 hidden units and the last layer with 2000 hidden units. The later RBM receives as input the 500 hidden units that were computed by the second RBM plus the label units. This described architecture was first proposed by Hinton in the previously referred paper [54].

The DBN layers were trained with an initial learning rate of 0.01 and an initial momentum of 0.5, which were decreased and increased, respectively, during the learning process. The accuracy of this model in the original MNIST dataset slightly increased from the simple RBM with an accuracy of 97.74% on the training set and 97.63% on the test set.

The experiments with the original MNIST dataset worked as a baseline to guide the sparse generated dataset experiments. Having in mind that Hinton's architecture [54] performed well with the original MNIST hints at the possibility that it can overpass the accuracy reached by the RBM in the sparse codes.

By considering the same training parameterization used with the original MNIST dataset, a comparison between a 2 layered DBN and a 3 layered DBN was made. Both models start with a first layer of 500 hidden units, then the DBN with 3 layers has another 500 hidden units layer preceding the last layer. Figure 4.13 shows the accuracy of the sparse MNIST dataset with increasing hidden units in the last layer of the models, which is the layer that receives the label neurons as input in both DBNs.

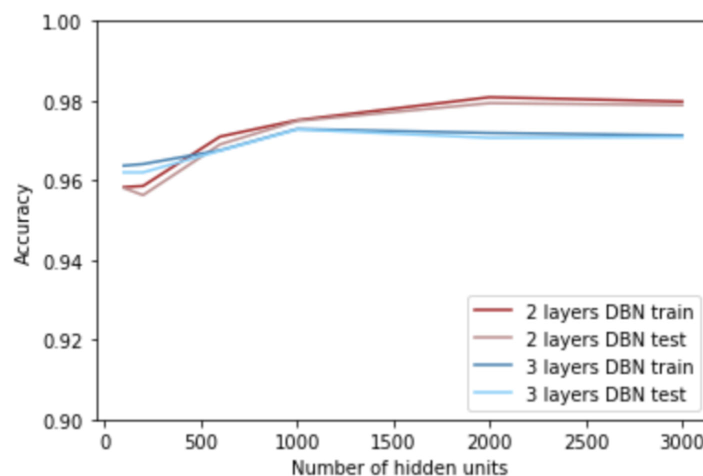


Figure 4.13: Train and test accuracies of sparse MNIST datasets given the increasing number of hidden units in the last layer of the DBN.

By analysing the plot in Figure 4.13 one concludes that, for the sparse MNIST generated dataset, having a DBN with 3 layers does not bring any performance advantage. The maximum accuracy with this architecture was achieved by the DBN with 2 layers, with 2000 hidden units in the second layer.

By considering that we decided to continue our experiments with Sparse dataset 2, Table 4.2 resumes the best accuracy results achieved by the architectures described in this section. In fact, a DBN with 500 and 2000 hidden units in the first and second layers, respectively, surpasses the test accuracy achieved by the LR with a train and test accuracy of 98.09% and 97.94%, respectively. Besides having a better accuracy, we continue to have close train and test accuracies which indicates that it has a high generalization power, which means that the rules learned during training are equally valid to the test set.

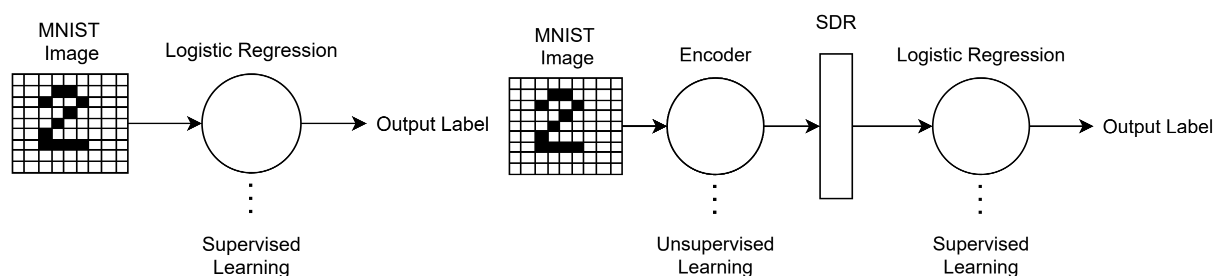
	Train accuracy	Test accuracy
Logistic Regression	100%	97.64%
Deep Belief Network with 2 layers	98.09%	97.94%
Deep Belief Network with 3 layers	97.62%	97.58%

Table 4.2: Train and test accuracy of DBN with 2 layers, DBN with 3 layers and LR given the Sparse datasets 2.

4.3.3 Restricted Boltzmann Machine followed by Logistic Regression

In a second stage of experiments, we tried a different approach. Instead of using a RBM or a DBN to model the joint probability distribution of input images and the corresponding labels, we used them to model the input images into activation of the hidden units. In this new approach the classification is not performed by the Stochastic Models. They have the role of providing the classifier, which in this case is a LR, with a compact representation of the input.

Prior to the implementation of this model, we went back to the baseline model, a LR trained on the MNIST dataset (Figure 4.14(a)) and on the generated sparse codes of MNIST dataset (Figure 4.14(b)). In these baseline experiments we concluded that the LR trained directly on the original MNIST dataset (dense dataset) has a training accuracy of 93.9% and a test accuracy of 92.5%. When it comes to the generated sparse codes (Sparse dataset 2) and as already presented in the previous sections, the LR has a training accuracy of 100% and a test accuracy of 97,64%.



(a) Logistic regression applied to MNIST dataset.

(b) Logistic regression applied to sparse MNIST dataset.

Figure 4.14: Scheme of baseline models.

The main idea behind the approach represented in Figure 4.15 is to give the images as input to the RBM. Then, after the model is trained, it computes the activations of the hidden units, which give a compact representation of the input. That compact generated representation is then used as input to the LR. In this way, instead of giving a high-dimensional sparse vector to the LR, a compact representation generated by the hidden units of the RBM is provided.

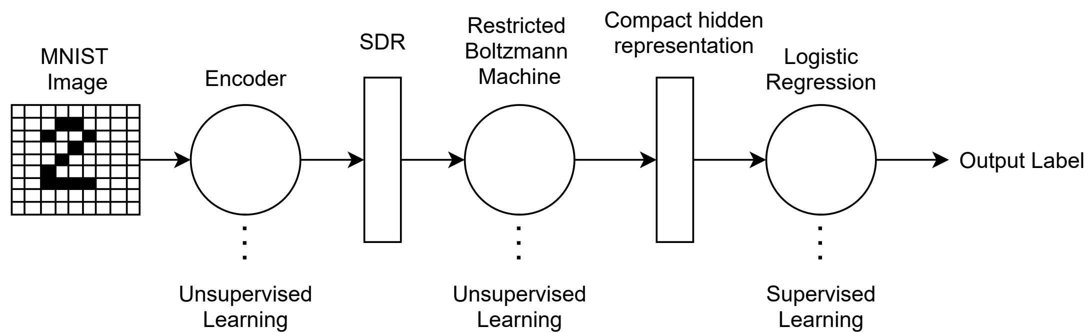


Figure 4.15: Scheme of the classifier composed by a RBM followed by a LR

By using the described pipeline to perform several experiments, we were able to take meaningful conclusions about this approach. Once more, these experiments were performed with an increasing number of hidden units of the RBM to explore the model architecture that could result in a higher accuracy.

By analysing the plot in Figure 4.16 one can conclude that the pipeline suggested in Figure 4.15 is the model that better classifies the original MNIST dataset with a 99.64% accuracy on the training set and 98.48% on the test set.

Regarding the sparse MNIST dataset, the difference between the train and test accuracy slightly decreases compared with the LR applied directly on the sparse dataset. The accuracy with a 5000 hidden unit RBM followed by a LR is 100% on the training set and 97.98% on the test set.

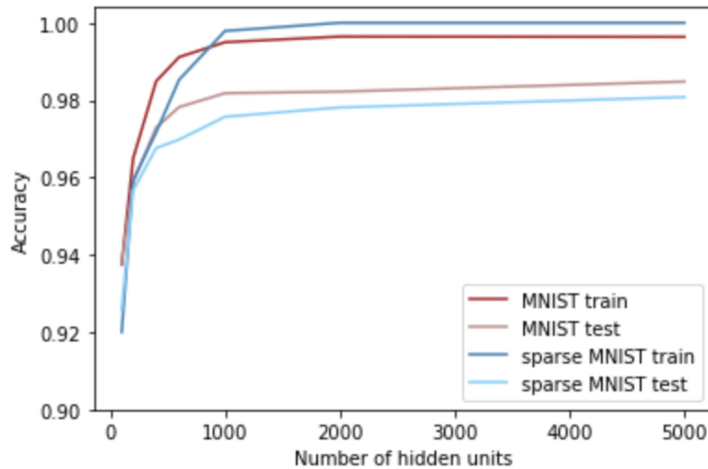


Figure 4.16: Train and test accuracies of original MNIST and sparse MNIST datasets given the increasing number of hidden units.

In fact, by observing the results in Table 4.3, one can conclude that the LR classifier does not produce significantly better results when receiving a hidden representation of the RBM instead of the sparse codes. In addition, the large number of hidden units extremely increases the time spent in the learning process and it does not reduce significantly the difference between train and test accuracies.

	Train accuracy	Test accuracy
Logistic Regression	100%	97.64%
RBM followed by Logistic Regression	100%	97.98%

Table 4.3: Train and test accuracy of LR and RBM followed by LR models considering the same sparse dataset (Sparse dataset 2)

The stochastic models that were used before as classifiers, now have the role of reducing the input's dimensionality. Consequently, the LR, instead of receiving a high-dimensional sparse input, it receives a compact representation given by the Stochastic Model's hidden units. In this section, we concluded that the RBM does not provide the gradual dimensionality reduction we needed to decrease the difference between train and test accuracies. Thus, in the following section, the use of the DBN model to obtain a gradual dimensionality reduction was explored.

4.3.4 Deep Belief Network followed by Logistic Regression

Instead of having just one hidden layer to get the compact hidden representation, one can have a gradual dimensionality reduction. This is possible by stacking more than one RBM, which is also known as a DBN. For these experiments the pipeline shown in Figure 4.17 was used.

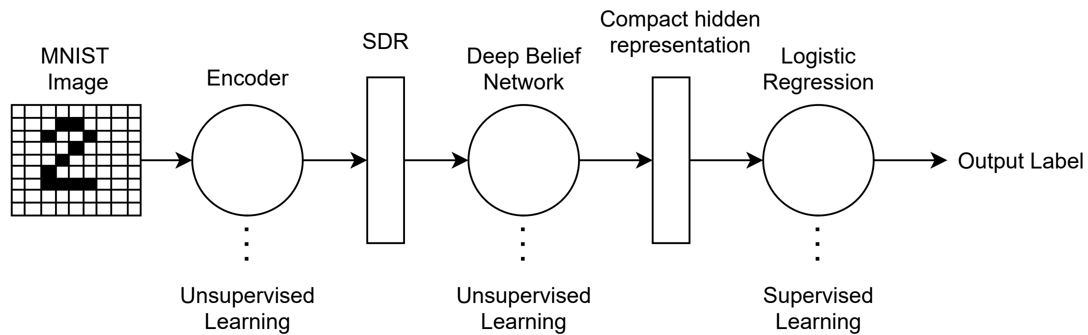


Figure 4.17: Scheme of the classifier composed by a DBN followed by a LR.

Various experiments were carried out with this architecture to understand if the dimensionality reduction from layer to layer could increases even more the accuracy when classifying the sparse codes. In fact, after analysing all the results obtained, the best performance was achieved with a DBN with 3 layers, in which the first layer has 2000 hidden units, the second 1000 and the last 500. The train and test accuracies with this DBN architecture were 98.32% and 97.36%, respectively, as shown in Table 4.4.

	Train accuracy	Test accuracy
Logistic Regression	100%	97.64%
DBN followed by Logistic Regression	98.32%	97.36%

Table 4.4: Train and test accuracy of LR and DBN followed by LR models considering the same sparse dataset (Sparse dataset 2)

Despite the dimensionality reduction given by the decreasing number of hidden units from one layer to the next, this model was not the one that resulted in a better accuracy.

In fact, it is noteworthy that the difference between train and test accuracies was reduced. By providing a compact representation as input to the LR, the generalization capability of the LR increased. However, the test accuracy slightly decreased which means that the hidden compact representation of the input does not perfectly represent the high-dimensional sparse input.

Nevertheless, before concluding the research on the classifiers inspired by Stochastic Models, in the next section is suggested a variation of the RBM model, which seems to be more biologically plausible. This variation of the RBM model is compared with the one described in section 4.3.1 and assessed if it brings any advantage when classifying the generated MNIST sparse codes.

4.3.5 Restricted Boltzmann Machine with 3 state neurons

RBMs are biologically plausible models. Grounded by this premise, a new implementation of the RBM based on the brain inspired idea of 3 state neurons RBM is suggested. This idea is described in section

3.4.7 and was first mentioned in the article “Restricted Boltzmann Machines for Collaborative Filtering” [2].

Before diving into the model description, one needs to clarify the idea of 3 state neurons, according to which each neuron can be in one of the following 3 states: the excitatory state which means that the neurons are stimulated to be active, the inhibitory state, which correspond to neurons that are stimulated to be inactive and the non-stimulated neurons, which are neither active nor inactive.

By considering once more the sparse codes, one has *N* samples, a new coordinate system plane of *QxQ* and *K* features. Consequently, the dataset follows the shape: *(N, QxQ, K)*.

In the previous implementation of the RBM model, these original dimensions were reshaped and thereby the model received the data with shape *(N, QxQxK)*. This means that for each example *N* we had *QxQxK* visible units, which connected to *H* hidden units. The *QxQxK* visible units had two possible states: they were either active or inactive, assuming the values 1 or 0, respectively.

With 3 state neurons, every RBM has the same number of hidden units, but a RBM only has visible softmax units for the positions of the new coordinate system plane where one of the *K* features is present. In each sample of the dataset, the number of *QxQ* positions that have a feature can vary, which means there is no fixed number of visible units as in the previous models. Thus, a RBM has few connections if that new coordinate system plane has few features. The RBM model, however, needs a fixed number of visible and hidden units to be trained. The solution proposed towards this problem is the implementation of a single example RBM. Each RBM only has a single training case, but all the corresponding weights and biases are tied together. Then if two samples of the dataset have a feature in the same position of the new coordinate system plane, their two RBMs must use the same weights between the softmax visible unit and the hidden units.

Figure 3.7 in section 3.4.7 illustrates a prototype of the idea we just described. In this RBM there is *V* visible connection that correspond to the *QxQ* positions in which a feature *K* is present. In the case where there is no feature present, then, that input position is considered as no information and no connection to the hidden units is established. In addition to the symmetric weights between each hidden unit and each of the *K* values of a softmax unit, there are *K* biases for each softmax unit and one for each hidden unit.

Revisiting the core idea, in this implementation the visible neurons can be described by having 3 states. In the positions in which a feature is present, there is a connection in the RBM (Figure 3.7). For the *k* neurons associated with that connection, the unit corresponding to the presence of a feature is set to 1, which can be linked with the excitatory state of a neuron. The remaining *k-1* units are set to 0, which is associated with the inhibitory state of the *k-1* neurons. The positions that have the presence of a feature can be associated with the winner-take-all computational principle. In each one of the *QxQ* positions that has a feature, then the feature that is present wins the activation and the remaining *k-1*

features shut down. In the *QxQ* positions in which no feature is present, there are no connections with the hidden units and therefore no weights are learned. The units that have no connections in the RBM are treated as non stimulated neurons.

In order to perform experiments with the implementation of a 3 state neurons RBM, a sparse dataset is needed. First, we started by using a sample of Sparse dataset 1 generated with the parameters described in section 4.3.1. Recalling the used parameters, one has *K = 7*, which means that the seven most significant features were considered and *Q = 12*, entailing a new coordinate system plane with size *12 x 12*. Also, the features window size is *5 x 5*. The final shape of the dataset is *(N,144,7)*, in which *N* describes the number of samples.

Actually, each single example RBM, between the 144 positions (*Q x Q*), only considers the connections with the positions that have a feature. Since this dataset is highly sparse, we will have few positions with the presence of a feature, which results in *N* single example RBMs with few connections.

In fact, these *N* single example RBMs will contribute to a shared weight matrix and biases, in which for each one of the *N* examples, the gradient will only update the indexes in which that specific example has the presence of a feature. Additionally, all the class connections are present as in the previous model.

The prediction phase consists in calculating the probability of activation for the ten visible units corresponding to the label. To calculate this probability, one needs the hidden units' probabilities, the weights, and the bias of the visible units. The calculation of the hidden units' probabilities in this network implementation is not trivial. Considering that each new instance has different number of features present in different positions, first the positions where the new instance has a feature need to be found. Then, the hidden units' probabilities will exclusively consider the entries of weight matrix for those positions. Finally, to calculate the net input vector for the label units one multiplies the hidden units' probabilities by the weights of the label units and sum the bias of the visible units corresponding to the label. To this vector a softmax function is applied and the label corresponding to that input image is the index of the maximum probability value.

By using a sample of the dataset described, with 5000 training examples and 1000 test examples, we started by training the model with an initial learning rate of 0.01 and a momentum of 0.9. During the learning process we decreased the learning rate to 0.005 and increased the weight decay from 0 to 0.001.

In the prediction phase, a training accuracy of 87.32% and a test accuracy of 80.07% were obtained. By performing this same experiment with all the dataset and comparing with the first RBM implementation the results shown in Table 4.5 were achieved. These accuracies were reached after an exhaustive parameter tuning, during which we concluded that the optimal number of hidden units for this sparse dataset is 100.

	Train accuracy	Test accuracy
2 state neurons RBM	92.35%	92.02%
3 state neurons RBM	86.31%	85.475%

Table 4.5: Train and test accuracy comparison between the original RBM implementation (2 state neurons) and the 3 state neurons implementation given the same sparse dataset

By analysing the results presented in Table 4.5 one can conclude that, for the generated dataset, the 2 state neurons RBM implementation described in section 4.3.1 has a better accuracy than the new proposed implementation.

In fact, the idea behind the 3 state neurons RBM implementation, explained in 3.4.7, is suggested by the article "Restricted Boltzmann Machines for Collaborative Filtering" [2] as a strategy to deal with missing values, i.e. entries of the dataset where information is missing.

By applying this idea to the generated MNIST sparse codes, the *QxQ* positions where no feature was present were treated as missing information by the model, i.e. no connection between those visible units and the hidden units were considered. However, these positions do not represent missing information but the absence of a feature in a certain position. Thus, the results of the experiments performed using the 3 state neurons RBM, let one concludes that considering exclusively the connections with the dimensions in which a feature is present discards relevant information that the model needs to accurately learn the sparse dataset.

For the RBM to represent the correlations between active features on the hidden units, the connections between inactive features have to be considered. Otherwise, the RBM exclusively receives as input units where a feature is present and it is not capable of accurately learn the correlations between them.

By taking into consideration this conclusion, the 2 state neurons RBM classifier described in section 4.3.1 was used in the next sections.

4.4 Models comparison

Throughout the former sections, the implementation of several Stochastic Models was described as well as the results derived from the study of their behaviour when dealing with the high-dimensional sparse dataset generated from MNIST. To make the final remarks about the performance of these models given the proposed dataset, in Table 4.6 the best performance results of each studied model are presented.

	Train accuracy	Test accuracy
Logistic Regression	100%	97.64%
Restricted Boltzmann Machine (RBM)	97.35%	96.98%
Deep Belief Network (DBN)	98.09%	97.94%
RBM followed by Logistic Regression	100%	97.98%
DBN followed by Logistic Regression	98.32%	97.36%

Table 4.6: Train and test accuracy of all the models implemented considering the same sparse dataset (Sparse dataset 2)

All the models presented in the table show to perform well given a high-dimensional sparse dataset generated from MNIST. The model which showed a better test performance was the RBM followed by a LR, however, the DBN with 2 hidden layers achieved almost the same accuracy.

One of the advantages of using the RBM and DBN is the fact that besides classifiers, they are generative models. Consequently, additionally to predicting the labels of the dataset, they can perform image reconstruction as described in section 4.2, whereas, in the other models the classification is done by a LR, which is exclusively a classifier. Besides, these classifiers show a smaller difference between train and test accuracies, which means that they have a greater generalization performance when classifying the high-dimensional sparse dataset generated from the original MNIST.

Actually, the LR scores 100% on the training set, which suggests that this model is highly adapted to the training data and consequently more prone to overfitting. However, the good results archived by the LR classifier in the test set leave us wondering whether or not these generated codes are high-dimensional and sparse. It is plausible to think that the generated sparse data lies on a low-dimensional manifold embedded in a higher-dimensional space. As a result, the learning problem becomes too easy and the LR can accurately classify the test set. To have a well-grounded research one cannot be restricted to the generated MNIST sparse codes. In the following section, a deeper research with a different way to generate sparse data is presented.

4.5 Learn from a Sparse Normal Distributed Dataset

Revisiting the problem raised in the former section, it is plausible to consider that the generated sparse MNIST data lies on a low-dimensional manifold embedded in a higher-dimensional space, which means that, although the data has many features, it only has a few degrees of freedom.

With this idea in mind, and with the desire to understand if the RBM model accurately classifies high-dimensional sparse data while LR falls into overfitting, several experiments were carried out in which both a LR and a RBM had to perform the same classification task.

In fact, generating binary sparse data is not a trivial task as it is hard to find a complexity balance in the learning problem. Thus, instead of generating binary data, we decided to generate a dataset where

each class follows a multivariate normal distribution.

The implementation of the RBM we used in the experiments is modelled with Bernoulli visible and hidden units, which means that this RBM is prepared to receive input data in the range [0,1]. By considering that the generated data is real-valued, some exploratory experiments with a Gaussian-Bernoulli RBM were performed, but tuning the value of the Standard Deviation parameter is a hard task, which can produce an unstable learning process [50].

Since the expected results using Gaussian visible units were not achieved, the Bernoulli-Bernoulli RBM was explored to address this problem. In fact, the only difference between using Bernoulli or Gaussian visible units occurs when sampling the visible units of the negative phase of the learning algorithm. Constraining the values of the visible units to be between 0 and 1 imparts a kind of regularization to the learning process. In the sampling phase, the use of Bernoulli visible units is necessary as the sampled label is binary.

In the next steps, the dataset generation pipeline and the experiments performed are explained, with the aim of reaching meaningful conclusions on the behaviour of LR with sparse data, compared with the Stochastic Models; in this case, only the RBM model was addressed.

4.5.1 Dataset generation

To reach the desired results, we start by explaining the dataset generation. Each dataset is generated with two classes, in which each one follows a Gaussian distribution. The first half of the samples belong to class 0 and follow a Gaussian distribution with mean centered in the origin, the other half corresponds to class 1 and follows another Gaussian distribution with mean centered in five. The covariance matrix for each class is defined as a diagonal matrix, in which the diagonal values are set to the norm of the difference between the mean vector of each class multiplied by a small number as to reach a good balance in the problem complexity.

Moreover, the number of features will be further defined for each experiment. In the case the dataset to be is intended to be dense, the number of features is set to a low value, whereas in the case the dataset is intended to be high-dimensional, the number of features is fixed to a high value.

4.5.2 Pipeline

With the objective of structuring the steps performed by the experiments, a simple pipeline is described. Thus, for each experiment, we started by setting the parameters and then running 10 times the following pipeline:

1. Populating the dataset by sampling from the two multivariate normal distribution with the previously defined parameters and associate each multivariate normal distribution to a class, either 0 or 1.

2. Centring the data, which consists in subtracting the mean of each feature to every value of that feature.

3. Transforming the dataset into sparse data, which means choosing a few random features to keep in each sample and set the remaining features to zero.

4. Dividing the samples of the dataset into train and test. For this step the function $train_test_split()$ from the sklearn library was used, in which the input parameter $test_size$ was set to 20%.

5. Training a LR model with the generated train set.

6. Evaluating a LR model by computing and storing the train and test accuracies.

7. Training the RBM with the generated train set using PCD algorithm.

8. Evaluating the RBM by performing the Gibbs sampling to get the reconstruction of the class unit for both train and test sets (explained with more detail in section 3.4.6). After having all the reconstructions, the model's train and test accuracies can be computed and stored.

After running the described pipeline, four lists with 10 train and test accuracy values for both models were obtained. Subsequently, the mean and the standard deviation for each list was calculated. In the end, a single train accuracy for both models and a single test accuracy for both models were stored, as well as the respective standard deviations.

4.5.3 Experimental Analysis

This research was instigated by the desire to understand if Stochastic Models perform better than classic Machine Learning models, like LR, when classifying high-dimensional sparse data. For that reason, and with the aim of having a baseline experiment which guided the next steps, a comparison was made with the LR performance classifying a non-sparse dense dataset and a high-dimensional sparse dataset.

Starting by the dense dataset, it was generated as described in section 4.5.1. For the dataset to be dense, the number of features was fixed to 500 and all the values of the data were kept. When generating the high-dimensional sparse dataset, the methodology described in the section 4.5.1 was also used. In this case, the number of features was set to 5000. Furthermore, the sparsity was fixed to 95%, which means that for each sample 5% of the features were kept and the remaining values set to 0.

In Figure 4.18, the results show that LR performs well when the dataset is dense, with a mean train accuracy of 100% and a mean test accuracy of 99.25%. However, when analysing its performance on the high-dimensional sparse dataset a huge overfitting is observed, with a mean train accuracy of 100% and a mean test accuracy of 63.5%.

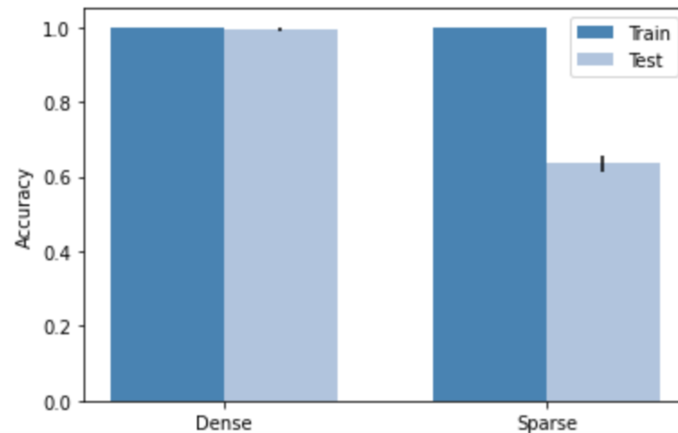


Figure 4.18: Performance of the LR in a dense versus a high-dimensional sparse dataset.

This first experiment provides a baseline to guide the next steps. In what follows, the intend was to show that the RBM performs accurately in a classification task with high-dimensional sparse data. Before diving into the experiments, the parameters must be defined. With these experiments, the aim was to access the behaviour of a LR and a RBM with increasing sparseness of the dataset. For this reason, the remaining parameters of both models were fixed to the same values, as to achieve a trustful comparison between models.

The number of samples was fixed to 2000 and the dimensionality of the input to 5000. As far as the parameters of the RBM architecture were concerned, the number of hidden units was set to 500. Additionally, a batch size of 50 and a learning rate of 0.1 was used.

With the final objective of taking meaningful conclusions about both models when the dataset sparsity increases, i.e., the number of zero values increases, the pipeline described in section 4.5.2 was followed. To make an easier comparison between models, the sparsity value was set to each x-axis value and the mean accuracies of LR and RBM were plotted in Figure 4.19.

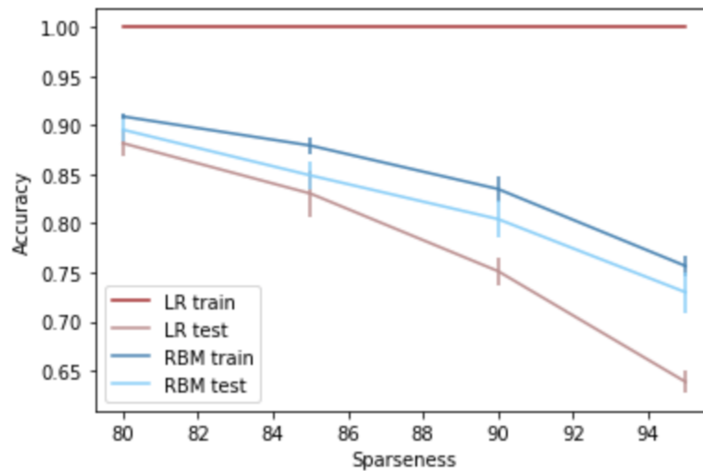


Figure 4.19: Comparison between accuracies of LR and RBM classifiers with increasing percentage of sparseness.

By analysing the results plotted in Figure 4.19, one can observe that the LR classifier has an accuracy of 100% on the training set, though it is not able to perform accurately on the test set, which suggests that this model is learning the noise in the training data. As the data gets sparser the learning problem becomes harder and test set accuracy decreases. This means LR can represent the training set of sparse data perfectly but unable to generalize, which results in a poor performance in the test set.

On the contrary, the RBM classifier can generalize the learning problem. Although, with the increasing sparseness the model's performance decreases, it never falls into overfitting as LR does. When the data is generated with 95% sparseness, the LR has a mean test accuracy of 63.5%, while the RBM shows a mean train and test accuracies of 75.67% and 73.15%, respectively. So, comparing the test set performance, the RBM is nearly 10% more accurate than LR.

To understand how both models behave with different dimensionalities and sparseness, both models varying these two parameters were run. The heatmap in Figure 4.20, shows the difference between train and test accuracies of RBM and LR considering the y-axis as the dimensionality values and the x-axis as the sparseness percentage used with each different dimensionality dataset. In the heatmap on the right, one can observe that as the data gets sparser the difference between both test accuracies increases and the RBM shows to perform better than the LR. With lower sparseness the RBM and the LR have similar test results. The dimensionality behaves in the same way, which means that with higher dimensional data it is easier to notice the better results achieved by the RBM. The heatmap on the left shows the train accuracy difference, which is always lower than 0. The training accuracy of the LR is always 100%, so darker colours means a higher difference between the training accuracies. With less sparseness the train accuracy of the RBM increases as shown in Figure 4.19.

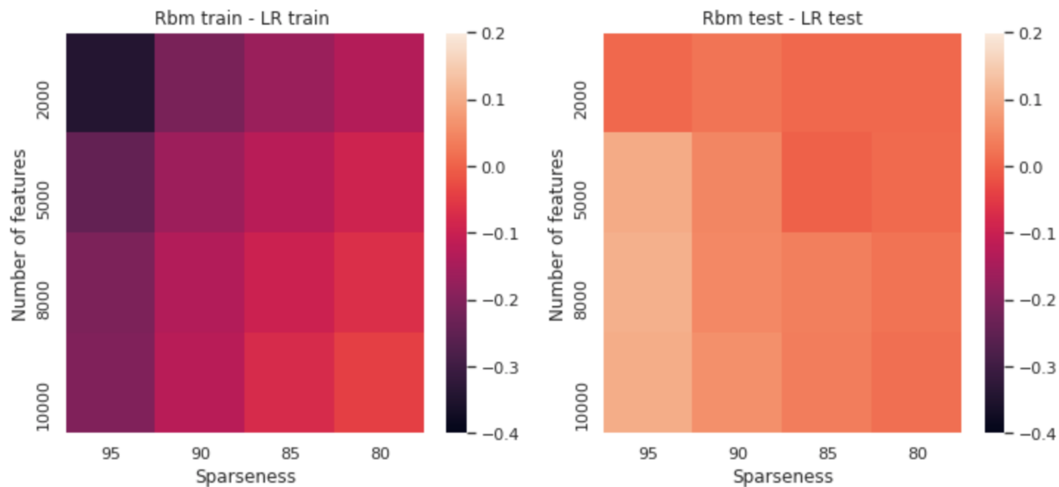


Figure 4.20: The left heatmap shows the difference between train accuracies of RBM and LR considering increasing dimensionality and sparseness. The right heatmap shows the difference between test accuracies of RBM and LR considering increasing dimensionality and sparseness percentage.

The good results and generalization performance achieved by the RBM classifier can be mainly justified by the fact that it has a hidden layer that represents hidden correlations between active features of sparse vectors. Therefore, this model can map a high-dimensional sparse vector into a lower dimensional hidden layer, which would catch the relevant features present on the high-dimensional sparse vector.

Besides, the LR learns the conditional probability of the class given the features, while the RBM learns the joint probability of the features and class. The difference between learning the conditional or the joint probability may be a factor influencing the performance of each model. The LR is learning a simpler problem than the RBM. Thus, given its huge capacity, instead of learning just the training patterns, the model is also learning the noise. Consequently, the LR becomes too adapted to the training set, which leads to the overfitting problem.

This justification seems to be well grounded, although one may still wonder: Can the good performance of the RBM be justified by the presence of hidden units, which makes it a non-linear classifier?

To answer this question, a comparison was made between the performances of the RBM and a Multi-Layer Perceptron (MLP). To derive meaningful conclusions, a RBM and a MLP with the same number of hidden units were defined. Additionally, the MLP activation function for the hidden layer units used was the logistic sigmoid function.

More experiments were carried out, in which normally distributed datasets were created following the pipeline described in section 4.5.2. However, instead of comparing the RBM to a LR, the comparison was made with a MLP. In this experiment, the number of samples was fixed to 2000, the dimensionality of the input to 5000 and the number of hidden units of the RBM and the MLP is fixed to 500. With these parameters and a sparsity of 95%, the results in Table 4.7 were obtained.

	Mean train accuracy	Mean test accuracy
Multi-Layer Perceptron (MLP)	100%	64.12%
Restricted Boltzmann Machine (RBM)	76.18%	73.98%

Table 4.7: Train and test mean accuracies of MLP and RBM given the generated data.

By analysing the results, the MLP classifier has an accuracy of 100% on the training set, although it is not able to perform accurately on the test set. This means that the model is overfitting the training data, and so, it lacks generalization capability.

As a matter of fact, the overfitting undergone by the MLP model led to the conclusion that, the generalization capability of the RBM is not a consequence of being a non-linear classifier.

Consequently, the good generalization performance of the RBM is justified by the hidden neurons that represent correlations exclusively between active features of the sparse vectors. The model receives a high-dimensional sparse vector and, by capturing the relevant features in a much lower dimensional hidden layer, it can perform classification without falling into overfitting.

After the research performed throughout this chapter, the question raised in section 1.2 can, finally, be answered. Actually, our intuition pointed in the right direction and, thus, one can conclude that the main reason for the good generalization performance of the RBM is the fact that hidden neurons learn correlations between active features of the high-dimensional sparse vectors.

5

Conclusion

SDRs are the fundamental form of representing information in the brain. The activity of any population of neurons in the neocortex is sparse, where a low percentage of neurons are highly active, and the remaining neurons are inactive [5]. Previous research explored these representations with biologically plausible models to perform associative memory tasks. To learn a good and general classifier without running into the “curse of dimensionality” problem, however, is a hard task. Deep learning models progressively reduce the dimensionality of the SDR from layer to layer and have some success in tasks in which there is a great amount of data with labels, although they use a non-biologically plausible algorithm.

The present thesis explores the capabilities of classifiers inspired in Stochastic Models to side step the limitations that classic Machine Leaning models have, when classifying high dimensional sparse data.

The main evidence that motivated the use of Stochastic Models is the fact that hidden units in these models represent hidden correlations between active neurons of sparse vectors. As sparse vectors have few active neurons, then stochastic models can map a high-dimensional sparse vector into a hidden layer with few hidden units, which represents the visible correlations between present dimensions of the high-dimensional sparse vector.

With this motivation in mind, we started by explaining the main concepts associated with SDRs, Associative Memories, MCMC methods, namely Metropolis algorithm and Gibbs Sampling, which set the ground to understand the Stochastic Models presented in chapter 3, and finally the optimization algorithm called Simulated Annealing.

To solve the problem that instigated this research work, in chapter 3 we described in detail the main characteristics of the Stochastic Methods. This chapter gathers all the theoretical knowledge needed to understand the models implemented and tested in the experiments, described in chapter 4.

During the experimental analysis, we started by using the strategy structured in [1] to generate the sparse codes. With these codes, we tested the implemented classifiers inspired in Stochastic Models and compared their accuracy results with a simple Logistic Regression. Both the Stochastic Models and the LR achieved good results when classifying these codes. In fact, these good results archived by the LR classifier left us wondering if the generated codes were effectively high-dimensional and sparse. Actually, it is plausible to think that the generated MNIST sparse data reside on a low-dimensional manifold embedded in a higher-dimensional space. This suggests that the real dimensionality of the data is highly inferior to the defined number of features of the dataset, which can justify the good performance of LR.

With the desire to study the performance of a RBM and compare it with a LR in high-dimensional sparse data, we defined a different dataset generation strategy, in which each class followed a multivariate normal distribution. To control the sparseness of the data a pre-defined number of features were

randomly deleted from each sample. We performed several experiments using this high-dimensional sparse data, in which datasets with varying dimensionalities and sparseness were generated. The good generalization capability achieved by the RBM showed that this model can map a high-dimensional sparse vector into a hidden layer, which catches the relevant features present on the high-dimensional sparse vector, while the LR, with the increasing dimensionality and sparseness, becomes too adapted to the training set, which leads to the overfitting problem.

To understand if the good generalization performance of the RBM is really justified by its capability to map a high-dimensional sparse vector into few hidden units, which capture the relevant features present in the data, or if it results from being a non-linear classifier, a comparison was made between the RBM and the MLP with the same number of hidden units. Considering the poor generalization performance obtained by the MLP when classifying the high-dimensional sparse data, one can conclude that the main reason for the good performance of the RBM is the ability to capture correlations between active features of the high-dimensional sparse vectors.

Thus, one can conclude that the motivation for this research work pointed in the right direction. The results achieved by the implemented Stochastic Models demonstrate that by learning the correlations between active features of the sparse input data, good results can be achieved by side stepping the overfitting problem, which affects classic Machine Learning models.

Bibliography

[1] L. Sa-Couto and A. Wichert, “Storing object-dependent sparse codes in a willshaw associative network,” *Neural Computation*, vol. 32, no. 1, pp. 136–152, 2020.

[2] R. Salakhutdinov, A. Mnih, and G. Hinton, “Restricted boltzmann machines for collaborative filtering,” in *Proceedings of the 24th international conference on Machine learning*, 2007, pp. 791–798.

[3] S. Ahmad and J. Hawkins, “Properties of sparse distributed representations and their application to hierarchical temporal memory,” *arXiv preprint arXiv:1503.07469*, 2015.

[4] G. E. Hinton, “Distributed representations,” 1984.

[5] J. Hawkins, S. Ahmad, S. Purdy, and A. Lavin, “Biological and machine intelligence (bami),” 2016, initial online release 0.4.

[6] J. A. Hertz, *Introduction to the theory of neural computation*. CRC Press, 2018.

[7] G. Palm, “Chapter xii how useful are associative memories?” in *North-Holland Mathematics Studies*. Elsevier, 1982, vol. 58, pp. 145–153.

[8] B. Ouyang, Y. Li, Y. Song, F. Wu, H. Yu, Y. Wang, M. Bauchy, and G. Sant, “Learning from sparse datasets: Predicting concrete’s strength by machine learning,” *arXiv preprint arXiv:2004.14407*, 2020.

[9] J. Bissmark and O. Wärnling, “The sparse data problem within classification algorithms: The effect of sparse data on the naïve bayes algorithm,” 2017.

[10] M. Tan, L. Wang, and I. W. Tsang, “Learning sparse svm for feature selection on very high dimensional datasets,” in *ICML*, 2010.

[11] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1, no. 2.

[12] X. Li, “Classification with large sparse datasets: Convergence analysis and scalable algorithms,” 2017.

[13] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from data*. AMLBook New York, NY, USA:, 2012, vol. 4.

[14] G. E. Hinton, "Boltzmann machine," *Scholarpedia*, vol. 2, no. 5, p. 1668, 2007.

[15] B. A. Olshausen and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images," *Nature*, vol. 381, no. 6583, pp. 607–609, 1996.

[16] Z. Zhang, Y. Xu, J. Yang, X. Li, and D. Zhang, "A survey of sparse representation: algorithms and applications," *IEEE access*, vol. 3, pp. 490–530, 2015.

[17] G. Palm, "Neural associative memories and sparse coding," *Neural Networks*, vol. 37, pp. 165–171, 2013.

[18] G. A. Carpenter, "Neural network models for pattern recognition and associative memory," *Neural networks*, vol. 2, no. 4, pp. 243–257, 1989.

[19] G. Palm, "On associative memory," *Biological cybernetics*, vol. 36, no. 1, pp. 19–31, 1980.

[20] T. Kohonen, *Self-organization and associative memory*. Springer Science & Business Media, 2012, vol. 8.

[21] A. Fischer and C. Igel, "Training restricted boltzmann machines: An introduction," *Pattern Recognition*, vol. 47, no. 1, pp. 25–39, 2014.

[22] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The american statistician*, vol. 49, no. 4, pp. 327–335, 1995.

[23] S. S. Haykin *et al.*, "Neural networks and learning machines/simon haykin." 2009.

[24] A. M. Wichert, *Principles Of Quantum Artificial Intelligence: Quantum Problem Solving And Machine Learning*. World Scientific, 2020.

[25] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 721–741, 1984.

[26] C. M. Carlo, "Markov chain monte carlo and gibbs sampling," *Lecture notes for EEB*, vol. 581, 2004.

[27] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[28] A. Das and B. K. Chakrabarti, *Quantum annealing and related optimization methods*. Springer Science & Business Media, 2005, vol. 679.

[29] T. Tieleman, "Training restricted boltzmann machines using approximations to the likelihood gradient," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 1064–1071.

[30] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," *Proceedings of the national academy of sciences*, vol. 81, no. 10, pp. 3088–3092, 1984.

[31] G. Joya, M. Atencia, and F. Sandoval, "Hopfield neural networks for optimization: study of the different dynamics," *Neurocomputing*, vol. 43, no. 1-4, pp. 219–237, 2002.

[32] J. Šíma and P. Orponen, "Continuous-time symmetric hopfield nets are computationally universal," *Neural Computation*, vol. 15, no. 3, pp. 693–733, 2003.

[33] S. Seung, "The hopfield model," *Introduction to Computational Neuroscience*, pp. 1–6, 2004.

[34] E. Orhan, "The hopfield model," Technical report, NYU, Tech. Rep., 2014.

[35] J. Milnor, "On the concept of attractor," in *The theory of chaotic attractors*. Springer, 1985, pp. 243–264.

[36] H. S. Seung, "Continuous attractors and oculomotor control," *Neural Networks*, vol. 11, no. 7-8, pp. 1253–1258, 1998.

[37] D. J. Amit, H. Gutfreund, and H. Sompolinsky, "Spin-glass models of neural networks," *Physical Review A*, vol. 32, no. 2, p. 1007, 1985.

[38] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[39] G. E. Hinton *et al.*, "Boltzmann machines." 2017.

[40] I. Stoianov, M. Zorzi, S. Becker, and C. Umilta, "Associative arithmetic with boltzmann machines: The role of number representations," in *International Conference on Artificial Neural Networks*. Springer, 2002, pp. 277–283.

[41] T. J. Sejnowski, "Higher-order boltzmann machines," in *AIP Conference Proceedings*, vol. 151, no. 1. American Institute of Physics, 1986, pp. 398–403.

[42] C. C. Aggarwal *et al.*, *Neural networks and deep learning*. Springer, 2018.

[43] J. J. DeStefano, "Logistic regression and the boltzmann machine," in *1990 IJCNN International Joint Conference on Neural Networks*. IEEE, 1990, pp. 199–204.

[44] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for boltzmann machines," *Cognitive science*, vol. 9, no. 1, pp. 147–169, 1985.

[45] G. E. Hinton, T. J. Sejnowski *et al.*, "Learning and relearning in boltzmann machines," *Parallel distributed processing: Explorations in the microstructure of cognition*, vol. 1, no. 282-317, p. 2, 1986.

[46] A. Meyder and C. Kiderlen, "Fundamental properties of hopfield networks and boltzmann machines for associative memories," *Machine Learning, vt*, 2008.

[47] G. E. Hinton, "Deep belief nets." 2010.

[48] L. Younes, "Stochastic gradient estimation strategies for markov random fields," in *Bayesian inference for inverse problems*, vol. 3459. International Society for Optics and Photonics, 1998, pp. 315–325.

[49] V. Mnih, H. Larochelle, and G. E. Hinton, "Conditional restricted boltzmann machines for structured output prediction," *arXiv preprint arXiv:1202.3748*, 2012.

[50] G. E. Hinton, "A practical guide to training restricted boltzmann machines," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 599–619.

[51] N. Srivastava, R. Salakhutdinov *et al.*, "Multimodal learning with deep boltzmann machines." in *NIPS*, vol. 1. Citeseer, 2012, p. 2.

[52] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural computation*, vol. 14, no. 8, pp. 1771–1800, 2002.

[53] M. A. Carreira-Perpinan and G. Hinton, "On contrastive divergence learning," in *International workshop on artificial intelligence and statistics*. PMLR, 2005, pp. 33–40.

[54] G. E. Hinton, "Learning multiple layers of representation," *Trends in cognitive sciences*, vol. 11, no. 10, pp. 428–434, 2007.

[55] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[56] A.-r. Mohamed, G. Dahl, G. Hinton *et al.*, "Deep belief networks for phone recognition," in *Nips workshop on deep learning for speech recognition and related applications*, vol. 1, no. 9. Vancouver, Canada, 2009, p. 39.

[57] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[58] R. Sarikaya, G. E. Hinton, and A. Deoras, "Application of deep belief networks for natural language understanding," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 4, pp. 778–784, 2014.

[59] R. Salakhutdinov and G. Hinton, “Deep boltzmann machines,” in *Artificial intelligence and statistics*, 2009, pp. 448–455.